HARDWARE DECODER INTEGRATION GUIDE

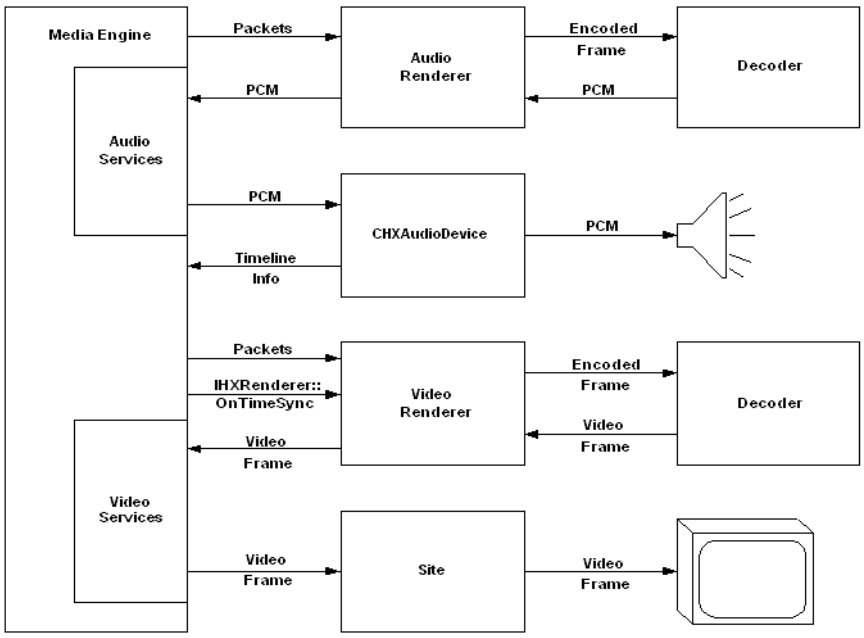
This document describes recommended ways to integrate hardware decoding of audio and/or video with the Helix media engine. Hardware decoders come in various forms, the most common being digital signal processors (DSPs). The intended audience for this document is developers implementing hardware designs in association with the Helix Embedded Dev Kit v1.1.

> **For More Information:** To learn more about the Helix Embedded Dev Kit v1.1, see **https://helix-client.helixcommunity.org/2004/branches**.

Overview of Client Engine

The Helix media engine was designed to handle multiple simultaneous audio and video streams. The streams can share a common timeline or groups of these streams can run on independent timelines. An overview of the data flow and layout of the different components of the media engine is shown in the following figure.

Figure 1: Default Helix Media Engine Configuration



As can be seen in Figure 1, the current media engine design has the audio decoders passing the decoded PCM data back to the renderer. The renderer then passes the PCM data for that one stream to the audio services module for mixing and resampling with any other audio streams that are present in this presentation. After all streams are mixed, the resulting PCM data is then passed on to the audio device module for rendering to the actual hardware.

It is important to note that the audio device is the master source of the presentation timeline. As audio is played through the device, the `CHXAudioDevice` class keeps track of how many bytes have been played. It is the responsibility of the `CHXAudioDevice` class to periodically inform the media engine where the timeline is at by converting the number of bytes that have played to milliseconds using the current sample rate, bits per sample, and number of channels. Once the media engine is informed of the current playback time, it will then, in turn, inform all renderers in the system what the current timeline is using `IHXRenderer::OnTimeSync` calls. Visual renderers use this information to determine what frame of video should be shown at that given moment.

Audio Hardware Decoder Integration

There are many different ways to integrate hardware decoders in the Helix media engine depending on what functionality they provide and what type of data you are decoding. For example, some hardware decoder setups can provide information back into the media engine while others cannot, or you may be decoding video in the hardware decoder as opposed to audio.

The simplest scenario for integrating hardware decoders would be to provide hardware decoding of audio and then passing the decoded data directly back to the renderer. The more typical scenario for hardware decoder integration, however, is one where encoded data is passed into the hardware decoder and it never comes back out, but instead is passed directly to the hardware audio device. These audio hardware decoder scenarios are described in the following sections.

Audio Decode Only — Multiple Audio Streams With or Without Video

As you can see from the discussion in the overview and Figure 1, providing hardware decoding of audio and passing the decoded data back to the software renderer very closely mimics what is already being done in software. Your hardware decoder simply replaces the software decoder and nothing else has to change. You will need to properly interface your hardware decoder with the renderer to pass in setup information, but that is about all that needs to be done. Video synchronization is automatic in this case because, again, nothing is really changing in the system except that the decoding is being done in hardware. This scenario also handles the case of multiple audio streams because the media engine gets the decoded PCM data back and is able to do all of the mixing itself.

Audio Decode and Render — Single Audio Stream With or Without Video

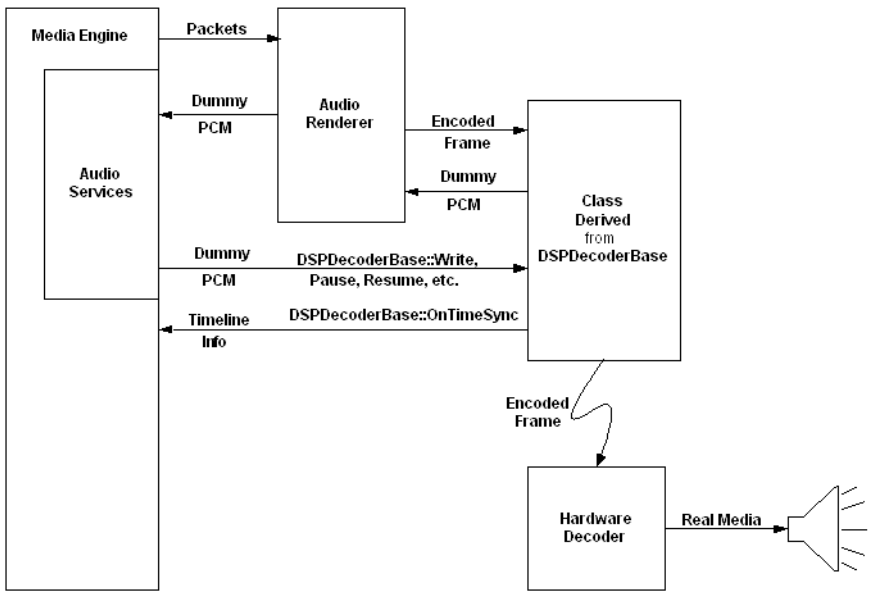
A few more issues need to be addressed in set ups where the hardware decoder decodes the audio data and passes it directly to an onboard audio output device (such as a speaker). The first is that the system is designed such that the renderers need PCM data to pass to the rest of the media engine. The typical solution to this issue is to pass back dummy PCM data (dummy PCM data must be all zeros so that any mixing that occurs with other streams won't be affected). The reason this works is that since your hardware decoder is outputting the actual sound, the media engine can just throw away the PCM data

when it reaches the end of the pipeline. Passing the dummy PCM data will keep the rest of the media engine moving along with the proper pacing. In cases where there are multiple streams (not too likely in typical hardware decoder setups), passing zeros for the dummy PCM data allows that data to be safely mixed with other audio streams. Also, it is important to pass back the correct amount of dummy PCM data. Both the renderer and audio services expects a certain amount of PCM data given a certain amount of encoded data.

In set ups where you decode and render the encoded data, another issue is that of audio timeline generation and audio state information. As mentioned in the overview, `CHXAudioDevice` is the source of the master timeline. The problem arises when the hardware decoder starts sending the decoded PCM data directly to the speaker bypassing `CHXAudioDevice`. `CHXAudioDevice` receives the proper amount of PCM dummy data but does not know how fast the actual audio data is being played. One solution is to change `CHXAudioDevice` to use wall clock time instead of bytes played to generate the timeline. While this works for short-to-medium duration clips, it can lead to audio/video synchronization drift over time. Also, there is the issue of commands like pause and resume being issued on `CHXAudioDevice`, but the actual pumping of the data is being done in a separate module; the hardware decoder running under the renderer. To solve both of these issues, a new class was introduced from which your hardware decoder implementation can derive, `DSPDecoderBase`.

What `DSPDecoderBase` does when instantiated is to try to replace the built-in audio device in the media engine, `CHXAudioDevice`, with itself. Once the default audio device is replaced, `DSPDecoderBase` then starts to receive all of the commands like pause and resume. In addition, it brings the source of the timeline, now `DSPDecoderBase`, and the consumption of the PCM data into the same place. By default, `DSPDecoderBase` uses wall-clock time for the timeline, but the hooks are in place so that any class deriving from it can use the actual bytes consumed to drive the timeline if that information is available. Using this set up, we now have a configuration as show in the following figure.

Figure 2: Audio-only Hardware Decoder Configuration



More information on exactly how to use DSPDecoderBase can be found in "How to Use DSPDecoderBase" on page 6. However, to sum up, you will need to write a thin client-side wrapper around the API you use to communicate with your hardware decoder. The client-side wrapper will derive from DSPDecoderBase and will be what the renderer directly communicates with to decode data. The only methods that need to be added are those directly related to the interface between the renderer and decoder, such as those needed to initialize the hardware decoder and pass in encoded data. A sample for WAV playback is presented in "How to Use DSPDecoderBase" on page 6.

Video synchronization isn't a problem in this scenario as long as the timeline information matches the audio actually being played.

Audio Decode and Render — Multiple Audio Streams With or Without Video

In cases where multiple audio streams are present in a given presentation, things get more complicated. Since you have at least one hardware decoder present, you will have the situation above where a hardware decoder has derived from DSPDecoderBase and is now acting as the default audio device.

Let's first look at the case where all the other audio streams are decoded in software or with a hardware decoder that decodes only (that is, it passes back the decoded PCM data to the renderer). The media engine will be receiving all the decoded PCM data from all other audio streams, mixing and resampling it, then sending the final PCM data mix to DSPDecoderBase::Write. In this case it would then be the responsibility of the derived class (the hardware decoder API wrapper) to render this PCM data to the hardware's audio output device. The only issue here is the synchronization of the PCM data coming in through DSPDecoderBase::Write and the PCM data that the hardware decoder is decoding. For more information on using DSPDecoderBase, see "How to Use DSPDecoderBase" on page 6.

Let's now consider the case where there are exactly two audio streams, both being decoded by hardware decoders that render the decoded PCM data directly to an onboard audio output device. Just as in the case with a single audio stream, you will want to have both of those hardware decoder client-side API wrappers be derived from DSPDecoderBase. For example, assume you have a stream that has a AAC audio stream and a AMR-NB stream, and that you have hardware decoders for both. There is no guarantee which stream will be initialized first, and therefore no way to tell which decoder will ask the core to replace the audio device first. After the audio device is replaced, it is no longer possible to replace it again. So, when the second decoder derived from DSPDecoderBase gets initialized, it will fail to replace the audio device.

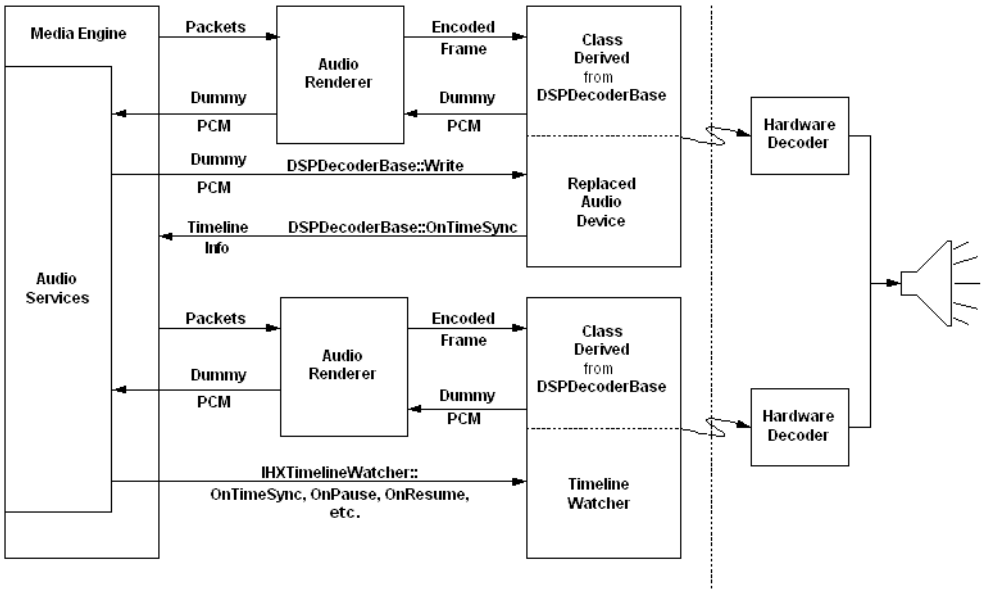
When this happens, DSPDecoderBase registers itself with the media engine as an IHXTimelineWatcher. This interface is listed in the following sample:

```
DECLARE_INTERFACE_(IHXTimelineWatcher, IUnknown)
{
   STDMETHOD(QueryInterface) (THIS_ REFIID riid, void** ppvObj) PURE;
   STDMETHOD_(ULONG32,AddRef) (THIS) PURE;
   STDMETHOD_(ULONG32,Release) (THIS) PURE;

   STDMETHOD(OnPause) (THIS) PURE;
   STDMETHOD(OnResume) (THIS) PURE;
   STDMETHOD(OnClose) (THIS) PURE;
   STDMETHOD(OnTimeSync) (THIS_ UINT32 currentTime ) PURE;
};
```

IHXTimelineWatcher provides callbacks from the media engine when that timeline state changes. These include pause events, resume events, closing events, and current timeline information. So, even though the second decoder could not replace the audio device, it will still receive notifications of important events that it can use to pace the decode and be in sync with the other streams. A diagram of this is shown in the following figure.

Figure 3: Multiple Hardware Decoders



If this is generalized to multiple audio streams, with some being decoded in software and others in hardware decoders, the very first DSPDecoderBase object that gets initialized is responsible for generating the timeline and mixing in any PCM data that comes from the software decoders through the IHXAudioDevice::Write call. Any additional DSPDecoderBase objects will need to pace their decoding using the IHXTimelineWatcher callbacks.

Video DSP Integration

Decoding of video with hardware decoders is a bit simpler due to the lack of any need to generate timeline information. There are in fact only two different scenarios: video decode only or video decode and display.

Video Decode Only

When you are only decoding the video frames and not displaying them, the situation is similar to the audio decode only case described in “Audio Decode Only — Multiple Audio Streams With or Without Video” on page 2. Since you going to hand back the raw I420 frames (in the case of RealVideo) just as the software decoder does, there is really no change to the media engine needed at all. Again, the only work needed is to write a layer that goes between the renderer and your hardware decoder API that conforms to the existing decoder API the renderer expects.

Video Decode and Display

If you are decoding and then displaying directly from the hardware decoder, you must pace the presentation of your video frames and make them synchronize with the audio stream. The way to do this is the same as the case of multiple audio streams; just register as a IHXTimelineWatcher with the media engine. Once you do this you will get callbacks informing you of a pause, resume, and the current playback time. You can then either pass that information to the hardware decoder so it knows what frame to display, or just send the encoded video frame for immediate display to the hardware decoder as the frame is needed. It just depends on how your hardware decoder is implemented.

DSPDecoderBase handles registering as a IHXTimelineWatcher for audio decoders. For video you should do this work yourself. You only need to inherit from IHXTimelineWatcher and then, while being initialized, query the media engine for IHXTimelineManager and use it to add yourself as a watcher. The following example shows some sample code on how to do this:

```
m_pContext->QueryInterface( IID_IHXTimelineManger,(void**)&m_pTimeLineManager);
if( m_pTimeLineManager )
{
   m_pTimeLineManager->AddTimelineWatcher((IHXTimelineWatcher*)this);
}
```

Be sure to remove yourself as a timeline watcher on shut down.

How to Use DSPDecoderBase

The source code for DSPDecoderBase is located in datatype/common/dspcodec. The header file information for DSPDecoderBase is listed here for reference:

```
class DSPDecoderBase: public IHXAudioDevice
              , public IHXCallback
              , public IHXTimelineWatcher
{
 public:
   DSPDecoderBase();
   ~DSPDecoderBase();

   //The renderer needs to pass in a client engine context.
   void Init(IUnknown* pContext);

   //IHXAudioDevice methods (to replace built in one)
   STDMETHOD(QueryInterface)  (REFIID riid, void** ppvObj);
   STDMETHOD_(ULONG32,AddRef) ();
   STDMETHOD_(ULONG32,Release) ();
   STDMETHOD(Open)          (const HXAudioFormat* pAudioFormat,
                       IHXAudioDeviceResponse* pDeviceResponse );
   STDMETHOD(Close)         (const BOOL bFlush );
   STDMETHOD(Pause)         ();
   STDMETHOD(Resume)         ();
   STDMETHOD(Write)         (const HXAudioData* pAudioData);
   STDMETHOD_(BOOL,InitVolume) (const UINT16 uMinVolume, const UINT16 uMaxVolume);
   STDMETHOD(SetVolume)      (const UINT16 uVolume);
```

```
    STDMETHOD_(UINT16,GetVolume)();
    STDMETHOD(Reset)         ();
    STDMETHOD(Drain)         ();
    STDMETHOD(CheckFormat)     (const HXAudioFormat* pAudioFormat);
    STDMETHOD(GetCurrentAudioTime) ( REF(ULONG32) ulCurrentTime);

    //IHXCallback
    STDMETHOD(Func) (THIS);

    //IHXTimelineWatcher methods.
    STDMETHOD(OnPause)   ();
    STDMETHOD(OnResume)  ();
    STDMETHOD(OnClose)   ();
    STDMETHOD(OnTimeSync) (UINT32 currentTime );

  protected:

    IUnknown*           m_pContext;
    IHXCommonClassFactory*  m_pCommonClassFactory;
    ULONG32             m_ulRefCount;
    ULONG32             m_ulCurrentTime;
    ULONG32             m_ulPauseTime;
    IHXAudioDeviceResponse* m_pDeviceResponse;
    IHXScheduler*        m_pScheduler;
    ULONG32             m_ulCallbackHandle;
    IHXTimelineManager*    m_pTimeLineManager;
    UINT16              m_nMinVolume;
    UINT16              m_nMaxVolume;
};
```

The first step to integrating a hardware decoder is to look at the source code for the renderer that is going to be using it. You will need to look at that renderers interface with its codec and make sure that the class you write provides the same interface needed by the renderer. You then write your new class deriving directly from `DSPDecoderBase`, overriding only those methods that you need. For example, `DSPDecoderBase` by default uses wall clock time to report the playback position. If your hardware decoder API provides more precise positional information, you would override `GetCurrentAudioTime` with your own implementation. As a simple example, let's take the WAV renderer that is part of Helix and change it to use a hypothetical hardware PCM decoder. You will find the existing WAV renderer in `datatype/wav/renderer/pcm`. The actual changes to `pcmfmt.cpp` won't be shown, but they are minor and the only important lines from the diff are:

```
+    //Now init the DSP decoder
+    m_pDSPDecoder = new SampleDSP();
+    m_pDSPDecoder->Init(m_pContext);
+    m_pDSPDecoder->PCMInit(m_bSwapSampleBytes, m_bZeroOffsetPCM);
+
```

Notice that the renderer only has to be changed to create and use your new hardware decoder wrapper instead of whatever decoder is currently being instantiated. You then need to call `DSPDecoderBase::Init`, passing in a context to the client engine or player. `DSPDecoderBase` uses that context to get a pointer to

the scheduler and the timeline watcher if needed. The call to PCMInit is specific to this decoder and is in the header file for SampleDSP, as shown here:

```
#include "dspdecoderbase.h"

class SampleDSP: public DSPDecoderBase
{
 public:
  SampleDSP();
  ~SampleDSP();

  //Renderer<-->decoder interface.
  HX_RESULT DecodeAudioData(HXAudioData&  audioData, CMediaPacket* pPacket);
  void     PCMInit( BOOL swapBytes, BOOL zeroOffset);

 private:

  //Decoder specific data.
  BOOL m_bZeroOffsetPCM;
  BOOL m_bSwapSampleBytes;
};
```

For reference, the implementation of SampleDSP is shown below. If you look at datatype/wav/renderer/pcm/pcmfmt.cpp, where the decoding takes place in the WAV renderer, you will see that the CPCMAudioFormat::DecodeAudioData method has just been moved into this example hardware decoder to mimic what would really happen in your hardware decoder. Also note that in this very simple case, the decoded PCM is just being passed back, where in the real world that PCM data would probably be passed directly to the hardware audio output device. In that case, the dummy PCM data would just pass back, as mentioned in the sections above.

```
#include "sampleDSP.h"
#include "hxtick.h"

SampleDSP::SampleDSP()
   : m_bZeroOffsetPCM(FALSE),
     m_bSwapSampleBytes(FALSE)
{
};

SampleDSP::~SampleDSP()
{
}

void SampleDSP::PCMInit(BOOL swapBytes,
                  BOOL zeroOffset)
{
   m_bSwapSampleBytes = swapBytes;
   m_bZeroOffsetPCM = zeroOffset;
}

HX_RESULT SampleDSP::DecodeAudioData( HXAudioData& audioData,
```

```
                        CMediaPacket* pPacket)
{
   HX_RESULT retVal = HXR_FAIL;
   if (pPacket && m_pCommonClassFactory)
   {
      // Create an IHXBuffer
      IHXBuffer* pBuffer = NULL;
      m_pCommonClassFactory->CreateInstance(CLSID_IHXBuffer, (void**) &pBuffer);
      if (pBuffer)
      {
         // Byteswap the samples if necessary
         if(m_bSwapSampleBytes)
         {
            SwapWordBytes((UINT16*) pPacket->m_pData, pPacket->m_ulDataSize >> 1);
         }
         // Offset 8-bit samples if necessary
         if (m_bZeroOffsetPCM)
         {
            UCHAR* pTmp = (UCHAR*) pPacket->m_pData;
            for (UINT32 i = 0; i < pPacket->m_ulDataSize; i++)
            {
               pTmp[i] -= 128;
            }
         }
         // Copy the media packet into this buffer
         retVal = pBuffer->Set(pPacket->m_pData, pPacket->m_ulDataSize);
         if (SUCCEEDED(retVal))
         {
            audioData.pData         = pBuffer;
            audioData.ulAudioTime     = pPacket->m_ulTime;
            audioData.uAudioStreamType = STREAMING_AUDIO;
            audioData.pData->AddRef();
         }
      }
      HX_RELEASE(pBuffer);
   }

   return retVal;
}
```

As you can see, DSPDecoderBase does all the work you need for this very simple case. In a more complicated setup, you might find you need to override some of DSPDecoderBase's methods. For example, if your renderer/decoder pair is decoding data way ahead of time, you may need to override the DSPDecoderBase::Pause and DSPDecoderBase::Resume methods to pass the command to pause or resume down into your hardware layer. In the example above we might do:

```
HX_RESULT SampleDSP::Pause ()
{
   m_pMyDSPAPI->Pause();
   DSPDecoderBase::Pause();
}
```

Notice that the base class's `Pause` method still needs to be called so that it can keep track of all the needed timing information. Also note that you may not be the first hardware decoder loaded. In this case, you will not get calls to `DSPDecoderBase::Pause` because the first hardware decoder replaced the audio device. Instead you will get `IHXTimelineWatcher::OnPause` calls.