Using H.264/AVC DirectX* Video Acceleration with the Intel®
G45/GM45 Express Chipsets

Whitepaper

April 2009

Revision 1.0
Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

This document, as well as the software described in it, is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

The Product Name may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

The code name "Code Name" presented in this document is only for use by Intel to identify a product, technology, or service in development, that has not been made commercially available to the public, i.e., announced, launched or shipped. It is not a "commercial" name for products or services and is not intended to function as a trademark.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel and the Intel Logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2008, Intel Corporation. All rights reserved.
Contents

1 Introduction ................................................................................................... 5

2 Intel® Graphics Media Accelerator X4500HD H.264 Acceleration Requirements ................................................................. 6

3 Establishing a connection ............................................................................... 7
   3.1 Setting up The Decoder ........................................................................ 7
   3.2 Notifying the Video Render .................................................................. 9
   3.3 Create the Decoder .............................................................................. 9

4 DXVA Decoding Operations ........................................................................... 11
   4.1 Overview .......................................................................................... 11
   4.2 Slice Control Structure ....................................................................... 13
   4.3 Reference Picuture Management ......................................................... 15

5 Conclusion .................................................................................................... 20

A References .................................................................................................... 21

B About the Author .......................................................................................... 22
## Revision History

<table>
<thead>
<tr>
<th>Document Number</th>
<th>Revision Number</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxx</td>
<td>0.1</td>
<td>Initial release of the document.</td>
<td>Jan/2009</td>
</tr>
<tr>
<td></td>
<td>0.7</td>
<td>Rewritten to include MPC</td>
<td></td>
</tr>
</tbody>
</table>

§
1 Introduction

The Intel® G45/GM45 Express Chipsets includes the next-generation Intel® Graphics Media Accelerator X4500HD with built-in support for full 1080p high definition video playback, including Blu-ray® movies. The powerful video engine provides users with smooth playback without the need for add-in cards or decoders. Acceleration is provided via the Microsoft DirectX® Video Acceleration API.

Microsoft DirectX® Video Acceleration (DXVA) is an application programming interface for speeding up the decode process of video content by using the capabilities of the graphics hardware. Software codec’s and applications can use DXVA to offload certain intensive operations which frees the CPU to do additional work.

This whitepaper discusses the implementation guidelines for the decoding of H.264/AVC video using DXVA on the Intel® G45/GM45 Express Chipsets. The information is intended to be used in conjunction with the DirectX Video Acceleration Specification for H.264/AVC Decoding, available from the Microsoft Corporation. The content in this paper was developed with close coordination of the authors of Media Player Classic Home Cinema® http://mpc-hc.sourceforge.net/. All of the reference code is available for download in the SourceForge repository.
2 Intel® Graphics Media
Accelerator X4500HD H.264
Acceleration Requirements

The Intel® Graphics Media Accelerator X4500HD provides hardware assisted decode for the H.264/AVC "high profile" bitstreams with up to 40mbps. Only I, P, and B slice types may be present, and arbitrary slice order is not allowed. Please refer to the H.264 specification for more information on the "high profile" requirements.

Software applications and codec's can invoke the media accelerators decode hardware pipeline by adhering to the following requirements:

- Microsoft DirectX Video Acceleration version 2 (DXVA2) is used to communicate with the video hardware. The DXVA2 interface requires Windows Vista* or later.
- Rendering of video requires the Enhanced Video Renderer (EVR) using DirectShow or Media Foundation
- Intel driver revisions must be equal or greater to version or 15.16.
3 Establishing a connection

The following demonstrates how to initialize the DirectX Video Acceleration (DXVA) 2.0 interface within a DirectShow decoder filter. The reader is assumed to have a solid understanding of DirectShow. See the DirectShow SDK for further information.

3.1 Setting up The Decoder

The first step in establishing a connection to the media accelerators decode hardware is to retrieve the Direct3D object by using the IMFGetService interface, open a device handle, and then retrieve a DirectX Video Acceleration service interface pointer.

```c
// Query the pin for IMFGetService.
hr = pPin->QueryInterface(__uuidof(IMFGetService), (void**)&pGetService);

// Get the Direct3D device manager.
if (SUCCEEDED(hr))
{
    hr = pGetService->GetService(
        MR_VIDEO_ACCELERATION_SERVICE,
        __uuidof(IDirect3DDeviceManager9),
        (void**)&pDeviceManager);
}

// Open a new device handle.
if (SUCCEEDED(hr))
{
    hr = pDeviceManager->OpenDeviceHandle(&hDevice);
}

// Get the video decoder service.
if (SUCCEEDED(hr))
{
    hr = pDeviceManager->GetVideoService(
        hDevice,
        __uuidof(IDirectXVideoDecoderService),
        (void**)&pDecoderService);
}
```

Next, the application must retrieve an array of GUIDs that identify the decoder devices supported by the media accelerator.

```c
hr = pDecoderService->GetDecoderDeviceGuids(&cDecoderGuids,
                                        &pDecoderGuids);
```

On the G45, as many as 13 different GUIDS are returned by the above method. This array contains entry points for all the file formats the hardware can handle. An application would typically loop through the returned list, and search for the preferred mode of operation. For H.264/AVC the following GUIDS are relevant:
The above have a 1:1 correspondence with the GUIDS as documented in the Microsoft DirectX Video Acceleration documentation.

- Intel_ModeH264_A provides hardware assist for Motion Compensation without Film Grain Support
- Intel_ModeH264_C provides hardware assist Inverse Discrete Transform and Motion Compensation without Film Grain Support
- Intel_ModeH264_E provides hardware assist for Variable Length Decode, Inverse Discrete Transform, and Motion Compensation without Film Grain Support.

Of all the GUIDS returned, only Intel_ModeH264_E provides “full acceleration” of the picture decode process and it will be the focus of the remainder of this document. The other modes require complex sharing and communication between the application and driver, and are not a recommended mode of acceleration.

When a match on the preferred GUID is found, the next step is to retrieve the supported render targets for the device:

```c
// Find the valid render target formats for this decoder GUID.
hr = pDecoderService->GetDecoderRenderTargets(guidDecoder, &cFormats, &pFormats);
```

The Intel® Graphics Media Accelerator X4500HD will return only a single render format as a D3DFormat variable. The information returned is used to fill out a DXVA2_VideoDesc structure and pass it to the final call in the setup sequence:

```c
hr = pDecoderService->GetDecoderConfigurations(guidDecoder, &m_VideoDesc, NULL, &cConfigurations, &pConfig);
```

The call to GetDecoderConfigurations() retrieves the configurations supported by the media accelerator in an array of DXVA2_ConfigurePictureDecode structures. For the Intel_ModeH264_E, the value of the “ConfigBitStreamRaw” must set to 1 which indicates the host will send raw bit stream data to the accelerator. There are other important configuration parameters that vary depending on the application, the reader is encouraged to explore the different settings by referring to the DXVA documentation online at: [http://msdn.microsoft.com/en-us/library/ms694823(VS.85).aspx](http://msdn.microsoft.com/en-us/library/ms694823(VS.85).aspx)

Once a configuration has been selected, the next step is to notify the video renderer. Remember that the Intel® Graphics Media Accelerator X4500HD requires the EVR video render filter.
3.2 Notifying the Video Render

Once the decoders configuration has been selected, the video renderer needs to be informed that the hardware will be participating in the decode process.

```cpp
CComPtr<IMFGetService> pGetService;
CComPtr<IDirectXVideoMemoryConfiguration> pVideoConfig;

// Query the pin for IMFGetService.
hr = pPin->QueryInterface(__uuidof(IMFGetService),
(void**)&pGetService);

// Get the IDirectXVideoMemoryConfiguration interface.
if (SUCCEEDED(hr))
{
    hr = pGetService->GetService(
        MR_VIDEO_ACCELERATION_SERVICE,
        __uuidof(IDirectXVideoMemoryConfiguration),
        (void**)&pVideoConfig);
}

// Notify the EVR.
if (SUCCEEDED(hr))
{
    DXVA2_SurfaceType surfaceType;
    for (DWORD iTypeIndex = 0; ; iTypeIndex++)
    {
        hr = pVideoConfig->
            GetAvailableSurfaceTypeByIndex(iTypeIndex, &surfaceType);
        if (FAILED(hr))
            break;
        if (surfaceType == DXVA2_SurfaceType_DecoderRenderTarget)
        {
            hr = pVideoConfig->
                SetSurfaceType(DXVA2_SurfaceType_DecoderRenderTarget);
            break;
        }
    }
}
```

3.3 Create the Decoder

After hooking EVR into the pipeline, the application would typically create the buffers it needs to perform the decoding. This involves creating a custom allocator in the DirectShow filter, and calling the CreateSurface() member of the IDirectXVideoAccelerationService. It's beyond the scope of this whitepaper to detail the setup of a custom allocator. Please refer to the DirectShow documentation for more information.

The final step to establishing the decoding session is to create the video decoder. This is done with the call to IDirectXVideoDecoderService::CreateVideoDecoder method. In this method
the application/codec passes in the DXVADDI_Intel_ModeH264_E GUID, a description of the video source format, the decoder configuration, and a pointer to the surfaces the custom allocator created. If everything works out, a handle to the decoding context is returned. It’s time to start decoding.
4 DXVA Decoding Operations

4.1 Overview

The decoding of video pictures using DXVA2.0 consists of a basic sequence of operations by the application/codec. The application/codec’s job is to parse the bit stream header information to fill out each of the required structures.

The DVXA design for H.264/AVC restricts the sequence of buffer types that can be sent to the graphics hardware during the Execute call. For the DXVADDI_IntelModeH264_E interface, the following is required from the application/codec:

- One Picture Parameters Buffer
- One Quantization Matrix Buffer
- One Slice control Buffer
- One Bitstream data buffer

This is analogous to the “Type 3” mode of operation referenced in the DirectX Video Acceleration for H.264/MPEG-4 AVC decoding specification. The graphics hardware will perform bit stream parsing of the video data and compressed picture decoding.

Decode functions are accessed via the IDirectXVideoDecoder interface.

1. BeginFrame(): Signals the start of one or more decoding operations, which will cause the graphics accelerator to lock the destination surface for writing.

2. Execute(): Sends one or more compressed data buffers to the graphics accelerator, and specifies the operations to perform on the buffers. Each call to Execute affects one destination surface.

3. EndFrame(): Signals that the application/codec has sent all the data needed for this begin frame call. After the picture has been decoded, the graphics accelerator will unlock the destination surface.
To illustrate how DXVA2 decodes pictures, the following pseudo code is from Media Player Classic Home Cinema’s H.264/DXVA Decode function:

```c
CDXVADecoderH264::DecodeFrame( BYTE *pDataIn, UINT nSize, REFERENCE_TIME rtStart, REFERENCE_TIME rtStop)
{
    //pDataIn is a pointer to the bytestream;
    While Not at End of the data for this frame
    {
        ReadNext NAL Packet  //(Network Abstraction Packet)
        If NAL Type is a SLICE
            Build/fill in an Array of DXVA_Slice_H264_Long Structures  //There can be more than one slice per frame
        }
    Allocate and Fill in a Picture Parameters Buffer
    Allocate and Fill in a Quantization Matrix Buffer
    Call BeginFrame()
        AddExecuteBuffer (DXVA2_PictureParametersBufferType, ...)
        Execute();  //Sends the Pic Params to the decode hw
    AddExecuteBuffer (DXVA2_BitStreamDataBufferType, nSize, pDataIn, ..)
    Execute();   //Sends the Bitstream decode hw
    AddExecuteBuffer(DXVA2_SliceControlBufferType, ...)
    Execute();  // Sends the SliceControlBufferType
    AddExecuteBuffer(DXVA2_InverseQuantizationMatrixBufferType, ...)
    Execute(); // Sends the quantization matrix buffer
    Call EndFrame(nSurfaceIndex); //All data is sent decode it!
```

The DecodeFrame function is called as soon as the DirectShow filter receives a media sample across its input pin with the pDataIn being a pointer to the actual h.264 data. Communicating with the hardware via DXVA requires that the application/codec parse the bit stream and set up the required structures before passing the data to the hardware. Calling BeginFrame locks the buffer, and EndFrame unlocks it.

If all the parameters are correct, the hardware will write the uncompressed picture to the surface that was passed during the EndFrame() Call. The Graphics Media Accelerator X4500HD follows the Microsoft DirectX Video Acceleration 2.0 specification for H.264 video decoding; however there are some caveats that must be accounted for when developing your application. The remainder of this whitepaper focuses on these key implementation elements.
### 4.2 Slice Control Structure

Two structures are defined in the DXVA Specification for slice control data – DXVA_Slice_H264_Long and DXVA_Slice_H264_Short. The choice of the structure depends on the value of bConfigBitstreamRaw.

The Intel® Graphics Media Accelerator X4500HD requires the use the DXVA_Slice_H264_Long structure. The DXVA_Slice_H264_Short is not supported at this time, and thus the bConfigBitstreamRaw value must be either 0 or 1.

The DXVA_Slice_H264_Long structure:
```c
typedef struct _DXVA_Slice_H264_Long {
    UINT BSNALunitDataLocation;
    UINT SliceBytesInBuffer;
    USHORT wBadSliceChopping;
    USHORT first_mb_in_slice;
    USHORT NumMbsForSlice;
    USHORT BitOffsetToSliceData;
    UCHAR slice_type;
    UCHAR luma_log2_weight_denom;
    UCHAR chroma_log2_weight_denom;
    UCHAR num_ref_idx_l0_active_minus1;
    UCHAR num_ref_idx_l1_active_minus1;
    CHAR slice_alpha_c0_offset_div2;
    CHAR slice_beta_offset_div2;
    UCHAR Reserved8Bits;
    DXVA_PicEntry_H264 RefPicList[2][32];
    SHORT Weights[2][32][3][2];
    CHAR slice_qs_delta;
    CHAR slice_qp_delta;
    UCHAR redundant_pic_cnt;
    UCHAR direct.spatial_mv_pred_flag;
    UCHAR cabac_init_idc;
    UCHAR disable_deblocking_filter_idc;
    USHORT slice_id;
} DXVA_Slice_H264_Long, *LPDXVA_Slice_H264_Long;
```

If the application/codec encounters multiple slices per frame, each slice needs a separate structure allocated to it. Once again, it’s up to the application/codec to parse the bit stream and fill out the values for each member variable. The entire array of DXVA_Slice_H264_Long structures are passed via the Execute call at one time per frame. The following variables have certain caveats/or need special attention when developing your application/codec:

- **BSNALunitDataLocation:**
  This is the byte offset from the start of the bitstream data buffer to the first byte of the current slice. In multi-slice frame this keeps track of each slices data position.

- **SliceBytesInBuffer:**
  The DXVA Type 3 mode of operation requires that the total amount of data passed in the bitstream buffers to be an integer multiple of 128 bytes. Sometimes, codecs pad the buffer to get aligned on a 128 byte boundary. For this accelerator, the SliceBytesInBuffer should NOT count those padded bits.
RefPicList[][]:  
The Index7Bits represents the actual surface index, not the index to RefFrameList[]. See “Reference Picture Management” below.

BitOffsetToSliceData:  
BitOffsetToSliceData is the bit offset to the first bit of the slice data for the current slice. The position can be determined by parsing the slice_header data with an index. There is a special circumstance that needs to be accounted for when setting the BitOffsetToSliceData – the emulation prevention bytes. A H.264 bitstream of data usually has an emulation prevention byte “0x03” inserted into stream data. It acts to differentiate the emulation bytes from the normal stream of data.

When parsing the H.264 bitstream data, the hardware can generally detect and remove the emulation prevention bytes stuffed into the bit stream. However, if the emulation bits span the slice header and slice data boundaries, there may be situations where the hardware will not have enough information to remove the prevention byte sequence. The application must look for this sequence and remove prevention byte if this corner case occurs. The following illustrate this situation:

<table>
<thead>
<tr>
<th>Start Code (prefix) 3 Bytes</th>
<th>Start Code (suffix) 1 Byte</th>
<th>Slice Header Variable Bytes</th>
<th>Slice Data Variable Bytes 00 03 01</th>
</tr>
</thead>
</table>

If a slice starts in the middle of the emulation prevention sequence: 0x00, 0x00, 0x03, 0x00, that would be byte1 (0x00) or byte2 (0x03), then the codec needs to overwrite the stuffed byte (0x03) with the previous byte (0x00) and shift the byte offset for the slice accordingly by 1. This applies to the start of all slices in a multi-slice picture as well.
4.3 Reference Picuture Management

The hardware decoder will utilize a number of previously decoded reference pictures for motion compensated prediction. It is the role of the application/codec to maintain a "picture store" of the previously decoded frames that reside in the decoded picture buffer (DPB), and then make them available to the hardware through both the DXVA_PicParams_H264 buffer and DXVA2_Slice_H264_Long structures.

The DXVA_PicParams_H264 buffer contains the member DXVA_PicParams_H264.RefFrameList[16] which stores up to 16 reference picture elements for each frame. RefFrameList[] is declared with _DXVA_PicEntry_H264 type, and is defined as:

```c
typedef struct _DXVA_PicEntry_H264 {
    union {
        struct {
            UCHAR Index7Bits : 7;
            UCHAR AssociatedFlag : 1;
        };
        UCHAR bPicEntry;
    };
} DXVA_PicEntry_H264, *LPDXVA_PicEntry_H264;
```

- **Index7Bits**: The Surface Index of the uncompressed surface.
- **AssociatedFlag**: Optional 1-bit flag associated with the surface whose meaning depends on the context in which its used.
- **bPicEntry**: Assesses the entire 8 bits of the union.

The DXVA_PicParams_H264 contains the master list of reference pictures for the current frame. The RefFrameList[] is built by parsing the bit stream data for each frame. The bit stream will signify whether or not the current frame can be used as a reference picture (See the H.264 Specification), and also define the total number of reference pictures needed to be passed to the hardware. After the frame is decoded, the frames that are marked to be used as reference pictures are added to the end of the RefFrameList[], and the buffers are shifted up by one. The following comes from Media Player Classic Home Cinema's CDXVADecoderH264::UpdateRefFramesList()

```c
if (bRefFrame)
{
    // Shift buffers if needed
    if (!m_DXVAPicParams.RefFrameList[m_nCurRefFrame].AssociatedFlag && (!bAdded || !m_DXVAPicParams.field_pic_flag))
    {
        if (m_DXVAPicParams.RefFrameList[0].bPicEntry != 255)
            RemoveRefFrame (m_DXVAPicParams.RefFrameList[0].Index7Bits);
        for (i=1; i<m_DXVAPicParams.num_ref_frames; i++)
        {
            m_DXVAPicParams.FrameNumList[i-1] = m_DXVAPicParams.FrameNumList[i];
            memcpy (&m_DXVAPicParams.RefFrameList[i-1], &m_DXVAPicParams.RefFrameList[i],
```
References

The application/codec also needs to send a complete/full set of non-retired reference frames in the DXVA_PicEntry_H264 RefPicList structure, instead of just the reference frames used for decoding the current picture.

For the DXVA_PicParams_H264, the reference picture list is simply a circular buffer whose contents are filled by the parsing in the bitstream. The profile of the source clip will specify the amount of reference pictures the RefFrameList[] will hold at any one time. The application stores the DirectX surface index for the uncompressed picture in the Index7Bits, and thus the hardware has a pointer back to its reference.
Entries that will not be used for decoding of the current picture or subsequent pictures are
indicated by setting the bPicEntry to 0xFF. If the RefFrameList[] element does not have
bPicEntry set to 0xFF, it may be used to decode the current picture – or future pictures in the
decoding order. All of the surfaces that are not marked as “used for reference” must not
appear in the RefFrameList[]. The application/codec needs to inform the hardware it is
sending a full list of reference pictures by setting the DXVA_PicEntry_H264.Reserved16Bits
to 0x34c.

The DXVA_Slice_H264_Long structure also requires the application/codec to pass a list of
reference pictures for each slice of the picture but unlike the DXVA_PicParams_H264 buffer,
the Slice structure is a bit more complicated.

The DXVA_Slice_H264_Long maintains 2 lists of reference pictures. The member variable
DXVA_Slice_H264_Long.RecPicList[2][32] array is also declared as
_DXVA_PicEntry_H264_Type. This variable contains 2 separate lists of reference picture
structures. RefPicList[0] is used for motion compensated prediction of inter macro blocks in P
slices. It can contain short term and long term references. RefPicList[1] is used for inter
coded macro blocks of B slices. B slices can also use RefPicList[0] for its prediction. When
only RefPicList[0] is used (P Slices), RefPicList[1] needs to be cleared by setting bPicEntry to
0xFF for each entry. Regardless of which list is used, it’s important to note that both lists are
filled with the same reference pictures as those found in the DXVA_PicParams_H264 structure
that will be passed to the hardware for this frame. See the H.264 Specification for more
information about short and long term reference pictures.

One of the fundamental differences about the implementation on the Intel® Graphics Media
Accelerator X4500HD should be called out at this point. In other implementations, the
RefPicList[][][].Index7Bits contains a index into the DXVA_PicParams_H264.ReframeList[].
The index is its position in the “master” reference list. On the Intel® GMA X4500HD, the
RefPicList[][][].Index7Bits needs to contain the surface index – so essentially the reference
picture structure for slices has the same information that the master list has in its Index7Bits
field.

The implementation with Media Player Classic Home Theater required the use of a 3rd
variable in the DXVA_PicParams_H264 structure to get reference to the surface index. The
DXVA_PicParams_H264.FrameNumList[16] maintains a list of frame numbers and their
associated entries in DXVA_PicParams_H264.ReframeList[]. The surface pointer was
inserted by this additional level of indirection. The following is sample code from Media Player
Classic Home Cinema’s FF264BuildSliceLong() function:

```c
if(h->slice_type != FF_I_TYPE && h->slice_type != FF_SI_TYPE)
{  if(h->ref_count[0] > 0){
    for(i=0; i < h->ref_count[0]; i++)
    {
      pSlice->RefPicList[0][i].Index7Bits =
      FFH264FindRefFrameIndex (h->ref_list[0][i].frame_num, pDXVAPicParams);
pSlice->RefPicList[0][i].AssociatedFlag = 0;
    if((h->s.picture_structure != PICT_FRAME))
      {
        if((h->seipicstruct == SEI_PIC_STRUCT_BOTTOM_FIELD) ||
            (h->seipicstruct == SEI_PIC_STRUCT_TOP_BOTTOM) ||
            (h->seipicstruct == SEI_PIC_STRUCT_TOP_TOP)){
```
The code above handles the selection of the appropriate list to put the reference picture information into. The num_ref_idx_l(0/1)_active_minus1 contains the active list count. If either of the lists is not used for the current slice, it is cleared out by setting the bPicEntry to 0xff. As mentioned above, to get the Index7Bits field correct, the surface index needs to be retrieved from the master list in the DXVA_PicParams_H264 buffer. The following code handles this lookup:
USHORT FFH264FindRefFrameIndex(USHORT num_frame, DXVA_PicParams_H264* pDXVAPicParams)
{
    int i;
    for (i=0; i<pDXVAPicParams->num_ref_frames; i++)
    {
        if (pDXVAPicParams->FrameNumList[i] == num_frame)
            return pDXVAPicParams->RefFrameList[i].Index7Bits;
    }
    return 127;
}

The management of the reference pictures is typically one of the more challenging aspects of implementing the DXVA application/codec. The key point to remember is that the application/codec is responsible for maintaining its own decoded picture buffer and providing references to hardware via the DXVA_PicParams_H264 and DXVA2_Slice_H264_Long structures for each frame. The Intel® Graphics Media Accelerator X4500HD also requires that the DXVA2_Slice_H264_Long.RefPicList[][].Index7Bits contain the actual surface index and NOT an index into the master list found in the DXVA_PicParams_H264.
Microsoft DirectX Video Acceleration version 2.0 is a means to accelerate hi-definition video playback, including Blu-Ray* movies on The Intel® G45 Express Chipset with the Intel® Graphics Media Accelerator X4500HD and Intel® Clear Video Technology. This whitepaper detailed the implementation guidelines for decoding H.264/AVC content on the G45 Chipset. The project was undertaken with the help of the Intel Software Network community, and could not have been completed without the help of the author’s of Media Player Classic Home Cinema http://mpc-hc.sourceforge.net/. Readers are encouraged to download the code to explore some of the additional complexities of the DXVA interface that this paper did not cover.
A References


Eric Sardella is a senior software engineer in the Software and Services Group at Intel. He has been with Intel for 12 years, and spent most of the time working with graphics. Eric has a BS in Computer Science from California State University, Chico. He is a husband, dad, gamer, fly fisher, racquetball player, who is currently building his own house in scenic Newcastle, CA.