Hardware
Accelerated H.264
Video Encoding
using VAAPITM on
the Intel® Atom™
Processor E6xx
Series

February 2011
Abstract

The Intel® Atom™ Processor E6xx Series for the embedded devices market includes the POWERVR® VXE core that provides video encoding capabilities, allowing to encode high definition video streams in the highly compressed H.264 format with a very low main CPU utilization, releasing the general purpose processor for other parallel workloads. This processor feature is available to application developers by means of the open Video Acceleration API (VAAPI [1]). This paper explains how the VAAPI can be applied to a real time video encoding task, explaining the VAAPI function calls flow, and the corresponding parameters.
Contents

1 Introduction ............................................................................................................... 4
2 Video stream source ................................................................................................... 5
3 Sample encoding application ...................................................................................... 6
4 Encoding call sequence ............................................................................................... 7
   4.1 Overview ......................................................................................................... 7
   4.2 Library initialization ........................................................................................... 7
   4.3 Settings encoding context .................................................................................. 8
   4.4 Allocate destination (encoded video data) buffer ................................................ 9
   4.5 Prepare a source frame .................................................................................... 10
   4.6 Encode source frame ....................................................................................... 12
   4.7 Save coded buffer ........................................................................................... 14
5 Reference List .......................................................................................................... 16

Figures

Figure 1: VA-API initialization code ............................................................................. 7
Figure 2: VA-API cleanup .......................................................................................... 7
Figure 3: Create encoder configuration ....................................................................... 8
Figure 4: Create source data surfaces ......................................................................... 9
Figure 5: Create encoding context ............................................................................. 9
Figure 6: Delete encoding context ............................................................................ 9
Figure 7: Creating coded data buffer ......................................................................... 9
Figure 8: Accessing source surface data buffer ......................................................... 10
Figure 9: Packed I422 YUYV video frame ................................................................. 11
Figure 10: Planar NV12 frame buffer ......................................................................... 11
Figure 11: Planar NV12 macropixel .......................................................................... 12
Figure 12: Release source data buffer ....................................................................... 12
Figure 13: Begin encoding ......................................................................................... 13
Figure 14: Setting video sequence parameters ......................................................... 13
Figure 15: Setting picture parameters ....................................................................... 13
Figure 16: Setting slice parameters .......................................................................... 14
Figure 17: Encode frame ............................................................................................ 14
Figure 18: Completing encoding of a frame .............................................................. 14
Figure 19: Accessing coded buffer data .................................................................... 14
Figure 20: Getting coded buffer size, offset and video data pointer ............................ 14
Figure 21: Save encoded h264 data .......................................................................... 15
1 Introduction

The features and wide acceptance of the H.264 video standard make it applicable to a wide variety of applications and markets including the embedded devices segment.

While providing good video quality at a very low bit rate H.264 encoding requires significant computing power, which makes it a challenging task for the embedded CPU’s that operate within a very limited power envelope. Low-power platforms based on the Intel® Atom™ Processor E6xx Series allow offloading this specific task to the POWERVR™ VXE core, releasing the main CPU for other workload.

Under Linux, this capability is made available to application developers through the same open source Video Acceleration API – VAAPI – that also supports the more ubiquitous task of offloading video decode when doing video playback. There are many articles and open source applications that show how to use VAAPI for video decoding and playback. Video encoding, however remains a less known feature of VAAPI.

Intel also provides support for hardware accelerated video encoding through the GStreamer framework, by means of closed source GStreamer plugins. Application writers that would want more control of the encoding process, targeting more specific and high performance applications might still want to use the VAAPI directly.

This paper covers step by step the application of the VAAPI for video encoding of a raw video data stream captured using a simple web camera.
Before a video stream can be encoded, it must first be acquired in digital form. A well known video source is a camera that could be connected via USB or another hardware interface. In case of Intel® In-Vehicle Infotainment Reference Design platform “Northville” the video can be acquired via the video input port implemented in the ConneXt STA2x11 chip.

The common software interface to the video input devices on Linux is the Video4Linux device driver and a well documented V4L2 API [2].

Video acquisition with the V4L2 API is a relatively simple task from a software point of view. The main thing to consider here is the video data compatibility between video input device and the encoder.

Most video cameras produce video stream in packed 4:2:2 format. On the other hand, most video compression standards operate with video frames in planar YUV 4:2:0 format. The POWERVR* VXE encoder requires the data to be in the NV12 format which is similar to YUV 4:2:0 except that the Y plane is followed by an interleaved U/V plane.

The video data stream from most available video sources is not directly compatible with the POWERVR* VXE encoder hardware. The pixel format conversion and potential de-interlacing has to be done in software running on the main CPU.
3 Sample encoding application

To illustrate how the VAAPI is used for the encoding, we use a sample application that captures the uncompressed I422 video stream from a USB web camera by means of Video4Linux API, encode it in real time using the VAAPI and store the resulting H.264 video stream in a file. It can be downloaded from http://edc.intel.com/Software/Downloads/Code-Samples/License.aspx?id=4746.

**Note:** Real applications would have to handle live video and audio acquisition, synchronization, and packaging into a network data stream or a media container, for example MP4.

The application is using the capturing example covered in the Video for Linux Two API Specification [2], as a code base. The original version is available for download [3].

The encoder-related procedures are placed in a separate file h264enc.c, covering encoder initialization, de-initialization and frame encode functions.

To better demonstrate the principles of video encoding, the application uses the simple frame-by-frame encoding scheme without buffering. It waits for a single video frame to be completely encoded before submitting the next one, while a real application might need to submit multiple frames at a time to utilize the encoder hardware more efficiently.

As mentioned above, it is necessary to convert the source video stream into a format that is compatible with the encoder. The source video format may vary depending on video capture hardware [4]. The encoding application used in this article does I422 YUYV to NV12 pixel format conversion.

The sample application has been tested on Linux, on a platform using an Intel® Atom™ Processor E6xx Series and with Intel® Embedded Media and Graphics Driver (Intel® EMGD) version 1.5.
4 Encoding call sequence

4.1 Overview

In general the video encoding using VAAP consist of the following steps:

1. Initialize VA library
2. Set encoding context
3. Allocate a buffer for encoded data
4. Prepare a source frame
5. Encode the source frame
6. Store the encoded video buffer

The first three steps are usually performed only once per video and the created context and coded data buffer are reused until the last frame of the video is encoded.

Steps 4, 5 and 6 are repeated for every frame of the video sequence.

4.2 Library initialization

Before any task is performed, the libva library must be properly initialized. The initialization code is shown in Figure 1.

Figure 1: VAAP initialization code

```c
int major_ver, minor_ver;
Display *x11_display = XOpenDisplay(":0.0");
VADisplay display = vaGetDisplay(x11_display);
VAStatus status = vaInitialize(display, &major_ver, &minor_ver);
```

Although the X11 display is not generally utilized in the encoding process, it is needed in order to set the driver context for the encoding. Here the library implicitly selects the driver associated with the X display ":0.0"

When the encoding API is no longer needed it must be uninitialized.

Figure 2: VAAP cleanup

```c
vaTerminate(display);
XCloseDisplay(x11_display);
```
4.3 Settings encoding context

In order to set the encoder context, the application needs to specify the encoder configuration as well as the data format and to create input buffers for the source data. This is accomplished in three VA API function calls:

1. Create the encoder configuration with vaCreateConfig()
2. Create source data surfaces with vaCreateSurfaces()
3. Create VA context with the configuration created at step 1 and surfaces created at step 2, using vaCreateContext() function call.

In the encoder configuration, the application specifies the codec profile which, in case of H.264, can be Baseline, Main or High, and the entry point that must be set to VAEntrypointEncSlice in order to set the library in the encoding mode.

The complete list of profiles supported by the API can be found in the VAProfile enum in the library header file va.h. The encoder hardware might not support every profile enumerated in the VAProfile enum. To validate if the desired profile is supported by the encoder hardware the application should invoke vaQueryConfigProfiles() to retrieve all supported profiles and then vaQueryConfigEntrypoints() to validate which profile is supported in the encoding mode. If the returned list of entry points contains VAEntrypointEncSlice entry point, then the encoder hardware does support the specified profile.

The **vainfo** utility from the libva source package allows users to find out what profile/entry point combinations are supported by the hardware.

In addition to the parameters described above the configuration includes a set of encoding attributes. One of the necessary attributes is the video format to be submitted to the encoder. The possible values generally are YUV420, YUV422, and YUV444. As of this writing, the video encoder present in the Intel® Atom™ Processor E6xx Series only supports the YUV420 format.

The second encoder attribute to be specified is the encoder rate control type (VAConfigAttribRateControl). The possible values are VA_RC_CBR (constant bit rate), VA_RC_VBR (variable bit rate) and VA_RC_NONE (no rate control).

**Figure 3: Create encoder configuration**

```c
VAConfigAttrib attrib[2];
attrib[0].type = VAConfigAttribRTFormat;
attrib[0].value = VA_RT_FORMAT_YUV420;
attrib[1].type = VAConfigAttribRateControl;
attrib[1].value = VA_RC_VBR;
VAConfigID config;
va_status = vaCreateConfig(display, VAProfileH264Baseline,
VAEntrypointEncSlice, attrib, 2, &config);
```

To receive the source uncompressed video data and store reference and reconstructed frames, the encoder requires video surfaces to be allocated.
The VA API allows users to create an array of VA surfaces in a single function call. For buffering purposes, it makes sense to create multiple inbound surfaces to buffer incoming data. For simplicity reasons we are however going to use only one source surface. We need 2 additional surfaces to hold reference and reconstructed frames.

![Create source data surfaces](image)

```c
#define NUMSURFACES 3;
VASurfaceID surfaces[NUMSURFACES];
va_status = vaCreateSurfaces(display, frame_width, frame_height,
VA_RT_FORMAT_YUV420, NUMSURFACES, surfaces);
```

Completing the creation of the encoding context, the function vaCreateContext() takes as parameters the encoding config created earlier as well as the reference to the created surfaces to be used in the encoding.

![Create encoding context](image)

```c
VAContextID context;
va_status = vaCreateContext(display, config, frame_width, frame_height,
0, surfaces, NUMSURFACES, &context);
```

The created configuration, context and surfaces will be used during the entire encoding process. After the encoding is done they will need to be destroyed:

![Delete encoding context](image)

```c
vaDestroySurfaces(display, surfaces, NUMSURFACES);
vaDestroyConfig(display, config);
vaDestroyContext(display, context);
```

### 4.4 Allocate destination (encoded video data) buffer

The purpose of this step is to allocate internal library memory for the encoded video frame.

![Creating coded data buffer](image)

```c
unsigned int codedbuf_size = (frame_width * frame_height * 400) / (16*16);
status = vaCreateBuffer(display, context, VAEncCodedBufferType,
codedbuf_size, 1, NULL, &coded_buf);
```

The buffer type must always be VAEncCodedBufferType. The coded buffer size is calculated using the recommended formula for the POWERVR* VXE core.

It is important to understand the lifecycle of different buffer types. Usually the internal buffer is destroyed and recycled when vaRenderPicture() is called (which usually happens during video decoding - playback).
The VAEncCodedBufferType type of buffer is allocated on hardware and can be reused during the encoding of the entire video sequence. The vaRenderPicture() is not supposed to be called for this buffer type. Consequently, the corresponding buffer ID must be destroyed when not needed using vaDestroyBuffer() function call.

It is acceptable to just call vaCreateBuffer(VAEncCodedBufferType) before each frame; the library will return id of a next free hardware buffer. However we will do it in a cleaner way, keeping the encoded buffer_id and explicitly destroying it after we save the last encoded frame by calling vaDestroyBuffer(display, buffer_id);

4.5 Prepare a source frame

Once a source video data is available for encoding, it must be loaded into the source surface. In order to access the source surface data buffer by its surface_id, the application calls vaDeriveImage(surface_id) which fills a VAImage structure.

The retrieved VAImage data contains an image_id which allows the application to obtain a pointer to the input data buffer by calling vaMapBuffer(image_id). The buffer can then be filled with the source video data.

Figure 8: Accessing source surface data buffer

```c
VAImage image;
unsigned char *pbuffer=NULL;
vaDeriveImage(display, surface_id, &image); /* get data layout information */
vaMapBuffer(display, image.buf, &pbuffer); /* get base data pointer */
```

The input video data buffer layout is optimized for the underlying hardware and it does not necessary assume a linear data stream. In fact the hardware expects each of the YUV components and each image row to reside at a particular offset and the application is responsible for rearranging the source data accordingly.

The VAImage structure tells the application exactly how the library expects to receive the data. We will review the VAImage structure in detail.

Most video cameras produce video in packed I422 format. The USB web camera used to test the code for this paper only supported packed I422 YUYV format, meaning that the Y component for every pixel is provided in the data stream together with the corresponding U and V component.

The encoder on the other hand expects the video in planar NV12 format, providing Y all components for the entire video frame followed by interleaved U and V components.
Figure 9: Packed I422 YUYV video frame

![Packed I422 YUYV video frame](image)

Figure 9 shows the position of the YUV components of a single 4x4 frame produced by the used USB camera.

Every two pixels in the data stream (Y11 and Y12) share the same U and V values.

Figure 10: Planar NV12 frame buffer

<table>
<thead>
<tr>
<th>ptr</th>
<th>Y11</th>
<th>Y12</th>
<th>Y13</th>
<th>Y14</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr+8</td>
<td>Y21</td>
<td>Y22</td>
<td>Y23</td>
<td>Y24</td>
</tr>
<tr>
<td>ptr+16</td>
<td>Y31</td>
<td>Y32</td>
<td>Y33</td>
<td>Y34</td>
</tr>
<tr>
<td>ptr+24</td>
<td>Y41</td>
<td>Y42</td>
<td>Y43</td>
<td>Y44</td>
</tr>
<tr>
<td>ptr+32</td>
<td>U11</td>
<td>V11</td>
<td>U13</td>
<td>V13</td>
</tr>
<tr>
<td>ptr+40</td>
<td>U21</td>
<td>V21</td>
<td>U23</td>
<td>V23</td>
</tr>
</tbody>
</table>

Figure 10 shows a NV12 buffer with a 4x4 video frame formatted as specified in the VAImage structure.

The result of vaMapBuffer() gives the application the base address of the internal data buffer – `ptr`. However it does not mean that the data should be placed at the beginning of the buffer.

All Y components of a frame are placed separately. The `VAImage.offsets[3]` array provides the values of the buffer offsets for every component of the image. Which array index corresponds to which component is shown in the `VAImage.component_order[4]`: e.g. `{'Y', 'U', 'V', 0}`

If `VAImage.format.fourcc` is set to `VA_FOURCC_NV12`, then the U and V components have the same common offset.

In the above image, the offset of Y component would be 1, offset of the interleaved UV is 33.

Every line of the frame is usually aligned. The `VAImage.pitches[3]` array provides the pitches for every components. The pitch is the distance in memory, in byte unit, between two vertically adjacent pixels within a field. For the above case the pitch for the Y component (`VAImage.pitches[0]`) would be set to 7, as well as `VAImage.pitches[1]` for UV.
It is necessary to mention that in the I420 planar format the U and V components are shared for the four Y values.

**Figure 11: Planar NV12 macropixel**

\[
\begin{array}{cc}
Y_{11} & Y_{21} \\
Y_{31} & Y_{41} \\
U_{11} & V_{11}
\end{array}
\]

This means that, during the necessary pixel conversion, the application simply discards the U and V components for every second line.

In the attached code example, the procedure that converts the source video frame data to the format expected by the hardware is called `load_yuyv_frame(unsigned char *inbuf, VASurfaceID surface_id)`. It accepts a pointer to the source frame and the `surface_id` of the target surface.

Once the data is in the source buffer, the buffer may be unmapped and the image structure destroyed.

**Figure 12: Release source data buffer**

```
vaUnmapBuffer(display, image.buf);
vaDestroyImage(display, image.image_id);
```

The source data is now put to the source surface and we can proceed with encoding this data.

### 4.6 Encode source frame

The encoding process starts with `vaBeginPicture(display_id, context_id, src_surface_id)`. This function is invoked for each frame that we need to encode. Additionally, during the encoding of the first frame, we have to specify the sequence parameters. The sequence parameters are passed to the driver with a dedicated buffer of type `VAEncSequenceParameterBufferType`.

Working with the VA buffer starts with `vaCreateBuffer()` returning a buffer ID, and finishes with `vaRenderPicture()` which submits the buffer data to the library.

The actual buffer data can be copied to the internal buffer memory from the address passed as parameter 6 of `vaCreateBuffer()`.

An alternative approach would be to use `vaMapBuffer()` to map the internal memory buffer to the application address space and access it directly. Each `vaMapBuffer()` should be followed by a `vaUnmapBuffer()` function call.

Note that the sequence data buffer created with `vaCreateBuffer()` should not be explicitly destroyed with `vaDestroyBuffer()` as the `vaRenderPicture()` function call takes care for recycling it.
Figure 13: Begin encoding

```c
vaBeginPicture(display, context, src_surface_id);
```

Set sequence parameters. This is only required to do once per sequence - for the first frame.

Figure 14: Setting video sequence parameters

```c
If (current_frame == 0){
    VABufferID seq_buf_id;
    VAEncSequenceParameterBufferH264 seq_h264 = {0};
    seq_h264.seq_parameter_set_id = 0;
    seq_h264.level_idc = SH_LEVEL_3;
    seq_h264.picture_width_in_mbs = frame_width / 16;
    seq_h264.picture_height_in_mbs = frame_height / 16;
    seq_h264.bits_per_second = 14000000;
    seq_h264.frame_rate = frame_rate;
    seq_h264.initial_qp = 24;
    seq_h264.min_qp = 1;
    seq_h264.basic_unit_size = 0;
    seq_h264.intra_period = intra_count;
    seq_h264.vui_flag = 0;
    vaCreateBuffer(display, context, VAEncSequenceParameterBufferType,
                   sizeof(seq_h264), 1, &seq_h264, &seq_buf_id);
    vaRenderPicture(display, context, &seq_buf_id, 1);
}
```

Set picture parameters. Here we specify where to take the source data (source surface) from and where to put the encoded data (encoded buffer id).

Figure 15: Setting picture parameters

```c
VAEncPictureParameterBufferH264 pic_h264;
VASurfaceID src_surface_id, rec_surface_id, ref_surface_id;
rec_surface_id = surfaces[NUM_SURFACES - 1];
ref_surface_id = surfaces[NUM_SURFACES - 2];
pic_h264.reference_picture = ref_surface;
pic_h264.reconstructed_picture = rec_surface;
pic_h264.coded_buf = coded_buf;
pic_h264.picture_width = frame_width;
pic_h264.picture_height = frame_height;
pic_h264.last_picture = (current_frame==frame_count-1);
vaCreateBuffer(va_dpy, context_id,VAEncPictureParameterBufferType,
                sizeof(pic_h264), 1, &pic_h264, &pic_param_buf);
vaRenderPicture(va_dpy,context_id, &pic_param_buf, 1);
```
Figure 16: Setting slice parameters

```c
VAEncSliceParameterBuffer slice_h264;
slice_h264.start_row_number = 0;
slice_h264.slice_height = frame_height/16; /* Measured by MB */
slice_h264.slice_flags.bits.is_intra = ((i % intra_count) == 0);
slice_h264.slice_flags.bits.disable_deblocking_filter_idc = 0;
vaCreateBuffer(va_dpy, context_id, VAEncSliceParameterBufferType,
sizeof(slice_h264), 1, &slice_h264, &slice_param_buf);
vaRenderPicture(va_dpy, context_id, &slice_param_buf, 1);
```

Actual encoding process of a frame starts with the vaEndPicture() VAAPi function call:

Figure 17: Encode frame

```c
vaEndPicture(va_dpy, context_id);
```

This function call only starts the encoding process and returns immediately. In order to make sure that the encoder finished encoding the current frame, the application calls vaSyncSurface()

Figure 18: Completing encoding of a frame

```c
vaSyncSurface(display, src_surface_id);
vaQuerySurfaceStatus(va_dpy, src_surface_id, &surface_status);
```

4.7 Save coded buffer

The application gets access to the encoded data by mapping the internal coded_buf_id data into the user space.

Figure 19: Accessing coded buffer data

```c
void *coded_p=NULL;
vaMapBuffer(va_dpy, coded_buf_id, &coded_p);
```

The above function call returns a pointer to the internal data buffer which contains both the data and data size. The first two DWORDs of the coded data buffer contain the data size and data offset.

Figure 20: Getting coded buffer size, offset and video data pointer

```c
coded_size = *((unsigned long *) coded_p);
coded_offset = *((unsigned long *) (coded_p + 4));
unsigned char* pdata = coded_p + coded_offset;
```

Now the application knows enough to write the H.264 data to a file.
This completes the encoding of a single video frame. The application shall follow the same pattern for encoding of every subsequent video frame.
5 Reference List


§

Authors

Igor Shaula is a Senior Software Engineer with the Intel Architecture Group at Intel Corporation.

Terminology

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>VAAPI</td>
<td>Video Acceleration API</td>
</tr>
</tbody>
</table>