Software Integration Guide

7280 Multi-format Encoder

Version 1.0

Copyright Information

Hantro Products Oy
Kiviharjunlenkki 1
90220 Oulu
FINLAND
www.hantro.com

Glossary

720p	High Definition resolution 1280x720, progressive video
API	Application Programming Interface
AVC	Advanced Video Coding, same as H.264
bps	Bits per second
CIF	Common Interchange Format (resolution 352x288 pixels)
EWL	Encoder System Wrapper Layer
fps	Frames per second
IDR	Instantaneous Decoding Refresh
I/O	Input/Output
IRQ	Interrupt Request
ISR	Interrupt Service Routine
H.263	A video coding standard developed by ITU-T
H.264	A video coding standard developed jointly by ITU-T and ISO/IEC
HD	High Definition
HW	Hardware
JPEG	Joint Photographic Experts Group, also a common term for still images.
kB	Kilobyte, 1024 bytes
MB	Macroblock, a data unit consisting of 16x16 luminance samples and 8x8 Cb and Cr samples
MPEG-4	A video coding standard developed by Motion Picture Experts Group
NAL	Network Abstraction Layer
NTSC	National Television System Committee, defines resolution 720x480
OS	Operation System
PAL	Phase Alternate Line (resolution 720x576 pixels)
QCIF	Quarter Common Interchange Format (resolution 176x144 pixels)
QP	Quantization Parameter
QVGA	Quarter Video Graphics Array (resolution 320x240 pixels)
RLC	Run Length Coding
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
SIGIO	Asynchronous signal used in Linux OS
SW	Software
SXGA	Super eXtended Graphics Array (resolution 1280x1024 pixels)
VLC	Variable Length Coding
VGA	Video Graphics Array (resolution 640x480 pixels)
VS	Video Stabilization
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)
YCbCr 4:2:0	YCbCr sampling format, where Cb and Cr components are sub-sampled by two both horizontally and vertically.
YCbCr 4:2:2	YCbCr sampling format, where Cb and Cr components are sub-sampled by two horizontally
YUV	Commonly used alternative for YCbCr
WS	Wait state; unit used in expressing memory access speeds

Version History

Document Version	Changes/Comments
1.0	Original version

Table of Contents

1 Introduction

This document describes the features, functionality and system requirements of the 7280 multi-format 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 H.264, MPEG-4, 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. In chapters 4 and 5 the memory requirements and performance figures of the encoder are described. 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. References are presented in the end of the document.

In the document all functions, parameters, data types and code are described in `Courier new (syntax style)` font. Notes and filenames are written in *italic* expression.

2 Features of the Product

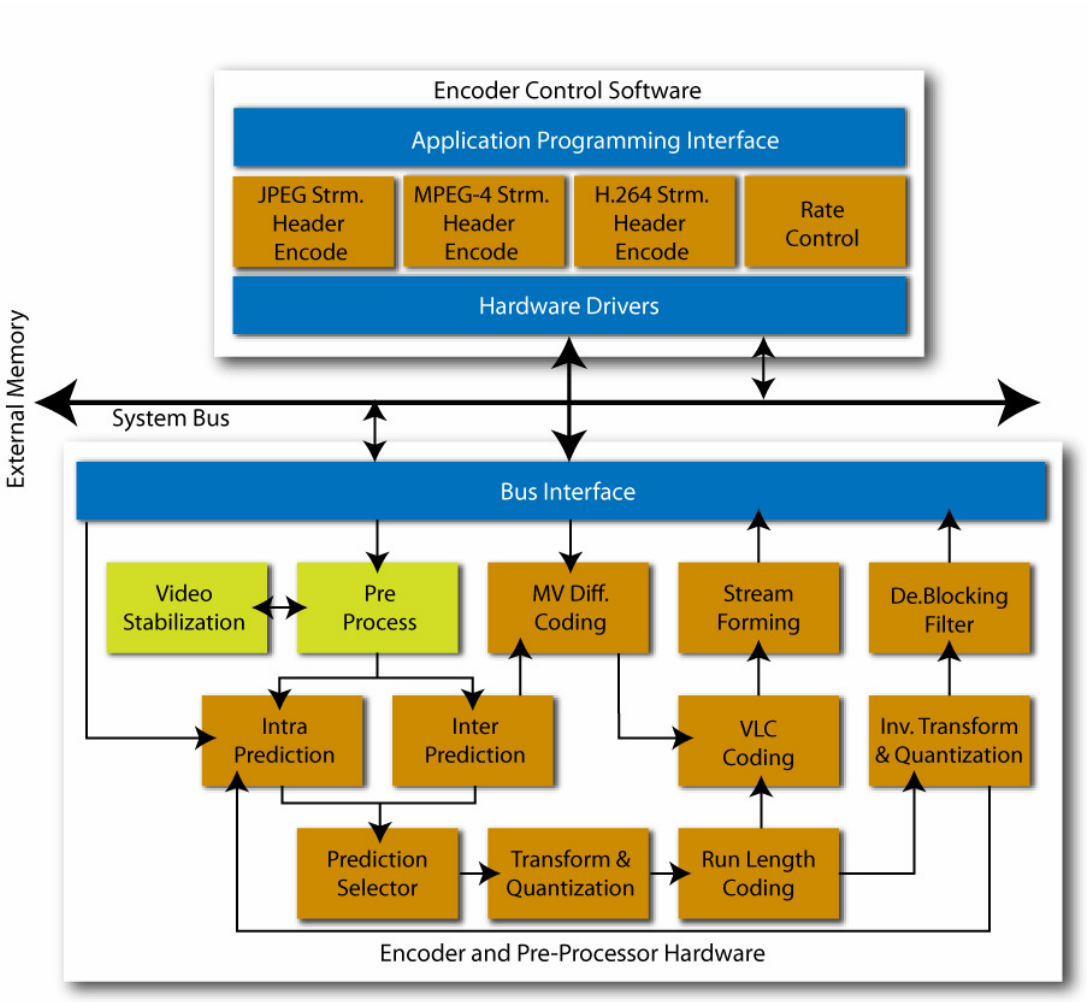

FIGURE 1. 7280 ENCODER FUNCTIONAL BLOCK DIAGRAM.

2.1 Supported standards and tools

The 7280 encoder is capable of encoding H.264, MPEG-4 and JPEG formats that conform to the supported standards, profiles and levels presented in Table 1. The supported video tools are shown in Table 2, Table 3 and Table 4. The block diagram of the encoder is presented in Figure 1.

TABLE 1. SUPPORTED STANDARDS, PROFILES AND LEVELS

Standard	Encoder support
H.264 Profile and level	Baseline Profile, levels 1-3.2
MPEG-4 Visual profile and level	Simple Profile, levels 0-6

	Main Profile, level 4[1)]
H.263 profile and level	Profile 0, levels 10-70. Image size up to 720x576, time code extensions not supported
JPEG profile and level	Baseline

[1)] Only Simple profile tools are supported.

TABLE 2. SUPPORTED H.264 TOOLS

Tool	Encoder support
Slices	I and P slices
Entropy encoding	CAVLC
Basic	• Constrained intra prediction • Maximum MV range +-16 pixels • MV accuracy ¼ pixels • All Block sizes from 4x4 to 16x16 supported • All Intra modes supported
Number of reference frames	1
Maximum number of slice groups	1

TABLE 3. SUPPORTED MPEG-4 VISUAL TOOLS

Visual tool	Encoder support
Basic	• I and P-VOPs • Maximum MV range +-16 pixels • MV accuracy ½ pixels • 1 or 4 MV/Macroblock
Error resilience	• Video Packets • Data partitioning (SW performs entropy encoding) • Reversible VLC
Number of reference frames	1
Quantization	Method 2
Number of visual objects	1
Short Video Header	Yes

TABLE 4. SUPPORTED H.263 VISUAL TOOLS

Visual tool	Encoder support
Basic	• I and P-VOPs • Maximum MV range +-16 pixels • MV accuracy ½ pixels • 1 MV/Macroblock
Error resilience	GOB
Number of reference frames	1

2.2 Encoding features

The features of the encoder for each supported standard are shown in Table 5, Table 6 and Table 7.

TABLE 5. H.264 FEATURES

Feature	Encoder support
Input data format	YCbCr 4:2:0 planar or semi-planar YCbYCr and CbYCrY 4:2:2 Interleaved [1)]
Output data format	H.264 byte or NAL unit stream
Supported image size	• 96 x 96 to 1280 x 1024 • Step size 4 pixels
Maximum frame rate	25 fps at 720x576, or 30 fps at 720x480 or 30 fps at 1280x720, or 30 fps at 1280x1024[2)]
Maximum bit rate	20 Mbps

[1)] Internally 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. The given figure 30 fps at 1280x720 requires logic clock frequency of 181 MHz. 30 fps at 1280x1024 requires logic clock frequency of 257 MHz.

TABLE 6. MPEG-4 / H.263 FEATURES

Feature	Encoder support
Input data format	YCbCr 4:2:0 planar or semi-planar YCbYCr and CbYCrY 4:2:2 Interleaved [1)]
Output data format	MPEG-4 / H.263 elementary video stream
Supported image size	• 96 x 96 to 1280 x 720 • Step size 4 pixels
Maximum frame rate	25 fps at 720x576, or 30 fps at 720x480 30 fps at 1280x720, or 30 fps at 1280x1024[2)]
Maximum bit rate	16 Mbps

[1)] Internally 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. The given figure 30 fps at 1280x720 requires logic clock frequency 160 MHz. 30 fps at 1280x1024 requires logic clock frequency of 227 MHz.

TABLE 7. JPEG FEATURES

Feature	Encoder support
Input data format	YCbCr 4:2:0 planar or semi-planar YCbYCr and CbYCrY 4:2:2 Interleaved [1)]
Output data format	• JFIF file format 1.02 • Non-progressive JPEG
Supported image size	• 80x16 to 4672 x 3504 (16.4 million pixels) • Step size 4 pixels
Maximum data rate	Up to 28 million pixels per second [2)]
Thumbnail encoding	JPEG compressed thumbnails supported

[1] Internally 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 encoder and it can be used only with 7280 encoder. Pre-processing features are presented in Table 8.

TABLE 8. PRE-PROCESSING FEATURES

Feature	Encoder support
Color space conversion	YCbYCr or CbYCrY 4:2:2 Interleaved or semi-planar 4:2:0 to YCbCr 4:2:0
Cropping	JPEG - from 4672 x 4672 to any supported encoding size Video - from 1920 x 1920 to any supported encoding size
Rotation	90 or 270 degrees

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 the two input picture buffer simultaneously. Stabilization functionality requires minimum 8 pixels larger input picture as actual resolution which is wanted to encode. Figure 2 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 4 pixels left. The offsets around the stabilized picture (shown in dark) are cropped out when encoding the video thus creating a more stable video.

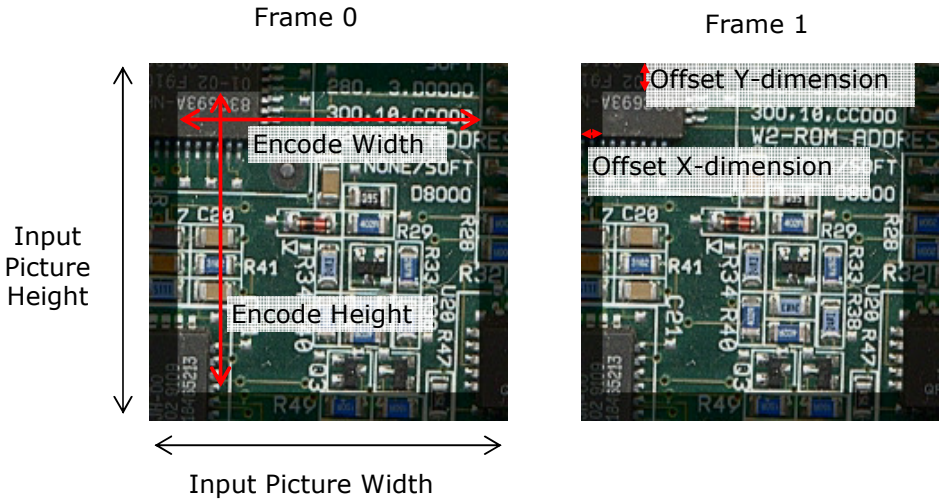

FIGURE 2. STABILIZATION PICTURE DIMENSION

Video stabilization can be used pipelined with 7280 video encoding or in standalone mode when 7280 video encoding is disabled. See more details about the usage of video stabilization in API manuals [2], [3], and [4]. Video stabilization features are explained in Table 9.

TABLE 9. VIDEO STABILIZATION FEATURES

Feature	Encoder support
Maximum stabilization move in pixels for two sequential input video pictures	+-16 pixels
Adaptive motion compensation filter	Motion is filtered based on upto 40 sequential video pictures
Offset around stabilized picture	• Minimum 8 pixels in standalone mode • Minimum 16 pixels when pipelined with video encoder • Recommended 64 pixels • Maximum not limited

2.5 Connectivity features

The encoder supports the connectivity features presented in Table 10. The usage of these features is described in more detail in the Hardware Integration Guide [1]. Note that the endian modes can be separately set for input and output data.

TABLE 10. CONNECTIVITY FEATURES

Feature	Encoder support
AHB precise burst / data discard [1)]	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
Restricting maximum issued OCP burst length	Yes, to any value between 1-31
Interrupt method	Polling or level based interrupting
32-bit little endian	Yes, byte order 3-2-1-0
32-bit big endian	Yes, byte order 0-1-2-3
64-bit little endian	Yes, byte order 7-6-5-4-3-2-1-0
64-bit big endian	Yes, byte order 0-1-2-3-4-5-6-7
Mixed 32-bit little endian in a 64-bit bus	Yes, byte order 3-2-1-0-7-6-5-4
Mixed 32-bit big endian in a 64-bit bus	Yes, byte order 4-5-6-7-0-1-2-3

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

2.6 Product configurable options

Depending on the Licensee needs and the agreement the 7280 Encoder product can be scaled to support limited a feature set. This is called to product configurability.

The 7280 Product can be configured to leave out some of the encoding standards or video stabilization according to Table 11, change the maximum video resolution according to options in Table 12 or to select different bus protocols according to Table 13. The amount of supported video encoding standards affects to needed silicon area.

7280 Encoder product release can be any combination of configurable items.

TABLE 11. 7280 ENCODER PRODUCT DELIVERY CONFIGURATION

7280 Encoder Product Configurability Options
H.264 Video Encoder
MPEG-4/H.263 Video Encoder
JPEG Encoder
Video Stabilization

TABLE 12. 7280 ENCODER PRODUCT CONFIGURABLE MAXIMUM VIDEO RESOLUTION

7280 Encoder Video Resolution Options	Maximum resolution
SXGA	1280x1024
720p	1280x720
D1	720x576
CIF	352x288

TABLE 13. 7280 ENCODER PRODUCT CONFIGURABLE SYSTEM BUS PROTOCOL

7280 System Bus Protocol Options	Interface Definition
OCP 2.0	32-bit or 64-bit master interface 32-bit slave interface
AHB 2.0	32-bit or 64-bit master interface 32-bit slave interface
AXI 1.0	32-bit or 64-bit master interface 32-bit slave interface
AMBA 3 APB 1.0	32-bit slave interface

3 System Overview

3.1 Functionality of the product

In the 7280 encoder the encoding tasks are divided between hardware and software. This task partitioning is described in Figure 1. When starting a picture encoding 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.). A finished stream (no more SW processing needed) is produced when encoding H.264, H.263, JPEG and plain or video packet (no data partitioning) MPEG-4 streams.

When data partitioned MPEG-4 stream is required the hardware will produce RLC data and the needed control data, so that software can produce the final stream. The control data includes:

- Macroblock type
- Quantization parameter
- Motion vectors
- RLC data counters

In this case software performs DC and MV differential encoding, (R)VLC encoding, and creates the finished stream.

In order to synchronize its operations with the software, the hardware can raise an interrupt if one of the following happens:

- A pre-defined macroblock interrupt interval is reached
- Output buffer limit was reached
- Whole picture was encoded
- An error response is 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 2 pictures at the same time (one is encoded and the other one is stabilized). The video stabilization can be run also in standalone mode (without video encoding), controlled by its own API.

3.2 Software composition of the product

The 7280 encoder software is implemented in ANSI-C and its composition can be seen in Figure 3. The 7280 encoder has two main software interfaces. One is the top-level APIs, which are used by any application needing H.264, MPEG-4, H.263 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 Encoder system Wrapper Layer (EWL), which provides system dependent resources to the 7280 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 7280 encoder is ported to a particular system are grouped in a clear 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.

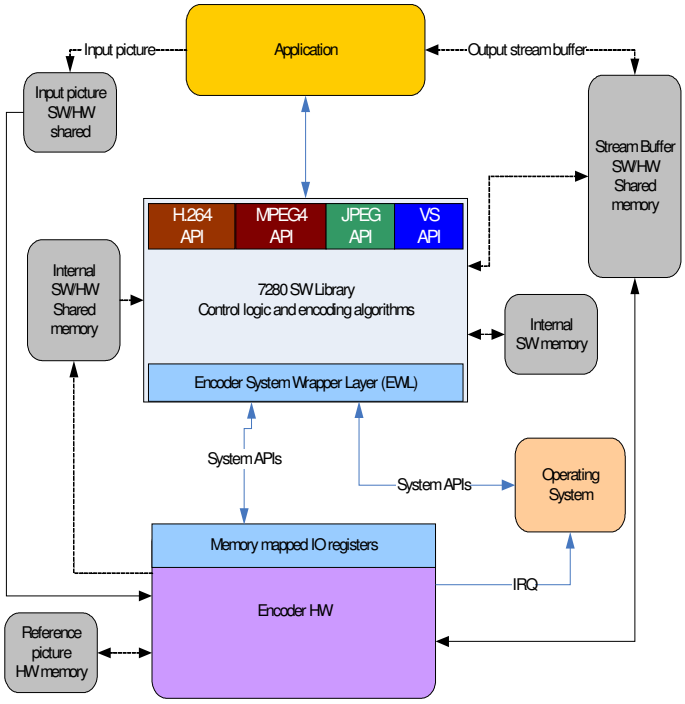

FIGURE 3. INTEGRATED STRUCTURE OF THE ENCODER AND ITS MAIN INTERFACES

As can be seen in Figure 3, the 7280 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 SW/HW Shared Memory). These memory spaces are used to store the hardware generated data for the software. Special care has to 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 have to 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 doesn't 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 has to be allocated also by the application using the encoder. When encoding MPEG-4 data partitioned streams, this buffer is not used by the encoder hardware because the output stream data is produced entirely by software.

The software internals are shown in Figure 4. The software is divided into 5 parts: MPEG-4, H.264, JPEG, Video Stabilization (we call these components) and a common part. The common part 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 (`EWLReleaseHw`) the hardware so that another component instance can acquire it and use it.

The common part of the software takes care of the HW memories, controlling the HW registers and buffering RLC data between the HW and the SW. It uses the EWL functions for reading and writing the HW registers and for polling the HW interrupts. The video stabilization algorithm implementation is also common for all components.

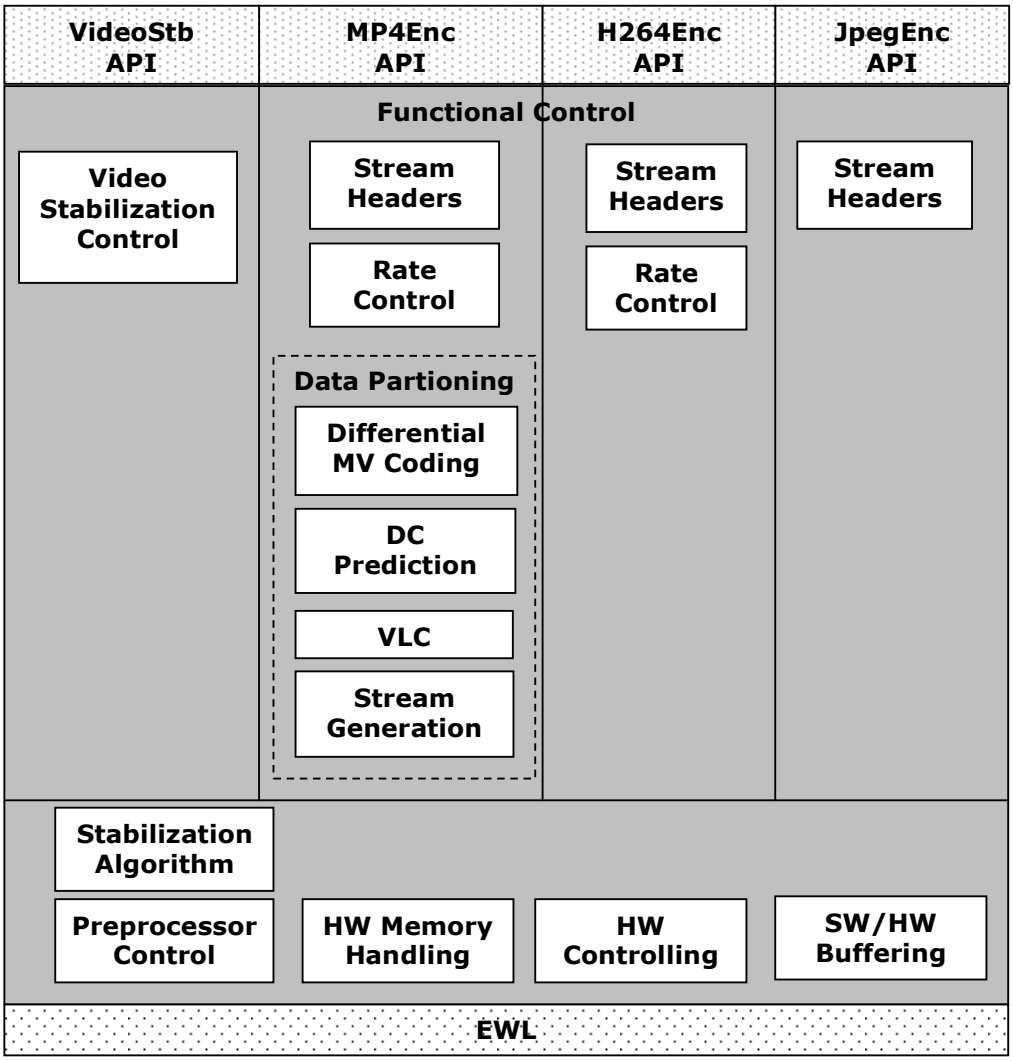

FIGURE 4. ENCODER LIBRARY INTERNAL STRUCTURE AND INTERFACE

4 Memory Requirements

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

4.1 Input picture buffer

The input picture buffer is a memory buffer for HW use only which has to be allocated by external means (i.e. it is not allocated by the encoder software). This will contain the raw YCbCr picture data. The buffer has to be linear and contiguous, and allocated in a memory area accessible by the HW. It also has to 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. Four types of input picture formats are supported. The same kind of input buffer is used for H.264, MPEG-4/H.263 and JPEG codec and video stabilization functionality.

Table 14 represents the amount of memory required for the input picture buffer depending on the input picture format, and the maximum buffer size.

TABLE 14. 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	1 966 080	24 556 032
YCbCr 4:2:0 semiplanar	1 or 2	384	1 966 080	24 556 032
YCbCr 4:2:2 interleaved YCbYCr	1 or 2	512	2 621 440	32 741 376
YCbCr 4:2:2 interleaved CbYCrY	1 or 2	512	2 621 440	32 741 376

[1)] – One SXGA (1280x1024) resolution containing 5120 macroblocks
[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, except the video packaged MPEG-4 streams, the output buffer has to be linear and contiguous, and allocated in a memory area accessible by the HW. It also has to be 64-bit aligned.

As the amount of data produced by the encoder can vary allot, this buffer's limits are not predefined. The only limitation from the encoder is that the buffer has to be big enough for one frame, since if the buffer end is reached while encoding, the current frame will be lost.

On request, H.264 Encoder can return information about the NAL unit sizes and respectively the MPEG-4 Encoder about the video package sizes.

TABLE 15. NAL OR VP SIZE BUFFER

Name	Maximum size [bytes]
H.264 Slice size buffer[1)]	332
MPEG-4 VP size buffer[2)]	14404

[1)] A slice contains at least one row of macroblocks, so the maximum number of coded slices is 80 (rotated SXGA resolution 1024*1280). Encoder can also generate SEI messages and filler NAL units.
[2)] A video package contains at the limit just one macroblock, so the maximum amount of videopackets is 5120 (SXGA resolution).

4.3 H.264 Encoder

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

4.3.1 HW internal buffers

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

Allocations are done with `EWLMallocRefFrm()` (See 6.3.5) which is implemented during the integration phase. 4 buffers are allocated, 2 luminance data and 2 chrominance data buffers. The double buffering is needed in order to simplify the hardware operations, in the sense that one buffer acts as read buffer and the other as write buffer.

TABLE 16. H.264 HW INTERNAL BUFFER SIZE IN BYTES

Name	Amount	Size per MB [byte]	Size per picture [bytes]					
			QCIF	CIF	VGA	PAL	720p	SXGA
Luminance (Y) Buffer	2	256	50688	202752	614400	829440	1843200	2621440
Chrominance (CbCr) Buffer	2	128	25344	101376	307200	414720	921600	1310720
Total	4	384	76032	304128	921600	1244160	2764800	3932160

4.3.2 HW/SW shared memories

The HW/SW shared memories are memory buffers shared between the HW and SW components of the encoder. Each individual buffer has to be linear and contiguous, and allocated in a memory area accessible by both the HW and SW. They all have to be 64-bit aligned. If memory-caching mechanisms are in use, the consistency of these memory areas has to be taken care of.

Allocations are done with `EWLMallocLinear()` (See 6.3.7), which is implemented during the integration phase.

TABLE 17. H.264 HW/SW SHARED MEMORY BUFFERS

Name	Amount	Size per MB [byte]	Default Size [byte]	Max Size [byte]
NAL size buffer	1		332	332
Total			332	332

4.3.3 SW/SW shared memories

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` (See 6.3.16 and 6.3.17), which are implemented during the integration phase.

TABLE 18. H.264 SW MEMORY BUFFERS

Memory type	Video Resolution						
	QCIF	QVGA	CIF	VGA	PAL	720p	SXGA
SW/SW memory buffers	2064	2064	2064	2064	2064	2064	2064

4.3.4 Overall memory usage

Table 19 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 19 MEMORY USAGE OF H.264 ENCODER WITH DIFFERENT PICTURE SIZES

Memory type	Video Resolution						
	QCIF [kB]	QVGA [kB]	CIF [kB]	VGA [kB]	PAL [kB]	720p [kB]	SXGA [kB]
HW internal	76	226	304	900	1216	2700	3840
HW/SW	1	1	1	1	1	1	1
SW/SW	2	2	2	2	2	2	2
Input picture[1)]	38	113	152	450	608	1350	1920
Total default	117	342	459	1353	1827	4053	5763

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

4.4 MPEG-4/H.263 Encoder

The memory needs of MPEG-4/H.263 Encoder are described in this chapter.

4.4.1 HW internal buffer

The HW internal buffer stores the internal reference picture used by the HW for motion estimation. This buffer is exactly the same as for the H.264 encoder. Please check chapter 4.3.1 for further details.

4.4.2 HW/SW shared memories

The HW/SW shared memories are memory buffers shared between the HW and SW components of the encoder. Each individual buffer has to be linear and contiguous, and allocated in a memory area accessible by both the HW and SW. They all have to be 64-bit aligned. If memory-caching mechanisms are in use, the consistency of these memory areas has to be taken care of.

Allocations are done with `EWLMallocLinear()` (See 6.3.7), which is implemented during the integration phase.

When video packaged MPEG-4 streams are produced the encoder needs to allocate RLC and control data buffers. The minimum size of the RLC buffer has to fit the maximum amount of data generated for one macroblock, which is 1536 bytes. The total RLC buffer size is configurable by setting the allocated number of bytes per macroblock. The encoder will split equally the bytes per macroblock between all the RLC buffers in use. The hardware will stop when the RLC buffer limit is reached and once a new RLC buffer is provided it can continue the processing. This is the reason why using 2 RLC buffers can improve the overall performance of the system.

By default the encoder is configured to allocate 2 RLC buffers with 256 bytes per macroblock. It is not recommended to set a very low RLC buffer size because that will cause overall performance degradation hence the HW will have to stop every time it reaches the buffer limit.

TABLE 20. MPEG-4 HW/SW SHARED MEMORY BUFFERS

Name	Amount	Size per MB [byte]	Default Size[2)] [byte]	Max Size[2)] [byte]
RLC data	2	1-1536 [1)]	921600	7864320
Macroblock control data	1	24	86400	122880
H.263 GOB sizes buffer[3)]	1		96	96
Total			1008096	7987296

[1)] The RLC buffer amount and buffer size is configurable. Minimum RLC data buffer size is 1536 bytes (maximum RLC data for one macroblock is 1536 bytes). Default 256 bytes.
[2)] SXGA size picture used for the maximum values (5120 macroblocks)
[3)] H.263 supports maximum PAL size picture

4.4.3 SW/SW shared memories

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` (See 6.3.16 and 6.3.17), which are implemented during the integration phase.

TABLE 21. MPEG-4 SW MEMORY BUFFERS

Memory type	Video Resolution						
	QCIF [byte]	QVGA [byte]	CIF [byte]	VGA [byte]	PAL [byte]	720p [byte]	SXGA [byte]
SW/SW memory buffers Data Partitioning enabled	3976	6160	7056	14640	18320	36400	48560
SW/SW memory buffers plain MPEG-4 or H.263	2480	2480	2480	2480	2480	2480	2480

4.4.4 Overall memory usage

Table 22 presents the default and Table 23 the maximal amount of memory allocated by the encoder for different picture resolutions. The stream output buffer is not included in the calculations.

TABLE 22. DEFAULT MEMORY USAGE OF MPEG-4 ENCODER WITH DIFFERENT PICTURE SIZES

Memory type	Video Resolution						
	QCIF [kB]	QVGA [kB]	CIF [kB]	VGA [kB]	PAL [kB]	720p [kB]	SXGA [kB]
HW internal	76	226	304	900	1216	2700	3840
HW/SW common	1	1	1	1	1	1	1
SW/SW common	2.5	2.5	2.5	2.5	2.5	2.5	2.5
Input picture[1)] (YCbCr4:2:0)	38	113	152	450	608	1350	1920
Total	117.5	342.5	459.5	1353.5	1827.5	4053.5	5763.5
HW/SW RLC mode	27	88	108	328	443	984	1400
SW/SW RLC mode	1.5	3.6	4.5	11.9	14.5	33.2	45.0
Total RLC mode[2)]	146	434.1	572	1693.4	2294	5070.7	7213

[1)] Not allocated by the encoder internally
[2)] When video packaged MPEG-4 streams are produced. The default values are using RLC buffers with 256 bytes/macroblock.

TABLE 23. MAXIMAL MEMORY USAGE OF MPEG-4 ENCODER WITH DIFFERENT PICTURE SIZES

Memory type	Video Resolution						
	QCIF [kB]	QVGA [kB]	CIF [kB]	VGA [kB]	PAL [kB]	720p [kB]	SXGA [kB]
HW internal	76	230	304	921	1244	2700	3840
HW/SW	151	488	603	1828	2468	5484	7800
SW/SW	4	6.1	7	14.4	17	35.7	45.5
Input picture[1)] (YCbCr4:2:0)	38	113	152	450	608	1350	1920
Total maximal	269	837.1	1066	3213.4	4337	9569.7	13610

[1)] Not allocated by the encoder internally

4.5 JPEG Encoder

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

4.5.1 SW/SW shared memories

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` (See 6.3.16 and 6.3.17), which are implemented during the integration phase.

TABLE 24. JPEG SW MEMORY BUFFERS

Memory type	Picture Resolution				
	1 Mpixels [byte]	3 Mpixels [byte]	5 Mpixels [byte]	8 Mpixels [byte]	16 Mpixels [byte]
SW/SW memory buffers	1424	1424	1424	1424	1424

4.5.2 Overall memory usage

Table 25 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 25. MEMORY USAGE OF JPEG ENCODER WITH DIFFERENT PICTURE SIZES

Memory type	Picture Resolution				
	1 Mpixels [kB]	1 Mpixels [kB]	5 Mpixels [kB]	8 Mpixels [kB]	16 Mpixels [kB]
SW/SW	1.4	1.4	1.4	1.4	1.4
Input picture[6] (YCbCr4:2:0)	1482	4421.3	7371	11741.6	23981.5
Total default [1]	1483.4	4422.7	7372.4	11743	23982.9

[1] – 1Mpixel (1216x832) size picture = 3952 macroblocks
[2] – 3Mpixel (2096x1440) size picture = 11790 macroblocks
[3] – 5Mpixel (2688x1872) size picture = 19656 macroblocks
[4] – 8Mpixel (3408x2352) size picture = 31311 macroblocks
[5] – 16Mpixel (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.

4.6 Video Stabilization

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

4.6.1 SW/SW shared memories

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` (See 6.3.16 and 6.3.17), which are implemented during the integration phase.

TABLE 26. VIDEO STABILIZATION SW MEMORY BUFFERS

Memory type	Picture Resolution						
	QCIF [bytes]	QVGA [bytes]	CIF [bytes]	VGA [bytes]	PAL [bytes]	720p [bytes]	SXGA [bytes]
SW/SW memory buffers	696	696	696	696	696	696	696

4.6.2 Overall memory usage

Table 27 presents the amount of memory allocated by the Video Stabilization for different picture resolutions.

TABLE 27. MEMORY USAGE OF VIDEO STABILIZER WITH DIFFERENT PICTURE SIZES

Memory type	Picture Resolution						
	QCIF [kB]	QVGA [kB]	CIF [kB]	VGA [kB]	PAL [kB]	720p [kB]	SXGA [kB]
SW/SW	0.68	0.68	0.68	0.68	0.68	0.68	0.68
Input picture[1)] (YCbCr4:2:0)	74.25	240	297	900	1215	2700	3840
Total default	74.93	240.68	297.68	900.68	1215.68	2700.68	3840.68

[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 theoretically the picture buffer could be smaller.

4.7 Code size

In addition to the dynamically allocated memories the encoder software library code size is approximately 56kB when compiled with ARM RealView tools (cpu=ARM926EJ-S), release build with full time optimization. The software read-only data size, which contains mainly VLC tables, is ~16 kB. These sums up to a total code footprint of ~62kB.

The total code size does not include the encoder system wrapper layer which does not have a common implementation for all platforms. Anyway the size of this should not change significantly the total library size.

5 Performance Figures

5.1 H.264 Encoder

The H.264 Encoder software has a minimal processor load. Only things software does are the stream header generation and the picture based rate control. This load is about 4000 CPU cycles for each encoded picture.

5.2 MPEG-4 Encoder

When plain or video packaged MPEG-4 mode or H.263 mode is used, the whole encoding is done by the hardware and the software load is negligible, about 4000 CPU cycles are used for each encoded picture.

Producing data partitioned MPEG-4 stream requires much higher processing power. Table 28 and Table 29 present simulated CPU load figures for several MPEG-4 streams. The video sequence "Shields" was used in all streams. The simulations were done using ARM's RealView tools.

TABLE 28. SIMULATED CPU LOAD OF THE ENCODER FOR TYPICAL MPEG-4 STREAMS (DATA PARTITIONED)

MPEG-4 Simple Profile Level	Video Resolution	Frame rate (fps)	Bit Rate (kbps)	ARM926[1)] CPU load [MHz]	ARM1136[2)] CPU load [MHz]
1	QCIF	15	64	4	4
2			128	5	4
2		30	128	8	7
3			192	9	8
3	QVGA	15	192	11	10
3			256	12	11
3		30	256	20	18
3			384	22	19
3	CIF	15	256	14	13
3			384	16	15
3		30	384	26	23
3			512	28	25
4A	VGA	15	512	39	35
4A			1024	47	42
4A		30	1024	77	68
4A			2048	92	82
5	NTSC	15	1024	50	45
5			1536	58	52
5		30	1536	92	81
5			3072	116	104

[1)] - ARM926 with 266MHz CPU clock, 133MHz bus clock, 0 W/S memory
[2)] - ARM1136 with 399MHz CPU clock, 133MHz bus clock, 0 W/S memory

TABLE 29. SIMULATED CPU LOAD OF THE ENCODER FOR TYPICAL MPEG-4 STREAMS (DATA PARTITIONED)

MPEG-4 Main Profile Level	Video Resolution	Frame rate (fps)	Bit Rate (kbps)	ARM926 CPU load [MHz]	ARM1136 CPU load [MHz]
4	720p	15	2048	120	107
4			3072	135	122
4		30	4096	236	211
4			6144	266	240

Table 30 presents performance figures of the encoder when 8/1 WS memories are in use. The streams were the same as used in the optimal case of 0 WS memories.

TABLE 30. SIMULATED CPU LOAD OF THE ENCODER FOR TYPICAL MPEG-4 STREAMS (DATA PARTITIONED) USING 8/1 WS MEMORIES

MPEG-4 Profile Level	Video Resolution	Frame rate (fps)	Bit Rate (kbps)	ARM926 CPU load [MHz]	ARM1136 CPU load [MHz]
SP[3] 1	QCIF	15	64	4	4
SP 2			128	5	5
SP 2		30	128	8	8
SP 3			192	9	9
SP 3	QVGA	15	192	12	11
SP 3			256	13	12
SP 3		30	256	21	19
SP 3			384	23	21
SP 3	CIF	15	256	15	14
SP 3			384	17	16
SP 3		30	384	28	25
SP 3			512	29	27
SP 4A	VGA	15	512	41	37
SP 4A			1024	49	45
SP 4A		30	1024	81	72
SP 4A			2048	97	88
SP 5	NTSC	15	1024	53	48
SP 5			1536	62	56
SP 5		30	1536	96	87
SP 5			3072	122	110
MP[4] 4	720p	15	2048	126	115
MP 4			3072	142	131
MP 4		30	4096	247	224
MP 4			6144	278	255

[3] Simple Profile
[4] Main Profile

5.3 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.

5.4 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 *7280_encoder/software/source* folder.

TABLE 31. PLATFORM INDEPENDENT SOURCES

Component	Source path:
API headers	*7280_encoder/software/source/inc*
EWL header	*7280_encoder/software/source/inc*
Encoder common sources	*7280_encoder/software/source/common*
MPEG-4/H.263 encoder	*7280_encoder/software/source/mpeg4*
H.264 encoder	*7280_encoder/software/source/h264*
JPEG encoder	*7280_encoder/software/source/jpeg*
Encoder software configuration header	*7280_encoder/software/source/common*
Video Stabilization sources	*7280_encoder/software/source/camstab*
Video Stabilization configuration header	*7280_encoder/software/source/camstab*

The software has been developed and integrated on ARM Versatile Platform (ARM 926) running Linux OS. This porting serves as a practical example for any further integration on different platforms. All porting related code is available in *7280_encoder/software/linux_reference* folder.

The encoder wrapper layer example in the *7280_encoder/software/linux_reference/ewl* subfolder is the most interesting part considering the integration work. It presents a functional implementation of the EWL interface described in chapter 6.3.

All testing related source code is located in *7280_encoder/software/linux_reference/test* folder.

TABLE 32. REFERENCE PORTING AND TESTING SOURCES

Component	Source path:
Encoder library build rules	*7280_encoder/software/linux_reference/Makefile*
Debug and tracing implementation	*7280_encoder/software/linux_reference/debug_trace*
EWL implementations	*7280_encoder/software/linux_reference/ewl*
Kernel module for the encoder	*7280_encoder/software/linux_reference/kernel_driver*
Linear memory kernel module	*7280_encoder/software/linux_reference/memalloc*
MPEG-4/H.263 testbench	*7280_encoder/software/linux_reference/test/mpeg4*
H.264 testbench	*7280_encoder/software/linux_reference/test/h264*
JPEG testbench	*7280_encoder/software/linux_reference/test/jpeg*
Video Stabilization testbench	*7280_encoder/software/linux_reference/test/camstab*

6.2 Behavior

The encoder API user manuals (see [2], [3] and [4]) should be consulted for a detailed description of all the API functions and their usage.

6.2.1 Encoder initializations

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 whole 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.

Figure 5 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 a mandatory thing when multi-instance support is required.

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 H.264, H.263, plain/VP MPEG-4 or JPEG formats. When data partitioned MPEG-4 stream is required extra software processing is needed for the data produced by the hardware (the final stream is created by software by VLC encoding the RLC data).

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

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.

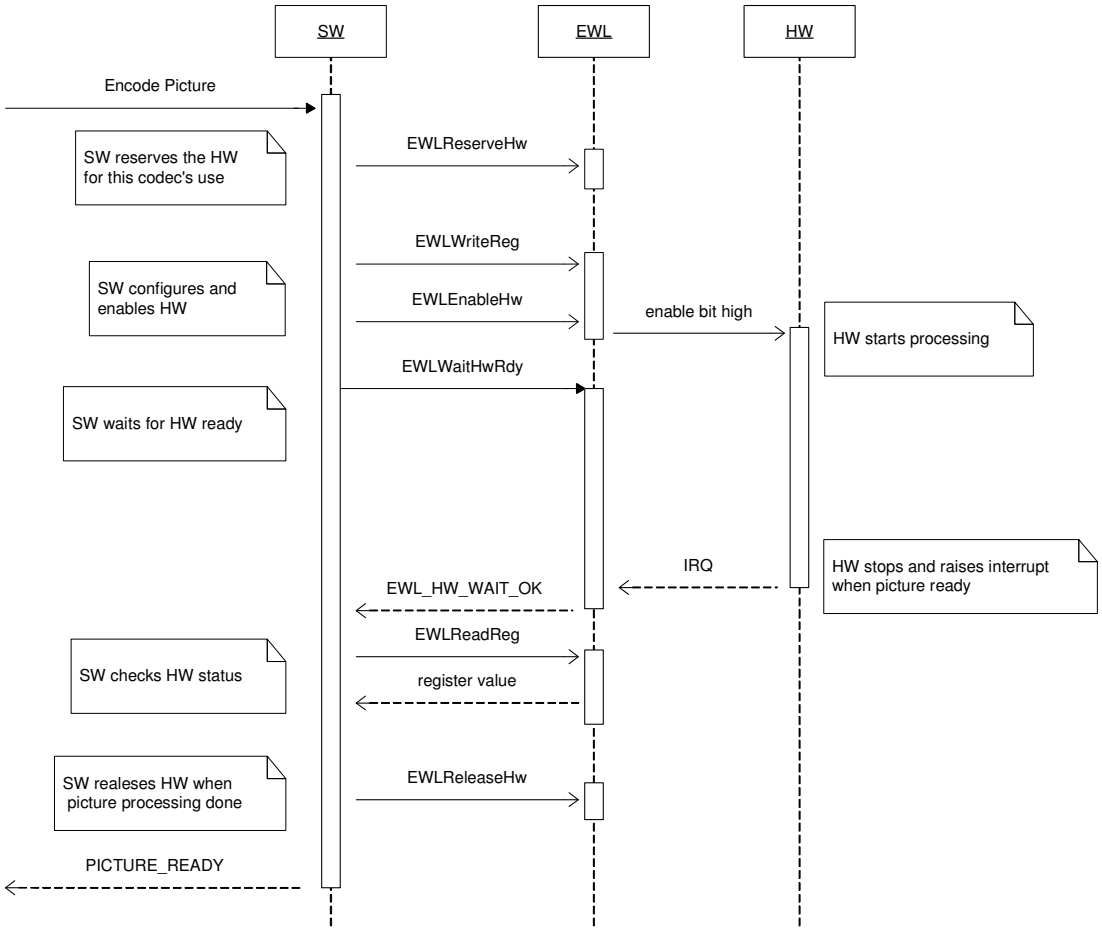

FIGURE 5. PICTURE ENCODING PROCESS

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 has to 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 Figure 6.

The encoder software will not access any HW 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`.

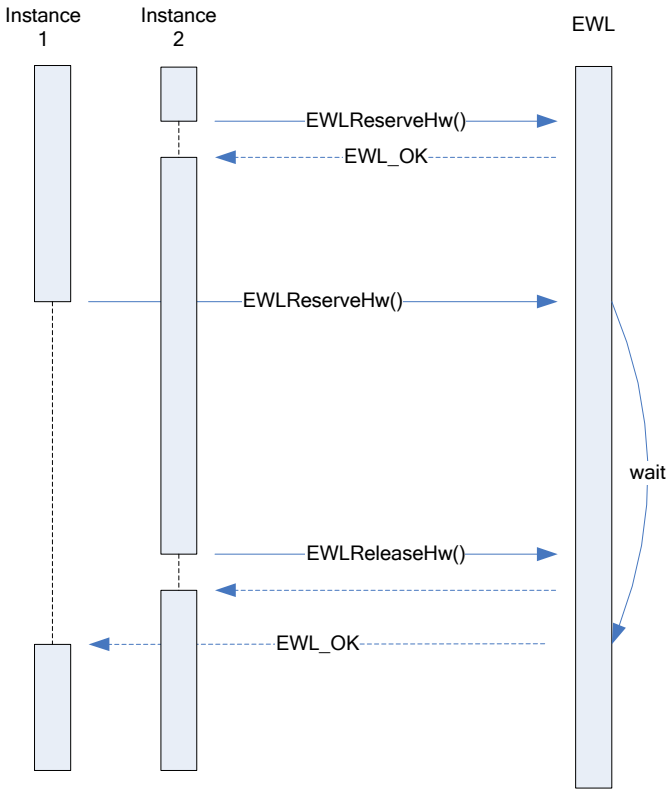

FIGURE 6. EXCLUSIVE ACCESS TO HARDWARE RESOURCES.

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 can't access the HW 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 it. 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. Figure 7 shows how the EWL read and write register functions are used by the encoder.

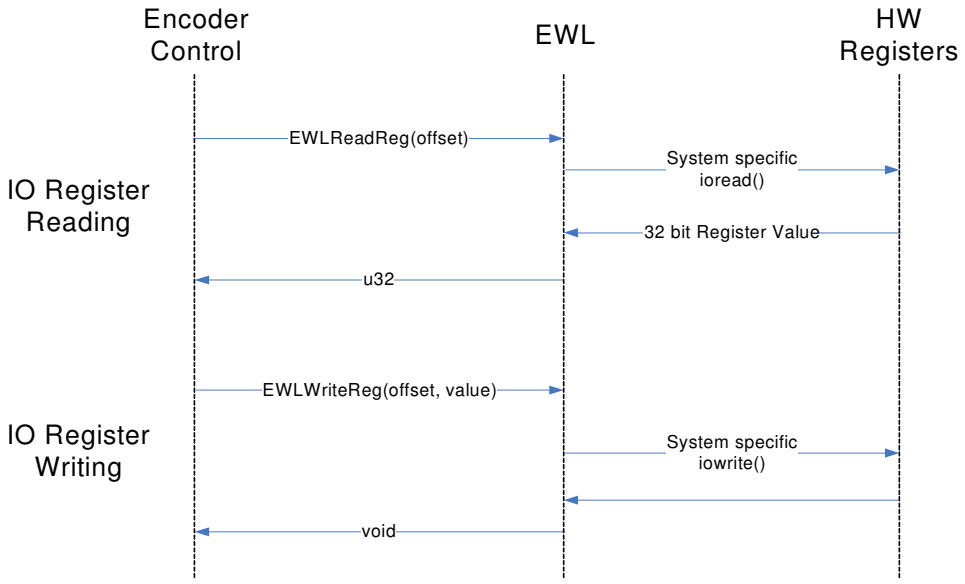

FIGURE 7. HARDWARE REGISTER ACCESS SEQUENCE

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 HW starting point allows a more flexible EWL implementation, in which the register accesses can be buffer. When the accesses are buffered, the cached values shall be written out to the HW when `EWLEnableHw` is called and refreshed, i.e. read back from the registers, always after a HW status change (IRQ).

The encoder can force a HW 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`, `EWLReadReg` and `EWLWriteReg` 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 SW 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 Figure 8. The HW can generate an IRQ when it has finished processing (if the IRQ generation is enabled) and also sets the interrupt status bits in the registers. The IRQ generation is controlled by setting the IRQ disable bit in the HW registers (See [1] for the hardware register description). The whole encoder has to be aware if the IRQ is in use or not. This is set in the *encfg.h* file (See 6.5 Building and configuring the software).

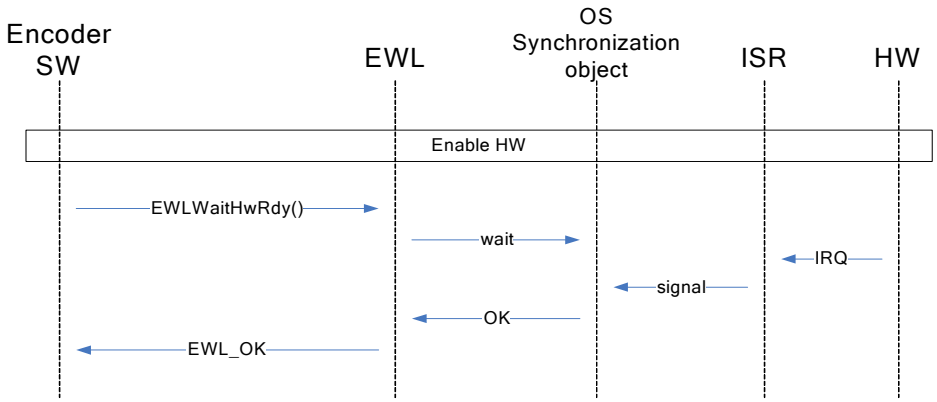

FIGURE 8. ENCODER HW/SW SYNCHRONIZATION. OK CASE

The `EWLWaitHwRdy` function is used so that it allows several kind 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 Figure 8
- Poll for status bits

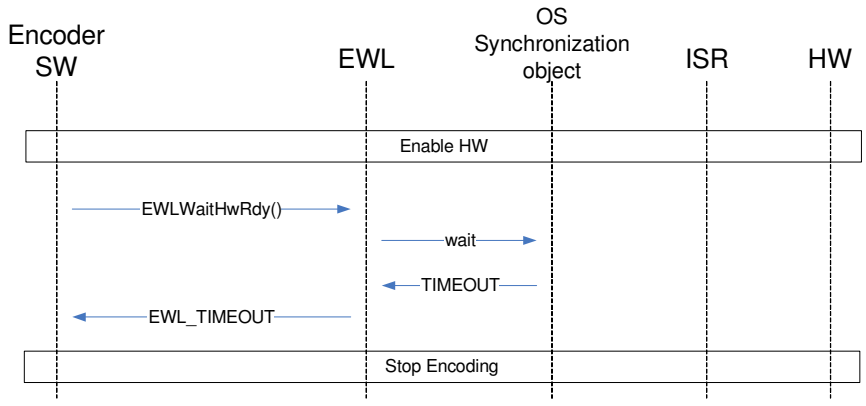

FIGURE 9. ENCODER HW/SW SYNCHRONIZATION. TIMEOUT CASE

If the HW processing takes too long time, the wait for it could timeout (See Figure 9). In this kind of situation the frame is lost; the control code will reset the hardware and return an error value to the application.

Also any error situation has to be returned to the encoder software as described in Figure 10. This error will cause the currently processed data to be lost, the HW to be reset and an error value returned to the application.

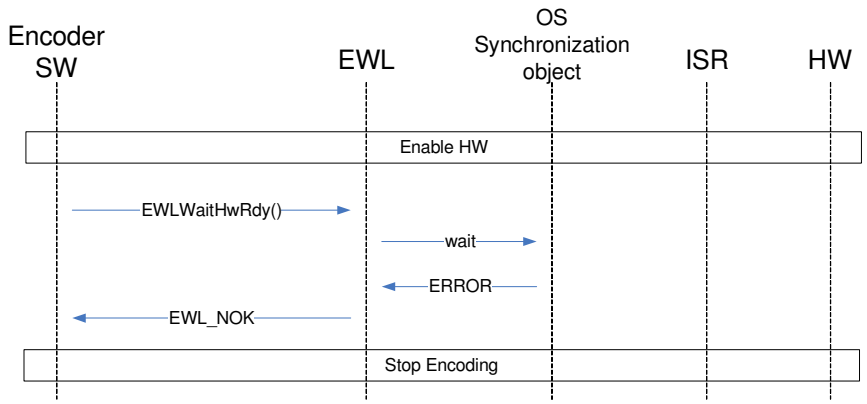

FIGURE 10. ENCODER HW/SW SYNCHRONIZATION. ERROR CASE

Every time the `EWLWaitHwRdy` returns the up-to-date status of the hardware it has to 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 have to be refreshed.

6.2.6 Video stabilization

When the video stabilization is enabled within the encoder preprocessing block, no special consideration has to 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. Notice that the stabilization function uses only the luminance component of the stabilized picture. For YCbCr 4:2:0 input the luminance data is in a separate plane but for YCbCr 4:2:2 the luminance samples are interleaved with the chrominance samples.

The video stabilization can be used in standalone mode, controlled by its own API [5]. 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 6.2.3). The hardware configuration and HW/SW synchronization are done in a same way as at the encoder side (See 0 and 6.2.5).

6.3 Encoder Wrapper Layer 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 `ewl.h` file.

6.3.1 EWLReadAsicID

Syntax

```
u32 EWLReadAsicID(void)
```

Purpose

This function returns the ID of the encoder HW. 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.

Parameters

None.

Return value

Returns the ID number of the encoder HW.

6.3.2 EWLReadAsicConfig

Syntax

```
EWLHwConfig_t EWLReadAsicConfig(void)
```

Purpose

This function returns the static configuration info of the HW. This function shall not require the EWL initialization. The information returned contains the maximum supported picture width, the enabled/disabled status of the different HW components and other HW 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 info is available in one of the HW registers (See from [1]).

Parameters

None.

Return value

```
typedef struct EWLHwConfig
{
    u32 maxEncodedWidth;
    u32 h264Enabled
    u32 jpegEnabled;
    u32 mpeg4Enabled;
    u32 vsEnabled;
    u32 busType;
    u32 busWidth;
    u32 synthesisLanguage;
} EWLHwConfig_t;
```

`maxEncodedWidth`
: Specifies the maximum supported picture width (in pixels) for video encoding.

`h264Enabled`
: Set to 1 if H.264 encoding is supported. Otherwise to 0

`jpegEnabled`
: Set to 1 if JPEG encoding is supported. Otherwise to 0.

`mpeg4Enabled`
: Set to 1 if MPEG-4 encoding is supported. Otherwise to 0.

`vsEnabled`
: Set to 1 if video stabilization is supported. Otherwise to 0.

`busType`
: Purely informational field. Can have following values:

```
EWL_HW_BUS_TYPE_UNKNOWN=0
EWL_HW_BUS_TYPE_AHB=1
EWL_HW_BUS_TYPE_OCP=2
EWL_HW_BUS_TYPE_AXI=3
EWL_HW_BUS_TYPE_PCI=4
```

`busWidth`
: Purely informational field. Can have following values:

```
EWL_HW_BUS_WIDTH_UNKNOWN=0
EWL_HW_BUS_WIDTH_32BITS=1
EWL_HW_BUS_WIDTH_64BITS=2
EWL_HW_BUS_WIDTH_128BITS=3
```

`synthesisLanguage`
: Purely informational field. Can have following values:

```
EWL_HW_SYNTHESIS_LANGUAGE_UNKNOWN=0
EWL_HW_SYNTHESIS_LANGUAGE_VHDL=1
EWL_HW_SYNTHESIS_LANGUAGE_VERILOG=2
```

6.3.3 EWLInit

Syntax

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

Purpose

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.

Parameters

`EWLInitParam_t *` `param` - initialization parameters

```
typedef struct EWLInitParam
{
    u32 clientType;
} EWLInitParam_t;
```

`u32` `clientType` – the type of the client, which is trying to create the new EWL instance. This helps in doing a selective initialization depending on the type of the client.
Possible values:

`EWL_CLIENT_TYPE_H264_ENC`	– a 7280 H.264 encoder is the client
`EWL_CLIENT_TYPE_MPEG4_ENC`	– a 7280 MPEG-4/H263 encoder is the client
`EWL_CLIENT_TYPE_JPEG_ENC`	– a 7280 JPEG encoder is the client
`EWL_CLIENT_TYPE_VIDEOSTAB`	– a 7280 standalone video stabilization is the client

Return value

Returns a pointer to the newly created EWL instance or NULL in case of a failure.

6.3.4 EWLRelease

Syntax

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

Purpose

This function is called to release an instance of the system wrapper. This function releases the entire wrapper resources. After the release no call to the EWL (EWLReadAsicID is the exception) can 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.

Parameters

`const void` `*instance` – the EWL instance to be released

Return value

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

6.3.5 EWLMallocRefFrm

Syntax

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

Purpose

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 bigger if any system specific alignment has to be satisfied (ex. 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 need consideration.

Parameters

`const void` `*instance` – the EWL instance that will do the allocation

`EWLLinearMem_t` `*info` – address of a structure containing parameters of the memory area.

```
typedef struct EWLLinearMem
{
    u32 *virtualAddress;
    u32 busAddress;
    u32 size;
} EWLLinearMem_t;
```

`virtualAddress`
: at return this will contain the address of the allocated memory area. This is used for software accesses only.

`busAddress`
: at return this will contain the DMA address of the allocated memory area. This is used for hardware accesses.

`size`
: when called this contains the requested memory area size in bytes. At return it has to reflect the exact size of the allocated memory area.

Return value

Returns zero for a successful allocation or a negative error code for a failure.

6.3.6 EWLFreeRefFrm

Syntax

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

Purpose

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 which allocated it.

Parameters

`const void` `*instance` – the EWL instance that frees the memory block.
`EWLLinearMem_t` `*info` – a structure containing parameters of the memory area to be freed. This should be exactly the same information returned by the `EWLMallocRefFrm`.

Return value

None.

6.3.7 EWLMallocLinear

Syntax

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

Purpose

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 bigger if any system specific alignment has to be satisfied (ex. 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, OCP bus is 64 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.

Parameters

`const void` `*instance` – the EWL instance that will do the allocation
`EWLLinearMem_t` `*info` – address of a structure containing parameters of the memory area. See `EWLMallocRefFrm` for a detailed description of the structure.

Return value

Returns zero for a successful allocation or a negative error code for a failure.

6.3.8 EWLFreeLinear

Syntax

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

Purpose

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.

Parameters

`const void` `*instance` – the EWL instance that free the memory block.
`EWLLinearMem_t` `*info` – a structure containing parameters of the memory area to be freed. This should be exactly the same information returned by the `EWLMallocLinear`.

Return value

None.

6.3.9 EWLReadReg

Syntax

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

Purpose

This function is called to read a 32-bit value from a specified HW register. The instance in use can identify the client which is attempting the read access. Accesses to registers shall not be done just after an exclusive right is obtained with the successful call of `EWLReserveHw`. If such an access should occur it signals a flaw in the client design. Consult the 7280 Hardware Integration Guide [1] for a detailed description of the registers.

Parameters

`const void` `*instance` – the EWL instance that will do the reading.
`u32` `offset` – offset of the register to be read.

Return value

The data contained in the register.

6.3.10 EWLWriteReg

Syntax

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

Purpose

This function is called to write a 32-bit value to a specified HW register. The instance in use can identify the client which is attempting the write access. Accesses to registers shall not be done just after an exclusive right is obtained with the successful call of `EWLReserveHw`. If such an access occurs it signals a flaw in the client design. Consult the 7280 Hardware Integration Guide [1] for a detailed description of the registers.

Parameters

`const void` `*instance` – the EWL instance that will do the reading.
`u32` `offset` – offset of the register to be written.
`u32` `value` – data to be written to the specified register.

Return value

None.

6.3.11 EWLEnableHW

Syntax

```
void EWLEnableHw(const void *instance, u32 offset, u32 value)
```

Purpose

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 not be done just after an exclusive right is obtained with the successful call of `EWLReserveHw`. If such an access occurs it signals a flaw in the client design. Consult the 7280 Hardware Integration Guide [1] for a detailed description of the registers.

Parameters

`const void` `*instance` – the EWL instance that will do the writing.
`u32` `offset` – offset of the register to be written.
`u32` `value` – data to be written to the specified register.

Return value

None.

6.3.12 EWLDisableHW

Syntax

```
void EWLDisableHw(const void *instance, u32 offset, u32 value)
```

Purpose

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 not be done just after an exclusive right is obtained with the successful call of `EWLReserveHw`. If such an access occurs it signals a flaw in the client design. Consult the 7280 Hardware Integration Guide [1] for a detailed description of the registers.

Parameters

`const void` `*instance` – the EWL instance that will do the writing.
`u32` `offset` – offset of the register to be written.
`u32` `value` – data to be written to the specified register.

Return value

None.

6.3.13 EWLWaitHwRdy

Syntax

```
i32 EWLWaitHwRdy (const void *instance)
```

Purpose

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.

Parameters

`const void` `*instance` – the EWL instance in use.

Return value

`EWL_HW_WAIT_OK` – Hardware has stopped and everything is OK.
`EWL_HW_WAIT_TIMEOUT` – Timeout, hardware is still running and the wait for it has timed out.
`EWL_HW_WAIT_ERROR` – An error occurred during the wait process. This is returned for any error encountered during the wait.

6.3.14 EWLReserveHw

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

Purpose

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.

Parameters

`const void` `*instance` – the EWL instance in use.

Return value

Returns 0 for a successful reservation or a negative error code.

6.3.15 EWLReleaseHw

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

Purpose

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 another client can acquire the exclusive hardware access.

Parameters

`const void` `*instance` – the EWL instance in use.

Return value

None.

6.3.16 EWLmalloc

Syntax

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

Purpose

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

Parameters

`u32` `n` – size in bytes of the memory to be allocated

Return value

A valid pointer to the allocated memory or NULL for a failure.

6.3.17 EWLcalloc

Syntax

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

Purpose

This function allocates an array in memory with elements initialized to 0. It has the same functionality as the ANSI C `calloc`.

Parameters

`u32` `n` – number of elements to allocate

`u32` `s` – size in bytes of each element

Return value

A valid pointer to the allocated memory or NULL for a failure.

6.3.18 EWLfree

Syntax

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

Purpose

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

Parameters

`void` *p – previously allocated memory block to be freed.

Return value

None.

6.3.19 EWLmemcpy

Syntax

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

Purpose

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

Parameters

void *d – new buffer to copy to

const void *s – source buffer to copy from

u32 n – number of bytes to copy

Return value

The destination pointer *d*.

6.3.20 EWLmemset

Syntax

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

Purpose

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`.

Parameters

void *d – pointer to destination
i32 c – character to set
u32 n – number of characters to set in destination

Return value

The destination pointer *d*.

6.3.21 EWLWriteRegAll

This function is reserved for future extensions and is currently not in use.

6.3.22 EWLReadRegAll

This function is reserved for future extensions and is currently not in use.

6.3.23 EWLDCacheRangeFlush

This function is reserved for future extensions and is currently not in use.

6.3.24 EWLDCacheRangeRefresh

This function is reserved for future extensions and is currently not in use.

6.4 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 might 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 reentrant 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 reentrance 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.

6.4.1 EWL initialization and release

The `EWLInit` routine 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 HW related buffers
- memory for SW needs
- HW register access
- HW/SW synchronization
- HW 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 HW polling or IRQ. The polling is done by reading the HW 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 has to be compiled separately and loaded into the kernel. Common parts are implemented in *'ewl_x280_common.c'*

For SW development purposes two different kind of EWL implementation exists: *'ewl_x280_file.c'* and *'ewl_7280_system.c'*. The purpose is to allow development and SW CPU load simulations without the HW. Instead of using the HW encoder, these EWLs utilize the bit-exact system model of the hardware.

EWL system model implementation replaces the whole HW with the system model allowing control SW development and testing in any PC environment.

EWL file implementation uses the system model to create hardware output files and the HW output data is read from the files in the EWL. However, the system model and the EWL are not connected so the same encoding parameters have to be used for both the system model when creating the data and for the encoder SW when using the file EWL. This is easily achieved by using the testing scripts which are described in the next chapter. The implementation also calculates the amount of memories allocated and prints them out when tracing is enabled.

6.4.2 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.

6.4.3 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.

6.4.4 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 write and read a register having a specific offset.

6.4.5 Hardware sharing

To be able to run multiple encoder instances at the same time there has to 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.

6.5 Building and configuring the software

The whole 7280 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 (See the [1] for detailed description). You can override the default configuration by defining the flags for compiler in the '*Makefile'*.

6.5.1 Common encoder configuration

Common encoder settings are described in Table 33. These affect the functionality of all encoders included in the 7280 product.

TABLE 33 COMMON ENCODER CONFIGURATION PARAMETERS

Macro	Description	Values
ENC7280_AXI_WRITE_ID	Identification value for hardware write accesses when connected to an AXI master bus interface.	[0,255]
ENC7280_AXI_READ_ID	Identification value for hardware read accesses when connected to an AXI master bus interface.	[0,255]
ENC7280_INPUT_ENDIAN	Hardware input picture data endianess. In most of the cases this matches the native system endianess but it can be changed to match a video capture device output format.	0 – BIG endian 1 – LITTLE endian
ENC7280_OUTPUT_ENDIAN	Hardware output data endianess. This must be the same as the native system endianess.	0 – BIG endian 1 – LITTLE endian
ENC7280_INPUT_ENDIAN_WIDTH	Defines how the endianess is applied on HW's 64 bit wide memory accesses. This affects the reading of input	0 - for 32-bit endianess 1 - for 64-bit endianess

	pictures. For example: LITTLE for 32 bit will give byte order 32107654 LITTLE for 64 bit will give byte order 76543210	
ENC7280_OUTPUT_ENDIAN_WIDTH	Same as above but for output stream writing.	0 - for 32-bit endianess 1 - for 64-bit endianess
ENC7280_BURST_LENGTH	Maximum burst length for hardware bus transactions. Value 0 in AHB means INCR type bursts will be generated. For OCP and AXI a 0 value is forbidden.	[0,4,8,16] for AHB. [1,31] for OCP, [1,16] for AXI
ENC7280_BURST_INCR_TYPE_ENABLED	INCR type burst mode control	0 - enable INCR type bursts 1 - disable INCR type and use SINGLE instead
ENC7280_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
ENC7280_ASIC_CLOCK_GATING_ENABLED	HW internal clock gating control.	0 - disabled 1 - enabled
ENC7280_IRQ_DISABLE	Disables the HW interrupts, SW must poll the status register in order to get the HW status.	0 – IRQ enabled 1 – IRQ disabled

6.5.2 MPEG-4 specific configuration

MPEG-4 specific encoder settings are described in Table 34. These settings are used just when encoding video packaged MPEG-4 streams. Double buffering can give a faster processing by running both HW and SW in parallel (SW will start processing the data finished by hardware in a previous run and HW can continue its own processing). All other encoding modes process one picture at a time (JPEG can process also slices of a bigger picture).

TABLE 34 MPEG-4 ENCODER CONFIGURATION PARAMETERS

Macro	Description	Values
ENC7280_BUFFER_AMOUNT	Hardware output RLC buffer amount. Double buffering is the default and is	2

	recommended.	
ENC7280_BUFFER_SIZE_MB	Hardware output RLC buffer size per macroblock. The amount specified here is equally split between all RLC buffers in use.	256
ENC7280_IRQ_FREQUENCY_FRAME_START	Defines the first HW interrupt interval after starting a frame encoding. This is specified in macroblocks.	0 – generate IRQ when RLC buffer full.
ENC7280_IRQ_FREQUENCY	Defines the HW interrupt interval in use after the first IRQ was received. This is specified in macroblocks.	0 – generate IRQ when RLC buffer full or full picture encoded.

*Note! Always compile the MPEG4 software with **MPEG4_HW_RLC_MODE_ENABLED** defined. If not defined, the parts of the control code which are not needed in full hardware VLC mode will not be enabled. This will cause "undefined reference" errors during compilation. Undefined this only when any other encoder software is compiled without including the MPEG-4 related software.*

*Note! Always compile the MPEG4 software with **MPEG4_HW_VLC_MODE_ENABLED** defined. This will assure full HW acceleration when generating plain MPEG4 streams. Otherwise HW will generate just RLC data and software has to do the costly VLC encoding.*

6.5.3 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 35 STANDALONE VIDEO STABILIZATION CONFIGURATION PARAMETERS

Macro	Description	Values
VS7280_AXI_WRITE_ID	See ENC7280_AXI_WRITE_ID	[0,255]
VS7280_AXI_READ_ID	See ENC7280_AXI_READ_ID	[0,255]
VS7280_INPUT_ENDIAN	See ENC7280_INPUT_ENDIAN	[0,1]
VS7280_INPUT_ENDIAN_WIDTH	See ENC7280_INPUT_ENDIAN_WIDTH	[0,1]
VS7280_BURST_LENGTH	See ENC7280_BURST_LENGTH	[1,4,8,16] for AHB. [1,31] for OCP. [1,16] for AXI.
VS7280_BURST_INCR_TYPE_ENABLED	See ENC7280_BURST_INCR_TYPE_ENABLED	[0,1]

VS7280_BURST_DATA_DISCARD_ENABLED	See ENC7280_BURST_DATA_DISCARD_ENABLED	[0,1]
VS7280_ASIC_CLOCK_GATING_ENABLED	See ENC7280_ASIC_CLOCK_GATING_ENABLED	[0,1]
VS7280_IRQ_DISABLE	See ENC7280_IRQ_DISABLE	[0,1]

6.5.4 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_ASIC – writes traces from the HW interface into a file
- TRACE_REGS – writes traces from the HW registers into a file
- TRACE_RLC – writes traces from the RLC word parsing into a file
- TRACE_BUFFERING – writes traces from the RLC buffering into a file
- TRACE_MB – writes traces of each macroblock data into a file
- TRACE_STREAM – writes traces from output stream writing into a file. Requires the tracing functions define in *'enctracestream.h'* and uses the macros COMMENT and TRACE_BIT_STREAM defined in *'encdebug.h'.*
- TRACE_TIMING – adds timing information into the buffering trace file

The implementation of the trace functions is provided in the *7280_encoder/software/linux_reference/debug_trace* folder.

Define H7280_HAVE_ENCDEBUG_H if *'encdebug.h'* file is available.
Define H7280_HAVE_ENCTRACE_H if *'enctrace.h'* file is available.

6.5.5 API tracing

The API entries and exits can be traced by defining H264ENC_TRACE, MP4ENC_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.

6.6 Recommendations for memory allocation/optimization

Because the memory busload during the encoding can get really 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 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.

```
Usage:  h264_testenc [options] -i inputfile

  -i[s] --input             Read input from file. [input.yuv]
  -o[s] --output            Write output to file. [stream.h264]
  -a[n] --firstVop          First vop of input file. [0]
  -b[n] --lastVop           Last vop of input file. [100]
  -w[n] --lumWidthSrc       Width of source image. [176]
  -h[n] --lumHeightSrc      Height of source image. [144]
  -x[n] --width             Width of output image. [--lumWidthSrc]
  -y[n] --height            Height of output image. [--lumHeightSrc]
  -X[n] --horOffsetSrc      Output image horizontal offset. [0]
  -Y[n] --verOffsetSrc      Output image vertical offset. [0]
  -f[n] --outputRateNumer   1..65535 Output vop rate numerator. [30]
  -F[n] --outputRateDenom   1..65535 Output vop rate denominator. [1]
  -j[n] --inputRateNumer    1..65535 Input vop rate numerator. [30]
  -J[n] --inputRateDenom    1..65535 Input vop rate denominator. [1]

  -L[n] --level             10..32, H264 Level. [32]

  -R[n] --byteStream        Stream type. [1]
                             1 - byte stream according to Annex B.
                             0 - NAL units. Nal sizes returned in
                                 <nal_sizes.txt>

  -S[n] --sei               Enable/Disable SEI messages. [0]

  -r[n] --rotation          Rotate input image. [0]
                                0 - disabled
                                1 - 90 degrees right
                                2 - 90 degrees left
  -l[n] --inputFormat       Input YUV format. [0]
                                0 - YUV420
                                1 - YUV420 semiplanar
                                2 - YUYV422
                                3 - UYVY422
  -k[n] --videoRange        0..1 Video range. [0]
```

```
  -Z[n] --videostab         Video stabilization. n > 0 enabled [0]
  -T[n] --constIntraPred    0=OFF, 1=ON Constrained intra pred flag [0]
  -D[n] --disableDeblocking 0..2 Value of disable_deblocking_filter_idc [0]
  -I[n] --intraVopRate      Intra vop rate. [0]
  -V[n] --mbPerSlice        Slice size in macroblocks. Should be a
                            multiple of MBs per row. [0]

  -B[n] --bitPerSecond      Bitrate. [64000]
  -U[n] --vopRc             0=OFF, 1=ON Vop rc (Source model rc). [1]
  -u[n] --mbRc              0=OFF, 1=ON Mb rc (Check point rc). [1]
  -C[n] --hrdConformance    0=OFF, 1=ON HRD conformance. [0]
  -s[n] --vopSkip           0=OFF, 1=ON Vop skip rate control. [0]
  -q[n] --qpHdr             -1..51, Default frame header qp. [26]
                             -1=Encoder calculates initial QP
  -n[n] --qpMin             0..51, Minimum frame header qp. [10]
  -m[n] --qpMax             0..51, Maximum frame header qp. [51]

Testing parameters that are not supported for end-user:
  -Q[n] --chromaQpOffset    -12..12 Chroma QP offset. [0]
  -W[n] --filterOffsetA     -6..6 Deblocking filter offset A. [0]
  -E[n] --filterOffsetB     -6..6 Deblocking filter offset B. [0]
  -N[n] --burstSize          0..31 HW bus burst size. [16]
  -t[n] --burstType          0=SIGLE, 1=INCR HW bus burst type. [0]
  -e[n] --testId            Internal test ID. [0]
  -P[n] --trigger           Logic Analyzer trigger at picture <n>. [-1]
```

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: *7280_encoder/software/linux_reference/test/h264*.

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

7.2 Running H.264 tests

Shell scripts are provided for running all or any individual test case. Because of the many parameters of the encoder test bench, it may need some effort to create automated scripts for running all the test cases on other platforms.

The usage of shell scripts is described below:

test_data_parameter_h264.sh - script that contains all the parameters for every test case. This comes with the reference test data.

test_h264.sh <case-number> – runs the test case numbered <case-number>
test_h264.sh all – runs all the test cases
The script runs the cases and creates encoder logs in *results_h264.log* file. This script should be edited in order to configure the location of the input and reference test data and the name of the encoder testbench.

- Set **YUV_SEQUENCE_HOME** variable to contain the path to the input YUV sequences base directory, each resolution should be located in individual subfolders.

- Set **test_data_home** variable to contain the path of the output reference test data base directory, individual test cases should be located here in individual subfolders.
- Set **test_case_list_dir** variable to contain the path of the *test_data_parameter_h264.sh* script.

checkcase_h264.sh <case-number> - checks the result of individual test cases
Within the script set **test_data_home** variable to contain the path of the output reference test data base directory, individual test cases should be located here in individual subfolders.

checkall_h264.sh - checks the result of all test cases. This uses the script for checking one specific test case.
Within the script set **test_case_list_dir** variable to contain the path of the *test_data_parameter_h264.sh* script.

7.3 MPEG-4 testing

The MPEG-4/H263 testing is done similar way as for the H.264 described in the previous chapters. The test scripts are similar but the testbench parameter list is adapted to the MPEG-4 specifics.

```
Usage:  ./mpeg4_testenc [options] [-i <inputfile>] [-o <outputfile>]
  -H    --help             Display this help.

  -S[n] --scheme           0=MPEG4, 1=SVH, 3=H263. [0]

  -i[s] --input            Read input from file. [input.yuv]
  -o[s] --output           Write output to file. [stream.mpeg4]
  -a[n] --firstVop         First vop of input file. [0]
  -b[n] --lastVop          Last vop of input file. [100]
  -w[n] --lumWidthSrc      Width of source image. [176]
  -h[n] --lumHeightSrc     Height of source image. [144]
  -x[n] --width            Width of output image. [--lumWidthSrc]
  -y[n] --height           Height of output image. [--lumHeightSrc]
  -X[n] --horOffsetSrc     Output image horizontal offset. [0]
  -Y[n] --verOffsetSrc     Output image vertical offset. [0]
  -j[n] --inputRateNumer   Input vop rate numerator. [30]
  -J[n] --inputRateDenom   Input vop rate denominator. [1]
  -f[n] --outputRateNumer  Output vop rate numerator. [--inputRateNumer]
  -F[n] --outputRateDenom  Output vop rate denominator. [--inputRateDenom]

  -p[n] --profile          Profile and Level code. [5]
                               1=Simple Profile/Level 1,
                               2=Simple Profile/Level 2,
                               3=Simple Profile/Level 3,
                               4=Simple Profile/Level 4A,
                               5=Simple Profile/Level 5,
                               6=Simple Profile/Level 6,
                               8=Simple Profile/Level 0,
                               9=Simple Profile/Level 0B,
                             243=Advanced Simple Profile/Level 3,
                             244=Advanced Simple Profile/Level 4,
                             245=Advanced Simple Profile/Level 5.
```

```
  -k[n] --videoRange       Source video range.[0]
                                 n=0 [16,235]
                                 n=1 [0,255]
  -I[n] --intraVopRate     Intra vop rate. [0]
  -W[n] --goVopRate        Group of vop (GOVOP) header rate. [0]
  -V[n] --vpSize           Video packet size, bits. [0]
  -D[n] --dataPart         0=OFF, 1=ON Data partition. [0]
  -R[n] --rvlc             0=OFF, 1=ON Reversible vlc. [0]
  -E[n] --hec              0=OFF, 1=ON Header extension code. [0]
  -G[n] --gobPlace          Groups of blocks. Bit pattern define GOB place.
[0]

  -B[n] --bitPerSecond     Bitrate. [Profile/Level maximum]
  -U[n] --vopRc            0=OFF, 1=ON Vop qp (Source model rc). [1]
  -u[n] --mbRc             0=OFF, 1=ON Mb qp (Check point rc). [1]
  -v[n] --videoBufferSize  0=OFF, 1=ON Video buffer verifier. [1]
  -s[n] --vopSkip          0=OFF, 1=ON Vop skip rate control. [0]
  -q[n] --qpHdr            1..31, Default VOP header qp. [10]
                             -1=Encoder calculates initial QP
  -n[n] --qpMin            1..31, Minimum VOP header qp. [1]
  -m[n] --qpMax            1..31, Maximum VOP header qp. [31]

  -z[n] --userDataVos      User data file name of Vos.
  -c[n] --userDataVisObj   User data file name of VisObj.
  -d[n] --userDataVol      User data file name of Vol.
  -g[n] --userDataGov      User data file name of Gov.

  -r[n] --rotation         Source image rotation.[0]
                               n=0, no rotation.
                               n=1, 90 (clockwise rotation).
                               n=2, -90 (counter-clockwise rotation). [0]

  -l[n] --inputFormat      Source image YUV format.[0]
                               n=0, planar YCbCr 4:2:0.
                               n=1, semiplanar YCbCr 4:2:0.
                               n=2, YCbYCr 4:2:2.
                               n=3, CbYCrY 4:2:2.

  -Z[n] --videostab        Video stabilization. n > 0 enabled [0]

 The following parameters are not supported by the product API.
 They are controlled with software algorithms and are provided
 here only for internal testing purposes.
  -N[n] --burstSize         0..31 HW bus burst size. [16]
  -e[n] --testId            Internal test ID. [0]
  -P[n] --trigger           Logic Analyzer trigger at picture <n>. [-1]
```

7.4 Building JPEG 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.

```
Usage:  enc [options] -i inputfile

-H    --help              Display this help.
-W[n] --write             Write output. [0,1]
-i[s] --input             Read input from file. [input.yuv]
-I[s] --inputThumb        Read thumbnail input from file.
                          [inputThumbnail.yuv]
-o[s] --output            Write output to file. [stream.jpg]
-a[n] --firstVop          First vop of input file. [0]
-b[n] --lastVop           Last vop of input file. [0]
-w[n] --lumWidthSrc       Width of source image. [176]
-h[n] --lumHeightSrc      Height of source image. [144]
-x[n] --width             Width of output image. [--lumWidthSrc]
-y[n] --height            Height of output image. [--lumHeightSrc]
-X[n] --horOffsetSrc      Output image horizontal offset. [0]
-Y[n] --verOffsetSrc      Output image vertical offset. [0]
-R[n] --restartInterval   Restart interval in MCU rows. [0]
-q[n] --qLevel            0..9, quantization scale. [1]
-g[n] --frameType         Input YUV format. [0]
                               0 - YUV420
                               1 - YUV420 semiplanar
                               2 - YUYV422
                               3 - UYVY422
-G[n] --rotation          Rotation. [0]
                               0 - disabled
                               1 - 90 degrees right
                               2 - 90 degrees left
-p[n] --codingType        0=whole frame, 1=partial frame encoding. [0]
-t[n] --markerType        Quantization/Huffman table markers. [0]
                               0 = single marker
                               1 = multi marker
-u[n] --unitsType         Units type to APP0 header. [0]
                               0 = pixel aspect ratio
                               1 = dots/inch
                               2 = dots/cm
-k[n] --xdensity          Xdensity to APP0 header. [1]
-l[n] --unitsType         YDensity to APP0 header. [1]
-T[n] --thumbnail         0=NO, 1=YES Thumbnail to stream. [0]
-B[n] --lumWidthSrcThumb  Width of thumbnail source image. [176]
-e[n] --lumHeightSrcThumb Height of thumbnail source image. [144]
-K[n] --widthThumb        Width of thumbnail output image.
                              [--lumWidthSrcThumb]
-L[n] --heightThumb       Height of thumbnail output image.
                              [--lumHeightSrcThumb]
-A[n] --horOffsetSrcThumb Thumbnail output image horizontal offset. [0]
-O[n] --verOffsetSrcThumb Thumbnail output image vertical offset. [0]
```

```
-c[n] --comLength          Comment header length. [0]
-C[s] --pCom               Comment header data file. [com.txt]
```

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 for JPEG codec is located in the corresponding subfolder under *7280_encoder/software/linux_reference/test/jpeg.* The provided Makefile should be edited to match the host and target environment settings.

7.5 Running JPEG test bench

Shell scripts are provided for running all or any individual test case. Because of the many parameters of the encoder test bench, it may need some effort to create automated scripts for running all the test cases on other platforms.

The usage of shell scripts is described below:

parameter.sh - script that contains all the parameters for every test case

test_jpeg.sh <case-number> – runs the test case numbered <case-number>
test_jpeg.sh all – runs all the test cases
The script runs the cases and creates encoder logs in *results_jpeg.log* file.
The script should be edited in order to configure the location of the input and reference test data and the name of the encoder testbench.

- Set **YUV_SEQUENCE_HOME** variable to contain the path to the input YUV sequences base directory, each resolution should be located in individual subfolders.
- Set **test_data_home** variable to contain the path of the output reference test data base directory, individual test cases should be located here in individual subfolders.
- Set **test_dir** variable to contain the path of the *parameter.sh* script.

checkcase_jpeg.sh <*case-number*> - checks the result of individual test cases
Within the script set **test_data_home** variable to contain the path of the output reference test data base directory, individual test cases should be located here in individual subfolders.

checkall_jpeg.sh - checks the result of all test cases. This uses the script for checking one specific test case.
Within the script set **test_dir** variable to contain the path of the *parameter.sh* script.

7.6 Video stabilization testing

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 stabilization API to process pictures, and writes the results to a file. The testbench parameters allow controlling the video stabilization and testbench functionality. The descriptions for the parameters can be obtained by executing the testbench without parameters.

```
Usage:  camstabtest [options] –i inputfile

  –i[s] --input             Read input from file. [input.yuv]
  –a[n] --firstVop          First vop of input file. [0]
  –b[n] --lastVop           Last vop of input file. [100]
  –w[n] --lumWidthSrc       Width of source image. [176]
  –h[n] --lumHeightSrc      Height of source image. [144]
  –W[n] --width             Width of output image. [--lumWidthSrc]
  –H[n] --height            Height of output image. [--lumHeightSrc]
```

```
  -l[n] --inputFormat       Input YUV format. [0]
                                0 - YUV420
                                1 - YUV420 semiplanar
                                2 - YUYV422
                                3 - UYVY422

  -T    --traceresult       Write output to file <video_stab_result.trc>

Testing parameters that are not supported for end-user:
  -N[n] --burstSize          0..63 HW bus burst size. [16]
  -P[n] --trigger           Logic Analyzer trigger at picture <n>. [-1]
```

The testbench source code is located in:
7280_encoder/software/linux_reference/test/h264.

Edit the provided *Makefile* to match the host and target environment settings.

Shell scripts are provided for running all or any individual test case.
Shell scripts and their usage:

test_data_parameter_h264.sh – script that contains all the parameters for every test case. The pipelined encoder-stabilization cases will be run in standalone mode.

test_vs.sh <*case-number*> – runs the test case numbered <*case-number*>
test_vs.sh all – runs all the test cases
The script runs the cases and creates stabilization logs in *results_vs.log*.
Give `-T` parameter to the testbench to have results written to a file otherwise these will be only printed to `stdout`.
Edit the script in order to configure the location of the input and reference test data and the name of the testbench.

- Set **YUV_SEQUENCE_HOME** variable to contain the path to the input YUV sequences base directory, each resolution should be located in individual subfolders.
- Set **test_data_home** variable to contain the path of the output reference test data base directory, individual test cases should be located here in individual subfolders.
- Set **test_case_list_dir** variable to contain the path of the *test_data_parameter_h264.sh* script.

checkcase_vs.sh <*case-number*> - checks the result of individual test cases
checkall_vs.sh <*case-number*> - checks the result of all test cases. This uses the script for checking one specific test case.

- Set **test_data_home** variable to contain the path of the output reference test data base directory, individual test cases should be located here in individual subfolders.
- Set **test_case_list_dir** variable to contain the path of the *test_data_parameter_h264.sh* script.

7.7 Test data

The provided test data consists of raw input YUV sequences and output reference stream files. 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 video stabilization reference data consists in traces of the stabilization results (offsets).

References

[1] Hantro Products Oy (2008). 7280 Hardware Integration Guide
[2] Hantro Products Oy (2008). 7280 H.264 Encoder API User Manual
[3] Hantro Products Oy (2008). 7280 MPEG-4/H.263 Encoder API User Manual
[4] Hantro Products Oy (2008). 7280 JPEG Encoder API User Manual
[5] Hantro Products Oy (2008). 7280 Video Stabilization API User Manual
[6] OCP International Partnership (2003). Open Core Protocol Specification Release 2.0.
[7] ARM (1999) AMBA Specification Revision 2.0. URL: http://www.arm.com/products/solutions/AMBA_Spec.html
[8] ARM (2004) AMBA 3 AXI Protocol v1.0 Specification .URL: http://www.arm.com/products/solutions/AMBA3AXI.html
[9] ISO/IEC 14496-10 / ITU-T Recommendation H.264 (2003). Advanced video coding for generic audiovisual services.
[10] ITU-T Recommendation T.81 (1991). Information technology – Digital Compression and Coding of Continuous-tone Still Images – Requirements and Guidelines
[11] ITU-T Recommendation H.263 (1998). Video Coding for Low Bit Rate Communication.
[12] ISO/IEC 14496-2 (2001). Information technology – Generic Coding for Low Bit Rate Communication.

VISIBLY BETTER

Headquarters:
Hantro Products Oy
Kiviharjunlenkki 1, FI-90220 Oulu, Finland
Tel: +358 207 425100, Fax: +358 207 425299

Hantro Finland
Lars Sonckin kaari 14
FI-02600
Espoo
Finland
Tel: +358 207 425100
Fax: +358 207 425298

Hantro Germany
Ismaninger Str. 17-19
81675
Munich
Germany
Tel: +49 89 4130 0645
Fax: +49 89 4130 0663

Hantro USA
1762 Technology Drive
Suite 202. San Jose
CA 95110,
USA
Tel: +1 408 451 9170
Fax: +1 408 451 9672

Hantro Japan
Yurakucho Building 11F
1-10-1, Yurakucho
Chiyoda-ku, Tokyo
100-0006, Japan
Tel: +81 3 5219 3638
Fax: +81 3 5219 3639

Hantro Taiwan
6F, No. 331
Fu-Hsing N. Rd.
Taipei
Taiwan
Tel: +886 2 2717 2092
Fax: +886 2 2717 2097

Hantro Korea
#518 ho, Dongburoot, 16-2
Sunaedong, Bundanggu,
Seongnamsi, Kyunggido,
463-825 Korea
Tel: +82 31 718 2506
Fax: +82 31 718 2505

Email: sales@hantro.com
WWW.HANTRO.COM