



DirectShow Decoder Filter Implementation Guide for Windows Embedded Compact 7

Writer: Sohel Islam

Technical Reviewer: Damon Barry

Published: December 2011

Applies To: Windows Embedded Compact 7

Abstract

This paper is a guide for developers who are implementing a DirectShow filter to work with their hardware decoder. This paper provides specific information for the MPEG-1, MPEG-2, MPEG-4, and ASF (WMA and WMV) formats, but the general information applies to all formats. It includes:

- An overview of DirectShow terms and concepts
- Implementation of a decoder filter class
- Implementation of pin classes to connect to other filters
- Allocation of a buffer between the demultiplexer and the decoder filter
- A list of codecs supported by Windows Embedded Compact 7
- Design considerations and format-specific notes

Contents

Introduction	3
DirectShow Overview	3
DirectShow Terms	3
Playback Filter Graph.....	4
Registry Entries.....	6
DirectShow Codecs Supported by Windows Embedded Compact 7	8
Implementation Basics	8
Implementation Overview	8
Step 1: Implement the Decoder Filter Class.....	9
Step 2: Implement the Pin Classes to Connect the Filters	10
Create the Pins to Connect the Filters.....	11
Set up the Decoder Pin to Receive Samples	12
Step 3: Allocate a Buffer Between the Demultiplexer and the Decoder	13
Step 4: Specify the Type of Media Stream.....	13
MPEG-1 Media Types.....	14
MPEG-2 Media Types.....	15
MPEG-4 Media Types.....	15
ASF Media Types	16
Additional Design Considerations	17
Threading Model.....	17
Application and Streaming Threads.....	17
Starvation Avoidance	18
Queue Depth	18
Thread Priority	19
Bandwidth.....	19
Seeking, Passing Through, and Flushing.....	19
Scatter-Gather Technique	21
MPEG-4-Specific Notes	21
Conclusion	22
Additional Resources	22

Introduction

Windows Embedded Compact 7 supports an implementation of Microsoft DirectShow technology, which is an architecture that is used to play back, capture, and transform multimedia content. Embedded devices typically have hardware decoders for audio and video. Because the hardware for the device is specific to the embedded platform, developers need to implement a DirectShow filter to enable DirectShow to work with their particular platform.

This article covers some specifics of MPEG-1, MPEG-2, MPEG-4 and ASF (also known as WMA and WMV) formats, but you can apply the general guidelines to other formats also.

After a review of concepts and terminology, the basic steps to implement a decoder filter are described. Step 1 provides instructions on how to implement the decoder filter class. Step 2 covers how to implement the pin classes that connect your decoder filter to other filters. In Step 3, you allocate a buffer between the demultiplexer and the decoder, and in Step 4, you specify the type of media stream. For the quickest implementation of a DirectShow filter, we recommend that you use the DirectShow base classes provided with Windows Embedded Compact 7. As you walk through the implementation steps, you should refer to general documentation on implementing DirectShow filters as needed.

DirectShow Overview

This section contains a review of terminology and the following components.

- A playback filter graph
- Registry entries
- DirectShow codecs supported by Windows Embedded Compact 7

DirectShow Terms

Some common DirectShow terms are defined below.

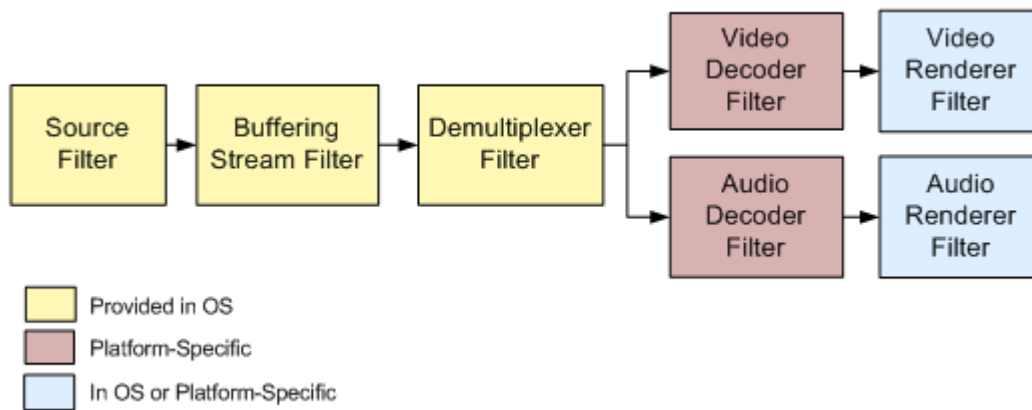
- **Filter:** A DirectShow component that performs a fundamental operation such as playback, capture, or a type of transform such as decompressing a compressed media stream.
- **Filter graph:** A set of connected DirectShow filters that, together, perform a more complex multimedia task.
- **Filter graph manager (FGM):** The DirectShow engine that sets up, tears down, and manages the filter graphs. The FGM makes calls to the filters to indicate whether they should pause, run, stop, and so on.
- **Playback pipeline:** A playback filter graph that is currently running.
- **Media:** Audio, video, or both audio and video data, compressed or uncompressed, that is appropriate for the context.

- **Samples:** Generic term for audio samples and video frames. The term “frame” is also used in the context of video decoding.
- **Container format:** Defines how the media is stored in a file or stream (or some other type of container) and is typically identified by the file extension, such as MP4.
- **Elementary streams:** The media content within a container. An elementary stream is also referred to as the payload. A container may contain multiple elementary streams.
- **Encoding format:** The compression format of an elementary stream within the container. Occasionally, the codec type is used to refer to the encoding format.
- **Demultiplexer:** The software component that separates the audio and video data from the interleaved streams of data found in the containers, timestamps them, and pushes them out through different output connections.
- **Decoder:** A filter that decompresses the compressed format of the media. A video decoder receives compressed video samples and produces uncompressed video frames. Similarly, an audio decoder receives compressed audio and produces uncompressed audio.
- **Renderer:** A filter that plays the media data through an appropriate device. A video renderer displays the uncompressed video frames it receives from the video decoder. An audio renderer plays the audio samples through the audio device.
- **Pin:** A unidirectional connection point that connects one filter to another filter.
- **Upstream filter:** A filter that produces data.
- **Downstream filter:** A filter that consumes data.

Playback Filter Graph

A playback filter graph has a source filter, a demultiplexer filter, one or more decoder filters, and one or more renderer filters. For certain formats, Windows Embedded Compact 7 has a new buffering filter, the Buffering Stream Filter, that sits between the source filter and the demultiplexer and manages pools of [IMediaSample](http://go.microsoft.com/fwlink/?LinkId=215025) (http://go.microsoft.com/fwlink/?LinkId=215025) buffers so that other filters downstream can work with minimal buffer management. With the Buffering Stream Filter in use, a typical filter graph looks like the following:

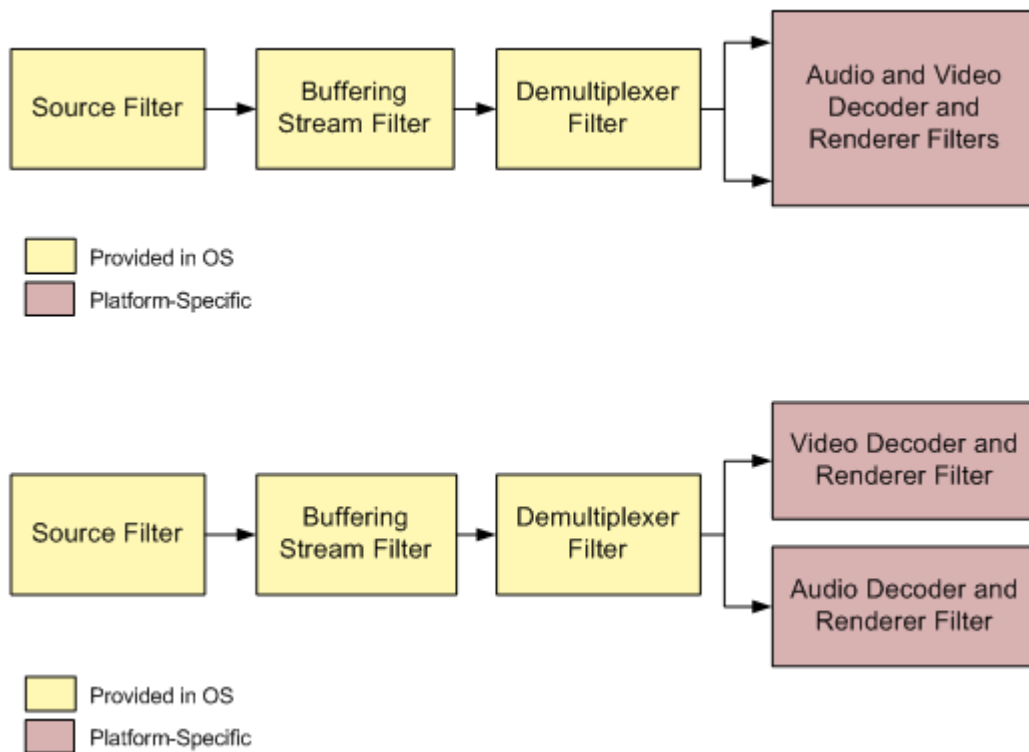
Figure 1: DirectShow Filter Graph



For a given platform, you typically implement the audio and video decoder filters to take advantage of any available decoding hardware on the platform. You might also implement the renderer filters if the platform has specific requirements (for example, requires writing to predetermined video memory for faster processing). Alternately, you can use the software renderers that are supplied by the OS.

Note that you do not have to implement the audio and video decoders and renderers in separate software components. A software module can consist of one or more binaries containing all of these parts. For example, assume that you are implementing both the decoders and the renderers. In this situation, you can create a component that combines the audio decoder and renderer and another component that combines the video decoder and renderer. Alternatively, you can create one single binary that contains all of these parts. Depending on your implementation, the filter graph could appear as one of the configurations shown in the figure below:

Figure 2: Alternate Decoder and Renderer Filter Configurations



Note also that some media formats do not require a buffering stream filter, so you may not have one in your filter graph. If the buffering stream filter is not used, the source filter connects directly to the demultiplexer filter.

Registry Entries

Every filter in DirectShow must have a GUID. A filter also must have registry entries that describe its input and output types so that DirectShow can discover the filter and make use of it. The following example code shows a set of registry entries for a typical decoder. In your own code, you would replace the placeholder text in the angle brackets with appropriate values for the decoder. All input and output types and subtypes are GUIDs, and all GUIDs must be specified in a {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx} form where each x represents a hexadecimal digit. For audio and video, the major types are **MEDIATYPE_Audio** and **MEDIATYPE_Video**, respectively. The input subtypes depend on the encoding format accepted by the decoder and the output subtype depends on the format of the decoded (uncompressed) data. In Windows Embedded Compact 7, the GUIDs for the major types and common subtypes can be found in %_WINCEROOT%\Public\Directx\SDK\Inc\uuids.h.

```
[HKEY_CLASSES_ROOT\Filter\<DecoderGUID>]
```

```
@="<DecoderName>"
```

```
[HKEY_CLASSES_ROOT\CLSID\<DecoderGUID>]
```

```
@="<DecoderName>"
```

```
"Merit"=dword:00600000
```

```
[HKEY_CLASSES_ROOT\CLSID\<Decoder GUID>\InprocServer32]
```

```
@="<DecoderBinary.dll>"
```

```
"ThreadingModel"="Both"
```

```
[HKEY_CLASSES_ROOT\CLSID\<Decoder GUID>\Pins\Input]
```

```
"Direction"=dword:00000000
```

```
"IsRendered"=dword:00000000
```

```
"AllowedZero"=dword:00000000
```

```
"AllowedMany"=dword:00000000
```

```
"ConnectsToPin"="Output"
```

```
[HKEY_CLASSES_ROOT\CLSID\<DecoderGUID>\Pins\Input\Types\<MajorType>\<Subtype1>]
```

```
[HKEY_CLASSES_ROOT\CLSID\<DecoderGUID>\Pins\Input\Types\<MajorType>\<Subtype2>]
```

...

```
[HKEY_CLASSES_ROOT\CLSID\<DecoderGUID>\Pins\Output]
```

```
"Direction"=dword:00000001
```

```
"IsRendered"=dword:00000000
```

```
"AllowedZero"=dword:00000000
```

```
"AllowedMany"=dword:00000000
```

```
"ConnectsToPin"="Input"
```

```
[HKEY_CLASSES_ROOT\CLSID\<DecoderGUID>\Pins\Output\Types\<MajorType>\<Subtype1>]
```

```
[HKEY_CLASSES_ROOT\CLSID\<DecoderGUID>\Pins\Output\Types\<MajorType>\<Subtype2>]
```

...

DirectShow Codecs Supported by Windows Embedded Compact 7

The following table summarizes the encoding formats supported by Windows Embedded Compact 7 for each container format.

Table 1: Supported DirectShow Codecs

Container format	File extension	Supported video coding	Supported audio coding	Buffering Stream Filter support
MPEG-1	.mpg, .mpa, .mp2, .mp3	MPEG-1	MPEG-1 Layers 1, 2, and 3 (MP3)	Yes
MPEG-2 Program Stream	.mpg, .mpeg, .m2p	MPEG-2	AC-3, LPCM, MPEG-2 Layers 1 and 2	Yes
MPEG-2 Transport Stream	.mpg, .mpeg, .m2t, .ts, .tts	MPEG-2, MPEG-4	AAC, AC-3, MPEG-2 Layers 1 and 2	Yes
MPEG-4	.mp4, .m4a, .m4v, .mp4v	MPEG-4 Part 2, H.264	AAC, AC-3	Yes
ASF	.asf, .wma, .wmv	Any video coding found within the container	Any audio coding found within the container	No

Implementation Basics

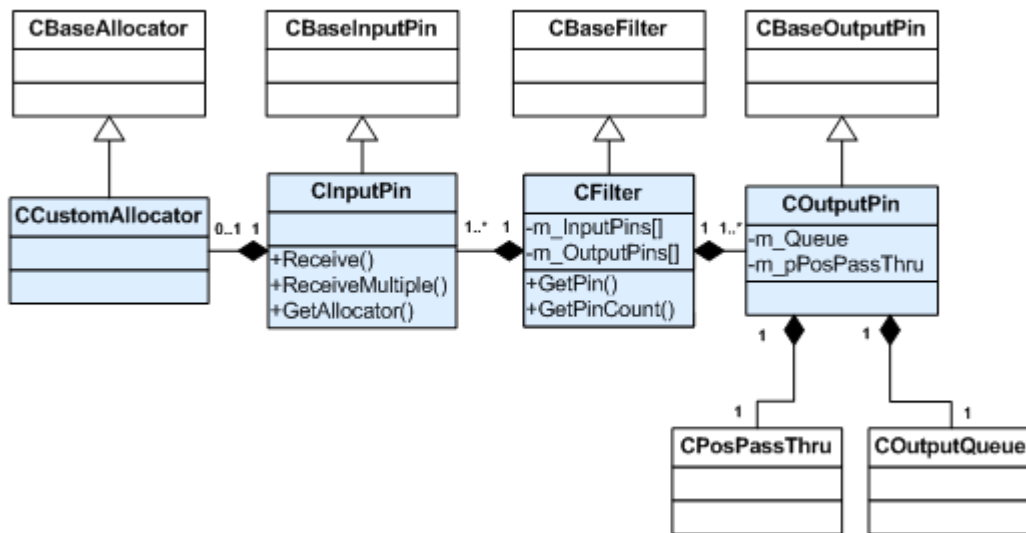
In your DirectShow filter, you must implement a number of interfaces and methods even for a minimal implementation. To reduce the amount of code that you need to write, the DirectShow base classes provide a large amount of common code. We recommend that you use the base classes wherever possible. This article refers to a number of base classes.

Implementation Overview

The main functional parts of a filter include the input pin, output pin, and the logic for the filter itself. However, keep in mind that a particular filter may have no input pin or no output pin, for example, in the case of a source filter and a renderer filter, respectively. Typically, you implement a class for each of these parts while deriving from some corresponding base class.

The following Unified Modeling Language (UML) class diagram shows the basics of a sample implementation of a decoder filter. Note that it is not a comprehensive class diagram; it shows the layout of only some of the classes and interfaces that are discussed later in this article. The shaded boxes contain the classes that the decoder developer implements. In your actual implementation, there will certainly be additional members and methods defined in each class, and possibly additional classes that you need to include for your specific decoder. The unshaded boxes are the base classes that implement a number of necessary interfaces, which are not shown in the diagram.

Figure 3: DirectShow Class Diagram



The steps that follow provide additional details about how to implement the classes shown in Figure 3. The steps are:

- Step 1: Implement the decoder filter class
- Step 2: Implement the pin classes to connect the filters
- Step 3: Allocate a buffer between the demultiplexer and the decoder
- Step 4: Specify the type of media stream

Step 1: Implement the Decoder Filter Class

The demultiplexer and decoder filter are the main filters in this scenario.

Your DirectShow filters must implement the [IBaseFilter](http://go.microsoft.com/fwlink/?LinkId=214790) (<http://go.microsoft.com/fwlink/?LinkId=214790>) interface. The [CBaseFilter](http://go.microsoft.com/fwlink/?LinkId=214791) (<http://go.microsoft.com/fwlink/?LinkId=214791>) base class has already implemented the interface, so derive your filter's main filter implementation class from **CBaseFilter**. Because your filter will create pins for input and output, override the [CBaseFilter::GetPin](http://go.microsoft.com/fwlink/?LinkId=214792) (<http://go.microsoft.com/fwlink/?LinkId=214792>) and [CBaseFilter::GetPinCount](http://go.microsoft.com/fwlink/?LinkId=214793) (<http://go.microsoft.com/fwlink/?LinkId=214793>) methods.

The following example code shows a basic sample filter class for a decoder filter:

```
// Filter class for sample decoder filter
class CFilter : public CBaseFilter
{
public:
    DECLARE_IUNKNOWN;
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);

    // CBaseFilter overridden functions
    CBasePin *GetPin(int n);
    int GetPinCount();

    STDMETHODIMP Stop();
    STDMETHODIMP Pause();
    STDMETHODIMP Run(REFERENCE_TIME tStart);

    CFilter(LPUNKNOWN lpunk, HRESULT *phr);
    ~CFilter();

    // Other methods
    ...

private:
    CInputPin *m_InputPins[];
    COutputPin *m_OutputPins[];
}
```

Step 2: Implement the Pin Classes to Connect the Filters

Next, you need to connect the demultiplexer filter to the decoder filter using pins.

All protocols of connection negotiation in a DirectShow filter graph apply to Windows Embedded Compact 7. For more information, see [How Filters Connect](http://go.microsoft.com/fwlink/?LinkId=214785) (<http://go.microsoft.com/fwlink/?LinkId=214785>) on MSDN.

Create the Pins to Connect the Filters

In DirectShow, all pins must implement the [IPin](http://go.microsoft.com/fwlink/?LinkId=214794) (<http://go.microsoft.com/fwlink/?LinkId=214794>) interface. The demultiplexers derive their output pin classes from [CBaseOutputPin](http://go.microsoft.com/fwlink/?LinkId=214786) (<http://go.microsoft.com/fwlink/?LinkId=214786>) which derives from [CBasePin](http://go.microsoft.com/fwlink/?LinkId=214787&clcid=0x409) (<http://go.microsoft.com/fwlink/?LinkId=214787&clcid=0x409>). Therefore, you also need to adhere to the implementation requirements for those base classes.

Both the input and output pin must override the [CBasePin::Active](http://go.microsoft.com/fwlink/?LinkId=214795) (<http://go.microsoft.com/fwlink/?LinkId=214795>) and [CBasePin::Inactive](http://go.microsoft.com/fwlink/?LinkId=214796) (<http://go.microsoft.com/fwlink/?LinkId=214796>) methods, which are called by the Filter Graph Manager before and after the streaming. These methods allocate and deallocate resources required for streaming.

The following example code shows a basic input pin class:

```
class CInputPin : public CBasePin
{
public:
    DECLARE_IUNKNOWN;

    HRESULT SetMediaType(const CMediaType *pmt);
    HRESULT CheckMediaType(const CMediaType *pMediaType);
    HRESULT GetMediaType(int iPosition, CMediaType *pMediaType);
    HRESULT CompleteConnect(IPin* pPin);
    HRESULT BreakConnect(void);
    HRESULT BeginFlush();
    HRESULT EndFlush();
    HRESULT Active();
    HRESULT Inactive();
    HRESULT ReadBytes(QWORD offset, DWORD size, BYTE *pDestination);

    CInputPin(HRESULT *pHr, CFilter *pParent, LPCWSTR pPinName);
    ~CInputPin();

    ...
}
```

The following example code shows a basic output pin class:

```
class COutputPin : public CBaseOutputPin
{
public:
    DECLARE_IUNKNOWN;

    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv);
    HRESULT CheckMediaType(const CMediaType *pMediaType);
    HRESULT GetMediaType(int iPosition, CMediaType *pMediaType);
    HRESULT CompleteConnect(IPin *pPin);
    HRESULT BreakConnect();
    HRESULT BeginFlush();
    HRESULT EndFlush();
    HRESULT Active();
    HRESULT Inactive();

    COutputPin(HRESULT *pHr, CFilter *pParent, LPCWSTR pPinName);
    ~COutputPin();
    ...
}
```

Set up the Decoder Pin to Receive Samples

The decoder needs to implement the [IMemInputPin](http://go.microsoft.com/fwlink/?LinkId=214788) (http://go.microsoft.com/fwlink/?LinkId=214788) interface on its input pin to connect to the upstream filter. The easiest way is to derive the input pin from the [CBaseInputPin](http://go.microsoft.com/fwlink/?LinkId=214789) (http://go.microsoft.com/fwlink/?LinkId=214789) base class which adds support for **IMemInputPin**.

[IMemInputPin::Receive](http://go.microsoft.com/fwlink/?LinkId=214797) (http://go.microsoft.com/fwlink/?LinkId=214797) and [IMemInputPin::ReceiveMultiple](http://go.microsoft.com/fwlink/?LinkId=214798) (http://go.microsoft.com/fwlink/?LinkId=214798) are the two functions defined to receive samples from upstream. The first is used to receive a single sample per call and the latter is to receive multiple samples. The decoder should override [CBaseInputPin::Receive](http://go.microsoft.com/fwlink/?LinkId=214801) (http://go.microsoft.com/fwlink/?LinkId=214801) to receive samples from the upstream filter. It can also override and implement the [CBaseInputPin::ReceiveMultiple](http://go.microsoft.com/fwlink/?LinkId=214803) (http://go.microsoft.com/fwlink/?LinkId=214803) function if it wants to receive batches of samples. If not overridden, the default implementation will call the **CBaseInputPin::Receive** function multiple times. Each sample is a COM object that exposes [IMediaSample](http://go.microsoft.com/fwlink/?LinkId=215025) (http://go.microsoft.com/fwlink/?LinkId=215025), which is reference counted. So it is important to

release the sample as soon as the decoder has finished processing it so that it can be reused by the pipeline for subsequent use.

Step 3: Allocate a Buffer Between the Demultiplexer and the Decoder

To pass data between the demultiplexer and decoder filter, you need to set up a buffer.

An allocator object is negotiated between the output pin of the demultiplexer and the input pin of the decoder. You use this object to allocate buffers that hold the samples as they are passed downstream to the decoder. Allocators in DirectShow must implement [IMemAllocator](http://go.microsoft.com/fwlink/?LinkId=214834) (<http://go.microsoft.com/fwlink/?LinkId=214834>) interface.

[CBaseInputPin::GetAllocator](http://go.microsoft.com/fwlink/?LinkId=214838) (<http://go.microsoft.com/fwlink/?LinkId=214838>) provides a default allocator of type [CMemAllocator](http://go.microsoft.com/fwlink/?LinkId=214835) (<http://go.microsoft.com/fwlink/?LinkId=214835>). The default allocator works in most cases. However, if the hardware decoder on the device has requirements, such as using a particular region of memory for better performance, the decoder filter must have its own allocator class that implements [IMemInputPin](http://go.microsoft.com/fwlink/?LinkId=214788) (<http://go.microsoft.com/fwlink/?LinkId=214788>) interface. A simple way to implement a custom allocator is to derive it from the [CBaseAllocator](http://go.microsoft.com/fwlink/?LinkId=214836) (<http://go.microsoft.com/fwlink/?LinkId=214836>) base class. In this custom allocator class, the decoder filter must manage the special region of the memory. The decoder filter must also override [IMemInputPin::GetAllocator](http://go.microsoft.com/fwlink/?LinkId=214840) (<http://go.microsoft.com/fwlink/?LinkId=214840>) on its input pin and return an instance of the allocator class.

Step 4: Specify the Type of Media Stream

When you connect DirectShow pins to each other, you must provide information about the type of media that is going to flow from the output pin to the input pin. The type of the media is described through the [AM_MEDIA_TYPE](http://go.microsoft.com/fwlink/?LinkId=214842) (<http://go.microsoft.com/fwlink/?LinkId=214842>) data type, which has the following structure:

```
typedef struct _MediaType{
    GUID majortype;
    GUID subtype;
    BOOL bFixedSizeSamples;
    BOOL bTemporalCompression;
    ULONG lSampleSize;
    GUID formattype;
    IUnknown* pUnk;
    ULONG cbFormat;
    BYTE __RPC_FAR* pbFormat;
} AM_MEDIA_TYPE;
```

Among the fields of **AM_MEDIA_TYPE**, the **majorType** field is almost always **MEDIATYPE_Video** or **MEDIATYPE_Audio**, although some demultiplexers can support other types, such as subtitle. The **subtype** field depends on the encoding or format of the stream. Although there are exceptions, typically, you derive the subtype from a four-character code (called FourCC) that defines the encoding type. For information about creating a GUID from a FourCC, see the [FOURCCMapClass](http://go.microsoft.com/fwlink/?LinkId=214844) (<http://go.microsoft.com/fwlink/?LinkId=214844>) documentation on MSDN. For more information about media types, see [DirectShow Media Types](http://go.microsoft.com/fwlink/?LinkId=214843) (<http://go.microsoft.com/fwlink/?LinkId=214843>) on MSDN.

The **pbFormat** pointer links to another data structure with additional format specific information that you will need for decoding the media. The **cbFormat** field specifies the length of that structure. The **formatType** field identifies the GUID specifying the format type.

The following tables describe the media types and subtypes and the type of the format structures as implemented in Windows Embedded Compact 7. The comments for specific formats may be useful for implementing the decoder. Note that the decoder must check the **formatType** and **cbFormat** fields to ensure the correct type and length before interpreting data pointed to by **pbFormat**. Also, it is common not to populate all the fields of the format structures. You can find the details of each format structure by looking up the corresponding documentation on MSDN.

MPEG-1 Media Types

Table 2 and Table 3 list the MPEG-1 media subtypes for media types **MEDIATYPE_Audio** and **MEDIATYPE_Video**, respectively.

Table 2: MPEG-1 MEDIATYPE_Audio

Media subtype	Format structure	Comment
MEDIASUBTYPE_MPEG1Packet	MPEG1WAVEFORMAT	None
	MPEGLAYER3WAVEFORMAT	None
MEDIASUBTYPE_MPEG1Payload	MPEG1WAVEFORMAT	None
	MPEGLAYER3WAVEFORMAT	None

Table 3: MPEG-1 MEDIATYPE_Video

Media subtype	Format structure	Comment
MEDIASUBTYPE_MPEG1Packet	MPEG1VIDEOINFO	None
MEDIASUBTYPE_MPEG1Payload	MPEG1VIDEOINFO	None

MPEG-2 Media Types

Table 4 and Table 5 list the MPEG-2 media subtypes for media types **MEDIATYPE_Audio** and **MEDIATYPE_Video**, respectively.

Table 4: MPEG-2 MEDIATYPE_Audio

Media subtype	Format structure	Comment
MEDIASUBTYPE_MPEG1AudioPayload	MPEG1WAVEFORMAT	None
MEDIASUBTYPE_MPEG2_AUDIO	WAVEFORMATEX	None
MEDIASUBTYPE_DOLBY_AC3	WAVEFORMATEX	AC3 audio from MPEG-2 container.
MEDIASUBTYPE_DVD_LPCM_AUDIO	WAVEFORMATEX	None
MEDIASUBTYPE_MPEG_ADTS_AAC	WAVEFORMATEX	None

Table 5: MPEG-2 MEDIATYPE_Video

Media subtype	Format structure	Comment
MEDIASUBTYPE_MPEG1Video	MPEG1VIDEOINFO	None
MEDIASUBTYPE_MPEG2_VIDEO	MPEG2VIDEOINFO	None
MEDIASUBTYPE_H264	VIDEOINFOHEADER	H.264 content in MPEG-2 TS container.
MEDIASUBTYPE_WVC1	VIDEOINFOHEADER	None

MPEG-4 Media Types

Table 6 and Table 7 list the MPEG-4 media subtypes for media types **MEDIATYPE_Audio** and **MEDIATYPE_Video**, respectively.

Table 6: MPEG-4 MEDIATYPE_Audio

Media subtype	Format structure	Comment
MEDIASUBTYPE_MPEG_ADTS_AAC	WAVEFORMATEX	AAC audio from MPEG-4 container. Each audio sample is prepended with a 7-byte Audio Data Transport Stream (ADTS)

Media subtype	Format structure	Comment
		frame header.
MEDIASUBTYPE_DOLBY_AC3	WAVEFORMATEX	AC3 audio from MPEG-4 container.

Table 7: MPEG-4 MEDIATYPE_Video

Media subtype	Format structure	Comment
MEDIASUBTYPE_MP4V	MPEG1VIDEOINFO	MPEG-4 Part 2 video from MPEG-4 container.
	VIDEOINFOHEADER2	
MEDIASUBTYPE_h264	VIDEOINFOHEADER2	MPEG-4 Part 10 (also known as H.264 or AVC) video from MPEG-4 container. Video frames have start codes.

ASF Media Types

Table 8 and Table 9 list the Advanced Systems Format (ASF) media subtypes for media types **MEDIATYPE_Audio** and **MEDIATYPE_Video**, respectively.

Table 8: ASF MEDIATYPE_Audio

Media subtype	Format structure	Comment
(Content Dependent)	WAVEFORMATEX	Subtype depends on the stream. Can be any subtype from ASF spec.

Table 9: ASF MEDIATYPE_Video

Media subtype	Format structure	Comment
(Content Dependent)	VIDEOINFOHEADER	Subtype depends on the stream. Can be any subtype from ASF spec.
	VIDEOINFOHEADER2	

Additional Design Considerations

This section contains information on design considerations that you may want to take into account while implementing a DirectShow decoder filter:

- To avoid deadlocks, you need to set up an appropriate **threading model**.
- To avoid jittery playback, you need to ensure **starvation avoidance**.
- To manage data flow, you need to support **seeking, passing through, and flushing**.
- To support data disassembly and reassembly, you can use a **scatter-gather technique**.
- To use MPEG-4 as your encoding format, it is helpful to read some **MPEG-4-specific notes**.

Threading Model

A decoder must process samples on a separate thread to enable asynchronous calls from the Filter Graph Manager to occur on the main thread. Even while busy processing samples, the decoder must be able to respond to calls from the Filter Graph Manager; otherwise, deadlocks can occur. In DirectShow, the application thread is the thread on which the Filter Graph Manager calls are made. The streaming threads are the threads that process and pass the media samples downstream. Because calls from the Filter Graph Manager can occur any time, the application thread and the streaming threads must be synchronized. The synchronization mechanism is described below.

Application and Streaming Threads

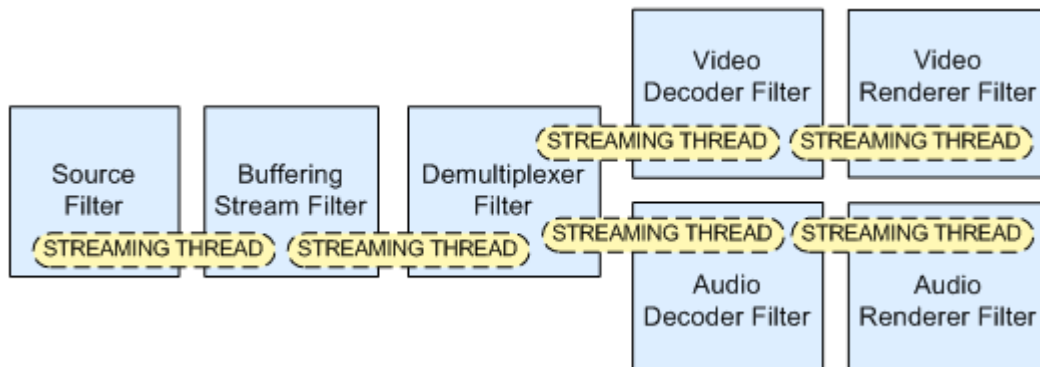
DirectShow base classes already define and use some synchronization mechanisms (some critical section objects and related classes) to ensure mutual exclusivity between the application and the streaming threads. Your code can use the same mechanisms wherever needed. For more information, see [The Streaming and Application Threads](http://go.microsoft.com/fwlink/?LinkId=215009) (<http://go.microsoft.com/fwlink/?LinkId=215009>) on MSDN.

Demultiplexers use the [COutputQueue](http://go.microsoft.com/fwlink/?LinkId=215010) (<http://go.microsoft.com/fwlink/?LinkId=215010>) class for each output pin to manage a queue and a streaming thread to push out samples. A decoder can therefore expect to receive a call to its **IMemInputPin::Receive** or **IMemInputPin::ReceiveMultiple** method from that thread. Because this thread is solely for streaming and is separate from the application thread where the calls from the Filter Graph Manager occur, decoders do not need to create a different thread. It may seem intuitive to handle the decoding asynchronously in a different thread because the operation is CPU-intensive, but by using **COutputQueue** in the demultiplexer, you free the upstream thread that fetches the next sample. It is acceptable to keep the caller thread busy during decoding.

However, to push out the decoded samples, the decoder must use a separate thread. Instead of implementing all the code to maintain a queue for holding the decoded frames and the associated thread, you can use the same **COutputQueue** base class in the decoder to easily handle the job.

Figure 4 shows the threads in a typical filter graph. All of the threads shown in the figure are separate threads. Each thread is shown to cross filter boundaries to represent the function calls that are made from one filter to functions in the next filter in the context of that same thread.

Figure 4: Threads used in a Filter Graph



As mentioned earlier, some platforms may have slightly different componentization for the decoders and renderers. The final stage in Figure 4 above will vary accordingly. However, despite the different componentization, there must still be separate streaming threads for decoding and rendering because the decoder's streaming thread continues to run as long as samples come in without any regard to the timestamp, while the renderer only renders when the timestamp on the sample becomes current.

Starvation Avoidance

Starvation refers to a lack of data to process in any part of the playback pipeline. When starvation occurs, an event is sent to the Filter Graph Manager that pauses the playback so that more data can be buffered and playback can resume. This pause results in jittery playback and a bad user experience. A number of conditions can cause starvation at different stages within the playback pipeline.

Queue Depth

A common tendency during implementation of a DirectShow filter is to keep a sizable working queue to hold samples in case the incoming sample rate cannot keep up with the playback rate. However, in practice, an upstream filter (either the buffering stream filter, if present, or the source filter) can buffer enough samples to ensure smooth playback; therefore, it is better for the decoder filter to maintain the smallest possible sample queue. Otherwise, multiple filters may compete to buffer the samples. This competition can cause starvation by queuing up too many samples in the decoder while the upstream buffer keeps waiting for a free spot to fill.

We recommend that you keep the input queue to a minimum size. For example, a simple audio decoder should keep a queue of only two samples. While one sample is being decoded, the queue is free to receive an incoming sample.

As mentioned earlier, the sample that has been decoded should be released as soon as possible. However, some decoders need to hold on to a few additional samples because of the encoding format of the media. For example, a video decoder might use a Group of Pictures (GOP) method in which the decoding of a sample depends on a previous sample. In another scenario, the sample might be only a part of the full frame. In that case, the decoder must collect all the samples that belong to the frame before it decodes the whole frame. (For more information, see the discussion on scatter-gather techniques below.) In general, ensure that the decoder holds only the minimum number of samples required.

Thread Priority

If the upstream thread that is responsible for buffering does not get enough CPU cycles, it can cause starvation. In other words, if the downstream filters consume data faster than the upstream thread can buffer, it causes the buffer to run out of samples and cause starvation. Therefore, it is important that you do not run the streaming threads in the decoders at more than normal priority.

Bandwidth

If the source filter has too little bandwidth to read the file or receive the stream, it is obvious that starvation will inevitably occur. For example, starvation can occur when a device attempts to play a high bitrate file over a relatively slow network connection. In this situation, there is not much that the decoder can do to avoid starvation.

When starvation occurs, the buffering component in the pipeline sends an **EC_STARVATION** notification to the Filter Graph Manager and the Filter Graph Manager pauses the playback to let the buffers fill up so the user experiences a longer pause instead of frequent shorter glitches. However, to ensure a good user experience, the bitrate of the media must be lower than the available bandwidth at the source.

Seeking, Passing Through, and Flushing

Playback filter graphs handle seek requests from the Filter Graph Manager through the interfaces [IMediaSeeking](http://go.microsoft.com/fwlink/?LinkId=215017) (http://go.microsoft.com/fwlink/?LinkId=215017) or [IMediaPosition](http://go.microsoft.com/fwlink/?LinkId=215018) (http://go.microsoft.com/fwlink/?LinkId=215018) (for older applications). These interfaces are queried on each filter's output pin. Each filter in the filter graph needs to either handle the call or pass it on to the upstream filter's output pin. A decoder does not handle these calls, but it must be able to pass these calls upstream. DirectShow provides the [CPosPassThru](http://go.microsoft.com/fwlink/?LinkId=215019) (http://go.microsoft.com/fwlink/?LinkId=215019) base class for this task.

To implement **CPosPassThru**, the decoder merely creates an instance of the class, passing the pointer to its input pin (which is connected to the upstream filter's output pin), and forwards the [QueryInterface](http://go.microsoft.com/fwlink/?LinkId=215023) (http://go.microsoft.com/fwlink/?LinkId=215023) call for **IID_IMediaSeeking** or **IID_IMediaPosition** to its output pin to the instance of **CPosPassThru**, as shown in the following example code.

```
// Sample code showing use of CPosPassThru
```

```
// Assuming that m_pPosPassThru is a member of type CPosPassThru *
// m_pPosPassThru needs to be deleted in COutputPin d-tor

STDMETHODIMP COutputPin::NonDelegatingQueryInterface(REFIID riid, void **ppv)
{
    HRESULT hr = S_OK;
    {
        // if the CPosPassThru instance has not been created, create it now
        if (!m_pPosPassThru)
        {
            HRESULT hr = S_OK;
            m_pPosPassThru = new CPosPassThru(this, &hr, GetPin(0));
            if (!m_pPosPassThru)
            {
                return E_OUTOFMEMORY;
            }
            else if (FAILED(hr))
            {
                delete m_pPosPassThru;
                m_pPosPassThru = NULL;
                return hr;
            }
        }

        return m_pPosPassThru->NonDelegatingQueryInterface(riid, ppv);
    }
    else
    {
        // Other interfaces (not shown).
    }
}
```

Every decoder must also respond to flush calls. Flushing refers to discarding all streaming data in the pipeline and entering a state where incoming samples are not accepted. Flushing is necessary to support seeking. Flushing begins and ends with calls to [IPin::BeginFlush](http://go.microsoft.com/fwlink/?LinkId=215015) (<http://go.microsoft.com/fwlink/?LinkId=215015>) and [IPin::EndFlush](http://go.microsoft.com/fwlink/?LinkId=215016) (<http://go.microsoft.com/fwlink/?LinkId=215016>) on the input pin. **CBaseInputPin** already has the necessary code to set the **m_bFlushing** flag. The decoder must override at least **IPin::BeginFlush** to discard the samples in progress. It also needs to check for the flag within its **IMemInputPin::Receive** or **IMemInputPin::ReceiveMultiple** method and discard the samples while the flag is set. For details, see [CBaseInputPin::BeginFlush](http://go.microsoft.com/fwlink/?LinkId=215014) (<http://go.microsoft.com/fwlink/?LinkId=215014>).

Scatter-Gather Technique

Scatter-Gather refers to a technique of breaking a large sample into multiple parts and sending the parts separately (scatter operation). On the receiving end, the decoder filter merges the parts to recreate the large sample (gather operation). Theoretically, an upstream filter can send the parts out of order, in which case the recipient must reorder them.

Incoming frames in the filter graph may be spread across multiple fixed-size [IMediaSample](http://go.microsoft.com/fwlink/?LinkId=215025) (<http://go.microsoft.com/fwlink/?LinkId=215025>) buffers. Most demultiplexers in Windows Embedded Compact 7 ensure that a whole frame is contained within one sample before handing it off to the decoder. When you use one of these demultiplexers, the decoder does not need to handle scatter-gather.

However, the MPEG-2 demultiplexer does require a scatter-gather implementation. Because of the nature of MPEG-2 video encoding, the full frame size may not be known at the container level without actually parsing the sample payload. Instead of parsing the encoded bytes, the MPEG-2 demultiplexer just sends fixed-size **IMediaSample** buffers to the decoder. If you implement an MPEG-2 decoder, it must be able to gather the video samples.

If the underlying decoding hardware of the device supports scatter-gather, it is trivial for the decoder filter to reassemble the samples. If that is not the case, the MPEG-2 decoder filter must merge the parts of the samples by copying to memory in order to provide full samples to the hardware. The decoder filter determines the length of an entire sample by parsing the sample header and it can determine the length of a partial sample by calling [IMediaSample::GetActualDataLength](http://go.microsoft.com/fwlink/?LinkId=215026) (<http://go.microsoft.com/fwlink/?LinkId=215026>).

MPEG-4-Specific Notes

The following are some notes that are helpful for the decoder implementer to know when using MPEG-4 with Windows Embedded Compact 7.

- The MPEG-4 demultiplexer ensures that every sample fits in a buffer by requesting a buffer size equal to the largest sample in the media on the output allocator.
- MPEG-4 seeking always starts from an I-Frame (also known as a key frame or an intra-encoded frame; the decoding of these frames does not depend on information from other frames). If a seek

is requested to a non-I-Frame, the demultiplexer finds the closest I-Frame and seeks to that location. This process might result in a staggered seek experience if the media that is being played is encoded with very sparse I-Frames.

Conclusion

In Windows Embedded Compact 7, you can use DirectShow to capture, play back, and transform media on a device. Because hardware differs from platform to platform, and some hardware provides media-handling functionality, a developer may choose to create a decoder filter that takes advantage of the specific hardware that he or she is developing for.

This article describes how to use DirectShow in the context of creating a decoder filter for media playback. To develop a decoder filter for your hardware platform, you need to understand how the decoder filter works with the other components that handle media input and output. The overview that is presented in this article describes how the source filter, buffering stream filter, demultiplexer filter, the audio or video decoder filter, and the audio or video renderer filter are connected together to form a filter graph. With that knowledge, the basic implementation steps follow. Those steps include implementing the decoder filter class, implementing the pin classes to connect the filters, allocating a buffer between the demultiplexer and the decoder, and specifying the type of media stream.

As with any development task, there are technological and design issues that must be accounted for in the final product. Issues such as threading models; avoiding data starvation; data seeking, pass through, and flushing; and the scatter-gather technique for processing media samples are also covered, so you can design a more effective decoder filter that provides a better user experience for your customers.

Additional Resources

- [Windows Embedded website](http://go.microsoft.com/fwlink/?LinkID=203338) (http://go.microsoft.com/fwlink/?LinkID=203338)

This document is provided “as-is.” Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2011 Microsoft. All rights reserved.