Application Programming Interface 1.1

7280 H.264 Encoder

User Manual Version 1.2

Copyright Information

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

Glossary

720p	High Definition resolution of 1280x720 progressive video
API	Application Programming Interface
bps	Bits per second
CIF	Common Interchange Format (352x288 pixels)
CPB	Coded Picture Buffer
EOS	End of Sequence
fps	Frames per second
H.264	Video Coding Standard (ITU-T)
HD	High Definition
HRD	Hypothetical Reference Decoder
IDR	Instantaneous Decoder Refresh
MB	Macroblock (16x16 pixel)
mbps	Mega (1000000) bits per second
NAL	Network Abstraction Layer
NALU	NAL unit, the smallest decodable unit of a stream
PAL	Phase Alternate Line (resolution 720x576 pixels)
PPS	Picture Parameter Set
QCIF	Quarter CIF (176x144 pixels)
QVGA	Quarter Video Graphics Array (320x240 pixels)
QP	Quantization parameter
QQVGA	Quarter QVGA (160x120 pixels)
RC	Rate Control
SEI	Supplemental Enhancement Information
SPS	Sequence Parameter Set
sub-QCIF	Another video resolution (128x96 pixels)
SXGA	Super eXtended Graphics Array (resolution 1280x1024 pixels)
VGA	Video Graphics Array (resolution 640x480 pixels)
VLC	Variable Length Coding
VUI	Video Usability Information

Table of Contents

1 Introduction

This document presents the Application Programming Interface (API) of the Hantro 7280 H.264 hardware based encoder. The encoder is able to encode H.264 standard [1] baseline profile video streams. Because no conflicting tools are in use, the streams are also main profile compatible.

The encoder conforms to the H.264 Baseline profile and can encode streams up to a maximum picture size of 1280x1024 (SXGA) and bit rate of 20 Mbps. The maximum frame rate at the maximum picture size is 30 fps. Real performance is very much system dependent.

The usage of the API is described in chapter 3. Chapter 4 gives guidelines to video frame storage format. The detailed function interfaces and data types are introduced in chapter 5. Chapter 6 gives application examples. 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.

This document assumes that the reader understands the fundamentals of C-language and the H.264 standard.

2 API Version History

Table 1 describes the released API versions, any changes introduced.

TABLE 1 API VERSION HISTORY

API version	Changes/Comments
1.0	Original version
1.1	Added possibility to force encoding “not coded” frames

3 Usage of the Encoder

The block diagram of a typical encoding process is depicted in Figure 1. The figure shows the main steps of the encoder usage: the initialization, the configuration (optional), the stream producing with its own sub-steps (start, frame encoding and end) and finally the release of the encoder.

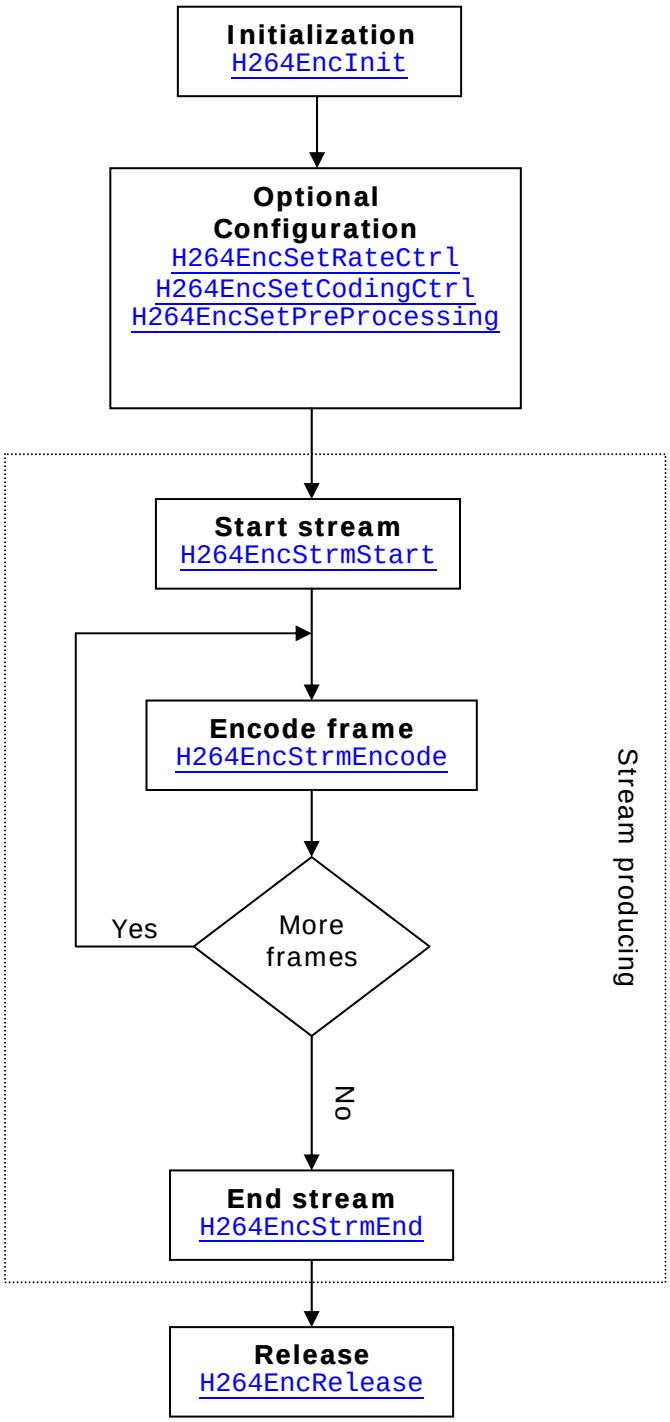

FIGURE 1 ENCODING PROCESS BLOCK DIAGRAM

3.1 Encoder limitations

Depending on the application needs the 7280 encoder hardware can be scaled down to support only lower resolutions (720p, D1, VGA, CIF, etc.). The encoder software will always check what the maximum supported resolution is and fail to initialize if too big picture size is used.
The video stabilization block can be left out also from the design, in which case the pre-processor will fail to initialize with the stabilization turned on.

NOTE! Check the exact specifications of the encoder version in use.

3.2 Initialization and release of the encoder

In order to be able to use the encoder, it has to be properly initialized first. The initialization is done by calling `H264EncInit`. The call will allocate all the resources needed by the encoder and will execute all the necessary setups in order to have a fully operational encoder. If successful, the initialization call will return a new instance of the encoder, which will be used as an identifier for all the subsequent encoder operations.

The hardware has support for multiple encoder instances. This means that it is possible to initialize two different encoders and use them for encoding two different streams at the same time. The only limitation is that the hardware can only encode one frame at a time.

The initialization call will return `H264ENC_OK` for a successful initialization. The error code returned in case of a failure will give hints about what went wrong during the initialization process.

The encoder needs a certain number of parameters when initializing. These parameters are the most significant ones, parameters that can't be changed during the whole encoding process.

Stream profile and level indication

This value is defined by H.264 standard and it sets restrictions to some of the encoder's parameters. For a more friendly approach, the supported profile and level codes are enumerated. By specifying a certain profile and level, the user sets restrictions to many of the encoder parameters. Table 2 represents the typical encoding values used with each profile and level. Table 3 represents the most important restrictions set by each profile and level.

TABLE 2 H.264 PROFILES AND THEIR TYPICAL USAGE

Profile&Level	Encoded Picture Size	Frame Rate [fps]	Bit Rate [kbps]
BASELINE_LEVEL_1	Sub-QCIF	15	64
	QCIF	15	64
BASELINE_LEVEL_1b	Sub-QCIF	15	128
	QCIF	15	128

BASELINE_LEVEL_1_1	QCIF	30	192
	QVGA	10	192
BASELINE_LEVEL_1_2	QVGA	20	384
	CIF	15	384
BASELINE_LEVEL_1_3	QVGA	30	768
	CIF	30	768
BASELINE_LEVEL_2	CIF	30	1000
BASELINE_LEVEL_2_1	512x384	25	2000
BASELINE_LEVEL_2_2	VGA	15	2000
	720x480	15	2000
BASELINE_LEVEL_3	VGA	30	4000
	720x576	25	4000
	720x480	30	4000
BASELINE_LEVEL_3_1	1280x720	30	8000
BASELINE_LEVEL_3_2	1280x1024	30	8000

TABLE 3 H.264 PROFILES AND THEIR MAXIMAL VALUES

Profile&Level	Max Encoded Picture Size [MB]	Max Macroblock Rate [MB/s]	Max Bit Rate [1200 bps]	Max CPB size [1200 bits]
BASELINE_LEVEL_1	99 (QCIF)	1485	64	175
BASELINE_LEVEL_1b	99 (QCIF)	1485	128	350
BASELINE_LEVEL_1_1	396 (CIF)	3000	192	500
BASELINE_LEVEL_1_2	396 (CIF)	6000	384	1000
BASELINE_LEVEL_1_3	396 (CIF)	11880	768	2000
BASELINE_LEVEL_2	396 (CIF)	11880	2000	2000
BASELINE_LEVEL_2_1	792	19800	4000	4000
BASELINE_LEVEL_2_2	1620 (PAL)	20250	4000	4000
BASELINE_LEVEL_3	1620 (PAL)	40500	10000	10000
BASELINE_LEVEL_3_1	3600 (HD 720p)	108000	14000	14000
BASELINE_LEVEL_3_2	5120 (SXGA)	216000	20000	20000

The standards define the limitations in a more complex way, so for more in detail information, please consult the relevant papers [1].

The profile and level selection cannot be changed after the initialization.

Stream type

H.264 standard defines two stream formats: NAL unit stream format and byte stream format. NAL unit stream format consists of plain NAL units. The byte stream format separates each NAL unit with zero bytes and a start code prefix which makes it easier to separate the NAL units from each other. When using NAL unit stream format the size of each NAL unit is not known by the stream itself so it has to be communicated some other way. The encoder is able to produce either of these stream formats depending on the application's needs.

When “byte stream” mode is selected (H264_BYTE_STREAM) the data produced by the encoder is ready to be stored or delivered as it is. This should be de default format when storing to file based on a standard format (3GPP, MP4).

If the application does require pure NAL units (H264_NAL_UNIT_STREAM) the data produced by the encoder does not contain the 4-byte start code at the beginning of each NAL unit. Without this the individual NAL units cannot be separated by simply parsing the stream. To allow the separation the encoder returns the size of each generated NAL unit in a separate user allocated buffer. This mode is most probably used in streaming applications.

Encoded picture size

The width and height of the encoded picture in pixels has to be specified when initializing an encoder instance. Notice the difference between the input picture size as captured by a camera and the final encoded image size (See Video pre-processor usage). The input image size can be different than the encoded one if before the encoding process the input image is cropped. The main limitations of the encoded picture’s size are set by the selected profile and level. See tables 1 and 2 for more details. Also some implementation specific limitations are in force: the encoded picture width has to be multiple of 4, the height a multiple of 2 and the smallest size is 96x96. The maximum encoded picture size is 1280x1024 (or 1024x1280 when rotated) pixels. The size cannot be altered after the initialization. Even though the encoded picture width can be a 4 multiple, the horizontal scanline has to be a 16 multiple, meaning that the memory offset form the start of a pixel row to the beginning of the next pixel row is always a 16 multiple. This assumption can be seen in the initial value of the pre-processor parameters, where the input source image width is the rounded up 16 multiple of the encoded with. There is no such limitation on the height of the picture. Figure 2 shows the described limitation regarding the input picture horizontal scanline.

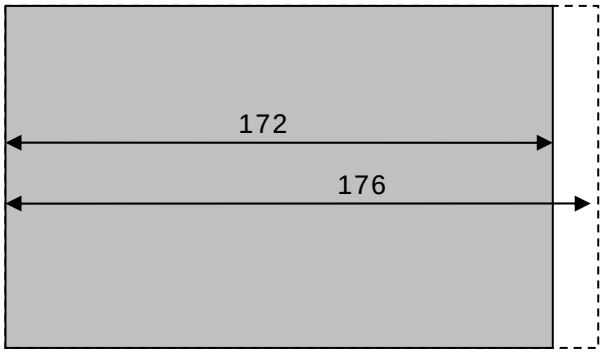

FIGURE 2. ENCODED PICTURE SIZE AND ASSUMED VIRTUAL INPUT PICTURE SIZE

Frame rate descriptor

In order to be able to efficiently control the bit rate the encoder needs a target frame rate. The frame rate is specified with a frame rate numerator and a frame rate denominator. The division of the frame rate numerator and the frame rate denominator defines the number of equal subintervals, called ticks, within a second. The ticks are used as a base unit for time increment of each individual frame. More details about the frame timing will be given in the Stream producing chapter.

NOTE! Keep in mind that this is just a target frame rate. The time increments of the encoded video frames will determine the real frame rate. In other words, even if a 30 fps frame rate (i.e. frame rate nominator 30 and denominator 1) is targeted but the time between two consecutive frames is always 2 ticks, the final stream will have a 15 fps frame rate.

Default encoder initial values

- The output bit rate is set to the typical bit rate for the selected profile and level (See Table 1 and 2).
- Picture and macroblock based rate control are enabled.
- Picture skipping is disabled.
- HRD is enabled.
- Pre-processing is disabled.
- Slices are disabled.

Encoder release

At the end of the encoding process the encoder in use has to be released. Use `H264EncRelease` to perform a safe release of all the resources allocated when the encoder was initialized. If a release fails most probably the encoder instance was corrupted and there is no safe way to assure that all the encoder's resources were freed.

3.3 Configuration of the encoder

There are several encoding parameters that can be updated after the encoder has been initialized. The configuration API calls take as parameter an encoder instance and the address of a specific data structure that contains the new values. It is possible to read the current values in use by the encoder providing an easy way of changing just some of the values from a bunch of parameters grouped together in a structure. Some of the encoder parameters can be changed only before a stream is started and others can be altered at any time between two frame encoding.

Bit rate control configuration

The rate control (RC) is configured initially to produce the maximum bit rate allowed within the selected stream profile and level. The bit rate and some other rate control parameters can be updated after the initialization of the encoder using `H264EncSetRateCtrl`. The current parameters can be retrieved using `H264EncGetRateCtrl`. The RC controls the output bit rate by changing the quantization parameter (QP) or in extreme cases by skipping entire frames from being encoded.

The rate control parameters values can be changed at any time during the encoding sequence. However, the rate control is reset whenever setting new values. A description of the different rate control settings is given below.

- **Picture based rate control** – This is a frame based rate control algorithm, which can be turned ON or OFF. If ON the RC can adjust the QP between frames.
- **MB based rate control** – This is a macroblock based rate control, which can be turned ON or OFF. If ON the RC can adjust the QP inside a frame.
- **Picture skipping** – When the output rate cannot be adjusted just via the QP changes picture skipping can help control it; it can be turned ON or OFF. Note that if HRD is enabled it may skip frames even if picture skipping is disabled.

- **Default QP** – This is the default or initial QP in use by the encoder. This will be used for the first encoded picture when the rate control is turned ON or for every encoded picture if the rate control is turned OFF. A value lower then 10 should not be used even if the standard allows it.
- **Min QP** – This is the minimum value QP that can be used by the encoder and is relevant just when the RC is turned ON. A value lower than 10 should not be used even if the standard allows it.
- **Max QP** – This is the maximum QP which can be set by the encoder; also in use just if the RC is turned ON
- **Output bit rate** – This is the target bit rate in bits per second for the output stream and it is used when one or more of the following is turned ON: picture based RC, MB based RC, picture skipping or HRD. It is initially set to the maximum value defined by the profile and level of the stream but it can be set this to higher value if needed.
- **HRD –** This is an algorithm for checking a bit stream with its bit rate, to verify that the amount of rate buffer memory required in a decoder is less than the standard-defined buffer size. This can be turned ON or OFF, but by turning it OFF the output stream might not be 100% standard compatible.
- **HRD CPB size** – This is the size of the CPB used in the HRD model. By default it is set to the maximum size allowed for the initialized encoder level. Do not change if not sure about it.
- **GOP length** – GOP is a group of successive pictures within a video stream. GOP usually contains an Intra frame (I-frame) and several Inter frames (Predicted frame). GOP length tells the distance between two Intra frames. Intra frames make video more easily seek-able and editable, but they use more bits which decreases the overall quality of the stream. Rate control uses the GOP length to match the average bit rate of each GOP to the target bit rate. Recommended setting for rate control GOP length is the distance between two Intra frames.

Usually it will be enough to just set the target bit rate and this can be handled by calling `H264EncGetRateCtrl` to get original values, changing the target bit rate and setting the values by `H264EncSetRateCtrl`.

To achieve accurate control of the output bit rate the macroblock based rate control is enabled in initialization. If QP shall not change in the middle of a picture this should be disabled.

In certain cases when target bit rate is low enough the encoder may not be able to reach the target no matter what QP is used. Therefore it may be useful to enable picture skipping for really low bit rates. If the output frame rate is not required to be the same as the input frame rate this parameter may be enabled to give the rate control algorithm more freedom to allocate bits for pictures.

Setting minimum and/or maximum QP can be used to restrict the rate control to only use QP values in the range [Min QP, Max QP]. Usually this is not needed.

In case the frame rate numerator and denominator set in the initialization do not correspond to average frame rate, the estimated time increment shall be set so that rate control can reasonably allocate bits for the first picture of the sequence. If, for example, the frame rate numerator was 1000 and denominator was 1, and average frame rate would be 25fps, the estimated time increment could be set to 40.

The HRD feature of the encoder is enabled by default. It can be disabled, but then the encoder will not be able to guarantee that restrictions placed on the standard [1] will

always be obeyed. When the HRD is enabled the encoder runs a model of the decoder input stream buffer (coded picture buffer) to make sure that at decoding time of certain picture the buffer contains enough data so that the decoder may decode the picture. On the other hand, the HRD model assures that the buffer can accommodate the stream data at any point of time. The model assumes that there is a constant bit rate channel between the encoder and the decoder. The size of the coded picture buffer is specified in the standard ([1]) for each level. When the HRD is disabled the operation of the rate control is alleviated so that the encoder does not have to keep the channel occupied all the time and the encoder may produce images whose size exceeds the buffer occupancy at the decoding time of the picture.

NOTE: When HRD is enabled, the selected bitrate and any custom CPB size have to be within the limits set by the initialized encoder level (See values in Table 3*). Once the stream has been started the rate control parameters (bitrate) cannot be altered anymore. This is due to the fact that the stream start headers contain information about the selected bitrate and CPB size.*

VBR and CBR video

Variable Bit Rate (**VBR**) is best choice for locally stored video. More bits are allocated for complex sections and less bits for simple sections. VBR has better quality versus space ratio than CBR video. In Hantro encoder VBR is constrained so that rate control tries to reach set average bit rate over the time. Refer to Table 3 and H.264 standard [1] for maximum bit rate limits.

Constant Bit Rate (**CBR**) is useful for streaming video over constant bandwidth channel. CBR is not good choice for storage, since it doesn't allocate enough bits for complex sections and wastes bits on simple sections. In a real time encoder a way to achieve CBR is to change QP of macroblocks inside the frame during encoding. This might have the adverse effect of lower part of the frame becoming blurry.

Using Hantro encoder in **VBR** mode:

```
pictureRc = 1
mbRc = 0
hrd = 0
```

Using Hantro encoder in **CBR** mode:

```
pictureRc = 1
mbRc = 1
hrd = 1
```

Coding control settings

There are several coding control parameters that can be altered using `H264EncSetCodingCtrl`. Current settings can be retrieved using `H264EncGetCodingCtrl`. The recommended way of changing any of the coding control parameters is first to retrieve the current settings and then change the desired ones. All the parameters with the exception of the "slice size" can be set only before starting the encoding.

Overview of the available options

- **Slice size** - Each encoded picture can be divided into several slices. Slices help in the error recovery of the erroneous streams and their role is to split the picture

data into smaller independent packages. The size is specified in macroblock rows. This value can be altered at any time during the encoding process.

- **SEI messages** – When enabled a SEI message containing picture timing information will be inserted into the stream before every encoded frame. Buffering period information will be inserted into the stream before every encoded IDR frame (when HRD is enabled). If not using these SEI messages the stream doesn't contain any timing information so it has to be communicated some other way.
- **Video full range** – Defines the input YCbCr data sample range that will be included in the stream headers. If not set correctly the video range dynamic might suffer degradation on the decoder side.
- **Constrained intra prediction** – Increases the stream's error recovery chances by constraining the intra prediction to use only intra macroblocks.
- **Disable de-blocking filter** – Disables the de-blocking filter thus lowering the encoding and decoding complexity but also lowering the video quality.
- **Sample aspect ratio width and height** – Defines the aspect ratio of the input picture samples that will be included in the stream headers. By default it is square.

Example:

```
H264EncInst encoder;
H264EncCodingCtrl codingCfg;

/* encoder is already initialized and we have an instance */

if(H264EncGetCodingCtrl(encoder, &codingCfg) != H264ENC_OK)
{
      /* handle error */
}
else
{
      codingCfg.sliceSize = 5; /* 5 macroblock rows in each slice */

      if(H264EncSetCodingCtrl(encoder, &codingCfg) != H264ENC_OK)
      {
            /* handle error */
      }
}
```

After the stream was started, from the above parameters only the slice size can be altered.

3.4 Stream producing

After a successful initialization and an optional configuration, the encoder is ready to produce a H.264 compliant stream. There are three separate phases for producing a stream:

1. **Starting the stream** by calling H264EncStrmStart
2. **Encoding video frames** by calling H264EncStrmEncode
3. **Ending the stream** by calling H264EncStrmEnd

Starting the stream

A stream has to be started by a call to H264EncStrmStart. SPS and PPS NAL units are produced at this time. The user has to provide an output buffer where the stream header data will be written. The output buffer is specified with a pointer to the buffer and the size of the buffer in bytes. If successful the call will return H264ENC_OK and the size of the generated stream in bytes. If NAL units are needed then the buffer for sizes has to be specified when starting the stream.

Encoding video frames

A video sequence can be encoded after a stream has been started. A video sequence is encoded frame by frame by calling H264EncStrmEncode. The main inputs for this function are an input buffer, which contains the video frame to be encoded, and an output buffer where the generated stream will be written.

The supported formats of the input video are planar YCbCr 4:2:0, semiplanar YCbCr 4:2:0 and interleaved YCbCr 4:2:2. The format in use has been selected during the initialization. Independently of what input format is used the memory area where the buffers are located has to be 64-bit aligned, linear and hardware accessible. With other words the addresses, which are passed as parameters to the H264EncStrmEncode function, are bus

addresses and will be given to the hardware unmodified and they will be used by the hardware to read the input video frame.

The output buffer is used by the software to construct the final stream. The pointer to the buffer and the size of the buffer will be passed to the encode function. The size of the buffer has to be correlated with the target bit rate and frame rate, for instance, when the target bit rate is 3 Mbps and the frame rate is 30 fps the average frame size is 100 000 bits. Keep in mind that this is an average value and single encoded frames can take many times more space. The maximum size of the encoded frame depends on the input picture. If the encoder reaches the end of the output buffer, it will discard the current frame and return H264ENC_OUTPUT_BUFFER_OVERFLOW. This is even more likely when the rate control is disabled. Even if the current frame is lost the encoding process can continue with a new frame, which will be INTRA coded just to assure that all the possible errors are not perpetrated.
One other important parameter for every video frame is its time stamp. This has to be specified at every call of H264EncStrmEncode. The time stamp is relative to the previously encoded video frame and it is specified as the time increment in ticks (time units). The duration of a tick is specified at the initialization according to the target frame rate. The first encoded frame normally has a time increment zero. For instance, if the time resolution was set to 30 ticks and the frame rate is 30 fps than the relative time between each frame is 1 tick. When the frame has been successfully encoded the H264EncStrmEncode returns H264ENC_FRAME_READY.

Each encoded frame can be divided into several slices. Slices help in the error recovery of the erroneous streams and their role is to split the frame data into smaller independent packages. The size is specified in macroblock rows.

The individual NAL unit sizes can be known also. The user has to provide a buffer where the sizes of all NAL units are returned. The encoder will fill this buffer with the size of each unit in bytes. After the last unit a zero value is written to the size buffer. This information is a prerequisite for separating each NAL unit from the byte stream in case of H264_NAL_UNIT_STREAM. A table of 332 bytes (83*sizeof(u32)) is enough to fit sizes for the theoretical maximum number of slices that can be generated by the encoder.

Ending the stream

A stream will be ended by a call to H264EncStrmEnd. If successful the call will return H264ENC_OK. Stream end consist in a NAL unit of type ‘End-of-Sequence’.

3.5 Video pre-processor usage

The encoder includes the following pre-processing blocks:

1. YCbYCr 4:2:2 to YCbCr 4:2:0 conversion
2. Picture rotation
3. Picture cropping
4. Video stabilization

By default the encoder input is considered to be in planar YUV 4:2:0 format. Cropping, rotation, and stabilization are disabled. The pre-processor is controlled by the H264EncSetPreProcessing and H264EncGetPreProcessing API functions. It is recommended to always retrieve first the current values from the encoder and then do the necessary alteration(s).

3.5.1 Color conversion

The conversion block reads the input picture in interleaved YCbCr 4:2:2 format. The picture is converted into planar YCbCr 4:2:0 format by sub-sampling the chrominance samples vertically and rearranging the samples. The conversion is configured at the initialization phase and it will affect every encoded picture.

3.5.2 Picture rotation

The rotation block rotates the picture 90 degrees clockwise or counter-clockwise. The rotation is configured at the initialization phase and it will affect every encoded picture. Figure 3 illustrates the picture dimensions when rotating the input.

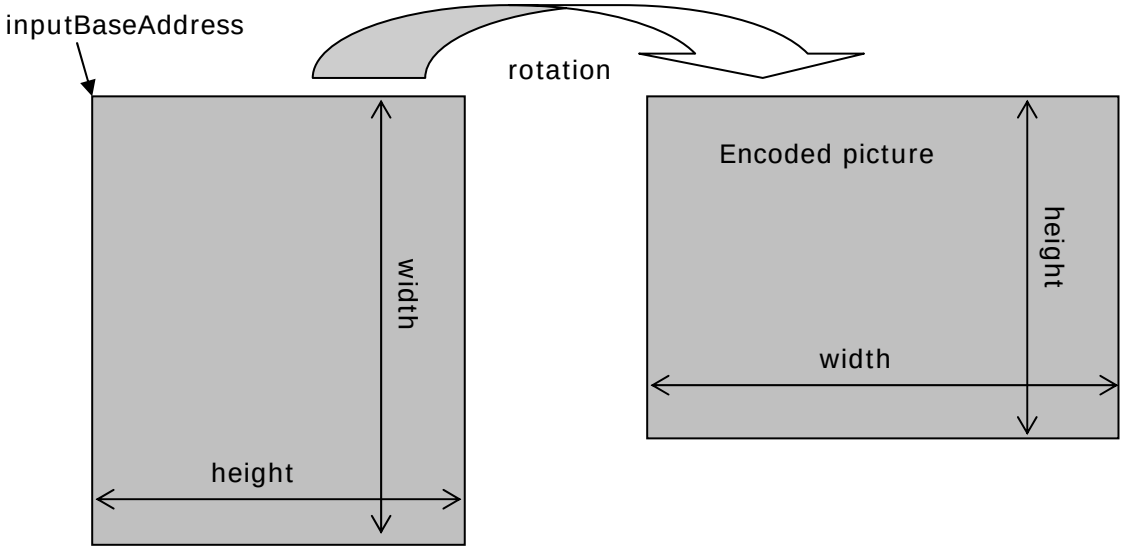

FIGURE 3 PICTURE DIMENSIONS WITH ROTATION

3.5.3 Picture cropping

The picture cropping provides a way to read and encode a smaller selected part of a bigger picture. The cropping is controlled by setting the size of the input source picture and the top-left corner coordinates of the encoded picture relative to the top-left corner of the input picture. Maximum size of the input source image can be 1920x1920 pixels.

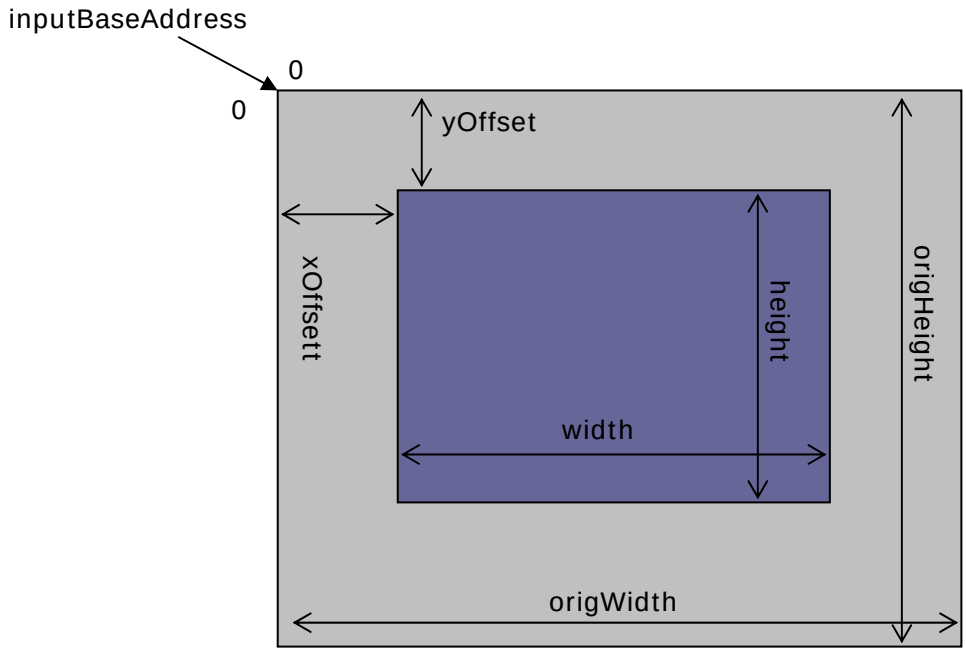

FIGURE 4 RELATIONS BETWEEN THE FULL INPUT IMAGE'S AND THE ENCODED IMAGE'S DIFFERENT SIZES

Figure 4 illustrates the relations between picture parameters. The grey picture is the input picture as captured by a camera and the blue picture is the one encoded by the encoder. Below are the conditions that have to be satisfied in order to enable the cropping block:

$$origWidth - xOffset - width \geq 0$$
$$origHeight - yOffset - height \geq 0$$

The input source picture horizontal stride (in pixels) has to take always 16 multiple values. This means that when for example, the input width is 170 pixels the horizontal stride has to be 176.

Offset of pixel *n* in the picture buffer:

pixel_offset = (n/width)((width+15) & (~15)) + (n%width))*

3.5.4 Video stabilization

The encoder can perform a video stabilization function, which will compensate for any unwanted movement introduced in the video sequence by a shaky capturing device. In order to enable the stabilization the input picture has to be bigger than the encoded picture by at least 8 pixels (both dimensions). The stabilization function will determine the best place from where to read the encoded picture within the boundaries of the input picture. The stabilization uses the cropping function of the encoder and because of this no other application specified cropping can be performed.
In this mode the encoder will always process 2 pictures at a time. One will be encoded and the other will be stabilized. The currently stabilized picture will be encoded next time and another picture will be stabilized for further encoding.

4 Video Frame Storage Format

The input picture format of the encoder is planar YCbCr 4:2:0, semiplanar YCbCr 4:2:0 or interleaved YCbCr 4:2:2. Figure 5 describes how the planar YCbCr 4:2:0 picture is stored in external memory. The luminance and both chrominance components are stored in three buffers and they must be located in a linear and physically contiguous memory block.

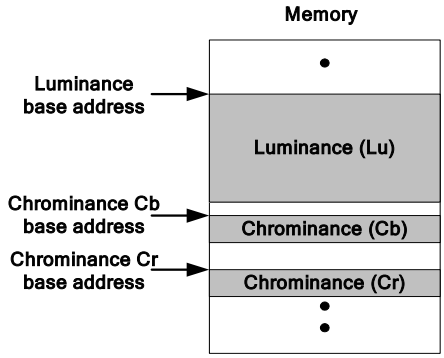

FIGURE 5. PLANAR YCBCR 4:2:0 PICTURE STORAGE IN EXTERNAL MEMORY

Figure 6 describes the format of semiplanar YCbCr 4:2:0 picture. The luminance is equal to planar format but the Cb and Cr chrominance components are interleaved together.

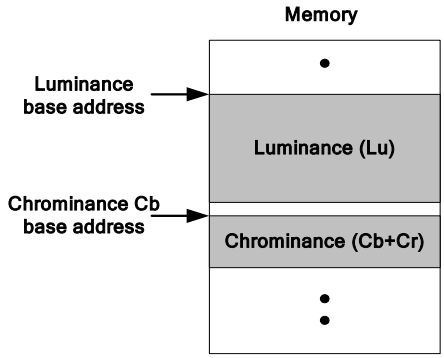

FIGURE 6. SEMIPLANAR YCBCR 4:2:0 PICTURE STORAGE IN EXTERNAL MEMORY

Figure 7 describes the format of interleaved YCbCr 4:2:2 picture where all the three components are interleaved. The sample order is selected at initialization to be either Y-Cb-Y-Cr or Cb-Y-Cr-Y.

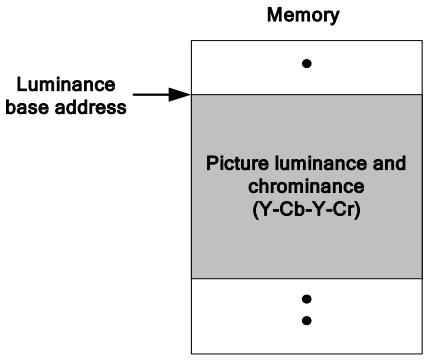

FIGURE 7. INTERLEAVED YCBCR 4:2:2 PICTURE STORAGE IN EXTERNAL MEMORY

The input picture data endianess is set when the encoder software is compiled and can't be changed during run-time. The sizes of the different components are based on the picture's dimensions and the input format:

Planar YCbCr 4:2:0	Y = width x height [bytes] Cb = width/2 x height/2 [bytes] Cr = width/2 x height/2 [bytes]
Semiplanar YCbCr 4:2:0	Y = width x height [bytes] Cb+Cr = width x height/2 [bytes]
Interleaved YCbCr 4:2:2	Y+Cr+Cr = width x height x 2 [bytes]

4.1 Frame size limitations

YCbCr 4:2:0 limitations regarding the input picture size and the encoded picture size:

```
encoded_width % 4  = 0
encoded_height % 2 = 0
input_width >= (encoded_width + 15) & (~15)
input_width % 16 = 0
input_height >= encoded_height
input_height % 2 = 0
```

YCbCr 4:2:2 limitations regarding the input picture size and the encoded picture size:

```
encoded_width % 4  = 0
encoded_height % 2 = 0
input_width >= (encoded_width + 15) & (~15)
input_width % 8 = 0
input_height >= encoded_height
input_height % 2 = 0
```

Where:

%	- remainder operator
&	- bitwise AND operator
~	- bitwise NOT operator

5 Interface Functions

This chapter describes the API declared in *h264encapi.h*.

The following common numeric data types are declared in *basetype.h*:
u8 – unsigned 8 bits integer value
i8 – signed 8 bits integer value
u16 – unsigned 16 bits integer value
i16 – signed 16 bits integer value
u32 – unsigned 32 bits value
i32 – signed 32 bits value

5.1 H264EncGetApiVersion

Syntax

H264EncApiVersion H264EncGetApiVersion(void)

Purpose

Returns the encoder's API version information.

Parameters

None.

Return value

H264EncApiVersion
A structure containing the API's major and minor version number.

Description

```
typedef struct
{
    u32 major;
    u32 minor;
} H264EncApiVersion;
```

major
The major version number.
minor
The minor version number.

5.2 H264EncGetBuild

Syntax

H264EncBuild H264EncGetBuild(void)

Purpose

Returns the hardware and software build information of the encoder. Does not require encoder initialization.

Parameters

None.

Return value

H264EncBuild
A structure containing the encoder ‘s build information

```
typedef struct
{
    u32 swBuild;
    u32 hwBuild;
} H264EncBuild;
```

u32 swBuild
The internal release version of the 7280 H.264 software.
u32 hwBuild
The internal release version of the 7280 hardware.

5.3 H264EncInit

Syntax

H264EncRet H264EncInit(const **H264EncConfig** *pEncCfg,
H264EncInst *instAddr)

Purpose

Call this function to init the encoder and get an instance of it.

Parameters

H264EncConfig *pEncCfg
Points to a structure that contains the encoder’s initial configuration parameters.

H264EncInst *instAddr
Points to a space where the new encoder instance pointer will be stored.

Return value

H264ENC_OK – initialization successful
H264ENC_NULL_ARGUMENT – error, a pointer argument had an invalid NULL value
H264ENC_INVALID_ARGUMENT – error, one of the arguments was invalid
H264ENC_MEMORY_ERROR – error, the encoder was not able to allocate memory
H264ENC_EWL_ERROR – error, the encoder's system interface failed to initialize
H264ENC_EWL_MEMORY_ERROR – error, the system interface failed to allocate memory.

Description

```
typedef struct
{
    H264EncStreamType streamType;
    H264EncProfileAndLevel profileAndLevel;
    u32 width;
    u32 height;
    u32 frameRateNum;
    u32 frameRateDenom;
    H264EncComplexityLevel complexityLevel;
} H264EncConfig;
```

streamType

Specifies the type of the stream generated. The possible values are:
H264_BYTE_STREAM – The encoder will produce H.264 stream in byte stream format.
H264_NAL_UNIT_STREAM – The encoder will produce H.264 NAL units.

profileAndLevel

Specifies according to the standard the profile and level of the generated stream. The possible values are:
H264ENC_BASELINE_LEVEL_1
H264ENC_BASELINE_LEVEL_1_b
H264ENC_BASELINE_LEVEL_1_1
H264ENC_BASELINE_LEVEL_1_2
H264ENC_BASELINE_LEVEL_1_3
H264ENC_BASELINE_LEVEL_2
H264ENC_BASELINE_LEVEL_2_1
H264ENC_BASELINE_LEVEL_2_2
H264ENC_BASELINE_LEVEL_3
H264ENC_BASELINE_LEVEL_3_1
H264ENC_BASELINE_LEVEL_3_2
For more information check the chapter about Initialization and release of the encoder.

width

Valid value range: [96, 1280]
The width of the encoded image in pixels. Horizontal stride is presumed to be the next 16 multiple equal or bigger than this value. Check limitations from 3.1.

height

Valid value range: [96, 1024]
The height of the encoded image in pixels. Check limitations from 3.1.

`frameRateNum`

Valid value range: [1, 65535]

The numerator part of the input frame rate. The frame rate is defined by the `frameRateNum/frameRateDenom` ratio. This value also defines the time resolution as ticks per second.

`frameRateDenom`

Valid value range: [1, 65535]

The denominator part of the input frame rate. This value has to be equal or less then the numerator part `frameRateNum`.

`complexityLevel`

Selects the complexity of the encoding. The possible values are:

`H264ENC_COMPLEXITY_1`

5.4 H264EncRelease

Syntax

H264EncRet H264EncRelease(**H264EncInst** *inst*)

Purpose

Releases an encoder instance. This will free all the resources allocated at the encoder initialization phase.

Parameters

H264EncInst `inst`

The encoder instance to be released. This instance was created earlier with a H264EncInit call.

Return value

`H264ENC_OK` – release successful

`H264ENC_NULL_ARGUMENT` – error, a pointer argument had an invalid NULL value.

`H264ENC_INSTANCE_ERROR` – error, the encoder instance is not valid or corrupted.

5.5 H264EncSetCodingCtrl

Syntax

```
H264EncRet H264EncSetCodingCtrl( H264EncInst inst,
                                 const H264EncCodingCtrl *pCodeParams )
```

Purpose

Sets the encoder's coding parameters. With the exception of the "slice size", the other parameters can be altered just before the stream is started.

Parameters

H264EncInst inst
The instance that defines the encoder in use.

H264EncCodingCtrl *pCodeParams
Pointer to a structure that contains the encoder's coding parameters.

Return value

H264ENC_OK – the setting was successful
H264ENC_NULL_ARGUMENT – error, a pointer argument had an invalid NULL value.
H264ENC_INSTANCE_ERROR – error, the encoder instance is invalid or corrupted.
H264ENC_INVALID_ARGUMENT – error, one of the arguments was invalid, none of the arguments was set.

Description

```
typedef struct
{
    u32 sliceSize;
    u32 seiMessages;
    u32 videoFullRange;
    u32 constrainedIntraPrediction;
    u32 disableDeblockingFilter;
    u32 sampleAspectRatioWidth;
    u32 sampleAspectRatioHeight;
} H264EncCodingCtrl;
```

sliceSize
Valid value range: [0, 63]
Sets the size of a slice in macroblock rows. Zero value disables the use of slices. This parameter can be updated during the encoding process, between any picture encoding. Default value is 0.

`seiMessages`
Valid value range: [0, 1]
Enables insertion of picture timing and buffering period SEI messages into the stream in the beginning of every encoded frame. Has to be set before starting the stream. Default value is 0.

`videoFullRange`
Valid value range: [0, 1]
0 = Y range in [16, 235], Cb&Cr range in [16, 240]
1 = Y, Cb and Cr range in [0, 255]
Input video signal sample range. Has to be set before starting the stream. Default value is 0.

`constrainedIntraPrediction`
Valid value range: [0, 1]
Sets the intra prediction to constrained mode allowing only intra macroblocks to be used for prediction. Has to be set before starting the stream. Default value is 0.

`disableDeblockingFilter`
Valid value range: [0, 2]
Sets the mode of the deblocking filter. Value 0 enables filtering on all macroblock edges. Value 2 disables filtering on edges that are part of a slice border. Value 1 disables entirely the deblocking filter. Has to be set before starting the stream. Default value is 0.

`sampleAspectRatioWidth`
Valid value range: [0, 65535]
Horizontal size of the sample aspect ratio (in arbitrary units), 0 for unspecified. Has to be set before starting the stream.

`sampleAspectRatioHeight`
Valid value range: [0, 65535]
Vertical size of the sample aspect ratio (in arbitrary units), 0 for unspecified. Has to be set before starting the stream.

5.6 H264EncGetCodingCtrl

Syntax

```
H264EncRet H264EncGetCodingCtrl( H264EncInst inst,
                                 H264EncCodingCtrl *pCodeParams )
```

Purpose

Retrieves the current coding parameters in use by the encoder.

Parameters

`H264EncInst` `inst`
The instance that defines the encoder in use.

`H264EncCodingCtrl` `*pCodeParams`
Pointer to a structure where the encoder's coding parameters will be saved.

Return value

H264ENC_OK – the retrieving was successful.
H264ENC_NULL_ARGUMENT – error, a pointer argument had an invalid NULL value.
H264ENC_INSTANCE_ERROR – error, the encoder instance is not valid or corrupted.

Description

For more info about H264EncCodingCtrl see the description for H264EncSetCodingCtrl.

5.7 H264EncSetRateCtrl

Syntax

```
H264EncRet H264EncSetRateCtrl( H264EncInst inst,
                               const H264EncRateCtrl *pRateCtrl )
```

Purpose

Sets the rate control parameters of the encoder.

Parameters

H264EncInst inst
The instance that defines the encoder in use.

H264EncRateCtrl *pRateCtrl
Pointer to a structure where the new rate control parameters are set.

Return value

H264ENC_OK – the setting was successful.
H264ENC_NULL_ARGUMENT – error, a pointer argument had an invalid NULL value.
H264ENC_INSTANCE_ERROR – error, the encoder instance is invalid or corrupted.
H264ENC_INVALID_ARGUMENT – error, one of the arguments was invalid, none of the arguments was set.
H264ENC_INVALID_STATUS – error, the stream was started with HRD enabled and consequently the rate control parameters cannot be altered.

Description

```
typedef struct
{
    u32 pictureRc;
    u32 mbRc;
    u32 pictureSkip;
    i32 qpHdr;
    u32 qpMin;
    u32 qpMax;
    u32 bitPerSecond;
    u32 hrd;
    u32 hrdCpbSize;
    u32 gopLen;
} H264EncRateCtrl;
```

`pictureRc`

Valid value range: [0, 1]
Enables rate control to adjust QP between frames.

`mbRc`

Valid value range: [0, 1]
Enables rate control to adjust QP inside frames.

`pictureSkip`

Valid value range: [0, 1]
Allow rate control to skip pictures if not enough bits are available. When HRD is enabled, the rate control may have to skip frames despite of this value.

`qpHdr`

Valid value range: -1 or [0, 51]
The default quantization parameter used by the encoder. If the rate control is enabled then this value is used just at the beginning of the encoding process. When the rate control is disabled then this QP value is used all the time. -1 lets RC calculate initial QP. Not recommended to be set lower than 10.

`qpMin`

Valid value range: [0, 51]
The minimum QP that can be set by the RC in the stream. Not recommended to be set lower than 10.

`qpMax`

Valid value range: [qpMin, 51]
The maximum QP that can be set by the RC in the stream.

`bitPerSecond`

Valid value range: [10000, 24000000]
The target bit rate in bits per second (bps) when the rate control is enabled. The rate control is considered enabled when pictureRc, mbRc, pictureSkip or hrd is enabled. When HRD is enabled the bitrate has to be within the limits set for the encoder level (See 3.1, Table 3)

`hrd`

Valid value range: [0,1]
Enables the use of Hypothetical Reference Decoder model to restrict the instantaneous bitrate. Enabling the HRD will automatically enable the picture and MB rate control.

`hrdCpbSize`

Valid value range: [0, MaxCPB]
Size in bits of the coded picture buffer (CPB) used by the HRD model. By default the encoder will use the maximum allowed size for the initialized encoder profile&level (See 3.1, Table 3). Setting this to 0 will always restore the default size.

`gopLen`

Valid value range: [1, 150]
Length of group of pictures. Rate control calculates bit reserve for this GOP length. Recommended value is same as the INTRA frame rate.

5.8 H264EncGetRateCtrl

Syntax

```
H264EncRet H264EncGetRateCtrl( H264EncInst inst,
                               H264EncRateCtrl *pRateCtrl )
```

Purpose

Retrieves the current rate control parameters in use by the encoder.

Parameters

`H264EncInst` `inst`

The instance that defines the encoder in use.

`H264EncRateCtrl` `*pRateCtrl`

Pointer to a structure where the new rate control parameters will be saved.

Return value

`H264ENC_OK` – the retrieving was successful
`H264ENC_NULL_ARGUMENT` **–** error, a pointer argument had an invalid NULL value.
`H264ENC_INSTANCE_ERROR` – error, the encoder instance is not valid or corrupted.

Description

For more info about `H264EncRateCtrl` see the descriptions for `H264EncSetRateCtrl`.

5.9 H264EncStrmStart

Syntax

```
H264EncRet H264EncStrmStart(  H264EncInst inst,
                              const H264EncIn *pEncIn,
                              H264EncOut *pEncOut )
```

Purpose

Starts a new H.264 stream.

Parameters

H264EncInst inst
: The instance that defines the encoder in use.

H264EncIn *pEncIn
: Pointer to a structure where the input parameters are provided.

H264EncOut *pEncOut
: Pointer to a structure where the output parameters will be stored.

Return value

H264ENC_OK – the setting was successful
H264ENC_NULL_ARGUMENT – error, a pointer argument had an invalid NULL value.
H264ENC_INSTANCE_ERROR – error, the encoder instance is invalid or corrupted.
H264ENC_INVALID_ARGUMENT – error, one of the arguments was invalid
H264ENC_INVALID_STATUS – error, a stream was already started
H264ENC_OUTPUT_BUFFER_OVERFLOW – error, the output buffer's size was too small to fit the generated stream. Allocate a bigger buffer and try again.

Description

```
typedef struct
{
    u32 busLuma;
    u32 busChromaU;
    u32 busChromaV;
    u32 timeIncrement;
    u32 *pOutBuf;
    u32 busOutBuf;
    u32 outBufSize;
    u32 *pNaluSizeBuf;
    u32 naluSizeBufSize;
    H264EncPictureCodingType codingType;
    u32 busLumaStab;
} H264EncIn;
```

This data structure is common for all the API calls, which generate stream. Not all the fields are used when starting a stream.

Here is the description of the relevant ones:

`pOutBuf`
: Pointer to the output buffer, which will be used to store the generated stream.

`outBufSize`
: Size of the stream buffer described above in bytes. Minimum value is 64.

`pNaluSizeBuf`
: Pointer to a buffer where the encoder will store the sizes of all the created NAL units. A zero value is stored after the last NAL unit.

`naluSizeBufSize`
: Size in bytes of the NAL unit size buffer. Should be at least `3*sizeof(u32)`.

```
typedef struct
{
    H264EncPictureCodingType codingType;
    u32 streamSize;
} H264EncOut;
```

This data structure is common for all the API calls, which generate stream. Not all the fields are used when starting a stream. Here is the description of the relevant ones:

`streamSize`
: The actual size of the generated stream in bytes is stored here. If the call was unsuccessful then this value is not relevant at all and it shall be ignored.

5.10 H264EncStrmEncode

Syntax

```
H264EncRet H264EncStrmEncode( H264EncInst inst,
                              const H264EncIn *pEncIn,
                              H264EncOut *pEncOut )
```

Purpose

Encodes a video frame.

Parameters

H264EncInst `inst`
: The instance that defines the encoder in use.

H264EncIn `*pEncIn`
: Pointer to a structure where the input parameters are provided.

H264EncOut `*pEncOut`
: Pointer to a structure where the output parameters will be saved.

Return value

H264ENC_FRAME_READY – a frame encoding was finished.
H264ENC_NULL_ARGUMENT – error, a pointer argument had an invalid NULL value.
H264ENC_INSTANCE_ERROR – error, the encoder instance is invalid or corrupted.
H264ENC_INVALID_ARGUMENT – error, one of the arguments was invalid.
H264ENC_INVALID_STATUS – error, a stream was not started yet.
H264ENC_OUTPUT_BUFFER_OVERFLOW – error, the output buffer's size was too small to fit the generated stream. The whole frame is lost. New frame encoding has to be started with a larger buffer.
H264ENC_HW_TIMEOUT – error, the wait for a hardware finish has timed out. The current frame is lost. New frame encoding has to be started.
H264ENC_HW_BUS_ERROR – error. This can be caused by invalid bus addresses, addresses that push the encoder to access an invalid memory area. New frame encoding has to be started.
H264ENC_HW_RESET – error. Hardware was reset by external means. The whole frame is lost.
H264ENC_SYSTEM_ERROR – error, a fatal system error was caught. The encoding can't continue. The encoder instance has to be released.
H264ENC_HW_RESERVED – error. Hardware could not be reserved for exclusive access.

Description

```
typedef struct
{
    u32 busLuma;
    u32 busChromaU;
    u32 busChromaV;
    u32 timeIncrement;
    u32 *pOutBuf;
    u32 busOutBuf;
    u32 outBufSize;
    u32 *pNaluSizeBuf;
    u32 naluSizeBufSize;
    H264EncPictureCodingType codingType;
    u32 busLumaStab;
} H264EncIn;
```

busLuma
The bus address of the buffer where the input picture's luminance component (for YCbCr420_PLANAR and YCbCr420_SEMIPLANAR formats) or the whole input picture (for YCbYCr422_INTERLEAVED) is located.

busChromaU
The bus address of the buffer where the input picture's first chrominance component (for YCbCr420_PLANAR) or both chrominance components (for YCbCr420_SEMIPLANAR) is located.

busChromaV
The bus address of the buffer where the input picture's second chrominance component (for YCbCr420_PLANAR) is located.

`timeIncrement`
: The time stamp of the frame relative to the last encoded frame. This is given in ticks. A zero value is usually used for the very first frame. The ticks per second were set at the encoder initialization phase (`frameRateNum`). For the correct rate control functioning is important to set this value appropriately.

`pOutBuf`
: Pointer to the output buffer, which will be used to store the generated stream. Has to be a linear, memory residing buffer. It has to be 8-byte aligned.

`busOutBuf`
: The bus address of the output buffer, required by the hardware operations.

`outBufSize`
: Size of the stream buffer described above in bytes. Minimum value is 4096.

`pNaluSizeBuf`
: Pointer to a buffer where the encoder will store the sizes of all the created NAL units. If the end of the buffer is not reached a zero value is stored after the last significant NAL unit size value. A NULL value will disable the tracing of the sizes. Always provide this if the required stream is `H264_NAL_UNIT_STREAM`.

`naluSizeBufSize`
: Size in bytes of the NAL unit size buffer. A zero value will disable the NAL unit size tracing. Recommended minimum size is `83*sizeof(u32)` bytes.

`codingType`
: The encoding type for specified picture. The possible values are:
`H264ENC_INTRA_FRAME` – the picture should be INTRA coded.
`H264ENC_PREDICTED_FRAME` – the picture should be INTER coded using the previous picture as a predictor.
`H264ENC_NOTCODED_FRAME` – the picture should be encoded as a “not coded” frame ie. all the macroblocks in the picture become skipped.

`busLumaStab`
: The bus address of the buffer where the luminance component of the picture to be stabilized is located. Used only when video stabilization is enabled.

```
typedef struct
{
    H264EncPictureCodingType codingType;
    u32 streamSize;
} H264EncOut;
```

`codingType`
: The encoding type of the last frame as it was encoded into the stream. This can be different from the type requested by the user. It can take the following values:
`H264ENC_INTRA_FRAME` – the frame was INTRA coded. All Intra frame are generated as IDR pictures.
`H264ENC_PREDICTED_FRAME` – the frame was INTER coded
`H264ENC_NOTCODED_FRAME` – the frame was not coded at the rate control’s or application’s request.

`streamSize`

The size of the generated stream in bytes is returned here. If the call was unsuccessful then this value is not relevant at all and it shall be ignored.

5.11 H264EncStrmEnd

Syntax

```
H264EncRet H264EncStrmEnd (   H264EncInst inst,
                              const H264EncIn *pEncIn,
                              H264EncOut *pEncOut )
```

Purpose

Ends a previously started stream. Stream end consist in a NAL unit of type 'End-of-Sequence'. After a stream is ended a new encoding sequence has to begin with H264EncStrmStart.

Parameters

H264EncInst *inst*

The instance that defines the encoder in use.

H264EncIn *pEncIn

Pointer to a structure where the input parameters are provided.

H264EncOut *pEncOut

Pointer to a structure where the output parameters will be saved.

Return value

H264ENC_OK – the setting was successful
H264ENC_NULL_ARGUMENT – error, a pointer argument had an invalid NULL value.
H264ENC_INSTANCE_ERROR – error, the encoder instance is invalid or corrupted.
H264ENC_INVALID_ARGUMENT – error, one of the arguments was invalid
H264ENC_INVALID_STATUS – error, the stream was not started

Description

```
typedef struct
{
    u32 busLuma;
    u32 busChromaU;
    u32 busChromaV;
    u32 timeIncrement;
    u32 *pOutBuf;
    u32 busOutBuf;
    u32 outBufSize;
    u32 *pNaluSizeBuf;
    u32 naluSizeBufSize;
    H264EncPictureCodingType codingType;
    u32 busLumaStab;
} H264EncIn;
```

This data structure is common for all the API calls, which generate stream. Not all the fields are used when ending a stream. Here is the description of the relevant ones:

`pOutBuf`
Pointer to the output buffer, which will be used to store the generated stream.

`outBufSize`
Size of the stream buffer described above in bytes. Minimum value is 64.

`pNaluSizeBuf`
Pointer to a buffer where the encoder will store the sizes of all the created NAL units. A zero value is stored after the last NAL unit.

`naluSizeBufSize`
Size in bytes of the NAL unit size buffer. Should be at least `2*sizeof(u32)`.

```
typedef struct
{
    H264EncPictureCodingType codingType;
    u32 streamSize;
} H264EncOut;
```

This data structure is common for all the API calls, which generate stream. Not all the fields are used when ending a stream. Here is the description of the relevant ones:

`streamSize`
The actual size of the generated stream in bytes is stored here. If the call was unsuccessful then this value is not relevant at all and it shall be ignored.

5.12 H264EncSetPreProcessing

Syntax

```
H264EncRet H264EncSetPreProcessing( H264EncInst inst,
                                    const H264EncPreProcessingCfg *pPreProcCfg )
```

Purpose

Sets the pre-processing block's parameters.

Parameters

`H264EncInst` `inst`
The instance that defines the encoder in use.

`H264EncPreProcessingCfg` `*pPreProcCfg`
Pointer to a structure where the new parameters are set.

Return value

`H264ENC_OK` – the setting was successful.
`H264ENC_NULL_ARGUMENT` – error, a pointer argument had an invalid NULL value.
`H264ENC_INSTANCE_ERROR` – error, the encoder instance is invalid or corrupted.
`H264ENC_INVALID_ARGUMENT` – error, one of the arguments was invalid.

H264ENC_SYSTEM_ERROR – error, a fatal system error was caught. The encoding can't continue. The encoder instance has to be released.

Description

```
typedef struct
{
    u32 origWidth;
    u32 origHeight;
    u32 xOffset;
    u32 yOffset;
    H264EncPictureYuvType yuvType;
    H264EncPictureRotation rotation;
    u32 videoStabilization;
} H264EncPreProcessingCfg;
```

origWidth

This is the input image's full width. This size has to be equal or bigger than the encoded image's width specified at the encoder initialization phase (See H264EncInit). A zero value disables the cropping. It is also restricted depending on the input image format (See Video Frame Storage Format). Horizontal stride is presumed to be the next 16 multiple equal or bigger than this value.
Valid value range: [96, 1920]

origHeight

This is the input image's full height. This size has to be equal or bigger than the encoded image's height specified at the encoder initialization phase (See H264EncInit). A zero value disables the cropping.
Valid value range: [96, 1920]

xOffset

This is the horizontal offset from the top-left corner of the input image to the top-left corner of the encoded image. Keep in mind that the minimum encoded picture width is 96.
Valid value range: [0, 1824]

yOffset

This is the vertical offset from the top-left corner of the input image to the top-left corner of the encoded image. Keep in mind that the minimum encoded picture height is 96.
Valid value range: [0, 1824]

yuvType

Specifies the input YUV picture format. The formats are described in Video Frame Storage Format chapter. The possible values are:
H264ENC_YUV420_PLANAR
H264ENC_YUV420_SEMIPLANAR
H264ENC_YUV422_INTERLEAVED_YUYV
H264ENC_YUV422_INTERLEAVED_UYVY

rotation

Specifies the YUV picture rotation before encoding. The possible values are:
H264ENC_ROTATE_0 – no rotation.
H264ENC_ROTATE_90R – rotates the picture clockwise by 90 degrees before the encoding.

H264ENC_ROTATE_90L – rotates the picture counter-clockwise by 90 degrees before the encoding.

videoStabilization
Enables or disables the video stabilization function. Set to a non-zero value will enable the stabilization. The input image's dimensions (*origWidth, origHeight*) have to be at least 8 pixels bigger than the final encoded image's. Also when enabled the cropping offset (xOffset, yOffset) values are ignored.

For more information check the Video pre-processor usage chapter. If any invalid configuration value will be set the cropping block will keep its old configuration.

5.13 H264EncGetPreProcessing

Syntax

```
H264EncRet H264EncGetPreProcessing( H264EncInst inst,
                                    H264EncPreProcessingCfg *pPreProcCfg )
```

Purpose

Retrieves the current pre-processing parameters in use by the encoder.

Parameters

H264EncInst inst
The instance that defines the encoder in use.

H264EncPreProcessingCfg *pPreProcCfg
Pointer to a structure where the retrieved parameters are saved

Return value

H264ENC_OK – the retrieving was successful
H264ENC_NULL_ARGUMENT – error, a pointer argument had an invalid NULL value.
H264ENC_INSTANCE_ERROR – error, the encoder instance is invalid or corrupted.

Description

For more info about *H264EncPreProcessingCfg* see the descriptions for H264EncSetPreProcessing. When the video stabilization is in use the xOffset and yOffset values hold the latest stabilization result.

5.14 H264EncSetSeiUserData

Syntax

```
H264EncRet H264EncSetPreProcessing( H264EncInst inst,
                                    const u8 *pUserData,
                                    u32 userDataSize )
```

Purpose

Enables or disables writing user data to the encoded stream. The user data will be written in the stream as a SEI message connected to all the following encoded frames. The SEI message payload type is marked as `user_data_unregistered`.

Parameters

`H264EncInst` `inst`

The instance that defines the encoder in use.

`u8` `*pUserData`

Pointer to a buffer containing the user data. Encoder stores a pointer to this buffer and reads data from the buffer during encoding of the following frames. Because the encoder reads data straight from this buffer it must not be freed before disabling the user data writing.

`u32` `userDataSize`

Size of the data in the `pUserData` buffer in bytes. If zero value is given the user data writing is disabled. Invalid value disables user data writing.
Valid value range: 0 or [16, 2048]

Return value

`H264ENC_OK` – the setting was successful.
`H264ENC_NULL_ARGUMENT` **–** error, a pointer argument had an invalid NULL value.
`H264ENC_INSTANCE_ERROR` – error, the encoder instance is invalid or corrupted.

6 Application Example

The following example codes demonstrate the basic usage of the encoder. The code uses the API functions and external functions `ReadYuv420PlanarFrame()`, `SaveStream()`, `AllocateDMABuffer()` and `FreeDMABuffer()`. The examples are simplified and there is no error handling. *`H264TestBench.c`* is a more thorough example on how to use the encoder.

The source code given below is not guaranteed to compile as such.

6.1 Encode 300 frames

```
#include "H264EncApi.h"
#include <stdlib.h>

H264EncRet ret;
H264EncInst encoder;
H264EncConfig cfg;
H264EncCodingCtrl codingCfg;
H264EncRateCtrl rcCfg;
H264EncIn encIn;
H264EncOut encOut;
i32 next, last;

u32 picture_size;
u32 pict_bus_address = 0;
u32 pict_bus_address_stab = 0;

u32 outbuf_size;
u32 outbuf_bus_address = 0;
u32 *outbuf_virt_address = NULL;

/* Step 1: Initialize an encoder instance */

/* 30 fps frame rate */
cfg.frameRateDenom = 1;
cfg.frameRateNum   = 30; /* this is the time resolution also */

/* VGA resolution */
cfg.width        = 640;
cfg.height       = 480;

/* Stream type */
cfg.streamType      = H264ENC_BYTE_STREAM;
cfg.profileAndLevel = H264ENC_BASELINE_LEVEL_3;

cfg.complexityLevel = H264ENC_COMPLEXITY_1;

/* Output buffer size */
outbuf_size = cfg.width * cfg.height;

/* picture buffer size YUV 420 */
picture_size = (cfg.width * cfg.height * 3) / 2;

if((ret = H264EncInit(&cfg, &encoder)) != H264ENC_OK)
{
    /* Handle here the error situation */
    goto end;
```

```
}


/* Step 2: Optional extra encoder configuration
   See the next example code for how to change the
   default encoder parameters
*/

/* Step 3: Allocate linear memory resources. This implementation
   is OS specific and not part of this example. */
AllocDMAMemory(&outbuf_virt_address, &output_bus_address, &outbuf_size);
AllocDMAMemory(NULL, &pict_bus_address, &picture_size);

encIn.pOutBuf    = outbuf_virt_address;
encIn.busOutBuf  = output_bus_address;
encIn.outBufSize = outbuf_size; /* bytes */

/* Step 4: Start the stream */

ret = H264EncStrmStart(encoder, &encIn, &encOut);
if(ret != H264ENC_OK)
{
    /* Handle here the error situation */
    goto close;
}

/* Save the generated stream. This implementation is OS
   Specific and not part of this example. */
SaveStream(encIn.pOutBuf, encOut.streamSize);

/* Step 5: Encode the video frames */

ret = H264ENC_FRAME_READY;
next = 0;
last = 300; /* encode 300 frames */

while( (next <= last) &&
      (ret == H264ENC_FRAME_READY ||
       ret == H264ENC_OUTPUT_BUFFER_OVERFLOW ))
{

    if(next == 0)
        encIn.timeIncrement = 0; /* start time is 0 */
    else
        encIn.timeIncrement = 1; /* time units between frames */

    /* Select frame type */
    if(next == 0)
        encIn.codingType = H264ENC_INTRA_FRAME; /* First frame must be intra */
    else
        encIn.codingType = H264ENC_PREDICTED_FRAME;

    /* Read next frame and get the bus address. This implementation
       is OS specific and not part of this example. */
    ReadYuv420PlanarFrame(pict_bus_address);

   /* we assume one linear buffer for the input image in */
   /* planar YCbCr 4:2:0 format */
    encIn.busLuma = pict_bus_address;
    encIn.busChromaU = encIn.busLuma + cfg.width*cfg.height;
    encIn.busChromaV = encIn.busChromaU + (cfg.width/2) * (cfg.height/2);

    ret = H264EncStrmEncode(encoder, &encIn, &encOut);

    switch (ret)
```

```
        {
            case H264ENC_FRAME_READY:
                SaveStream(encIn.pOutBuf, encOut.streamSize);
                break;

            case H264ENC_SYSTEM_ERROR:
                /* Fatal system error, can't continue */
                next = last;
                break;

            default:
                /* All the others are ERROR codes
                   The next frame will be forced to INTRA coded
                   by the encoder */
                break;
        }

        next++;
    }   /* End of main encoding loop */

    /* Step 6: End stream */
    ret = H264EncStrmEnd(encoder, &encIn, &encOut);
    if(ret != H264ENC_OK)
    {
        /* Handle here the error situation */
        goto close;
    }
    else
    {
        SaveStream(encIn.pOutBuf, encOut.streamSize);
    }

    /* Free all resources */
    FreeDMAMemory(outbuf_virt_address, output_bus_address, outbuf_size);
    FreeDMAMemory(NULL, pict_bus_address, picture_size);

close:

    /* Last Step: Release the encoder instance */
    if((ret = H264EncRelease(encoder)) != H264ENC_OK)
    {
        /*
           There is nothing much to do for an error here,
           just a notification message. There has to be a fault
           somewhere else that caused a failure of the
           encoder release.
        */
    }

end: /* nothing else to do */
```

6.2 Optional encoder settings

```
#include "H264EncApi.h"

H264EncRet ret;
H264EncInst encoder;

H264EncCodingCtrl codingCfg;
H264EncRateCtrl rcCfg;

/* encoder is already initialized and we have an instance */

/* Rate control */

/* get the current settings */
if((ret = H264EncGetRateCtrl(encoder, &rcCfg)) != H264ENC_OK)
{
    /* Handle here the error situation */
}
else
{
    rcCfg.bitPerSecond = 2000*1000; /* 2000kbps */
    rcCfg.qpHdr = 25; /* First frame QP */

    /* keep all the other parameters default */

    /* update the new settings */
    if((ret = H264EncSetRateCtrl(encoder, &rcCfg)) != H264ENC_OK)
    {
        /* Handle here the error situation */
    }
}

/* Coding control */

/* get the current settings */
if((ret = H264EncGetCodingCtrl(encoder, &codingCfg)) != H264ENC_OK)
{
    /* Handle here the error situation */
}
else
{
    codingCfg.sliceSize = 2;  /* 2 macroblock rows in a slice */
    codingCfg.seiMessages = 1; /* insert SEI messages in the stream */

    /* keep all the other parameters default */

    /* update the new settings */
    if((ret = H264EncSetCodingCtrl(encoder, &codingCfg)) != H264ENC_OK)
    {
        /* Handle here the error situation */
    }
}
```

6.3 Encode with video stabilization

```
#include "H264EncApi.h"
#include "basetype.h"
#include <stdlib.h>

H264EncRet ret;
H264EncInst encoder;
H264EncConfig cfg;
H264EncCodingCtrl codingCfg;
H264EncRateCtrl rcCfg;
H264EncPreProcessingCfg preProcCfg;
H264EncIn encIn;
H264EncOut encOut;
i32 last = 0;

u32 picture_size;
u32 pict_bus_address = 0;
u32 pict_bus_address_stab = 0;

u32 outbuf_size;
u32 outbuf_bus_address = 0;
u32 *outbuf_virt_address = NULL;

/* Step 1: Initialize an encoder instance */

/* 30 fps frame rate */
cfg.frameRateDenom = 1;
cfg.frameRateNum   = 30; /* this is the time resolution also */

/* VGA resolution */
cfg.width        = 640;
cfg.height       = 480;

/* Output buffer size */
outbuf_size = cfg.width*cfg.height;

/* Stream type */
cfg.streamType      = H264ENC_BYTE_STREAM;
cfg.profileAndLevel = H264ENC_BASELINE_LEVEL_3;
cfg.complexityLevel = H264ENC_COMPLEXITY_1;

if((ret = H264EncInit(&cfg, &encoder)) != H264ENC_OK)
{
    /* Handle here the error situation */
    goto end;
}

/* Step 2: Setup video stabilization */
if((ret = H264EncGetPreProcessing(encoder, &preProcCfg)) != H264ENC_OK)
{
    /* Handle here the error situation */
    goto close;
}
preProcCfg.videoStabilization = 1;
preProcCfg.yuvType = H264ENC_YUV420_PLANAR;
preProcCfg.rotation = H264ENC_ROTATE_0;
```

```
/* extra 16 pixels for stabilization */
preProcCfg.origWidth = cfg.width + 16;
preProcCfg.origHeight = cfg.height + 16;

if((ret = H264EncSetPreProcessing(encoder, &preProcCfg)) != H264ENC_OK)
{
    /* Handle here the error situation */
    goto close;
}

/* picture buffer size YUV 420 */
picture_size = (preProcCfg.origWidth * preProcCfg.origHeight * 3) / 2;

/* Step 3: Allocate linear memory resources. This implementation
   is OS specific and not part of this example. */
AllocDMAMemory(&outbuf_virt_address, &output_bus_address, &outbuf_size);
AllocDMAMemory(NULL, &pict_bus_address, &picture_size);
AllocDMAMemory(NULL, &pict_bus_address_stab, & picture_size);

encIn.pOutBuf    = outbuf_virt_address;
encIn.busOutBuf  = output_bus_address;
encIn.outBufSize = outbuf_size; /* bytes */

/* Step 4: Start the stream */

ret = H264EncStrmStart(encoder, &encIn, &encOut);
if(ret != H264ENC_OK)
{
    /* Handle here the error situation */
    goto close;
}
else
{
    /* Save the generated stream. This implementation is OS
       Specific and not part of this example. */
    SaveStream(encIn.pOutBuf, encOut.streamSize);
}

/* Step 5: Encode the video frames */

/* Read first frame and get the bus address. This implementation
   is OS specific and not part of this example. */
ReadYuv420PlanarFrame(pict_bus_address);

encIn.timeIncrement = 0; /* start time is 0 */
encIn.codingType = H264ENC_INTRA_FRAME; /* First frame must be intra */

while( !last )
{

   /* Read next frame to be stabilized */
    ReadYuv420PlanarFrame(pict_bus_address_stab);

    /* one linear buffer for the input image in planar YCbCr 4:2:0 format */
    encIn.busLuma = pict_bus_address;
    encIn.busChromaU = encIn.busLuma + cfg.width*cfg.height;
    encIn.busChromaV = encIn.busChromaU + (cfg.width/2) * (cfg.height/2);

    /* stabilization input, only luminance data used */
    encIn.busLumaStab = pict_bus_address_stab;

    /* encode and stabilize in one run */
    ret = H264EncStrmEncode(encoder, &encIn, &encOut);
```

```
    switch (ret)
    {
        case H264ENC_FRAME_READY:
            SaveStream(encIn.pOutBuf, encOut.streamSize);
            break;

        default:
            /* stop for any error */
            last = 1;
     break;
    }

    /* stabilization result if needed */
    H264EncGetPreProcessing(encoder, &preProcCfg)

    /* stabilized image will be encoded next (swap picture buffers) */
    {
        u32 tmp = pict_bus_address;
        pict_bus_address = pict_bus_address_stab;
        pict_bus_address_stab = tmp;
    }

    encIn.timeIncrement = 1; /* time units between frames */
    encIn.codingType = H264ENC_PREDICTED_FRAME; /* use prediction during encoding */

}   /* End of main encoding loop */

/* Step 6: End stream */
ret = H264EncStrmEnd(encoder, &encIn, &encOut);
if(ret != H264ENC_OK)
{
    /* Handle here the error situation */
    goto close;
}

SaveStream(encIn.pOutBuf, encOut.streamSize);

close:

/* Free all resources */
FreeDMAMemory(outbuf_virt_address, output_bus_address, outbuf_size);
FreeDMAMemory(NULL, pict_bus_address, picture_size);
FreeDMAMemory(NULL, pict_bus_address_stab, picture_size);

/* Last Step: Release the encoder instance */
if((ret = H264EncRelease(encoder)) != H264ENC_OK)
{
    /*
       There is nothing much to do for an error here,
       just a notification message. There has to be a fault
       somewhere else that caused a failure of the
       encoder release.
    */
}

end: /* nothing else to do */
```

References

[1] ISO/IEC 14496-10 / ITU-T Recommendation H.264 (2003). Advanced video coding for generic audiovisual services.

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