Codensity™ T408 & T432

Integration & Programming Guide

Reference Number: 19TD001-13
12/17/2021

Table of Contents

1 Legal Notice

Information in this document is provided in connection with NETINT products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in NETINT's terms and conditions of sale for such products, NETINT assumes no liability whatsoever and NETINT disclaims any express or implied warranty, relating to sale and/or use of NETINT products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right.

A "Mission Critical Application" is any application in which failure of the NETINT Product could result, directly or indirectly, in personal injury or death. Should you purchase or use NETINT's products for any such mission critical application, you shall indemnify and hold NETINT and its subsidiaries, subcontractors and affiliates, and the directors, officers, and employees of each, harmless against all claims costs, damages, and expenses and reasonable attorney's fees arising out of, directly or indirectly, any claim of product liability, personal injury, or death arising in any way out of such mission critical application, whether or not NETINT or its subcontractor was negligent in the design, manufacture, or warning of the NETINT product or any of its parts.

NETINT may make changes to specifications, technical documentation, and product descriptions at any time, without notice. The information here is subject to change without notice. Do not finalize a design with this information. The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications.

NETINT, Codensity, and NETINT Logo are trademarks of NETINT Technologies Inc. All other trademarks or registered trademarks are the property of their respective owners.

© 2022 NETINT Technologies Inc. All rights reserved.

2 Codensity T408/T432 Video Transcoders

NETINT provides high-density and efficient video transcoding solutions using the powerful video processing engines inside our Codensity G4 Application-Specific Integrated Circuit (ASIC). NETINT provides multiple stream transcoding functions and services directly to video content providers and Transcoding as a Service (TaaS) providers for integration into their video streaming systems and services. NETINT's functions and services can be used for highly efficient Video-on-Demand file transcoding, as well as real-time live video streaming applications.

This guide provides an overview of NETINT Codensity T408/T432 Massif video transcoding solution parameters, and the ways they could be used when integrating and managing the T4XX transcoding solutions into a customer's transcoding workflow.

3 Introducing the Codensity T408/T432 Massif Video Transcoders

Video content is the number one source of traffic on the Internet. Video is often generated using the ubiquitous H.264 AVC video encoding standard. Newer H.265 HEVC video delivers equivalent quality with up to a 50% reduction in file size and bandwidth requirements, making it the codec of choice for newer video end points and devices. Codensity T408/T432 Massif Video Transcoders (also referred to as T4XX) deliver scalable video transcoding between H.264 AVC and H.265 HEVC formats with up to 8K UHD video resolution.

NETINT © 2022

4 Intended Audience

This document is intended for developers wishing to integrate NETINT transcoding capabilities into their own media systems and customers directly using NETINT video utility programs and servers.

5 Compatibility

Software Compatibility

This guide is intended to be used with NETINT T4XX Video Transcoder software Release 2.6.0

Hardware Compatibility

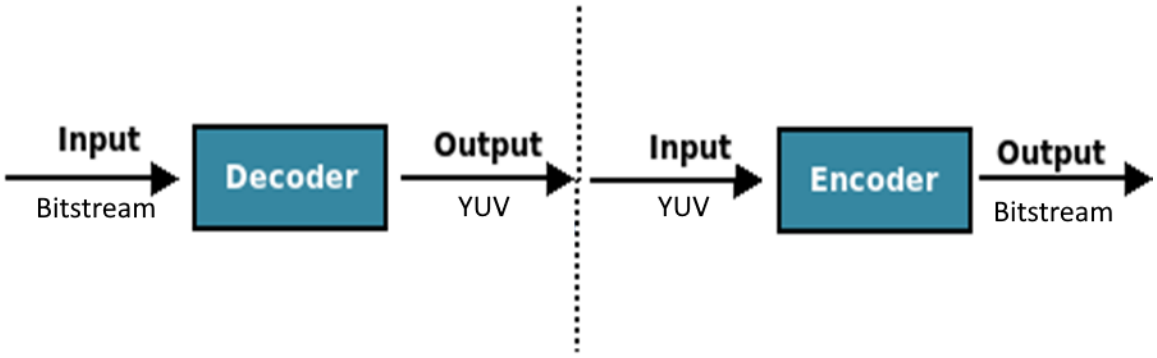
Release 2.6.0 supports NETINT T4XX Video Transcoder hardware.

6 Architecture Overview

The Architecture Overview explains Decoupled Decoding/Encoding and the Protocol Stack.

6.1 Decoupled Decoding and Encoding

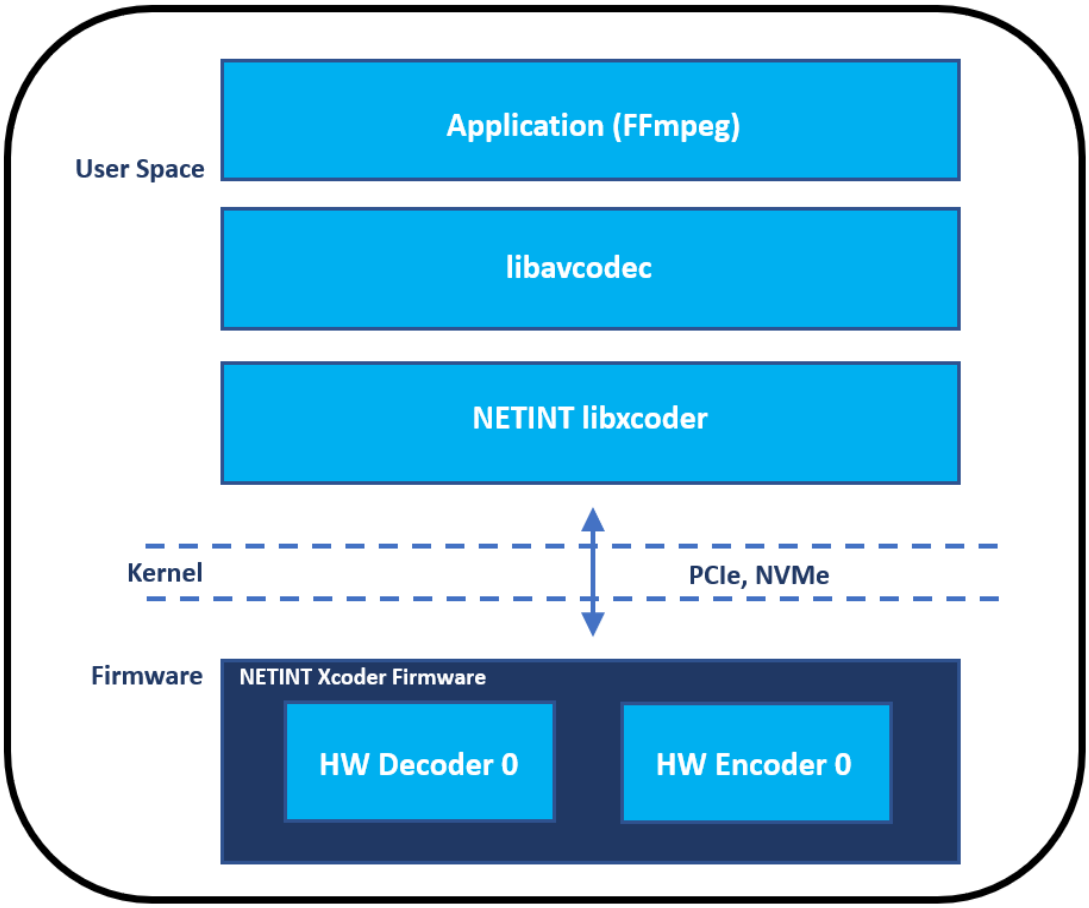
As illustrated in the image below, NETINT video decoding and encoding processes are decoupled from each other. Decoding and encoding engines can be used independently for their respective tasks. This makes it easier to integrate with existing decoding and encoding facilities, for example, a NETINT decoder working with an existing software encoding process, or a NETINT encoder with an existing software decoding process. Alternatively, both decoder and encoder can be placed in a pipeline to take full advantage of the NETINT hardware accelerated transcoding.



Decoding and encoding

6.2 Protocol Stack

The following diagram depicts the software architecture of NETINT transcoding infrastructure.



Transcoding software architecture

In the architecture illustrated in the above diagram, the transcoder hardware is controlled by *NETINT XCoder* firmware running on the hardware. In the user space, NETINT provides an API and library, the *libxcoder*. It provides an interface to the firmware for starting and stopping encoding and decoding instances, sending packets for decoding and retrieving the decoded results, and sending in raw YUV data for encoding and retrieving the encoded result.

The libxcoder is a low-level API employed by a higher layer software, which are codec libraries in most of the cases. Sitting above the libxcoder is usually the codec library that can be used by user applications for transcoding. The *libavcodec*, a free and open-source library of codecs widely used for encoding and decoding video and audio data, is such a library, and has been fully integrated with NETINT transcoding capabilities.

Third-party vendors using libavcodec as their built-in decoding and encoding engines can expect full compatibility after NETINT transcoding is integrated. At application level, those applications using libavcodec such as FFmpeg and NETINT's media processor applications in the NETINT Media Server framework also enjoy full functionality of NETINT transcoding capability

6.3 FFmpeg NETINT Command Options

6.3.1 Decoding

FFmpeg NETINT command options for decoding can be shown using the following command:

ffmpeg -help decoder=<ni_dec_name>

where <ni_dec_name> is *h264_ni_dec* or *h265_ni_dec,* name for NETINT AVC and HEVC decoder respectively. Example:

```
$ ffmpeg -help decoder=h264_ni_dec
Decoder h264_ni_dec [H.264 NetInt decoder v250R1E08]:
    General capabilities: delay avoidprobe
    Threading capabilities: none
h264_ni_dec AVOptions:
  -xcoder           <string>     .D.V..... Select which XCoder card to
use. (default "bestload")
     bestload                   .D.V..... Pick the least loaded
XCoder/decoder available.
     bestinst                   .D.V..... Pick the XCoder/decoder
with the least number of running decoding instances.
     list                       .D.V..... List the available XCoder
cards.
  -dec              <int>       .D.V..... Select which decoder to use
by index. First is 0, second is 1, and so on. (from -1 to INT_MAX)
(default -1)
  -iosize           <int>       .D.V..... Specify a custom NVMe IO
transfer size (multiples of 4096 only). (from -1 to INT_MAX) (default -
1)
  -keep_alive_timeout <int>       .D.V..... Specify a custom session
keep alive timeout in seconds. (from 1 to 100) (default 3)
  -user_data_sei_passthru <boolean>    .D.V..... Enable user data
unregistered SEI passthrough. (default false)
  -custom_sei_passthru <int>       .D.V..... Specify a custom SEI type
to passthrough. (from -1 to 254) (default -1)
  -low_delay        <int>       .D.V..... Specify a decode timeout
value (in milliseconds, recommended value is 600) to enable low delay
mode. Should be used only for streams that are in sequence. (from 0 to
10000) (default 0)
  -xcoder-params    <string>    E..V..... Set the XCoder
configuration using a :-separated list of key=value parameters
```

xcoder specifies which resource allocation strategy to be used to select a decoder for decoding. See section 9.3 for details.

dec assigns the decoding task to a specific decoder by its index.

iosize specifies a custom NVMe I/O transfer size.

keep_alive_timeout specifies a session keep alive timeout value. This is a periodical request/response between libxcoder and XCoder firmware that when timed out, the decoding instance on decoder will be terminated by XCoder firmware.

user_data_sei_passthru specifies to enable user data unregistered SEI passthrough. See App Note APPS020 User data unregistered SEI passthrough for details.

custom_sei_passthru specifies a custom type of SEI to passthrough. See App Note APPS033 Custom SEI passthrough for details.

low_delay enables decoder's low delay mode for in sequence stream by specifying a timeout value. In low delay mode, frames are sent for decoding one at a time and frame reordering is disabled in the decoder. If a decoded frame is not returned to host within the timeout period, the decoder will send another frame. This is done to in the case of bitstream corruption where a frame may be dropped during decoding. If a bitstream with out of sequence frames is received, the frames will be decoded and returned in the order they arrived and will not be reordered. This feature is intended only for low delay gops with in sequence frames.

xcoder-params specifies encoding configuration using a :-separated list of key=value parameters. See section 6.4 for details.

Decoding command example with keep_alive_timeout and low_delay enabled a dec index specified as 0:

```
ffmpeg -y -hide_banner -nostdin -vsync 0 -c:v h264_ni_dec -dec 0 -
keep_alive_timeout 10 -low_delay 600 -
i ../libxcoder/test/akiyo_352x288p25.264 -c:v rawvideo output_5.yuv
```

6.3.2 Encoding

FFmpeg NETINT command options for encoding can be shown using the following command:

ffmpeg -help encoder=<ni_enc_name>

where <ni_enc_name> is *h264_ni_enc* or *h265_ni_enc,* name for NETINT AVC and HEVC encoder respectively. Example:

```
$ ffmpeg -help encoder=h265_ni_enc
Encoder h265_ni_enc [H.265 NetInt encoder v250R1E08]:
    General capabilities: delay
    Threading capabilities: none
h265_ni_enc AVOptions:
  -xcoder           <string>    E..V..... Select which XCoder card to
use. (default "bestload")
    bestload                  E..V..... Pick the least loaded
XCoder/encoder available.
```

```
    bestinst                  E..V..... Pick the XCoder/encoder with the
least number of running encoding instances.
    list                      E..V..... List the available XCoder cards.
  -enc            <int>       E..V..... Select which encoder to use by
index. First is 0, second is 1, and so on. (from -1 to INT_MAX)
(default -1)
  -iosize          <int>      E..V..... Specify a custom NVMe IO
transfer size (multiples of 4096 only). (from -1 to INT_MAX) (default -
1)
  -keep_alive_timeout <int>       E..V..... Specify a custom session
keep alive timeout in seconds. (from 1 to 100) (default 3)
  -xcoder-params    <string>    E..V..... Set the XCoder configuration
using a :-separated list of key=value parameters
  -xcoder-gop       <string>    E..V..... Set the XCoder custom gop
using a :-separated list of key=value parameters
```

xcoder specifies which resource allocation strategy to be used to select an encoder for encoding. See section 9.3 for details.

enc assigns the encoding task to a specific encoder by its index.

iosize specifies a custom NVMe I/O transfer size.

keep_alive_timeout specifies a session keep alive timeout value. This is a periodical request/response between libxcoder and XCoder firmware that when timed out, the encoding instance on encoder will be terminated by XCoder firmware.

xcoder-params specifies encoding configuration using a :-separated list of key=value parameters. See section 6.6 for details.

xcoder-gop specifies a custom GOP for encoding using a :-separated list of key=value parameters. See section 6.6 for details.

Encoding command example:

```
ffmpeg -y -hide_banner -nostdin -f rawvideo -pix_fmt yuv420p -s:v
352x288 -r 25 -i ../libxcoder/test/akiyo_352x288p25.yuv -c:v
h264_ni_enc -keep_alive_timeout 10 output_ 7.h264
```

Finally, a transcoding command example:

```
ffmpeg -y -hide_banner -nostdin -vsync 0 -c:v h264_ni_dec -
keep_alive_timeout 10 -i ../libxcoder/test/1280x720p_Basketball.264 -
c:v h265_ni_enc -keep_alive_timeout 10 output_9.h265
```

6.4 Decoding Parameters

All values are integers.

savePkt

Sets the number of decoder input packets to save into a circular buffer. The range is from 0 to 1000. When value is >0, input packets are saved in the present working directory of the process at path: "./nvme**A**/stream**BB**/pkt-**CCCC**.bin".

A Nvme device index for the T408/T432 card from system's "/dev/nvmeA"

BB Two digits from 01 to 32, each representing a stream on the T408/T432 card. Packets will be saved to a folder with the lowest index if it does not already exist.

When exceeding 32, packets will be saved to the folder with the least recent changes according to file timestamp. This will overwrite previous contents of folder.

CCCC Each "pkt-CCCC.bin" represents one packet. CCCC represents index in circular buffer.

In addition, in the stream folder, there will be a text file named "process_session_id.txt" in which both the process ID and the session ID is written.

default 0

For example a command line to save the last 1000 packets input to the H.264 decoder while transcoding to H.265 would look like this:

```
ffmpeg -c:v h264_ni_dec -xcoder-params savePkt=1000 -i test.264 -c:v
h265_ni_enc -xcoder-params "RcEnable=1:bitrate=5000000" output.265
```

6.5 Encoding Formats

The supported encoding formats of the decoder input and encoder output stream are shown below.

Coder	T408/T432
Decoder Input	H.264 Baseline, Constrained Baseline, Main, High, and High10 profiles up to level 6.2 H.265 Main and Main 10 profiles up to level 6.2. Picture sizes from 32x32 to 8192x5120, bitrates from 64kbit/s to 700Mbit/s, SDR, HDR HLG, HDR10, HDR10+, CEA708 Close Captions
Encoder Output	H.264 Baseline, Extended, Main, High, and High10 profiles up to level 6.2 H.265 Main and Main10 profiles up to level 6.2. Picture sizes from 32x32 to 8192x5120, bitrates from 64kbit/s to 700Mbit/s, SDR, HDR HLG, HDR10, HDR10+, CEA708 Close Captions

Notes:

1. The T408/T432 supports progressive encoded video only. Interlaced video is not supported for encoding or decoding.
2. The T408/T432 H.264 baseline encoder is also compliant with constrained baseline.
3. While picture sizes as small as 32x32 are supported for encoding, they are first padded to 256x128 which is the minimum size of the hardware encoder. Cropping is applied so that they are decoded at 32x32. The decoder hardware can decode down to 32x32.

4. 8192x5120 is supported for encode or decode only and not transcoding due to hardware limitations. 8K (7680x4320) is fully supported for transcoding.

6.6 Encoding Parameters

All values are integers.

level

Sets the level for encoding. The level is a decimal value from 0 to 9.9 in 0.1 increments. If level=0 the encoder will automatically determine the level based on picture size, frame rate, and bitrate, otherwise the specified level be used. When a non-zero level is specified the encoder will use it regardless of the encoder parameters. Valid H.264 levels are: 1, 1.1, 1.2, 1.3, 2, 2.1, 2.2, 3, 3.1, 3.2, 4, 4.1, 4.2, 5, 5.1, 5.2, 6, 6.1, and 6.2. Valid H.265 levels are: 1, 2, 2.1, 3, 3.1, 4, 4.1, 5, 5.1, 5.2, 6, 6.1, and 6.2. Note that we do not support setting H.264 level 1b.
default 0

profile

Sets the profile for encoding. The valid profiles for H.264 and H.265 are shown below. Any profile can be used for 8 bit encoding but only the 10 bit profiles (main10 for H.265 and high10 for h.264) may be used for 10 bit encoding.

H.265:
1=main (8 bit default)
2= main10 (10 bit default)

H.264:
1=baseline (not compatible with B frames)
2=main
3=extended
4=high (8 bit default)
5= high10 (10 bit default)

Note that for H.264 baseline, the gop must not contain B frames, so the only supported values for gopPresetIdx=1, 2, 6, or 0 (custom gop with picType != 3)

losslessEnable

Enables lossless encoding mode for H.265. Lossless encoding bypasses the DCT and quantization stages of the encoder. This results in perfect reconstruction on decode at the cost of a much larger bitstream. Since quantization is bypassed, any feature that relies on quantization is not supported such as rate control, crf, region of interest, hvsQP, constant QP operation, etc. This feature is not supported for H.264. Supported values are:
0: Disable
1: Enable
default 0

frameRate

The numerator of the frame rate. Works in conjunction with frameRateDenom to support fractional framerates. The framerate is used by the encoder for rate control

(when enabled) and to set the VUI timing information. This parameter is intended for integrating directly with libxcoder.
default FFmpeg value

frameRateDenom

The encoder frame rate denominator that supports fraction frame rate together with frameRate. The frame rate would then be frameRate / frameRateDenom, e.g. frameRate=30000 and frameRateDenom=1001 represents frame rate of 30000/1001=29.97. This parameter is intended for integrating directly with libxcoder.
default FFmpeg value

RcEnable

Enables or disables rate control. Rate control is disabled by default (fixed QP mode). Supported values are:
0: Disable
1: Enable
default 0

bitrate

The encoding bitrate, in bits per second (bps). Used when RcEnable is enabled (set to 1); ignored otherwise. The range is 64000 to 700000000. As an example, set bitrate=3000000 for 3 Mbps.
default 200000

intraQP

Specifies the base value of the quantization parameter for I frames when rate control is disabled (RcEnable=0). The range of supported values is 0 to 51. The QP values for P and B frames are determined by the QP offset in the gop structure. See picQP in custom gop structure (section **Error! Reference source not found.**). This section also lists the definitions of all the gop presets.
default 22

RcInitDelay

Specifies the vbvBuffer size and initial buffer fill level in msec. The range of supported values is 10 to 3000. Higher values lead to better visual quality and greater bitrate variance. For example, a value of 3000 will set the rate control buffer model size to 3s * bitrate (bits/s). A decoder will nominally require a buffer larger than 3 * bitrate to accommodate the encoded stream. This allows rate control to target the average bitrate over 3 seconds to track the target bitrate. Greater flexibility for rate control generally improves image quality by allowing more bits for complex scenes whilst reducing bits for simple scenes.
This value is used when RCEnable is 1.
default 3000

crf

Enable or disable Constant Rate Factor (CRF) rate control. This option encodes with constant quality variable bitrate rate encoding. This option forces the following parameters: RcEnable = 0, intraQP = <crf value>, hvsQPEnable = 1, hvsQPScale = 2, maxDeltaQP = 51. The supported values are from 0-51 where lower is better quality.
default disabled (feature is disabled)

decodingRefreshType

Specifies the type of decoding refresh to apply at the intra frame period picture. Supported values are as follows:
0 applies an I picture (not a clean random access point).
1 applies a non-IDR clean random access (CRA) point (H.265 only).
2 applies an IDR random access point.

default 2

intraPeriod

Key frame interval. Must be multiple of GOP size as defined by the gopPresetIdx. The range is 0 to 1024. The I-frame type at the intraPeriod is determined by decodingRefreshType. A value of 0 implies an infinite period.
default 92

flushGop

Enables or disables flushing the GOP at intraPeriod boundaries. This is useful for HLS streaming when using out of sequence GOP patterns. With this parameter set, all frames of the last GOP are flushed before the intraPeriod IDR is inserted. This guarantees that each HLS segment will contain all the frames of that segment. Without this parameter there will always be a few frames at the beginning of each segment belonging to the previous segment. Note that this feature overrides the decodingRefreshType and always uses IDR frames. Supported values are as follows:
0 disable
1 enable
default 0

gopPresetIdx

Defines the group of picture pattern. For custom GOP, and details of the gop presets please see the Custom Gop Parameters Section **Error! Reference source not found.**.
Supported values are as follows:
0 : Custom Gop
1 : I-I-I-I,..I (all intra, gop_size=1)
2 : I-P-P-P,… P (consecutive P, gop_size=1)
3 : I-B-B-B,…B (consecutive B, gop_size=1)
4 : I-B-P-B-P,… (gop_size=2)
5 : I-B-B-B-P,… (gop_size=4)
6 : I-P-P-P-P,… (consecutive P, gop_size=4)
7 : I-B-B-B-B,… (consecutive B, gop_size=4)
8 : I-B-B-B-B-B-B-B-B,… (random access, gop_size=8)
9 : I-P-P-P,… P (consecutive P, gop_size=1, similar to preset 2 but with single reference)
default 5

useLowDelayPocType

When enabled, the encoder will use picture_order_count_type=2 in the H.264 SPS which lets decoders know that all frames are in sequence which typically results in lower delay while decoding. This feature is supported only for H.264 when all frames are in sequence, i.e, when using the low delay gop presets gopPresetIdx=1, 2, 3, 6, 7, and 9.

By default this feature is disabled and the encoder uses picture_order_count_type=0 which is compatible with all gop presets.
0 disables low delay pocType
1 enables low delay pocType
default 0

enableAUD

Specifies whether or not to include access unit delimiters (AUD) in the encoded bitstream. When enabled, access unit delimiters are placed at the boundaries between frames. Some containers such as transport stream require AUDs in the bitstream. AUD is supported for both H.264 and H.265.
0 disables AUD
1 enables AUD
default 0

hrdEnable

Specifies whether or not to include hypothetical reference decoder information (HRD) in the encoded bitstream. When enabled, an HRD section is included in the VUI and two SEI messages, buffering period and pic timing are inserted into the bitstream. Buffering period SEIs are inserted on every IDR (whether forced or intra period generated) and pic timing SEIs are inserted on every frame. The HRD information can be used to compute the fullness of the coded picture buffer (CPB) of the hypothetical reference decoder on a frame by frame basis. HRD is currently supported only for H.265 and FFmpeg 4.2.1 or higher. HRD requires rate control to be enabled and so enabling HRD, causes rate control to also be enabled, i.e, RcEnable=1.
0 disables HRD
1 enables HRD for H.265
default 0

dolbyVisionProfile

Specifies whether or not Dolby Vision compatibility is enabled for H.265 encoding and for what profile. Currently only profile 5 (single base layer) is supported. Setting dolbyVisionProfile=5 enables the profile 5 compatible VUI settings (video_format=5, video_full_range_flag=1, colour_primaries=2, transfer_characteristics=2, matrix_coeffs=2, and chroma_loc_info_present_flag=0) and also forces a number of other parameters required for Dolby Vision compatibility (enableAUD=1, hrdEnable=1, repeatHeaders=1, and decodingRefreshType=2). Dolby Vision compatibility is supported only for H.265 and for FFmpeg 4.2.1 or higher. Dolby Vision compatibility also requires the use of a GOP with all in-sequence frames such as gopPreset 2 or 7.
0 disables dolbyVision compatibility
5 enables dolbyVision profile 5 compatibility
default 0

cuLevelRCEnable (H.265 only)

Enable or disable coding unit level rate control. When enabled, the rate control can reduce or increase the QP mid-frame if needed to maintain rate control. Supported values are as follows:
0: disable
1: enable

default 1

mbLevelRcEnable (H.264 only)

Enable or disable macroblock level rate control. When enabled, the rate control can reduce or increase the QP mid-frame if needed to maintain rate control. Supported values are as follows:
0: disable
1: enable
default 1

hvsQPEnable

Enable or disable MB/CTU QP adjustment for subjective quality enhancement. This parameter works with or without rate control enabled. Supported values are as follows:
0: disable
1: enable
default 0

hvsQpScale

QP scaling factor for CU QP adjustment. The range of supported values is 0 to 4.
default 2

maxDeltaQp

Max delta Qp for rate control. The range of supported values is 0 to 51. This value is used when hvsQPEnable is 1
default 10

minQp

Min Qp for rate control. The range of supported values is 0 to 51.
default 8

maxQp

Max Qp for rate control. The range of supported values is 0 to 51.
default 51

confWinTop

Conformance top window size. This is the number of pixel rows at the top of the picture that should not be displayed when decoding. The range of supported values is 0 to 8192.
default 0

confWinBot

Conformance bottom window size. This is the number of pixel rows at the bottom of the picture that should not be displayed when decoding. The range of supported values is 0 to 8192.
default 0

confWinLeft

Conformance left window size. This is the number of pixel columns at the left side of the picture that should not be displayed when decoding. The range of supported values is 0 to 8192.
default 0

confWinRight

Conformance left window size. This is the number of pixel columns at the right side of the picture that should not be displayed when decoding. The range of supported values is 0 to 8192.
default 0

roiEnable

Enables the Region of Interest (ROI) feature. See the section on ROI below for more information. Supported values are as follows:
0: disable
1: enable
default 0

RoiDemoMode

Enables the ROI demo mode. When ROI is enabled (roiEnable=1), ROIDemoMode permits the ROI feature to be demonstrated using the standard FFmpeg command line without additional application development. ROI demo mode is currently supported only on FFmpeg 3.4.2. Supported values are as follows:
0: disable
1: ROI is enabled on frame 90 with QP=10 for the center 1/3 (vertically) of the picture and QP=40 everything else. ROI is disabled on frame 300. In this case the center 1/3 of the picture with the lower QP will be encoded with much higher quality than the other 2/3.
2: The same as 1 except that the regions are swapped, i.e, the center 1/3 of the picture has QP=40 and the rest is set to QP=10.
default 0

repeatHeaders

Specifies whether or not the encoder repeats the VPS/SPS/PPS headers on all I-frames. For HDR/HDR10+ streams, the HDR SEIs (content light level info, mastering display color volume, and alternative transfer characteristics) are also repeated. Repeated headers permit a bitstream to be decoded mid-stream. Supported values are as follows:
0: disable
1: enable
default 1

GenHdrs

Specifies whether or not the encoder generates headers in advance for user retrieval. The headers are usually stored in FFmpeg AVCodecContext.extradata. User of encoder can retrieve the headers after calling xcoder_encode_init which is the encoder's init callback function.
0: disable
1: enable
default 0

prefTRC

Specifies the preferred transfer characteristics value. Supported values are from 0 to 255. If this parameter is present, the encoder will include an alternative transfer

characteristics SEI in the bitstream with the preferred transfer characteristics field set to the value of this parameter. If the parameter is not present the SEI will not be present. The alternative transfer characteristics SEI is required by ETSI for HLG and specifies an alternative transfer characteristics from that provided in the VUI.

lowDelay

Specifies whether or not to enable the low latency mode in encoding. When low latency mode is enabled, gopPresetIdx must have a value of 1, 2, 3, 6, 7, 9 or 0 with consecutive (in sequence) frames. For more detail see the application note APPS0012 Low latency mode. Supported values are as follows:
0: disable
1: enable
default 0

transform8x8Enable (H.264 only)

Enables 8x8 intra prediction and 8x8 transform. Only compatible with H.264 high and high10 profiles, disabled for other profiles.
0: disable
1: enable
default 1

sliceMode

Works in conjunction with parameter sliceArg
0: single slice per frame
1: multiple slices per frame
default 0

sliceArg

If sliceMode = 1, this represents the number of CTUs/MBs in each slice. Value must be between 1 and the number of 64x64 CTUs (H.265) or 16x16 MBs (H.264) in the picture.

default 0

entropyCodingMode (H.264 only)

Selects the entropy coding mode used in encoding process. Note that CABAC is only compatible with H.264 Main, High, and High10 profiles and is disabled for other profiles.
0: CAVLC
1: CABAC
default 1

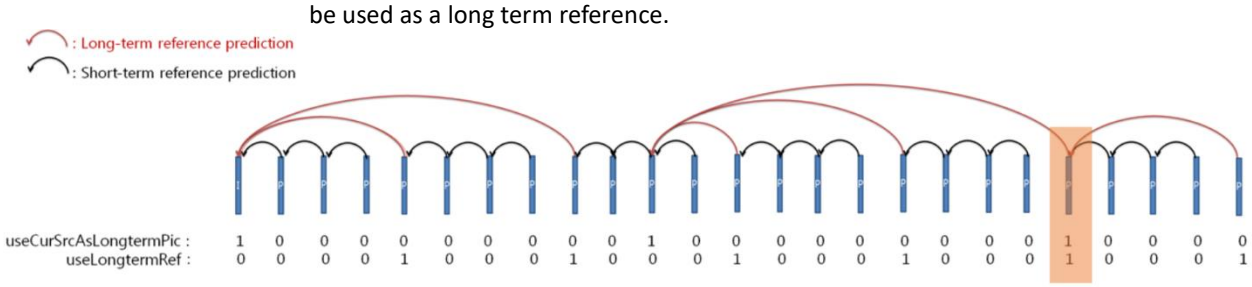
cbr

Enables or disables Constant Bitrate Rate (CBR) control. This option only takes effect when rate control is enabled (RcEnable=1) and the rate control is unable to use all of the configured bitrate. In this case the encoder pads the bitstream with filler NALs to maintain the bitrate at the specified value. Note that this parameter cannot be enabled if crf is specified. Supported values are as follows:
0: disable
1: enable
default 0

longTermReferenceEnable

Enables the long term reference (LTR) feature. With long term reference is enabled, an application can set a couple of parameters on a frame by frame basis, to set a frame to be used as a LTR (useCurSrcAsLongtermPic=1) and to set a frame to use a LTR reference (useLongtermRef=1). Note that only 1 frame can be used as a LTR at a time and so setting a new frame as LTR will clear any previous frames from being used as LTR. Also note that the occurrence of an IDR frame will clear any previous LTR until a new LTR is specified (referencing frames previous to an IDR is not allowed in the standards). One further note is that the encoder supports only 2 reference frames one of which can be LTR. When LTR is used it will replace reference L1 (see section **Error! Reference source not found.**.). LTR is only supported low delay gop structures (i.e. all frames in sequence) such as gopPresetIdx=1, 2, 3, 6, 7, and 9 or for custom gop where all frames are in sequence. Supported values are as follows:
0: disable
1: enable
default 0

The following example shows how the long term reference feature works. The highlighted frame shows that a frame can be set to both use a long term reference and be used as a long term reference.



More information on long term reference can be found in APP note APPS0028 Long Term Reference Frame Application Note.

intraRefreshMode

Intra Refresh coding is an error resilience tool supported for H.264 and H.265 that inserts intra-encoded MBs/CTUs in the encoded bitstream using several configurable modes so that over time the entire image is refreshed without need of an I-frame. The intra refresh interval is specified by another parameter intraRefreshArg. Note that only frames that are used as reference are updated with this feature.
0 : No intra refresh
1 : Row – rows are refreshed from top to bottom
2 : Column – columns are refreshed from left to right
3 : Step size – MBs/CTUs are refreshed with a pattern determined by the encoder
4 : Adaptive intra refresh (AIR) – Adaptive intra refresh as defined in MPEG-4 Part 2 (ISO/IES 14496-2 Annex E). Note that AIR is supported for H.265 only with gopsize=1.
default 0 (intra refresh disabled)

intraRefreshArg

Specifies the intra refresh interval. Depending on intraRefreshMode, it can mean one of the following:
• intraRefreshMode=1: Number of consecutive MB/CTU rows refreshed per frame. Must be less than or equal to the number of MB/CTU rows in the image.
• intraRefreshMode=2: Number of consecutive MB/CTU columns refreshed per frame Must be less than or equal to the number of MB/CTU columns in the image.
• intraRefreshMode=3: Step size in MB/CTU for refresh each frame. Must be less than or equal to the total number of MB/CTU in the image.
• intraRefreshMode=4: Number of MB/CTU for refresh each frame. Must be less than or equal to the total number of MB/CTU in the image.
default 0 (intra refresh disabled)

intraRefreshMinPeriod

Specifies the minimum intra refresh period in frames. This parameter applies to intraRefreshMode=1 through 3. When a non-zero value is specified, intra-refresh will stop after completion of a refresh cycle until intraRefreshMinPeriod frames have elapsed. If the intra refresh takes longer than intraRefreshMinPeriod, then this parameter has no effect and refresh continues as before. Valid values are 0-8191 where 0 disables the feature.
default 0 (no minimum intra refresh period)

Intra Refresh Mode Examples

The 3 images below show the effect of the first 3 refresh modes for an H.265 image with intraRefreshArg=2. The examples show 9 consecutive frames with intra-coded CTUs shaded in orange. The first frame in the upper left corner is an I-frame and so all CTUs are intra encoded.

Example 1 - Row Mode: intraRefreshMode=1 and intraRefreshArg=2 and so two rows of CTUs are refreshed (intra encoded) every frame.

Example 1 – Row Mode

Example 2 - Column Mode: intraRefreshMode=2 and intraRefreshArg=2 and so two columns of CTUs are refreshed (intra encoded) every frame.

Example 2 – Column Mode

Example 3 - Step Size Mode: intraRefreshMode=3 and intraRefreshArg=2 and so two CTUs are refreshed (intra encoded) every frame using an encoder generated pattern.

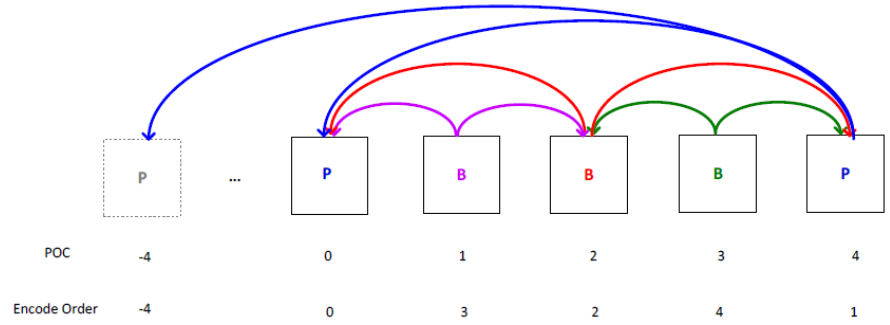
Example 3 – Step Mode

6.7 Custom GOP

The GOP structure table defines a cyclic GOP structure that is used repeatedly throughout the sequence. The frames are listed in encoding order, so Frame1 is the first frame in the encoding order, Frame2 is the second, and so on. Among other things, this table specifies reference pictures used by the current picture. Some specified reference frames for pictures encoded in the very first GOP after an IDR frame might not be available. This is handled automatically by the encoder, so the reference pictures can be given in the GOP structure table as if there were infinitely many identical GOPs before the current one. "customGopSize" defines the number of frames in a GOP structure, the valid range is from 1 to 8.

Element	Description
picType	Picture type. Supported values are as follows: 0: I picture 1: P picture 2: B picture
pocOffset	Display order of the frame within the GOP. The valid range is 1 to customGopSize.
picQp	The offset of the initial frame QP from intraQP for P and B frames. Not used for I frames which always use intraQP. When RcEnable=0 the initial frame QP is used directly, i.e, QP=intraQp+picQP. When RcEnable=1, the initial QP is modified by the rate control. A smaller picQP gives better quality to the frame, a large picQP gives

	poorer quality. This valid range is (0-intraQp) to (51-intraQp). For example, if intraQP=22 (default), the range of picQP is -22 to 29.
numRefPicL0	The number of reference L0 frames for P frames. Not used for other frame types. Valid range is 1 to 2.
refPocL0	The POC of the reference picture L0. Used for P and B frames only. Valid range is -16 to +16.
refPocL1	The POC of the reference picture L1 for B frames. The POC of the second reference picture L0 for P frames. Used only for P and B frames. refPocL1 can be the same as refPocL0 for B pictures, but for compression efficiency it is recommended that they are different . Valid range is -16 to +16.
temporalId	Temporal layer of the frame. A frame cannot use a frame with a higher temporalId as reference. Supported for H.265 only. Valid range is 0 to 15.



Frame#	Type	POC	QPoffset	temporal_id	1st_ref POC	2nd_ref_POC
Frame1	P	4	1	0	0	-4
Frame2	B	2	3	0	0	4
Frame3	B	1	5	0	0	2
Frame4	B	3	5	0	2	4

gopPresetIdx=5 example

In this example, the first frame to process in the “Encode order” is the “P” frame. Then the “B” frame in the middle which use both the I frame (1st_ref POC) and P frame (2nd_ref_POC), note as 0 and 4. The 3rd processed frame is the “B” frame after I frame which use I frame (note as 0) and the second B frame (POC 2 in this GOP) as reference. The last processed frame is the 3rd B frame which use 2 and 4 as reference. The parameters in FFmpeg command line is names with g*n* in the beginning of each parameter in the table where n is from 0 to customGopSize-1.

This is a H264->H264 transcoding example for a custom gop equivalent to gopPresetIdx=8:

```
ffmpeg -c:v h264_ni_dec -i input.264 -c:v h264_ni_enc -xcoder-params
"gopPresetIdx=0:intraPeriod=128:RcEnable=1:bitrate=2000000" -xcoder-gop
"customGopSize=8:g0picType=2:g0pocOffset=8:g0picQp=1:g0numRefPicL0=1:g0
```

```
refPocL0=0:g0refPocL1=-
8:g0temporalId=0:g1picType=2:g1pocOffset=4:g1picQp=3:g1numRefPicL0=1:g1
refPocL0=0:g1refPocL1=8:g1temporalId=0:g2picType=2:g2pocOffset=2:g2picQ
p=5:g2numRefPicL0=1:g2refPocL0=0:g2refPocL1=4:g2temporalId=0:g3picType=
2:g3pocOffset=1:g3picQp=7:g3numRefPicL0=1:g3refPocL0=0:g3refPocL1=2:g3t
emporalId=0:g4picType=2:g4pocOffset=3:g4picQp=7:g4numRefPicL0=1:g4refPo
cL0=2:g4refPocL1=4:g4temporalId=0:g5picType=2:g5pocOffset=6:g5picQp=5:g
5numRefPicL0=1:g5refPocL0=4:g5refPocL1=8:g5temporalId=0:g6picType=2:g6p
ocOffset=5:g6picQp=7:g6numRefPicL0=1:g6refPocL0=4:g6refPocL1=6:g6tempor
alId=0:g7picType=2:g7pocOffset=7:g7picQp=7:g7numRefPicL0=1:g7refPocL0=6
:g7refPocL1=8:g7temporalId=0" -y output.264
```

The custom GOP structure definition should be put into a paragraph that starts with key word: "-xcoder-gop". It must include the parameter "customGopSize" and parameters for each frame. In this example, customGopSize=8, so you have parameters from g0~g7.

This example uses the same setting as gopPresetIdx=8, so if you replace gopPresetIdx=0 to gopPresetIdx=8, it will have the same encoding parameter. gopPresetIdx defines the 9 most commonly used GOP structures. gopPresetIdx=0 allows user to use custom GOP structure.

Here is the predefined GOP structure for H.264 and H.265:

gopPresetIdx	frame#	picType	pocOffset	picQp	numRefPicL0	refPocL0	refPocL1	temporalId
1	1	0	1	0	0	x	x	0
2	1	1	1	1	2	0	-1	0
3	1	2	1	1	X	0	-1	0
4	1	1	2	1	2	0	-2	0
	2	2	1	3	X	0	2	0
5	1	1	4	1	2	0	-4	0
	2	2	2	3	X	0	4	0
	3	2	1	5	X	0	2	0
	4	2	3	5	X	2	4	0
6	1	1	1	5	2	0	-4	0
	2	1	2	3	2	1	0	0
	3	1	3	5	2	2	0	0
	4	1	4	1	2	3	0	0
7	1	2	1	5	X	0	-4	0
	2	2	2	3	X	1	0	0
	3	2	3	5	X	2	0	0
	4	2	4	1	X	3	0	0
8	1	2	8	1	X	0	-8	0

	2	2	4	3	X	0	8	0
	3	2	2	5	X	0	4	0
	4	2	1	8	X	0	2	0
	5	2	3	8	X	2	4	0
	6	2	6	5	X	4	8	0
	7	2	5	8	X	4	6	0
	8	2	7	8	X	6	8	0
9	1	1	1	1	1	0	x	0

6.8 Supported Versions of FFmpeg

Currently supported versions of FFmpeg are 2.7.1, 3.4.2, 4.1.3, 4.2.1, 4.3, 4.3.1, 4.3.2 and 4.4. Both Windows and Linux are supported, however only version 4.2.1 has been validated for Windows. Note that not all Features are supported on all versions of FFmpeg. The reason for this is that some features require support that is not available in older version of FFmpeg. When this is the case it will be clearly stated in the feature description, otherwise the feature should be supported in all FFmpeg versions.

7 Integration

7.1 Transcoding Using FFmpeg

The most straightforward way to transcode with NETINT transcoders is to use the NETINT transcoding enabled application *FFmpeg*. FFmpeg is a vast suite of software libraries and programs for audio/video and other multimedia file and stream handling.

By fully integrating NETINT transcoding into FFmpeg, users can take advantage of all of FFmpeg's existing functionalities like format transcoding, editing, video scaling, video post-production effects, and standards compliance.

Running FFmpeg applications with NETINT transcoders is a simple matter of supplying command line options to FFmpeg. The codec names of the NETINT decoder and encoder are *h264_ni_dec, h265_ni_dec, h264_ni_enc and h265_ni_enc* respectively. The available command line options for the NETINT decoder and encoder are listed below.

To transcode from H.264 to H.265 using the default setting, use the following command (CQP):

```
ffmpeg -hide_banner -r 25 -c:v h264_ni_dec -
i ../libxcoder/test/test_720p.264 -c:v h265_ni_enc output.265
```

To decode H.264 file to raw data (YUV format), use the following command:

```
ffmpeg -hide_banner -c:v h264_ni_dec -i test_720p.264 -c:v rawvideo
output.yuv
```

To encode raw data in YUV format to H.265, use the following command (CQP):

```
ffmpeg -hide_banner -f rawvideo -pix_fmt yuv420p -s:v 1920x1080 -i
input.yuv -c:v h265_ni_enc output.265
```

The command line can be used for passing encoding parameters to NETINT encoder, for example:

```
ffmpeg -c:v h264_ni_dec -i test.264 -c:v h265_ni_enc -xcoder-params
"gopPresetIdx=5:intraPeriod=92:RcEnable=1:RcInitDelay=3000:bitrate=5000
000" output.265
```

Here is an example of 10 bit encoding with a 10 bit YUV 420 little endian input (for big endian input use -pix_fmt yuv420p10be):

```
ffmpeg -f rawvideo -pix_fmt yuv420p10le -s:v 2560x1600 -r 60 -i
2560x1600_60_10bit_le.yuv -c:v h265_ni_enc -xcoder-params
"gopPresetIdx=4:intraPeriod=128:RcEnable=1:RcInitDelay=3000:bitrate=100
00000:decodingRefreshType=1" 10bit2560x1600.265
```

Here is an example of 10 bit decoding with 10 bit YUV 420 little endian output (for big endian output specify -pix_fmt yuv420p10be) :

```
ffmpeg -c:v h265_ni_dec -i 2560x1600_60_10bit.265
2560x1600_10bit_le.yuv
```

Here is an example of 10 bit transcoding:

```
ffmpeg -c:v h265_ni_dec -i input_10bit.265  -c:v h264_ni_enc -xcoder-
params
"gopPresetIdx=4:intraPeriod=128:RcEnable=1:RcInitDelay=3000:bitrate=100
00000:decodingRefreshType=1" output_10bit.264
```

NOTE: In the examples above, hardware decoder and/or encoder instances used in the transcoding have been picked automatically by the NETINT video transcoding resource mechanism that works in the background on the server. User intervention in the video resource management is minimum if you choose to do so. Otherwise users may develop their own resource management schemes based on a NETINT API provided for this purpose. See section 9 for details.

7.2 Feature Support

7.2.1 HDR HLG/HDR/HDR10+/Dolby Vision

The T408/T432 completely supports 3 HDR standards, HLG, HDR10 and HDR10+ for H.264 and H.265 encode and decode. These standards all use 10 bit color for greater dynamic range, a wider range of colors as per ITU-R BT.2020. For Dolby Vision, the T408/T432 supports a compatibility mode such that the Dolby Encoding Engine can use the T408/T432 for single base layer profile 5 Dolby Vision encoding with H.265. This mode is enabled by setting the encoding parameter dolbyVisionProfile=5.

HDR10/10+ use a Perceptual Quantization curve as per SMPTE ST 2084 that supports a much larger range of brightness but is not backwards compatible with standard dynamic range (SDR). The colors of HDR10/10+ content played back on an SDR monitor appear very faded. HLG on the other hand uses the ARIB STD-B67 transfer curve which provides greater dynamic range at high brightness and is backward compatible with the SDR gamma curve at low brightness and so an HLG stream can be played on both SDR and HDR monitors.

The 3 standards specify the color description in the VUI as follows:

Standard	**VUI Color Information**
HLG ATSC A/341	color_primaries=9 (ITU-R BT.2020-2 Wide Gamut Color) transfer_characteristics=18 (ARIB STD-B67 HLG Transfer Curve) matrix_coeffs=9 (ITU-R BT.2020-2 Non-constant Luminance)
HLG ETSI ETSI TS 101 154	color_primaries=9 (ITU-R BT.2020-2 Wide Gamut Color) transfer_characteristics=14 (ITU-R BT.2020-2 Functionally equivalent to BT.709) matrix_coeffs=9 (ITU-R BT.2020 Non-constant Luminance)
HDR10/10+	color_primaries=9 (ITU-R BT.2020-2 Wide Gamut Color) transfer_characteristics=16 (SMPTE ST2084 PQ Transfer Curve) matrix_coeffs=9 (ITU-R BT.2020-2 Non-constant Luminance)

HDR10 and HDR10+ also specifies static metadata containing the parameters of the mastering display using two SEI payloads, content_light_level_info, and mastering_display_color_volume. HDR10+ also adds dynamic metadata that can update the color information on a frame by frame basis. This metadata is stored in T35 SEI payloads as per SMPTE 2094-40.

There are no special commands to enable HDR transcoding. The T408/T432 decoder will pass HDR color information and SEIs up to FFmpeg if the bitstream contains it and the T408/T432 encoder will insert HDR

color information and SEIs in the bitstream if supplied by FFmpeg. Transcoding a compliant HDR10 bitstream will result in a compliant HDR10 bitstream. The same for HDR10+ and HLG.

For HDR encoding, FFmpeg supports specifying the color information on the command line with 3 parameters that map to the VUI color parameters as follows. These parameters may be specified in the input or output sections of the FFmpeg command line. If the color information is specified on the command line, the color information from the input AVFrames to the encoder will be used. These will be properly set by the decoder if transcoding.

FFmpeg Color Parameter	VUI Color Parameter
color_primaries	color_primaries
color_trc	transfer_characteristics
colorspace	matrix_coeffs

The following is an example FFmpeg command line to encode a 10 bit HLG YUV file to H.265 as per ATSC requirements:

```
ffmpeg -f rawvideo -pix_fmt yuv420p10le -s:v 3840x2160 -r 60 -
color_primaries 9 -color_trc 18 -colorspace 9 -i
Input_3840x2160_10bit_le.yuv -enc 0 -c:v h265_ni_enc -xcoder-params
"RcEnable=1:bitrate=20000000" outputATSCHlgT408.265
```

Note that while ETSI specifies transfer characteristics=14 for HLG in the VUI they also specify inclusion of an alternative transfer characteristics SEI that specifies a preferred transfer characteristics of 18. The NETINT decoder will return the preferred transfer characteristics instead of the VUI transfer characteristics if this SEI is present. The NETINT Encoder has a parameter (prefTRC) to specify the inclusion of this SEI and to set it's value.

For example, the following command line to encode a 10 bit HLG YUV file to H.265 as per ETSI requirements is as follows:

```
ffmpeg -f rawvideo -pix_fmt yuv420p10le -s:v 3840x2160 -r 60 -
color_primaries 9 -color_trc 14 -colorspace 9 -I
Input_3840x2160_10bit_le.yuv -enc 0 -c:v h265_ni_enc -xcoder-params
"RcEnable=1:bitrate=20000000:prefTRC=18" outputETSIHlgT408.265
```

Note that FFmpeg does not currently support specifying the static and dynamic metadata for HDR10/10+. We will be supporting this through future libxcoder parameters.

An example of HDR transcoding between H.265 to H.264 is as follows. If the input is 10 bits, then the output will be 10 bits. Any HDR VUI color information from the input bitstream will be transferred to the output bitstream. Any static or dynamic HDR10/10+ metadata from the input bitstream will be transferred to the output bitstream. When a ETSI HLG bitstream is decoded, the preferred transfer characteristics will be used in the VUI of the output bitstream.

```
ffmpeg -dec 0 -c:v h265_ni_dec -i inputHDR.ts -c:a copy -enc 0 -c:v
h264_ni_enc -xcoder-params "RcEnable=1:bitrate=20000000" outputHDR.ts
```

If an ETSI compliant output bitstream is required then the VUI transfer characteristics can be overwritten on the command line and the preferred transfer characteristics specified.

```
ffmpeg -dec 0 -c:v h265_ni_dec -i inputHDR.ts -c:a copy -enc 0 -
color_trc 14 -c:v h264_ni_enc -xcoder-params
"RcEnable=1:bitrate=20000000: prefTRC=18" outputHDR.ts
```

NOTE: HLG is supported in all supported versions of FFmpeg. HDR is supported in FFmpeg version 4.1.3 or higher while HDR10+ and Dolby Vision compatibility are supported only in FFmpeg 4.2.1 or higher.

7.2.2 Region of Interest (ROI)

ROI is a feature of the encoder that permits the quality of some regions to be improved at the expense of other regions. This is done by specifying an ROI map containing the QP for each 16x16 pixel block for H.264, and 32x32 pixel block for H.265. A higher QP means lower quality, a lower QP means higher quality. If rate control is disabled, the QPs are used directly for encoding, if rate control is enabled, the encoder scales the QPs as necessary to meet the bitrate target. When ROI is enabled, the ROI map can be updated, enabled, or disabled on a frame by frame basis.

As of version 4.2.1, FFmpeg supports an API for ROI that permits a number of rectangular ROI regions to be specified. As of version 4.3.1, FFmpeg support an ROI filter (addroi) that permits a number of ROI regions to be specified on the command line. The NETINT encoder supports this API. For more detail see the application note APPS009 Region of Interest.

7.2.3 Closed Captions

The T408/T432 supports EIA CEA-708 closed captions for H.264 and H.265 encode and decode. There are no special encoder parameters to set, the T408/T432 decoder automatically passes closed captions up to FFmpeg if present in the bitstream and the T408/T432 encoder will automatically insert closed captions in the encoded bitstream if they are present in the incoming stream to encoder. FFmpeg stores CE708 closed captions as ATSC A53 Part 4 Closed Captions side data. Closed captions are stored in the encoded bitstreams as T.35 SEI payloads formatted according to CEA-708.

7.2.4 Rate Control

There are 3 rate control modes supported by the NETINT encoder:

CQP: Constant QP mode, enabled by setting RCEnable=0, uses a fixed QP specified by “intraQP” for I-frames plus an offset defined in the GOP structure for other frames. This mode is usually used for encoder quality evaluation and is not recommended to achieve the best encoding efficiency. By default, “RcEnable” parameter is 0 which means CQP mode.

CRF: Constant Rate Factor Mode, enabled by setting the rate factor parameter crf, is similar to constant QP mode except that the QP is distributed within each frame to maximize quality through the use of the hvsQp feature. This option encodes with constant quality using a variable bit rate.

ABR: Average Bitrate Mode, enabled by setting RCEnable=1, varies the QP on a frame by frame basis to maintain an average bitrate as set by the “bitrate” parameter. In this mode, the encoder buffers up an amount of bitstream as specified by the RCInitDelay parameter to perform the rate control. This buffer is typically known as a video buffering verifier or vbv buffer. The larger it is, the better for rate control, but this comes with an increase in delay.

7.2.5 User Data Unregistered SEI Passthrough

The NETINT T4xxx supports passthrough of user data unregistered SEI payloads during transcoding. This can be enabled by specifying the NETINT decoder codec parameter user_data_sei_passthru as per the following example:

```
ffmpeg -c:v h264_ni_dec -user_data_sei_passthru 1 -i input.264 -c:v
h265_ni_enc output.265
```

This feature is intended for passing through smaller user data unregistered SEI messages up to 50 bytes in size. User data may also be input to the NETINT encoder by a customer’s application. For more details see the Application Note APPS0020 User data unregistered SEI passthrough.

7.2.6 Forcing IDR frames

The NETINT encoder supports forcing IDR frames at any point. Forcing an IDR is useful for a number of reasons.

- When doing commercial substitution, an I-frame is required in the bitstream upon returning from the commercial. This frame will likely not coincide with the intra period and so forced IDR frame can be used.
- Another application is for HLS streaming. The NETINT encoder uses an open Gop structure which means that frames from a previous gop can appear after the intra period generated I-frame. This makes HLS segmentation difficult. This can be resolved by disabling intraPeriod I-frames (intraPeriod=0) and using forced IDRs instead. When generating a forced IDR, the encoder flushes out any remaining frames and starts a new gop so in effect it generates a closed gop. See application note APPS0021 HTTP live streaming for more detail.

FFmpeg supports forcing IDRs using the -force_key_frames parameter. This parameter can accept list of frame numbers or times for forcing. It also supports regular expressions in the form of -force_key_frames ‘expr:gte(t,n_forced*REFRESH_PERIOD)’ where REFRESH_PERIOD is the refresh period in seconds (ex. 1,2,etc). The NETINT encoder generates IDR frames in response to FFmpeg key frame requests. The period can also be specified in frames using -force_key_frames ‘expr:gte(n,n_forced*REFRESH_FRAMES)’ where REFRESH_FRAMES is the refresh period in frames.

NOTE: These forced IDR frames are in addition to the periodic I, CRA, or IDR frames generated using the intraPeriod and decodingRefreshType parameters.

An example FFmpeg command line to encode a 1920x1080 YUV420 video to H.265 and force IDR pictures every 2 seconds (-force_key_frames). The intraPeriod parameter is set to zero so that the only I frames are the forced ones:

```
ffmpeg -f rawvideo -pix_fmt yuv420p -s:v 1920x1080 -r 30 -i input.yuv -
force_key_frames 'expr:gte(t,n_forced*2)' -c:v h265_ni_enc
-xcoder-params "intraPeriod=0:RcEnable=1:bitrate=7500000" output.265
```

The force_key_frames parameter can also be used while transcoding to force I-frames at the same positions as in the source file as shown in the following example:

```
ffmpeg -c:v h264_ni_dec -i input.264 -force_key_frames source
-c:v h265_ni_enc -xcoder-params
"intraPeriod=0:RcEnable=1:bitrate=7500000" output.265
```

See the FFmpeg documentation for more information on -force-key_frames parameter. For more information on frame forcing on the NETINT encoder see application note APPS006 Frame Type Forcing.

7.2.7 YUV Bypass

By default, decoded YUV data is transferred back to the host during transcoding and then back again to the T4xx device for encoding. Decoded YUV frames are large and consume a lot of PCIe bandwidth and they take some time to be transferred. YUV Bypass is an optimization to skip the YUV transfers altogether and leave the decoded YUV frames on the device as a hardware frame for encoding. YUV bypass is particularly useful if a decoded stream needs to be encoded multiple times, since the same hardware frame can be used for each encode avoiding even more YUV transfers.

YUV bypass uses a concept in FFmpeg know as a hardware frame. A hardware frame is a YUV frame that exists on an external device such as the T4xx. Hardware frames are specific to a particular device and can only be used on that device unless transferred back to the host to become a normal software frame.

There are two ways to enable the YUV-bypass transcoding:

- xcoder-params “out=hw”
- -hwframes 1

Care must be taken if transcoding an input with sequence changes with hardware frames since by default, FFmpeg will automatically perform scaling at a sequence change. Since scaling does not support hardware frames, we must use the “-noautoscale” parameter to disable scaling at sequence changes.

The following are FFmpeg command line examples for using the YUV-bypass feature:

Regular path transcoding (no YUV Bypass):

```
ffmpeg -vsync 0 -c:v h265_ni_dec -i input.h265 -c:v h264_ni_enc -
xcoder-params "RcEnable=1:bitrate=7500000" output.h264 -y
```

YUV-bypass transcoding:

```
ffmpeg -vsync 0 -c:v h265_ni_dec -dec 0 -xcoder-params "out=hw" -
i input.h265 -c:v h264_ni_enc -enc 0 -xcoder-params
"RcEnable=1:bitrate=7500000" output.h264 -y
```

or

```
ffmpeg -vsync 0 -c:v h265_ni_dec -dec 0 -hwframes 1 -i input.h265 -
c:v h264_ni_enc -enc 0 -xcoder-params "RcEnable=1:bitrate=7500000"
output.h264 -y
```

Regular path transcoding (YUV Bypass explicitly disabled):

```
ffmpeg -vsync 0 -c:v h265_ni_dec -dec 0 -xcoder-params "out=sw" -
i input.h265 -c:v h264_ni_enc -enc 0 -xcoder-params
"RcEnable=1:bitrate=7500000" output.h264 -y
```

or

```
ffmpeg -vsync 0 -c:v h265_ni_dec -dec 0 -hwframes 0 -i input.h265 -
c:v h264_ni_enc -enc 0 -xcoder-params "RcEnable=1:bitrate=7500000"
output.h264 -y
```

YUV-bypass transcoding with Sequence Change:

```
sudo ffmpeg -vsync 0 -c:v h265_ni_dec -dec 0 -xcoder-params "out=hw" -
i input.h265 -noautoscale -c:v h264_ni_enc -enc 0 -xcoder-params
"RcEnable=1:bitrate=7500000" output.h264 -y
```

A hwupload filter can be used to upload a software frame to the device to create a hardware frame for encoding. This can be especially useful if the software frame needs to be encoded multiple times since it only needs to be transferred to the device once. In the following example, "device_name" is an arbitrary name, and needs to match with filter_hw_device. In the example Uploader Device ID (ni=device_name:0) needs to be matched with the encoder ID (-enc 0).

Single Upload Example:

```
sudo ffmpeg -init_hw_device ni=device_name:0 -pix_fmt yuv420p -
s:v 1920x1080 -r 30 -i input.yuv -filter_hw_device device_name -
vf 'format=yuv420p,hwupload' -c:v h265_ni_enc -enc 0 -xcoder-params
" RcEnable=1:bitrate=7500000" output.265 -y
```

A hardware frame can be used in an encoding ladder where the same frame needs to be encoded multiple times with different encoding parameters.

Upload Example with Spilt Filter:

```
sudo ffmpeg -init_hw_device ni= device_name:0 -pix_fmt yuv420p -
s:v 1920x1080 -r 30  -i input.yuv -filter_hw_device device_name -
filter_complex 'format=yuv420p,hwupload,split=2[out1][out2]' -map
'[out1]' -c:v h265_ni_enc -enc 0 -xcoder-params
" RcEnable=1: bitrate=3000000"  dinner_upload_split_low.265 -y -map
'[out2]' -c:v h265_ni_enc -enc 0 -xcoder-params
"RcEnable=1:bitrate=6000000"  dinner_upload_split_high.265 -y
```

Upload Example with Scale Filter:

```
sudo ffmpeg -init_hw_device ni= device_name:0 -pix_fmt yuv420p -
s:v 1920x1080 -r 60  -i input.yuv -filter_hw_device device_name -
vf scale=1280:720,format=yuv420p,hwupload -c:v h265_ni_enc  -xcoder-
params 'RcEnable=1:bitrate=2000000' output.h265
```

A hardware frame can also be downloaded from the device to the host using a hwdownload filter to create a software frame that can then be used for filtering for example. An example where this is useful is when transcoding to an encoding ladder where some encoders require scaling. The decoded hardware frame can be used directly for the encoders that do not require scaling and downloaded to the host for scaling for the encoders that do require scaling. Another example is if a hardware frame is needed on another device for encoding. In this case, the hardware frame can be downloaded from the first device and then uploaded to the second device.

Download YUV Example:

```
ffmpeg -vsync 0 -c:v h264_ni_dec -xcoder-params 'out=hw' -dec 0  -
i input.264 -vf hwdownload,format=yuv420p -c:v rawvideo output.yuv -y
```

Download with Crop Filter in Transcoder:

```
ffmpeg -y -hide_banner -nostdin -vsync 0 -xcoder-params 'out=hw' -dec
0 -c:v h264_ni_dec -i input.264 -
vf hwdownload,format=yuv420p,crop=640:360 -c:v h265_ni_enc -enc
0  output.h265
```

7.3 Integrating with libavcodec

If users don't have access to a command line to run FFmpeg as an executable, or if they want to use just a small part of FFmpeg inside their own program for transcoding using NETINT T408/T432, they can choose to integrate with FFmpeg's libavcodec library, which provides a decoding and encoding API, and all the supported codecs, among them the NETINT T408/T432 decoder and encoder. For details of libavcodec API and its usage example with T408/T432, refer to section 8 libavcodec API.

7.4 Direct libxcoder API Integration

User applications can also be integrated directly with the libxcoder API but this is much more complicated and you lose the rich video functionality provided by FFmpeg.

8 Libavcodec API

8.1 Introduction

FFmpeg is built on top of a few libraries: libavformat, libavcodec,libavdevice,libavutil,libswscale and libavfilter. For basic transcoding operation, we just need libavformat and libavcodec. If need to apply effects, we may have to add libavfilter.

NETINT follows the standard encoder/decoder interface and integrated T408/T432 with libavcodec through libxcoder library. For general user, using FFmpeg command line is the easiest way. Advanced users who want to use libavcodec directly can refer to libavcodec API described in this section. The API for libavcodec is not specific to NETINT T408/T432 but a general interface for any of codecs within libavcodec.

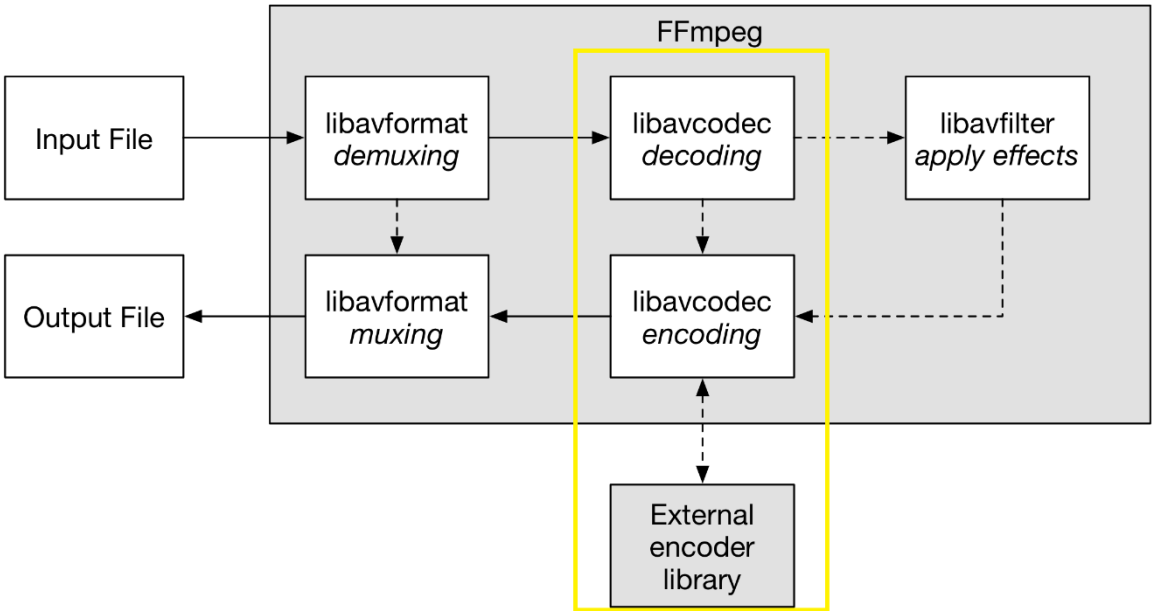


Figure 1 - Libavcodec API

In Figure 1 the input file goes to libavformat and is demuxed to elementary video packets. Coded video packets will be sent to libavcodec for decoding.

Once the packet is decoded, user needs to receive the decoded frame from libavcodec.

For encoding, the raw data frame will be sent to libavcodec and then user needs to receive the coded packet from libavcodec.

The following table lists the libavcodec API functions.

Decoding :	**Description**
avcodec_find_decoder_by_name	find the decoder, for T408/T432, that's h264_ni_dec or h265_ni_dec
avcodec_send_packet	send a coded packet to libavcodec
avcodec_receive_frame	read back a decoded frame from libavcodec
Encoding :	
avcodec_find_encoder_by_name	find the encoder, for T408/T432, that's h264_ni_enc or h265_ni_enc
avcodec_send_frame	send a raw data frame to libavcodec
avcodec_receive_packet	read an encoded packet from libavcodec

From FFmpeg 3.x, avcodec_decode_video2 is not recommended any more, avcodec_send_packet and avcodec_receive_frame is used for decoding. The reason to separate sending packet and receiving frame is to give libavcodec freedom on handling decoding. It can determine when the output is ready, and the write/read operation is completely separated.

NETINT provides decoding and encoding examples in tools/apiexample folder.

For decoding example, please refer to ni_demuxing_decoding.c

For encoding example, please refer to ni_encode_video.c

The decoding and encoding call flows are very similar. Users send data to libavcodec, then check the return. If there is no error (return code >= 0), then try to read data back from libavcodec.

There are 4 different cases:

- AVERROR(EAGAIN): need more input data to generate output
- AVERROR_EOF: no more data to output, the coding is finished.
- <0, error happened
- Other: OK, the data is here!

8.2 Additional API Information

For detailed API function description, please refer to FFmpeg website:

https://FFmpeg.org/documentation.html

Go to API Documentation section, choose the right FFmpeg version. Here we use FFmpeg v4.1 as reference.

8.2.1 Decoding

AVCodec* avcodec_find_decoder_by_name (const char * name)

Find a registered decoder with the specified name.

Parameters

name name of the requested decoder

Returns

A decoder if one was found, NULL otherwise.

int avcodec_send_packet (AVCodecContext * avctx,
const AVPacket * avpkt
)

Supply raw packet data as input to a decoder.

Internally, this call will copy relevant **AVCodecContext** fields, which can influence decoding per-packet, and apply them when the packet is actually decoded. (For example **AVCodecContext.skip_frame**, which might direct the decoder to drop the frame contained by the packet sent with this function.)

Warning: The input buffer, avpkt->data must be AV_INPUT_BUFFER_PADDING_SIZE larger than the actual read bytes because some optimized bitstream readers read 32 or 64 bits at once and could read over the end.

Do not mix this API with the legacy API (like **avcodec_decode_video2()**) on the same **AVCodecContext**. It will return unexpected results now or in future libavcodec versions.

Note: The **AVCodecContext** MUST have been opened with **avcodec_open2()** before packets may be fed to the decoder.

Parameters

avctx codec context

[in] **avpkt** The input **AVPacket**. Usually, this will be a single video frame, or several complete audio frames. Ownership of the packet remains with the caller, and the decoder will not write to the packet. The decoder may create a reference to the packet data (or copy it if the packet is not reference-counted). Unlike with older APIs, the packet is always fully consumed, and if it contains multiple frames (e.g. some audio codecs), will require you to call **avcodec_receive_frame()** multiple times afterwards before you can send a new packet. It can be NULL (or an **AVPacket** with data set to NULL and

size set to 0); in this case, it is considered a flush packet, which signals the end of the stream. Sending the first flush packet will return success. Subsequent ones are unnecessary and will return AVERROR_EOF. If the decoder still has frames buffered, it will return them after sending a flush packet.

Returns

0 on success, otherwise negative error code: AVERROR(EAGAIN): input is not accepted in the current state - user must read output with avcodec_receive_frame() (once all output is read, the packet should be resent, and the call will not fail with EAGAIN). AVERROR_EOF: the decoder has been flushed, and no new packets can be sent to it (also returned if more than 1 flush packet is sent) AVERROR(EINVAL): codec not opened, it is an encoder, or requires flush AVERROR(ENOMEM): failed to add packet to internal queue, or similar other errors: legitimate decoding errors

```
int avcodec_receive_frame ( AVCodecContext *  avctx,
                            AVFrame *         frame
                          )
```

Return decoded output data from a decoder.

Parameters

avctx codec context

frame This will be set to a reference-counted video or audio frame (depending on the decoder type) allocated by the decoder. Note that the function will always call av_frame_unref(frame) before doing anything else.

Returns

0: success, a frame was returned AVERROR(EAGAIN): output is not available in this state - user must try to send new input AVERROR_EOF: the decoder has been fully flushed, and there will be no more output frames AVERROR(EINVAL): codec not opened, or it is an encoder other negative values: legitimate decoding errors

8.2.2 Encoding

```
AVCodec* avcodec_find_encoder_by_name ( const char *  name )
```

Find a registered encoder with the specified name.

Parameters

name name of the requested encoder

Returns

An encoder if one was found, NULL otherwise.

int **avcodec_send_frame** (AVCodecContext * **avctx,**
const AVFrame * **frame**
)

Supply a raw video or audio frame to the encoder.

Use avcodec_receive_packet() to retrieve buffered output packets.

Parameters

avctx codec context

[in] **frame** AVFrame containing the raw audio or video frame to be encoded. Ownership of the frame remains with the caller, and the encoder will not write to the frame. The encoder may create a reference to the frame data (or copy it if the frame is not reference-counted). It can be NULL, in which case it is considered a flush packet. This signals the end of the stream. If the encoder still has packets buffered, it will return them after this call. Once flushing mode has been entered, additional flush packets are ignored, and sending frames will return AVERROR_EOF.

For audio: If AV_CODEC_CAP_VARIABLE_FRAME_SIZE is set, then each frame can have any number of samples. If it is not set, frame->nb_samples must be equal to avctx->frame_size for all frames except the last. The final frame may be smaller than avctx->frame_size.

Returns

0 on success, otherwise negative error code: AVERROR(EAGAIN): input is not accepted in the current state - user must read output with avcodec_receive_packet() (once all output is read, the packet should be resent, and the call will not fail with EAGAIN). AVERROR_EOF: the encoder has been flushed, and no new frames can be sent to it AVERROR(EINVAL): codec not opened, refcounted_frames not set, it is a decoder, or requires flush AVERROR(ENOMEM): failed to add packet to internal queue, or similar other errors: legitimate decoding errors

int **avcodec_receive_packet** (AVCodecContext * **avctx,**

AVPacket * avpkt

)

Read encoded data from the encoder.

Parameters

avctx codec context

avpkt This will be set to a reference-counted packet allocated by the encoder. Note that the function will always call av_frame_unref(frame) before doing anything else.

Returns

0 on success, otherwise negative error code: AVERROR(EAGAIN): output is not available in the current state - user must try to send input AVERROR_EOF: the encoder has been fully flushed, and there will be no more output packets AVERROR(EINVAL): codec not opened, or it is an encoder other errors: legitimate decoding errors

9 Resource Management

A resource management mechanism is in place on the NETINT server for the management of video transcoding resources. It provides functionality for query/allocation of transcoding resources to its users, in the form of utility programs, and a C language library and API that are ready to integrate with third party application software packages such as FFmpeg.

9.1 Transcoding Resources

The transcoding resources on a host are hardware transcoder cards and decoder/encoder chips inside those cards. Each decoder/encoder has a certain processing capacity that can handle a limited number of video streams based on resolution and frame rate. The resource management's tasks are to present inventory and status on available resources and enable resource distribution. User applications can build their own resource management schemes on top of this resource pool or leave this task to the NETINT server for some default simplified resource distribution scheme.

9.2 Device Load and Software Transcoding Instance

At system run time, device firmware maintains a value for each hardware codec representing the processing load currently on the codec. This number is obtained by accumulating clock cycles spent decoding and encoding streams and dividing it by the maximum number of cycles available during a period of time. This reflects how heavy the codec is being used for the stream processing.

For each stream being decoded or encoded, a software decoding or encoding instance is created on the hardware codec. The number of active software transcoding instances on a hardware instance is another measure of load on transcoding resources.

The firmware also tracks the model load for each card. The model load is calculated by width*height*FPS. The model load will be increased from the firmware side once an instance is created successfully, either for encoder or decoder. When an instance is closed successfully the model load will be deducted as well.

9.3 Resource Distribution Strategy

Users may query real time load numbers as described above and devise resource distribution schemes. Some possible strategies are:

- Use the least model loaded codec meeting the capacity requirement for a stream transcoding to maximize the performance (with the smallest delay, such as in the real time streaming applications). In this scheme, it's best for the total processing NOT to exceed the maximum capacity of a codec model load which is 100. This is the default NETINT server behavior.
- Use a pool of reserved codecs for certain types of tasks (offline transcoding for example), by collocating the processing of multiple streams on the same codec as much as possible to maximize the resource usage, without regard for the processing performance.

NOTE: In both examples, the resource management will not reject requests for transcoding resource allocation, even if the request would eventually result in exceeding the maximum processing capacity of

the codec. This allows users to cram high-latency tolerant tasks such as off-line transcoding on a single codec. However, such requests could be rejected by the codec firmware at run-time due to the limit of resources such as memory restraint.

The Resource Management has been integrated into FFmpeg. When running FFmpeg, command line options can be used to exercise the API functions as follows.

The **-xcoder** [strategy] argument specifies which resource allocation strategy shall be used for decoding or encoding. The **-enc** [device] and **-dec** [device] arguments can be used to assign the encoding or decoding task to a specific codec device respectively. See FFmpeg command line option help text for NETINT codecs for details.

Examples.

Allow the least model loaded decoder to be used for decoding (default).

$ ffmpeg -c:v h264_ni_dec -i input.264 output.yuv

Allow the encoder with the least number of running encoding instance to be used for encoding.

$ ffmpeg -i input.yuv -c:v h265_ni_enc -xcoder bestinst output.265

Allow the least loaded encoder that can handle the encoding task in real time to be used for encoding.

$ ffmpeg -i input.yuv -c:v h265_ni_enc -xcoder bestload output.265

Use decoder of index 0 and encoder of index 1 for decoding and encoding respectively.

$ ffmpeg -c:v h264_ni_dec -dec 0 -i input.264 -c:v h265_ni_enc -enc 1 output.265

9.4 NETINT Command-Line Interface (CLI)

A few utility programs are provided to list and monitor resource usage. Running the utility `/usr/local/bin/ni_rsrc_list` produces similar results as that of running FFmpeg with the resource listing option. Another utility is `/usr/local/bin/ni_rsrc_mon`, that actively monitors the resource usage on the server and initializes resources. A sample output is shown below:

```
************************************************
1 devices retrieved from current pool at start up
Sat Apr 18 14:19:58 2020 up 00:00:00 v162R2N02
Num decoders: 1
BEST INDEX LOAD MODEL_LOAD INST DEVICE          NAMESPACE
L    0     0    16         0    /dev/nvme0      /dev/nvme0n1
Num encoders: 1
BEST INDEX LOAD MODEL_LOAD INST DEVICE          NAMESPACE
L    0     0    0          0    /dev/nvme0      /dev/nvme0n1
************************************************
```

10 Resource Management API

At each reboot of NETINT server, the hardware devices are scanned and the information for available transcoding resources is collected and saved in a resource pool. This pool is subsequently queried and managed by applications through the *Resource Management API*. The Resource Management API is provided in the form of a C language API with a header file of C function declarations and structures used for passing information, as well as a C library to be linked with user applications. In addition, utility programs are provided to demonstrate the main functionality and how to integrate with third-party software.

10.1 Device Contexts

When accessing resource pools for query and management, exclusive access is necessary to maintain the resource pool integrity. To provide maximum flexibility and efficiency, a coder context is provided to be used for operations on the codec's information storage.

NOTE All operations in the API, with or without explicit coder context usage, may block the caller, indicating that another user is currently accessing the resource pool for either a particular codec, or the whole resource pool for allocation.

10.1.1 The Device Context Structure

The coder context structure defined below is used in the API for accessing the stored information of a decoder/encoder, which is explained in the following sections.

The `typedef struct ni_device_context_t` is as follows:

```
char   shm_name[NI_MAX_DEVICE_NAME_LEN];
ni_lock_handle_t    lock;
ni_device_info_t * p_device_info;
```

NOTE: This structure is not supposed to be read/written by the caller directly, only be passed in the subsequent calls to API.

The coder context shall be obtained before any operations, including updates and queries, execute on the codec. This is done by providing the coder's type and GUID, a globally unique ID among the same type of coders: decoder or encoder. Based on this information, the coder context is retrieved. The device context should be freed after use.

```
/*!*
* Get the device context. To be used for load update and codec query.
*
* ¥param[in]  type      Decoder or encoder
* ¥param[in]  guid      unique coder(decoder or encoder) id
*
* ¥return     pointer to ni_device_context_t if found, NULL otherwise
* Note:  the returned ni_device_context_t content is not supposed to be used by
*        caller directly: should only be passed to API in the subsequent
*        calls; also after its use, the context should be released by
*        calling ni_rsrc_free_device_context.
*/
LIB_API ni_device_context_t * ni_rsrc_get_device_context(ni_device_type_t type,
int guid);
```

```
        /*!*
        * Free the device context returned by ni_rsrc_get_device_context after use.
        *
        * ¥param[in]  p_ctxt  The device context returned by ni_rsrc_get_device_context
        *
        */

        LIB_API void ni_rsrc_free_device_context(ni_device_context_t *p_ctxt);
```

10.1.2 Retrieve/Free Device Context

The retrieve/free device context sample is as follows:

```
ni_device_info_t *ni_rsrc_get_device_info(ni_device_type_t device_type, int guid)
{
    ni_device_info_t *p_device_info = NULL;
    ni_device_context_t* p_device_context = NULL;

    p_device_context = ni_rsrc_get_device_context(device_type, guid);
    if (NULL == p_device_context)
    {
        LRETURN;
    }

    p_device_info = malloc(sizeof(ni_device_info_t));
    if (NULL == p_device_info)
    {
        LRETURN;
    }

#ifdef _WIN32
    if (WAIT_ABANDONED == WaitForSingleObject(p_device_context->lock, INFINITE))  // no
time-out interval) //we got the mutex
    {
        printf("ERROR: ni_rsrc_get_device_info() failed to obtain mutex: %p¥n",
p_device_context->lock);
        free(p_device_info);
        LRETURN;
    }

    memcpy(p_device_info, p_device_context->p_device_info, sizeof(ni_device_info_t));
    ReleaseMutex(p_device_context->lock);
#elif __linux
    lockf(p_device_context->lock, F_LOCK, 0);

    memcpy(p_device_info, p_device_context->p_device_info, sizeof(ni_device_info_t));

    lockf(p_device_context->lock, F_ULOCK, 0);
#endif

    END;

    ni_rsrc_free_device_context(p_device_context);

    return p_device_info;
}
```

10.2 Device Information

10.2.1 The DeviceCapability Structure

The Resource Manager maintains a data structure that records coder information for each decoder and encoder.

```
typedef struct _ni_device_video_capability
{
    int                     max_res_width; /*! max resolution */
    int                     max_res_height;
    int                     min_res_width; /*! min resolution */
    int                     min_res_height;
    char                  profiles_supported[NI_MAX_PROFILE_NAME_LEN];
    char                    level[NI_MAX_LEVEL_NAME_LEN];
    char                    additional_info[NI_MAX_ADDITIONAL_INFO_LEN];
} ni_device_video_capability_t;

typedef struct _ni_sw_instance_info
{
    int                 id;
    ni_sw_instance_status_t status;
    ni_codec_t          codec;
    int                 width;
    int                 height;
    int                 fps;
} ni_sw_instance_info_t;

typedef struct _ni_device_info
{
    char                    dev_name[NI_MAX_DEVICE_NAME_LEN];
    char                    blk_name[NI_MAX_DEVICE_NAME_LEN];
    int                     hw_id;
    int                     module_id; /*! global unique id, assigned at creation
*/
    int                     load;       /*! p_load value retrieved from f/w */
    int                     model_load; /*! p_load value modelled internally */
    unsigned long           xcode_load_pixel; /*! xcode p_load in pixels: encoder
only */
    int                     fw_ver_compat_warning;
    uint8_t                 fw_rev[8]; // fw revision
    uint8_t                 fw_commit_hash[41];
    uint8_t                 fw_commit_time[26];
    uint8_t                 fw_branch_name[256];

    /*! general capability attributes */
    int                     max_fps_1080p; /*! max fps for 1080p (1920x1080) */
    int                     max_instance_cnt; /*! max number of instances */
    int                     active_num_inst; /*! active numver of instances */
    ni_device_type_t        device_type;     /*! decoder or encoder */
    /*! decoder/encoder capabilities */
    int                     supports_h264; /*! supports "type" (enc/dec) of H.264
*/
    ni_device_video_capability_t h264_cap;

    int                     supports_h265; /*! supports "type" (enc/dec) of H.265
*/
    ni_device_video_capability_t h265_cap;

    ni_sw_instance_info_t sw_instance[NI_MAX_SW_INSTANCE_COUNT];
} ni_device_info_t;
```

10.2.2 Device capability output

Output of ni_rsrc_list program has the following output showing the transcoder card's capability:

```
Num decoders: 1
Decoder #0
  DeviceID: /dev/nvme0
  BlockID: /dev/nvme0n1
  H/W ID: 0
  F/W rev: 200R1A00
  F/W & S/W compatibility: yes
  F/W branch: T408_XCODER_FW_RELEASE_2.0.0
  F/W commit hash: c5f1a7acf411e0bb1da73bc9d87fb698c3b3adc1
  F/W commit time: 2020-04-06 16:04:53 -0700
  MaxNumInstances: 32
  ActiveNumInstances: 0
  Max1080pFps: 240
  CurrentLoad: 0
  H.264Capabilities:
    Supported: yes
    MaxResolution: 8192x5120
    MinResolution: 32x32
    Profiles: Baseline, Constrained Baseline, Main, High, High10
    level: Level 6.2
    additional info:
  H.265Capabilities:
    Supported: yes
    MaxResolution: 8192x5120
    MinResolution: 32x32
    Profiles: Main, Main10
    level: Level 6.2 Main-Tier
    additional info:
  num. s/w instances: 0

Num encoders: 1
Encoder #0
  DeviceID: /dev/nvme0
  BlockID: /dev/nvme0n1
  H/W ID: 1
  F/W rev: 200R1A00
  F/W & S/W compatibility: yes
  F/W branch: T408_XCODER_FW_RELEASE_2.0.0
  F/W commit hash: c5f1a7acf411e0bb1da73bc9d87fb698c3b3adc1
  F/W commit time: 2020-04-06 16:04:53 -0700
  MaxNumInstances: 32
  ActiveNumInstances: 0
  Max1080pFps: 240
  CurrentLoad: 0
  H.264Capabilities:
    Supported: yes
    MaxResolution: 8192x5120
    MinResolution: 32x32
    Profiles: Baseline, Extended, Main, High, High10
    level: Level 6.2
    additional info:
  H.265Capabilities:
    Supported: yes
    MaxResolution: 8192x5120
```

```
    MinResolution: 32x32
    Profiles: Main, Main10
    level: Level 6.2 Main-Tier
    additional info:
  num. s/w instances: 0
```

10.2.3 List All Devices

This API function retrieves information of all the decoders and encoders of the system from the coder info storage.

```
/*!*
* List all the devices (encoder and decoder) with their full info including
* s/w instances on the system.
*
* ¥param[out]  p_device  The device' info returned.
*
* ¥return 0 on success, < 0 failure
* Note: caller is responsible for allocating enough memory for "p_device".
*/
LIB_API int ni_rsrc_list_all_devices(ni_device_t *p_device);
```

10.2.4 List Information for Selected Devices

Another function can be used to retrieve detailed information of all the decoders or encoders of the system.

```
/*!*
* List device(s) of a certain type (encoder/decoder) with their full info
* including s/w instances on the system.
*
* ¥param[in]   type    Decoder or encoder
* ¥param[out]  p_device  The p_device' info returned.
* ¥param[out]  p_device_count     The number of ni_device_info_t returned.
* ¥return 0 on success, < 0 failure
* Note: caller is responsible for allocating enough memory for "p_device".
*/
LIB_API ni_retcode_t ni_rsrc_list_devices(ni_device_type_t device_type,
                    ni_device_info_t *p_device_info, int *p_device_count);
```

10.2.5 Retrieve Detailed Information for a Particular Device

Another one can be used for one coder's detailed info query.

```
/*!*
* Query a specific device with detailed info on the system.
*
* ¥param[in]  type      Decoder or encoder
* ¥param[in]  guid      unique device(decoder or encoder) id
*
* ¥return     pointer to ni_device_info_t if found, NULL otherwise
* Note: caller is responsible for releasing the memory allocated for
*       coder info.
*/
LIB_API ni_device_info_t* ni_rsrc_get_device_info(ni_device_type_t device_type,
int guid);
```

10.2.6 Update Device Information

After obtaining the device context, the device coder's load value can be updated by passing in the context object and latest load values. This is usually done by the resource management programs.

```
/*!*
* Update the load value and s/w instances info of a specific decoder or
* encoder. This is used by resource management daemon to update periodically.
*
* ¥param[in]  p_ctxt  The coder context returned by ni_rsrc_get_device_context
* ¥param[in]  p_load  The latest p_load value to update
* ¥param[in]  nb_sw_insts Number of s/w instances
* ¥param[in]  sw_insts Info of s/w instances
*
* ¥return 0 on success, < 0 failure
*/
LIB_API int ni_rsrc_update_device_load(ni_device_context_t *p_ctxt, int load,
                    int nb_sw_insts, const ni_sw_instance_info_t sw_insts[]);
```

10.3 Resource Allocation

The distribution of encoding/decoding processing resource is across all the transcoder cards on the same host, i.e. a stream's decoding and encoding are not restricted to co-locating on the same chip. Two types of resource allocation approaches are available: user directed allocation, auto-allocation.

10.3.1 User-Directed Resource Allocation

In user directed resource allocation, users devise their own resource allocation scheme based on the resource pool information provided by the resource manager API and utility programs. Typical scenario involves user querying and getting detailed information of the resource pool, and based on the obtained information, explicitly specifying which decoder/encoder to use in the transcoding. This procedure allows finer and total control of resource allocation by users. However, users must take care not to create any race conditions with multiple applications accessing the resource info that may inadvertently put too much load on a single codec, since there is always time gap between query and allocation operations. The following API is used for this use case.

```
/*!*
* Allocate resources for decoding/encoding, by designating explicitly
* the device to use.
*
* ¥param[in]  type      Decoder or encoder
* ¥param[in]  guid      unique coder(decoder or encoder) module id
* ¥param[in]  codec     EN_H264 or EN_H265
* ¥param[in]  width     width of video resolution
* ¥param[in]  height    height of video resolution
* ¥param[in]  frame_rate video stream frame rate
* ¥param[out] p_load      the p_load that will be generated by this encoding
*                        task. Returned *only* for encoder for now.
*
* ¥return     pointer to ni_device_context_t if found, NULL otherwise
* Note:  codec, width, height, fps need to be supplied by encoder; they
*        are ignored for decoder.
* Note:  the returned ni_device_context_t content is not supposed to be used by
*        caller directly: should only be passed to API in the subsequent
*        calls; also after its use, the context should be released by
*        calling ni_rsrc_free_device_context.
*/
```

```
LIB_API ni_device_context_t* ni_rsrc_allocate_direct(ni_device_type_t type, int
guid,
                                     ni_codec_t codec,
                                     int width, int height, int frame_rate,
                                     unsigned long *p_load);
```

10.3.2 Auto Resource Allocation

This allocation procedure leaves the query and allocation task to the NETINT resource management, and it is done automatically so that the race condition mentioned above can be eliminated. Two auto-allocation rules can be specified by user: least-load (the codec that has the least load value available shall be picked), least-instances (the codec that has the least number of transcoding software instances shall be picked). When not specified, the default rule is least-load.

```
/*!*
* Allocate resources for decoding/encoding, based on the provided rule
*
* ¥param[in]  type      Decoder or encoder
* ¥param[in]  rule      allocation rule
* ¥param[in]  codec     EN_H264 or EN_H265
* ¥param[in]  width     width of video resolution
* ¥param[in]  height    height of video resolution
* ¥param[in]  frame_rate video stream frame rate
* ¥param[out] p_load      the p_load that will be generated by this encoding
*                        task. Returned *only* for encoder for now.
*
* ¥return     pointer to ni_device_context_t if found, NULL otherwise
* Note:  codec, width, height, fps need to be supplied by encoder; they
*        are ignored for decoder.
* Note:  the returned ni_device_context_t content is not supposed to be used by
*        caller directly: should only be passed to API in the subsequent
*        calls; also after its use, the context should be released by
*        calling ni_rsrc_free_device_context.
*/
LIB_API ni_device_context_t* ni_rsrc_allocate_auto(ni_device_type_t type,
                                ni_alloc_rule_t rule,
                                ni_codec_t codec,
                                int width, int height, int frame_rate,
                                unsigned long *p_load);
```

10.3.3 Sample usage

The following is a simple example of allocating decoding resources on a specific codec (decoder GUID 10) for decoding a stream of resolution 1080p (1920x1080) at frame rate of 30.

```
        ni_device_context_t *p_ctxt = NULL;
unsigned long model_load;
        p_ctxt = ni_rsrc_allocate_auto ( NI_DEVICE_TYPE_DECODER, 10, EN_H264, 1920, 1080,
        30, &model_load);

        if (p_ctxt) {

          /* codec operations here … */

            ni_rsrc_free_device_context (p_ctxt);
        }
```

Here is another example of querying all the codec information on the host.

```
          int i;
         ni_device_t  devices = {0};
         if (ni_rsrc_list_all_devices (&devices) == 0) {
           /* print out devices in the order based on their guid */
    printf("Num decoders: %d¥n", devices.decoders_cnt);
           for (i = 0; i < devices.decoders_cnt; i++) {
             ni_rsrc_print_device_info (&(devices.decoders[i]));
           }
           printf("Num encoders: %d¥n", devices.encoders_cnt);
           for (i = 0; i < devices.encoders_cnt; i++) {
             ni_rsrc_print_device_info (&(devices.encoders[i]));
           }
         }
```

11 Debugging

11.1 NETINT Codec Library Debug Log

The NETINT Codec Library (including libxcoder) provides full logging of event sequences and information, as well as the log timestamp in run time for troubleshooting and debugging purposes.

When using NETINT Codec Library, the logging is implemented such that libxcoder will use the same logging level as what is specified by FFmpeg's command line option "**-loglevel**". Reference FFmpeg manual page for details.

If your application imports libxcoder directly, the logging level may be set by importing ni_utils.h and calling the ni_log_set_level() function. Please refer to the below code excerpt from ni_util.h for enumerations and functions relevant to libxcoder logging.

```
typedef enum
{
  NI_LOG_NONE  = 0,
  NI_LOG_FATAL = 1,
  NI_LOG_ERROR = 2,
  NI_LOG_INFO  = 3,
  NI_LOG_DEBUG = 4,
  NI_LOG_TRACE = 5
} ni_log_level_t;

void ni_log_set_level(ni_log_level_t level);
ni_log_level_t ni_log_get_level(void);
ni_log_level_t ff_to_ni_log_level(int fflog_level);
```

12 List of Application Notes

A number of Application Notes have been written to explain details of a certain feature in NETINT transcoding:

- APPS001 NETINT Encoder quality
- APPS002 Operation Troubleshooting
- APPS003 NUMA IO performance optimization
- APPS004 Crash Auto Recovery
- APPS006 Frame Type Forcing
- APPS007 QP Forcing
- APPS008 Encode reconfiguring
- APPS009 Region of Interest
- APPS010 Sequence Change
- APPS012 Low latency mode
- APPS014 NVME IO size
- APPS015 BD-RATE calculation
- APPS016 HLG –VUI Parameters
- APPS017 libxcoder API examples
- APPS018 Crash Recovery
- APPS019 Bitrate reconfiguration
- APPS020 User data unregistered SEI passthrough
- APPS021 HTTP live streaming
- APPS022 Android setup
- APPS023 Crash Recovery (Windows VM) Application Note
- APPS025 SR-IOV Configuration and Usage Guide
- APPS026 Multi-NameSpaces Application Note
- APPS027 H.265 Encoder Algorithm Tuning Application Note
- APPS028 Long Term Reference Frame Application Note

- APPS029 Rate control (RC) related parameters dynamic change Application Note
- APPS030 Intra parameters reconfig Application Note
- APPS031 VUI reconfig Application Note
- APPS032 Latency Reporting Application Note
- APPS033 Custom SEI passthrough Application Note
- APPS034 Nvidia GPUDirectForVideo Application Note
- APPS035 NVMe-oF RDMA and TCP Measurements
- APPS036 Kubernetes Configuration Application Note
- APPS039 Host Memory and User Process Optimization
- APPS040 Docker Interworking
- APPS041 SMBus App Note
- APPS042 Encoder Latency Measurement Procedure App Note
- APPS043 Vendor get-log Power Measurement App Note
- APPS044 Temperature Sensor Thresholds App Note