

原 LAV Filter 源代码分析 4：LAV Video（2）

2013年10月27日 21:47:43 阅读数：6112

上一篇文章分析了LAV Filter 中的LAV Video的两个主要的类：CLAVVideo和CDecodeThread。文章：[LAV Filter 源代码分析 3：LAV Video（1）](#)

在这里继续上篇文章的内容。文章中提到LAVVideo主要通过 CDecodeThread这个类进行解码线程的管理,其中有一个关键的管理函数：ThreadProc(), 包含了解码线程的各种操作。函数如下所示：

```
[cpp]
1. //包含了对进程的各种操作
2. DWORD CDecodeThread::ThreadProc()
3. {
4.     HRESULT hr;
5.     DWORD cmd;
6.
7.     BOOL bEOS = FALSE;
8.     BOOL bReinit = FALSE;
9.
10.    SetThreadName(-1, "LAVVideo Decode Thread");
11.
12.    HANDLE hWaitEvents[2] = { GetRequestHandle(), m_evInput };
13.    //不停转圈, 永不休止
14.    while(1) {
15.        if (!bEOS && !bReinit) {
16.            // Wait for either an input sample, or an request
17.            WaitForMultipleObjects(2, hWaitEvents, FALSE, INFINITE);
18.        }
19.        //根据操作命令的不同
20.        if (CheckRequest(&cmd)) {
21.            switch (cmd) {
22.                //创建解码器
23.                case CMD_CREATE_DECODER:
24.                {
25.                    CAutoLock lock(&m_ThreadCritSec);
26.                    //创建
27.                    hr = CreateDecoderInternal(m_ThreadCallContext.pmt, m_ThreadCallContext.codec);
28.                    Reply(hr);
29.
30.                    m_ThreadCallContext.pmt = NULL;
31.                }
32.                break;
33.                case CMD_CLOSE_DECODER:
34.                {
35.                    //关闭
36.                    ClearQueues();
37.                    SAFE_DELETE(m_pDecoder);
38.                    Reply(S_OK);
39.                }
40.                break;
41.                case CMD_FLUSH:
42.                {
43.                    //清楚
44.                    ClearQueues();
45.                    m_pDecoder->Flush();
46.                    Reply(S_OK);
47.                }
48.                break;
49.                case CMD_EOS:
50.                {
51.                    bEOS = TRUE;
52.                    m_evEOSDone.Reset();
53.                    Reply(S_OK);
54.                }
55.                break;
56.                case CMD_EXIT:
57.                {
58.                    //退出
59.                    Reply(S_OK);
60.                    return 0;
61.                }
62.                break;
63.                case CMD_INIT_ALLOCATOR:
64.                {
65.                    CAutoLock lock(&m_ThreadCritSec);
66.                    hr = m_pDecoder->InitAllocator(m_ThreadCallContext.allocator);
67.                    Reply(hr);
68.
69.                    m_ThreadCallContext.allocator = NULL;
70.                }
71.                break;
72.                case CMD_POST_CONNECT:
73.                {
74.                    CAutoLock lock(&m_ThreadCritSec);
75.                    hr = PostConnectInternal(m_ThreadCallContext.pin);
76.                    Reply(hr);
```

```

77.         m_ThreadCallContext.pin = NULL;
78.     }
79.     break;
80. case CMD_REINIT:
81.     {
82.         //重启
83.         CMediaType &mt = m_pLAVVideo->GetInputMediaType();
84.         CreateDecoderInternal(&mt, m_Codec);
85.         m_TempSample[1] = m_NextSample;
86.         m_NextSample = m_FailedSample;
87.         m_FailedSample = NULL;
88.         bReinit = TRUE;
89.         m_evEOSDone.Reset();
90.         Reply(S_OK);
91.         m_bDecoderNeedsReInit = FALSE;
92.     }
93.     break;
94. default:
95.     ASSERT(0);
96. }
97. }
98. }
99.
100. if (m_bDecoderNeedsReInit) {
101.     m_evInput.Reset();
102.     continue;
103. }
104.
105. if (bReinit && !m_NextSample) {
106.     if (m_TempSample[0]) {
107.         m_NextSample = m_TempSample[0];
108.         m_TempSample[0] = NULL;
109.     } else if (m_TempSample[1]) {
110.         m_NextSample = m_TempSample[1];
111.         m_TempSample[1] = NULL;
112.     } else {
113.         bReinit = FALSE;
114.         m_evEOSDone.Set();
115.         m_evSample.Set();
116.         continue;
117.     }
118. }
119. //获得一份数据
120. IMediaSample *pSample = GetSample();
121. if (!pSample) {
122.     // Process the EOS now that the sample queue is empty
123.     if (bEOS) {
124.         bEOS = FALSE;
125.         m_pDecoder->EndOfStream();
126.         m_evEOSDone.Set();
127.         m_evSample.Set();
128.     }
129.     continue;
130. }
131. //解码
132. DecodeInternal(pSample);
133.
134. // Release the sample
135. //释放
136. SafeRelease(&pSample);
137.
138. // Indicates we're done decoding this sample
139. m_evDecodeDone.Set();
140.
141. // Set the Sample Event to unblock any waiting threads
142. m_evSample.Set();
143. }
144.
145. return 0;
146. }

```

该函数中，DecodeInternal(pSample)为实际上真正具有解码功能的函数，来看看它的源代码吧：

```

1.  STDMETHODIMP CDecodeThread::DecodeInternal(IMediaSample *pSample)
2.  {
3.      HRESULT hr = S_OK;
4.
5.      if (!m_pDecoder)
6.          return E_UNEXPECTED;
7.      //调用接口进行解码
8.      hr = m_pDecoder->Decode(pSample);
9.
10.     // If a hardware decoder indicates a hard failure, we switch back to software
11.     // This is used to indicate incompatible media
12.     if (FAILED(hr) && m_bHWDDecoder) {
13.         DbgLog((LOG_TRACE, 10, L"::Receive(): Hardware decoder indicates failure, switching back to software"));
14.         m_bHWDDecoderFailed = TRUE;
15.
16.         // Store the failed sample for re-try in a moment
17.         m_FailedSample = pSample;
18.         m_FailedSample->AddRef();
19.
20.         // Schedule a re-init when the main thread goes there the next time
21.         m_bDecoderNeedsReInit = TRUE;
22.
23.         // Make room in the sample buffer, to ensure the main thread can get in
24.         m_TempSample[0] = GetSample();
25.     }
26.
27.     return S_OK;
28. }

```

该函数比较简短，从源代码中可以看出，调用了 m_pDecoder 的 Decode() 方法。其中 m_pDecoder 为 ILAVDecoder 类型的指针，而 ILAVDecoder 是一个接口，并不包含实际的方法，如下所示。注意，从程序注释中可以看出，每一个解码器都需要实现该接口规定的函数。

```

1.  /**
2.   * Decoder interface
3.   *
4.   * Every decoder needs to implement this to interface with the LAV Video core
5.   */
6.  //接口
7.  interface ILAVDecoder
8.  {
9.      /**
10.       * Virtual destructor
11.       */
12.       virtual ~ILAVDecoder(void) {};
13.
14.       /**
15.        * Initialize interfaces with the LAV Video core
16.        * This function should also be used to create all interfaces with external DLLs
17.        *
18.        * @param pSettings reference to the settings interface
19.        * @param pCallback reference to the callback interface
20.        * @return S_OK on success, error code if this decoder is lacking an external support dll
21.        */
22.        STDMETHOD(InitInterfaces)(ILAVVideoSettings *pSettings, ILAVVideoCallback *pCallback) PURE;
23.
24.       /**
25.        * Check if the decoder is functional
26.        */
27.        STDMETHOD(Check)() PURE;
28.
29.       /**
30.        * Initialize the codec to decode a stream specified by codec and pmt.
31.        *
32.        * @param codec Codec Id
33.        * @param pmt DirectShow Media Type
34.        * @return S_OK on success, an error code otherwise
35.        */
36.        STDMETHOD(InitDecoder)(AVCodecID codec, const CMediaType *pmt) PURE;
37.
38.       /**
39.        * Decode a frame.
40.        *
41.        * @param pSample Media Sample to decode
42.        * @return S_OK if decoding was successfull, S_FALSE if no frame could be extracted, an error code if the decoder is not compatible
43.        * with the bitstream
44.        *
45.        * Note: When returning an actual error code, the filter will switch to the fallback software decoder! This should only be used for
46.        * catastrophic failures,
47.        * like trying to decode a unsupported format on a hardware decoder.
48.        */
49.        STDMETHOD(Decode)(IMediaSample *pSample) PURE;
50.
51.       /**
52.        * Flush the decoder after a seek.
53.        * The decoder should discard any remaining data.
54.        */

```

```

53.  * @return unused
54.  */
55.  STDMETHOD(Flush)() PURE;
56.
57.  /**
58.   * End of Stream
59.   * The decoder is asked to output any buffered frames for immediate delivery
60.   *
61.   * @return unused
62.   */
63.  STDMETHOD(EndOfStream)() PURE;
64.
65.  /**
66.   * Query the decoder for the current pixel format
67.   * Mostly used by the media type creation logic before playback starts
68.   *
69.   * @return the pixel format used in the decoding process
70.   */
71.  STDMETHOD(GetPixelFormat)(LAVPixelFormat *pPix, int *pBpp) PURE;
72.
73.  /**
74.   * Get the frame duration.
75.   *
76.   * This function is not mandatory, and if you cannot provide any specific duration, return 0.
77.   */
78.  STDMETHOD_(REFERENCE_TIME, GetFrameDuration)() PURE;
79.
80.  /**
81.   * Query whether the format can potentially be interlaced.
82.   * This function should return false if the format can 100% not be interlaced, and true if it can be interlaced (but also progressive).
83.   */
84.  STDMETHOD_(BOOL, IsInterlaced)() PURE;
85.
86.  /**
87.   * Allows the decoder to handle an allocator.
88.   * Used by DXVA2 decoding
89.   */
90.  STDMETHOD(InitAllocator)(IMemAllocator **ppAlloc) PURE;
91.
92.  /**
93.   * Function called after connection is established, with the pin as argument
94.   */
95.  STDMETHOD(PostConnect)(IPin *pPin) PURE;
96.
97.  /**
98.   * Get the number of sample buffers optimal for this decoder
99.   */
100.  STDMETHOD_(long, GetBufferCount)() PURE;
101.
102.  /**
103.   * Get the name of the decoder
104.   */
105.  STDMETHOD_(const WCHAR*, GetDecoderName)() PURE;
106.
107.  /**
108.   * Get whether the decoder outputs thread-safe buffers
109.   */
110.  STDMETHOD(HasThreadSafeBuffers)() PURE;
111.
112.  /**
113.   * Get whether the decoder should sync to the main thread
114.   */
115.  STDMETHOD(SyncToProcessThread)() PURE;
116. };

```

下面来看看封装libavcodec库的类吧，该类的定义位于decoders文件夹下，名为avcodec.h，如图所示：

该类名字叫CDecAvcodec，其继承了CDecBase。而CDecBase继承了ILAVDecoder。

```

1.  /* 雷霄骅
2.   * 中国传媒大学/数字电视技术
3.   * leixiaohua1020@126.com
4.   *
5.   */
6.  /*
7.   * Copyright (C) 2010-2013 Hendrik Leppkes
8.   * http://www.1f0.de
9.   *
10.  * This program is free software; you can redistribute it and/or modify
11.  * it under the terms of the GNU General Public License as published by
12.  * the Free Software Foundation; either version 2 of the License, or
13.  * (at your option) any later version.
14.  */

```

```

15.  * This program is distributed in the hope that it will be useful,
16.  * but WITHOUT ANY WARRANTY; without even the implied warranty of
17.  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
18.  * GNU General Public License for more details.
19.  *
20.  * You should have received a copy of the GNU General Public License along
21.  * with this program; if not, write to the Free Software Foundation, Inc.,
22.  * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
23.  */
24.
25. #pragma once
26.
27. #include "DecBase.h"
28. #include "H264RandomAccess.h"
29.
30. #include <map>
31.
32. #define AVCODEC_MAX_THREADS 16
33.
34. typedef struct {
35.     REFERENCE_TIME rtStart;
36.     REFERENCE_TIME rtStop;
37. } TimingCache;
38. //解码器 (AVCODEC) (其实还有WMV9, CUVID等)
39. class CDecAvcodec : public CDecBase
40. {
41. public:
42.     CDecAvcodec(void);
43.     virtual ~CDecAvcodec(void);
44.
45.     // ILAVDecoder
46.     STDMETHODIMP InitDecoder(AVCodecID codec, const CMediaType *pmt);
47.     //解码
48.     STDMETHODIMP Decode(const BYTE *buffer, int buflen, REFERENCE_TIME rtStart, REFERENCE_TIME rtStop, BOOL bSyncPoint, BOOL bDiscontinuity);
49.     STDMETHODIMP Flush();
50.     STDMETHODIMP EndOfStream();
51.     STDMETHODIMP GetPixelFormat(LAVPixelFormat *pPix, int *pBpp);
52.     STDMETHODIMP_(REFERENCE_TIME) GetFrameDuration();
53.     STDMETHODIMP_(BOOL) IsInterlaced();
54.     STDMETHODIMP_(const WCHAR*) GetDecoderName() { return L"avcodec"; }
55.     STDMETHODIMP HasThreadSafeBuffers() { return S_OK; }
56.     STDMETHODIMP SyncToProcessThread() { return m_pAVCtx && m_pAVCtx->thread_count > 1 ? S_OK : S_FALSE; }
57.
58.     // CDecBase
59.     STDMETHODIMP Init();
60.
61. protected:
62.     virtual HRESULT AdditionalDecoderInit() { return S_FALSE; }
63.     virtual HRESULT PostDecode() { return S_FALSE; }
64.     virtual HRESULT HandleDXVA2Frame(LAVFrame *pFrame) { return S_FALSE; }
65.     //销毁解码器, 各种Free
66.     STDMETHODIMP DestroyDecoder();
67.
68. private:
69.     STDMETHODIMP ConvertPixFmt(AVFrame *pFrame, LAVFrame *pOutFrame);
70.
71. protected:
72.     AVCodecContext      *m_pAVCtx;
73.     AVFrame              *m_pFrame;
74.     AVCodecID            m_nCodecId;
75.     BOOL                 m_bDXVA;
76.
77. private:
78.     AVCodec              *m_pAVCodec;
79.     AVCodecParserContext *m_pParser;
80.
81.     BYTE                 *m_pFFBuffer;
82.     BYTE                 *m_pFFBuffer2;
83.     int                  m_nFFBufferSize;
84.     int                  m_nFFBufferSize2;
85.
86.     SwsContext           *m_pSwsContext;
87.
88.     CH264RandomAccess    m_h264RandomAccess;
89.
90.     BOOL                 m_bNoBufferConsumption;
91.     BOOL                 m_bHasPalette;
92.
93.     // Timing settings
94.     BOOL                 m_bFFReordering;
95.     BOOL                 m_bCalculateStopTime;
96.     BOOL                 m_bRVDropBFrameTimings;
97.     BOOL                 m_bInputPadded;
98.
99.     BOOL                 m_bBFrameDelay;
100.    TimingCache          m_tcBFrameDelay[2];
101.    int                  m_nBFramePos;
102.
103.    TimingCache          m_tcThreadBuffer[AVCODEC_MAX_THREADS];
104.    int                  m_CurrentThread;

```

```

105.
106.     REFERENCE_TIME      m_rtStartCache;
107.     BOOL                m_bResumeAtKeyFrame;
108.     BOOL                m_bWaitingForKeyFrame;
109.     int                 m_iInterlaced;
110. };

```

从 CDecAvcodec 类的定义可以看出，包含了各种功能的函数。首先我们看看初始化函数Init()

```

1. // ILAVDecoder
2. STDMETHODIMP CDecAvcodec::Init()
3. {
4.     #ifdef DEBUG
5.         DbgSetModuleLevel (LOG_CUSTOM1, DWORD_MAX); // FFMPEG messages use custom1
6.         av_log_set_callback(lavf_log_callback);
7.     #else
8.         av_log_set_callback(NULL);
9.     #endif
10.    //注册
11.    avcodec_register_all();
12.    return S_OK;
13. }

```

可见其调用了ffmpeg的API函数avcodec_register_all()进行了解码器的注册。

我们再来看看其解码函数Decode()：

```

1. //解码
2. STDMETHODIMP CDecAvcodec::Decode(const BYTE *buffer, int buflen, REFERENCE_TIME rtStartIn, REFERENCE_TIME rtStopIn, BOOL bSyncPoint,
3.    BOOL bDiscontinuity)
4. {
5.     int    got_picture = 0;
6.     int    used_bytes = 0;
7.     BOOL   bParserFrame = FALSE;
8.     BOOL   bFlush = (buffer == NULL);
9.     BOOL   bEndOfSequence = FALSE;
10.    //初始化Packet
11.    AVPacket avpkt;
12.    av_init_packet(&avpkt);
13.
14.    if (m_pAVCtx->active_thread_type & FF_THREAD_FRAME) {
15.        if (!m_bFFReordering) {
16.            m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
17.            m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
18.        }
19.
20.        m_CurrentThread = (m_CurrentThread + 1) % m_pAVCtx->thread_count;
21.    } else if (m_bFFFrameDelay) {
22.        m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
23.        m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
24.        m_nBFFramePos = !m_nBFFramePos;
25.    }
26.
27.    uint8_t *pDataBuffer = NULL;
28.    if (!bFlush && buflen > 0) {
29.        if (!m_bInputPadded && !(m_pAVCtx->active_thread_type & FF_THREAD_FRAME) || m_pParser) {
30.            // Copy bitstream into temporary buffer to ensure overread protection
31.            // Verify buffer size
32.            if (buflen > m_nFFBufferSize) {
33.                m_nFFBufferSize = buflen;
34.                m_pFFBuffer = (BYTE *)av_realloc_f(m_pFFBuffer, m_nFFBufferSize + FF_INPUT_BUFFER_PADDING_SIZE, 1);
35.                if (!m_pFFBuffer) {
36.                    m_nFFBufferSize = 0;
37.                    return E_OUTOFMEMORY;
38.                }
39.            }
40.
41.            memcpy(m_pFFBuffer, buffer, buflen);
42.            memset(m_pFFBuffer+buflen, 0, FF_INPUT_BUFFER_PADDING_SIZE);
43.            pDataBuffer = m_pFFBuffer;
44.        } else {
45.            pDataBuffer = (uint8_t *)buffer;
46.        }
47.
48.        if (m_nCodecId == AV_CODEC_ID_H264) {
49.            BOOL bRecovered = m_h264RandomAccess.searchRecoveryPoint(pDataBuffer, buflen);
50.            if (!bRecovered) {
51.                return S_OK;
52.            }
53.        } else if (m_nCodecId == AV_CODEC_ID_VP8 && m_bWaitingForKeyFrame) {
54.            if (!(pDataBuffer[0] & 1)) {
55.                DbgLog((LOG_TRACE, 10, L":Decode(): Found VP8 key-frame, resuming decoding"));
56.                m_bWaitingForKeyFrame = FALSE;
57.            } else {
58.                return S_OK;
59.            }
60.        }
61.    }
62.
63.    if (bFlush) {
64.        if (m_pParser) {
65.            m_pParser->flush(pDataBuffer, buflen);
66.        }
67.        if (m_bFFReordering) {
68.            m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
69.            m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
70.        }
71.        if (m_bFFFrameDelay) {
72.            m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
73.            m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
74.            m_nBFFramePos = !m_nBFFramePos;
75.        }
76.    }
77.
78.    if (m_pParser) {
79.        m_pParser->decode(pDataBuffer, buflen, &got_picture, &used_bytes, &bParserFrame, &bEndOfSequence);
80.    }
81.
82.    if (got_picture) {
83.        if (bFlush) {
84.            if (m_pParser) {
85.                m_pParser->flush(pDataBuffer, buflen);
86.            }
87.            if (m_bFFReordering) {
88.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
89.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
90.            }
91.            if (m_bFFFrameDelay) {
92.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
93.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
94.                m_nBFFramePos = !m_nBFFramePos;
95.            }
96.        }
97.
98.        if (bEndOfSequence) {
99.            return S_OK;
100.        }
101.
102.        if (bParserFrame) {
103.            return S_OK;
104.        }
105.
106.        if (bFlush) {
107.            if (m_pParser) {
108.                m_pParser->flush(pDataBuffer, buflen);
109.            }
110.            if (m_bFFReordering) {
111.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
112.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
113.            }
114.            if (m_bFFFrameDelay) {
115.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
116.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
117.                m_nBFFramePos = !m_nBFFramePos;
118.            }
119.        }
120.
121.        if (bEndOfSequence) {
122.            return S_OK;
123.        }
124.
125.        if (bParserFrame) {
126.            return S_OK;
127.        }
128.
129.        if (bFlush) {
130.            if (m_pParser) {
131.                m_pParser->flush(pDataBuffer, buflen);
132.            }
133.            if (m_bFFReordering) {
134.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
135.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
136.            }
137.            if (m_bFFFrameDelay) {
138.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
139.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
140.                m_nBFFramePos = !m_nBFFramePos;
141.            }
142.        }
143.
144.        if (bEndOfSequence) {
145.            return S_OK;
146.        }
147.
148.        if (bParserFrame) {
149.            return S_OK;
150.        }
151.
152.        if (bFlush) {
153.            if (m_pParser) {
154.                m_pParser->flush(pDataBuffer, buflen);
155.            }
156.            if (m_bFFReordering) {
157.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
158.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
159.            }
160.            if (m_bFFFrameDelay) {
161.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
162.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
163.                m_nBFFramePos = !m_nBFFramePos;
164.            }
165.        }
166.
167.        if (bEndOfSequence) {
168.            return S_OK;
169.        }
170.
171.        if (bParserFrame) {
172.            return S_OK;
173.        }
174.
175.        if (bFlush) {
176.            if (m_pParser) {
177.                m_pParser->flush(pDataBuffer, buflen);
178.            }
179.            if (m_bFFReordering) {
180.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
181.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
182.            }
183.            if (m_bFFFrameDelay) {
184.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
185.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
186.                m_nBFFramePos = !m_nBFFramePos;
187.            }
188.        }
189.
190.        if (bEndOfSequence) {
191.            return S_OK;
192.        }
193.
194.        if (bParserFrame) {
195.            return S_OK;
196.        }
197.
198.        if (bFlush) {
199.            if (m_pParser) {
200.                m_pParser->flush(pDataBuffer, buflen);
201.            }
202.            if (m_bFFReordering) {
203.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
204.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
205.            }
206.            if (m_bFFFrameDelay) {
207.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
208.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
209.                m_nBFFramePos = !m_nBFFramePos;
210.            }
211.        }
212.
213.        if (bEndOfSequence) {
214.            return S_OK;
215.        }
216.
217.        if (bParserFrame) {
218.            return S_OK;
219.        }
220.
221.        if (bFlush) {
222.            if (m_pParser) {
223.                m_pParser->flush(pDataBuffer, buflen);
224.            }
225.            if (m_bFFReordering) {
226.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
227.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
228.            }
229.            if (m_bFFFrameDelay) {
230.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
231.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
232.                m_nBFFramePos = !m_nBFFramePos;
233.            }
234.        }
235.
236.        if (bEndOfSequence) {
237.            return S_OK;
238.        }
239.
240.        if (bParserFrame) {
241.            return S_OK;
242.        }
243.
244.        if (bFlush) {
245.            if (m_pParser) {
246.                m_pParser->flush(pDataBuffer, buflen);
247.            }
248.            if (m_bFFReordering) {
249.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
250.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
251.            }
252.            if (m_bFFFrameDelay) {
253.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
254.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
255.                m_nBFFramePos = !m_nBFFramePos;
256.            }
257.        }
258.
259.        if (bEndOfSequence) {
260.            return S_OK;
261.        }
262.
263.        if (bParserFrame) {
264.            return S_OK;
265.        }
266.
267.        if (bFlush) {
268.            if (m_pParser) {
269.                m_pParser->flush(pDataBuffer, buflen);
270.            }
271.            if (m_bFFReordering) {
272.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
273.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
274.            }
275.            if (m_bFFFrameDelay) {
276.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
277.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
278.                m_nBFFramePos = !m_nBFFramePos;
279.            }
280.        }
281.
282.        if (bEndOfSequence) {
283.            return S_OK;
284.        }
285.
286.        if (bParserFrame) {
287.            return S_OK;
288.        }
289.
290.        if (bFlush) {
291.            if (m_pParser) {
292.                m_pParser->flush(pDataBuffer, buflen);
293.            }
294.            if (m_bFFReordering) {
295.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
296.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
297.            }
298.            if (m_bFFFrameDelay) {
299.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
300.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
301.                m_nBFFramePos = !m_nBFFramePos;
302.            }
303.        }
304.
305.        if (bEndOfSequence) {
306.            return S_OK;
307.        }
308.
309.        if (bParserFrame) {
310.            return S_OK;
311.        }
312.
313.        if (bFlush) {
314.            if (m_pParser) {
315.                m_pParser->flush(pDataBuffer, buflen);
316.            }
317.            if (m_bFFReordering) {
318.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
319.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
320.            }
321.            if (m_bFFFrameDelay) {
322.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
323.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
324.                m_nBFFramePos = !m_nBFFramePos;
325.            }
326.        }
327.
328.        if (bEndOfSequence) {
329.            return S_OK;
330.        }
331.
332.        if (bParserFrame) {
333.            return S_OK;
334.        }
335.
336.        if (bFlush) {
337.            if (m_pParser) {
338.                m_pParser->flush(pDataBuffer, buflen);
339.            }
340.            if (m_bFFReordering) {
341.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
342.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
343.            }
344.            if (m_bFFFrameDelay) {
345.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
346.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
347.                m_nBFFramePos = !m_nBFFramePos;
348.            }
349.        }
350.
351.        if (bEndOfSequence) {
352.            return S_OK;
353.        }
354.
355.        if (bParserFrame) {
356.            return S_OK;
357.        }
358.
359.        if (bFlush) {
360.            if (m_pParser) {
361.                m_pParser->flush(pDataBuffer, buflen);
362.            }
363.            if (m_bFFReordering) {
364.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
365.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
366.            }
367.            if (m_bFFFrameDelay) {
368.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
369.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
370.                m_nBFFramePos = !m_nBFFramePos;
371.            }
372.        }
373.
374.        if (bEndOfSequence) {
375.            return S_OK;
376.        }
377.
378.        if (bParserFrame) {
379.            return S_OK;
380.        }
381.
382.        if (bFlush) {
383.            if (m_pParser) {
384.                m_pParser->flush(pDataBuffer, buflen);
385.            }
386.            if (m_bFFReordering) {
387.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
388.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
389.            }
390.            if (m_bFFFrameDelay) {
391.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
392.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
393.                m_nBFFramePos = !m_nBFFramePos;
394.            }
395.        }
396.
397.        if (bEndOfSequence) {
398.            return S_OK;
399.        }
400.
401.        if (bParserFrame) {
402.            return S_OK;
403.        }
404.
405.        if (bFlush) {
406.            if (m_pParser) {
407.                m_pParser->flush(pDataBuffer, buflen);
408.            }
409.            if (m_bFFReordering) {
410.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
411.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
412.            }
413.            if (m_bFFFrameDelay) {
414.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
415.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
416.                m_nBFFramePos = !m_nBFFramePos;
417.            }
418.        }
419.
420.        if (bEndOfSequence) {
421.            return S_OK;
422.        }
423.
424.        if (bParserFrame) {
425.            return S_OK;
426.        }
427.
428.        if (bFlush) {
429.            if (m_pParser) {
430.                m_pParser->flush(pDataBuffer, buflen);
431.            }
432.            if (m_bFFReordering) {
433.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
434.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
435.            }
436.            if (m_bFFFrameDelay) {
437.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
438.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
439.                m_nBFFramePos = !m_nBFFramePos;
440.            }
441.        }
442.
443.        if (bEndOfSequence) {
444.            return S_OK;
445.        }
446.
447.        if (bParserFrame) {
448.            return S_OK;
449.        }
450.
451.        if (bFlush) {
452.            if (m_pParser) {
453.                m_pParser->flush(pDataBuffer, buflen);
454.            }
455.            if (m_bFFReordering) {
456.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
457.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
458.            }
459.            if (m_bFFFrameDelay) {
460.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
461.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
462.                m_nBFFramePos = !m_nBFFramePos;
463.            }
464.        }
465.
466.        if (bEndOfSequence) {
467.            return S_OK;
468.        }
469.
470.        if (bParserFrame) {
471.            return S_OK;
472.        }
473.
474.        if (bFlush) {
475.            if (m_pParser) {
476.                m_pParser->flush(pDataBuffer, buflen);
477.            }
478.            if (m_bFFReordering) {
479.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
480.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
481.            }
482.            if (m_bFFFrameDelay) {
483.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
484.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
485.                m_nBFFramePos = !m_nBFFramePos;
486.            }
487.        }
488.
489.        if (bEndOfSequence) {
490.            return S_OK;
491.        }
492.
493.        if (bParserFrame) {
494.            return S_OK;
495.        }
496.
497.        if (bFlush) {
498.            if (m_pParser) {
499.                m_pParser->flush(pDataBuffer, buflen);
500.            }
501.            if (m_bFFReordering) {
502.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
503.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
504.            }
505.            if (m_bFFFrameDelay) {
506.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
507.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
508.                m_nBFFramePos = !m_nBFFramePos;
509.            }
510.        }
511.
512.        if (bEndOfSequence) {
513.            return S_OK;
514.        }
515.
516.        if (bParserFrame) {
517.            return S_OK;
518.        }
519.
520.        if (bFlush) {
521.            if (m_pParser) {
522.                m_pParser->flush(pDataBuffer, buflen);
523.            }
524.            if (m_bFFReordering) {
525.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
526.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
527.            }
528.            if (m_bFFFrameDelay) {
529.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
530.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
531.                m_nBFFramePos = !m_nBFFramePos;
532.            }
533.        }
534.
535.        if (bEndOfSequence) {
536.            return S_OK;
537.        }
538.
539.        if (bParserFrame) {
540.            return S_OK;
541.        }
542.
543.        if (bFlush) {
544.            if (m_pParser) {
545.                m_pParser->flush(pDataBuffer, buflen);
546.            }
547.            if (m_bFFReordering) {
548.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
549.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
550.            }
551.            if (m_bFFFrameDelay) {
552.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
553.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
554.                m_nBFFramePos = !m_nBFFramePos;
555.            }
556.        }
557.
558.        if (bEndOfSequence) {
559.            return S_OK;
560.        }
561.
562.        if (bParserFrame) {
563.            return S_OK;
564.        }
565.
566.        if (bFlush) {
567.            if (m_pParser) {
568.                m_pParser->flush(pDataBuffer, buflen);
569.            }
570.            if (m_bFFReordering) {
571.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
572.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
573.            }
574.            if (m_bFFFrameDelay) {
575.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
576.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
577.                m_nBFFramePos = !m_nBFFramePos;
578.            }
579.        }
580.
581.        if (bEndOfSequence) {
582.            return S_OK;
583.        }
584.
585.        if (bParserFrame) {
586.            return S_OK;
587.        }
588.
589.        if (bFlush) {
590.            if (m_pParser) {
591.                m_pParser->flush(pDataBuffer, buflen);
592.            }
593.            if (m_bFFReordering) {
594.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
595.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
596.            }
597.            if (m_bFFFrameDelay) {
598.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
599.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
600.                m_nBFFramePos = !m_nBFFramePos;
601.            }
602.        }
603.
604.        if (bEndOfSequence) {
605.            return S_OK;
606.        }
607.
608.        if (bParserFrame) {
609.            return S_OK;
610.        }
611.
612.        if (bFlush) {
613.            if (m_pParser) {
614.                m_pParser->flush(pDataBuffer, buflen);
615.            }
616.            if (m_bFFReordering) {
617.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
618.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
619.            }
620.            if (m_bFFFrameDelay) {
621.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
622.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
623.                m_nBFFramePos = !m_nBFFramePos;
624.            }
625.        }
626.
627.        if (bEndOfSequence) {
628.            return S_OK;
629.        }
630.
631.        if (bParserFrame) {
632.            return S_OK;
633.        }
634.
635.        if (bFlush) {
636.            if (m_pParser) {
637.                m_pParser->flush(pDataBuffer, buflen);
638.            }
639.            if (m_bFFReordering) {
640.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
641.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
642.            }
643.            if (m_bFFFrameDelay) {
644.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
645.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
646.                m_nBFFramePos = !m_nBFFramePos;
647.            }
648.        }
649.
650.        if (bEndOfSequence) {
651.            return S_OK;
652.        }
653.
654.        if (bParserFrame) {
655.            return S_OK;
656.        }
657.
658.        if (bFlush) {
659.            if (m_pParser) {
660.                m_pParser->flush(pDataBuffer, buflen);
661.            }
662.            if (m_bFFReordering) {
663.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
664.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
665.            }
666.            if (m_bFFFrameDelay) {
667.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
668.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
669.                m_nBFFramePos = !m_nBFFramePos;
670.            }
671.        }
672.
673.        if (bEndOfSequence) {
674.            return S_OK;
675.        }
676.
677.        if (bParserFrame) {
678.            return S_OK;
679.        }
680.
681.        if (bFlush) {
682.            if (m_pParser) {
683.                m_pParser->flush(pDataBuffer, buflen);
684.            }
685.            if (m_bFFReordering) {
686.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
687.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
688.            }
689.            if (m_bFFFrameDelay) {
690.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
691.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
692.                m_nBFFramePos = !m_nBFFramePos;
693.            }
694.        }
695.
696.        if (bEndOfSequence) {
697.            return S_OK;
698.        }
699.
700.        if (bParserFrame) {
701.            return S_OK;
702.        }
703.
704.        if (bFlush) {
705.            if (m_pParser) {
706.                m_pParser->flush(pDataBuffer, buflen);
707.            }
708.            if (m_bFFReordering) {
709.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
710.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
711.            }
712.            if (m_bFFFrameDelay) {
713.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
714.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
715.                m_nBFFramePos = !m_nBFFramePos;
716.            }
717.        }
718.
719.        if (bEndOfSequence) {
720.            return S_OK;
721.        }
722.
723.        if (bParserFrame) {
724.            return S_OK;
725.        }
726.
727.        if (bFlush) {
728.            if (m_pParser) {
729.                m_pParser->flush(pDataBuffer, buflen);
730.            }
731.            if (m_bFFReordering) {
732.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
733.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
734.            }
735.            if (m_bFFFrameDelay) {
736.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
737.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
738.                m_nBFFramePos = !m_nBFFramePos;
739.            }
740.        }
741.
742.        if (bEndOfSequence) {
743.            return S_OK;
744.        }
745.
746.        if (bParserFrame) {
747.            return S_OK;
748.        }
749.
750.        if (bFlush) {
751.            if (m_pParser) {
752.                m_pParser->flush(pDataBuffer, buflen);
753.            }
754.            if (m_bFFReordering) {
755.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
756.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
757.            }
758.            if (m_bFFFrameDelay) {
759.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
760.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
761.                m_nBFFramePos = !m_nBFFramePos;
762.            }
763.        }
764.
765.        if (bEndOfSequence) {
766.            return S_OK;
767.        }
768.
769.        if (bParserFrame) {
770.            return S_OK;
771.        }
772.
773.        if (bFlush) {
774.            if (m_pParser) {
775.                m_pParser->flush(pDataBuffer, buflen);
776.            }
777.            if (m_bFFReordering) {
778.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
779.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
780.            }
781.            if (m_bFFFrameDelay) {
782.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
783.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
784.                m_nBFFramePos = !m_nBFFramePos;
785.            }
786.        }
787.
788.        if (bEndOfSequence) {
789.            return S_OK;
790.        }
791.
792.        if (bParserFrame) {
793.            return S_OK;
794.        }
795.
796.        if (bFlush) {
797.            if (m_pParser) {
798.                m_pParser->flush(pDataBuffer, buflen);
799.            }
800.            if (m_bFFReordering) {
801.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
802.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
803.            }
804.            if (m_bFFFrameDelay) {
805.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
806.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
807.                m_nBFFramePos = !m_nBFFramePos;
808.            }
809.        }
810.
811.        if (bEndOfSequence) {
812.            return S_OK;
813.        }
814.
815.        if (bParserFrame) {
816.            return S_OK;
817.        }
818.
819.        if (bFlush) {
820.            if (m_pParser) {
821.                m_pParser->flush(pDataBuffer, buflen);
822.            }
823.            if (m_bFFReordering) {
824.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
825.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
826.            }
827.            if (m_bFFFrameDelay) {
828.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
829.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
830.                m_nBFFramePos = !m_nBFFramePos;
831.            }
832.        }
833.
834.        if (bEndOfSequence) {
835.            return S_OK;
836.        }
837.
838.        if (bParserFrame) {
839.            return S_OK;
840.        }
841.
842.        if (bFlush) {
843.            if (m_pParser) {
844.                m_pParser->flush(pDataBuffer, buflen);
845.            }
846.            if (m_bFFReordering) {
847.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
848.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
849.            }
850.            if (m_bFFFrameDelay) {
851.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
852.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
853.                m_nBFFramePos = !m_nBFFramePos;
854.            }
855.        }
856.
857.        if (bEndOfSequence) {
858.            return S_OK;
859.        }
860.
861.        if (bParserFrame) {
862.            return S_OK;
863.        }
864.
865.        if (bFlush) {
866.            if (m_pParser) {
867.                m_pParser->flush(pDataBuffer, buflen);
868.            }
869.            if (m_bFFReordering) {
870.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
871.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
872.            }
873.            if (m_bFFFrameDelay) {
874.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
875.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
876.                m_nBFFramePos = !m_nBFFramePos;
877.            }
878.        }
879.
880.        if (bEndOfSequence) {
881.            return S_OK;
882.        }
883.
884.        if (bParserFrame) {
885.            return S_OK;
886.        }
887.
888.        if (bFlush) {
889.            if (m_pParser) {
890.                m_pParser->flush(pDataBuffer, buflen);
891.            }
892.            if (m_bFFReordering) {
893.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
894.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
895.            }
896.            if (m_bFFFrameDelay) {
897.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
898.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
899.                m_nBFFramePos = !m_nBFFramePos;
900.            }
901.        }
902.
903.        if (bEndOfSequence) {
904.            return S_OK;
905.        }
906.
907.        if (bParserFrame) {
908.            return S_OK;
909.        }
910.
911.        if (bFlush) {
912.            if (m_pParser) {
913.                m_pParser->flush(pDataBuffer, buflen);
914.            }
915.            if (m_bFFReordering) {
916.                m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
917.                m_tcThreadBuffer[m_CurrentThread].rtStop = rtStopIn;
918.            }
919.            if (m_bFFFrameDelay) {
920.                m_tcBFFrameDelay[m_nBFFramePos].rtStart = rtStartIn;
921.                m_tcBFFrameDelay[m_nBFFramePos].rtStop = rtStopIn;
922.                m_nBFFramePos = !m_nBFFramePos;
923.            }
924.        }
925.
926.        if (bEndOfSequence) {
927.
```

```

57.         return 0;
58.     }
59. }
60. }
61.
62. while (buflen > 0 || bFlush) {
63.     REFERENCE_TIME rtStart = rtStartIn, rtStop = rtStopIn;
64.
65.     if (!bFlush) {
66.         //设置AVPacket中的数据
67.         avpkt.data = pDataBuffer;
68.         avpkt.size = buflen;
69.         avpkt.pts = rtStartIn;
70.         if (rtStartIn != AV_NOPTS_VALUE && rtStopIn != AV_NOPTS_VALUE)
71.             avpkt.duration = (int)(rtStopIn - rtStartIn);
72.         else
73.             avpkt.duration = 0;
74.         avpkt.flags = AV_PKT_FLAG_KEY;
75.
76.         if (m_bHasPalette) {
77.             m_bHasPalette = FALSE;
78.             uint32_t *pal = (uint32_t *)av_packet_new_side_data(&avpkt, AV_PKT_DATA_PALETTE, AVPALETTE_SIZE);
79.             int pal_size = FFMIN((1 << m_pAVCtx->bits_per_coded_sample) << 2, m_pAVCtx->extradata_size);
80.             uint8_t *pal_src = m_pAVCtx->extradata + m_pAVCtx->extradata_size - pal_size;
81.
82.             for (int i = 0; i < pal_size/4; i++)
83.                 pal[i] = 0xFF<<24 | AV_RL32(pal_src+4*i);
84.         }
85.     } else {
86.         avpkt.data = NULL;
87.         avpkt.size = 0;
88.     }
89.
90.     // Parse the data if a parser is present
91.     // This is mandatory for MPEG-1/2
92.     // 不一定需要
93.     if (m_pParser) {
94.         BYTE *pOut = NULL;
95.         int pOut_size = 0;
96.
97.         used_bytes = av_parser_parse2(m_pParser, m_pAVCtx, &pOut, &pOut_size, avpkt.data, avpkt.size, AV_NOPTS_VALUE, AV_NOPTS_VALUE, 0
98.
99.         if (used_bytes == 0 && pOut_size == 0 && !bFlush) {
100.             DbgLog((LOG_TRACE, 50, L"::Decode() - could not process buffer, starving?"));
101.             break;
102.         }
103.
104.         // Update start time cache
105.         // If more data was read then output, update the cache (incomplete frame)
106.         // If output is bigger, a frame was completed, update the actual rtStart with the cached value, and then overwrite the cache
107.         if (used_bytes > pOut_size) {
108.             if (rtStartIn != AV_NOPTS_VALUE)
109.                 m_rtStartCache = rtStartIn;
110.         } else if (used_bytes == pOut_size || ((used_bytes + 9) == pOut_size)) {
111.             // Why +9 above?
112.             // Well, apparently there are some broken MKV muxers that like to mux the MPEG-
113.             2 PICTURE_START_CODE block (which is 9 bytes) in the package with the previous frame
114.             // This would cause the frame timestamps to be delayed by one frame exactly, and cause timestamp reordering to go wrong.
115.             // So instead of failing on those samples, lets just assume that 9 bytes are that case exactly.
116.             m_rtStartCache = rtStartIn = AV_NOPTS_VALUE;
117.         } else if (pOut_size > used_bytes) {
118.             rtStart = m_rtStartCache;
119.             m_rtStartCache = rtStartIn;
120.             // The value was used once, don't use it for multiple frames, that ends up in weird timings
121.             rtStartIn = AV_NOPTS_VALUE;
122.         }
123.
124.         bParserFrame = (pOut_size > 0);
125.
126.         if (pOut_size > 0 || bFlush) {
127.             if (pOut && pOut_size > 0) {
128.                 if (pOut_size > m_nFFBufferSize2) {
129.                     m_nFFBufferSize2 = pOut_size;
130.                     m_pFFBuffer2 = (BYTE *)av_realloc_f(m_pFFBuffer2, m_nFFBufferSize2 + FF_INPUT_BUFFER_PADDING_SIZE, 1);
131.                     if (!m_pFFBuffer2) {
132.                         m_nFFBufferSize2 = 0;
133.                         return E_OUTOFMEMORY;
134.                     }
135.                 }
136.                 memcpy(m_pFFBuffer2, pOut, pOut_size);
137.                 memset(m_pFFBuffer2+pOut_size, 0, FF_INPUT_BUFFER_PADDING_SIZE);
138.
139.                 avpkt.data = m_pFFBuffer2;
140.                 avpkt.size = pOut_size;
141.                 avpkt.pts = rtStart;
142.                 avpkt.duration = 0;
143.
144.                 const uint8_t *eosmarker = CheckForEndOfSequence(m_nCodecId, avpkt.data, avpkt.size, &m_MpegParserState);
145.                 if (eosmarker) {
146.                     bEndOfSequence = TRUE;

```

```

147.     }
148. } else {
149.     avpkt.data = NULL;
150.     avpkt.size = 0;
151. }
152. //真正的解码
153. int ret2 = avcodec_decode_video2 (m_pAVCtx, m_pFrame, &got_picture, &avpkt);
154. if (ret2 < 0) {
155.     DbgLog((LOG_TRACE, 50, L"::Decode() - decoding failed despite successfull parsing"));
156.     got_picture = 0;
157. }
158. } else {
159.     got_picture = 0;
160. }
161. } else {
162.     used_bytes = avcodec_decode_video2 (m_pAVCtx, m_pFrame, &got_picture, &avpkt);
163. }
164.
165. if (FAILED(PostDecode())) {
166.     av_frame_unref(m_pFrame);
167.     return E_FAIL;
168. }
169.
170. // Decoding of this frame failed ... oh well!
171. if (used_bytes < 0) {
172.     av_frame_unref(m_pFrame);
173.     return S_OK;
174. }
175.
176. // When Frame Threading, we won't know how much data has been consumed, so it by default eats everything.
177. // In addition, if no data got consumed, and no picture was extracted, the frame probably isn't all that useufl.
178. // The MJPEB decoder is somewhat buggy and doesn't let us know how much data was consumed really...
179. if ((!m_pParser && (m_pAVCtx->active_thread_type & FF_THREAD_FRAME || (!got_picture && used_bytes == 0))) || m_bNoBufferConsumption || bFlush) {
180.     buflen = 0;
181. } else {
182.     buflen -= used_bytes;
183.     pDataBuffer += used_bytes;
184. }
185.
186. // Judge frame usability
187. // This determines if a frame is artifact free and can be delivered
188. // For H264 this does some wicked magic hidden away in the H264RandomAccess class
189. // MPEG-2 and VC-1 just wait for a keyframe..
190. if (m_nCodecId == AV_CODEC_ID_H264 && (bParserFrame || !m_pParser || got_picture)) {
191.     m_h264RandomAccess.judgeFrameUsability(m_pFrame, &got_picture);
192. } else if (m_bResumeAtKeyFrame) {
193.     if (m_bWaitingForKeyFrame && got_picture) {
194.         if (m_pFrame->key_frame) {
195.             DbgLog((LOG_TRACE, 50, L"::Decode() - Found Key-Frame, resuming decoding at %I64d", m_pFrame->pkt_pts));
196.             m_bWaitingForKeyFrame = FALSE;
197.         } else {
198.             got_picture = 0;
199.         }
200.     }
201. }
202.
203. // Handle B-frame delay for frame threading codecs
204. if ((m_pAVCtx->active_thread_type & FF_THREAD_FRAME) && m_bBFrameDelay) {
205.     m_tcBFrameDelay[m_nBFramePos] = m_tcThreadBuffer[m_CurrentThread];
206.     m_nBFramePos = !m_nBFramePos;
207. }
208.
209. if (!got_picture || !m_pFrame->data[0]) {
210.     if (!avpkt.size)
211.         bFlush = FALSE; // End flushing, no more frames
212.     av_frame_unref(m_pFrame);
213.     continue;
214. }
215.
216. // Determine the proper timestamps for the frame, based on different possible flags.
217. // Determine the proper timestamps for the frame, based on different possible flags.
218. // Determine the proper timestamps for the frame, based on different possible flags.
219. if (m_bFFReordering) {
220.     rtStart = m_pFrame->pkt_pts;
221.     if (m_pFrame->pkt_duration)
222.         rtStop = m_pFrame->pkt_pts + m_pFrame->pkt_duration;
223.     else
224.         rtStop = AV_NOPTS_VALUE;
225. } else if (m_bBFrameDelay && m_pAVCtx->has_b_frames) {
226.     rtStart = m_tcBFrameDelay[m_nBFramePos].rtStart;
227.     rtStop = m_tcBFrameDelay[m_nBFramePos].rtStop;
228. } else if (m_pAVCtx->active_thread_type & FF_THREAD_FRAME) {
229.     unsigned index = m_CurrentThread;
230.     rtStart = m_tcThreadBuffer[index].rtStart;
231.     rtStop = m_tcThreadBuffer[index].rtStop;
232. }
233.
234. if (m_bRVDropBFrameTimings && m_pFrame->pict_type == AV_PICTURE_TYPE_B) {
235.     rtStart = AV_NOPTS_VALUE;
236. }

```



```

237.
238.     if (m_bCalculateStopTime)
239.         rtStop = AV_NOPTS_VALUE;
240.
241.     ////////////////////////////////////////////
242.     // All required values collected, deliver the frame
243.     ////////////////////////////////////////////
244.     LAVFrame *pOutFrame = NULL;
245.     AllocateFrame(&pOutFrame);
246.
247.     AVRational display_aspect_ratio;
248.     int64_t num = (int64_t)m_pFrame->sample_aspect_ratio.num * m_pFrame->width;
249.     int64_t den = (int64_t)m_pFrame->sample_aspect_ratio.den * m_pFrame->height;
250.     av_reduce(&display_aspect_ratio.num, &display_aspect_ratio.den, num, den, 1 << 30);
251.
252.     pOutFrame->width      = m_pFrame->width;
253.     pOutFrame->height     = m_pFrame->height;
254.     pOutFrame->aspect_ratio = display_aspect_ratio;
255.     pOutFrame->repeat     = m_pFrame->repeat_pict;
256.     pOutFrame->key_frame   = m_pFrame->key_frame;
257.     pOutFrame->frame_type  = av_get_picture_type_char(m_pFrame->pict_type);
258.     pOutFrame->ext_format  = GetDXVA2ExtendedFlags(m_pAVCtx, m_pFrame);
259.
260.     if (m_pFrame->interlaced_frame || (!m_pAVCtx->
>progressive_sequence && (m_nCodecId == AV_CODEC_ID_H264 || m_nCodecId == AV_CODEC_ID_MPEG2VIDEO)))
261.         m_iInterlaced = 1;
262.     else if (m_pAVCtx->progressive_sequence)
263.         m_iInterlaced = 0;
264.
265.     pOutFrame->interlaced = (m_pFrame->interlaced_frame || (m_iInterlaced == 1 && m_pSettings->
>GetDeinterlacingMode() == DeintMode_Aggressive) || m_pSettings->GetDeinterlacingMode() == DeintMode_Force) && !(m_pSettings->GetDei
nterlacingMode() == DeintMode_Disable);
266.
267.     LAVDeintFieldOrder fo = m_pSettings->GetDeintFieldOrder();
268.     pOutFrame->tff        = (fo == DeintFieldOrder_Auto) ? m_pFrame->top_field_first : (fo == DeintFieldOrder_TopFieldFirst);
269.
270.     pOutFrame->rtStart    = rtStart;
271.     pOutFrame->rtStop     = rtStop;
272.
273.     PixelFormatMapping map = getPixFmtMapping((AVPixelFormat)m_pFrame->format);
274.     pOutFrame->format      = map.lavpixfmt;
275.     pOutFrame->bpp         = map.bpp;
276.
277.     if (m_nCodecId == AV_CODEC_ID_MPEG2VIDEO || m_nCodecId == AV_CODEC_ID_MPEG1VIDEO)
278.         pOutFrame->avgFrameDuration = GetFrameDuration();
279.
280.     if (map.conversion) {
281.         ConvertPixFmt(m_pFrame, pOutFrame);
282.     } else {
283.         for (int i = 0; i < 4; i++) {
284.             pOutFrame->data[i] = m_pFrame->data[i];
285.             pOutFrame->stride[i] = m_pFrame->linesize[i];
286.         }
287.
288.         pOutFrame->priv_data = av_frame_alloc();
289.         av_frame_ref((AVFrame *)pOutFrame->priv_data, m_pFrame);
290.         pOutFrame->destruct = lav_avframe_free;
291.     }
292.
293.     if (bEndOfSequence)
294.         pOutFrame->flags |= LAV_FRAME_FLAG_END_OF_SEQUENCE;
295.
296.     if (pOutFrame->format == LAVPixFmt_DXVA2) {
297.         pOutFrame->data[0] = m_pFrame->data[4];
298.         HandleDXVA2Frame(pOutFrame);
299.     } else {
300.         Deliver(pOutFrame);
301.     }
302.
303.     if (bEndOfSequence) {
304.         bEndOfSequence = FALSE;
305.         if (pOutFrame->format == LAVPixFmt_DXVA2) {
306.             HandleDXVA2Frame(m_pCallback->GetFlushFrame());
307.         } else {
308.             Deliver(m_pCallback->GetFlushFrame());
309.         }
310.     }
311.
312.     if (bFlush) {
313.         m_CurrentThread = (m_CurrentThread + 1) % m_pAVCtx->thread_count;
314.     }
315.     av_frame_unref(m_pFrame);
316. }
317.
318. return S_OK;
319. }

```

终于，我们从这个函数中看到了很多的ffmpeg的API，结构体，以及变量。比如解码视频的函数avcodec_decode_video2()。

解码器初始化函数：InitDecoder()

```
[cpp]
1. //创建解码器
2. STDMETHODIMP CDecAvcodec::InitDecoder(AVCodecID codec, const CMediaType *pmt)
3. {
4.     //要是有的话，先销毁
5.     DestroyDecoder();
6.     DbgLog((LOG_TRACE, 10, L"Initializing ffmpeg for codec %S", avcodec_get_name(codec)));
7.
8.     BITMAPINFOHEADER *pBMI = NULL;
9.     videoFormatTypeHandler((const BYTE *)pmt->Format(), pmt->FormatType(), &pBMI);
10.    //查找解码器
11.    m_pAVCodec = avcodec_find_decoder(codec);
12.    CheckPointer(m_pAVCodec, VFW_E_UNSUPPORTED_VIDEO);
13.    //初始化上下文环境
14.    m_pAVCtx = avcodec_alloc_context3(m_pAVCodec);
15.    CheckPointer(m_pAVCtx, E_POINTER);
16.
17.    if(codec == AV_CODEC_ID_MPEG1VIDEO || codec == AV_CODEC_ID_MPEG2VIDEO || pmt->subtype == FOURCCMap(MKTAG('H','2','6','4')) || pmt-
>subtype == FOURCCMap(MKTAG('h','2','6','4')) {
18.        m_pParser = av_parser_init(codec);
19.    }
20.
21.    DWORD dwDecFlags = m_pCallback->GetDecodeFlags();
22.
23.    LONG biRealWidth = pBMI->biWidth, biRealHeight = pBMI->biHeight;
24.    if (pmt->formattype == FORMAT_VideoInfo || pmt->formattype == FORMAT_MPEGVideo) {
25.        VIDEOINFOHEADER *vih = (VIDEOINFOHEADER *)pmt->Format();
26.        if (vih->rcTarget.right != 0 && vih->rcTarget.bottom != 0) {
27.            biRealWidth = vih->rcTarget.right;
28.            biRealHeight = vih->rcTarget.bottom;
29.        }
30.    } else if (pmt->formattype == FORMAT_VideoInfo2 || pmt->formattype == FORMAT_MPEG2Video) {
31.        VIDEOINFOHEADER2 *vih2 = (VIDEOINFOHEADER2 *)pmt->Format();
32.        if (vih2->rcTarget.right != 0 && vih2->rcTarget.bottom != 0) {
33.            biRealWidth = vih2->rcTarget.right;
34.            biRealHeight = vih2->rcTarget.bottom;
35.        }
36.    }
37.    //各种赋值
38.    m_pAVCtx->codec_id = codec;
39.    m_pAVCtx->codec_tag = pBMI->biCompression;
40.    m_pAVCtx->coded_width = pBMI->biWidth;
41.    m_pAVCtx->coded_height = abs(pBMI->biHeight);
42.    m_pAVCtx->bits_per_coded_sample = pBMI->biBitCount;
43.    m_pAVCtx->error_concealment = FF_EC_GUESS_MVS | FF_EC_DEBLOCK;
44.    m_pAVCtx->err_recognition = AV_EF_CAREFUL;
45.    m_pAVCtx->workaround_bugs = FF_BUG_AUTODETECT;
46.    m_pAVCtx->refcounted_frames = 1;
47.
48.    if (codec == AV_CODEC_ID_H264)
49.        m_pAVCtx->flags2 |= CODEC_FLAG2_SHOW_ALL;
50.
51.    // Setup threading
52.    int thread_type = getThreadFlags(codec);
53.    if (thread_type) {
54.        // Thread Count. 0 = auto detect
55.        int thread_count = m_pSettings->GetNumThreads();
56.        if (thread_count == 0) {
57.            thread_count = av_cpu_count() * 3 / 2;
58.        }
59.
60.        m_pAVCtx->thread_count = max(1, min(thread_count, AVCODEC_MAX_THREADS));
61.        m_pAVCtx->thread_type = thread_type;
62.    } else {
63.        m_pAVCtx->thread_count = 1;
64.    }
65.
66.    if (dwDecFlags & LAV_VIDEO_DEC_FLAG_NO_MT) {
67.        m_pAVCtx->thread_count = 1;
68.    }
69.    //初始化AVFrame
70.    m_pFrame = av_frame_alloc();
71.    CheckPointer(m_pFrame, E_POINTER);
72.
73.    m_h264RandomAccess.SetAVCNALSize(0);
74.
75.    // Process Extradata
76.    //处理Extradata
77.    BYTE *extra = NULL;
78.    size_t extralen = 0;
79.    getExtradata(*pmt, NULL, &extralen);
80.
81.    BOOL bH264avc = FALSE;
82.    if (extralen > 0) {
83.        DbgLog((LOG_TRACE, 10, L"-> Processing extradata of %d bytes", extralen));
84.        // Reconstruct AVC1 extradata format
85.        if (pmt->formattype == FORMAT_MPEG2Video && (m_pAVCtx->codec_tag == MAKEFOURCC('a','v','c','1') || m_pAVCtx-
>codec_tag == MAKEFOURCC('A','V','C','1') || m_pAVCtx->codec_tag == MAKEFOURCC('C','C','V','1')) {
```

```

86.     MPEG2VIDEOINFO *mp2vi = (MPEG2VIDEOINFO *)pmt->Format();
87.     extralen += 7;
88.     extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
89.     extra[0] = 1;
90.     extra[1] = (BYTE)mp2vi->dwProfile;
91.     extra[2] = 0;
92.     extra[3] = (BYTE)mp2vi->dwLevel;
93.     extra[4] = (BYTE)(mp2vi->dwFlags ? mp2vi->dwFlags : 4) - 1;
94.
95.     // Actually copy the metadata into our new buffer
96.     size_t actual_len;
97.     getExtraData(*pmt, extra+6, &actual_len);
98.
99.     // Count the number of SPS/PPS in them and set the length
100.    // We'll put them all into one block and add a second block with 0 elements afterwards
101.    // The parsing logic does not care what type they are, it just expects 2 blocks.
102.    BYTE *p = extra+6, *end = extra+6+actual_len;
103.    BOOL bSPS = FALSE, bPPS = FALSE;
104.    int count = 0;
105.    while (p+1 < end) {
106.        unsigned len = (((unsigned)p[0] << 8) | p[1]) + 2;
107.        if (p + len > end) {
108.            break;
109.        }
110.        if ((p[2] & 0x1F) == 7)
111.            bSPS = TRUE;
112.        if ((p[2] & 0x1F) == 8)
113.            bPPS = TRUE;
114.        count++;
115.        p += len;
116.    }
117.    extra[5] = count;
118.    extra[extralen-1] = 0;
119.
120.    bH264avc = TRUE;
121.    m_h264RandomAccess.SetAVCSize(mp2vi->dwFlags);
122. } else if (pmt->subtype == MEDIASUBTYPE_LAV_RAWVIDEO) {
123.     if (extralen < sizeof(m_pAVCtx->pix_fmt)) {
124.         DbgLog((LOG_TRACE, 10, L"-> LAV RAW Video extradata is missing.."));
125.     } else {
126.         extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
127.         getExtraData(*pmt, extra, NULL);
128.         m_pAVCtx->pix_fmt = *(AVPixelFormat *)extra;
129.         extralen -= sizeof(AVPixelFormat);
130.         memmove(extra, extra+sizeof(AVPixelFormat), extralen);
131.     }
132. } else {
133.     // Just copy extradata for other formats
134.     extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
135.     getExtraData(*pmt, extra, NULL);
136. }
137. // Hack to discard invalid MP4 metadata with AnnexB style video
138. if (codec == AV_CODEC_ID_H264 && !bH264avc && extra[0] == 1) {
139.     av_freep(&extra);
140.     extralen = 0;
141. }
142. m_pAVCtx->extradata = extra;
143. m_pAVCtx->extradata_size = (int)extralen;
144. } else {
145.     if (codec == AV_CODEC_ID_VP6 || codec == AV_CODEC_ID_VP6A || codec == AV_CODEC_ID_VP6F) {
146.         int cropH = pBMI->biWidth - biRealWidth;
147.         int cropV = pBMI->biHeight - biRealHeight;
148.         if (cropH >= 0 && cropH <= 0x0f && cropV >= 0 && cropV <= 0x0f) {
149.             m_pAVCtx->extradata = (uint8_t *)av_mallocz(1 + FF_INPUT_BUFFER_PADDING_SIZE);
150.             m_pAVCtx->extradata_size = 1;
151.             m_pAVCtx->extradata[0] = (cropH << 4) | cropV;
152.         }
153.     }
154. }
155.
156. m_h264RandomAccess.flush(m_pAVCtx->thread_count);
157. m_CurrentThread = 0;
158. m_rtStartCache = AV_NOPTS_VALUE;
159.
160. LAVPinInfo lavPinInfo = {0};
161. BOOL bLAVInfoValid = SUCCEEDED(m_pCallback->GetLAVPinInfo(lavPinInfo));
162.
163. m_bInputPadded = dwDecFlags & LAV_VIDEO_DEC_FLAG_LAVSPLITTER;
164.
165. // Setup codec-specific timing logic
166. BOOL bVC1IsPTS = (codec == AV_CODEC_ID_VC1 && !(dwDecFlags & LAV_VIDEO_DEC_FLAG_VC1_DTS));
167.
168. // Use ffmpegs logic to reorder timestamps
169. // This is required for H264 content (except AVI), and generally all codecs that use frame threading
170. // VC-1 is also a special case. Its required for splitters that deliver PTS timestamps (see bVC1IsPTS above)
171. m_bFFReordering = ( codec == AV_CODEC_ID_H264 && !(dwDecFlags & LAV_VIDEO_DEC_FLAG_H264_AVI))
172.                  || codec == AV_CODEC_ID_VP8
173.                  || codec == AV_CODEC_ID_VP3
174.                  || codec == AV_CODEC_ID_THEORA
175.                  || codec == AV_CODEC_ID_HUFFYUV
176.                  || codec == AV_CODEC_ID_FFHUFF

```

```

177.         || codec == AV_CODEC_ID_MPEG2VIDEO
178.         || codec == AV_CODEC_ID_MPEG1VIDEO
179.         || codec == AV_CODEC_ID_DIRAC
180.         || codec == AV_CODEC_ID_UTVIDEO
181.         || codec == AV_CODEC_ID_DNXHD
182.         || codec == AV_CODEC_ID_JPEG2000
183.         || (codec == AV_CODEC_ID_MPEG4 && pmt->formattype == FORMAT_MPEG2Video)
184.         || bVC1IsPTS;
185.
186.     // Stop time is unreliable, drop it and calculate it
187.     m_bCalculateStopTime = (codec == AV_CODEC_ID_H264 || codec == AV_CODEC_ID_DIRAC || (codec == AV_CODEC_ID_MPEG4 && pmt->formattype
188.     e == FORMAT_MPEG2Video) || bVC1IsPTS);
189.
190.     // Real Video content has some odd timestamps
191.     // LAV Splitter does them allright with RV30/RV40, everything else screws them up
192.     m_bRVDropBFrameTimings = (codec == AV_CODEC_ID_RV10 || codec == AV_CODEC_ID_RV20 || ((codec == AV_CODEC_ID_RV30 || codec == AV_CODEC_ID_RV40) && !(dwDecFlags & LAV_VIDEO_DEC_FLAG_LAVSPLITTER) || (bLAVInfoValid && (lavPinInfo.flags & LAV_STREAM_FLAG_RV34_MKV))));
193.
194.     // Enable B-Frame delay handling
195.     m_bBFrameDelay = !m_bFFReordering && !m_bRVDropBFrameTimings;
196.
197.     m_bWaitingForKeyFrame = TRUE;
198.     m_bResumeAtKeyFrame =
199.         || codec == AV_CODEC_ID_MPEG2VIDEO
200.         || codec == AV_CODEC_ID_VC1
201.         || codec == AV_CODEC_ID_RV30
202.         || codec == AV_CODEC_ID_RV40
203.         || codec == AV_CODEC_ID_VP3
204.         || codec == AV_CODEC_ID_THEORA
205.         || codec == AV_CODEC_ID_MPEG4;
206.
207.     m_bNoBufferConsumption =
208.         || codec == AV_CODEC_ID_MJPEGB
209.         || codec == AV_CODEC_ID_LOCO
210.         || codec == AV_CODEC_ID_JPEG2000;
211.
212.     m_bHasPalette = m_pAVCtx->bits_per_coded_sample <= 8 && m_pAVCtx->extradata_size && !
213.     (dwDecFlags & LAV_VIDEO_DEC_FLAG_LAVSPLITTER)
214.         && (codec == AV_CODEC_ID_MSVIDEO1
215.         || codec == AV_CODEC_ID_MSRLE
216.         || codec == AV_CODEC_ID_CINEPAK
217.         || codec == AV_CODEC_ID_8BPS
218.         || codec == AV_CODEC_ID_QPEG
219.         || codec == AV_CODEC_ID_QTRLE
220.         || codec == AV_CODEC_ID_TSCC);
221.
222.     if (FAILED(AdditionalDecoderInit())) {
223.         return E_FAIL;
224.     }
225.
226.     if (bLAVInfoValid) {
227.         // Setting has_b_frames to a proper value will ensure smoother decoding of H264
228.         if (lavPinInfo.has_b_frames >= 0) {
229.             DbgLog((LOG_TRACE, 10, L"-> Setting has_b_frames to %d", lavPinInfo.has_b_frames));
230.             m_pAVCtx->has_b_frames = lavPinInfo.has_b_frames;
231.         }
232.     }
233.
234.     // Open the decoder
235.     //打开解码器
236.     int ret = avcodec_open2(m_pAVCtx, m_pAVCodec, NULL);
237.     if (ret >= 0) {
238.         DbgLog((LOG_TRACE, 10, L"-> ffmpeg codec opened successfully (ret: %d)", ret));
239.         m_nCodecId = codec;
240.     } else {
241.         DbgLog((LOG_TRACE, 10, L"-> ffmpeg codec failed to open (ret: %d)", ret));
242.         DestroyDecoder();
243.         return VFW_E_UNSUPPORTED_VIDEO;
244.     }
245.
246.     m_iInterlaced = 0;
247.     for (int i = 0; i < countof(ff_interlace_capable); i++) {
248.         if (codec == ff_interlace_capable[i]) {
249.             m_iInterlaced = -1;
250.             break;
251.         }
252.     }
253.
254.     // Detect chroma and interlaced
255.     if (m_pAVCtx->extradata && m_pAVCtx->extradata_size) {
256.         if (codec == AV_CODEC_ID_MPEG2VIDEO) {
257.             CMPEG2HeaderParser mpeg2Parser(extra, extralen);
258.             if (mpeg2Parser.hdr.valid) {
259.                 if (mpeg2Parser.hdr.chroma < 2) {
260.                     m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV420P;
261.                 } else if (mpeg2Parser.hdr.chroma == 2) {
262.                     m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV422P;
263.                 }
264.                 m_iInterlaced = mpeg2Parser.hdr.interlaced;
265.             }
266.         } else if (codec == AV_CODEC_ID_H264) {
267.             CH264SequenceParser h264parser;
268.             if (h264parser)

```

```

205.         nzb4parser.ParseNALs(extra+b, extralen-b, 4);
266.     else
267.         h264parser.ParseNALs(extra, extralen, 0);
268.     if (h264parser.sps.valid)
269.         m_iInterlaced = h264parser.sps.interlaced;
270. } else if (codec == AV_CODEC_ID_VC1) {
271.     VC1HeaderParser vc1parser(extra, extralen);
272.     if (vc1parser.hdr.valid)
273.         m_iInterlaced = (vc1parser.hdr.interlaced ? -1 : 0);
274. }
275. }
276.
277. if (codec == AV_CODEC_ID_DNXHD)
278.     m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV422P10;
279. else if (codec == AV_CODEC_ID_FRAPS)
280.     m_pAVCtx->pix_fmt = AV_PIX_FMT_BGR24;
281.
282. if (bLAVInfoValid && codec != AV_CODEC_ID_FRAPS && m_pAVCtx->pix_fmt != AV_PIX_FMT_DXVA2_VLD)
283.     m_pAVCtx->pix_fmt = lavPinInfo.pix_fmt;
284.
285. DbgLog((LOG_TRACE, 10, L"AVCodec init successfull. interlaced: %d", m_iInterlaced));
286.
287. return S_OK;
288. }

```

解码器销毁函数：DestroyDecoder()

```

[cpp]
1. //销毁解码器, 各种Free
2. STDMETHODIMP CDecAvcodec::DestroyDecoder()
3. {
4.     DbgLog((LOG_TRACE, 10, L"Shutting down ffmpeg..."));
5.     m_pAVCodec = NULL;
6.
7.     if (m_pParser) {
8.         av_parser_close(m_pParser);
9.         m_pParser = NULL;
10.    }
11.
12.    if (m_pAVCtx) {
13.        avcodec_close(m_pAVCtx);
14.        av_freep(&m_pAVCtx->extradata);
15.        av_freep(&m_pAVCtx);
16.    }
17.    av_frame_free(&m_pFrame);
18.
19.    av_freep(&m_pFFBuffer);
20.    m_nFFBufferSize = 0;
21.
22.    av_freep(&m_pFFBuffer2);
23.    m_nFFBufferSize2 = 0;
24.
25.    if (m_pSwsContext) {
26.        sws_freeContext(m_pSwsContext);
27.        m_pSwsContext = NULL;
28.    }
29.
30.    m_nCodecId = AV_CODEC_ID_NONE;
31.
32.    return S_OK;
33. }

```

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/13272409>

文章标签：[LAVFilter](#) [源代码](#) [解码](#) [ffmpeg](#) [directshow](#)

个人分类：[LAV Filter](#)

所属专栏：[开源多媒体项目源代码分析](#)

此PDF由spyyg生成, 请尊重原作者版权!!!

我的邮箱: liushidc@163.com