**Writing DirectShow Filters**

If you are developing a filter for use in a Microsoft DirectShow filter graph, read the articles in this section. In general, you do not have to read this section if you are writing a DirectShow application. Most applications do not access filters or the filter graph at the level discussed in this section.

I. Introduction to DirectShow Filter Development

This section gives a brief outline of the tasks involved in developing a custom DirectShow filter. It also provides links to topics that discuss these tasks in greater detail. Before reading this section, read the topics in About DirectShow, which describe the overall DirectShow architecture.

DirectShow Base Class Library

The DirectShow SDK includes a set of C++ classes for writing filters. Although they are not required, these classes are the recommended way to write a new filter. To use the base classes, compile them into a static library and link the .lib file to your project, as described in Building DirectShow Filters.

The base class library defines a root class for filters, the `CBaseFilter` class. Several other classes derive from `CBaseFilter`, and are specialized for particular kinds of filters. For example, the `CTransformFilter` class is designed for transform filters. To create a new filter, implement a class that inherits from one of the filter classes. For example, your class declaration might be as follows:

```cpp
class CMyFilter : public CTransformFilter
{
private:
    /* Declare variables and methods that are specific to your filter. */
public:
    /* Override various methods in CTransformFilter */
};
```

II. Building DirectShow Filters

For more information about the DirectShow base classes, see the following topics:

- DirectShow Base Classes
- Building DirectShow Filters

Creating Pins

A filter must create one or more pins. The number of pins can be fixed at design time, or the filter can create new pins as needed. Pins generally derive from the `CBasePin` class, or from a class that inherits `CBasePin`, such as `CBaseInputPin`. The filter’s pins should be declared as member variables in the filter class. Some of the filter classes already define the pins, but if your filter inherits directly from `CBaseFilter`, you must declare the pins in your derived class.

Negotiating Pin Connections

When the Filter Graph Manager tries to connect two filters, the pins must agree on various things. If they cannot, the connection attempt fails. Generally, pins negotiate the following:

- Transport. The transport is the mechanism that the filters will use to move media samples from the output pin to the input pin. For example, they can use the `IMemInputPin` interface ("push model") or the `IAsyncReader` interface ("pull model").
- Media type. Almost all pins use media types to describe the format of the data they will deliver.
- Allocator. The allocator is the object that creates the buffers that hold the data. The pins must agree which pin will provide the allocator. They must also agree on the size of the buffers, the number of buffers to create, and other buffer properties.

The base classes implement a framework for these negotiations. You must complete the details by overriding various methods in the base class. The set of methods that you must override depends on the class and on the functionality of your filter. For more information, see How Filters Connect.

Processing and Delivering Data

The primary function of most filters is to process and deliver media data. How that occurs depends on the type of filter:

- A push source has a worker thread that continuously fills samples with data and delivers them downstream.
- A pull source waits for its downstream neighbor to request a sample. It responds by writing data into a sample and delivering the sample to the downstream filter. The downstream filter creates the thread that drives the data flow.
- A transform filter has samples delivered to it by its upstream neighbor. When it receives a sample, it processes the data and delivers it downstream.
- A renderer filter receives samples from upstream, and schedules them for rendering based on the time stamps.
Other tasks related to streaming include flushing data from the graph, handling the end of the stream, and responding to seek requests. For more information about these issues, see the following topics:

- Data Flow for Filter Developers
- Quality-Control Management
- Threads and Critical Sections

Supporting COM

DirectShow filters are COM objects, typically packaged in DLLs. The base class library implements a framework for supporting COM. It is described in the section DirectShow and COM.

II. Building DirectShow Filters

The DirectShow base classes are recommended for implementing DirectShow filters. To build with the base classes, perform the following steps, in addition to those listed in Setting Up the Build Environment:

- Build the base class library, located in the directory Samples\Multimedia\DirectShow\BaseClasses, under the SDK root directory. There are two versions of the library: a retail version (Strmbase.lib) and a debug version (Strmbasd.lib).
- Include the header file Streams.h.
- Use the __stdcall calling convention.
- Use the multithreaded C run-time library (debug or retail, as appropriate).
- Include a definition (.def) file that exports the DLL functions. The following is an example of a definition file. It assumes that the output file is named MyFilter.dll.

```
LIBRARY MYFILTER.DLL
EXPORTS
    DllMain             PRIVATE
    DllGetClassObject   PRIVATE
    DllCanUnloadNow     PRIVATE
    DllRegisterServer   PRIVATE
    DllUnregisterServer PRIVATE
```

- Link to the following lib files.
  - Debug Build: Strmbasd.lib, Msvcrtd.lib, Winmm.lib
  - Retail Build: Strmbase.lib, Msvcrt.lib, Winmm.lib

- Choose the "ignore default libraries" option in the linker settings.

- Declare the DLL entry point in your source code, as follows:

```c
extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, LPVOID lpReserved) {
  return DllEntryPoint((HINSTANCE)(hModule), dwReason, lpReserved);
}
```

III. How Filters Connect

This article describes how filters connect in Microsoft® DirectShow®. The article is intended for filter developers. If you are writing a DirectShow application, you can probably ignore the details presented here.

This article contains the following topics:

1. Pin Connections
2. Negotiating Media Types
3. Negotiating Allocators
4. Providing a Custom Allocator
5. Reconnecting Pins

See Also:

- CBasePin Connection Process

III.1 Pin Connections

Filters connect at their pins, through the IPin interface. Output pins connect to input pins. Each pin connection has a media type, described by the AM_MEDIA_TYPE structure.

An application connects filters by calling methods on the Filter Graph Manager, never by calling methods on the filters or pins themselves. The application can directly specify which filters to connect, by calling the IFilterGraph::ConnectDirect or IGraphBuilder::Connect method; or it can connect filters indirectly, using a graph-building method such as IGraphBuilder::RenderFile.

For the connection to succeed, both filters must be in the filter graph. The application can add a filter to the graph by calling the IFilterGraph::AddFilter method. The Filter Graph Manager may add filters to the graph, as well. When a filter is added, the Filter Graph Manager calls the filter's IBaseFilter::JoinFilterGraph method to notify the filter.

The general outline of the connection process is the following:

1. The Filter Graph Manager calls IPin::Connect on the output pin, passing a pointer to the input pin.
2. If the output pin accepts the connection, it calls IPin::ReceiveConnection on the input pin.
3. If the input pin also accepts the connection, the connection attempt succeeds and the pins are connected.
Extrait de la documentation DirectShow

Some pins can be disconnected and reconnected while the filter is active. This type of reconnection is called a dynamic reconnection. For more information, see Dynamic Graph Building. However, most filters do not support dynamic reconnection. Filters are usually connected in downstream order—in other words, the filter’s input pins are connected before its output pins. A filter should always support this order of connection. Some filters also support connections in the opposite order—output pins first, followed by input pins. For example, it might be possible to connect a MUX filter’s output pin to the file-writer filter, before connecting the MUX filter’s input pins.

When a pin’s Connect or ReceiveConnection method is called, the pin must verify that it can support the connection. The details depend on the particular filter. The most common tasks include the following:

- Check that the media type is acceptable
- Negotiate an allocator
- Query the other pin for required interfaces.

### III.2 Negotiating Media Types

When the Filter Graph Manager calls the IPin::Connect method, it has several options for specifying a media type:

- **Complete type:** If the media type is fully specified, the pins attempt to connect with that type. If they cannot, the connection attempt fails.
- **Partial media type:** A media type is partial if the major type, subtype, or format type is GUID_NULL. The value GUID_NULL acts as a wildcard, indicating that any value is acceptable. The pins negotiate a type that is consistent with the partial type.
- **No media type:** If the Filter Graph Manager passes a NULL pointer, the pins can agree to any media type that is acceptable to both pins.

If the pins do connect, the connection always has a complete media type. The purpose of the media type given by the Filter Graph Manager is to limit the possible connection types.

During the negotiation process, the output pin proposes a media type by calling the IPin::ReceiveConnection method. The input pin can accept or reject the proposed type. This process repeats until either the input pin accepts a type, or the output pin runs out of types and the connection fails.

How an output pin selects media types to propose depends on the implementation. In the DirectShow base classes, the output pin calls IPin::EnumMediaTypeTypes on the input pin. This method returns an enumerator that enumerates the input pin’s preferred media types. Failing that, the output pin enumerates its own preferred types.

### Working with Media Types

In any function that receives an AM_MEDIA_TYPE parameter, always validate the values of cbFormat and formattype before dereferencing the pbFormat member. The following code is incorrect:

```c
if (pmt->formattype == FORMAT_VideoInfo) &&
    (pmt->cbFormat > sizeof(VIDEOINFOHEADER) &&
     (pbFormat != NULL))

// Now you can dereference pVIH.
```

The following code is correct:

```c
if ((pmt->formattype == FORMAT_VideoInfo) &&
    (pmt->cbFormat > sizeof(VIDEOINFOHEADER) &&
     (pbFormat != NULL))

// Now you can dereference pVIH.
```

### III.3 Negotiating Allocators

When two pins connect, they need a mechanism for exchanging media data. This mechanism is called the transport. In general, the DirectShow architecture is neutral about transports. Two filters can agree to connect using any transport that both support. The most common transport is the local memory transport, in which the media data resides in main memory. Two flavors of local memory transport exist, the push model and the pull model. In the push model, the source filter pushes data to the downstream filter, using the IMemInputPin interface on the downstream filter’s input pin. In the pull model, the downstream filter requests data from the source filter, using the IAsyncReader interface on the source filter’s output pin. For more information on these two data-flow models, see Data Flow in the Filter Graph.

In local memory transport, the object responsible for allocating memory buffers is called the allocator. An allocator supports the IMemAllocator interface. Both pins share a single allocator. Either pin can provide an allocator, but the output pin selects which allocator to use.

The output pin also sets the allocator properties, which determine how many buffers are created by the allocator, the size of each buffer, and the memory alignment. The output pin might defer to the input pin for the buffer requirements.

In an IMemInputPin connection, allocator negotiation works as follows:

1. Optionally, the output pin calls IMemInputPin::GetAllocatorRequirements. This method retrieves the input pin’s buffer requirements, such as memory alignment. In general, the output pin should honor the input pin’s request, unless there is a good reason not to.
2. Optionally, the output pin calls IMemInputPin::GetAllocator. This method requests an allocator from the input pin. The input pin provides one, or returns an error code.
3. The output pin selects an allocator. It can use one provided by the input pin, or create its own.
4. The output pin calls IMemAllocator::SetProperties to set the allocator properties. However, the allocator might not honor the requested properties. (For example, this can happen if the input pin provides the allocator.)
allocator returns the actual properties as an output parameter in the
SetProperties method.
5. The output pin calls IMemInputPin::NotifyAllocator to inform the input pin of
the selection.
6. The input pin should call IMemAllocator::GetProperties to verify whether the
allocator properties are acceptable.
7. The output pin is responsible for committing and decommitting the allocator.
   This occurs when streaming starts and stops.
In an IAsyncReader connection, allocator negotiation works as follows:
1. The input pin calls IAsyncReader::RequestAllocator on the output pin. The
   input pin specifies its buffer requirements and, optionally, provides an
   allocator.
2. The output pin selects an allocator. It can use the one provided by the input
   pin, if any, or create its own.
3. The output pin returns the allocator as an outgoing parameter in the
   RequestAllocator method. The input pin should check the allocator properties.
4. The input pin is responsible for committing and decommitting the allocator.
5. At any time during the allocator negotiation process, either pin can fail the
   connection.
6. If the output pin uses the input pin's allocator, it can use that allocator only to
   deliver samples to that input pin. The owning filter must not use the allocator
   to deliver samples to other pins.

See Also
- Providing a Custom Allocator

III.4 Providing a Custom Allocator
This section describes how to provide a custom allocator for a filter. Only
IMemInputPin connections are described, but the steps for an IAsyncReader
connection are similar.
First, define a C++ class for the allocator. Your allocator can derive from one of the
standard allocator classes, CBaseAllocator or CMemAllocator, or you can create an
entirely new allocator class. If you create a new class, it must expose the
IMemAllocator interface.
The remaining steps depend on whether your allocator belongs to an input pin or an
output pin on your filter. Input pins play a different role than output pins during the
allocator negotiation phase, because the output pin ultimately selects the allocator.

Providing a Custom Allocator for an Input Pin
To provide an allocator for an input pin, override the input pin's
CBaseInputPin::GetAllocator method. Within this method, check the m_pAllocator
member variable. If this variable is non-NULL, it means the allocator has already been
selected for this connection, so the GetAllocator method must return a pointer to that
allocator. If m_pAllocator is NULL, it means the allocator has not been selected, so
the GetAllocator method should return a pointer to the input pin's preferred allocator.
In that case, create an instance of your custom allocator and return its IMemAllocator
pointer. The following code shows how to implement the GetAllocator method:

```cpp
STDMETHODIMP CMyInputPin::GetAllocator(IMemAllocator **ppAllocator)
{
    CheckPointer(ppAllocator, E_POINTER);
    if (m_pAllocator)
    {
        // We already have an allocator, so return that one.
        *ppAllocator = m_pAllocator;
        (*ppAllocator)->AddRef();
        return S_OK;
    }
    // No allocator yet, so propose our custom allocator. The exact code
    // here will depend on your custom allocator class definition.
    HRESULT hr = S_OK;
    CMyAllocator *pAlloc = new CMyAllocator(hr);
    if (!pAlloc)
    {
        return E_OUTOFMEMORY;
    }
    if (FAILED(hr))
    {
        delete pAlloc;
        return hr;
    }
    // Return the IMemAllocator interface to the caller.
    return pAlloc->QueryInterface(IID_IMemAllocator, (void**)ppAllocator);
}
```

When the upstream filter selects an allocator, it calls the input pin's
IMemInputPin::NotifyAllocator method. Override the
CBaseInputPin::NotifyAllocator method to check the allocator properties. In some
cases, the input pin might reject the allocator if it is not your custom allocator,
although this may cause the entire pin connection to fail.

Providing a Custom Allocator for an Output Pin
To provide an allocator for an output pin, override the
CBaseOutputPin::InitAllocator method to create an instance of your allocator:

```cpp
HRESULT MyOutputPin::InitAllocator(IMemAllocator **ppAlloc)
{
    HRESULT hr = S_OK;
    CMyAllocator *pAlloc = new CMyAllocator(hr);
    if (!pAlloc)
    {
        return E_OUTOFMEMORY;
    }
    if (FAILED(hr))
    {
        delete pAlloc;
        return hr;
    }
}
```
By default, the CBaseOutputPin class requests an allocator from the input pin first. If that allocator is not suitable, the output pin creates its own allocator. To force the connection to use your custom allocator, override the CBaseOutputPin::DecideAllocator method. However, be aware that this can prevent your output pin from connecting with certain filters, because the other filter may also require its own custom allocator.

See Also

- Negotiating Allocators

### III.5  Reconnecting Pins

During a pin connection, a filter can disconnect and reconnect one of its other pins, as follows:

4. The filter calls IPin::QueryAccept on the other filter's pin, and specifies the new media type.

5. If QueryAccept returns S_OK, the filter calls IFilterGraph2::ReconnectEx to reconnect the pins.

The following are some examples of when a filter might need to reconnect pins:

- **Tee filters.** A tee filter splits an input stream into multiple outputs without changing the data in the stream. A tee filter can accept a range of media types, but the types must match across all pin connections. Therefore, when the input pin disconnects, the filter might need to renegotiate any existing connections on the output pins, and vice versa. For an example, see the InfoTee Filter Sample.

- **Trans-in-place filters.** A trans-in-place filter modifies the input data in the original buffer instead of copying the data to a separate output buffer. A trans-in-place filter must use the same allocator for both the upstream and the downstream connections. The first pin to connect (input or output) negotiates an allocator in the usual way. When the other pin connects, however, the first allocator might not be acceptable. In that case, the second pin chooses a different allocator, and the first pin reconnects using the new allocator. For an example implementation, see the CTransInPlaceFilter class.

In the ReconnectEx method, the Filter Graph Manager asynchronously disconnects and reconnects the pins. The filter must attempt the reconnection unless QueryAccept returns S_OK. Otherwise, the pin will be left disconnected, causing graph errors. Also, the filter should request the reconnection from inside the IPin::Connect method, on the same thread. If the Connect method returns on one thread, while another thread makes the reconnection request, the Filter Graph Manager might run the graph before it makes the reconnection, causing graph errors.

---

**IV. Data Flow for Filter Developers**

This section describes in detail how data moves through the filter graph. It focuses on local memory transport using the IMemInputPin or IAsyncReader interface. It is intended for developers who are writing their own custom filters. For a general introduction to how Microsoft DirectShow handles data flow, see Data Flow in the Filter Graph.

A lot of data moves through a filter graph. It falls roughly into two categories: media data and control data. In general, media data travels downstream and control data travels upstream. Media data includes the video frames, audio samples, MPEG packets, and so forth that make up a stream, but also includes flush commands, end-of-stream notifications, and other data that travels with the stream. Control data is not part of the media stream. Examples of control data are quality-control requests and seek commands.

This section contains the following articles.

1. Delivering Samples
2. Processing Data
3. End-of-Stream Notifications
4. New Segments
5. Flushing
6. Seeking
7. Dynamic Format Changes

See Also

- Threads and Critical Sections
- Quality-Control Management

### IV.1  Delivering Samples

This article describes how a filter delivers a sample. It describes both the push model, using IMemInputPin methods, and the pull model, using IAsyncReader.

**Push Model: Delivering a Sample**

The output pin delivers a sample by calling the IMemInputPin::Receive method or the IMemInputPin::ReceiveMultiple method, which is equivalent but delivers an array of samples. The input pin can block inside Receive (or ReceiveMultiple). If the pin might block, its IMemInputPin::ReceiveCanBlock method should return S_OK. If the pin guarantees never to block, ReceiveCanBlock should return S_FALSE. The S_OK return value does not mean that Receive always blocks, just that it might. Although Receive can block to wait for a resource to become available, it should not block to wait for more data from the upstream filter. Doing so can cause a deadlock where the upstream filter waits for the downstream filter to release the sample, which never happens because the downstream filter is waiting on the upstream filter. If a filter has multiple input pins, however, one pin can wait for another pin to receive data. For example, the AVI Mux filter does this so that it can interleave audio and video data.
Extrait de la documentation DirectShow

A pin might reject a sample for a number of reasons:
- The pin is flushing (see Flushing).
- The pin is not connected.
- The filter is stopped.
- Some other error occurred.

The Receive method should return S_FALSE in the first case, and a failure code in the other cases. The upstream filter should stop sending samples when the return code is anything other than S_OK.

If the first three cases to be "expected" failures, in the sense that the filter was in the wrong state to receive samples. An unexpected failure will be one that causes the pin to reject a sample even though the pin is in a receiving state. If an error of this type occurs, the pin should send an end-of-stream notification downstream, and send an EC_ERRORABORT event to the Filter Graph Manager.

In the DirectShow base classes, the CBaseInputPin::CheckStreaming method checks for the general failure cases—flushing, stopped, and so forth. The derived class will need to check for failures that are specific to the filter. In case of an error, the CBaseInputPin::Receive method sends the end-of-stream notification and the EC_ERRORABORT event.

Pull Model: Requesting a Sample

In the IAsyncReader interface, the input pin requests samples from the output pin by calling one of the following methods:
- IAsyncReader::Request
- IAsyncReader::SyncRead
- IAsyncReader::SyncReadAligned

The Request method is asynchronous, the input pin calls IAsyncReader::WaitForNext to wait for the request to complete. The other two methods are synchronous.

When to Deliver Data

A filter always delivers samples while it is in the running state. In most cases, a filter also delivers samples while paused. This enables the graph to cue up the data so that playback starts immediately when Run is called (see Filter States). If your filter does not deliver data while paused, the filter's IMediaFilter::GetState method should return VFW_S_CANT_CUE in the paused state. This return code signals the filter graph not to wait for data from your filter before it completes the pause transition. Otherwise, the Pause method will block indefinitely. For example code, see CBaseFilter::GetState.

Here are some examples of when a filter might need to return VFW_S_CANT_CUE:
- Live sources, such as capture filters, should not send data while paused. See Producing Data in a Capture Filter.
- A splitter filter might or might not send data while paused, depending on the implementation. If the filter uses separate threads to queue data on each output pin, then it can send data while paused. But if the filter uses a single thread for every output pin, the first pin might block the thread when it calls Receive.

Errors During Streaming

which will prevent the other pins from sending data. In that case, you should return VFW_S_CANT_CUE.
- A filter might deliver data sporadically. For example, it might parse a custom data stream and filter out some packets while delivering others. In that case, the filter may not be guaranteed to deliver data while paused.
- A source filter (using the push model) or a parser filter (using the push/pull model) creates one or more streaming threads, which deliver samples as quickly as possible. Downstream filters, such as decoders and transforms, typically send data only when Receive is called on their input pins.

See Also
- Receiving and Delivering Samples

IV.2 Processing Data

Parsing Media Data

If your filter parses media data, do not trust headers or other self-describing data in the content. For example, do not trust size values that appear in AVI RIFF chunks or MPEG packets. Common examples of this kind of error include:
- Reading N bytes of data, where the value of N came from the content, without checking that the actual size of your buffer.
- Jumping to a byte offset within a buffer, without verifying that the offset falls within the buffer.

Another common class of errors involves not validating format descriptions that are found in the content. For example:
- A BITMAPINFOHEADER structure can be followed by a color table. The BITMAPINFO structure is defined as a BITMAPINFOHEADER structure followed by an array of RGBQUAD values that make up the color table. The size of the array is determined by the value of biClrUsed. Never copy a color table into a BITMAPINFO without first checking the size of the buffer that was allocated for the BITMAPINFO structure.
- A WAVEFORMATEX structure might have extra format information appended to the structure. The cbSize member specifies the size of the extra information.

During pin connection, a filter should verify that all format structures are well-formed and contain reasonable values. If not, reject the connection. In the code that validates the format structure, be especially careful about arithmetic over/flow. For example, in a BITMAPINFOHEADER, the width and height are 32-bit long values but the image size (which is a function of the product of the two) is only a DWORD value. If format data from the source is larger than your allocated buffer, do not truncate the source data in order to copy it into your buffer. Doing so can create a structure whose implicit size is larger than its actual size. For example, a bitmap header might specify a palette table which no longer exists. Instead, reallocate the buffer or simply fail the connection.
Extrait de la documentation DirectShow

When the graph is running, if your filter receives malformed content, it should terminate streaming. Do the following:

• Return an error code from **Receive**.
• Call **IPin::EndOfStream** on the downstream filter.
• Call **CBaseFilter::NotifyEvent** to post an **EC_ERRORABORT** event.

### Format Changes

Several mechanisms exist for filters to change formats mid-stream. Each of them involves more than one step, which creates the potential for false acceptances. If your filter gets a request for a dynamic format change, it must either reject the request, or else honor the new format when it arrives. Similarly, never switch formats unless the other filter agrees. For more information, see Dynamic Format Changes.

### IV.3 End-of-Stream Notifications

When a source filter is done sending data, it calls the **IPin::EndOfStream** method on the downstream input pin. The downstream filter propagates the call to the next filter, and so on. When the **EndOfStream** call reaches the renderer, the renderer sends an **EC_COMPLETE** event to the Filter Graph Manager. If the renderer has multiple input pins, it delivers the **EC_COMPLETE** event after every input pin has received the end-of-stream notification.

A filter must serialize **EndOfStream** calls with other streaming calls, such as **IMemInputPin::Receive**. (In other words, the downstream filter must always receive the calls in the correct order.)

In some cases, a downstream filter might detect the end of the stream before the source filter does. (For example, the downstream filter might be parsing the stream.) In that case, the downstream filter can send the end-of-stream notification, in which case it should return **S_FALSE** from **IMemInputPin::Receive** until the graph stops or flushes. The **S_FALSE** return value informs the source filter to stop sending data.

**Default Handling of EC_COMPLETE**

By default, the Filter Graph Manager does not forward every **EC_COMPLETE** event to the application. Instead, it waits until all streams have signaled **EC_COMPLETE** and then sends a single **EC_COMPLETE** event. Thus, the application receives the event after every stream has completed.

To determine the number of streams, the Filter Graph Manager counts the number of filters that support seeking (through **IMediaSeeking** or **IMediaPosition**) and have a rendered input pin, which is defined as an input pin with no corresponding outputs. The Filter Graph Manager determines whether a pin is rendered in one of two ways:

- The pin's **IPin::QueryInternalConnections** method returns zero in the ndPin parameter.
- The filter exposes the **IAMFilterMiscFlags** interface and returns the **AM_FILTER_MISC_FLAGS_IS_RENDERER** flag.

**End-of-Stream Notifications in Pull Mode**

Extrait de la documentation DirectShow

In an **IAasyncReader** connection, the source filter does not send an end-of-stream notification. Instead, this is done by the downstream filter, which is typically a parser filter. The parser sends the **EndOfStream** call downstream. It does not send one upstream to the source filter.

**See Also**

- Delivering the End of Stream

### IV.4 New Segments

A segment is a group of media samples that share a common start time, stop time, and playback rate. The **IPin::NewSegment** method signals the start of a new segment. It provides a way for a source filter to inform downstream filters that the time and rate information has changed. For example, if the source filter seeks to a new point in the stream, it calls **NewSegment** with the new start time.

Some downstream filters use the segment information when they process samples. For example, in a format that uses interframe compression, if the stop time falls on a delta frame, the source filter may need to send additional samples after the stop time. This enables the decoder to decode the final delta frame. To determine the correct final frame, the decoder refers to the segment stop time. As another example, audio renderers use the segment rate along with the audio sampling rate to generate the correct audio output.

In the push model, the source filter initiates the **NewSegment** call. In the pull model, this is done by the parser filter. In either case, the filter calls **NewSegment** on the downstream input pin, which propagates the call to the next filter, until the call reaches the renderer. Filters must serialize **NewSegment** calls with other streaming calls, such as **IMemInputPin::Receive**. Stream time resets to zero after each new segment. Time stamps on samples delivered after the segment start from zero.

### IV.5 Flushing

While the filter graph is running, arbitrary amounts of data can be moving through the graph. Some of it might be in queues, waiting to be delivered. There are times when the filter graph needs to remove this pending data as quickly as possible and replace it with new data. After a seek command, for example, the source filter generates samples from a new position in the source. To minimize latency, downstream filters should discard any samples that were created before the seek command. The process of discarding samples is called flushing. It enables the graph to be more responsive when events alter the normal data flow.

Flushing is handled slightly differently by the pull model than the push model. This article starts by describing the push model; then it describes the differences in the pull model.

Flushing happens in two stages:

- First, the source filter calls **IPin::BeginFlush** on the downstream filter's input pin. The downstream filter starts rejecting samples from upstream. It also
Filters support seeking through the IMediaSeeking interface. The application queries the Filter Graph Manager for IMediaSeeking and uses it to issue seek commands. The Filter Graph Manager distributes each seek command to all of the renderer filters in the graph. Each renderer passes the command upstream, through the output pins of the upstream filters, until it reaches a filter that can execute the seek. Typically a source filter or parser filter, such as the AVI Splitter, carries out the seek operation.

When a filter performs a seek operation, it flushes any pending data. The result is to minimize the latency of seek commands, because existing data is flushed from the graph. After a seek command, stream time resets to zero.

The following diagram illustrates the sequence of events:

![Diagram](image)

If a parser filter has more than one output pin, it typically designates one of them to accept seek commands. The other pins reject or ignore any seek commands they receive. In this way, the parser keeps all of its streams synchronized. However, all output pins should implement IMediaSeeking::GetCapabilities and IMediaSeeking::CheckCapabilities to return the filter's seeking capabilities. This ensures that the Filter Graph Manager returns the correct value to the application.

See Also
- Flushing
- Time and Clocks in DirectShow

IV.7 Dynamic Format Changes

When two filters connect, they agree on a media type, which describes the format of the data that the upstream filter will deliver. In most cases, the media type is fixed for the duration of the connection. However, DirectShow does offer limited support for filters to change the media type. When a filter switches media types, it is called a dynamic format change. If you are writing a DirectShow filter, you should be aware of the mechanisms for dynamic format changes. Even if your filter does not support such changes, it should respond correctly if another filter requests a new format.

DirectShow defines several distinct mechanisms for dynamic format changes, depending on the state of filter graph and the type of change that is required.

- If the graph is stopped, the pins can reconnect and renegotiate the media type.
  - For more information, see Reconnecting Pins.
- Some filters can reconnect pins even while the graph is active (running or paused). For more information about this mechanism, see Dynamic Reconnection.

Otherwise, if the graph is active, but the filters in question do not support dynamic pin reconnections, there are three possible mechanisms for changing the format:
Extrait de la documentation DirectShow

V. Threads and Critical Sections

This article describes threading in DirectShow filters, and the steps you should take to avoid crashes or deadlocks in a custom filter. This article contains the following topics.
1. The Streaming and Application Threads
2. Pausing
3. Receiving and Delivering Samples
4. Delivering the End of Stream
5. Flushing Data
6. Stopping
7. Getting Buffers
8. Streaming Threads and the Filter Graph Manager
9. Summary of Filter Threading

See Also
- Data Flow for Filter Developers
- Handling Format Changes from the Video Renderer

The examples in this article use pseudocode to illustrate the code you will need to write. They assume that a custom filter is using classes derived from the DirectShow base classes, as follows:
- CMyInputPin: Derived from CBaseInputPin
- CMyOutputPin: Derived from CBaseOutputPin
- CMyFilter: Derived from CBaseFilter
- CMyInputAllocator: The input pin's allocator, derived from CMemAllocator. Not every filter needs a custom allocator. For many filters, the CMemAllocator class is sufficient.

V.1 The Streaming and Application Threads

Any DirectShow application contains at least two important threads: the application thread, and one or more streaming threads. Samples are delivered on the streaming threads, and state changes happen on the application thread. The main streaming thread is created by a source or parser filter. Other filters might create worker threads that deliver samples, and these are considered streaming threads as well.

Some methods are called on the application thread, while others are called on a streaming thread. For example:
- Streaming thread(s): IMemInputPin::Receive,
  IMemInputPin::ReceiveMultiple, IPin::EndOfStream,
  IMemAllocator::GetBuffer,
- Application thread: IMediaFilter::Pause, IMediaFilter::Run,
  IMediaFilter::Stop, IMediaSeeking::SetPositions, IPin::BeginFlush,
  IPin::EndFlush
- Either: IPin::NewSegment.

Having a separate streaming thread allows data to flow through the graph while the application thread waits for user input. The danger of multiple threads, however, is that a filter may create resources when it pauses (on the application thread), use them inside a streaming method, and destroy them when it stops (also on the application thread). If you are not careful, the streaming thread might try to use the resources after they are destroyed. The solution is to protect resources using critical sections, and synchronize streaming methods with state changes.

A filter needs one critical section to protect the filter state. The CBaseFilter class has a member variable for this critical section, CBaseFilter::m_pLock. This critical section is called the filter lock. Also, each input pin needs a critical section to protect resources used by the streaming thread. These critical sections are called streaming locks; you must declare them in your derived pin class. It is easiest to use the CCritSec class, which wraps a Windows CRITICAL_SECTION object and can be locked using the CAutoLock class. The CCritSec class also provides some useful debugging functions. For more information, see Critical Section Debugging Functions.

When a filter stops or flushes, it must synchronize the application thread with the streaming thread. To avoid deadlocking, it must first unblock the streaming thread, which might be blocked for several reasons:
- It is waiting to get a sample inside the IMemInputPin::GetBuffer method, because all of the allocator's samples are in use.
- It is waiting for another filter to return from a streaming method, such as Receive.
- It is waiting inside one of its own streaming methods, for some resource to become available.
- It is a renderer filter waiting for the presentation time of the next sample.
- It is a renderer filter waiting inside the Receive method while paused.

Therefore, when the filter stops or flushes, it must do the following:
- Release any sample it is holding for any reason. Doing so unblocks the GetBuffer method.
- Return from any streaming method as quickly as possible. If a streaming method is waiting for a resource, it must stop waiting immediately.
- Start rejecting samples in Receive, so that the streaming thread does not access any more resources. (The CBaseInputPin class handles this automatically.)
The Stop method must decommit all of the filter's allocators. (The CBaseInputPin class handles this automatically.) Flushing and stopping both happen on the application thread. A filter stops in response to the IMediaControl::Stop method. The Filter Graph Manager issues the stop command in upstream order, starting from the renderers and working backward to the source filters. The stop command happens completely inside the filter's CBaseFilter::Stop method. When the method returns, the filter should be in a stopped state.

Flushing typically occurs because of a seek command. A flush command starts from the source or parser filter, and travels downstream. Flushing happens in two stages: The IPin::BeginFlush method informs a filter to discard all pending and incoming data; the IPin::EndFlush method signals the filter to accept data again. Flushing requires two stages because the BeginFlush call is on the application thread, during which the streaming thread continues to deliver data. Therefore, some samples may arrive after the BeginFlush call. The filter should discard these. Any samples that arrive after the EndFlush call are guaranteed to be new, and should be delivered. The sections that follow contain code samples showing how to implement the most important filter methods, such as Pause, Receive, and so forth, in ways that avoid deadlocks and race conditions. Every filter has different requirements, however, so you will need to adapt these examples to your particular filter.

Note The CTransformFilter and CTransInPlaceFilter base classes handle many of the issues described in this article. If you are writing a transform filter, and your filter does not wait on events inside a streaming method, or hold onto samples outside of Receive, then these base classes should be sufficient.

V.2 Pausing

All filter state changes must hold the filter lock. In the Pause method, create any resources the filter needs:

```cpp
HRESULT CMyFilter::Pause()
{
    CAutoLock lock_it(m_pLock);
    /* Create filter resources. */
    return CBaseFilter::Pause();
}
```

The CBaseFilter::Pause method sets the correct state on the filter (State_Paused) and calls the CBasePin::Active method on every connected pin in the filter. The Active method informs the pin that the filter has become active. If the pin creates resources, override the Active method, as follows:

```cpp
HRESULT CMyInputPin::Active()
{
    // You do not need to hold the filter lock here. It is already held in Pause.
    return CBaseInputPin::Active();
}
```

V.3 Receiving and Delivering Samples

The following pseudocode shows how to implement the IMemInput::Receive method:

```cpp
HRESULT CMyInputPin::Receive(IMediaSample *pSample)
{
    CAutoLock cObjectLock(&m_csReceive);
    // Perhaps the filter needs to wait on something.
    WaitForSingleObject(m_hSomeEventThatReceiveNeedsToWaitOn, INFINITE);
    // Before using resources, make sure it is safe to proceed. Do not continue if the base-class method returns anything besides S_OK.
    hr = CBaseInputPin::Receive(pSample);
    if (hr != S_OK)
    {
        return hr;
    }
    /* It is safe to use resources allocated in Active and Pause. */
    // Deliver sample(s), via your output pin(s).
    for (each output pin)
    pOutputPin->Deliver(pSample);
    return hr;
}
```

The Receive method holds the streaming lock, not the filter lock. The filter might need to wait on some event before it can process the data, shown here by the call to WaitForSingleObject. Not every filter will need to do this. The CBaseInputPin::Receive method verifies some general streaming conditions. It returns VFW_E_WRONG_STATE if the filter is stopped, S_FALSE if the filter is flushing, and so forth. Any return code other than S_OK indicates that the Receive method should return immediately and not process the sample. After the sample is processed, deliver it to the downstream filter by calling CBaseOutputPin::Deliver. This helper method calls IMemInputPin::Receive on the downstream input pin. A filter might deliver samples to several pins.

V.4 Delivering the End of Stream

When the input pin receives an end-of-stream notification, it propagates the call downstream. Any downstream filters that receive data from this input pin should also get the end-of-stream notification. Again, take the streaming lock and not the filter lock. If the filter has pending data that was not yet delivered, the filter should deliver it...
now, before it sends the end-of-stream notification. It should not send any data after
the end of the stream.

HRESULT CMyInputPin::EndOfStream()
{
    CAutoLock lock_it(&m_csReceive);
    /* If the pin has not delivered all of the data in the stream
    (based on what it received previously), do so now. */
    // Propagate EndOfStream call downstream, via your output pin(s).
    for (each output pin)
    {
        hr = pOutputPin->DeliverEndOfStream();
    }
    return S_OK;
}

The CBaseOutputPin::DeliverEndOfStream method calls IPin::EndOfStream on
the downstream input pin.

V.5 Flushing Data

The following pseudocode shows how to implement the IPin::BeginFlush method:

HRESULT CMyInputPin::BeginFlush()
{
    CAutoLock lock_it(m_pLock);
    // First, make sure the Receive method will fail from now on.
    HRESULT hr = CBaseInputPin::BeginFlush();
    // Force downstream filters to release samples. If our Receive method
    // is blocked in GetBuffer or Deliver, this will unblock it.
    for (each output pin)
    {
        hr = pOutputPin->DeliverBeginFlush();
    }
    // Unblock our Receive method if it is waiting on an event.
    SetEvent(m_hSomeEventThatReceiveNeedsToWaitOn);
    // At this point, the Receive method can't be blocked. Make sure
    // it finishes, by taking the streaming lock. (Not necessary if this
    // is the last step.)
    { 
        CAutoLock lock_2(&m_csReceive);
        /* Now it's safe to do anything that would crash or hang
        if Receive were executing. */
    }
    return hr;
}

When flushing starts, the BeginFlush method takes the filter lock, which serializes the
state change. It is not yet safe to take the streaming lock, because flushing happens on
the application thread, and the streaming thread might be in the middle of a Receive
call. The pin needs to guarantee that Receive is not blocked, and that any subsequent
calls to Receive will fail. The CBaseInputPin::BeginFlush method sets an internal flag, CBaseInputPin::m_bFlushing. When the flag is TRUE, the Receive method fails.

By delivering the BeginFlush call downstream, the pin guarantees that all downstream
filters release their samples and return from Receive calls. This in turn guarantees that
the input pin is not blocked waiting for GetBuffer or Receive. If your pin's Receive
method ever waits on an event (for example, to get resources), the BeginFlush method
should force the wait to terminate by setting the event. At this point, the Receive
method is guaranteed to return, and the m_bFlushing flag prevents new Receive calls
during any work.

For some filters, that is all BeginFlush needs to do. The EndFlush method will signal
to the filter that it can start receiving samples again. Other filters may need to use
variables or resources in BeginFlush that are also used in Receive. In that case, the
filter should hold the streaming lock first. Be sure not to do this before any of
the previous steps, because you might cause a deadlock.

The EndFlush method holds the filter lock and propagates the call downstream:

HRESULT CMyInputPin::EndFlush()
{
    CAutoLock lock_it(m_pLock);
    for (each output pin)
    hr = pOutputPin->DeliverEndFlush();
    return CBaseInputPin::EndFlush();
}

The CBaseInputPin::EndFlush method resets the m_bFlushing flag to FALSE,
which allows the Receive method to start receiving samples again. This should be
the last step in EndFlush, because the pin must not receive any samples until flushing is
complete and all downstream filters are notified.

V.6 Stopping

The Stop method must unblock the Receive method and decommit the filter's
allocators. Decommitting an allocator forces any pending GetBuffer calls to return,
which unblocks upstream filters that are waiting for samples. The Stop method holds
the filter lock and then calls the CBaseFilter::Stop method, which calls
CBasePin::Inactive on all of the filter's pins:

HRESULT CMyFilter::Stop()
{
    CAutoLock lock_it(m_pLock);
    // Inactivate all the pins, to protect the filter resources.
    hr = CBaseFilter::Stop();
    /* Safe to destroy filter resources used by the streaming thread. */
}
Override the input pin's `Inactive` method as follows:

```cpp
HRESULT CMyInputPin::Inactive()
{
    // You do not need to hold the filter lock here. // It is already locked in Stop.
    // Unblock Receive. SetEvent(m_hSomeEventThatReceiveNeedsToWaitOn);
    // Make sure Receive will fail. // This also decommits the allocator.
    HRESULT hr = CBaseInputPin::Inactive();
    // Make sure Receive has completed, and is not using resources.
    {  
        CAutoLock c(&m_csReceive);
        /* It is now safe to destroy filter resources used by the streaming thread. */
    }
    return hr;
}
```

V.7 Getting Buffers

If your filter has a custom allocator that uses filter resources, the `GetBuffer` method should hold the streaming lock, as with other streaming methods:

```cpp
HRESULT CMyInputAllocator::GetBuffer(
    IMediaSample **ppBuffer,
    REFERENCE_TIME *pStartTime,
    REFERENCE_TIME *pEndTime,
    DWORD dwFlags)
{
    CAutoLock cObjectLock(&m_csReceive);
    /* Use resources. */
    return CMemAllocator::GetBuffer(ppBuffer, pStartTime, pEndTime, dwFlags);
}
```

V.8 Streaming Threads and the Filter Graph Manager

When the Filter Graph Manager stops the graph, it waits for all streaming threads to shut down. This has the following implications for filters:

- A filter must never call methods on the Filter Graph Manager from a streaming thread.

VI. Quality-Control Management

Quality control is a mechanism for adjusting the rate of data flow through the filter graph in response to run-time performance. If a renderer filter is receiving too much data or too little data, it can send a quality message. The quality message requests an adjustment in the data rate. By default, quality messages travel upstream from the renderer until they reach a filter that can respond (if any). An application can also
implement a custom quality manager. In that case, the renderer passes quality messages directly to the application's quality manager. This article contains the following topics.

- Quality Messages
- Default Quality Control

VII. DirectShow and COM

This section contains the following articles.

1. How to Implement IUnknown
2. How to Create a DLL
3. How to Register DirectShow Filters

VII.1 How to Implement IUnknown

Microsoft DirectShow is based on the Component Object Model (COM). If you write your own filter, you must implement it as a COM object. The DirectShow base classes provide a framework from which to do this. Using the base classes is not required, but it can simplify the development process. This article describes some of the internal details of COM objects and their implementation in the DirectShow base classes. This article assumes that you know how to program COM client applications—in other words, that you understand the methods in IUnknown—but does not assume any prior experience developing COM objects. DirectShow handles many of the details of developing a COM object. If you have experience developing COM objects, you should read the section Using CUnknown, which describes the CUnknown base class.

COM is a specification, not an implementation. It defines the rules that a component must follow; putting those rules into effect is left to the developer. In DirectShow, all objects derive from a set of C++ base classes. The base class constructors and methods do most of the COM "bookkeeping" work, such as keeping a consistent reference count. By deriving your filter from a base class, you inherit the functionality of the class. To use base classes effectively, you need a general understanding of how they implement the COM specification. This article contains the following topics.

a. How IUnknown Works
b. Using CUnknown

VII.1.a How IUnknown Works

The methods in IUnknown enable an application to query for interfaces on the component and manage the component's reference count.

Reference Count

The reference count is an internal variable, incremented in the AddRef method and decremented in the Release method. The base classes manage the reference count and synchronize access to the reference count among multiple threads.

Interface Queries

Querying for an interface is also straightforward. The caller passes two parameters: an interface identifier (IID), and the address of a pointer. If the component supports the requested interface, it sets the pointer to the interface, increments its own reference count, and returns S_OK. Otherwise, it sets the pointer to NULL and returns E_NOINTERFACE. The following pseudocode shows the general outline of the QueryInterface method. Component aggregation, described in the next section, introduces some additional complexity.

```c
if (IID == IID_IUnknown)
    set pointer to (IUnknown *)this
    AddRef
    return S_OK
else if (IID == IID_ISomeInterface)
    set pointer to (ISomeInterface *)this
    AddRef
    return S_OK
else if ...
else
    set pointer to NULL
    return E_NOINTERFACE
```

The only difference between the QueryInterface method of one component and the QueryInterface method of another is the list of IIDs that each component tests. For every interface that the component supports, the component must test for the IID of that interface.

Aggregation and Delegation

Component aggregation must be transparent to the caller. Therefore, the aggregate must expose a single IUnknown interface, with the aggregated component deferring to the outer component's implementation. Otherwise, the caller would see two different IUnknown interfaces in the same aggregate. If the component is not aggregated, it uses its own implementation.

To support this behavior, the component must add a level of indirection. A delegating IUnknown delegates the work to the appropriate place: to the outer component, if there is one, or to the component's internal version. A nondelegating IUnknown does the work, as described in the previous section.

The delegating version is public and keeps the name IUnknown. The nondelegating version is renamed INonDelegatingUnknown. This name is not part of the COM specification, because it is not a public interface.

When the client creates an instance of the component, it calls the IClassFactory::CreateInstance method. One parameter is a pointer to the aggregating component's IUnknown interface, or NULL if the new instance is not aggregated. The component uses this parameter to store a member variable indicating which IUnknown interface to use, as shown in the following example.
CMyComponent::CMyComponent(IUnknown *pOuterUnknown)
{
    if (pOuterUnknown == NULL)
        m_pUnknown = (IUnknown *)(INonDelegatingUnknown *)this;
    else
        m_pUnknown = pOuterUnknown;
    [ ... more constructor code ... ]
}

Each method in the delegating IUnknown calls its nondelegating counterpart, as shown in the following example:

HRESULT QueryInterface(REFIID iid, void **ppv)
{
    return m_pUnknown->QueryInterface(iid, ppv);
}

By the nature of delegation, the delegating methods look identical in every component. Only the nondelegating versions change.

VII.1.b Using CUnknown

DirectShow implements IUnknown in a base class called CUnknown. You can use CUnknown to derive other classes, overriding only the methods that change across components. Most of the other base classes in DirectShow derive from CUnknown, so your component can inherit directly from CUnknown or from another base class.

INonDelegatingUnknown

CUnknown implements INonDelegatingUnknown. It manages reference counts internally, and in most situations your derived class can inherit the two reference-counting methods with no change. Be aware that CUnknown deletes itself when the reference count drops to zero. On the other hand, you must override CUnknown::NonDelegatingQueryInterface, because the method in the base class returns E_NOINTERFACE if it receives any IID other than IID_IUnknown. In your derived class, test for the IIDs of interfaces that you support, as shown in the following example:

STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_ISomeInterface)
        return GetInterface((ISomeInterface*)this, ppv);
    // default
    return CUnknown::NonDelegatingQueryInterface(riid, ppv);
}

The utility function GetInterface (see COM Helper Functions) sets the pointer, increments the reference count in a thread-safe way, and returns S_OK. In the default case, call the base class method and return the result. If you derive from another base class, call its NonDelegatingQueryInterface method instead. This enables you to support all the interfaces that the parent class supports.

IUnknown

As mentioned earlier, the delegating version of IUnknown is the same for every component, because it does nothing more than invoke the correct instance of the nondelegating version. For convenience, the header file Combase.h contains a macro, DECLARE_IUNKNOWN, which declares the three delegating methods as inline methods. It expands to the following code:

STDMETHODIMP QueryInterface(REFIID riid, void **ppv) {
    return GetOwner()->QueryInterface(riid, ppv);
};
STDMETHODIMP_(ULONG) AddRef() {
    return GetOwner()->AddRef();
};
STDMETHODIMP_(ULONG) Release() {
    return GetOwner()->Release();
};

The utility function CUnknown::GetOwner retrieves a pointer to the IUnknown interface of the component that owns this component. For an aggregated component, the owner is the outer component. Otherwise, the component owns itself. Include the DECLARE_IUNKNOWN macro in the public section of your class definition.

Class Constructor

Your class constructor should invoke the constructor method for the parent class, in addition to anything it does that is specific to your class. The following example is a typical constructor method:

CMyComponent(TCHAR *tszName, LPUNKNOWN pUnk, HRESULT *phr)
: CUnknown(tszName, pUnk, phr)
{ /* Other initializations */
}

The method takes the following parameters, which it passes directly to the CUnknown constructor method.

• tszName specifies a name for the component.
• pUnk is a pointer to the aggregating IUnknown.
• pHr is a pointer to an HRESULT value, indicating the success or failure of the method.

Summary

The following example shows a derived class that supports IUnknown and a hypothetical interface named ISomeInterface:

class CMyComponent : public CUnknown, public ISomeInterface
public:
DECLARE_IUNKNOWN;

STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv)
{  
    if( riid == IID_ISomeInterface )
    {  
        return GetInterface((ISomeInterface*)this, ppv);
    }  
    return CUnknown::NonDelegatingQueryInterface(riid, ppv);
}

CMyComponent(TCHAR *tszName, LPUNKNOWN pUnk, HRESULT *phr)
: CUnknown(tszName, pUnk, phr)
{  
    /* Other initializations */
};

// More declarations will be added later.
}

This example illustrates the following points:

- The CUnknown class implements the IUnknown interface. The new component inherits from CUnknown and from any interfaces that the component supports. The component could derive instead from another base class that inherits from CUnknown.
- The DECLARE_IUNKNOWN macro declares the delegating IUnknown methods as inline methods.
- The CUnknown class provides the implementation for INonDelegatingUnknown.
- To support an interface other than IUnknown, the derived class must override the NonDelegatingQueryInterface method and test for the IID of the new interface.
- The class constructor invokes the constructor method for CUnknown. For more information, see How to Create a DLL.

VII.2 How to Create a DLL

This article describes how to implement a component as a dynamic-link library (DLL) in Microsoft DirectShow. This article is a continuation from How to Implement IUnknown, which describes how to implement the IUnknown interface by deriving your component from the CUnknown base class. This article contains the following sections:

a. Class Factories and Factory Templates
b. Factory Template Array
c. DLL Functions

dll::class-template::registry

dll::class-template::registration

dll::class-template::class-template

CoGetClassObject calls the DllGetClassObject function, which is defined in the DLL. This function creates the class factory and returns a pointer to an interface on the class factory. DirectShow implements DllGetClassObject for you, but the function relies on your code in a specific way. To understand how it works, you must understand how DirectShow implements class factories.

A class factory is a COM object dedicated to creating another COM object. Each class factory has one type of object that it creates. In DirectShow, every class factory is an instance of the same C++ class, CClassFactory. Class factories are specialized by means of another class, CFactoryTemplate, also called the factory template. Each class factory holds a pointer to a factory template. The factory template contains information about a specific component, such as the component's class identifier (CLSID), and a pointer to a function that creates the component. The DLL declares a global array of factory templates, one for each component in the DLL. When DllGetClassObject makes a new class factory, it searches the array for a template with a matching CLSID. Assuming it finds one, it creates a class factory that holds a pointer to the matching template. When the client calls IClassFactory::CreateInstance, the class factory calls the instantiation function defined in the template.

The following illustration shows the sequence of method calls.
The benefit of this architecture is that you can define just a few things that are specific to your component, such as the instantiation function, without implementing the entire class factory.

### VII.2.b Factory Template Array
The factory template contains the following public member variables:

- `const WCHAR * m_Name;` // Name
- `const CLSID * m_ClsID;` // CLSID
- `LPFNNewCOMObject m_lpfnNew;` // Function to create an instance of the component
- `LPFNInitRoutine m_lpfnInit;` // Initialization function (optional)
- `const AMOVIESETUP_FILTER * m_pAMovieSetup_Filter;` // Set-up information (for filters)

The two function pointers, `m_lpfnNew` and `m_lpfnInit`, use the following type definitions:

```c
typedef CUnknown *(CALLBACK *LPFNNewCOMObject)(LPUNKNOWN pUnkOuter, HRESULT *phr)
typedef void (CALLBACK *LPFNInitRoutine)(BOOL bLoading, const CLSID *rclsid)
```

The first is the instantiation function for the component. The second is an optional initialization function. If you provide an initialization function, it is called from inside the DLL entry-point function. (The DLL entry-point function is discussed later in this article.)

Suppose you are creating a DLL that contains a component named CMyComponent, which inherits from `CUnknown`. You must provide the following items in your DLL:

- The initialization function, a public method that returns a new instance of CMyComponent.

```c
CUnknown * WINAPI CMyComponent::CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CMyComponent *pNewObject = new CMyComponent(NAME("My Component"), pUnk, pHr);
    if (pNewObject == NULL) {
        *pHr = E_OUTOFMEMORY;
    }
    return pNewObject;
}
```

```c
CFactoryTemplate g_Templates[1] =
{
    {L"My Component", &CLSID_MyComponent, CMyComponent::CreateInstance, NULL, NULL} // Name CLSID Method to create an instance of MyComponent NULL, // Initialization function (for filters)
};
```

The `CreateInstance` method calls the class constructor and returns a pointer to the new class instance. The parameter `pUnk` is a pointer to the aggregating `IUnknown`. You can simply pass this parameter to the class constructor. The parameter `pHr` is a pointer to an HRESULT value. The class constructor sets this to an appropriate value, but if the constructor fails, set the value to `E_OUTOFMEMORY`.

The `NAME` macro generates a string in debug builds but resolves to NULL in retail builds. It is used in this example to give the component a name that appears in debug logs but does not occupy memory in the final version. The `CreateInstance` method can have any name, because the class factory refers to the function pointer in the factory template. However, `g_Templates` and `g_cTemplates` are global variables that the class factory expects to find, so they must have exactly those names.

### VII.2.c DLL Functions
A DLL must implement the following functions so that it can be registered, unregistered, and loaded into memory.
Extrait de la documentation DirectShow

- **DllMain**: The DLL entry point. The name **DllMain** is a placeholder for the library-defined function name. The DirectShow implementation uses the name **DllEntryPoint**. For more information, see the Platform SDK.
- **DllGetClassObject**: Creates a class factory instance. Described in the previous sections.
- **DllCanUnloadNow**: Queries whether the DLL can safely be unloaded.
- **DllRegisterServer**: Creates registry entries for the DLL.
- **DllUnregisterServer**: Removes registry entries for the DLL.

Of these, the first three are implemented by DirectShow. If your factory template provides an initialization function in the **m_lPFNInit** member variable, that function is called from inside the DLL entry-point function. For more information on when the system calls the DLL entry-point function, see **DllMain** in the Platform SDK.

You must implement **DllRegisterServer** and **DllUnregisterServer**, but DirectShow provides a function named **AMovieDllRegisterServer2** that does the necessary work.

Your component can simply wrap this function, as shown in the following example:

```c
STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2( TRUE );
}

STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
}
```

However, within **DllRegisterServer** and **DllUnregisterServer** you can customize the registration process as needed. If your DLL contains a filter, you might need to do some additional work. For more information, see How to Register DirectShow Filters.

In your module-definition (.def) file, export all the DLL functions except for the entry-point function. The following is an example .def file:

```c
EXPORTS
    DllGetClassObject PRIVATE
    DllCanUnloadNow PRIVATE
    DllRegisterServer PRIVATE
    DllUnregisterServer PRIVATE
```

You can register the DLL using the Regsvr32.exe utility.

**VII.3 How to Register DirectShow Filters**

This article describes how to make a Microsoft DirectShow filter self-registering. It contains the following sections:

a. **Layout of the Registry Keys**
b. **Declaring Filter Information**
c. **Declaring the Factory Template**
d. **Implementing DllRegisterServer**
e. **Guidelines for Registering Filters**

This article does not describe how to create a DLL. For information on creating DLLs, see How to Create a DLL.

**VII.3.a Layout of the Registry Keys**

DirectShow filters are registered in two places:

- The DLL that contains the filter is registered as the filter's COM server. When an application calls **CoCreateInstance** to create the filter, the Microsoft Windows COM library uses this registry entry to locate the DLL.
- Additional information about the filter can be registered within a filter category. This information enables the System Device Enumerator and the Filter Mapper to locate the filter.

Filters are not required to register the additional filter information. As long as the DLL is registered as the COM server, an application can create the filter and add it to a filter graph. However, if you want your filter to be discoverable by the System Device Enumerator or the Filter Mapper, you must register the additional information.

The registry entry for the DLL has the following keys:

- **HKEY_CLASSES_ROOT\CLSID\**
  - **Filter CLSID**: (Default) = Friendly name
  - **InprocServer32**: (Default) = File name of the DLL
  - **ThreadingModel**: Both

The registry entry for the filter information has the following keys:

- **HKEY_CLASSES_ROOT\CLSID\**
  - **Instance**: (Default) = Friendly name

Category is the GUID of a filter category. (See Filter Categories.) The filter information is packed into a binary format. The **IFilterMapper2** interface unpacks this data when it searches the registry for a filter.

All of the filter category GUIDs are listed in the registry under the following key:

- **HKEY_CLASSES_ROOT\CLSID\**

**VII.3.b Declaring Filter Information**

The first step is to declare the filter information, if needed. DirectShow defines the following structures for describing filters, pins, and media types:
**Structure**  
**Description**

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMOVIESETUP_FILTER</td>
<td>Describes a filter.</td>
</tr>
<tr>
<td>AMOVIESETUP_PIN</td>
<td>Describes a pin.</td>
</tr>
<tr>
<td>AMOVIESETUP_MEDIATYPE</td>
<td>Describes a media type.</td>
</tr>
</tbody>
</table>

These structures are nested. The `AMOVIESETUP_FILTER` structure has a pointer to an array of `AMOVIESETUP_PIN` structures, and each of these has a pointer to an array of `AMOVIESETUP_MEDIATYPE` structures. Taken together, these structures provide enough information for the `IFilterMapper2` interface to locate a filter. They are not a complete description of a filter. For example, if the filter creates multiple instances of the same pin, you should declare only one `AMOVIESETUP_PIN` structure for that pin. Also, a filter is not required to support every combination of media types that it registers; nor is required to register every media type that it supports.

Declare the set-up structures as global variables within your DLL. The following example shows a filter with one output pin:

```c
static const WCHAR g_wszName[] = L"Some Filter";
AMOVIESETUP_MEDIATYPE sudMediaTypes[] = {
    { &MEDIATYPE_Video, &MEDIASUBTYPE_RGB24 },
    { &MEDIATYPE_Video, &MEDIASUBTYPE_RGB32 },
};
AMOVIESETUP_PIN sudOutputPin = {
    L"",            // Obsolete, not used.
    FALSE,          // Is this pin rendered?
    TRUE,           // Is it an output pin?
    FALSE,          // Can the filter create zero instances?
    FALSE,          // Does the filter create multiple instances?
    &GUID_NULL,     // Obsolete.
    NULL,           // Obsolete.
    2,              // Number of media types.
    sudMediaTypes   // Pointer to media types.
};
AMOVIESETUP_FILTER sudFilterReg = {
    &CLSID_SomeFilter, // Filter CLSID.
    g_wszName,         // Filter name.
    MERIT_NORMAL,      // Merit.
    1,                 // Number of pin types.
    &sudOutputPin      // Pointer to pin information.
};
```

The filter name is declared as a static global variable, because it will be used again elsewhere.

**AMOVIESETUP_FILTER Structure**

The `AMOVIESETUP_FILTER` structure contains information for registering a filter.

### Syntax

```c
typedef struct _AMOVIESETUP_FILTER {
    const CLSID           *clsID;
    const WCHAR           *strName;
    DWORD                 dwMerit;
    UINT                  nPins;
    const AMOVIESETUP_PIN *lpPin;
} AMOVIESETUP_FILTER, *PAMOVIESETUP_FILTER, *FAR LPAMOVIESETUP_FILTER;
```

### Members
- `clsID` : Class identifier of the filter.
- `strName` : Name of the filter.
- `dwMerit` : Filter merit. Used by the `IGraphBuilder` interface when constructing a filter graph.
- `nPins` : Number of pins on the filter.
- `lpPin` : Pointer to an array of `AMOVIESETUP_PIN` structures, of size `nPins`.

### Requirements
- **Header**: Dshow.h.

**AMOVIESETUP_PIN Structure** : See `REGFILTERPINS`.

**REGFILTERPINS Structure**

The `REGFILTERPINS` structure contains pin information for registering a filter.

### Syntax

```c
typedef struct {
    LPWSTR            strName;
    BOOL              bRendered;
    BOOL              bOutput;
    BOOL              bZero;
    BOOL              bMany;
    const CLSID       *clsConnectsToFilter;
    const WCHAR       *strConnectsToPin;
    UINT              nMediaTypes;
    const REGPINTYPES *lpMediaType;
} REGFILTERPINS;
```

### Members
- `strName` : Name of the pin. (Obsolete.)
- `bRendered` : If TRUE, the filter renders the input from this pin. (Applies only to input pins. For output pins, the value is always FALSE.)
- `bOutput` : If TRUE, this pin is an output pin. Otherwise, the pin is an input pin.
- `bZero` : If TRUE, the filter can have zero instances of this pin.
- `bMany` : If TRUE, the filter can create more than one instance of this type of pin.
Extrait de la documentation DirectShow

**clsConnectsToFilter:** Class identifier (CLSID) of the filter to which this pin connects. (Obsolete.)

**strConnectsToPin:** Name of the pin to which this pin connects. (Obsolete.)

**nMediaTypes:** Number of media types supported by this pin.

**lpMediaType:** Pointer to an array of `REGPINTYPES` structures, of size `nMediaTypes`.

**Remarks**

This structure is used in the `IFilterMapper2` interface for filter registration. If you use this structure, set the `dwVersion` member of the `REGFILTER2` structure to 1. If you need to register a medium or pin category for the pin, use the `REGFILTERPINS2` structure instead. In that case, set the `REGFILTER2` structure's `dwVersion` member to 2. The equivalent AMOVIESETUP_PIN type is used in class factory templates (`CFactoryTemplate`).

The `strName`, `clsConnectsToFilter`, and `strConnectsToPin` members are obsolete. Their values are not added to the registry. For more information, see How to Register DirectShow Filters.

**Requirements**

Header: `Dshow.h`.

**See Also**

- DirectShow Structures

AMOVIESETUP_MEDIATYPE Structure: see `REGPINTYPES`.

**REGPINTYPES Structure**

The `REGPINTYPES` structure contains media type information for registering a filter.

**Syntax**

```c
typedef struct
{
    const CLSID *clsMajorType;
    const CLSID *clsMinorType;
} REGPINTYPES;
```

**Members**

- `clsMajorType`: Major type GUID of the media type.
- `clsMinorType`: Sub type GUID of the media type. Can be `MEDIASUBTYPE_NULL`.

**Remarks**

Extrait de la documentation DirectShow

This structure is used by the `IFilterMapper2` interface to identify the media types that a pin supports. The equivalent AMOVIESETUP_MEDIATYPE type is used in class factory templates (`CFactoryTemplate`). To register a range of subtypes within the same major type, use the value `MEDIASUBTYPE_NULL`. For more information, see How to Register DirectShow Filters.

**Requirements**

Header: `Dshow.h`.

**See Also**

- DirectShow Structures
- Media Types

**VII.3.c Declaring the Factory Template**

The next step is to declare the factory template for your filter. A factory template is a C++ class that contains information for the class factory. In your DLL, declare a global array of `CFactoryTemplate` objects, one for each filter or COM component in your DLL. The array must be named `g_Templates`. For more information about factory templates, see How to Create a DLL.

The `m_pAMovieSetup_Filter` member of the factory template is a pointer to the AMOVIESETUP_FILTER structure described previously. The following example shows a factory template, using the structure given in the previous example:

```c
CFactoryTemplate g_Templates[] = {
    {g_wszName,                      // Name.
      &CLSID_SomeFilter,              // CLSID.
      CSomeFilter::CreateInstance,    // Creation function.
      NULL,
      &sudFilterReg                   // Pointer to filter information.
    },
    {};

    int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

If you did not declare any filter information, `m_pAMovieSetup_Filter` can be NULL.

**VII.3.d Implementing DllRegisterServer**

The final step is to implement the `DllRegisterServer` function. The DLL that contains the component must export this function. The function will be called by a set-up application, or when the user runs the Regsvr32.exe tool.

The following example shows a minimal implementation of `DllRegisterServer`:

```c
STDAPI DllRegisterServer(void)
{
    return AMovieDllRegisterServer2(TRUE);
}
```
The **AMovieDllRegisterServer2** function creates registry entries for every component in the **g_Templates** array. However, this function has some limitations. First, it assigns every filter to the "DirectShow Filters" category (CLSID_LegacyAmFilterCategory), but not every filter belongs in this category. Capture filters and compression filters, for example, have their own categories. Second, if your filter supports a hardware device, you might need to register two additional pieces of information that **AMovieDllRegisterServer2** does not handle: the **medium** and the **pin category**. A medium defines a method of communication in a hardware device, such as a bus. The pin category defines the function of a pin. For information on mediums, see **KSPIN_MEDIUM** in the Microsoft Windows Driver Development Kit (DDK). For a list of pin categories, see **Pin Property Set**.

If you want to specify a filter category, a medium, or a pin category, call the **IFilterMapper2::RegisterFilter** method from within **DllRegisterServer**. This method takes a pointer to a **REGFILTER2** structure, which specifies information about the filter.

To complicate matters somewhat, the **REGFILTER2** structure supports two different formats for registering pins. The **dwVersion** member specifies the format:

- If **dwVersion** is 1, the pin format is **AMOVIESETUP_PIN** (described previously).
- If **dwVersion** is 2, the pin format is **REGFILTERPINS2**.

The **REGFILTERPINS2** structure includes entries for pin mediums and pin categories. Also, it uses bit flags for some items that **AMOVIESETUP_PIN** declares as Boolean values.

The following example shows how to call **IFilterMapper2::RegisterFilter** from inside **DllRegisterServer**:

```csharp
REGFILTER2 rf2FilterReg = {
    1,              // Version 1 (no pin mediums or pin category).
    MERIT_NORMAL,   // Merit.
    1,              // Number of pins.
    &sudPins        // Pointer to pin information.
};

STDAPI DllRegisterServer(void)
{
    HRESULT hr;
    IFilterMapper2 *pFM2 = NULL;
    hr = AMovieDllRegisterServer2(TRUE);
    if (FAILED(hr))
        return hr;
    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
                           IID_IFilterMapper2, (void **)&pFM2);
    if (FAILED(hr))
        return hr;
    hr = pFM2->RegisterFilter(
        CLSID_SomeFilter,                // Filter CLSID.
        g_wszName,                       // Filter name.
        NULL,                            // Device moniker.
        &CLSID_VideoCompressorCategory,  // Video compressor category.
        g_wszName,                       // Instance data.
        &rf2FilterReg                    // Pointer to filter information.
    );
    pFM2->Release();
    return hr;
}
```

### VII.3.e Guidelines for Registering Filters

The filter registry information determines how the Filter Graph Manager functions during Intelligent Connect. Thus, it affects every application written for DirectShow, not just the ones that will use your filter. You should make sure that your filter behaves correctly, by following these guidelines.

1. Do you need the filter data in the registry? For many custom filters, there is no reason to make the filter visible to the Filter Mapper or the System Device Enumerator. As long as you register the DLL, your application can create the filter using **CoCreateInstance**. In that case, simply omit the **AMOVIESETUP_FILTER** structure from the factory template. (One drawback is that your filter will not be visible in GraphEdit. To get around this, you can create a private "Testing" category using the **IFilterMapper2::CreateCategory** method. You should only do this for debug builds.)

2. Choose the correct filter category. The default "DirectShow Filters" category is for general purpose filters. Whenever appropriate, register your filter in a more specific category. When **IFilterMapper2** searches for a filter, it ignores any category whose merit is **MERIT_DO_NOT_USE** or less. Categories not intended for normal playback have low merit.

3. Avoid specifying **MEDIATYPE_None**, **MEDIASUBTYPE_None**, or **GUID_NULL** in the **AMOVIESETUP_MEDIATYPE** information for a pin. **IFilterMapper2** treats these as wildcards, which can slow the graph-building process.

4. Choose the lowest merit value possible. Here are some guidelines:

<table>
<thead>
<tr>
<th>Type of filter</th>
<th>Recommended merit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default renderer</td>
<td>MERIT_PREFERRED. For standard media types, however, a custom renderer should never be the default.</td>
</tr>
<tr>
<td>Non-default renderer</td>
<td>MERIT_DO_NOT_USE or MERIT_UNLIKELY</td>
</tr>
<tr>
<td>Max</td>
<td>MERIT_DO_NOT_USE</td>
</tr>
<tr>
<td>Decoder</td>
<td>MERIT_NORMAL</td>
</tr>
<tr>
<td>Spitter, parser</td>
<td>MERIT_NORMAL or lower</td>
</tr>
<tr>
<td>Special purpose filter; any filter that is created directly by the</td>
<td>MERIT_DO_NOT_USE</td>
</tr>
</tbody>
</table>
**VIII. Writing Transform Filters**

This section describes how to write a *transform filter*, defined as a filter that has exactly one input pin and one output pin. To illustrate the steps, this section describes a hypothetical transform filter that outputs run-length encoded (RLE) video. It does not describe the RLE-encoding algorithm itself, only the tasks that are specific to DirectShow. (DirectShow already provides an RLE codec through the AVI Compressor filter.)

---

**VII.3.f Unregistering a Filter**

To unregister a filter, implement the `DllUnregisterServer` function. Within this function, call the DirectShow `AMovieDllRegisterServer2` function with a value of FALSE. If you called `IFilterMapper2::RegisterFilter` when you registered the filter, call the `IFilterMapper2::UnregisterFilter` method here.

The following example shows how to unregister a filter:

```
STDAPI DllUnregisterServer()
{
    HRESULT hr;
    IFilterMapper2 *pFM2 = NULL;
    hr = AMovieDllRegisterServer2(FALSE);
    if (FAILED(hr))
     return hr;
    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
                          IID_IFilterMapper2, (void **)&pFM2);
    if (FAILED(hr))
     return hr;
    hr = pFM2->UnregisterFilter(&CLSID_VideoCompressorCategory,
                               g_wszName, CLSID_SomeFilter);
    pFM2->Release();
    return hr;
}
```

---

**Important**

In-place video transforms can have a serious impact on rendering performance. In-place transforms require read-modify-write operations on the buffer. If the memory resides on a graphics card, read operations are significantly slower. Moreover, even a copy transform can cause unintended read operations if you do not implement it.
**VIII.2  Step 2: Declare the Filter Class**

Start by declaring a C++ class that inherits the base class:
```cpp
class CRleFilter : public CTransformFilter
{
    /* Declarations will go here. */
};
```

Each of the filter classes has associated pin classes. Depending on the specific needs of your filter, you might need to override the pin classes. In the case of `CTransformFilter`, the pins delegate most of their work to the filter, so you probably don't need to override the pins.

You must generate a unique CLSID for the filter. You can use the Guidgen or Uuidgen utility; never copy an existing GUID. There are several ways to declare a CLSID. The following example uses the DEFINE_GUID macro:

```cpp
#define RLEFILT CLSID_RLEFilter,
0x1915C5C7, 0x2AA, 0x415F, 0x89, 0xF, 0x76, 0xD9, 0x4C, 0x85, 0xAA, 0xF1);
```

Next, write a constructor method for the filter:
```cpp
CRleFilter::CRleFilter()
    : CTransformFilter(NAME("My RLE Encoder"), 0, CLSID_RLEFilter)
    { /* Initialize any private variables here. */ }
```

Notice that one of the parameters to the `CTransformFilter` constructor is the CLSID defined earlier.

**VIII.3  Step 3: Support Media Type Negotiation**

When two pins connect, they must agree on a media type for the connection. The media type describes the format of the data. Without the media type, a filter might deliver one kind of data, only to have another filter treat it as something else.

The basic mechanism for negotiating media types is the `IPin::ReceiveConnection` method. The output pin calls this method on the input pin with a proposed media type.

The input pin accepts the connection or rejects it. If it rejects the connection, the output pin can try another media type. If no suitable types are found, the connection fails. Optionally, the input pin can advertise a list of types that it prefers, through the `IPin::EnumMediaTypes` method. The output pin can use this list when it proposes media types, although it does not have to.

The `CTransformFilter` class implements a general framework for this process, as follows:

- The input pin has no preferred media types. It relies entirely on the upstream filter to propose the media type. For video data, this makes sense, because the media type includes the image size and the frame rate. Typically, that information must be supplied by an upstream source filter or parser filter. In the case of audio data, the set of possible formats is smaller, so it may be practical for the input pin to offer some preferred types. In that case, override `CBasePin::GetMediaType` on the input pin.
- When the upstream filter proposes a media type, the input pin calls the `CTransformFilter::CheckInputType` method, which accepts or rejects the type.
- The output pin will not connect unless the input pin is connected first. This behavior is typical for transform filters. In most cases, the filter must determine the input type before it can set the output type.
- When the output pin does connect, it has a list of media types that it proposes to the downstream filter. It calls the `CTransformFilter::GetMediaTypes` method to generate this list. The output pin will also try any media types that the downstream filter proposes.
- To check whether a particular output type is compatible with the input type, the output pin calls the `CTransformFilter::CheckMediaType` method.

The three `CTransformFilter` methods listed previously are pure virtual methods, so your derived class must implement them. None of these methods belongs to a COM interface; they are simply part of the implementation provided by the base classes.

The following sections describe each method in more detail:

a. **Step 3A. Implement the CheckInputType Method**
b. **Step 3B. Implement the GetMediaTypes Method**
c. **Step 3C. Implement the CheckMediaType Method**

**See Also**
- How Filters Connect
not valid, return VFW_E_TYPE_NOT_ACCEPTED. If the entire media type is valid, return S_OK.

For example, in the RLE encoder filter, the input type must be 8-bit or 4-bit uncompressed RGB video. There is no reason to support other input formats, such as 16- or 24-bit RGB, because the filter would have to convert them to a lower bit depth, and DirectShow already provides a Color Space Converter filter for that purpose. The following example assumes that the encoder supports 8-bit video but not 4-bit video:

```c++
HRESULT CRleFilter::CheckInputType(const CMediaType *mtIn)
{
    if ((mtIn->majortype != MEDIATYPE_Video) ||
        (mtIn->subtype != MEDIASUBTYPE_RGB8) ||
        (mtIn->formattype != FORMAT_VideoInfo) ||
        (mtIn->cbFormat < sizeof(VIDEOINFOHEADER)))
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    VIDEOINFOHEADER *pVih =
        reinterpret_cast<VIDEOINFOHEADER*>(mtIn->pbFormat);
    if ((pVih->bmiHeader.biBitCount != 8) ||
        (pVih->bmiHeader.biCompression != BI_RGB))
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    // Check the palette table.
    if (pVih->bmiHeader.biClrUsed > PALETTE_ENTRIES(pVih))
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    DWORD cbPalette = pVih->bmiHeader.biClrUsed * sizeof(RGBQUAD);
    if (mtIn->cbFormat < sizeof(VIDEOINFOHEADER) + cbPalette)
    {
        return VFW_E_TYPE_NOT_ACCEPTED;
    }

    // Everything is good.
    return S_OK;
}
```

In this example, the method first checks the major type and subtype. Then it checks the format type, to make sure the format block is a VIDEOINFOHEADER structure. The filter could also support VIDEOINFOHEADER2, but in this case there would be no real benefit. The VIDEOINFOHEADER2 structure adds support for interlacing and non-square pixels, which are not likely to be relevant in 8-bit video.

If the format type is correct, the example checks the `biBitCount` and `biCompression` members of the VIDEOINFOHEADER structure, to verify that the format is 8-bit uncompressed RGB. As this example shows, you must coerce the `pbFormat` pointer to the correct structure, based on the format type. Always check the format type GUID (formattype) and the size of the format block (cbFormat) before casting the pointer.

---

In this example, the method first checks the major type and subtype. Then it checks the format type, to make sure the format block is a VIDEOINFOHEADER structure. The filter could also support VIDEOINFOHEADER2, but in this case there would be no real benefit. The VIDEOINFOHEADER2 structure adds support for interlacing and non-square pixels, which are not likely to be relevant in 8-bit video.

If the format type is correct, the example checks the `biBitCount` and `biCompression` members of the VIDEOINFOHEADER structure, to verify that the format is 8-bit uncompressed RGB. As this example shows, you must coerce the `pbFormat` pointer to the correct structure, based on the format type. Always check the format type GUID (formattype) and the size of the format block (cbFormat) before casting the pointer.
Extrait de la documentation DirectShow

- It assigns a new subtype GUID, which is constructed from the FOURCC code 'MRLE', using the FOURCCMap class.
- It calls the CMediaType::SetVariableSize method, which sets the bFixedSizeSamples flag to FALSE and the lSampleSize member to zero, indicating variable-sized samples.
- It calls the CMediaType::SetTemporalCompression method with the value FALSE, indicating that every frame is a key frame. (This field is informational only, so you could safely ignore it.)
- It sets the biCompression field to BI_RLE8.
- It sets the biSizeImage field to the image size.

VIII.3.c  Step 3C: Implement the CheckTransform Method

Note This step is not required for filters that derive from CTransInPlaceFilter.

The CTransformFilter::CheckTransform method checks if a proposed output type is compatible with the current input type. The method is also called if the input pin reconnects after the output pin connects. The following example verifies whether the format is RLE8 video; the image dimensions match the input format; and the palette entries are the same. It also rejects source and target rectangles that do not match the image size.

```cpp
HRESULT CRIeFilter::CheckTransform(
const CMediaType *mtIn, const CMediaType *mtOut)
{
  // Check the major type.
  if (mtOut->majortype != MEDIATYPE_Video)
    return VFW_E_TYPE_NOT_ACCEPTED;

  // Check the subtype and format type.
  FOURCCMap fccMap = FCC('MRLE');
  if (mtOut->subtype != static_cast<GUID>(fccMap))
    return VFW_E_TYPE_NOT_ACCEPTED;
  if ((mtOut->formattype != FORMAT_VideoInfo) ||
      (mtOut->cbFormat < sizeof(VIDEOINFOHEADER)))
    return VFW_E_TYPE_NOT_ACCEPTED;

  // Compare the bitmap information against the input type.
  BITMAPINFOHEADER *pBmiOut = HEADER(mtOut->pbFormat);
  BITMAPINFOHEADER *pBmiIn = HEADER(mtIn->pbFormat);
  if ((pBmiOut->biCompression != BI_RLE8) ||
      (pBmiOut->biWidth != pBmiIn->biWidth) ||
      (pBmiOut->biHeight != pBmiIn->biHeight))
    return VFW_E_TYPE_NOT_ACCEPTED;

  // Compare source and target rectangles.
  RECT rcImg;
  SetRect(&rcImg, 0, 0, pBmiIn->biWidth, pBmiIn->biHeight);
  RECT *prcSrc = &((VIDEOINFOHEADER*)(mtIn->pbFormat))->rcSource;
  RECT *prcTarget = &((VIDEOINFOHEADER*)(mtOut->pbFormat))->rcTarget;
  if (!IsRectEmpty(prcSrc) && !EqualRect(prcSrc, &rcImg))
    return VFW_E_INVALIDMEDIATYPE;
  if (!IsRectEmpty(prcTarget) && !EqualRect(prcTarget, &rcImg))
    return VFW_E_INVALIDMEDIATYPE;

  if (pBmiOut->biClrUsed != pBmiIn->biClrUsed)
    return VFW_E_TYPE_NOT_ACCEPTED;
  DWORD cbPalette = pBmiOut->biClrUsed * sizeof(RGBQUAD);
  if (mtOut->cbFormat < sizeof(VIDEOINFOHEADER) + cbPalette)
    return VFW_E_TYPE_NOT_ACCEPTED;
  if (0 != memcmp(pBmiOut + 1, pBmiIn + 1, cbPalette))
    return VFW_E_TYPE_NOT_ACCEPTED;

  // Everything is good.
  return S_OK;
}
```

Pin Reconnections

Applications can disconnect and reconnect pins. Suppose an application connects both pins, disconnects the input pin, and then reconnects the input pin using a new image size. In that case, CheckTransform fails because the dimensions of the image no longer match. This behavior is reasonable, although the filter could also try reconnecting the output pin with a new media type.

See Also
- Reconnecting Pins
- Source and Target Rectangles in Video Renderers

VIII.4  Step 4: Set Allocator Properties

Note This step is not required for filters that derive from CTransInPlaceFilter.
After two pins agree on a media type, they select an allocator for the connection and negotiate allocator properties, such as the buffer size and the number of buffers. In the `CTransformFilter` class, there are two allocators, one for the upstream pin connection and one for the downstream pin connection. The upstream filter selects the upstream allocator and also chooses the allocator properties. The input pin accepts whatever the upstream filter decides. If you need to modify this behavior, override the `CBaseInputPin::NotifyAllocator` method.

The transform filter's output pin selects the downstream allocator. It performs the following steps:

1. If the downstream filter can provide an allocator, the output pin uses that one.
   Otherwise, the output pin creates a new allocator.
2. The output pin gets the downstream filter's allocator requirements (if any) by calling `IMemInputPin::GetAllocatorRequirements`.
3. The output pin calls the transform filter's `CTransformFilter::DecideBufferSize` method, which is pure virtual. The parameters to this method are a pointer to the allocator and an `ALLOCATOR_PROPERTIES` structure with the downstream filter's requirements. If the downstream filter has no allocator requirements, the structure is zeroed out.
4. In the `DecideBufferSize` method, the derived class sets the allocator properties by calling `IMemAllocator::SetProperties`.

Generally, the derived class will select allocator properties based on the output format, the downstream filter's requirements, and the filter's own requirements. Try to select properties that are compatible with the downstream filter's request. Otherwise, the downstream filter might reject the connection.

In the following example, the RLE encoder sets minimum values for the buffer size, buffer alignment, and buffer count. For the prefix, it uses whatever value the downstream filter requested. The prefix is typically zero bytes, but some filters require it. For example, the `AVI Mux` filter uses the prefix to write RIFF headers.

```cpp
HRESULT CRleFilter::DecideBufferSize(IMemAllocator *pAlloc, ALLOCATOR_PROPERTIES *pProp)
{
    AM_MEDIA_TYPE mt;
    HRESULT hr = m_pOutput->ConnectionMediaType(&mt);
    if (FAILED(hr))
    {
        return hr;
    }
    ASSERT(mt.formattype == FORMAT_VideoInfo);
    HRESULT hr = p_poutput->ConnectionMediaType(&mt);
    if (FAILED(hr))
    {
        return hr;
    }
    ASSERT(mt.formattype == FORMAT_VideoInfo);
    BITMAPINFOHEADER *pbmi = HEADER(mt.pbFormat);
    pProp->cbBuffer = DIBSIZE(*pbmi) * 2;
    if (pProp->cBuffers == 0)
    {
        pProp->cBuffers = 1;
    }
    if (pProp->cAlign == 0)
    {
        pProp->cAlign = 1;
    }
    if (pProp->cBuffers == 0)
    {
        // Release the format block.
        FreeMediaType(mt);
        // Set allocator properties.
        ALLOCATOR_PROPERTIES Actual;
        hr = pAlloc->SetProperties(pProp, &Actual);
        if (!FAILED(hr))
        {
            return hr;
        }
        // Even when it succeeds, check the actual result.
        if (pProp->cbBuffer > Actual.cbBuffer)
        {
            return E_FAIL;
        }
        return S_OK;
    }
    return S_OK;
}
```

The allocator may not be able to match your request exactly. Therefore, the `SetProperties` method returns the actual result in a separate `ALLOCATOR_PROPERTIES` structure (the `Actual` parameter, in the previous example). Even when `SetProperties` succeeds, you should check the result to make sure they meet your filter's minimum requirements.

### Custom Allocators

By default, all of the filter classes use the `CMemAllocator` class for their allocators. This class allocates memory from the virtual address space of the client process (using `VirtualAlloc`). If your filter needs to use some other kind of memory, such as DirectDraw surfaces, you can implement a custom allocator. You can use the `CBaseAllocator` class or write an entirely new allocator class. If your filter has a custom allocator, override the following methods, depending on which pin uses the allocator:

- **Input pin**: `CBaseInputPin::GetAllocator` and `CBaseInputPin::NotifyAllocator`.
- **Output pin**: `CBaseOutputPin::DecideAllocator`.

If the other filter refuses to connect using your custom allocator, your filter can either fail the connection, or else connect with the other filter's allocator. In the latter case, you might need to set an internal flag indicating the type of allocator. For an example of this approach, see `CDrawImage Class`.

### VIII.5  Step 5: Transform the Image

The upstream filter delivers media samples to the transform filter by calling the `IMemInputPin::Receive` method on the transform filter's input pin. To process the data, the transform filter calls the `Transform` method, which is pure virtual. The `CTransformFilter` and `CTransInPlaceFilter` classes use two different versions of this method:

- `CTransformFilter::Transform` takes a pointer to the input sample and a pointer to the output sample. Before the filter calls the method, it copies the
sample properties from the input sample to the output sample, including the
time stamps.

- **CTransInPlaceFilter::Transform** takes a pointer to the input sample. The
  filter modifies the data in place.

If the **Transform** method returns S_OK, the filter delivers the sample downstream. To
skip a frame, return S_FALSE. If there is a streaming error, return a failure code.

The following example shows how the RLE encoder might implement this method.
Your own implementation might differ considerably, depending on what your filter
does.

```cpp
HRESULT CBaseFilter::NonDelegatingQueryInterface(REFIID iid, void **ppv)
{
    // Get pointers to the underlying buffers.
    BYTE *pBufferIn, *pBufferOut;
    hr = pSource->GetPointer(&pBufferIn);
    if (FAILED(hr))
    {
        return hr;
    }
    hr = pDest->GetPointer(&pBufferOut);
    if (FAILED(hr))
    {
        return hr;
    }
    // Process the data.
    DWORD cbDest = EncodeFrame(pBufferIn, pBufferOut);
    ENSURE((long)cbDest <= pDest->GetSize());
    pDest->SetActualDataLength(cbDest);
    pDest->SetSyncPoint(TRUE);
    return S_OK;
}
```

This example assumes that EncodeFrame is a private method that implements the RLE
encoding. The encoding algorithm itself is not described here; for details, see the topic
"Bitmap Compression" in the Platform SDK documentation.

First, the example calls **IMediaSample::GetPointer** to retrieve the addresses of the
underlying buffers. It passes these to the private EncoderFrame method. Then it calls
**IMediaSample::SetActualDataLength** to specify the length of the encoded data. The
downstream filter needs this information so that it can manage the buffer properly.

Finally, the method calls **IMediaSample::SetSyncPoint** to set the key frame flag to
TRUE. Run-length encoding does not use any delta frames, so every frame is a key
frame. For delta frames, set the value to FALSE.

Other issues that you must consider include:

- **Time stamps.** The **CTransformFilter** class timestamps the output sample
  before calling the **Transform** method. It copies the time stamp values from the
  input sample, without modifying them. If your filter needs to change the time
  stamps, call **IMediaSample::SetTime** on the output sample.

- **Format changes.** The upstream filter can change formats mid-stream by
  attaching a media type to the sample. Before doing so, it calls

```
return CBaseFilter::NonDelegatingQueryInterface(riid, ppv);
```

For more information, see Dynamic Format Changes.

- **Threads.** In both **CTransformFilter** and **CTransInPlaceFilter**, the transform
  filter delivers output samples synchronously inside the **Receive** method. The
  filter does not create any worker threads to process the data. Typically, there is
  no reason for a transform filter to create worker threads.

VIII.6  Step 6: Add Support for COM
The final step is adding support for COM.

**Reference Counting**
You do not have to implement **AddRef** or **Release**. All of the filter and pin classes
derive from **CUnknown**, which handles reference counting.

**QueryIface**
All of the filter and pin classes implement **QueryIface** for any COM interfaces
they inherit. For example, **CTransformFilter** inherits **IBaseFilter** (through
**CBaseFilter**). If your filter does not expose any additional interfaces, you do not have
to do anything else.

To expose additional interfaces, override the
**CUnknown::NonDelegatingQueryInterface** method. For example, suppose your
filter implements a custom interface named **IMyCustomInterface**. To expose this
interface to clients, do the following:

- Derive your filter class from that interface.
- Put the **DECLARE_IUNKNOWN** macro in the public declaration section.
- Override **NonDelegatingQueryInterface** to check for the IID of your interface
  and return a pointer to your filter.

The following code shows these steps:

```cpp
HRESULT CMYFilter::NonDelegatingQueryInterface(REFIID iid, void **ppv)
{
    if (iid == IID_IUnknown)
    {
        return GetInterface(static_cast<IMyCustomInterface*>(this), ppv);
    }
    return CBaseFilter::NonDelegatingQueryInterface(iid, ppv);
}
```
Object Creation

If you plan to package your filter in a DLL and make it available to other clients, you must support CoCreateInstance and other related COM functions. The base class library implements most of this; you just need to provide some information about your filter. This section gives a brief overview of what to do. For details, see How to Create a DLL.

First, write a static class method that returns a new instance of your filter. You can name this method anything you like, but the signature must match the one shown in the following example:

```cpp
CUnknown * WINAPI CRleFilter::CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CRleFilter *pFilter = new CRleFilter();
    if (pFilter == NULL)
    {
        *pHr = E_OUTOFMEMORY;
    }
    return pFilter;
}
```

Next, declare a global array of CFactoryTemplate class instances, named g_Templates. Each CFactoryTemplate class contains registry information for one filter. Several filters can reside in a single DLL; simply include additional CFactoryTemplate entries. You can also declare other COM objects, such as property pages.

```cpp
static WCHAR g_wszName[] = L“My RLE Encoder”;
CFactoryTemplate g_Templates[] = {
    { g_wszName, CLSID_RLEFilter, CRleFilter::CreateInstance, NULL, NULL }
};
```

Define a global integer named g_cTemplates whose value equals the length of the g_Templates array:

```cpp
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0])
```

Finally, implement the DLL registration functions. The following example shows the minimal implementation for these functions:

```cpp
STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2( TRUE );
}
STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
}
```

Filter Registry Entries

The previous examples show how to register a filter's CLSID for COM. For many filters, this is sufficient. The client is then expected to create the filter using CoCreateInstance and add it to the filter graph by calling IFilterGraph::AddFilter.

In some cases, however, you might want to provide additional information about the filter in the registry. This information does the following:

- Enables clients to discover the filter using the Filter Mapper or the System Device Enumerator.
- Enables the Filter Graph Manager to discover the filter during automatic graph building.

The following example registers the RLE encoder filter in the video compressor category. For details, see How to Register DirectShow Filters. Be sure to read the section Guidelines for Registering Filters, which describes the recommended practices for filter registration.

```cpp
// Declare media type information.
FOURCCMap fccMap = FCC('MRLE');
REGPINTYPES sudInputTypes = { &MEDIATYPE_Video, &GUID_NULL };
REGPINTYPES sudOutputTypes = { &MEDIATYPE_Video, (GUID*)&fccMap };

// Declare pin information.
REGFILTERPINS sudPinReg[] = {
    // Input pin.
    { 0, FALSE, // Rendered?
       FALSE, // Output?
       FALSE, // Zero?
       FALSE, // Many?
       0, 0,
       1, &sudInputTypes // Media types.
    },
    // Output pin.
    { 0, FALSE, // Rendered?
       TRUE, // Output?
       FALSE, // Zero?
       FALSE, // Many?
       0, 0,
       1, &sudOutputTypes // Media types.
    }
};
```

Finally, implement the DLL registration functions. The following example shows the minimal implementation for these functions:

```cpp
STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2( TRUE );
}
STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
}
```
2,                // Number of pins.
susPinReg         // Pointer to pin information.;
}

STDAPI DllRegisterServer(void)
{
    HRESULT hr = AMovieDllRegisterServer2(TRUE);
    if (FAILED(hr))
    {
        return hr;
    }
    IFilterMapper2 *pFM2 = NULL;
    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
        IID_IFilterMapper2, (void **)&pFM2);
    if (SUCCEEDED(hr))
    {
        hr = pFM2->RegisterFilter(
            CLSID_RLEFilter,                // Filter CLSID.
            g_wszName,                       // Filter name.
            NULL,                            // Device moniker.
            &CLSID_VideoCompressorCategory,  // Video compressor category.
            g_wszName,                       // Instance data.
            &rf2FilterReg                    // Filter information.
        );
        pFM2->Release();
    }
    return hr;
}

STDAPI DllUnregisterServer()
{
    HRESULT hr = AMovieDllRegisterServer2(FALSE);
    if (FAILED(hr))
    {
        return hr;
    }
    IFilterMapper2 *pFM2 = NULL;
    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
        IID_IFilterMapper2, (void **)&pFM2);
    if (SUCCEEDED(hr))
    {
        hr = pFM2->UnregisterFilter(&CLSID_VideoCompressorCategory,
            g_wszName, CLSID_RLEFilter);
        pFM2->Release();
    }
    return hr;
}

Also, filters do not have to be packaged inside DLLs. In some cases, you might write a specialized filter that is designed only for a specific application. In that case, you can compile the filter class directly in your application, and create it with the new operator, as shown in the following example:

```
#include "MyFilter.h"  // Header file that declares the filter class.
// Compile and link MyFilter.cpp.
int main()
```

### IX. Writing 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.

1. **Alternative Video Renderers**
2. **Source and Target Rectangles in Video Renderers**

#### IX.1 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.

**Note** Instead of writing a custom video renderer, it is recommended that you write a plug-in allocator-presenter for the Video Mixing Renderer (VMR). This approach will give you all of the benefits of the VMR, including support for DirectX Video Acceleration (DXVA), hardware deinterlacing, and frame stepping, and is likely to be more robust than a custom video renderer. For more information, see VMR Renderless Playback Mode (Custom Allocator-Presenters).

This article contains the following sections:

- Writing an Alternative Renderer
- Handling End-of-stream and Flushing Notifications

See Also
- Intelligent Connect
IX.1.a 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 DirectShow base classes 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 renderer. 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 correctly stream notifications it receives and in sending appropriate Filter Graph Manager notifications.

IX.1.b 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 seek has 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 S_FALSE. 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.)
**IX.1.c  Handling State Changes and Pause Completion**

A video renderer 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 can 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. 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 state change.

**IX.1.d  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 graph-stopped 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 enables 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).

**IX.1.e  Handling Dynamic Format Changes**

Video renderers in DirectShow accept only video formats that can be drawn efficiently. For example, the Video Renderer filter accepts only the RGB format that matches the current display device mode (for example, RGB565 when the display is set to 16-bit color). As a last resort, it also accepts 8-bit paletized 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 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 display cards can draw this format efficiently.

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 graph-stopped 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 enables 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).
all of the types that it can accept, however, so the renderer should offer some types even if the decoder does not advertise them.

If the decoder can switch to the requested format, it returns S_OK from QueryAccept. The renderer then attaches the new media type to the next media sample on the upstream allocator. For this to work, the renderer must provide a custom allocator that implements a private method for attaching the media type to the next sample. (Within this private method, call MediaSample::SetMediaType to set the type.) The renderer's input pin should return the renderer's custom allocator in the IMemInputPin::GetAllocator method. Override IMemInputPin::NotifyAllocator so that it fails if the upstream filter does not use the renderer's allocator.

With some codecs, setting biHeight to a positive number on YUV types causes the decoder to draw the image upside down. (This is incorrect, and should be considered a bug in the decoder.) 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.

IX.1.f 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.

IX.1.g 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 enables 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.

IX.1.h Handling Notifications in Full-Screen Mode

The VideoWindow 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
Extrait de la documentation DirectShow

**IX.1.i  Summary of Notifications**

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

<table>
<thead>
<tr>
<th>Event notification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC ACTIVATE</td>
<td>Sent by video renderers in full-screen rendering mode for each WM_ACTIVATEAPP message received.</td>
</tr>
<tr>
<td>EC COMPLETE</td>
<td>Sent by renderers after all data has been rendered.</td>
</tr>
<tr>
<td>EC DISPLAY COMPLETE</td>
<td>Sent by video renderers when a display format changes.</td>
</tr>
<tr>
<td>EC PALETTE CHANGED</td>
<td>Sent whenever a video renderer detects a palette change in the stream.</td>
</tr>
<tr>
<td>EC REPAINT</td>
<td>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.</td>
</tr>
<tr>
<td>EC USERABORT</td>
<td>Sent by video renderers to signal a closure that the user requested (for example, a user closing the video window).</td>
</tr>
<tr>
<td>EC VIDEO SIZE CHANGE 0</td>
<td>Sent by video renderers whenever a change in native video size is detected.</td>
</tr>
<tr>
<td>EC WINDOW DESTROYE 0</td>
<td>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.</td>
</tr>
</tbody>
</table>

Extrait de la documentation DirectShow

First, there is a size in the `bmiHeader` member of these structures. The `bmiHeader` member is a `BITMAPINFOHEADER` structure with its own width and height members, `bmiHeader.biWidth` and `bmiHeader.biHeight`.

Second, there is a rectangle in the `rcSource` member of these structures, and last, there is a rectangle in the `rcTarget` member of these structures.

Assume you have two filters, A and B, and that these filters are connected to each other (A on the left, or upstream, and B on the right, or downstream) with a certain video media type. The buffers that pass between filters A and B have the size (`bmiHeader.biWidth`, `bmiHeader.biHeight`). Filter A should take a portion of its input video determined by `rcSource` and stretch that video to fill the `rcTarget` portion of the buffer. The portion of the input video to use is based on how `rcSource` compares to the (`bmiWidth`, `bmiHeight`) size of the media type that filters A and B originally connected with. If `rcSource` is empty, filter A uses its entire input video. If `rcTarget` is empty, filter A fills the entire output buffer.

For example, assume filter A is receiving video data that is 160 x 120 pixels. Assume also that filter A is connected to filter B with the following media type.

(biWidth, biHeight): 320, 240
rcSource: (0, 0, 0, 0)
rcTarget: (0, 0, 0, 0)

This means that filter A will stretch the video it receives by 2 in both the x and y directions, and fill a 320 x 240 output buffer.

As another example, assume filter A is receiving 160 x 120 video data, and that it is connected to filter B with the following media type.

(biWidth, biHeight): 320, 240
rcSource: (0, 0, 160, 240)
rcTarget: (0, 0, 0, 0)

The `rcSource` member is relative to the connected buffer size of 320, 240. Because the specified `rcSource` (0, 0, 160, 240) is the left half of the buffer, filter A will take the left half of its input video, or the (0, 0, 80, 120) portion, and stretch the video to a size of (320, 240) (by 4 in the x direction, and by 2 in the y direction) and filling the 320 x 240 output buffer.

Now assume that filter A calls `CBaseAllocator::GetBuffer`, and the media sample returned has a media type attached to it, signifying that filter B wants filter A to provide a different size or kind of video than it has previously been providing. Assume the new media type is:

(biWidth, biHeight): 640, 480
rcSource: (0, 0, 160, 120)
rcTarget: (0, 0, 80, 60)

This means that the media sample has a buffer that is 640 x 480 in size. The `rcSource` member is relative to the original connected media type (320, 240) not to the new media type of (640, 480), so `rcSource` specifies that the top-left corner (25%) of the input video to be used. This portion of the input video is placed in the top-left (80, 60) pixels of the 640 x 480 output buffer, as specified by `rcTarget` of (0, 0, 80, 60).
Because filter A is receiving 160 x 120 video, the top-left corner of the input video is an (80, 60) piece, the same size of the output bitmap, and no stretching is required. Filter A will place no data in the other pixels of the output buffer, and will leave those bits untouched. The rcSource member is bounded by the biWidth and biHeight of the original connected media type between filters A and B, and rcTarget is bounded by the new biWidth and biHeight of the media sample. In the preceding example, rcSource could not go outside the boundaries of (0, 0, 320, 240) and rcTarget could not go outside the boundaries of (0, 0, 640, 480).

X. Writing Capture Filters

Writing audio and video capture filters is no longer recommended. Instead, DirectShow provides automatic support for audio and video capture devices, using wrapper filters and the System Device Enumerator. For more information about implementing a device driver, refer to the Windows Driver Development Kit (DDK) documentation. This section is intended only for developers who need to capture some kind of custom data from an unusual hardware device.

This article contains the following sections:
1. Pin Requirements for Capture Filters
2. Implementing a Preview Pin (Optional)
3. Producing Data in a Capture Filter

X.1 Pin Requirements for Capture Filters

Pin Name
You can give a pin any name. If the pin name begins with the tilde (~) character, the Filter Graph Manager does not automatically render that pin when an application calls IGraphBuilder::RenderFile. For example, if the filter has a capture pin and a preview pin, you might name them ~Capture and Preview, respectively. If an application renders that filter in a graph, the preview pin will connect to its default renderer, and nothing will connect to the capture pin, which is a reasonable default behavior. This can also apply to pins that deliver informational data that is not meant to be rendered, or pins that need custom properties set. Note that pins with the tilde (~) prefix can still be connected manually by the application.

Pin Category
A capture filter always has a capture pin, and might have a preview pin. Some capture filters might have other output pins, to deliver other types of data, such as control information. Each output pin must expose the IKsPropertySet interface. Applications use this interface to determine the pin category. The pin typically returns either PIN_CATEGORY_CAPTURE or PIN_CATEGORY_PREVIEW. For more information, see Pin Property Set.

The following example shows how to implement IKsPropertySet to return the pin category on a capture pin:

```c++
// Set: Cannot set any properties.
HRESULT CMyCapturePin::Set(REFGUID guidPropSet, DWORD dwID, void *pInstanceData, DWORD cbInstanceData, void *pPropData, DWORD cbPropData)
{
    return E_NOTIMPL;
}
// Get: Return the pin category (our only property).
HRESULT CMyCapturePin::Get(
   REFGUID guidPropSet,   // Which property set.
    DWORD dwPropID,        // Which property in that set.
    void *pInstanceData,   // Instance data (ignore).
    DWORD cbInstanceData,  // Size of the instance data (ignore).
    void *pPropData,       // Buffer to receive the property data.
    DWORD cbPropData,      // Size of the buffer.
    DWORD *pcbReturned     // Return the size of the property.
)
{
    if (guidPropSet != AMPROPSETID_Pin)
        return E_PROP_SET_UNSUPPORTED;
    if (dwPropID != AMPROPERTY_PIN_CATEGORY)
        return E_PROP_ID_UNSUPPORTED;
    if (pPropData == NULL && pcbReturned == NULL)
        return E_POINTER;
    if (pcbReturned)
        *pcbReturned = sizeof(GUID);
    if (pPropData == NULL)  // Caller just wants to know the size.
        return S_OK;
    if (cbPropData < sizeof(GUID)) // The buffer is too small.
        return E_UNEXPECTED;
    *(GUID *)pPropData = PIN_CATEGORY_CAPTURE;
    return S_OK;
}
// QuerySupported: Query whether the pin supports the specified property.
HRESULT CMyCapturePin::QuerySupported(REFGUID guidPropSet, DWORD dwPropID, DWORD *pTypeSupport)
{
    if (guidPropSet != AMPROPSETID_Pin)
        return E_PROP_SET_UNSUPPORTED;
    if (dwPropID != AMPROPERTY_PIN_CATEGORY)
        return E_PROP_ID_UNSUPPORTED;
    if (pTypeSupport)
        // We support getting this property, but not setting it.
        *pTypeSupport = KSPROPERTY_SUPPORT_GET;
    return S_OK;
}
```

X.2 Implementing a Preview Pin (Optional)

If your filter has a preview pin, the preview pin must send a copy of the data delivered by the capture pin. Only send data from the preview pin when doing so will not cause the capture pin to drop frames. The capture pin always has priority over the preview pin.

The capture pin and the preview pin must send data with the same format. Therefore, they must connect using identical media types. If the capture pin connects first, the preview pin should offer the same media type, and reject any others types. If the
preview pin connects first, and the capture pin connects with a different media type, the preview pin should reconnect using the new media type. If the filter downstream from the preview pin rejects the new type, the capture pin should also reject the type. Use the IPin::QueryAccept method to query the filter downstream from the preview pin, and use the IFilterGraph::Reconnect method to reconnect the pin. These rules also apply if the Filter Graph Manager reconnects the capture pin.

The following example shows an outline of this process:

```cpp
// Override CBasePin::CheckMediaType.
CCapturePin::CheckMediaType(CMediaType *pmt)
{
    if (m_pMyPreviewPin->IsConnected())
    {
        // The preview pin is already connected, so query the pin it is connected to. If the other pin rejects it, so do we.
        hr = m_pMyPreviewPin->GetConnected()->QueryAccept(pmt);
        if (hr != S_OK)
        {
            // The preview pin cannot reconnect with this media type.
            return E_INVALIDARG;
        }
        // The preview pin will reconnect when SetMediaType is called.
    }
    // Decide whether the capture pin accepts the format.
    BOOL fAcceptThisType = ...  // (Not shown.)
    return (fAcceptThisType? S_OK : E_FAIL);
}

// Override CBasePin::SetMediaType.
CCapturePin::SetMediaType(CMediaType *pmt);
{
    if (m_pMyPreviewPin->IsConnected())
    {
        // The preview pin is already connected, so it must reconnect.
        if (m_pMyPreviewPin->GetConnected()->QueryAccept(pmt) == S_OK)
        {
            // The downstream pin will accept the new type, so it's safe to reconnect.
            m_pFilter->m_pGraph->Reconnect(m_pMyPreviewPin);
        }
        else
        {
            return VFW_E_INVALIDMEDIATYPE;
        }
    }
    // Now do anything that the capture pin needs to set the type.
    hr = MyInternalSetMediaType(pmt);
    // And finally, call the base-class method.
    return CBasePin::SetMediaType(pmt);
}

CPreviewPin::CheckMediaType(CMediaType *pmt)
{
    if (m_pMyCapturePin->IsConnected())
    {
        // The preview pin must connect with the same type.
        CMediaType cmt = m_pMyCapturePin->m_mt;
        return (*pmt == cmt ? S_OK : VFW_E_INVALIDMEDIATYPE);
    }
    // Decide whether the preview pin accepts the format. You can use your knowledge of which types the capture pin will accept. Regardless, when the capture pin connects, the preview pin will reconnect.
    return (fAcceptThisType? S_OK : E_FAIL);
}
```

See Also

- How Filters Connect

X.3 Producing Data in a Capture Filter

State Changes

A capture filter should produce data only when the filter is running. Do not send data from your pins when the filter is paused. Also, return VFW_S_CANT_CUE from the CBaseFilter::GetState method when the filter is paused. This return code informs the Filter Graph Manager that it should not wait for any data from your filter while the filter is paused. For more information, see Filter States.

The following code shows how to implement the GetState method:

```cpp
CMyVidcapfilter::GetState(DWORD dw, FILTER_STATE *pState)
{
    CheckPointer(pState, E_POINTER);
    *pState = m_State;
    if (m_State == State_Paused)
    return VFW_S_CANT_CUE;
    else
    return S_OK;
}
```

Controlling Individual Streams

A capture filter's output pins should support the IAMStreamControl interface, so that an application can turn each pin on or off individually. For example, an application can preview without capturing, and then switch to capture mode without rebuilding the filter graph. You can use the CBaseStreamControl class to implement this interface.

Time Stamps

When the filter captures a sample, time stamp the sample with the current stream time. The end time is the start time plus the duration. For example, if the filter is capturing at ten samples per second, and the stream time is 200,000,000 units when the filter captures the sample, the time stamps should be 200000000 and 201000000. (There are 10,000,000 units per second.) Use the IReferenceClock::GetTime method to get the current reference time, and the IMediaSample::SetTime method to set the time stamps.
Time stamps must always increase from one sample to the next. This is true even when the filter pauses. If the filter runs, pauses, and then runs again, the first sample after the pause must have a larger time stamp than the last sample before the pause. Samples from the preview pin should not have time stamps, however. Preview samples always reach the renderer slightly after the time of capture, so the renderer will treat them as late samples. This might cause the renderer to drop samples, in a futile attempt to catch up. Note that the IAMStreamControl interface requires the pin to keep track of sample times. For a preview pin, you might need to modify the implementation so that it does not rely on time stamps. Depending on the data you are capturing, it might be appropriate to set the media time on the samples. For more information, see Time and Clocks in DirectShow.

XI. 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. 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 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.

This article contains the following topics:
1. Video Capabilities
2. Audio Capabilities
3. Reconnecting Your Input to Ensure Specific Output Types

XI.1 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. For audio-related examples of GetStreamCaps, see Audio Capabilities. Suppose your capture card supports JPEG format at all resolutions between 160 x 120 pixels and 320 x 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 your card also supports the size 640 x 480. The following illustrates this single resolution and the above range of resolutions (all sizes between 160 x 120 pixels and 320 x 240 pixels).

Also, suppose it supports 24-bit color RGB format at resolutions between 160 x 120 and 320 x 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 x 120
- 168 x 120
- 168 x 128
- 176 x 128
- 176 x 136
- ... additional resolutions ...
- 312 x 232
- 320 x 240

Use GetStreamCaps to expose these color format and dimension capabilities by offering a media type of 320 x 240 JPEG (if that is your default or preferred size) coupled with minimum capabilities of 160 x 120, maximum capabilities of 320 x 240, and a granularity of 1. The next pair you expose by using GetStreamCaps is a media type of 640 x 480 JPEG coupled with a minimum of 640 x 480 and a maximum of 640 x 480 and a granularity of 1. The third pair includes a media type of 320 x 240, 24-bit RGB with minimum capabilities of 160 x 120, maximum capabilities of 320 x 240, and a granularity of 8. In this way you can publish almost every format and capability.
XI.3  Reconnecting Your Input to Ensure Specific Output Types

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

A transform filter may not have any preferred output types unless their input pin is connected. In that case, the SetFormat and IAMStreamConfig::GetStreamCaps methods should return VFW_E_NOT_CONNECTED until the input pin is connected. Otherwise, these methods can function as usual.

XI.2  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.

For video-related examples relating to GetStreamCaps, see Video Capabilities. Suppose you support pulse code modulation (PCM) wave format (as represented by the WAVEFORMATEX 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.

XI.2  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.

For video-related examples relating to GetStreamCaps, see Video Capabilities. Suppose you support pulse code modulation (PCM) wave format (as represented by the WAVEFORMATEX 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.
In certain cases it is useful to reconnect pins when you are offering a format on an established connection. For example, suppose that a filter can compress 24-bit RGB video into format X, and that it can compress 8-bit RGB video into format Y. The output pin can do either of the following:

- Always offer both X and Y in GetStreamCaps, and always accept both X and Y in SetFormat.
- Offer and accept just format X if the input type is 24-bit RGB. Offer and accept just format Y if the input type is 8-bit RGB. Fail both methods if the input pin is not connected.

In either case, you will need some reconnecting code that looks like this:

```c++
HRESULT MyOutputPin::CheckMediaType(const CMediaType *pmtOut)
{
    // Fail if the input pin is not connected.
    if (!m_pFilter->m_pInput->IsConnected()) {
        return VFW_E_NOT_CONNECTED;
    }
    // (Not shown: Reject any media types that you know in advance your filter cannot use. Check the major type and subtype GUIDs.)
    // (Not shown: If SetFormat was previously called, check whether pmtOut exactly matches the format that was specified in SetFormat.
    // Return S_OK if they match, or VFW_E_INVALIDMEDIATYPE otherwise.)
    // Now do the normal check for this media type.
    HRESULT hr;
    hr = m_pFilter->CheckTransform(
        &m_pFilter->m_pInput->CurrentMediaType(), // The input type.
        pmtOut  // The proposed output type.
    );
    if (hr == S_OK) {
        // This format is compatible with the current input type.
        return S_OK;
    }
    // This format is not compatible with the current input type.
    // Maybe we can reconnect the input pin with a new input type.
    // Enumerate the upstream filter's preferred output types, and see if one of them will work.
    CMediaType *pmtEnum = NULL;
    BOOL fFound = FALSE;
    IEnumMediaTypes *pEnum;
    hr = m_pFilter->m_pInput->GetConnected()->EnumMediaTypes(&pEnum);
    if (hr != S_OK) {
        return E_FAIL;
    }
    while (hr = pEnum->Next(1, (AM_MEDIA_TYPE **)&pmtEnum, NULL), hr == S_OK) {
        // Check this input type against the proposed output type.
        hr = m_pFilter->CheckTransform(pmtEnum, pmtOut);
        if (hr != S_OK)
            continue; // Try the next one.
        // This input type is a possible candidate. But, we have to make sure that the upstream filter can switch to this type.
        hr = m_pFilter->m_pInput->GetConnected()->QueryAccept(pmtEnum);
        if (hr != S_OK) {
            // The upstream filter will not switch to this type.
            DeleteMediaType(pmtEnum);
            continue; // Try the next one.
        } else {
            fFound = TRUE;
            DeleteMediaType(pmtEnum);
            break;
        }
    }
    if (fFound) {
        // This output type is OK, but if we are asked to use it, we will need to reconnect our input pin. (See SetFormat, below.)
        return S_OK;
    } else {
        return VFW_E_INVALIDMEDIATYPE;
    }
}
```

```c++
HRESULT MyOutputPin::SetFormat(AM_MEDIA_TYPE *pmt)
{
    CheckPointer(pmt, E_POINTER);
    HRESULT hr;
    // Hold the filter state lock, to make sure that streaming isn't in the middle of starting or stopping:
    CAutoLock cObjectLock(m_pFilter->m_csFilter);
    // Cannot set the format unless the filter is stopped.
    if (m_pFilter->m_State != StateStopped)
        return VFW_E_NOT_STOPPED;
    // The set of possible output formats depends on the input format, so if the input pin is not connected, return a failure code.
    if (!m_pFilter->m_pInput->IsConnected())
        return VFW_E_NOT_CONNECTED;
    // If the pin is already using this format, there's nothing to do.
    if (IsConnected() && CurrentMediaType() == *pmt)
        return S_OK;
    // The set of possible output formats depends on the input format...
    // But we know that the pin is not connected, so we can use the default behavior.
    if (pmt->cbFormat != sizeof(AM_MEDIA_TYPE))
        return E_INVALIDARG;
    hr = m_pFilter->m_pInput->GetConnected()->EnumMediaTypes(&pEnum);
    if (hr != S_OK) {
        return E_FAIL;
    }
    while (hr = pEnum->Next(1, (AM_MEDIA_TYPE **)&pmtEnum, NULL), hr == S_OK) {
        // Check this input type against the proposed output type.
        hr = m_pFilter->CheckTransform(pmtEnum, pmtOut);
        if (hr != S_OK)
            continue; // Try the next one.
    }
}
```
Extrait de la documentation DirectShow

// Override CTransformFilter::SetMediaType to reconnect the input pin.
// This method is called immediately after the media type is set on a pin.
HRESULT MyFilter::SetMediaType(
PIN_DIRECTION direction,
const CMediaType *pmt)
{
    HRESULT hr;
    if (direction == PINDIR_OUTPUT)
    {
        // Before we set the output type, we might need to reconnect
        // the input pin with a new type.
        if (m_pInput && m_pInput->IsConnected())
        {
            // Check if the current input type is compatible.
            hr = CheckTransform(&m_pInput->CurrentMediaType(),
                                 &m_pOutput->CurrentMediaType());
            if (SUCCEEDED(hr))
            {
                return S_OK;
            }
        }
        // Otherwise, we need to reconnect the input pin.
        hr = m_pGraph->Reconnect(m_pInput);
        return hr;
    }
    return S_OK;
}

XII Registering a Custom File Type

This article describes how the Filter Graph Manager locates a source filter, given a file
name. You can use this mechanism to register your own custom file types. Once the
file type is registered, DirectShow will automatically load the correct source filter
whenever an application calls IGraphBuilder::RenderFile or
IGraphBuilder::AddSourceFilter.

To locate a source filter from a given file name, the Filter Graph Manager attempts to
do the following, in order:

1. Match the protocol, if any.
2. Match the file extension.
3. Match patterns of bytes within the file, called check bytes.

Protocols

Protocol names such as ftp or http are registered under the HKEY_CLASSES_ROOT
key, with the following structure:

HKEY_CLASSES_ROOT
<protocol>
Source Filter = <Source filter CLSID>
Extensions
    <.ext1> = <Source filter CLSID>
    <.ext2> = <Source filter CLSID>

If the file name contains a colon (":'), the Filter Graph Manager attempts to use the
portion before the ":" as a protocol name. For example, if the name is
"myprot://myfile.ext", it searches for a registry key named myprot. If this key exists
and contains a subkey named Extensions, the Filter Graph Manager searches within
that subkey for entries that match the file extension. The value of the key must be a
GUID in string form; for example, "{00000000-0000-0000-0000-000000000000}". If the
Filter Graph Manager cannot match anything within the Extensions subkey, it
looks for a subkey named Source Filter, which must also be a GUID in string form.
If the Filter Graph Manager finds a matching GUID, it uses this as the CLSID of the
source filter, and attempts to load the filter. If it does not find a match, it uses the File
Source (URL) filter, which treats the file name as a URL.

There are two exceptions to this algorithm:

• To exclude driver letters, single-character strings are not considered protocols.
• If the string is "file:" or "file://", it is not treated as a protocol.
**File Extensions**

If there is no protocol in the file name, the Filter Graph Manager looks in the registry for entries with the key `HKEY_CLASSES_ROOT\Media Type\Extensions\.ext`, where `ext` is the file extension. If this key exists, the value `Source Filter` contains the CLSID of the source filter, in string form. Optionally, the key can have values for `Media Type` and `Subtype`, which give the major type and subtype GUIDs.

**Check Bytes**

Some file types can be identified by specific patterns of bits occurring at specific byte offsets in the file. The Filter Graph Manager looks in the registry for keys with the following form:

`HKEY_CLASSES_ROOT\MediaType\{major type\}\{subtype\}`

where `major type` and `subtype` are GUIDs that define the media type for the byte stream. Each key contains one or more subkeys, usually named `1`, `2`, and so on, which define the check bytes; and a subkey named `Source Filter` that gives the CLSID of the source filter, in string form. The check-byte subkeys are strings that contain one or more quads of numbers:

`offset,cb,mask,val`

To match the file, the Filter Graph Manager reads `cb` bytes, starting from byte number `offset`. It then performs a bitwise-AND against the value in `mask`. If the result equals `val`, the file is a match for that quad. The values `mask` and `val` are given in hex. A blank entry for `mask` is treated as a string of 1s of length `cb`. A negative value for `offset` indicates an offset from the end of the file. In order to match the key, the file must match all of the quads in any of the subkeys.

For example, assume the registry contains the following keys under `HKCR\Media Type`:

- `{e436eb83-524f-11ce-9f53-0020af0ba770}`
- `{7364696D-0000-0010-8000-00AA00389B71}`
  - `0, 4, , ABCD1234, -4, 4, , ABAB00AB`

Also, there can be multiple entries listed under a single media type. A match to any of them is sufficient. This scheme allows for a set of alternative masks; for instance, `.wav` files that might or might not have a RIFF header.

**Note** This scheme is similar to the one used by the Microsoft® Win32® function `GetClassFile`.

**Loading the Source Filter**

Assuming that the Filter Graph Manager finds a matching source filter for the file, it adds that filter to the graph, queries the filter for the `IFileSourceFilter` interface, and calls `IFileSourceFilter::Load`. The arguments to the `Load` method are the file name and the media type, as determined from the registry.

If the Filter Graph Manager cannot find anything from the registry, it defaults to using the Async File Source filter. In that case, it sets the media type to `MEDIATYPE_Stream`, `MEDIASUBTYPE_None`.

---

**XIII. Creating a Filter Property Page**

This section describes how to create a property page for a custom DirectShow filter, using the `CBasePropertyPage` class. The example code in this section shows all the steps needed to create a property page. The example shows a property page for a hypothetical video effect filter that supports a saturation property. The property page has a slider, which the user can move to adjust the filter's saturation level.

This section contains the following topics:

1. Step 1. Define a Mechanism for Setting the Property
2. Step 2. Implement `ISpecifyPropertyPages`
4. Step 4. Create the Property Page
5. Step 5. Store a Pointer to the Filter
6. Step 6. Initialize the Dialog
7. Step 7. Handle Window Messages
8. Step 8. Apply Property Changes
9. Step 9. Disconnect the Property Page
10. Step 10. Support COM Registration