VC8000NanoE
Small Footprint Video Encoder

Software Integration Guide

Revision 1.02
08 July 2020

This document is compatible with
VC8000NanoE hardware versions 5.0.x

Legal Notices

COPYRIGHT INFORMATION

This document contains proprietary information of VeriSilicon Holdings Co., Ltd. VeriSilicon reserves the right to make changes to any products herein at any time without notice. VeriSilicon does not assume any responsibility or liability arising out of the application or use of any product described herein, except as expressly agreed to in writing by VeriSilicon; nor does the purchase or use of a product from VeriSilicon convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual property rights of VeriSilicon or third parties.

DISCLOSURE/RE-DISTRIBUTION LIMITATIONS

The information contained in this VeriSilicon proprietary copyright document may be re-distributed by VeriSilicon customer SoC vendors who have licensed the related Hantro IP, with the understanding that the material is re-packaged/re-branded as a licensee (SoC vendor) written/labeled document. Those portions of the SoC vendor document adapted from the VeriSilicon Hantro material should contain a notice similar to: “This chapter contains copyright material disclosed with permission of VeriSilicon Holdings Co., Ltd.”

(VeriSilicon Distribution Level **A: PROPRIETARY INFORMATION**)

TRADEMARK ACKNOWLEDGMENT

VeriSilicon, the VeriSilicon logo, Hantro and the Hantro logo are the trademarks of VeriSilicon Holdings Co., Ltd. in the United States and/or other jurisdictions. All other trademarks are the property of their respective holders.

For our current distributors, sales offices, design resource centers, and product information, visit our web site located at *http://www.verisilicon.com*.

For technical support, please email *hantro-support@verisilicon.com*.

Preface

Glossary and Acronyms

1080p	Progressive high-definition resolution of 1920 x 1080 pixels
4:2:0	YCbCr sampling format, where Cb and Cr components are sub-sampled by two both horizontally and vertically.
4:2:2	YCbCr sampling format, where Cb and Cr components are sub-sampled by two only horizontally.
720p	Progressive high-definition resolution of 1280 x 720 pixels
AC	Alternating Current
AHB	Advanced High Performance Bus, introduced in AMBA 2.0 specification.
AMBA	Advanced Microcontroller Bus Architecture, on-chip bus specification and an open standard
API	Application Programming Interface
APB	AMBA Peripheral Bus protocol specification
AXI	AMBA Advanced eXtensible Interface protocol specification
BIST	Built-In Self Test
CABAC	Context-based Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable Length Coding
CEN	Chip Enable, an input pin to an internal memory
CIF	Common Intermediate Format (352x288 pixels)
D1	Standard television data rate; 720x576 pixels with 25 pictures per second in a PAL system, or 720x480 pixels with 30 pictures per second in an NTSC system
DC	Zero frequency component
DFFRXL	D-type flip-flop example component
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
fps	Frames per second
Inter picture	Coding of a picture using inter prediction, which is derived from samples of reference pictures other than the current encoded picture
Intra picture	Coding of a picture using intra prediction, which is derived from the samples of the current picture
H.264	A video coding standard developed jointly by ITU-T and ISO/IEC
HW	Hardware
IP	Intellectual Property
JFIF	JPEG File Interchange Format
JPEG	An image coding standard developed by Joint Photographic Experts Group
kbps	Kilobits per second
Macroblock	A data unit of four 8x8 luminance pixel blocks, one 8x8 Cb and one 8x8 Cr block
MAD	Mean Absolute Difference. A measure used for assessing macroblock contents.
MB	See macroblock
Mbps	Megabits per second
MV	Motion Vector
MVC	Multi-view Coding, an extension to the H.264 recommendation
NONSEQ	Non-sequential memory transaction (the first transfer of a burst or a single transfer)
NAL	Network Abstraction Layer
Planar	A YCbCr storage format, where all three components form a separate plane in memory
QVGA	Quarter Video Graphics Array (320x240)
QP	Quantisation Parameter, determines the coarseness of quantisation employed during encoding
RAM	Random Access Memory
RGB	Red-Green-Blue color space representation
RLC	Run-Length Coding
RTL	Register Transfer Level
SA	Search Area

SAD	Sum of absolute difference. A measure used in motion estimation.
SCMD	Single Command, Multiple Data.An efficient way to issue multiple addresses in the AXI bus system by signaling the first address only and the length of the burst.
SDRAM	Synchronous Dynamic Random Access Memory
Semi-planar	A YCbCr storage format, where the luminance samples form one plane in memory, and the pixel by pixel interleaved Cb and Cr samples form another
SEQ	Sequential memory transaction
SRAM	Static Random Access Memory
SXGA	Super eXtended Graphics Array (1280x1024)
Verilog	Verilog Hardware Description Language
VGA	Video Graphics Array (640x480)
VLC	Variable-Length Coding
VP8	Video coding standard developed by WebM Project
WEN	Write Enable, an input pin to an internal memory
YCbCr	A color space representation, where color and intensity data are in separate components: Y contains the black and white image (luminance), Cb and Cr the color information (chrominance)
Denoise	Noise reduction for image

`Consolas` font is frequently used when describing code or parameters in the document. *Italic* expression is used in case of file names, paths or commands, and also in notes in text requiring special attention.

Table of Contents

List of Figures

List of Tables

1 Software Integration Overview

This document describes the features, functionality, and system requirements of the Hantro VC8000NanoE multi-format video encoder product and covers all the issues that need to be considered when the encoder is being integrated to a particular software environment. It is assumed that the reader understands the fundamentals of C-language. A prior introduction to the VP8, H.264 and JPEG standards will also help understanding the functionality of the encoder.

Chapter 2 introduces the main features of the product.
Chapter 3 presents a structural overview of the encoder.
Chapters 4 and 5 describe the memory requirements and performance figures of the encoder.
Chapter 6 presents issues related to software porting and integration and shows a practical Linux porting example.
Chapter 7 instructs in the final testing of the integrated product and describes the test benches and scripts.

The functional block diagram of the encoder is shown below.

1.1 References

- ARM (1999) AMBA 2 Specification Revision 2.0.
- ARM (2004) AMBA 3 AXI Protocol v1.0 Specification
- ARM (2004) AMBA 3 APB Protocol v1.0 Specification
- ISO/IEC 14496-10 / ITU-T Recommendation H.264. Advanced video coding for generic audio-visual services.
- ITU-T Recommendation T.81. Information technology – Digital Compression and Coding of Continuous-tone Still Images – Requirements and Guidelines.
- ITU-T Recommendation H.263. Video Coding for Low Bit Rate Communication.
- MPEG-4 Part 2 (formerly ISO/IEC 14496-2). Information technology – Generic Coding for Low Bit Rate Communication.

The following VeriSilicon Hantro reference documents may be useful during integration.

- **Hantro VC8000NanoE Video Encoder Hardware Integration Guide**
- **Hantro VC8000NanoE Video Encoder V5.0.x Software Accessible Registers**
- **Hantro VC8000NanoE H.264 Encoder API User Manual**

2 Features of the Product

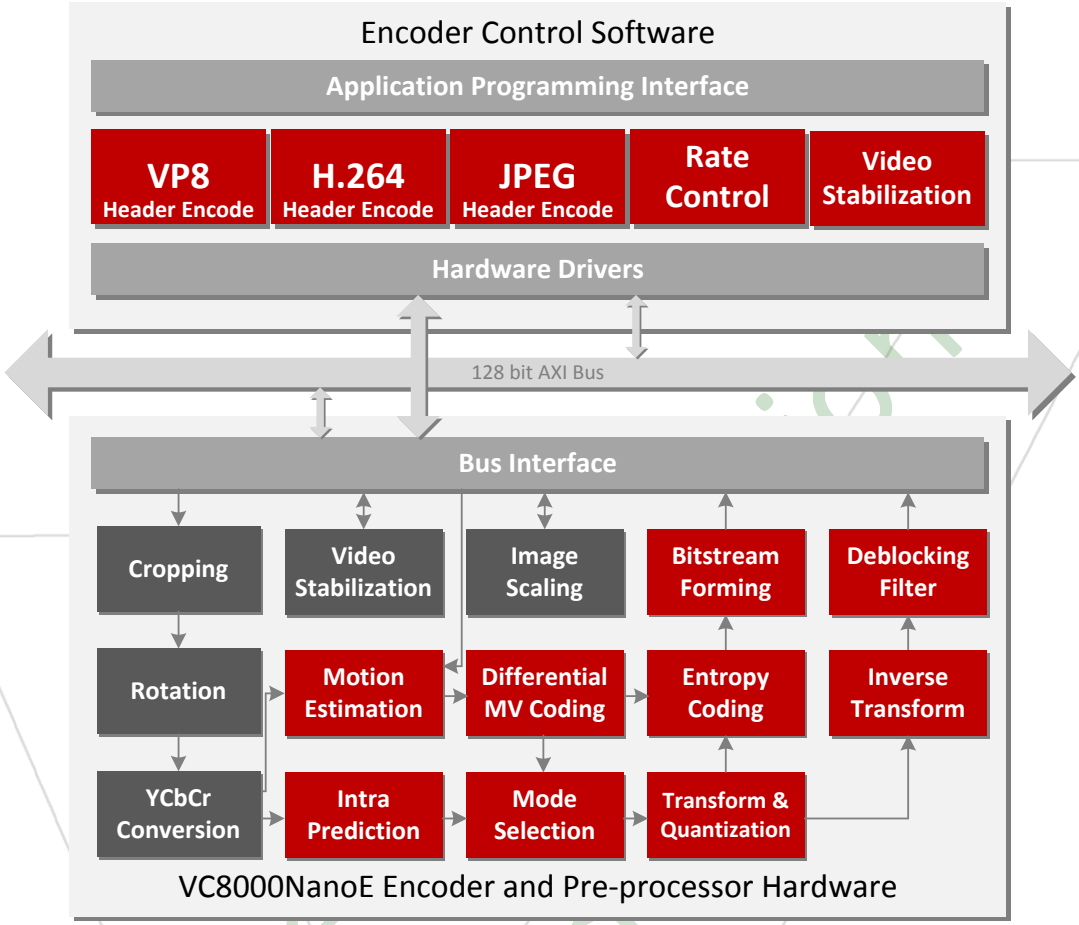

Figure 1. Encoder Functional Block Diagram

(Note: all modules shown may not be available in your configuration)

2.1 Supported Standards and Tools

The VC8000NanoE multi-format encoder is compatible with AHB, AXI and AMBA 3 APB bus interfaces. The encoder supports 32-bit or 64-bit AHB or AXI master interface. The supported address bus width is 64-bit or 32-bit for AHB or AXI master interface. The supported slave interfaces are 32-bit AHB, AXI or APB.

The supported standards, profiles and levels are presented in the table below.

Table 1. Supported Standards, Profiles and Levels

Standard	Encoder support
VP8	• Version levels 1 and 2, bilinear reconstruction filter, all loop filters
H.264 Profile and level	• Baseline Profile, levels 1 – 5.1 • Main Profile, levels 1 - 5.1 • High Profile, levels 1 - 5.1
JPEG profile and level	• Baseline (DCT sequential)
MVC extension to H.264	• Stereo High

The supported VP8 video tools are shown in the table below.

Table 2. Supported VP8 Tools

Tool	Encoder support
Frame Types	• I, P, Golden and droppable frames
Basic coding tools	• 14 intra prediction modes • 4x4 transform • Loop filter: normal, simple and off • Error resilience: ○ Stream mode with no cumulative probability updates
Motion estimation	• Configurable search area: - vertical ±30 pixels, horizontal ±126 pixels or - vertical ±14 pixels, horizontal ±126 pixels • Pixel accuracy: ½ or ¼ • MB and Sub-MB partitions: 16x16, 16x8, 8x16, 8x8 and 4x4 • Algorithm: Indexed motion estimation • Reconstruction filter: bicubic and bilinear
Number of reference frames	• Up to 2
Maximum number of segments	• Up to 4 different quantization parameters per frame
Maximum number of residual partitions	• 5 (control + up to 4 residual partitions)

The supported H.264 video tools are shown in the table below.

Table 3. Supported H.264 Tools

Tool	Encoder support
Slices	• I and P slices • Programmable slice type (mixed I and P slices in one frame)
Entropy encoding	• CAVLC • CABAC • Mixed CAVLC (for intra frames) / CABAC (for inter frames

Basic	• Error resilience:
	  ○ Constrained intra prediction
	  ○ Slices, multiple of macroblocks rows
	• Maximum motion vector length:
	  ○ Vertical +/-30 pixels
	  ○ Horizontal +/-126 pixels
	• Motion vector pixel accuracy:
	  ○ ¼ or ½ pixels
	• Macroblock and sub-macroblock partitions:
	  ○ 16x16, 8x16, 16x8, 8x8, 4x8, 8x4 and 4x4
	• 12 intra prediction modes
	  ○ 4x4 and 8x8 transforms

Number of reference frames	• Up to 2
Maximum number of segments	• Up to 4 different quantization parameters per frame
Maximum number of slice groups	• 1

2.2 Encoding Features

2.2.1 Encoding Feature Support for VP8/H.264/MVC

Table 4. Encoding Features for VP8/H.264/MVC

Feature	Encoder support
Input data format	• YCbCr formats: ○ YCbCr 4:2:0 planar ○ YCbCr 4:2:0 semi-planar ○ YCrCb 4:2:0 semi-planar ○ YCbYCr 4:2:2 raster-scan [1)] ○ CbYCrY 4:2:2 raster-scan [1)] • RGB formats:[1)] ○ RGB444 and BGR444 ○ RGB555 and BGR555 ○ RGB565 and BGR565 ○ RGB888 and BRG888 ○ RGB101010 and BRG101010
Output data format	• VP8: ○ VP8 bitstream • H.264/MVC: ○ Byte unit stream ○ NAL unit stream
Supported image size	• 144 x 96 to 4080 x 4080 • Step size 4 pixels
Maximum frame rate	• Unlimited by the hardware • 30 fps at 1920 x 1080 for VP8 [2)] • 60 fps at 1920 x 1080 for H.264 [2)]
Bit rate	• Minimum: 10 kbps • Maximum: 40 Mbps at 1080P 60 fps
Motion detection	• Encoder outputs the best matching motion vector, the motion estimation quality information (an SAD value) and other coding information for each macroblock
Region-of-interest	• Two user definable rectangular areas with separate quantizer setting
Region-of-intra	• One user definable rectangular area coded as intra
Cyclic Intra Refresh	Supported

[1)] *Internally, the encoder handles images only in 4:2:0 format*

[2)] *Actual maximum frame rate will depend on the logic clock frequency and the system bus performance and the number of cores.*

2.2.2 Encoding Feature Support for JPEG

Table 5. Encoding Features for JPEG

Feature	Encoder support
Input data format	• YCbCr formats: ○ YCbCr 4:2:0 planar ○ YCbCr 4:2:0 semi-planar ○ YCrCb 4:2:0 semi-planar ○ YCbYCr 4:2:2 raster-scan [1)] ○ CbYCrY 4:2:2 raster-scan [1)] • RGB formats:[1)] ○ RGB444 and BGR444 ○ RGB555 and BGR555 ○ RGB565 and BGR565 ○ RGB888 and BRG888 ○ RGB101010 and BRG101010
Output data format	• JFIF file format 1.02 • Non-progressive JPEG
Supported image size	• 96 x 32 to 8176 x 8176 (64 million pixels) • Step size 4 pixels
Maximum data rate	• Up to 90 million pixels per second [2)]
Thumbnail insertion	• RGB 8-bits, RGB 24-bits and JPEG compressed thumbnails supported

[1)] Internally, the encoder handles images only in 4:2:0 format
[2)] Actual maximum frame rate will depend on the logic clock frequency and JPEG compression rate

2.3 Pre-Processing Features

Pre-processing is pipelined with the encoder and can be used only with the VC8000NanoE encoder. Pre-processing features are presented in the table below.

Table 6. Pre-Processing Features

Feature	Encoder support
RGB to YCbCr 4:2:0 color space conversion	• BT.601, BT.709 or user defined coefficients conversion for RGB: ○ RGB444 and BGR444 ○ RGB555 and BGR555 ○ RGB565 and BGR565 ○ RGB888 and BRG888 ○ RGB101010 and BRG101010
YCbCr 4:2:2 to YCbCr 4:2:0 color space conversion	• YCbCr formats: ○ YCbCr 4:2:0 planar ○ YCbCr 4:2:0 semi-planar ○ YCrCb 4:2:0 semi-planar ○ YCbYCr 4:2:2 raster-scan ○ CbYCrY 4:2:2 raster-scan
Cropping	• Video - from 8192 x 8192 to any supported encoding size
Rotation	• 90 or 270 degrees
Image down-scaling	• Proprietary averaging filter • Arbitrary, non-integer scaling ratio separately for both dimensions • Unlimited down-scaling ratio (e.g., from 1080P to QCIF)

2.4 Video Stabilization Features

Digital video stabilization detects and compensates undesired jitter effect on the video while the desired effects like panning are maintained. Stabilization operates with two input picture buffers simultaneously. Stabilization functionality requires at least 8 pixels larger input picture than the actual resolution which is wanted to be encoded.

The figure below shows the relationship of the picture dimensions used by stabilization and demonstrates the effect of stabilizing a video frame. Frame 0 is the first frame and the stabilized picture is positioned in the middle of the camera picture. Frame 1 has been stabilized and the stabilized picture has moved four pixels left. The offsets around the stabilized picture (shown in dark) are cropped out when encoding the video thus creating a more stable video.

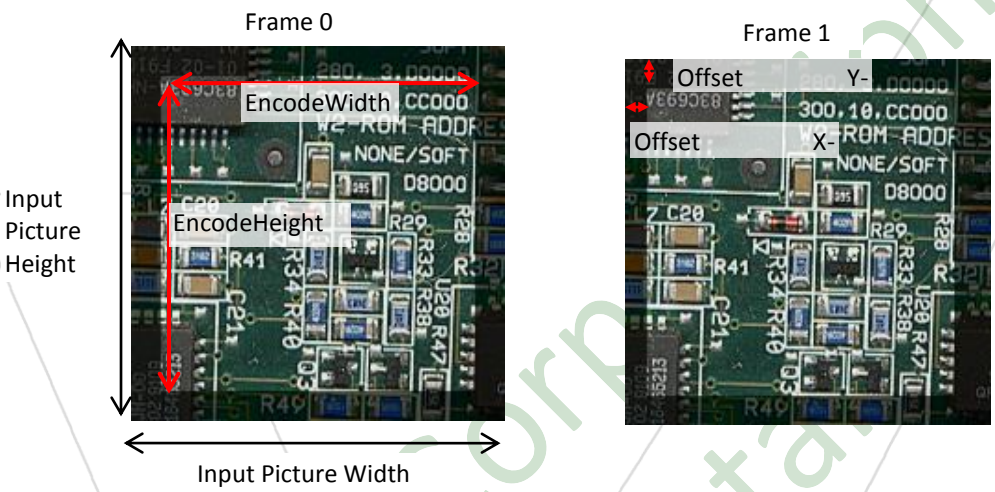

Figure 2. Stabilization Picture Dimension

Video stabilization can be used pipelined with video encoding or in standalone mode when video encoding is disabled. Video stabilization can detect scene changes in the video sequence. Key frames are encoded when a scene change is detected. This will help improve the encoding quality.

Video stabilization features are described in the table below.

Table 7. Video Stabilization Features

Feature	Encoder support
Maximum stabilization displacement in pixels for two sequential input video pictures	• +/-16 pixels
Adaptive motion compensation filter	• From 6 to 40 sequential video pictures noticed in unwanted and wanted movement separation
Offset around stabilized picture	• Minimum 8 pixels in standalone mode • Minimum 16 pixels when pipelined with video encoder • Recommended 64 pixels • Maximum not limited
Scene detection from video sequence	• Encodes key frame, when scene change noticed • Improve encoding quality

2.5 Connectivity Features

The encoder supports the connectivity features presented in the table below. The usage of these features is described in more detail in the *Hantro VC8000NanoE Hardware Integration Guide*. Note that the endian modes can be set individually for input and output data.

Table 8. Connectivity Features

Feature	Encoder support
AHB precise burst / data discard [1)]	Yes
AHB INCR burst type enable [2)]	Yes
AXI SCMD disable	Yes
AXI simultaneous read and write channels	Yes
Restricting maximum issued AHB burst length	Yes, to 4, 8 or 16
Restricting maximum issued AXI burst length	Yes, to any value between 1-16
Interrupt method	Polling or level-based interrupting
Byte ordering	• 32-bit swap • 16-bit swap • 8-bit swap • Any combination of the above separately for input YUV, RGB16, RGB32 and output data.

[1)] When enabled, the bus interface will convert all INCR type read bursts into INCR4 and internally discard the extra data.
[2)] When enabled, the bus interface will use INCR type bursts instead of SINGLE bursts

2.6 Multi-Instance Features

VC8000NanoE encoder is multi-instance capable. As the encoder hardware has no information of previously encoded frames stored in internal memories or registers, the input image can be changed each time a frame is encoded. The video format and resolution can be totally different from the previous one.

External memory availability sets the only limitation for the number of streams to be encoded this way.

2.7 Multicore Features

The VC8000NanoE H.264 encoder is multicore capable. The multicore functionality allows multiple hardware cores to simultaneously encode different frames in a video sequence. This enables the VC8000NanoE encoder to support higher frame rates and resolutions. The figure below shows the difference between single core and multi-core encoding.

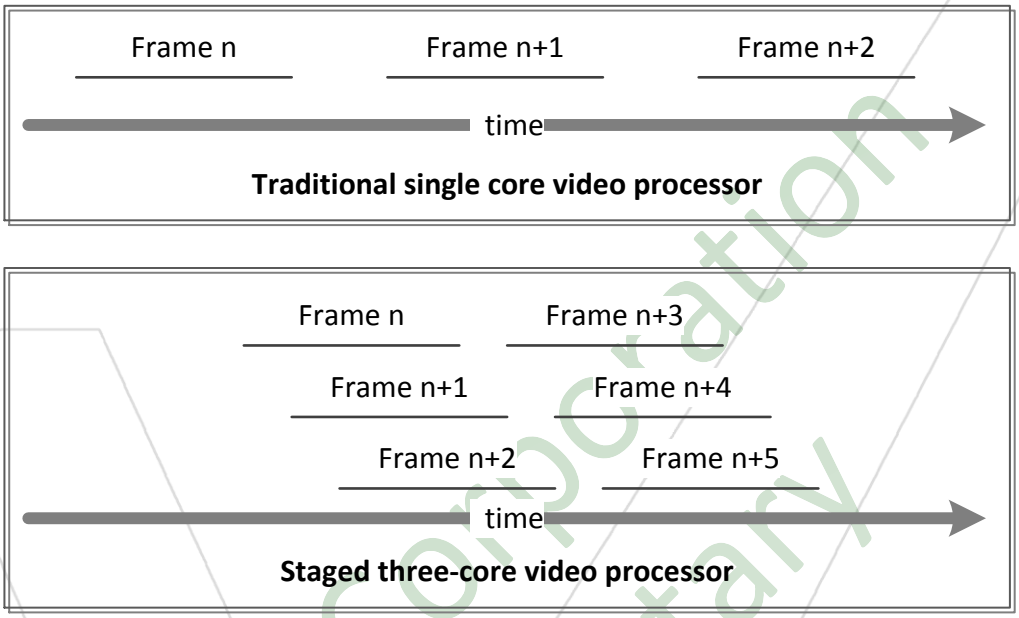

Figure 3. Performance Example of Multicore Instances

The multicore encoder can simultaneously encode as many frames as there are available encoder cores. The core initialization timing and interrupting is asynchronous. The synchronization between the cores is based on the availability of the reference frame data and is done through external memory, i.e., each reference frame buffer contains information about its completeness.

The cores can be asymmetric and secondary cores need to support only the formats requiring the higher performance. From the integration point of view the cores are independent, i.e., each core has their own memories and bus interfaces. Since the cores are independent in the hardware level, they can also be used to encode multiple streams at the same time.

Note: The multicore functionality is supported for H.264 MP, HP and MVC except for interlaced format. The video stabilization is not supported with multicore mode. With a multicore configuration, for H.264, the denoise, ultra low-latency and reference frame compression and some quality improvement features cannot be selected.

3 System Overview

3.1 Functionality of the Product

In the VC8000NanoE encoder, the encoding tasks are divided between hardware and software. When starting to encode a picture, the software will generate any needed stream headers, run the picture-based rate control (only in video mode), and setup the hardware for operation. The hardware encodes the picture macroblock by macroblock and writes out the generated stream to the specified buffer. Hardware can perform all the processing necessary to produce a finished stream data (motion estimation, DCT, quantization, RLC and VLC, etc.).

To synchronize its operations with the software, the hardware can raise an interrupt if one of the following occurs:

- Output buffer limit was reached
- Whole picture was encoded
- An error response was received from the bus
- Hardware was reset

When IRQ has been raised, a status register will indicate the reason for the interrupt. The status register can be used for polling mode of operation when the IRQ can also be disabled.

The encoder can do video stabilization at the same time when it does the video encoding. This requires processing of two pictures at the same time (one is encoded and the other one is stabilized). The video stabilization can be also run in standalone mode (without video encoding), controlled by its own API.

3.2 Software Composition of the Product

The VC8000NanoE encoder software is implemented in ANSI-C and its composition can be seen in the figure below.

The VC8000NanoE encoder has two main software interfaces. One is the top-level APIs, which are used by any application needing VP8, H.264 or JPEG encoding capability. The encoder contains a video stabilization block which can work in pipeline with any of the video encoder modes or it can be used by itself (standalone mode) via the Video Stabilization (VS) API. The other interface is the **Encoder Wrapper Layer** (EWL), which provides system dependent resources to the VC8000NanoE encoder. All these system level actions, such as physical memory allocation, hardware I/O register access and SW/HW synchronization, will require special system dependent implementation. With this approach the modifications to be done when the VC8000NanoE encoder is ported to a system are grouped in a well-defined separate part. Notice that depending on the particularities of the target system, some modifications may be needed in other parts of the software as well.

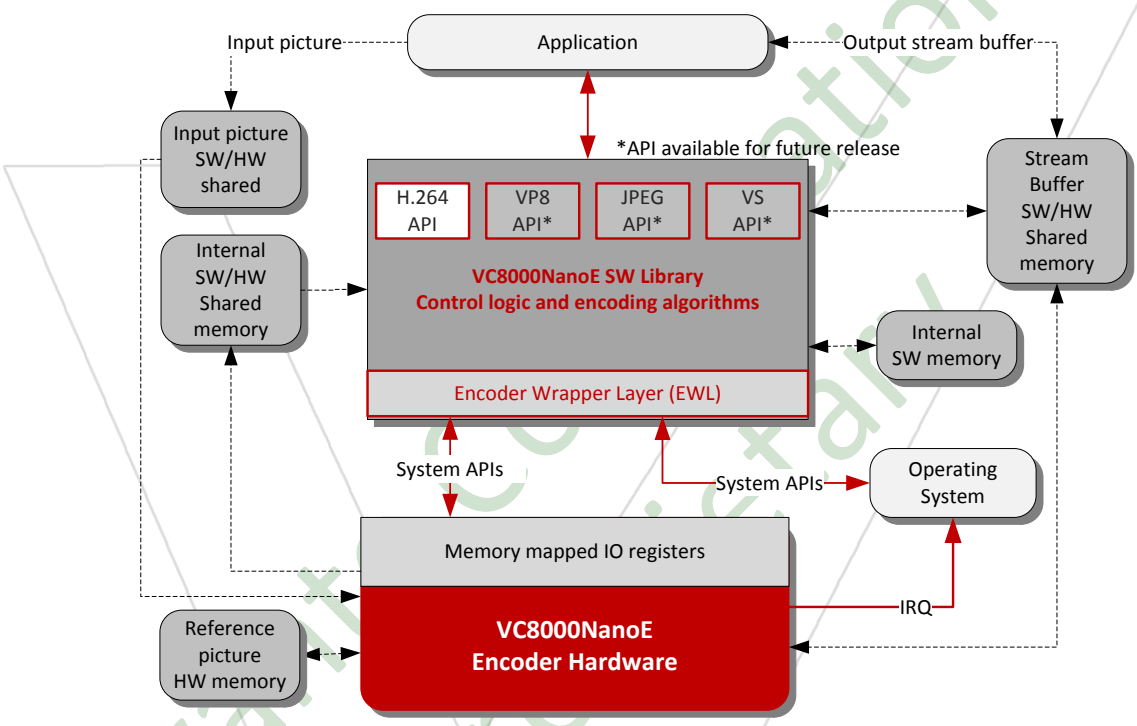

Figure 4. Integrated Structure of the Encoder and its Main Interfaces

As can be seen in the figure above, the VC8000NanoE encoder requires several types of memory buffers.

- One is the memory shared between the different software components (Internal SW memory) and which is usually allocated using the C standard library memory allocation routines (malloc, calloc). This SW/SW shared memory does not require any special treatment. The EWL interface contains a series of functions that will provide these SW/SW memory resources (EWLmalloc, EWLcalloc, EWLmemcpy, etc.)

- The second type of memory is shared between hardware and software (Internal HW/SW Shared Memory). These memory spaces are used to store the hardware generated data for the software. Special care must be taken when caching mechanism are in use, so that these HW/SW memory buffers will stay coherent between software and hardware accesses. Allocated HW/SW memories must be linear, contiguous physical memory buffers. This memory is allocated using the EWL function EWLMallocLinear.

- The third type of memory is only used by the hardware (Reference picture HW Memory). This memory is used by the encoder hardware for storing the reference picture of the encoder. This memory is allocated using the EWL function EWLMallocRefFrm. *Note: Reference frames are used only for video encoding and not for still images.*

- The fourth type of memory (Input picture SW/HW shared memory) is used by the encoder hardware and depending on the application may be used by the software as well. This is the input picture for the encoder which may be coming straight from the camera. The buffer is allocated and written outside the encoder and the encoder hardware reads the picture when encoding. The encoder software does not access this buffer. This buffer is allocated externally from the encoder.

- The last memory area is the one occupied by the output stream buffer (Stream Buffer SW/HW shared memory). This must be allocated also by the application using the encoder.

The encoder software library internal structure is shown in the figure below.

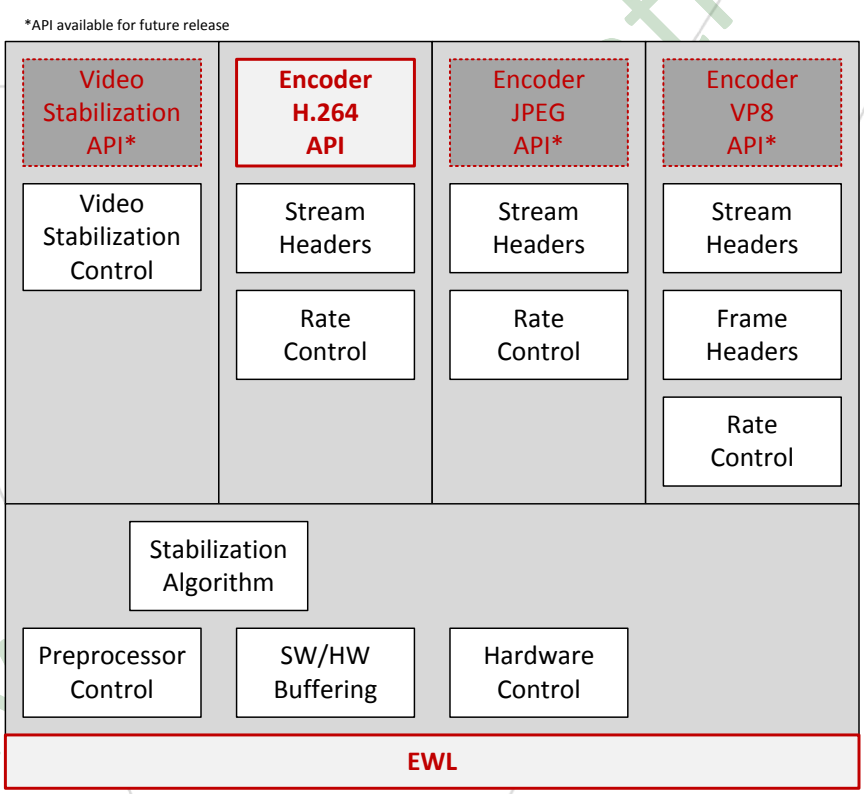

Figure 5. Encoder Library Internal Structure and Interface

The software is divided into 5 parts:

1. Video Stabilization (components)
2. H.264
3. JPEG
4. VP8 and
5. Common part which is used by all the components.

Each component is independent and re-entrant so multiple instances can be used at the same time. The video stabilization can be run standalone by its own API or can be pipelined with the video encoder operation. In

pipelined mode, the encoder will take care of the stabilization control. But since there is a single hardware, only one encoder or standalone stabilization can be running at a time. The hardware sharing is done picture based. This means that one component reserves the hardware (EWLReserveHw), processes one picture and then releases the hardware (EWLReleaseHw) so that another component instance can acquire it and use it.

The common part of the software takes care of the hardware memories and controlling the hardware registers. It uses the EWL functions for reading and writing the hardware registers and for polling the hardware interrupts. The video stabilization algorithm implementation is also common for all components.

4 Memory Requirements

This chapter describes the encoder's memory requirements. Each required memory type is presented in detail.

4.1 Input Picture Buffer

The input picture buffer is a memory buffer for hardware use only which must be allocated by external means (i.e., it is not allocated by the encoder software). This will contain the YCbCr or RGB picture data. The buffer must be linear and contiguous, and allocated in a memory area accessible by the hardware. It also must be 64-bit aligned. Depending on the application and camera implementation it may be needed to use double buffering for input picture. When video stabilization is enabled two input pictures are required. The size of the input picture buffer depends on the input picture format. The same kind of input buffer is used for VP8, H.264, and JPEG codec and video stabilization functionality.

The table below represents the amount of memory required for the input picture buffer depending on the input picture format, and the maximum buffer size.

Table 9. Input Picture Buffer Size

Input Picture Format	Amount	Size per MB (byte)	Max Size for Video (byte) [1)]	Max Size for JPEG (byte) [2)]
YCbCr 4:2:0 planar	1 or 2	384	3,110,400	24,556,032
YCbCr 4:2:0 semiplanar	1 or 2	384	3,110,400	24,556,032
YCbCr 4:2:2 interleaved YCbYCr	1 or 2	512	4,147,200	32,741,376
YCbCr 4:2:2 interleaved CbYCrY	1 or 2	512	4,147,200	32,741,376
RGB565 or BGR565	1 or 2	512	4,147,200	32,741,376
RGB555 or BGR555	1 or 2	512	4,147,200	32,741,376
RGB444 or BGR444	1 or 2	512	4,147,200	32,741,376
RGB888 or BGR888	1 or 2	1024	8,294,400	32,741,376
RGB101010 or BGR101010	1 or 2	1024	8,294,400	65,482,752

[1)] –1080p (1920x1080) resolution

[2)] – One 16Mpixel (4672x3504) size picture, containing 63948 macroblocks

Note: Video stabilization always needs 2 input pictures.

4.2 Output Stream Buffer

Output buffer of the encoder is externally allocated memory area where the produced stream data is available to the application. In all encoding modes the output buffer must be linear and contiguous, and allocated in a memory area accessible by the hardware. It also must be 64-bit aligned.

As the amount of data produced by the encoder varies depending on the resolution and quantization, this buffer's size cannot be predefined. The limitation for minimum buffer size is that the buffer must be big enough for one frame, since if the buffer end is reached while encoding, the encoded frame will be lost. The maximum buffer size is limited by the encoder register storing the buffer size. 25 bits is used to store the buffer size in 64-bit addresses, which means that the maximum stream buffer size is 256 Mbytes.

In typical cases it is recommended to use output stream buffer size equal to encoded picture size.

4.3 VC8 and H.264 Encoder

The memory needs of VP8 and H.264 Encoder are described in this chapter.

4.3.1 Hardware Internal Buffers

The hardware internal buffers store the internal reference picture used by the hardware for motion estimation. The buffers must be linear and contiguous, and allocated in a memory area accessible by the hardware. They also must be 64-bit aligned. The buffer size depends on the encoded picture resolution.

Allocations are done with EWLMallocRefFrm(), which is implemented during the integration phase. The number of allocated buffers depends on the stream view mode that is chosen. For luminance data, one or two picture buffers are needed and for chrominance, two or three buffers are needed. The number of buffers needed depends on the encoder's ability to skip encoded frames and the hardware restriction that for chrominance, one buffer acts as a read buffer and the other as a write buffer. The following three tables indicate the buffer sizes in bytes.

Table 10. Luminance and Chrominance Buffer Size in Bytes

Name	Size per MB (byte)	Buffer Size (bytes)				
		QCIF	CIF	VGA	720p	1080p
Luminance (Y) Buffer	256	25344	101376	307200	921600	2088960
Chrominance (CbCr) Buffer	128	12672	50688	153600	460800	1044480

Table 11. H.264 Internal Buffer Size in Bytes

Name	Size per MB (byte)	Buffer Size (bytes)				
		QCIF	CIF	VGA	720p	1080p
Single buffer	1 luma + 2 chroma	50688	202752	614400	1843200	4177920
Double buffer	2 luma + 2 chroma	76032	304128	921600	2764800	6266880
MVC Stereo inter view prediction	1 luma + 2 chroma	50688	202752	614400	1843200	4177920
MVC Stereo inter prediction	2 luma + 3 chroma	88704	354816	1075200	3225600	7311360

Table 12. VP8 Hardware Internal Buffer Size in Bytes

Name	Size per MB (byte)	Buffer Size (bytes)				
		QCIF	CIF	VGA	720p	1080p
1	1 luma + 2 chroma	50688	202752	614400	1843200	4177920
2	2 luma + 3 chroma	88704	354816	1075200	3225600	7311360
3	3 luma + 4 chroma	126720	506880	1536000	4608000	10444800

4.3.2 HW/SW Shared Memory

The HW/SW shared memories are memory buffers shared between the hardware and software components of the encoder. Each individual buffer must be linear and contiguous and allocated in a memory area accessible by both the hardware and software. They all must be 64-bit aligned. If memory-caching mechanisms are in use, the consistency of these memory areas must be taken care of. The following two tables indicate the buffer sizes in bytes.

Table 13. H.264 HW/SW Shared Memory Size in Bytes

Name	Size per MB (byte)	Buffer Size (bytes)				
		QCIF	CIF	VGA	720p	1080p
NAL size buffer	(heightMbs+4)*4	56	88	136	200	288
MV output buffer	Mbtotal*40	3960	15840	48000	144000	326400
CABAC context table	48256	48256	48256	48256	48256	48256
Segmentation map	mbTotal/2	56	200	600	1800	4080
Total		**52328**	**64384**	**96992**	**194256**	**379024**

Table 14. VP8 HW/SW Shared Memory Buffer Size in Bytes

Name	Size per MB (byte)	Buffer Size (bytes)				
		QCIF	CIF	VGA	720p	1080p
Partition size buffer	(heightMbs+4)*4	56	88	136	200	288
MV output buffer	Mbtotal*40	3960	15840	48000	144000	326400
Entropy probabilities	1208	1208	1208	1208	1208	1208
Probability counters	488	488	488	488	488	488
Segmentation map	mbTotal/2	56	200	600	1800	4080
Total		**5768**	**17824**	**50432**	**147696**	**332464**

Allocations are done with EWLMallocLinear(), which is implemented during the integration phase.

NAL size buffer and MV output buffer are written by the hardware during encoding and pointers are output to the application after frame completion. The application must read and store the data in these buffers because the next frame encoding will overwrite the buffer.

CABAC context table is written by the control software once in the beginning of stream encoding. The hardware will read this table in the beginning of each frame encoding.

4.3.3 SW/SW Shared Memory

The SW/SW shared memories are memory buffers shared between the software components of the encoder. These are normally allocated using common dynamic software memory allocation routines. Allocations are done with EWLmalloc or EWLcalloc, which are implemented during the integration phase.

Table 15. SW Memory Buffers in Bytes

Memory Type	Buffer Size (bytes)				
	QCIF	CIF	VGA	720p	1080p
H.264 SW/SW memory buffers	5556	5556	5556	5556	5556
VP8 SW/SW memory buffers	31644	34020	40452	59652	96132

4.3.4 Overall Memory Usage

The table below presents the amount of memory allocated by the encoder for different picture resolutions. The stream output buffer is external and not included in the table.

Table 16. Memory Usage of H.264 Encoder With Different Picture Sizes

Memory Type	Buffer Size (Kbytes)				
	QCIF	CIF	VGA	720p	1080p
HW internal, default double buffered view mode	75	297	900	2700	6120
HW/SW	52	63	95	188	367
SW/SW	6	6	6	6	6
Input picture[1)]	38	149	450	1350	3060
Total default	**171**	**515**	**1451**	**4244**	**9553**

[1)] (YCbCr4:2:0) Not allocated by the encoder internally

Table 17. Memory Usage of VP8 Encoder With Different Picture Sizes

Memory Type	Buffer Size (Kbytes)				
	QCIF	CIF	VGA	720p	1080p
HW internal, default double buffered view mode	50	198	600	1800	4080
HW/SW	6	18	50	145	325
SW/SW	31	34	40	59	94
Input picture[1)]	38	149	450	1350	3060
Total default	**125**	**399**	**1140**	**3354**	**7559**

1) (YCbCr4:2:0) Not allocated by the encoder internally

4.4 JPEG Encoder

The JPEG encoder's memory needs are described in this chapter.

4.4.1 SW/SW Shared Memory

The SW/SW shared memories are memory buffers shared between the software components of the encoder. These are normally allocated using common dynamic software memory allocation routines. Allocations are done with EWLmalloc or EWLcalloc, which are implemented during the integration phase.

Table 18. JPEG Software Memory Buffers

Memory Type	Size (bytes)
SW/SW memory buffers	1428

4.4.2 Overall Memory Usage

The table below presents the amount of memory allocated by the encoder for different picture resolutions. The stream output buffer is external and is not included in the table.

Table 19. Memory Usage of JPEG Encoder With Different Picture Sizes

Memory Type	Picture Resolution				
	1 Mpixels (Kbytes)	1 Mpixels (Kbytes)	5 Mpixels (Kbytes)	8 Mpixels (Kbytes)	16 Mpixels (Kbytes)
SW/SW	2	2	2	2	2
Input picture[6] (YCbCr4:2:0)	1482	4422	7371	11742	23981,5
Total default [1]	**1484**	**4424**	**7373**	**11744**	**23984**

[1] – 1 Mpixel (1216x832) size picture = 3952 macroblocks
[2] – 3 Mpixel (2096x1440) size picture = 11790 macroblocks
[3] – 5 Mpixel (2688x1872) size picture = 19656 macroblocks
[4] – 8 Mpixel (3408x2352) size picture = 31311 macroblocks
[5] – 16 Mpixel (4672x3504) size picture = 63948 macroblocks
[6] Not allocated by the encoder internally

Note: JPEG encoding can be done in smaller slices, so the input image buffer can be much lower. This presumes that the input image is captured one slice at a time.

4.5 Video Stabilization

The memory needs of Video Stabilization are described in this chapter.

4.5.1 SW/SW Shared Memory

The SW/SW shared memories are memory buffers shared between the SW components of the encoder. These are normally allocated using common dynamic software memory allocation routines. Allocations are done with EWLmalloc or EWLcalloc, which are implemented during the integration phase.

Table 20. Video Stabilization Software Memory Buffers

Memory Type	Size (bytes)
SW/SW memory buffers	712

4.5.2 Overall Memory Usage

The table below presents the amount of memory allocated by the Video Stabilization for different picture resolutions.

Memory Type	Picture Resolution					
	QCIF (Kbytes)	CIF (Kbytes)	VGA (Kbytes)	PAL (Kbytes)	720p (Kbytes)	1080p (Kbytes)
SW/SW	1	1	1	1	1	1
Input picture[1)] (YCbCr4:2:0)	75	297	900	1215	2700	6120
Total default	**76**	**298**	**901**	**1216**	**2701**	**6121**

[1)] Video stabilization always needs 2 input pictures to work on. Not allocated by the video stabilization internally. Also, stabilization needs just luminance data so in standalone mode the picture buffer can be smaller.

4.6 Code Size

In addition to the dynamically allocated memories, the encoder software library code size is approximately 225 Kbytes. *Note: This size may be different depending on your gcc version.*

The total code size includes the reference encoder system wrapper layer implementation for Linux delivered with the source code.

5 Performance Figures

5.1 VP8 and H.264 Encoder

The VP8 and H.264 Encoder software has a minimal processor load. Software only handles the stream header generation and the picture-based rate control. VP8 software also updates the probability tables. Overall load is less than 100K CPU cycles for each encoded picture. The performance will be impacted when register access is slow.

5.2 JPEG Encoder

The JPEG Encoder software has a flat processor load caused by the stream header generation and the hardware control. This load is about 400K CPU cycles for each encoded picture.

5.3 Video Stabilization

The standalone Video Stabilization software has a flat processor load, about 2000 CPU cycles for each processed picture.

6 Integration of the Product

6.1 Software Source Hierarchy

All platform independent software source files can be found in the **`vc8000ne/software/source`** folder.

6.2 Behavior

The Hantro VC8000NanoE API User Guides should be consulted for a detailed description of all the API functions and their usage.

6.2.1 Encoder Initialization

The encoder initialization is accomplished by calling one of the API initialization functions to create an encoder instance. At this phase the encoder will check the configuration parameters (uses `EWLReadAsicConfig` to check if configuration is supported by hardware), setup its own internal structures, initialize the EWL (`EWLInit`) and allocate all needed memories. If any of these tasks fails, the encoder instance creation will fail.

6.2.2 Encoding Pictures

The encoder generally processes a full picture at a time. The exception is encoding large JPEG images. In this last case the encoder can encode smaller slices of a bigger picture. This sliced mode can reduce drastically the input picture memory consumption in cases where the capture device supports also capturing a picture in several slices.

The figure below shows the process of encoding a picture. For multi-instance purposes, first the software has to lock the hardware resources for exclusive use before attempting any access of it. This is mandatory when multi-instance support is required.

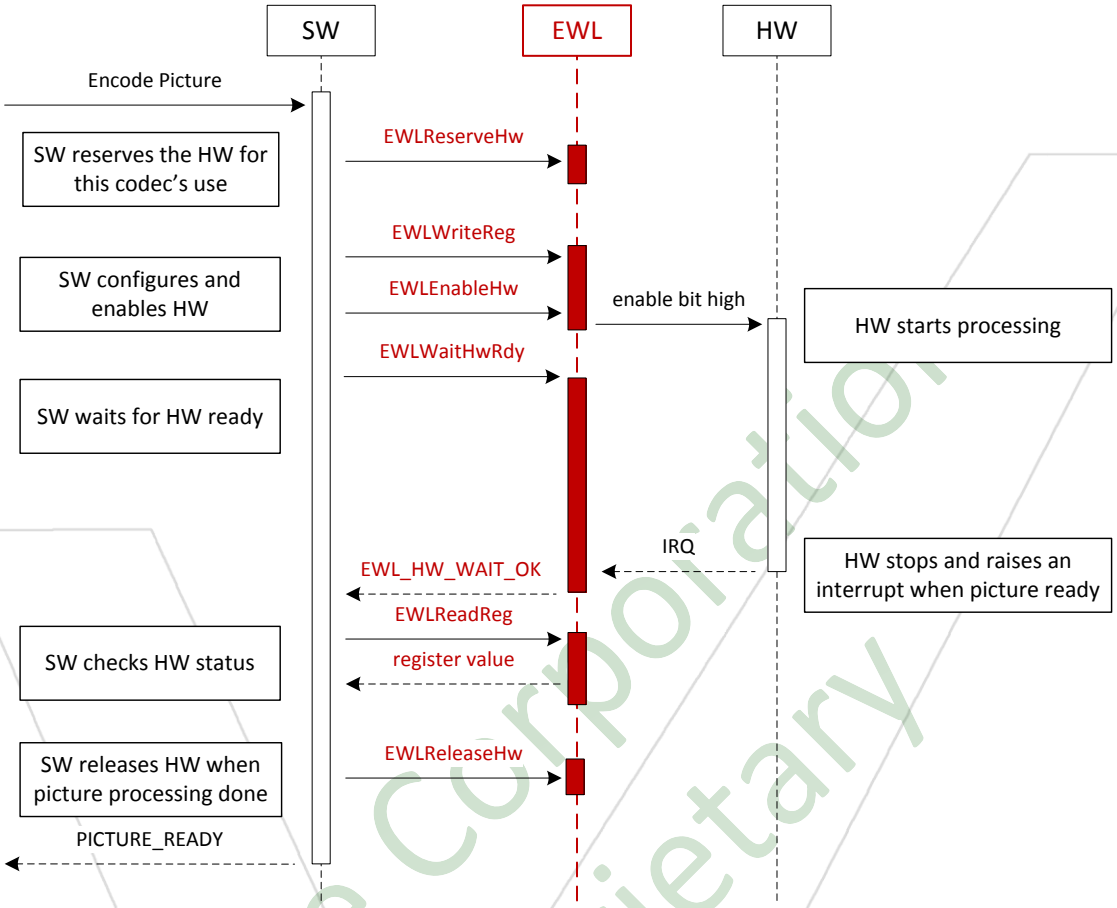

Figure 6. Picture Encoding Process

When encoder hardware has been setup and enabled, it will process the input picture and produce the desired encoded data. Hardware produces fully encoded streams in VP8, H.264 or JPEG formats.

Once it has finished processing the picture, hardware is released to be available for another encoder instance.

For H.264 encoding, the picture encoding process is started by the application by calling `H264EncStrmEncode`. This function will check and setup all frame dependent parameters and call rate control function. At this point the rate control may choose to skip the frame before it is even encoded. If the rate control decides that the frame should be encoded, it calculates the QP and other rate control parameters for that frame. Then the function `H264CodeFrame`, which will take care of the frame encoding, is called. A similar approach is used for VP8 as well.

6.2.3 Hardware Sharing for Multi-Instance Encoding

Multi-instance encoding is supported by the encoder software when a method for sharing the hardware resource between several instances is implemented. This means that a proper locking mechanism must be provided so that exclusive access to the hardware can be granted and assured for one instance at a time. The exclusive accessing process is presented in the figure below.

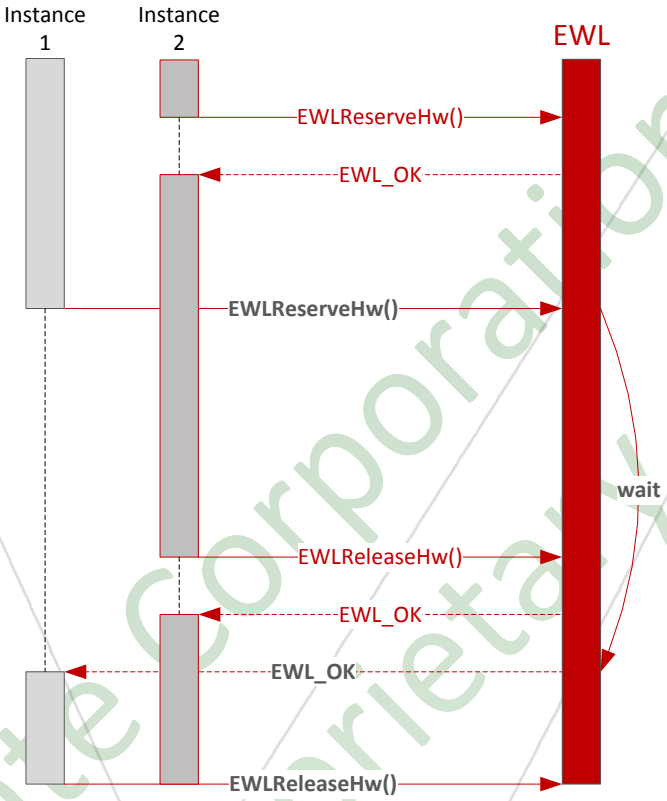

Figure 7. Exclusive Access to Hardware Resources

The encoder software will not access any hardware resources (registers, IRQ) until exclusive access is granted. Access is considered granted when the `EWLReserveHw` call has successfully returned. The exclusive access is given up with a call of `EWLReleaseHw`.

Implementing the `EWLReserveHw` and `EWLReleaseHw` functions is one part of the porting task. The most important thing is to make sure that two codec instances cannot access the hardware at the same time. In the reference implementation, this is achieved by using a system wide process semaphore.

6.2.4 Hardware Configuration

The hardware has a set of memory mapped I/O registers, which are used to configure the hardware. Reading and writing of the individual values in the registers is done by functions declared in `common/encasiccontroller.h`. These functions are using the EWL read and write register functions (`EWLReadReg`, `EWLWriteReg`). Because these EWL calls are returning full register values the only extra operation done is masking out the relevant bits for a certain parameter. The figure below shows how the EWL read and write register functions are used by the encoder.

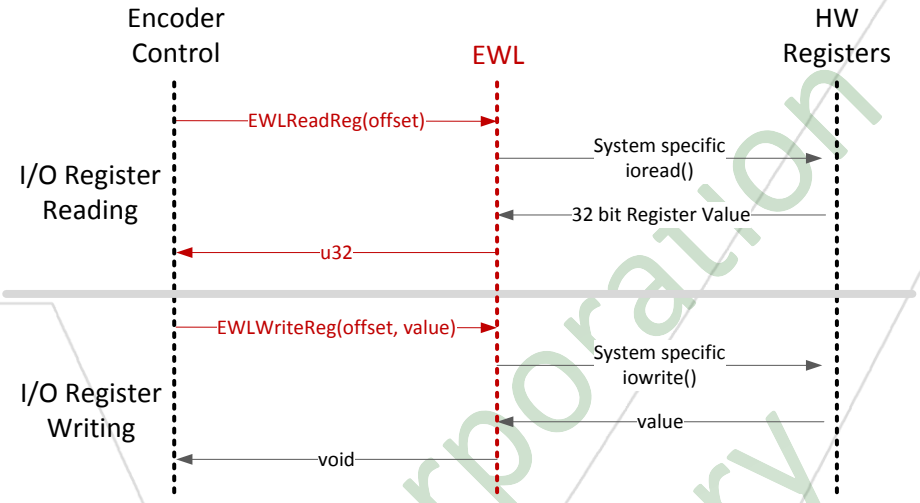

Figure 8. Hardware Register Access Sequence

The encoder control code is implemented so that when the hardware configuration is done, i.e. all the registers are setup correctly, the `EWLEnableHw` will enable the hardware with a final writing to a one specific register. Having this clear hardware starting point allows a more flexible EWL implementation, in which the register accesses can be buffered. When the accesses are buffered, the cached values shall be written out to the hardware when `EWLEnableHw` is called and refreshed, i.e., read back from the registers, always after a hardware status change (IRQ).

The encoder can force a hardware stop by calling `EWLDisableHw` which will disable the hardware by writing the enable bit low. This register access cannot be buffered.

Implementing the `EWLEnableHw`, `EWLDisableHw`, **`Error! Reference source not found.`** and **`Error! Reference source not found.`** functions is OS-specific and part of the porting task.

6.2.5 HW/SW Synchronization

The software component of the encoder needs to synchronize its job with the hardware runs. When the encoder software can't continue until the hardware has finished, it will call `EWLWaitHwRdy` in order to wait for the hardware to finish its part. This process is shown in the figure below.

The hardware can generate an IRQ when it has finished processing (if the IRQ generation is enabled) and sets the interrupt status bits in the registers. The IRQ generation is controlled by setting the IRQ disable bit (swreg1: Bit[1]) in the hardware registers (See *Hantro VC8000NanoE Video Encoder V5.0.x Software Accessible Registers*) for the hardware register description). The whole encoder must be aware if the IRQ is in use or not. This is set in the `encfg.h` file (See: Section 6.6 Building and Configuring the Software).

The `EWLWaitHwRdy` function is used so that it allows several types of implementations depending on the target OS. Implementing this function is one of the most important parts of the porting and it can utilize any of the following methods, which ever suits the target system the best:

- IRQ wait - this is used in the reference implementation and described in the figure below
- Poll for status bits

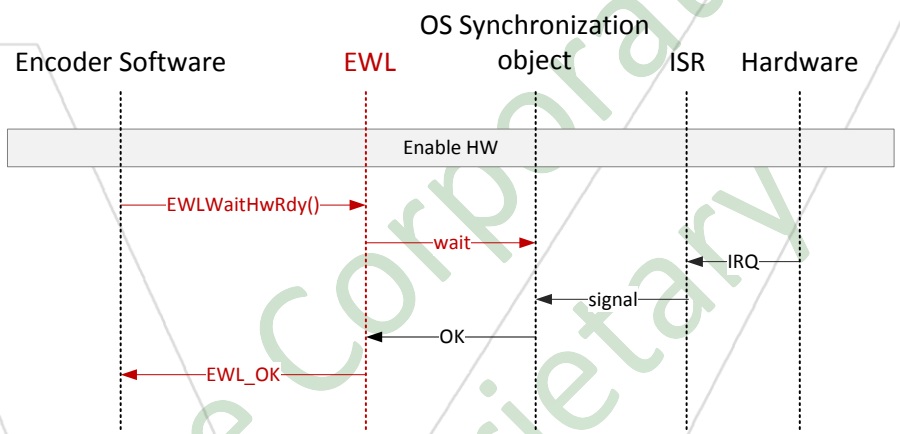

Figure 9. Encoder HW/SW Synchronization - Normal Case

If the hardware processing takes too long time, the wait for it could timeout as shown in the figure below. In this kind of situation, the frame is lost; the control code will reset the hardware and return an error value to the application.

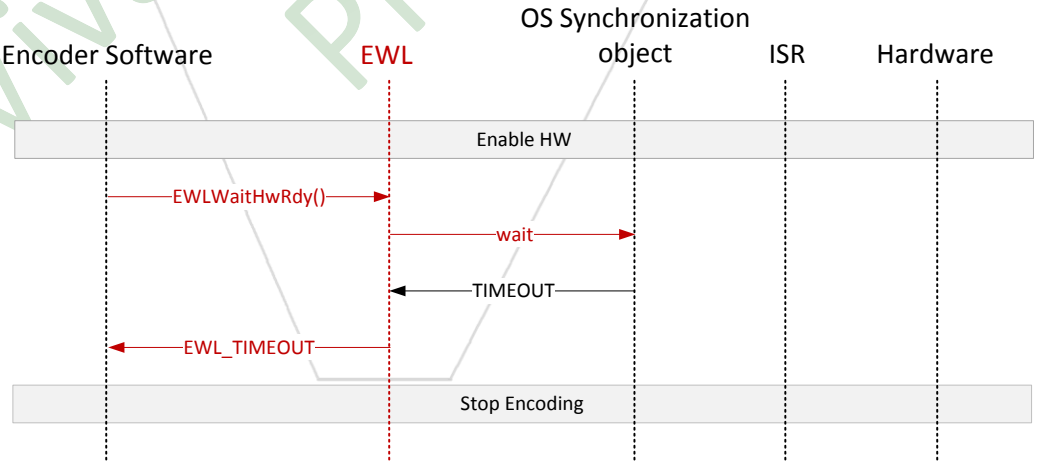

Figure 10. Encoder HW/SW Synchronization - Timeout Case

Also, any error situation has to be returned to the encoder software as illustrated in the figure below. This error will cause the currently processed data to be lost, the hardware to be reset and an error value returned to the application.

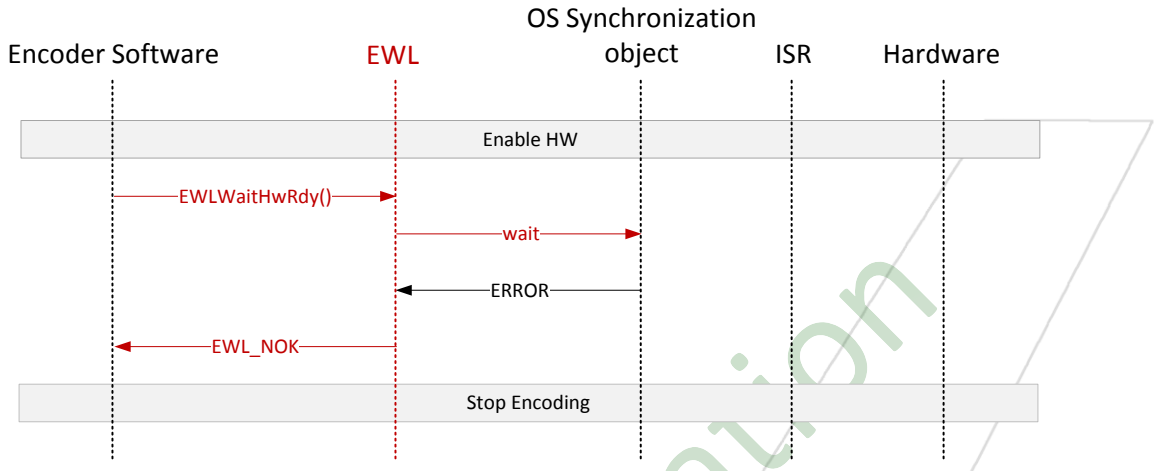

Figure 11. Encoder HW/SW Synchronization - Error Case

Every time the `EWLWaitHwRdy` returns the up-to-date status of the hardware it must be available for the software to read (`EWLReadReg`). This means that if any buffering of the register accesses is done, the EWL cached register values must be refreshed.

6.2.6 Video Stabilization

When the video stabilization is enabled within the encoder preprocessing block, no special consideration must be taken. The only difference is that the encoder will process 2 frames at a time, one will be encoded and the other will be stabilized. `H264EncStrmEncode` will encode the previously stabilized picture and stabilize a new picture. When the function returns both the encoding and stabilization functions are done.

The video stabilization can be used in standalone mode, controlled by its own API. The same EWL layer is used here also (it will be initialized with the stabilization ID). In this mode it has to acquire full exclusive access of the hardware which means that no other processing can be done at the same time (See Section 6.2.3 Hardware Sharing for Multi-Instance Encoding). The hardware configuration and HW/SW synchronization are done in a same way as at the encoder side (See Section 6.2.5 HW/SW Synchronization).

6.2.7 H.264 Multicore Software Architecture

1) The application creates several threads to run the encoder control software. The number of threads depends on the amount of physical hardware encoder cores available. To utilize all the available cores the same number of encoding threads is needed. One more encoding thread is needed for maximum hardware core utilization. This extra thread can run the control software to prepare the next frame and start a hardware core immediately when it becomes available.

   The encoder testbench is used as an example implementation of a video encoder application. The main thread retrieves the frames in capture order from the camera or reads from an input file. The capture order is maintained by passing the frames through a FIFO to the encoding threads.

   The encoding threads share a single instance of the encoder. The threads call the API function `EnterCritical` to lock the instance to make sure the encoding order is equal to the capture order of frames. The encoding threads then get the next frame from FIFO and call the API function to encode the frame. The encoded stream is then fed to a transport channel or written to a file in encoding order.

2) Encoding of a video frame takes place in the API function `EncodeFrame`. Upon calling this function the instance has already been locked thus it is made sure that this frame is indeed the next frame in encoding order.

   The control software handles rate control, pre-encoding frame skip, obtaining correct reference and reconstruction frame buffers and updating the encoding probabilities based on previously encoded frames. This can happen before a physical hardware core is reserved for this frame.

   The control software then obtains a hardware core to encode this frame. The `EWLReserveHW` function will block until a hardware core becomes available. Once a core is available it reserves the core and outputs its number.

   Based on the reserved core number the control software writes to the core's registers and enables it to encode the frame. The control software can then exit the critical section since the encoding order is preserved, and the other threads may go on with either starting the next frame or finalizing the previous frames.

   The thread stops to wait for the core to finish encoding. Once the IRQ is received, the thread enters the critical section again to update the statistics of the encoded frame into the common instance. Finally, the thread exits the critical section and returns the newly created video stream.

3) The Encoder Wrapper Layer serves as the interface between OS independent control software and the OS specific low-level hardware driver. In the reference Linux implementation of EWL, IOCTLs are used to communicate between the user-space EWL and the kernel driver.

   EWL tracks the use of cores, which ones are free and which ones are reserved.

4) The kernel driver is configured according to the physical hardware. It must know the number of hardware cores that exist and their addresses.

6.3 Encoder Wrapper Layer (EML) Structures

6.3.1 EWLHwConfig_t

A structure containing the EWL hardware configuration parameters.

EWLHwConfig_t Members	Type	Description
maxEncodedWidth	u32	Specified the maximum supported picture width (in pixels) for video encoding.
h264Enabled	u32	Hardware supports H.264. 0: not supported; 1: supported
jpegEnabled	u32	Hardware supports JPEG encoder. 0: not supported; 1: supported
vp8Enabled	u32	VP8 supported by hardware. 0: not supported; 1: supported
vsEnabled	u32	Video stabilization supported by hardware. 0: not supported; 1: supported
rgbEnabled	u32	RGB to YUV conversion supported by hardware. 0: not supported; 1: supported
searchAreaSmall	u32	Search area 0: default vertical motion search +/- 31 pixels 1: smaller search area memory is used. This will limit the vertical motion vectors to +/- 15 pixels.
inputYuvTiledEnabled	u32	Always set to 0.
busType	u32	Hardware bus connection in use: 0: Unknown 1: AHB 2: OCP. 3: AXI 4: PCI 5: AXIAHB 6: AXIAPB
busWidth	u32	Bus width of hardware: 0: Unknown 1: 32 bits 2: 64 bits 3: 128 bits
synthesisLanguage	u32	Systhesis language. Informational field only. 0: Unknown 1: VHDL 2: Verilog

6.3.2 EWLInitParam_t

A structure used to pass parameters during initialization of the encoder wrapper layer.

EWLInitParam_t Members	Type	Description
clientType	u32	Client Type: 0: H.264 1: JPEG 2: VP8 3: Standalone video stabilization

6.3.3 EWLLinearMem_t

A structure containing the allocated linear memory area information.

EWLLinearMem_t Members	Type	Description
virtualAddress	u32 *	Pointer to the virtual address of the allocated memory area. This is used for software accesses only.
busAddress	u32	DMA address of the allocated memory area. This is used for hardware accesses.
size	u32	The requested memory area size in bytes. At return it has to reflect the exact size of the allocated memory area.

6.4 Encoder Wrapper Layer (EWL) Interface Functions

The EWL interface provides all the resources that are system dependent, such as physical memory allocation, hardware I/O register access, synchronization routines, etc. The interface is defined in the `ewl.h` file.

EWLReadAsicID

Description:
This function returns the ID of the encoder hardware. This function shall not require the EWL initialization. This is a purely informational function, which helps in identifying the hardware version and its implementation. It will not affect the functionality of the product.

Syntax:

```
u32 EWLReadAsicID ( void );
```

Parameters:

None

Returns:
The hardware ID number of the encoder hardware.

EWLReadAsicConfig

Description:
This function returns the static configuration information of the hardware. This function does not require the EWL initialization. The information returned contains the maximum supported picture width, the enabled/disabled status of the different hardware components and other hardware build time information. It is important that the capability fields (maximum width, component enabled flags) are returning correct values because the top-level software relies on them. This information is available in one of the hardware registers. Refer to the *Hantro VC8000NanoE Hardware Integration Guide* for more details.

Syntax:

```
EWLHwConfig_t EWLReadAsicConfig ( void );
```

Parameters:

None

Returns:
`EWLHwConfig_t` structure

EWLInit

Description:
This function is called to create and initialize a new instance of the system wrapper layer. No EWL provided resource is available until the initialization is successfully completed.

Syntax:

```
const void *EWLInit (
        EWLInitParam_t              *param
);
```

Parameters:

param	Points to the EWLInitParam_t structure that contains the client type parameter.

Returns:

A pointer to the newly created EWL instance or NULL if an error occurred.

EWLRelease

Description:
This function is called to release an instance of the system wrapper. This function releases the entire wrapper resources. After the release of the resource, no calls to the EWL (EWLReadAsicID is the exception) may be done.

In case of an error during the release, the function will return an error code and the instance will not be valid anymore. The client should not attempt to release again.

Syntax:

```
i32 EWLRelease (
        const void    *instance
);
```

Parameters:

instance	A pointer to the EWL instance to be released.

Returns:
Returns 0 for a successful release or a negative error code in case of a failure.

EWLMallocRefFrm

Description:
This function is called to allocate a memory block for a reference frame. The buffer has to be a contiguous, linear memory buffer residing in the physical memory. The caller specifies the minimum size of the buffer to be allocated. The returned size might be larger if a system specific alignment has to be satisfied (e.g., page size alignment). Also the base of the allocated buffer has to be properly aligned with the system data bus width (32-bit and 64 bit data buses are supported by the encoder hardware).

These memory areas will NOT be accessed by the encoder software at any time, so no caching issues will need to be considered.

Syntax:

```
i32 EWLMallocRefFrm (
      const void                *instance,
      EWLLinearMem_t            *info
);
```

Parameters:

instance	A pointer to the EWL instance.
info	Pointer to the EWLLinearMem_t structure for the returned memory block.

Returns:
Returns 0 for a successful allocation or a negative error code in case of a failure.

EWLFreeRefFrm

Description:
This function is called to free a memory block, previously allocated by EWLMallocRefFrm. The EWL instance that will free the block has to be the same instance which allocated it.

Syntax:

```
void EWLFreeRefFrm (
      const void                *instance,
      EWLLinearMem_t            *info
);
```

Parameters:

instance	A pointer to the EWL instance.
info	Pointer to the EWLLinearMem_t structure for the memory block to free. This should be exactly the same information returned by the EWLMallocRefFrm.

Returns:
None

EWLMallocLinear

Description:
This function is called to allocate memory for a SW/HW shared buffer. The buffer has to be a contiguous, linear memory buffer residing in the physical memory. The caller has to specify the minimum size of the buffer to be allocated. The returned size might be larger if a system specific alignment has to be satisfied (e.g., page size alignment). Also the base of the allocated buffer has to be properly aligned with the system data bus width (AHB bus is 32-bit aligned).

These memory areas will be accessed both by the encoder software and hardware, so if caches are in use, the coherency of the cached data has to be assured.

Syntax:

```
i32 EWLMallocLinear (
        const void              *instance,
        u32                     size,
        EWLLinearMem_t          *info
);
```

Parameters:

inst	A pointer to the EWL instance.
size	Size of the requested memory buffer (in bytes).
info	Pointer to the EWLLinearMem_t structure for the memory block allocated.

Returns
Returns 0 for a successful allocation or a negative error code in case of a failure.

EWLFreeLinear

Description:
This function is called to free a memory block, previously allocated by EWLMallocLinear. The EWL instance that will free the block has to be the same which allocated it.

Syntax:

```
void EWLFreeLinear (
        const void              *instance,
        EWLLinearMem_t          *info
);
```

Parameters:

inst	A pointer to the EWL instance.
info	Pointer to the EWLLinearMem_t structure for the memory block to free.

Returns:
None

EWLReadReg

Description:

- This function is called to read a 32-bit value from a specified hardware register. The instance in use can identify the client which is attempting the read access. Accesses to registers shall be done just after an exclusive right is obtained with the successful call of **Error! Reference source not found.**. If such a non-exclusive access should occur, it signals a flaw in the client design. Refer to the *Hantro VC8000NanoE Video Encoder V5.0.x Software Accessible Registers* for a detailed description of the registers.

Syntax:

```
u32 EWLReadReg (
      const void              *instance,
      u32                     offset
);
```

Parameters:

instance	A pointer to the EWL instance.
offset	Register offset is relative to the hardware ID register (#0) in bytes.

Returns:
The value of the specified hardware register.

EWLWriteReg

Description:
This function is called to write a 32-bit value to a specified hardware register. The instance in use can identify the client which is attempting the write access. Accesses to registers shall be done just after an exclusive right is obtained with the successful call of **Error! Reference source not found.**. If such a non-exclusive access occurs, it signals a flaw in the client design. Refer to the *Hantro VC8000NanoE Video Encoder V5.0.x Software Accessible Registers* for a detailed description of the registers.

Syntax:

```
void EWLWriteReg (
      const void              *instance,
      u32                     offset,
      u32                     value
);
```

Parameters:

instance	A pointer to the EWL instance.
offset	Register offset is relative to the hardware ID register (#0) in bytes.
value	Value to write to the hardware register.

Returns
None

EWLEnableHW

Description:
This is a particular register writing function that enables the hardware by writing a specific register. At this point it is expected that the hardware has been properly setup and can start processing. The instance in use can identify the client which is attempting the write access. Accesses to registers shall be done just after an exclusive right is obtained with the successful call of **Error! Reference source not found.**. If such a non-exclusive access occurs, it signals a flaw in the client design. Refer to the *Hantro VC8000NanoE Video Encoder V5.0.x Software Accessible Registers* for a detailed description of the registers.

Syntax:

```
void EWLEnableHW (
        const void              *instance,
        u32                     offset,
        u32                     value
);
```

Parameters:

instance	A pointer to the EWL instance.
offset	Register offset.
value	Value to enable hardware.

Returns
None

EWLDisableHW

Description:
This is a particular register writing function that stops and disables the hardware by writing a specific register. The instance in use can identify the client which is attempting the write access. Accesses to registers shall be done just after an exclusive right is obtained with the successful call of **Error! Reference source not found.**. If such a non-exclusive access occurs, it signals a flaw in the client design. Refer to the *Hantro VC8000NanoE Video Encoder V5.0.x Software Accessible Registers* for a detailed description of the registers.

Syntax:

```
void EWLDisableHW (
        const void              *instance,
        u32                     offset,
        u32                     value
);
```

Parameters:

instance	A pointer to the EWL instance.
offset	Register offset.
value	Value to disable hardware.

Returns:
None

EWLWaitHwRdy

Description:
This function is called to synchronize the software with the hardware. The instance in use can identify the calling client. The returned value will indicate the result of the waiting process. The function is called after enabling the hardware to wait for interrupt from the hardware. If the slicesReady pointer is given, at input it will contain the number of completed slices already received. The function will return when the hardware has finished encoding the next slice, in case the slice interrupts are enabled by config. Upon return, the slicesReady pointer will contain the number of slices that are ready and available in the hardware output buffer.

Syntax:

```
i32 EWLWaitHwRdy (
      const void                *instance,
      u32                       *slicesReady
);
```

Parameters:

instance	A pointer to the EWL instance.
slicesReady	A pointer to a 32-bit variable. If not null, the variable is used for input and output. As input this variable will store the amount of slices received for the current frame. As output from the function the variable will hold the amount of complete slices in hardware output buffer. This value will be passed on to the SliceReadyCallback of the H.264 API.

Returns:
EWL_HW_WAIT_OK, EWL_HW_WAIT_ERROR, EWL_HW_WAIT_TIMEOUT

EWLReserveHw

Description:
This function is part of the hardware sharing mechanism in use for multi-instance encoding support. The instance in use can identify the calling client. When called, it shall block and return when exclusive access to the required hardware can be granted to the calling client.

Syntax:

```
i32 EWLReserveHw (
      const void                *instance,
);
```

Parameters:

instance	Pointer to the EWL instance.

Returns:
Returns 0 for a successful reservation or a negative error code in case of a failure.

EWLReleaseHw

Description:
This function is part of the hardware sharing mechanism in use for multi-instance encoding support. The instance in use can identify the calling client. When called it shall release a previously reserved hardware resource. After this call, another client can acquire the exclusive hardware access.

Syntax:

```
void EWLReleaseHw (
      const void                *instance
);
```

Parameters:

instance	A pointer to the hardware resource instance to be released.

Returns:
None

EWLmalloc

Description:
This function allocates a memory chunk of at least *n* bytes. It has same functionality as the ANSI C malloc.

Syntax:

```
void *EWLmalloc (
      u32                       n
);
```

Parameters:

n	Size of memory block in bytes to be allocated.

Returns:
On success, a pointer to the memory block allocated by the function. If the function failed to allocate the requested block of memory, a NULL pointer is returned.

EWLcalloc

Description:
This function allocates an array in memory with elements initialized to 0. It has the same functionality as the ANSI C calloc. *Note: Control software relies that the memory area it allocates using this function is initialized to zero. Not doing so may result in invalid operation.*

Syntax:

```
void *EWLcalloc (
        u32                     n,
        u32                     s
);
```

Parameters:

n	Number of elements to allocate.
s	Size of each element in bytes.

Returns
On success, a pointer to the memory block allocated by the function. If the function failed to allocate the requested block of memory, a NULL pointer is returned.

EWLfree

Description:
This function deallocates or frees a memory block. It has the same functionality as the ANSI C free().

Syntax:

```
void EWLfree (
        void                    *p
);
```

Parameters:

p	Pointer to a memory block previously allocated with EWLmalloc() or EWLcalloc().

Returns
None.

EWLmemcpy

Description:
This function copies specified number of characters from one buffer to another. It has the same functionality as the ANSI C memcpy.

Syntax:

```
void *EWLmemcpy (
        void                    *d,
        const void              *s,
        u32                     n
);
```

Parameters:

d	Pointer to the destination array where the content is to be copied.
s	Pointer to the source of data to be copied.
n	Number of bytes to copy.

Returns
A pointer to the destination array, d, is returned.

EWLmemset

Description:
This function sets buffers to a specified character. Sets the first *n* chars of *d* to the character *c*. It has same functionality as the ANSI C memset.

Syntax:

```
void *EWLmemset (
        void                    *d,
        i32                     c,
        u32                     n
);
```

Parameters:

d	Pointer to the block of memory to fill.
c	Value to be set.
n	Number of bytes to be set to the value c.

Returns:
A pointer to the block of memory to fill, d, is returned.

EWLmemcmp

Description:
Compares the first n bytes of the block of memory pointed to by s1 to the first n bytes of the block of memory pointed to by s2.

Syntax:

```
int EWLmemcmp (
        void                    *s1,
        const void              *s2,
        u32                     n
);
```

Parameters:

s1	Pointer to block of memory for string1.
s2	Pointer to block of memory for string2.
n	Number of bytes to compare.

Returns
The function returns zero if the two strings are identical, otherwise returns the difference between the first two differing bytes.

6.5 OS Porting Example

The encoder software has been designed so that when porting it the Encoder Wrapper Layer (EWL) hides the OS limitations and OS specific issues from the algorithms and control software. In most of the cases the EWL will be the only part that will need attention when porting, but exceptions may still exist. The reference porting located in the '`linux_reference`' folder is described as an example. The encoder software is built up as a user space static library.

NOTE: The encoder's platform independent code is re-entrant but not totally multi-thread safe. By default, it is not safe to control one instance of the encoder over multiple threads, but it is safe to have multiple instances of the encoder each running and controlled in its own thread.

The overall thread-safety and re-entrance is very much dependent on how the EWL is implemented. In the example porting it is safe to allocate one instance for each of the encoders, but it is not allowed to have multiple instances of the same encoder (this limitation is mainly because of the large linear physical memory needs). Many of the Linux system calls can be interrupted by signals so this needs special consideration.

6.5.1 EWL Initialization and Release

The EWLInit function is provided so that the EWL can initialize itself before any other EWL call is made. The availability of the resources required by the encoder must be ensured at this initialization point. The resources needed by the encoder software are:

- linear, contiguous physical memory for hardware related buffers
- memory for SW needs
- hardware register access
- HW/SW synchronization
- hardware resource sharing

One way to get linear, contiguous physical memory in Linux is to instruct the kernel at loading time not to use the whole available RAM memory. Linux can leave the top of RAM unused, which an application can access by mapping it with the help of the memory device '`/dev/mem`'.

The same Linux memory device can be used to map the hardware I/O registers to a virtual address space. This way the encoder can have direct pointer access to them.

The HW/SW synchronization can be done with hardware polling or IRQ. The polling is done by reading the hardware status register at fixed time intervals. If IRQ is used, then a kernel device driver has to be created because IRQs can be served only in the Linux kernel space. To notify the encoder in user space, the kernel device driver can send a SIGIO signal to the listening encoder process. This signal will be sent any time an IRQ is received from the encoder hardware.

The advantages of the polling method are that the whole encoder software can reside in the user space and does not need a kernel driver for handling IRQs. The drawback is that the polling is usually done at fixed time intervals. If this interval is short the encoder will use more CPU and when it is longer the overall encoder performance is affected.

The IRQ based method is somehow more complicated to implement but it could assure the best encoder performance. Here also the IRQ response latency and any context switch will cause performance drops. The encoder can sleep during the IRQ wait and doing so will free the CPU for other tasks. In multi-instance environment there has to be a way to deliver the IRQ just to the instance that has reserved the hardware for its use.

Two implementations for the Linux EWL are provided as example code. One is using the polling method and the source code is in '`ewl_x280_polling.c`' file. The other method which relies on IRQ from the hardware is

implemented in files ‘ewl_x280_irq.c’ and ‘hx280enc.c’. The first file is the user space part of the EWL, which will be compiled together with the codec library. The second file is the kernel part of the EWL: a kernel driver which must be compiled separately and loaded into the kernel. Common parts are implemented in ‘ewl_x280_common.c’.

For software development purposes there is a different kind of EWL implementation: ‘ewl_system.c’. The purpose is to allow software development without the hardware. Instead of using the hardware encoder, these EWLs utilize the bit-exact system model of the hardware.

EWL system model implementation replaces the whole hardware with the system model allowing control software development and testing in any environment.

6.5.2 Linear Memory Allocation

A simple linear memory management module, memalloc, is provided to help with the allocation. It manages a predefined table of linear memory chunks. At request, it returns the bus address of the first free chunk that has at least the desired size. The EWL will then map this memory area to the user space by using /dev/mem. The predefined chunks table can provide memory for running all the encoders concurrently at maximal resolution. This needs more than then 64MB of free RAM.

6.5.3 SW/SW Memory Handling

All the SW/SW memory-related EWL calls can be implemented by using their ANSI-C counterparts, as can be seen in the example implementation source files. If for any reason the full ANSI-C library is not available in the target system, then these functions have to be implemented in a more system specific way but offering the same functionality.

6.5.4 Hardware Register Access

The hardware registers are mapped to the user space at the EWL initialization phase when a pointer to the base of the register bank is provided. The EWLReadReg and EWLWriteReg functions will read and write a register having a specific offset.

6.5.5 Hardware Sharing

To be able to run multiple encoder instances at the same time there must be a global way of controlling the access to only one encoder hardware. Under Linux a process semaphore is used to control this exclusive access to the shared hardware. The semaphore is created the first time an encoder is initialized and after that any new instance will just use it in order to get the exclusive access. Example of handling process semaphores can be seen in ewl_linux_lock.c.

The standalone video stabilization is using the same hardware as any of the encoders and it also needs exclusive access to it.

6.6 Building and Configuring the Software

The whole VC8000NanoE software library can be built up using the provided `Makefile`. `Makefile` must be edited to match the host and target environment settings. The encoder software will be built as a static library.

When full encoder software sources are available, some of the encoder parameters can be altered. The default values and descriptions for these configurable parameters are defined in '`enccfg.h`'. All the parameters alter the way the encoder software and hardware works. Refer to the *Hantro VC8000NanoE Hardware Integration Guide* for detailed description). You can override the default configuration by defining the flags for the compiler in the '`Makefile`'.

6.6.1 Common Encoder Configuration

Common encoder settings are described in the table below. These affect the functionality of all encoders included in the VC8000NanoE product.

Table 21. Common Encoder Configuration Parameters

Macro	Description	Values
ENCH1_AXI_WRITE_ID	Identification value for hardware write accesses when connected to an AXI master bus interface.	[0,255]
ENCH1_AXI_READ_ID	Identification value for hardware read accesses when connected to an AXI master bus interface.	[0,255]
ENCH1_INPUT_SWAP_32_YUV	Hardware YUV input picture data swapping for each pair of 32-bit words.	0 – no swap 1 – swap
ENCH1_INPUT_SWAP_16_YUV	Hardware YUV input picture data swapping for each pair of 16-bit words.	0 – no swap 1 – swap
ENCH1_INPUT_SWAP_8_YUV	Hardware YUV input picture data swapping for each pair of bytes.	0 – no swap 1 – swap
ENCH1_INPUT_SWAP_32_RGB16	Hardware 16-bit RGB input picture data swapping for each pair of 32-bit words.	0 – no swap 1 – swap
ENCH1_INPUT_SWAP_16_RGB16	Hardware 16-bit RGB input picture data swapping for each pair of 16-bit words.	0 – no swap 1 – swap
ENCH1_INPUT_SWAP_8_RGB16	Hardware 16-bit RGB input picture data swapping for each pair of bytes.	0 – no swap 1 – swap
ENCH1_INPUT_SWAP_32_RGB32	Hardware 32-bit RGB input picture data swapping for each pair of 32-bit words.	0 – no swap 1 – swap
ENCH1_INPUT_SWAP_16_RGB32	Hardware 32-bit RGB input picture data swapping for each pair of 16-bit words.	0 – no swap 1 – swap
ENCH1_INPUT_SWAP_8_RGB32	Hardware 32-bit RGB input picture data swapping for each pair of bytes.	0 – no swap 1 – swap
ENCH1_OUTPUT_SWAP_32	Hardware output data swapping for each pair of 32-bit words.	0 – no swap 1 – swap
ENCH1_OUTPUT_SWAP_16	Hardware output data swapping for each pair of 16-bit words.	0 – no swap 1 – swap
ENCH1_OUTPUT_SWAP_8	Hardware output data swapping for each pair of bytes.	0 – no swap 1 – swap
ENCH1_BURST_LENGTH	Maximum burst length for hardware bus transactions. Value 0 in AHB means INCR type bursts will be generated. For AXI a 0 value is forbidden.	[0,4,8,16] for AHB, [1,16] for AXI
ENCH1_BURST_INCR_TYPE_ENABLED	INCR type burst mode control	0 - enable INCR type bursts 1 - disable INCR type and use SINGLE instead

Macro	Description	Values
ENCH1_BURST_DATA_DISCARD_ENABLED	Data discard mode. When enabled read bursts of length 2 or 3 are converted to BURST4 and useless data is discarded. Otherwise use INCR type for that kind of read bursts.	0 - disabled 1 - enabled
ENCH1_ASIC_CLOCK_GATING_ENABLED	Hardware internal clock gating control.	0 - disabled 1 - enabled
ENCH1_IRQ_DISABLE	Disables the hardware interrupts; software must poll the status register in order to get the hardware status.	0 – IRQ enabled 1 – IRQ disabled
ENCH1_TIMEOUT_INTERRUPT	Enable hardware timeout interrupt.	0 - disabled 1 – enabled
ENCH1_SLICE_READY_INTERRUPT	Enable hardware interrupt after every slice.	0 - disabled 1 – enabled
ENCH1_INPUT_READ_CHUNK	Defines the input picture read chunk size used by the hardware.	0 – 4 MBs 1 – 1 MB

6.6.2 Data Endianness Configuration

Internally the hardware encoder handles all data in big endian mode. The bits for data swapping must be configured properly to maintain correct data endianness. The following table shows an example of swap bit settings in different endianness environments and hardware data bus widths.

Table 22. Data Endianness Configuration Examples

Macro	32-bit little endian	32-bit big endian	64-bit little endian	64-bit big endian
ENCH1_INPUT_SWAP_32_YUV	0	0	1	0
ENCH1_INPUT_SWAP_16_YUV	1	0	1	0
ENCH1_INPUT_SWAP_8_YUV	1	0	1	0
ENCH1_INPUT_SWAP_32_RGB16	0	0	1	0
ENCH1_INPUT_SWAP_16_RGB16	1	0	1	0
ENCH1_INPUT_SWAP_8_RGB16	0	0	0	0
ENCH1_INPUT_SWAP_32_RGB32	0	0	1	0
ENCH1_INPUT_SWAP_16_RGB32	0	0	0	0
ENCH1_INPUT_SWAP_8_RGB32	0	0	0	0
ENCH1_OUTPUT_SWAP_32	0	0	1	0
ENCH1_OUTPUT_SWAP_16	1	0	1	0
ENCH1_OUTPUT_SWAP_8	1	0	1	0

6.6.3 Standalone Video Stabilization Configuration

The standalone video stabilization can be configured in the same way as the encoder. The default values and descriptions for these parameters can be found in 'vidstabcfg.h'. The encoder's and stabilization's configurations are independent and influence only the behavior of the encoder and stabilization, respectively.

Table 23. Standalone Video Stabilization Configuration Parameters

Macro	Description	Values
VSH1_AXI_WRITE_ID	See ENCH1_AXI_WRITE_ID	[0,255]
VSH1_AXI_READ_ID	See ENCH1_AXI_READ_ID	[0,255]
VSH1_INPUT_SWAP_32_YUV	See ENCH1_INPUT_SWAP_32_YUV	[0,1]
VSH1_INPUT_SWAP_16_YUV	See ENCH1_INPUT_SWAP_16_YUV	[0,1]
VSH1_INPUT_SWAP_8_YUV	See ENCH1_INPUT_SWAP_8_YUV	[0,1]
VSH1_INPUT_SWAP_32_RGB16	See ENCH1_INPUT_SWAP_32_RGB16	[0,1]
VSH1_INPUT_SWAP_16_RGB16	See ENCH1_INPUT_SWAP_16_RGB16	[0,1]
VSH1_INPUT_SWAP_8_RGB16	See ENCH1_INPUT_SWAP_8_RGB16	[0,1]
VSH1_INPUT_SWAP_32_RGB32	See ENCH1_INPUT_SWAP_32_RGB32	[0,1]
VSH1_INPUT_SWAP_16_RGB32	See ENCH1_INPUT_SWAP_16_RGB32	[0,1]
VSH1_INPUT_SWAP_8_RGB32	See ENCH1_INPUT_SWAP_8_RGB32	[0,1]
VSH1_BURST_LENGTH	See ENCH1_BURST_LENGTH	[1,4,8,16] for AHB. [1,16] for AXI.
VSH1_BURST_INCR_TYPE_ENABLED	See ENCH1_BURST_INCR_TYPE_ENABLED	[0,1]
VSH1_BURST_DATA_DISCARD_ENABLED	See ENCH1_BURST_DATA_DISCARD_ENABLED	[0,1]
VSH1_ASIC_CLOCK_GATING_ENABLED	See ENCH1_ASIC_CLOCK_GATING_ENABLED	[0,1]
VSH1_IRQ_DISABLE	See ENCH1_IRQ_DISABLE	[0,1]

6.6.4 Internal Debug Tracing

Internal debugging traces can be enabled by defining in the 'Makefile' any of the following:

- _ASSERT_USED – enables the use of asserts for runtime invalid value checking. Requires the implementation of ASSERT macro (See 'encdebug.h').
- _DEBUG_PRINT – enables printing of debug information from the encoder modules. Requires the implementation of DEBUG_PRINT macro (See 'encdebug.h').
- TRACE_EWL – enables trace messages from the EWL implementation. Requires the implementation of a PTRACE macro (See the example EWL implementations).
- TRACE_REGS – writes traces from the hardware registers into a file
- TRACE_STREAM – writes traces from output stream writing into a file. Requires the tracing functions defined in 'enctracestream.h' and uses the macros COMMENT and TRACE_BIT_STREAM defined in 'encdebug.h'.

The implementation of the trace functions is provided in the vc8000ne_encoder/software/linux_reference/debug_trace folder.

Define HH1_HAVE_ENCDEBUG_H if 'encdebug.h' file is available.
Define HH1_HAVE_ENCTRACE_H if 'enctrace.h' file is available.

6.6.5 API Tracing

The API entries and exits can be traced by defining `VP8ENC_TRACE, H264ENC_TRACE, JPEGENC_TRACE` or `VIDEOSTB_TRACE`. The API tracing relies on the external implementation of a tracing function whose prototype is defined in the API header, e.g., `H264EncTrace` function defined in `h264encapi.h`. This trace function will get as parameter a null terminated char string. Example implementation that saves the traces to a file can be seen in the testbenches.

6.7 Recommendations for Memory Allocation/Optimization

Because the memory busload during the encoding can get very high, it makes sense to allocate the hardware related buffers to the fastest memory area.

For the reference picture memory, the chrominance data is the most critical. Therefore, it is recommended to give chrominance data a higher priority for using faster memory areas.

7 Testing of the Product

Test benches and reference test data is provided in order to check the final software-hardware integration. Even though the test benches are developed for Linux based platforms, it will be fairly easy to modify and adapt them to any other system. The only possible problem could be that the target platform does not have enough storage capacity for all the raw input data, which can have a considerable size.

7.1 Building H.264/VP8 Test Bench

The testbench is provided for testing the product under Linux environment. It is a command line tool which reads raw YUV image data from a file, uses the encoder API to encode the frames, and writes the output stream to a file. The testbench parameters allow controlling the encoder and testbench functionality. The descriptions for the parameters can be obtained by executing the testbench without parameters.

The testbench is using C standard library functions; the only OS specific thing is the encoder input picture buffer allocation, so it will be easy to adapt it to any other system. The testbench source code is located in: `vc8000ne_encoder/software/linux_reference/test/[h264/vp8]`.

The provided `Makefile` should be edited to match the host and target environment settings.

7.2 Building H.264/VP8/JPEG Test Bench

The provided test data consists of raw input YUV/RGB sequences, output reference stream files and script files containing the test bench parameters. The testing consists of a set of encoder functional test cases, which are designed to check the different encoding algorithms. Each test case has its own set of parameters, which are provided in a script and passed on to the encoder test bench. After running all the test cases the output streams can be compared against the provided reference data.

The reference output data and the scripts stored in `ref_streams/case_xyz` –folder. The input sequences are stored in `ref_yuv` –folder. In order to be able to run the script, `INPUT_YUV` environment value should be set to point to the `ref_yuv` –folder.

Document Revision History

This section describes top level differences in the versions of this document.

Note: This document is not necessarily updated for each patch or minor revision. The information in this document tends to be stable across a revision (nnn) series.

Doc Revision	Date	Compatible cores	Comments
1.02	2020-07-08	VC8000NanoE (for release v5.0.x swreg0 0x6E655000)	Section 4.3.4, Tables 16 and 17: changed Buffer Size unit to Kbytes. Minor enhancements.
1.01	2020-03-23	VC8000NanoE (for release v5.0.x swreg0 0x6E655000)	Section 2.1, Table 1: added details to VP8 encoder support. Section 2.1, Table 3: changed Basic Maximum motion vector length for vertical to +/-30 pixels, horizontal to +/-126 pixels. Section 2.7: Added note for H.264 features that cannot be selected with a multicore configuration. Section 5.1: added performance impact with register access is slow.
1.00	2020-02-10	VC8000NanoE (for release v5.0.x swreg0 0x6E655000)	Converted to VeriSilicon standard format, styles, tables.
0.80	2020-02-06	VC8000NanoE (for release v5.0.x swreg0 0x6E655000)	Initial - adapted from original H1v7 document version 1.31.