An Introduction to DirectShow Parser Filters

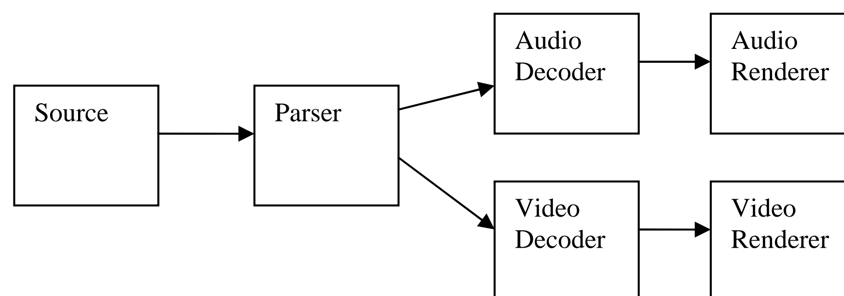
Geraint Davies

Introduction & Overview

DirectShow is a multimedia architecture on Windows® that is widely used for video playback as well as other tasks. Instead of a monolithic core with optional plug-ins, it is made up of separate, replaceable components that are connected together to make a *filter graph.* These filter graphs are built to perform the playback, recording or media processing task.

If there is a *playback engine* at the heart of DirectShow, it is the parser filter. This component is responsible for selecting the range of media data to be played and setting the timing of playback, as well as preparation of demultiplexed elementary stream data for decoding and delivery. The requirements and behaviour of the parser are not widely understood, although there are a number of cases where developers may need to develop their own. This article, and the accompanying sample parser, sets out to show the *how* and *why* of DirectShow parser development.

Shown below is a diagram of a very simple playback graph.



The source filter reads data from the original media source. This might be simply a read from a local file, but it could equally well fetch data from a URL or receive a UDP multicast. Essentially, the source filter provides access to the data without understanding it at all. The parser separates the elementary video and audio data, timestamps them to keep them sync with each other, and delivers them for decoding.

The parser's tasks are:

- Identify the format of each elementary stream that it wishes to expose, and create an appropriate media type for the corresponding output pin.

- Locate the start point. If the source provides random access to a file, this might mean use of an index or making estimates of the file

location. If the source is a stream without random access, this may mean simply discarding incoming data until an appropriate point is located.

- Identify the elementary stream frames or packets and deliver them to each pin.

- Timestamp each frame or packet if possible. For a seekable file, this will be based on the known file timing. For a live stream, the basis of timestamps will be determined on the fly when streaming begins.

To accompany this article, there is a fully working MPEG-1 parser available for download in source form from http://www.gdcl.co.uk. This is a seekable parser for local file playback in pull mode, which illustrates a number of the points discussed.

Identification and Connection

Getting the right filters

DirectShow plays back multimedia using a graph of connected filters. But it has no hard-wired knowledge of the graph layout required for any particular type of file. Instead, each graph is custom-built using a set of simple rules. The basic principles are the following:

- The source file is checked using a pattern-matching table that matches bit patterns at fixed locations in the file. This gives a source filter CLSID, and a media type (major/minor GUIDs) for the file type.

- Each output pin is rendered by trying in turn the filters available in the graph, and the filters registered for that media type.

- Output pin rendering is repeated recursively until the stream is rendered.

So how do you get your filter in the right place in the graph, and what role does it have it building the rest of the graph?

The first method uses the pattern matching table in HKEY_CLASSES_ROOT\Media Types. The keys and sub-keys listed here represent the known major types and subtypes and each value entry contains a pattern to match against fixed byte positions in the file. The graph manager's *AddSourceFilter* method scans this table, and when a match is found, the table gives the source filter's CLSID together with the major type and subtype for the file.

You can of course combine your source filter and parser, reading from the file as necessary. In this case the source filter CLSID points to your filter; the major type/subtype pair is not really used. However, most parsers will use the common source filter, so that they can work with other interchangeable source filters, such as the progressive download URL source. In this case, the source filter CLSID points to the File Source (Async), and the major type/subtype pair is used as a media type for the output pin of the source filter, and hence the input pin of your parser. This will typically be MAJORTYPE_Stream, and then a subtype describing the file format.

Your parser is then brought into the graph because it is registered as a handler for that type pair. You will then create output pins representing the elementary streams that you are going to output, and the rest of the graph is built stepwise from there.

The pattern match entries in the table specify bytes at a fixed position in the file, together with an AND pattern to be applied before the test. Each entry is a set of four strings (file position, length, mask and test); you can have multiple tests in a single entry, in which case all of them must apply. You can have several different entries for a particular subtype, and only one needs to apply. For an example, here is the entry for an MPEG-1 System file:

```
[HKEY_CLASSES_ROOT\Media Type\
        {e436eb83-524f-11ce-9f53-0020af0ba770}\
                {E436EB84-524F-11CE-9F53-0020AF0BA770}]
"0"="0, 16, FFFFFFFFF100010001800001FFFFFFFF,
000001BA2100010001800001000001BB"
"Source Filter"="{E436EBB5-524F-11CE-9F53-0020AF0BA770}"
```

This matches a pack header and system packet header at the very beginning of the file. Of course not all file types have known patterns at fixed positions. Mpeg-2 Transport Stream is easily recognisable since there are start codes with the value 0x47 every 188 bytes. However, if the file is part of a stream, it may not begin at a 0x47 start code. In this case, the pattern matching scheme cannot be used.

When the graph manager fails to match any entry in the table, it loads the default source filter *File Source (Async)* with a media type of MEDIATYPE_STREAM and MEDIASUBTYPE_NULL. Then any parsers registered for this wild card type will be loaded, and each can try to recognise the input format.

This is the mechanism chosen by the sample parser. You will see that the input pin is registered with this wild card subtype:

```
{
    &MEDIATYPE_Stream,
    &MEDIASUBTYPE_NULL
},
```

The input pin's CheckMediaType accepts any media type of MEDIATYPE_Stream. This means that it can be used whether or not the pattern-match table has succeeded in identifying the file format as MPEG-1.

Push and Pull

In the early days of DirectShow development, there was a good deal of nearly religious debate about the merits of push and pull as models for data delivery. In the push mode, the supplier delivers data when it is ready, but it is limited by flow control mechanisms, such as a limited buffer count. In the pull mode, the consumer requests data when it is ready to process it, but it is limited by the availability of data at the supplier. In the end, there is not much difference between them, and the push model was selected for all filters.

However, there is a case where the push model does not work effectively. The standard source filter is just a wrapper around the ReadFile API (although the sector-aligned unbuffered reads are efficient). But placing this functionality in a separate filter means that data can be received from other sources, most notably the URL source used for progressive download from web sites. Even here, a seekable push model would work for MPEG files, which can be read and played sequentially as a stream. But AVI files (and other table-based formats such as QuickTime and MPEG-4) require the reader to use an index. This means that the parser needs to have random access to the file during reading (even if, most of the time, the reads are very localised and sequential).

To allow the AVI parser to use a common file source, the IAsyncReader interface was designed. This is intended to permit random access to the source data using efficient, overlapped I/O. However, it has to be said that it introduces some complexity, and in many cases the benefits of interoperability with other parsers or sources are outweighed by the added complexity.

Mpeg files can be parsed and read as a stream, and so the sample parser does not need random access to the source data. It therefore uses a class at the input pin which uses a worker thread to simulate a push-mode interface – the worker thread simply requests each block in turn and delivers it to the input pin's receive method. If you use this scheme, your input pin needs only three interactions with the CPullPin object:

- Create an object derived from CPullPin when your input pin is connected.
- Activate the CPullPin object when the filter leaves stop mode. Note that, since the pulling and delivery occurs on another thread, the first data could be

delivered before the Active() call has returned. You therefore need to position the Active call carefully to make sure that the filter state has already been changed. In the sample filter, the filter's Pause method calls a method on the input pin to set the start position and activate the pulling thread.

- De-activate the CPullPin class when the filter enters stop mode – in the example, this is in the input pin's Inactive method.

In addition, the Receive method needs to be overridden to forward the data to the parser's main processing function – and that's essentially it.

I have made the sample parser more complicated because it supports variable bit rate files, and this requires calls to the Seek function of CPullPin made on the worker thread itself – but this is covered more clearly below. For this reason, you will see that the sample parser uses a CPullPin2 class.

Input recognition

One of the jobs of the parser is to identify the elementary streams contained in the multiplex and create correctly-typed output pins for each one. How this is done depends on the source format, but typically this requires the parser to scan parts of the file. This must be done when the input pin is connected, so that the output pins can be correctly rendered for playback.

In the sample parser, the input pin's CompleteConnect method passes control to the parser filter to allow it to verify the input format and create the correct output pins. The filter checks that this really is an MPEG-1 format that it can understand (see section *Getting the right filters* above). Then it scans the file for elementary stream headers that are used to create the fully-detailed media types required to connect to the decoders. At the same time, the sample parser establishes the time base and duration of the file.

The IAsyncReader interface was designed to allow parser filters to read the data at connection time for this very reason. The *SyncRead* method can be called even when the filter is in Stop state – for the standard source filter, this is simply a call to ReadFile. You can see in the sample parser, the CompleteConnect method, in PESScanner::SyncFill, uses IAsyncReader::SyncRead to read the beginning and the end of the file.

Not all source/parsers have this option. A parser for live data would not be able to have random access to the data, and would not access the data until the source is active. For this situation, two solutions have become common:

- For parsers that work with push-mode (IMemInputPin) source filters, it is common to implement IStream::Read on the source output pin so that the parser can analyse the data during connection. Alternatively, combine the source and parser, so that the parser can read the source data via a private interface.

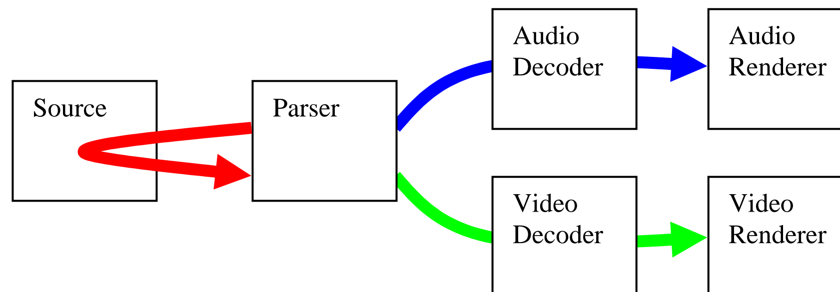
- Create the output pins using default media types. When the first data is received, detect the correct media type detail (and time base) and attach the media type as a dynamic type change to the first sample. This will only work if the number of elementary streams extracted is fixed in advance, and if the changes to the type detail do not require a different decoder.

Operation

Thread organisation

A DirectShow filter needs to be aware of a number of different threads that may execute code inside the filter at any time. These threads fall into two groups: state change threads such as the application's main thread and graph manager background threads, and worker threads that deliver data.

The normal rule in DirectShow is that each graph segment should have a separate delivery thread. You do not need a separate thread for each filter, but wherever a stream begins, or is split into multiple outputs, you will normally have a thread for each stream. So in the typical playback graph, there is a thread which delivers data to the parser, and then a thread in each parser output pin which delivers the data downstream. Both decoding and rendering take place on this parser output thread. So for our pull-mode sample parser, there will be three delivery threads, shown here in red, blue and green.



This shows three delivery threads:

- The red thread in the input pin calls the source output pin's SyncReadAligned method, and returns the data to the filter's Receive method. This thread is created in the CPullPin2 class, and executes the CPullPin2::Process method.

- The blue thread in the audio output pin delivers audio data downstream, taking packets off the queue and calling the Receive method on the audio decoder's input pin. This method will typically decompress the data and deliver the decompressed data to the renderer, where – still on the same thread inside the Receive call – the decompressed data will be delivered to the device driver for rendering.

- The green thread delivers compressed video data to the video decoder, where it will be decompressed and delivered to the video renderer. The video renderer will typically block this thread until it is time to complete the drawing of the frame – if the graph is paused, this might block indefinitely.

If the red source thread were used to deliver data downstream after parsing, it would not be possible to perform audio decoding and video decoding in parallel. More importantly, when the first video frame were delivered to the video renderer, this thread would block until the graph left Paused state, and no more audio would be decompressed. Of course, the graph might be waiting for decompressed audio before it can leave Paused state, so the graph could easily deadlock.

For this reason, parsers typically use a worker thread on each output pin. This is contained in the COutputQueue class in the base class library: sample buffers are placed on a queue at the output pin, and the COutputQueue worker thread collects them and delivers them to the connected input pin's Receive method.

Some Receive implementations will immediately queue the data without blocking, and then process it on some other thread. In this case, it is inefficient to have a worker thread in the COutputQueue class, as it just introduces an unnecessary thread switch. This is why the IMemInputPin interface contains a method *ReceiveCanBlock*. The COutputQueue class will only create a worker thread if *ReceiveCanBlock*. If not, the sample buffers are not queued, but are delivered immediately downstream. So if you have a filter that queues data immediately and does not block (within itself or further downstream), you can override the default implementation of this and avoid unneeded worker threads.

There are a few other points worth mentioning about the COutputQueue class:

- To minimise the cost of thread switching, the queue does not activate the worker thread until the batch limit is reached. The sample parser avoids problems with this by forcing a thread activation at the end of every input buffer: this is the reason for the SendAnyway() calls in the filter's Receive method.

- When a sample buffer is passed to the queue, a reference count is passed with it. That is, the COutputQueue class does not AddRef the sample, but it will Release it.

State changes between Stopped, Paused and Running states occur on the application's main thread, or sometimes on graph manager background threads. The parser does not need to handle Running state – paused and running are the same except to renderers – so there are three relevant methods in the filter:

Pause	The filter's pause method is called when the filter is going active. Transitions to Run will always call the Pause method first. The point to note here is that the three worker threads will be started during this call, and may start work before this method returns, so the order of operations is important. Remember that during your Pause method, downstream filters will already be paused, but those upstream will not.
Stop	The Inactive methods on all three pins will be called during the Stop processing. Since all the graph delivery threads are created by the parser, these Inactive methods will need to stop the worker threads. Since the graph manager has already stopped the downstream filters, there is no danger of the threads being blocked downstream.
Seek	This method is the most complicated, since it involves suspending the input pin thread when the rest of the graph is still running. This is discussed in Seeking.

Buffering

DirectShow is designed to minimise the number of unnecessary buffer copies introduced during copying. This was a serious problem with Video for Windows under some circumstances, and the DirectShow design tries to avoid forcing buffer copies between filters. Each pin-to-pin connection negotiates its own allocator for buffer space. To allow both sides to use memory efficiently, the negotiation includes the number of buffers available, the size of each buffer and details such as prefix space and buffer alignment. Both pins can propose an allocator, but the output pin has the final decision.

The default behaviour is an allocator which allocates a fixed number of buffers, all of the same size. This is the simplest to use since it does not involve any custom allocator creation. It works fine for data in a stream format (such as uncompressed audio) or with a fixed size of sample (such as uncompressed video, where each buffer contains one frame). However for the compressed elementary stream data in a parser, it is not ideal.

The sample parser uses a default allocator like this. There are two drawbacks: the compressed data must be copied from the input buffer into the output buffer, and the output buffers are all of a fixed size. The copy of data is unlikely to be a problem in our sample parser, since the bandwidth of compressed MPEG-1 is so small that the copy will use minimal resources. However, the fixed buffer size may be more of a problem. For compatibility with the decoders, we are expected to place one PES packet in each buffer. A PES packet can be up to 64Kb in length (or more in MPEG-2 Transport Stream). However, many are only a few hundred bytes. Using a default allocator means we have to make all our buffers 64Kb in length, and waste a lot of memory. This decision is made in the DecideBufferSize method of the ElementaryOutputPin class.

Using a custom allocator would solve both of these problems, but involves considerably more development effort, and thus is only appropriate where the memory usage or copying overhead is significant. The basis of the solution is the following:

- The input pin for the parser has a custom allocator derived from `CMemAllocator`. This overrides `GetBuffer` to ensure that all the buffers are contiguous and that they are allocated strictly in order. This makes sure that elementary stream objects are in a single contiguous block even if they are split across two input buffers.

- Each output pin has a custom allocator derived from CBaseAllocator. The samples point to memory within the input allocator's buffers, and they hold a refcount on the input samples.

- The parser scans through the set of input buffers as a single contiguous block (copying up data to avoid problems at the end of the block). When it locates an elementary stream packet, the output pin's custom allocator creates a sample header – just the CMediaSample object – with the data pointer pointing into the input buffer. The reference count held by this sample object ensures that the parser does not re-use that data area until the output sample has been released.

Buffering and Flow Control

Whichever buffering strategy is used, there is a fixed amount of buffering space available at the parser, and it is this limit that controls the build-up of compressed data waiting for the decoder. In the sample parser, the count of buffers at the output pin is limited to the 30 buffers specified in the DecideBufferSize call. If the graph is paused, or the video stream is blocked waiting for the render time of a frame, these buffers can all be in the COutputQueue queue waiting for delivery to the decoder. This means that the parser thread will block in DeliverData calling GetDeliveryBuffer, until the graph is un-paused or the render time is reached.

For the custom allocator design outlined above, the limitation is the input connection's buffers, and it is the input delivery to the parser that will be blocked until some of the output samples are freed and can dereference the input data area.

Problems can sometimes arise when the filter blocks indefinitely in this state. This can happen if the audio and video are widely separated in the interleaved stream, and the parser has insufficient buffering for this separation. The graph will not transition from paused to running state until data is received at all renderers, and it can happen that the parser is blocked, unable to get more buffers to deliver to the video output pin, when it has not delivered any audio yet. Similar problems can occur with the audio clock unable to advance until more audio data arrives, but the parser is blocked by the video which is waiting for the audio clock to advance. The solution in these cases is usually to increase the parser buffering to allow for the maximum offset between elementary streams in the interleaved stream.

Timestamp mapping

In DirectShow, synchronisation of playback uses timestamps on each sample which are compared against a common reference clock. These timestamps usually originate in the parser: as the elementary stream objects are separated and delivered downstream, they are timestamped relative to the seek start point. These are usually passed on, unused, by intermediate filters and then acted on only by the renderers. The video renderer will block until the timestamp before drawing the frame (unless it is falling behind, in which case it may drop the frame or even play it slightly early). The audio renderer derives the graph clock from the timestamps: that is, if it is playing a sample marked with time X, then the time now is, by definition, X.

How the timestamp conversion is done depends on the source format. AVI files enforce a fixed frame rate for the whole video stream, so the timestamp for a frame is simply the frame index multiplied by the frame duration. MPEG 1 and 2 files do not have a frame

index or a fixed frame rate, but they do have a number of timestamps and clock references in the file. An MPEG parser will normally establish some form of mapping between the MPEG clock references and the DirectShow timestamps.

The sample parser bases its timing on the audio PTS values that indicate the presentation time of the audio frames. These are reliable, and avoid the frame re-ordering issues that would result from using the video timestamps. They are also verifiable: it is quite straightforward to measure the duration of each audio frame and compare this to the advance in audio PTS values (although the sample parser does not perform this check).

The sample parser establishes the time mapping when its input pin is first connected to the source filter, in the filter's CompleteConnect method. It records the first PTS found on an audio PES packet as the baseline. Then it scans the final 2Mb of the file for the last audio PTS value. The difference between the two gives the file duration. Before delivering a PES packet, the baseline PTS is subtracted from the PES PTS and the result converted from 90KHz to 100ns units. This simple mechanism means that all PES packets are timestamped relative to the file start in a reliable manner.

However, there are a number of difficulties that this approach does not address:

- PTS values are 33-bit fields, and will wrap around to 0 roughly every 26.4 hours. Some encoders will wrap at 32 bits instead. If the wrap occurs within the file, both the duration and the relative timestamps will be wrong. It is possible to detect this by comparing the mux rate (in the pack header) against the bitrate calculated from the difference in PTS values.

- Discontinuities can occur within the file. These can be detected and the file played correctly – although the duration will be wrong and seeking will not be possible. You can detect discontinuities by comparing the audio duration with the audio PTS values. When a jump is detected, the baseline PTS value needs to be adjusted by the same amount, so that PTS values after the jump convert to DirectShow times that follow exactly the previous times.

- Live sources will have a different baseline PTS value each time the filter is activated. In this case, the baseline PTS detection code will need to be run when the first data arrives each time the filter goes into paused state. This simple change in behaviour from the default runtime parser is the main reason for re-writing parsers in recent years.

Seeking

DirectShow provides random access to the file via the IMediaSeeking interface. This allows an application to specify the start and stop point for playback (in time, frames or bytes) and to query the current playback position and media duration. It is the parser that selects which elementary stream objects are to be decompressed, and hence it is the parser that must implement seeking.

To implement seeking in your parser, you need to:

- Implement SetPositions to start playback from a new position.
- Adjust all timestamps to be relative to the seek position, rather than the file start
- Report the total duration of the media.

Current position is normally reported by the video renderer, if the parser reports a duration. The parser also needs to enable control over the stop position and the playback rate, although these are less significant.

There are three complex areas to be dealt with: the route by which the seeking calls arrive at the parser, the location of the seek start point in the media, and the thread interactions involved in seeking.

Routing of Seek calls

When the application makes a seek call to the graph, it will normally be handled by the parser. However, the graph manager does not know which filters in the graph can handle seeking, or how many parsers there are. Instead, it concentrates on the end result: it asks each of the renderers in the graph to seek their data source. Renderers and transform filters will then pass the seeking calls upstream until a filter is reached which can perform the seeking operation. In this way, the seeking calls are routed to the right filters without any hardwired knowledge in the graph manager.

A parser filter will receive seeking calls on each of its output pins. However, most parsers do not seek their elementary streams independently (and applications do not request them to). Most parsers will accept calls from one pin, and will ignore the other pins. The standard way to select a pin is to acknowledge the first pin to set the seeking time format to something other than TIME_FORMAT_NONE. You can see, in the sample parser, the SetTimeFormat method of the output pin tries to enable itself as the seeker, using a filter method that is protected by a dedicated critical section. Seeking calls via output pins that have not been selected are ignored – it is best to return a success code, since returning an error code can result in failures in some applications, such as Windows Media Player.

Seeking Thread Interactions

To ensure correct synchronisation of the playback from the new position, the graph cannot be running when the seeking call is made. If the graph is in run state, the graph manager will pause the graph before calling the seeking methods on the filters.Then the SetPositions call will be made on an application or a graph worker thread, while all the filter threads are blocked in paused state on GetBuffer or downstream Receive calls.

Your filter's Seek method – called from the output pin's SetPositions method – needs to suspend the parser's worker thread: that is, the input pin's thread on which the parsing is performed. This thread is likely to be blocked in a GetBuffer call. The way to unblock this thread is to call BeginFlush. This call, which needs to be propagated downstream, releases all buffers and causes all Receive calls to fail. Then, while the worker thread is cycling through a series of errors, you can signal the thread to stop. Once the thread has stopped executing, you can then call EndFlush, which returns the GetBuffer and Receive behaviour to normal. You can see all this code in the base class method CPullPin::Seek: note the way that BeginFlush and EndFlush methods are passed down the entire graph starting at the implPull override of BeginFlush.

Once the thread is suspended, the start position is set and the thread can be restarted. For a non-indexed format such as MPEG-1, the parser needs to locate the seek point in the file. This is normally done by a simple estimate such as

position = target x File Size / Duration.

However, in the sample parser I have added some extra complexity to support seeking of variable bitrate files (discussed below).

Finally the worker thread can be restarted – remembering that the filter may have been either stopped or paused when the seeking call was made.

Variable Bitrate Seeking

If the bitrate varies widely over the file, the simple estimated start position may be somewhat inaccurate. Of course the bitrate is normally stable over a medium period, and an MPEG parser needs to start at the previous GOP, so the simple estimate is usually more than accurate enough. However, for the purposes of illustration, I have implemented a variable-bitrate seeking strategy in the sample parser.

The basic idea is that parsing begins at the initial estimated position. First, the timestamps at this location are checked. If they are too inaccurate, a new estimate is made and a new seek call is issued. Each revised estimate is based on the bitrate actually found in the region of the last estimate.

The only complication that this introduces is that the standard CPullPin class cannot accept Seek calls made on the CPullPin worker thread itself. Unfortunately, too much of the CPullPin class is declared private, so the only way round this is to create a revised CPullPin2 class, which has the small changes needed. If you compare CPullPin and CPullPin2, you will see that the only change is the way that the seek call suspends the worker thread.