NETINT

Codensity™ T408 & T432 Massif™ libxcoder_logan API Guide

07/20/2022

Table of Contents

1 Legal Notice

2 Table of Abbreviations

Abbreviation:	**Full Form:**
GOP	Group of Pictures
HW	Hardware
FW	Firmware
SEI	Supplemental Enhancement Information
HDR	High Dynamic Range
ROI	Region Of Interest
RBSP	Raw Byte Sequence Payload
LTR	Long Term Reference
SEI	Supplemental Enhancement Information
NAL	Network Abstraction layer
VUI	Video Useability Information
SPS	Sequence Parameter Set

3 References

[1] NETINT T408 & T432 Massif^{TM} Integration & Programming Guide

[2] HDR10+: SMPTE ST-2094-40: https://www.atsc.org/wp-content/uploads/2018/02/S34-301r2-A341-Amendment-2094-40.pdf

[3] NETINT frame type forcing app note : APPS006

[4] NETINT Reconfiguration app note : APPS008

[5] NETINT ROI app note : APPS009

[6] NETINT sequence change app note : APPS010

[7] NETINT low latency mode app note : APPS012

[8] NETINT Bitrate reconfiguration app note : APPS019

[9] NETINT User data unregistered SEI passthrough: APPS020

[10]NETINT Long term reference app note : APPS028

[11]NETINT rate control (RC) related parameters reconfiguration: APPS029

[12]NETINT intra parameters reconfiguration : APPS030

[13]H.264 Standard: T-REC-H.264: https://www.itu.int/rec/T-REC-H.264

[14]H.265 Standard: T-REC-H.265: https://www.itu.int/rec/T-REC-H.265

[15]Closed caption standards: CEA807, ATSC A/53: https://shop.cta.tech/products/digital-television-dtv-closed-captioning

4 Background

This document describes the libxcoder_logan API, its usage and integration with the NETINT T408 and T432 transcoder of Logan product line.

4.1 Intended Audience

This document is intended to help engineers/technicians/developers using NETINT transcoders better understand the libxcoder_logan API.

4.2 Compatibility

Software Compatibility

This guide is intended to be used with NETINT Codensity T4xx Video Transcoder software Release 3.x.

Hardware Compatibility

Release 3.x supports NETINT Codensity T4xx Video Transcoder hardware.

Operation system

All OS supporting T4xx Video Transcoder.

5 libxcoder_logan API

5.1 Overview

The NETINT libxcoder_logan API is the lowest level at which a user application integrates with T4xx transcoder cards, as shown in Figure 1.

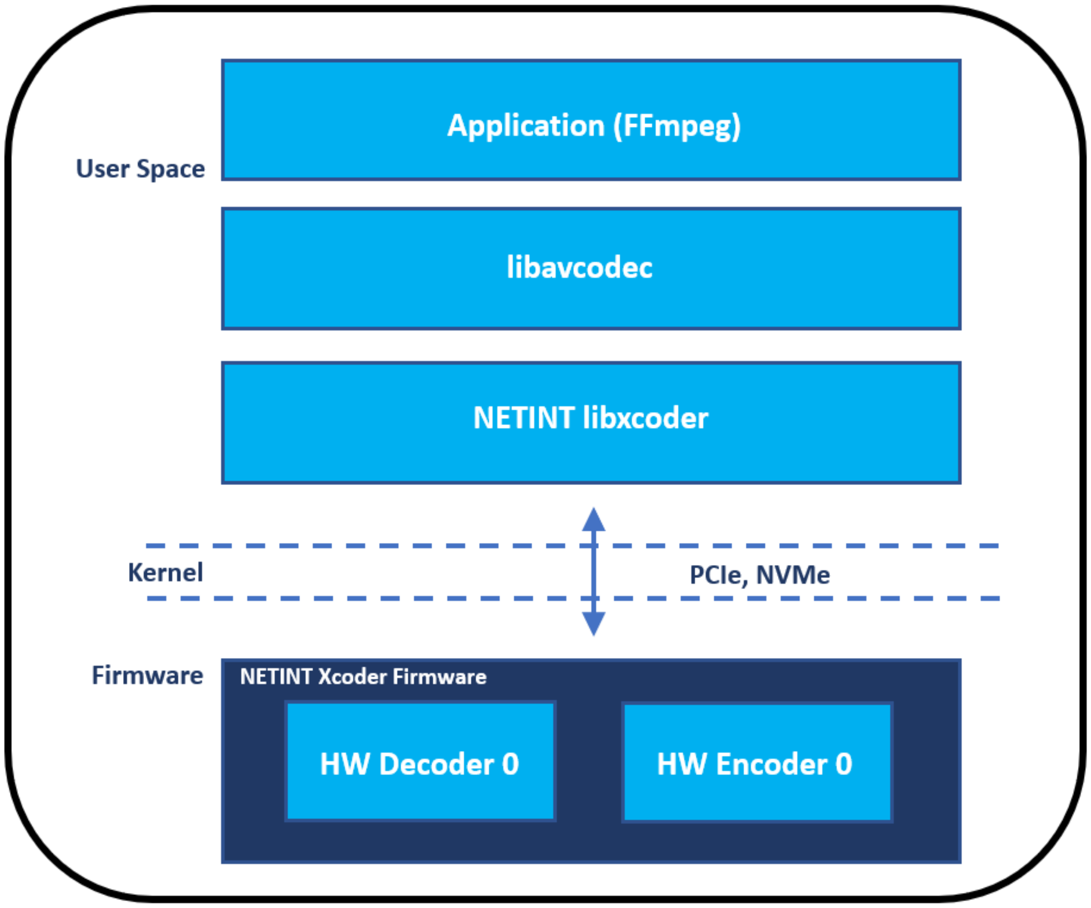


Figure 1 NETINT transcoder software framework

Figure 2 shows the NETINT decoding/encoding/transcoding process. The dotted line is the division between the encoding and decoding processes which can be used independently, or linked together as shown for transcoding.

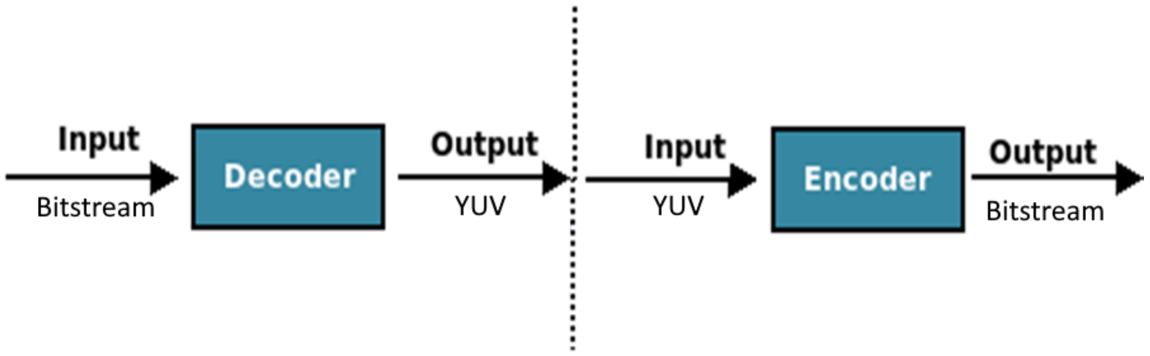


Figure 2 NETINT Decoding and Encoding Pipeline

5.2 Listed functions

The following table lists the libxcoder_logan API functions.

For finer details, check the Appendix Section 7.1 "API Description Details"

API Function	**Description**
ni_device_api_logan.h :	
ni_logan_device_session_context_alloc_init	Allocate and initialize a new ni_logan_session_context_t struct
ni_logan_device_session_context_init	Initialize already allocated session context to a known state
ni_logan_device_session_context_free	Frees previously allocated session context
ni_logan_create_event	Creates event and returns event handle if successful (**Windows only**)
ni_logan_close_event	Closes event and releases resources (**Windows only**)
ni_logan_device_open	Opens device and returns device device_handle if successful
ni_logan_device_close	Closes device and releases resources
ni_logan_device_capability_query	Queries device and returns device capability structure

API Function	Description
ni_logan_device_session_open	Opens a new device session depending on the device_type parameter
ni_logan_device_session_close	Closes device session that was previously opened by calling ni_logan_device_session_open
ni_logan_device_session_flush	Sends a flush command to the device
ni_logan_device_dec_session_save_hdrs	Save a stream's headers in a decoder session that can be used later for continuous decoding from the same source. This usually works with ni_logan_device_dec_session_flush.
ni_logan_device_dec_session_flush	Flush a decoder session to get ready to continue decoding. This is different from ni_logan_device_session_flush in that it closes the current decode session and opens a new one for continuous decoding.
ni_logan_device_session_write	Sends data to the device
ni_logan_device_session_read	Reads data from the device
ni_logan_device_session_query	Query session data from the device
ni_logan_frame_buffer_alloc	Allocate preliminary memory for the frame buffer for encoding based on provided parameters. Applicable to YUV420 Planar pixel format only, 8 or 10 bit/pixel.
ni_logan_encoder_frame_buffer_alloc	Allocate memory for the frame buffer for encoding based on provided parameters, taking into account pic line size and extra data, to be sent to encoder for encoding. Applicable to YUV420 Planar pixel format only, 8 or 10 bit/pixel. Cb/Cr size matches that of Y.
ni_logan_decoder_frame_buffer_alloc	Allocate memory for decoder frame buffer based on provided parameters; the memory is retrieved from

API Function	Description
	a memory buffer pool and will be returned to the same buffer pool by ni_logan_decoder_frame_buffer_free.
ni_logan_frame_buffer_free	Free frame buffer that was previously allocated with ni_logan_frame_buffer_alloc or ni_logan_encoder_frame_buffer_alloc
ni_logan_decoder_frame_buffer_free	Free decoder frame buffer that was previously allocated with ni_logan_decoder_frame_buffer_alloc, returning memory to a buffer pool.
ni_logan_decoder_frame_buffer_pool_return_buf	Return a memory buffer to memory buffer pool, for a decoder frame.
ni_logan_packet_buffer_alloc	Allocate memory for the packet buffer based on provided packet size
ni_logan_packet_buffer_free	Free packet buffer that was previously allocated with ni_logan_packet_buffer_alloc
ni_logan_packet_copy	Copy video packet accounting for alignment and padding
ni_logan_encoder_init_default_params	Initialize default encoder parameters
ni_logan_decoder_init_default_params	Initialize default decoder parameters
ni_logan_encoder_params_set_value	Set value referenced by name in encoder parameters structure
ni_logan_encoder_gop_params_set_value	Set GOP parameter value referenced by name in encoder parameters structure
ni_logan_get_num_reorder_of_gop_structure	Get GOP's max number of reorder frames

API Function	**Description**
ni_logan_get_num_ref_frame_of_gop_structure	Get GOP's number of reference frames
ni_logan_frame_new_aux_data	Add a new auxiliary data to a frame for encoding. Auxiliary data is non-video data associated with the frame such as closed captions, HDR metadata, ROI info, etc.
ni_logan_frame_new_aux_data_from_raw_data	Add a new auxiliary data to a frame and copy in the raw data
ni_logan_frame_get_aux_data	Retrieve from the frame auxiliary data of a given type if exists
ni_logan_frame_free_aux_data	If auxiliary data of the given type exists in the frame, free it and remove it from the frame.
ni_logan_frame_wipe_aux_data	Free and remove all auxiliary data from the frame.
ni_log_logan.h	
ni_log_set_level	Set libxcoder_logan library wide log level
ni_log_get_level	Get libxcoder_logan log level
ni_util_logan.h:	
ni_logan_get_hw_yuv420p_dim	Get dimension information of NETINT HW YUV420p frame to be sent to encoder for encoding.
ni_logan_copy_hw_yuv420p	Copy YUV data to NETINT HW YUV420p frame layout to be sent to encoder for encoding.
insert_emulation_prevent_bytes	Insert emulation prevention byte(s) as needed into the data buffer
ni_av_codec_logan.h:	

API Function	**Description**
ni_logan_set_vui	Set SPS VUI part of encoded stream header
ni_logan_should_send_sei_with_frame	Whether SEI should be sent together with this frame to encoder
ni_logan_dec_retrieve_aux_data	Retrieve auxiliary data (close caption, various SEI) associated with this frame that is returned by decoder, convert them to appropriate format and save them in the frame's auxiliary data storage for future use by encoding.
ni_logan_enc_prep_aux_data	Prepare auxiliary data that should be sent together with this frame to encoder based on the auxiliary data of the decoded frame.
ni_logan_enc_copy_aux_data	Copy auxiliary data that should be sent together with this frame to encoder

6 libxcoder_logan Integration

A sample utility program ***xcoder_logan*** is provided to demonstrate the libxcoder_logan API integration for video decoding/encoding/transcoding tasks. Its source file is located at **libxcoder_logan/source/ni_device_test_logan.c**. In the following sections, we use its source code as examples for writing your own application for video transcoding tasks, using the libxcoder_logan API.

Note that the ***xcoder_logan*** program is limited in its capability compared to full function applications such as FFmpeg in that it handles elementary video stream only. There is no support for containers, audio, or any transport protocols.

The libxcoder_logan decoder requires complete video frames to operate correctly which include all slices, headers, and any other associated NALs such as SEIs, delimiters, etc. For this reason, the ***xcoder_logan*** program provides frame which is currently limited to H.264 only.

6.1 List of Currently Unsupported *xcoder_logan* Features

- Audio
- Container, demuxer, muxer
- H.265 frame parsing
- Custom user SEI
- Encoder sequence change
- Decoder/encoder engine reset/recovery
- Dolby Vision

6.2 *xcoder_logan* Utility Program Command Line Options

```
Video decoder/encoder/transcoder application directly using Netint Libxcoder
API
Usage: xcoder_logan [options]

options:
-h | --help                Show help.
-l | --loglevel            Set loglevel of libxcoder API.
                           [none, fatal, error, info, debug, trace]
                           (Default: info)
-c | --card                Set card index to use.
                           See `ni_rsrc_mon` for cards on system.
                           (Default: 0)
-i | --input               Input file path.
-s | --size                Resolution of input file in format WIDTHxHEIGHT.
                           (eg. '1920x1080')
-m | --mode                Input to output codec processing mode in format:
                           INTYPE2OUTTYPE. [a2y, h2y, y2a, y2h, a2a, a2h, h2a,
h2h]
```

```
                            Type notation: y=YUV 420 Planar a=AVC, h=HEVC
-b | --bitdepth             Input and output bit depth. [8, 10]
                            (Default: 8)
-x | --xcoder-params        Encoding params. See "Encoding Parameters" chapter in
                            IntegrationProgrammingGuideT408_T432_FW*.pdf for help.
                            (Default: "")
-o | --output               Output file path.
```

6.3 Transcoder Resource Allocation

To initialize transcoder resources, run the resource pool init program, ***init_rsrc_logan,*** or resource monitor program, ***ni_rsrc_mon_logan,*** before executing any transcoding tasks.

The xcoder_logan program operates on a single NETINT transcoder device specified by the command line option **-c**, which is the same as the card index shown in the ***ni_rsrc_mon_logan*** output.

For each decode or encode session, a specific session context has to be set up initially:

```
ni_logan_session_context_t dec_ctx = {0};
ni_logan_session_context_t enc_ctx = {0};

dec_ctx.keep_alive_timeout = enc_ctx.keep_alive_timeout = 10;

ni_logan_device_session_context_init(&dec_ctx);
ni_logan_device_session_context_init(&enc_ctx);
```

The session context's **keep_alive_timeout** is a heart beat value in seconds keeping the channel open between the libxcoder_logan and T4xx firmware. This value is sent to T4xx firmware at session opening. If libxcoder_logan stops responding for more that the keep_alive_timeout value, the T4xx firmware will assume that the application is gone and close the session.

We then use the transcoder card index and set it together with the source codec format in the session context to indicate which transcoder to be used for the video coding task. The reservation and allocation of the video resource needed will be done automatically by libxcoder_logan at session opening. This is shown in the following code snippet for decoder (similar for encoder):

```
dec_ctx.codec_format = src_codec_format;
dec_ctx.hw_id = iXcoderGUID;
```

We use simple transcoder selection in our example here where the transcoder GUID (card index) is specified directly. Section 10.3 of the Integration & Programming Guide[1], describes other resource allocation methods such as least loaded card, etc.

Several attributes of the session context such as source bit depth (8 or 10 bit), endianness (little or big), must be specified before the session can be formally opened (sample code below is for decoder):

Note 1: For now, only 8 or 10 bit little endian YUV420 planar pixel format is supported by the ***xcoder_logan*** program.

```
    // default: little endian
    dec_ctx.src_bit_depth = bit_depth;
    dec_ctx.src_endian = NI_LOGAN_FRAME_LITTLE_ENDIAN;
    dec_ctx.bit_depth_factor = 1;
    if (10 == dec_ctx.src_bit_depth)
    {
      dec_ctx.bit_depth_factor = 2;
    }

    err = ni_logan_device_session_open(&dec_ctx, NI_LOGAN_DEVICE_TYPE_DECODER);
```

Note 2 : The session context also needs to have its **p_session_config** attribute set to appropriate values. For the decoder, the values are not currently used. For the encoder, a number of values related to the encoding configuration need to be set. The encoders parameters are described in detail in section 6.5 of the Integration and Programming Guide[1]. The encoding section 6.5 gives some detailed examples related to resolution and conformance window setting.

At this point, we have the decode/encode session open and can start video decoding/encoding operations.

6.4 Decoding

6.4.1 Decoding Loop

The decoding process involves sending encoded bitstream packets to the decoder, and receiving decoded YUV frames from the decoder. The NETINT decoder engine may need to buffer several bitstream packets before producing decoded frames depending on the stream's GOP structure (i.e. when frames are encoded out of sequence). The simple and effective way of decoding is to alternate between sending and receiving in a loop that runs until all the bitstream packets have been sent and all the decoded frames have been decoded and returned. The flow chart is as follows:

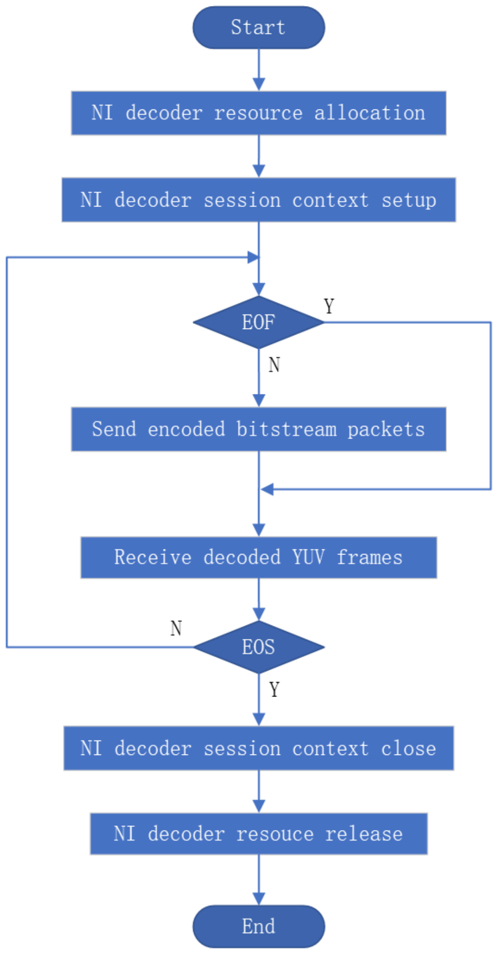

Start
NI decoder resource allocation
NI decoder session context setup
EOF
Y
N
Send encoded bitstream packets
Receive decoded YUV frames
N
EOS
Y
NI decoder session context close
NI decoder resouce release
End

Figure 3 NI Decoding Flow Chart

```
ni_logan_encoder_params_t api_param;
ni_logan_session_context_t dec_ctx;
ni_logan_session_data_io_t in_pkt = {0};
ni_logan_session_data_io_t out_frame = {0};

// Init session context
ni_logan_device_session_context_init(&dec_ctx);

// set up decoder parameters
ret = ni_logan_decoder_init_default_params(&api_param, 25, 1, 200000,
                                     video_width, video_height) < 0);
if (ret < 0) {
   fprintf(stderr, "Error: decoder p_config set up error\n");
   return -1;
```

```
  }

  ret = ni_logan_device_session_open(&dec_ctx,NI_LOGAN_DEVICE_TYPE_DECODER);
  if (ret < 0) {
    fprintf(stderr, "Error: ni_logan_decoder_session_open() failure!\n");
    return -1;
  }

  int eos_flag = 0;
  while (!eos_flag) {
     ni_logan_packet_t *pkt = &in_pkt.data.packet;
     ni_logan_frame_t * frame = &out_frame.data.frame;

     nal_size = read_nalu_from_file(fp, src_buf);  // A custom function
     if (nal_size == 0) {
        break;  // EOF
     }

     ni_logan_packet_buffer_alloc(pkt, nal_size);
     pkt->start_of_stream = sos_flag;

     ni_logan_packet_copy(pkt->p_data, src_buf, nal_size, dec_ctx.p_leftover, &dec_ctx.prev_size);

     // Sending
     send_size = ni_logan_device_session_write(&dec_ctx, &in_pkt,
                                    NI_LOGAN_DEVICE_TYPE_DECODER);
     if (send_size < 0) {
        fprintf(stderr, "Error: sending data error. rc:%d\n", send_size);
        ret = NI_LOGAN_RETCODE_FAILURE;
        break;
     } else if (send_size == 0) {
        // 0 byte sent this time, sleep and will re-try.
        ni_logan_usleep(10000);  // continue
     } else {
        ni_logan_packet_buffer_free(pkt);
     }

     // Receiving decoded YUV frame
     int alloc_mem = dec_ctx.active_video_width > 0 &&
                 dec_ctx.active_video_height > 0 ? 1 : 0;
     ret = ni_logan_decoder_frame_buffer_alloc(dec_ctx.dec_fme_buf_pool,
&p_out_data->data.frame, alloc_mem,
                 video_width, video_height,
                 dec_ctx.codec_format == NI_LOGAN_CODEC_FORMAT_H264,
                 dec_ctx.bit_depth_factor);
     if (ret < 0) {
        break;
     }
    recv_size = ni_logan_device_session_read(&dec_ctx, p_out_data,
                                    NI_LOGAN_DEVICE_TYPE_DECODER);
    if (recv_size < 0) {
      fprintf(stderr, "Error: receiving data error. rc:%d\n", rx_size);
  ni_logan_decoder_frame_buffer_free(&(p_out_data->data.frame));
      ret =  NI_LOGAN_RETCODE_FAILURE;
      break;
    } else if (rx_size == 0) {
      ret = 2;  // EAGAIN
      break;
    } else {
      fprintf(stderr, "Got frame # %"PRIu64" bytes %d\n", dec_ctx.frame_num, recv_size);
      ni_logan_dec_retrieve_aux_data(frame);
    }

    eos_flag = frame->end_of_stream;
    ni_logan_decoder_frame_buffer_free(frame);
  }

  ni_logan_device_session_close(&dec_ctx, 1, NI_LOGAN_DEVICE_TYPE_DECODER);
  ni_logan_rsrc_free_device_context(sdPara.p_dec_rsrc_ctx);
```

Note the code snippet is stripped from *xcoder_logan* demo and it cannot be built separately. The resource management involves allocation and release. The **ni_logan_decoder_init_default_params** API function sets the default values of **ni_logan_encoder_params_t** struct and **ni_logan_device_session_context_init** would initialize the attributes of **ni_logan_device_session_context_t** struct.

The decoder session context setup would call **ni_logan_decoder_session_open** API function after some attributes of **ni_logan_device_session_context_t** struct are specified. It would retrieve the device handle to manage the decoder device. And the **ni_logan_device_session_context_close** is used to release it.

It comes to the decoding main loop. The first step is to read the packet data chunk from the input that the data length would be calculated according to the resolution and bit depth.

The packet data chunk is held by the **ni_logan_packet_t** struct. It uses **ni_logan_packet_buffer_alloc** and **ni_logan_packet_copy** to create and prepare the content of **ni_logan_packet_t** struct, and **ni_logan_packet_buffer_free** to release the **ni_logan_packet_t** struct resource when the sending is successful.

xcoder_logan implements a memory buffer pool for storing YUV raw data retrieved from the decoder. This mechanism reduces frequent memory allocation/release and speeds up processing. And it is the recommended way for decoder YUV retrieval. *xcoder_logan* API functions **ni_logan_decoder_frame_buffer_alloc** and **ni_logan_decoder_frame_buffer_free** are provided for this purpose. Remember to call **ni_logan_dec_retrieve_aux_data** to retrieve auxiliary data (close caption, various SEI) associated with this frame that is returned by the decoder. Section 6.4.6 will introduce more detail information of it.

The bitstream packet sending involves calling *xcoder_logan* API function **ni_logan_device_session_write**, while the YUV frame receiving **ni_logan_device_session_read**. Check the *xcoder_logan* program source code for details on preparing the data sending and receiving, and the ways for the application to notify the decoder of start of stream (sos) and to get notified by the decoder of the end of stream (eos) using attributes in **ni_logan_packet_t** and **ni_logan_frame_t** struct.

6.4.2 Decoding Input

The NETINT decoder engine requires the bitstream to be passed in complete frames so libxcoder_logan expects the same for its input. The ***xcoder_logan*** program provides a very simple H.264 parser to do the frame partitioning. Currently **xcoder_logan** does not provide a parser for H.265.

6.4.3 NETINT YUV420 Format

The NETINT T4xx card uses a YUV420 planar video format in 8 or 10 bit for decoder input and encoder output. Note that 10 bit video uses 16 bit words in little endian format with the 6 most significant bits as don't care.

Figure 3 shows the NETINT 8 bit YUV format. The Y pixels are stored first followed by the U and then the V both subsampled by 2 in each direction. Each pixel is 8 bits.

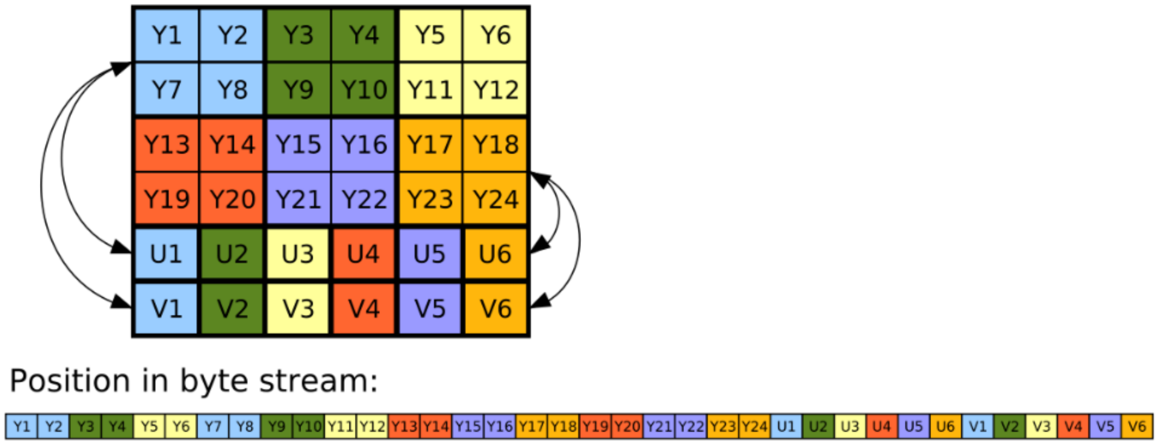


Figure 3 NETINT YUV420 8 bit Format

Figure 4 shows the NETINT 10 bit YUV format. The Y pixels are stored first followed by the U and then the V both subsampled by 2 in each direction. 10 bit pixels are stored as 16 bits as shown in Figure 5.

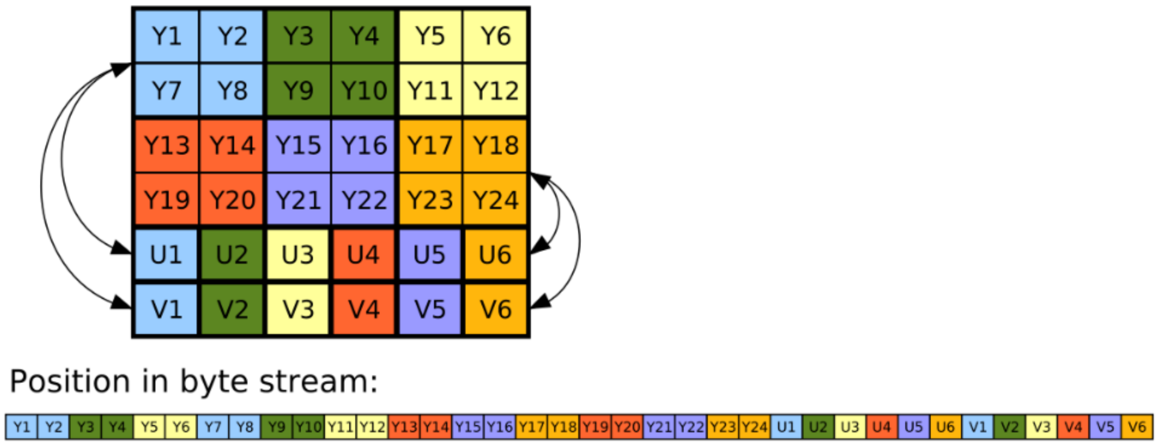


Figure 4 NETINT YUV420 10 bit Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved, should be 0.						Color Component Storage[9:0]									

Figure 5 NETINT 10 bit Pixel Format

6.4.4 YUV420 Alignment Requirements

The NETINT hardware places certain restrictions on the height and width of the YUV data. The picture width (in pixels) must be multiples of 32, and the height must be multiples of 8 for H.265, and 16 for H.264. Picture sizes that do not meet these requirements must be padded before encoding. Decoded YUV frames will be output aligned and may need to be cropped to the original picture size.

Figure 6 shows how the padding must be added to meet the alignment requirements. For example, to encode a 1080p (1920x1080) stream to H.264, the height of the YUV picture must be padded to 1088 to be divisible by 16. Another example, to encode a 720x480 stream to H.265, the width of the picture must be padded to 736 to be divisible by 32.

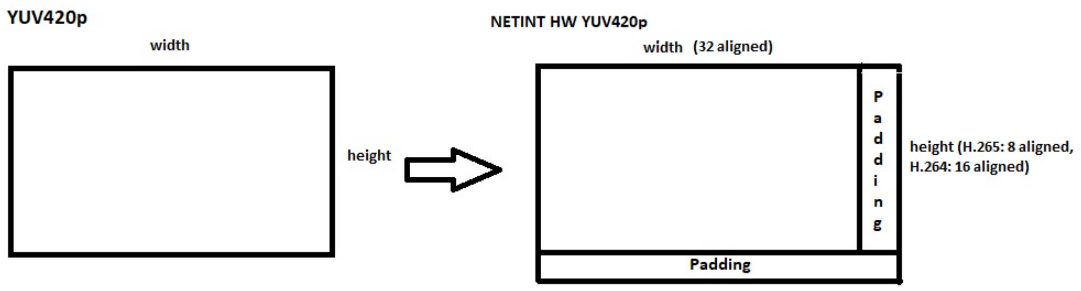


Figure 6 NETINT YUV420 Padding

To meet the alignment requirements, encoder input may need to be padded and decoder output may need to be cropped. libxcoder_logan provides a data structure, attributes and API functions for the frame operations as follows:

```
typedef struct _ni_logan_frame
{
  void * p_data[NI_MAX_NUM_DATA_POINTERS];
  uint32_t data_len[NI_MAX_NUM_DATA_POINTERS];
  uint32_t video_width;
  uint32_t video_height;
  uint32_t crop_top;
  uint32_t crop_bottom;
  uint32_t crop_left;
  uint32_t crop_right;
…
} ni_logan_frame_t ;
```

In this video frame data structure, **p_data** has the pointers to the data of picture planes, **data_len** specifies each plane size in bytes. As illustrated in Figure 7, **video_width** and **video_height** is the padded picture size , **crop_*** specifies a cropping window which is used to when decoding to crop the padded picture to original picture size (rectangle area in black in Figure 7):

- **crop_left** the horizontal pixel offset of top-left corner of rectangle from (0, 0),
- **crop_top** the vertical pixel offset of top-left corner of rectangle from (0, 0),
- **crop_right** the horizontal pixel offset of bottom-right corner of rectangle from (0, 0) and
- **crop_bottom** the vertical pixel offset of bottom-right corner of rectangle from (0, 0).

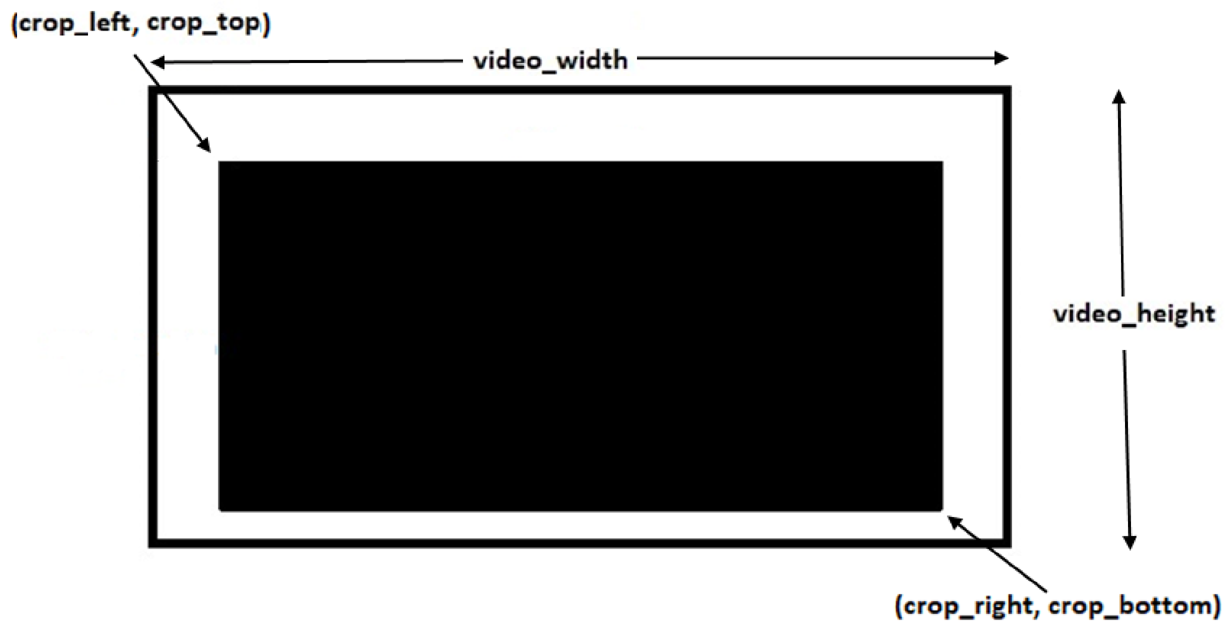


Figure 7 NETINT HW YUV420 Decoded Frame and Cropping Window

For example, when decoding an H.264 1080p bitstream video width and height would be 1920 x 1088, with **crop_left** = 0, **crop_right** = 1920, **crop_top** = 0 and **crop_bottom** = 1080.

6.4.5 NETINT Decoder Output Data Layout

In addition to the YUV data, the decoder returns other information associated with the frame such as frame type, cropping info, frame offset, and any SEI's associated with the frame such as closed captions, HDR colour information etc. Figure 8 shows the layout of the decoder output data.

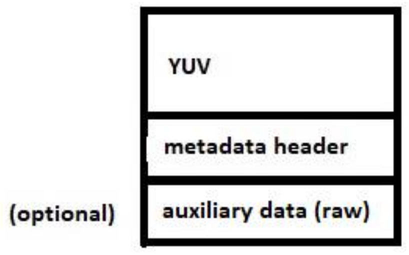


Figure 8 Decoder output layout

YUV and metadata header are mandatory. Metadata header has the information described in the following struct:

```
typedef struct _ni_logan_metadata_common
{
  uint16_t crop_left;
  uint16_t crop_right;
  uint16_t crop_top;
  uint16_t crop_bottom;
  union
  {
    uint64_t frame_offset;
    uint64_t frame_tstamp;
  } ui64_data;
  uint16_t frame_width;
  uint16_t frame_height;
  uint16_t frame_type;
  uint16_t reserved;
} ni_logan_metadata_common_t;

typedef struct _ni_logan_metadata_dec_frame
{
  ni_logan_metadata_common_t metadata_common;
  uint32_t                sei_header;
  uint16_t                sei_number;
  uint16_t                sei_size;
  uint32_t                frame_cycle;
  uint32_t                reserved;
} ni_logan_metadata_dec_frame_t;
```

The cropping and resolution information is used to fill in ni_logan_frame_t attributes presented in 6.4.3. Other information (shown as auxiliary data) from supported SEIs, if present, is then used to fill in the remaining attributes of ni_logan_frame_t as follows:

```
typedef struct _ni_logan_frame
{
 … …
  // SEI info of closed caption: returned by decoder or set by encoder
  unsigned int sei_cc_offset;
  unsigned int sei_cc_len;
  // SEI info of HDR: returned by decoder
  unsigned int sei_hdr_mastering_display_color_vol_offset;
  unsigned int sei_hdr_mastering_display_color_vol_len;
  unsigned int sei_hdr_content_light_level_info_offset;
  unsigned int sei_hdr_content_light_level_info_len;
  // SEI info of HDR10+: returned by decoder
  unsigned int sei_hdr_plus_offset;
  unsigned int sei_hdr_plus_len;
  // SEI info of User Data Unregistered SEI: returned by decoder
  unsigned int sei_user_data_unreg_offset;
  unsigned int sei_user_data_unreg_len;
} ni_logan_frame_t;
```

Each pair of attributes specify the location of the respective SEI raw data: offset from the start, and size of the data. This information will be used to extract and convert the SEI data into a more useable format by libxcoder_logan. The supported SEIs include: T35 closed captions, HDR static metadata (mastering display color volume, content light level), T35 HDR10+ dynamic metadata, and user data unregistered.

Closed captioned data is formatted as the T35 SEI payload data specified by CEA708[15]. The user data unregistered data includes the UUID and is formatted as per the H.264[13] and H.265[14] standards.

The static HDR metadata is defined in the following structures: ni_logan_dec_mastering_display_colour_volume_t, ni_content_light_level_info_t. The HDR10+ dynamic metadata is defined by the following structure: ni_dynamic_hdr_plus_t.

These SEIs, together with other information such as ROI (Region of Interest) and reconfiguration request (used for encoding) are called **auxiliary data**.

The following sections give more detail on the auxiliary data and its usage, and in the encoding section we see that the auxiliary data is passed to the encoder for insertion into the encoded stream.

6.4.6 Auxiliary data

Auxiliary data is information that accompanies a video frame. It is returned by decoder as a result of the decoding process, e.g. a closed caption that comes with a decoded video frame, or it can be supplied by the user to be included in the encoded bitstream or for the case of ROI to permit frame by frame control of the encoding quality. The following auxiliary data types have been defined:

```
typedef enum _ni_frame_aux_data_type
{
  NI_FRAME_AUX_DATA_NONE   = 0,

  // ATSC A53 Part 4 Closed Captions
  NI_FRAME_AUX_DATA_A53_CC,

  // HDR10 mastering display metadata associated with a video frame
  NI_FRAME_AUX_DATA_MASTERING_DISPLAY_METADATA,

  // HDR10 content light level (based on CTA-861.3). This payload contains
  // data in the form of ni_content_light_metadata_t struct.
  NI_FRAME_AUX_DATA_CONTENT_LIGHT_LEVEL,

  // HDR10+ dynamic metadata associated with a video frame. The payload is
  // a ni_dynamic_hdr_plus_t struct that contains information for color volume
  // transform - application 4 of SMPTE 2094-40:2016 standard.
  NI_FRAME_AUX_DATA_HDR_PLUS,

  // Regions of Interest, the payload is an array of ni_region_of_interest_t,
  // the number of array element is implied by:
  // ni_frame_aux_data.size / sizeof(ni_region_of_interest_t)
  NI_FRAME_AUX_DATA_REGIONS_OF_INTEREST,

  // NETINT: user data unregistered SEI data, which takes SEI payload type
  // USER_DATA_UNREGISTERED.
  // There will be no byte reordering.
  // Usually this payload would be: 16B UUID + other payload Bytes.
  NI_FRAME_AUX_DATA_UDU_SEI,

  // NETINT: custom SEI data, which takes SEI payload custom types.
  // There will be no byte reordering.
  // Usually this payload would be: 1B Custom SEI type + 16B UUID + other
  // payload Bytes.
  NI_FRAME_AUX_DATA_CUSTOM_SEI,

  // NETINT: custom bitrate adjustment, which takes int32_t type data as
  // payload that indicates the new target bitrate value.
  NI_FRAME_AUX_DATA_BITRATE,

  // NETINT: long term reference frame support, which is a struct of
  // ni_long_term_ref_t that specifies a frame's support of long term
  // reference frame.
  NI_FRAME_AUX_DATA_LONG_TERM_REF,

} ni_aux_data_type_t;
```

Note that the T408 firmware currently supports decoding a maximum of 1024 bytes of SEI payload.

6.4.6.1 Retrieval and Storage

The auxiliary data returned by decoder is in a low level format and can be converted into a more useable format by calling the **ni_logan_dec_retrieve_aux_data** API function. The new format can now be used when transcoding to pass on closed captions and HDR SEIs to the encoder.

Libxcoder_logan converts auxiliary data into an easy-to-use format and stores them in the frame, by the following struct:

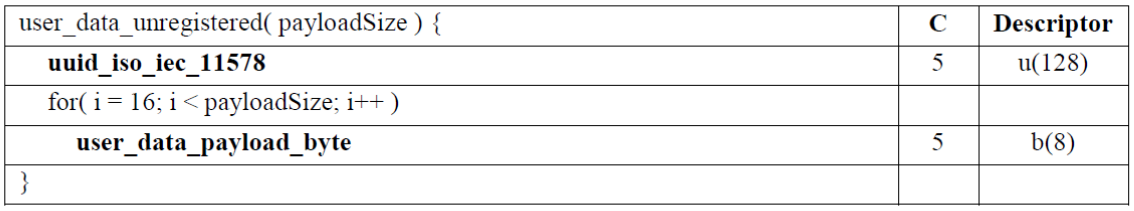
```
// struct to hold auxiliary data for ni_logan_frame_t
typedef struct _ni_aux_data
{
  ni_aux_data_type_t type;
  uint8_t *data;
  int       size;
} ni_aux_data_t;
```

```
typedef struct _ni_logan_frame
{
  // frame auxiliary data
  ni_aux_data_t *aux_data[NI_MAX_NUM_AUX_DATA_PER_FRAME];
  int             nb_aux_data;
}
```

The data attribute in ni_aux_data_t is either a struct defined for that info (e.g. ni_dynamic_hdr_plus_t for HDR10+), or a number of payload bytes (e.g. A53 closed captions or user data unregistered data). These are further detailed in the following sections.

6.4.6.2 User Data Unregistered SEI

The User Data Unregistered SEI, when available and retrieved, is stored as a number of payload bytes directly in the ni_aux_data_t. This payload includes 16 bytes of UUID plus other payload bytes as defined in the H.264[13] and H.265[14] standards and shown in Figure 9.



user_data_unregistered(payloadSize) {	C	Descriptor
uuid_iso_iec_11578	5	u(128)
for(i = 16; i < payloadSize; i++)		
user_data_payload_byte	5	b(8)
}		

Figure 9 User Data Unregistered SEI data format

6.4.6.3 Closed caption

The closed caption data, when available, is stored as a number of payload bytes directly in the ni_aux_data_t. The format is the T35 payload as specified by CEA-708[15] as shown in Figure 10 and Figure 11.

Length (bits)	Name	Type	Value
8	itu_t_t35_country_code	uimsbf	181
16	itu_t_t35_provider_code	uimsbf	49 or 47
32	ATSC_user_identifier (only if provider is 49)	ASCII bslbf	GA94
8	ATSC1_data_user_data_type_code (only if provider is 47 or 49)	uimsbf	3

8	DIRECTV_user_data_length (only if provider is 47)	uimsbf	variable
X*8	user_data_type_structure	binary	free form

Figure 10 CEA-708 T35 Payload Format

Length (bits)	Name	Type	Value
1 (b7)	process_em_data_flag	flag	1
1 (b6)	process_cc_data_flag	flag	1
1 (b5)	additional_data_flag	flag	0
5 (b0-b4)	cc_count	uimsbf	Variable
8	em_data (not in CDP data)	uimsbf	256
cc_count*24	cc_data_pkt's	bslbf	free form
8	marker_bits (not in CDP data)	patterned bslbf	256
34	ATSC_reserved_user_data (not in CDP data)	bslbf	free form

Figure 11 CEA-708 user_data_type_structure

6.4.6.4 HDR10 Mastering Display Metadata

The HDR10 mastering display metadata, when available, is stored in the following struct and saved in the ni_aux_data_t:

```
// struct describing HDR10 mastering display metadata
typedef struct _ni_mastering_display_metadata
{
  // CIE 1931 xy chromaticity coords of color primaries (r, g, b order).
  ni_rational_t display_primaries[3][2];

  // CIE 1931 xy chromaticity coords of white point.
  ni_rational_t white_point[2];

  // Min luminance of mastering display (cd/m^2).
  ni_rational_t min_luminance;

  // Max luminance of mastering display (cd/m^2).
  ni_rational_t max_luminance;

  // Flag indicating whether the display primaries (and white point) are set.
  int has_primaries;

  // Flag indicating whether the luminance (min_ and max_) have been set.
  int has_luminance;
} ni_mastering_display_metadata_t;
```

6.4.6.5 HDR10 Content Light Level Info

The HDR10 content light level info, when available, is stored in the following struct and saved in the ni_aux_data_t:

```
// struct describing HDR10 Content light level
typedef struct _ni_content_light_level
{
  // Max content light level (cd/m^2).
  unsigned max_cll;

  // Max average light level per frame (cd/m^2).
  unsigned max_fall;
} ni_content_light_level_t;
```

6.4.6.6 HDR10+ Dynamic Metadata

The HDR10+ Dynamic Metadata, when available, is stored in the following struct and associated sub-struct, and saved in the ni_aux_data_t:

```
// struct representing dynamic metadata for color volume transform -
// application 4 of SMPTE 2094-40:2016 standard.
typedef struct _ni_dynamic_hdr_plus
{
  // Country code by Rec. ITU-T T.35 Annex A. The value shall be 0xB5.
  uint8_t itu_t_t35_country_code;

  //  Application version in the application defining document in ST-2094
  //  suite. The value shall be set to 0.
  uint8_t application_version;

  //  The number of processing windows. The value shall be in the range
  //  of 1 to 3, inclusive.
  uint8_t num_windows;

  //  The color transform parameters for every processing window.
  ni_hdr_plus_color_transform_params_t params[3];

  //  The nominal maximum display luminance of the targeted system display,
  //  in units of 0.0001 candelas per square metre. The value shall be in
  //  the range of 0 to 10000, inclusive.
  ni_rational_t targeted_system_display_maximum_luminance;

  //  This flag shall be equal to 0 in bit streams conforming to this version
  //  of this Specification. The value 1 is reserved for future use.
  uint8_t targeted_system_display_actual_peak_luminance_flag;

  //  The number of rows in the targeted system_display_actual_peak_luminance
  //  array. The value shall be in the range of 2 to 25, inclusive.
  uint8_t num_rows_targeted_system_display_actual_peak_luminance;

  //  The number of columns in the
  //  targeted_system_display_actual_peak_luminance array. The value shall be
  //  in the range of 2 to 25, inclusive.
  uint8_t num_cols_targeted_system_display_actual_peak_luminance;

  //  The normalized actual peak luminance of the targeted system display. The
  //  values should be in the range of 0 to 1, inclusive and in multiples of
  //  1/15.
  ni_rational_t targeted_system_display_actual_peak_luminance[25][25];

  //  This flag shall be equal to 0 in bitstreams conforming to this version of
  //  this Specification. The value 1 is reserved for future use.
  uint8_t mastering_display_actual_peak_luminance_flag;

  //  The number of rows in the mastering_display_actual_peak_luminance array.
  //  The value shall be in the range of 2 to 25, inclusive.
  uint8_t num_rows_mastering_display_actual_peak_luminance;

  //  The number of columns in the mastering_display_actual_peak_luminance
  //  array. The value shall be in the range of 2 to 25, inclusive.
  uint8_t num_cols_mastering_display_actual_peak_luminance;

  //  The normalized actual peak luminance of the mastering display used for
  //  mastering the image essence. The values should be in the range of 0 to 1,
  //  inclusive and in multiples of 1/15.
  ni_rational_t mastering_display_actual_peak_luminance[25][25];
} ni_dynamic_hdr_plus_t;
```

6.5 Encoding

6.5.1 Encoding loop

The opposite of decoding, the encoding process involves sending the YUV frames to the encoder, and receiving the encoded bitstream packets from the encoder. The NETINT encoder engine may buffer several frames before producing encoded packets depending on the stream's GOP structure which may contain out of sequence frames. The simple and effective way of encoding is also to alternate between sending and receiving in a loop that runs until all the YUV frames have been sent and all the encoded bitstream packets have been returned. The flow chart is as follows:

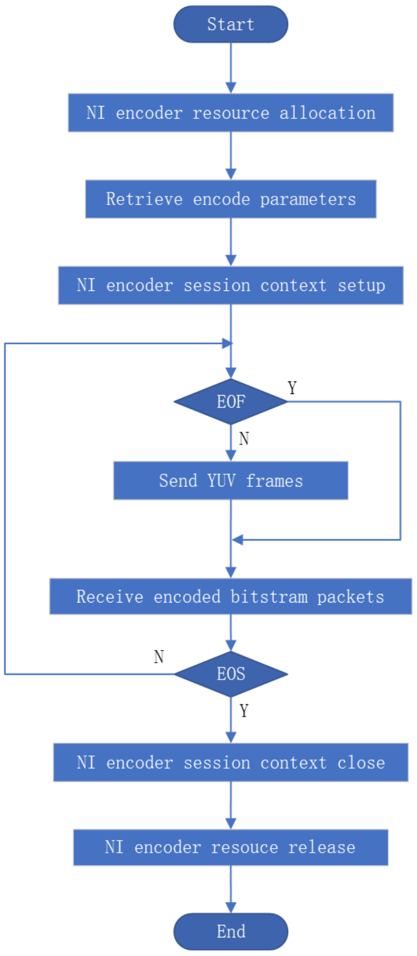

Start
NI encoder resource allocation
Retrieve encode parameters
NI encoder session context setup
EOF
Y
N
Send YUV frames
Receive encoded bitstram packets
N
EOS
Y
NI encoder session context close
NI encoder resouce release
End

Figure 12 NETINT Encoding Flow Chart

```
ni_logan_encoder_params_t api_param;
ni_logan_session_context_t enc_ctx;
  ni_logan_session_data_io_t in_frame = {0};
  ni_logan_session_data_io_t out_packet = {0};

// Init session context
ni_logan_device_session_context_init(&enc_ctx);

// set up decoder parameters
ret = ni_logan_encoder_init_default_params(&api_param, 30, 1, 200000,
                        video_width, video_height) < 0);
if (ret < 0) {
  fprintf(stderr, "Error: encoder parameter set up error\n");
  return -1;
```

```
}

// Parse the command line option and retrieve the parameters.
retrieve_xcoder_params(cmd_str, &api_param, &enc_ctx); // This is a custom function.

// VUI setting including color setting
ni_logan_set_vui(&api_param, color_primaries, color_trc, color_space,
  video_full_range_flag, sar_num, sar_den, dst_codec_format, hrd_params);

ret = ni_logan_device_session_open(&enc_ctx,NI_LOGAN_DEVICE_TYPE_ENCODER);
if (ret < 0) {
 fprintf(stderr, "Error: ni_logan_encoder_session_open() failure!\n");
 return -1;
}

int eos_flag = 0;
   while (!eos_flag)
   {
  ni_logan_frame_t *frame = &in_frame.data.frame;
  ni_logan_packet_t *pkt = &out_pkt.data.packet;

  chunk_size = read_yuv_from_file(fp, yuv_buf); // A custom function
  if (chunk_size == 0) {
    break; // EOF
  }

  frame->start_of_stream = sos_flag; // start of stream
  // calculate strides and linesizes
  ni_logan_get_hw_yuv420p_dim(video_width, video_height,
              enc_ctx.bit_depth_factor,
              enc_ctx.codec_format == NI_LOGAN_CODEC_FORMAT_H264,
              dst_stride, dst_height_aligned);


  ni_logan_encoder_frame_buffer_alloc(frame, video_width, video_height,
     dst_stride, enc_ctx.codec_format == NI_LOGAN_CODEC_FORMAT_H264,
     frame->extra_data_len, enc_ctx.bit_depth_factor);
  if (!frame->p_data[0]) {
    fprintf(stderr, "Error: could not allocate YUV frame buffer!");
    break;
  }

  // Copy YUV buffer into ni_logan_frame_t
  ni_logan_copy_hw_yuv420p((uint8_t **)(frame->p_data), yuv_buf,
          video_width, vvideo_height, enc_ctx.bit_depth_factor,
          dst_stride, dst_height_aligned, src_stride, src_height);

  send_size = ni_logan_device_session_write(&enc_ctx, &in_frame,
                           NI_LOGAN_DEVICE_TYPE_ENCODER);
  if (send_size < 0) {
          fprintf(stderr, "Error: failed ni_logan_device_session_write() for encoder\n");
          break;
       } else if (send_size == 0 && !p_enc_ctx->ready_to_close) {
          need_to_resend = 1;
       } else
          ni_logan_frame_buffer_free(frame);
       }

       // Receiving encoded bitstream packets
       ni_logan_packet_buffer_alloc(pkt, NI_LOGAN_MAX_TX_SZ);

recv_pkt:
       recv_size = ni_logan_device_session_read(&enc_ctx, &out_pkt,
                                NI_LOGAN_DEVICE_TYPE_ENCODER);
       if (recv_size > meta_size) {
          // Normal
       } else if (recv_size != 0) {
          fprintf(stderr, "Error: received %d bytes, <= metadata size %d!\n", recv_size, meta_size);
       } else {
          goto recv_pkt;
       }
       eos_flag = pkt->end_of_stream;
       ni_logan_packet_buffer_free(pkt);
         }

   ni_logan_device_session_close(&enc_ctx, 1, NI_LOGAN_DEVICE_TYPE_ENCODER);
   ni_logan_rsrc_free_device_context(sdPara.p_enc_rsrc_ctx);
```

Note the code snippet is stripped from *xcoder_logan* demo and it cannot be built separately. The resource management and the device session context operation work similarly to the decoder. See section 6.4.1 for details. In paticular **ni_logan_encoder_params_set_value** is used to retrieve the encode parameters specified by the command line option. And it needs to call **ni_logan_set_vui** before **ni_logan_device_session_open** to specify VUI parameters of SPS header.

It uses **ni_logan_frame_t** and **ni_logan_packet_t** struct to hold the input YUV frame and the output encoded packet data. The buffer management is similar to the decoder part of section 6.4.1. Section 6.5.2 introduces the input YUV frame preparation. And **ni_logan_enc_prep_aux_data** and **ni_logan_enc_copy_aux_data** API functions prepare the auxiliary data (various SEI, ROI etc.) if it is needed. See section 6.5.6 for more information.

As well the data sending and receiving involve calling xcoder_logan API function **ni_logan_device_session_write** and **ni_logan_device_session_read**. See the xcoder_loganprogram source code for details on preparing the data for sending and receiving, and the ways for the application to notify the encoder of start of stream (sos) and to get notified of the end of stream (eos) by the encoder using attributes in **ni_logan_frame_t** and **ni_logan_packet_t** struct.

6.5.2 Input YUV Frame Preparation

As described in section 6.4.3, the NETINT encoder accepts only YUV420 planar frame data (8 bit or 10 bit) meeting the alignment requirements, with picture width (in pixels) a multiple of 32, and height a multiple of 8 for H.265, and 16 for H.264. Frames that do not meet this alignment requirements must be padded.

Two libxcoder_logan API functions are provided: **ni_logan_get_hw_yuv420p_dim** and **ni_logan_copy_hw_yuv420p**, in ni_util_logan.h to prepare YUV data for sending to the encoder. They are used to get dimensional information of the NETINT YUV frame for allocating the data buffer, and to copy YUV data to NETINT YUV frame for sending to the encoder. For details, check the ni_util_logan.h header file and *xcoder_logan* program source for their sample usage.

6.5.3 Conformance Window Setting in Encoder Configuration

When input data needs to be padded to meet the alignment requirements, a conformance window can be specified to indicate any cropping that is required after decoding the encoded bitstream to get back to the original picture size. The offset parameters of this conformance window are specified in the following struct definition in ni_device_api_logan.h:

```
typedef struct _ni_logan_h265_encoder_params
{
  …
  // ConformanceWindowOffsets
  int conf_win_top;
  int conf_win_bottom;
  int conf_win_left;
  int conf_win_right;
} ni_logan_h265_encoder_params_t;

typedef struct _ni_logan_encoder_params
{
 …
  ni_logan_h265_encoder_params_t hevc_enc_params;
…
} ni_logan_encoder_params_t;
```

conf_win* are the number of pixels to discard from the top/bottom/left/right border of the frame to obtain the rectangle area intended for presentation. Their calculation, taking into consideration of resolution smaller than the minimum supported by NETINT encoder (NI_LOGAN_MIN_WIDTH x NI_LOGAN_MIN_HEIGHT, i.e. 256 x 128), is implemented in the following code:

```
    ni_logan_encoder_params_t  api_param;
    ni_logan_encoder_init_default_params(&api_param, …);
    enc_ctx.p_session_config = &api_param;
    int linesize_aligned = ((arg_width + 7) / 8) * 8;
    if (enc_ctx.codec_format == NI_LOGAN_CODEC_FORMAT_H264)
    {
      linesize_aligned = ((arg_width + 15) / 16) * 16;
    }
    if (linesize_aligned < NI_LOGAN_MIN_WIDTH)
    {
      api_param.hevc_enc_params.conf_win_right += NI_LOGAN_MIN_WIDTH -
arg_width;
      linesize_aligned = NI_LOGAN_MIN_WIDTH;
    }
    else if (linesize_aligned > arg_width)
    {
      api_param.hevc_enc_params.conf_win_right += linesize_aligned - arg_width;
    }
    api_param.source_width = linesize_aligned;

    int height_aligned = ((arg_height + 7) / 8) * 8;
    if (enc_ctx.codec_format == NI_LOGAN_CODEC_FORMAT_H264)
    {
      height_aligned = ((arg_height + 15) / 16) * 16;
    }
    if (height_aligned < NI_LOGAN_MIN_HEIGHT)
    {
      api_param.hevc_enc_params.conf_win_bottom += NI_LOGAN_MIN_HEIGHT -
arg_height;
      height_aligned = NI_LOGAN_MIN_HEIGHT;
    }
    else if (height_aligned > arg_height)
    {
      api_param.hevc_enc_params.conf_win_bottom += height_aligned - arg_height;
    }
    api_param.source_height = height_aligned;
```

For example, to encode to an H.264 1080p stream, the encoding configuration, api_param.source_width is set to 1920, api_param.source_height 1088 (1080 adjusted to be 1088 = 16 x 68), conf_win_top = conf_win_left = conf_win_right = 0, conf_win_bottom = 8.

6.5.4 Encoding Formats and Parameters

The encoding formats and a number of encoding parameters provided by the NETINT encoder have been explained in detail in T4xx Integration & Programming Guide[1], sections 6.4 and 6.5 respectively.

As described in section 6.1, the *xcoder_logan* supports the encoding parameters by using a colon separated list of key/value pairs, like the following:

sudo xcoder_logan -c 0 -i input.264 -o output.265 -s 1920x1080 -m a2h -x
frameRate=25:gopPresetIdx=5:intraPeriod=160:RcEnable=1:RcInitDelay=3000:bitrate=2000000

Sections 6.5.4.1 and 6.5.4.2 give a couple of examples on how to use these parameters.

6.5.4.1 Color Metrics Forcing

By default, the color metrics in the VUI section of the SPS of the encoded stream header are set to 2,2,2 (unspecified) for color primaries, color transfer characteristics and color space respectively. The VUI colour metrics can be set to other values as required for example for HDR10 where colour primaries is set to ITU-R BT2020 (9), color transfer characteristics set to SMPTE ST 2084 (16), and color space set to ITU-R BT2020 (9) non-constant luminance system for encoding to a 4K HDR10+ H.265 stream, the following command options are used:

sudo xcoder_logan -c 0 -i 4kHDR10+.264 -o 4k.xcoder.265 -s 3840x2160 -m a2h -b 10 -x
colorPri=9:colorTrc=16:colorSpc=9

The supported values of color metrics are listed in the enum types of **ni_color_primaries_t**, **ni_color_transfer_characteristic_t** and **ni_color_space_t** respectively in header file ni_av_codec_logan.h. Please see the H.265[14] and H.264[13] standards for more details on the VUI.

6.5.4.2 Low Delay Encoding

The encoder can be configured to function in a low latency mode, such that libxcoder_logan only sends a single frame to the encoder at a time and does not send the next frame until it receives an encoded frame. For details see APPS012[7] low latency mode application note.

To enable this feature, a low delay GOP (where all frames are in sequence) such as gopPresetIdx=2 must be used when lowDelay is enabled, as shown in the following command:

sudo xcoder_logan -c 0 -i input.264 -o output.265 -s 1920x1080 -m a2h -x
gopPresetIdx=2:lowDelay=1*:RcEnable=1:bitrate=4000000*

Since the first packet returned by the encoder contains only the bitstreams header, the application must initially read 2 packets to retrieve the first encoded frame as shown in the following ***xcoder_logan*** code:

```
receive_data:  rc = ni_logan_device_session_read(p_enc_ctx, p_out_data,
                                              NI_LOGAN_DEVICE_TYPE_ENCODER);

  end_flag = p_out_pkt->end_of_stream;
  rx_size = rc;

  ni_log(NI_LOG_TRACE, "encoder_receive_data: received data size=%d\n",
rx_size);

  if (rx_size > meta_size)
  {
    if (p_file && (fwrite((uint8_t*)p_out_pkt->p_data + meta_size,
                          p_out_pkt->data_len - meta_size, 1, p_file) != 1))
    {
      fprintf(stderr, "Writing data %d bytes error !\n",
              p_out_pkt->data_len - meta_size);
      fprintf(stderr, "ferror rc = %d\n", ferror(p_file));
    }

    *total_bytes_received += rx_size - meta_size;

    if (0 == p_enc_ctx->pkt_num)
    {
      p_enc_ctx->pkt_num = 1;
      ni_log(NI_LOG_TRACE, "got encoded stream header, keep reading ..\n");
      goto receive_data;
    }
    number_of_packets++;
```

6.5.5 Encoder Sequence Change

A sequence change is a change in picture size or bit depth of the input YUV while encoding. To support this with the NETINT encoder, the application must close the current encoding session and open a new one. The xcoder_logan program does not currently support this.

6.5.6 NETINT Encoder Input Data Layout

As with the decoder, the encoded YUV frame may have auxiliary data as described in section 6.4.3. The layout of the encoder input data is show in Figure 13.

The metadata header is defined by the following structure. Its attributes specify the sizes of reconfig, ROI and SEI if such data is present. The actual reconfig, ROI, and SEI data if present for this frame, follows the metadata header.

```
typedef struct _ni_logan_metadata_enc_frame
{
  ni_logan_metadata_common_t  metadata_common;
  uint32_t                    frame_roi_avg_qp;
  uint32_t                    frame_roi_map_size;
  uint32_t                    frame_sei_data_size;
  uint32_t                    enc_reconfig_data_size;
  uint16_t                    frame_force_type_enable;
  uint16_t                    frame_force_type;
  uint16_t                    force_pic_qp_enable;
  uint16_t                    force_pic_qp_i;
  uint16_t                    force_pic_qp_p;
  uint16_t                    force_pic_qp_b;
  uint16_t                    force_headers;
  uint8_t                     use_cur_src_as_long_term_pic;
  uint8_t                     use_long_term_ref;
} ni_logan_metadata_enc_frame_t;
```

Note that space for reconfig (data for reconfiguring stream properties at run time) is reserved even if reconfiguration is not enabled, but ROI and/or SEI information is available.

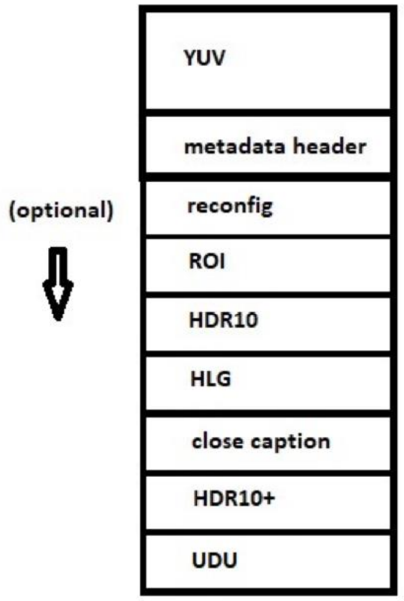


Figure 13 Encoder input layout

The libxcoder_logan API provides a couple of functions to assist in preparing the encoder data, **ni_logan_enc_prep_aux_data** and **ni_logan_enc_copy_aux_data.** These functions convert and copy the auxiliary data into the format required by the encoder.

The retrieval and storage of auxiliary data returned by the decoder is described in section 6.4.6.

All SEI data (User data unregistered, HDR10, HDR10+, close caption, HLG preferred characteristics) must be a in the NAL unit format for the configured codec as per the codec standard using the correct SEI specific RBSP syntax, including start and end codes, NAL header with proper start code emulation applied. This is done by libxcoder_logan. Check the syntax definition in the H.264[13] and H.265[14] standards for more detail. The construction of these NALs is hidden from the libxcoder_logan API user and is implemented in the functions **ni_logan_enc_prep_aux_data** and **ni_logan_enc_copy_aux_data**.

Note that the T408 firmware currently supports a maximum of 1024 bytes of SEI payload.

The following sections detail frame type forcing, the ROI, reconfiguration and various other auxiliary data and their usage.

6.5.6.1 Frame Type Forcing

Frame type forcing is generally used for forcing IDR frames as needed for scene changes, commercial substitution, etc. An IDR may be forced at any point during encoding. When an IDR frame is forced mid-GOP any encoded but unsent frames are flushed and a new GOP is started.

P frames may also be forced, but only if the frame type of the GOP is a B frames. An I-frame may not be forced as a P frame.

This feature is not directly supported by the **xcoder_logan** program. An example of frame type forcing used in the NETINT libavcodec code is as follows :

```
    ctx->api_fme.data.frame.ni_logan_pict_type = 0;
    if (ctx->api_ctx.force_frame_type)
    {
      switch (ctx->buffered_fme->pict_type)
      {
      case AV_PICTURE_TYPE_I:
        ctx->api_fme.data.frame.ni_logan_pict_type = LOGAN_PIC_TYPE_IDR;
        break;
      case AV_PICTURE_TYPE_P:
        ctx->api_fme.data.frame.ni_logan_pict_type = LOGAN_PIC_TYPE_P;
        break;
      default:
        ;
      }
    }
    else if (AV_PICTURE_TYPE_I == ctx->buffered_fme->pict_type)
    {
      ctx->api_fme.data.frame.force_key_frame = 1;
      ctx->api_fme.data.frame.ni_logan_pict_type = LOGAN_PIC_TYPE_IDR;
    }
```

In this example when forceFrameType = 1, the forced frame type is set to the source frame type, and when the source frame is I-frame, the forced encode frame is IDR.

The libxcoder_logan code that sets the metadata attributes based on the libavcodec setting is as follows:

```
  p_meta->frame_force_type_enable = p_meta->frame_force_type = 0;
  // frame type to be forced to is supposed to be set correctly
  // in p_frame->ni_logan_pict_type
  if (1 == p_ctx->force_frame_type || p_frame->force_key_frame)
  {
    if (p_frame->ni_logan_pict_type)
    {
      p_meta->frame_force_type_enable = 1;
      p_meta->frame_force_type = p_frame->ni_logan_pict_type;
    }
  }
```

6.5.6.2 ROI

To use the ROI feature, the following steps are required:

- The feature must be enabled by the libxcoder_logan encode option **roiEnable=1**;
- An ROI QP map specifying QP values and other information need to be supplied with each YUV frame. The ROI values can change and ROI can be enabled/disabled on a frame by frame basis.

There are a few ways to provide this QP map as explained in the following sections.

6.5.6.2.1 Custom QP Map and Average QP

This is the lowest level API where the user supplies the details of QP to the encoder. The details of preparing this QP map are described in APPS009-netint_apps_note_ROI[5], section 4.5. The coding details are in libxcoder_logan API function **set_roi_map** in file ni_av_codec_logan.c. This method is the most complicated but permits non-rectangular ROI regions to be defined if required.

6.5.6.2.2 libxcoder_logan ROI Struct

Alternatively, libxcoder_logan provides an ROI struct that permits a list of rectangular ROI regions to be specified:

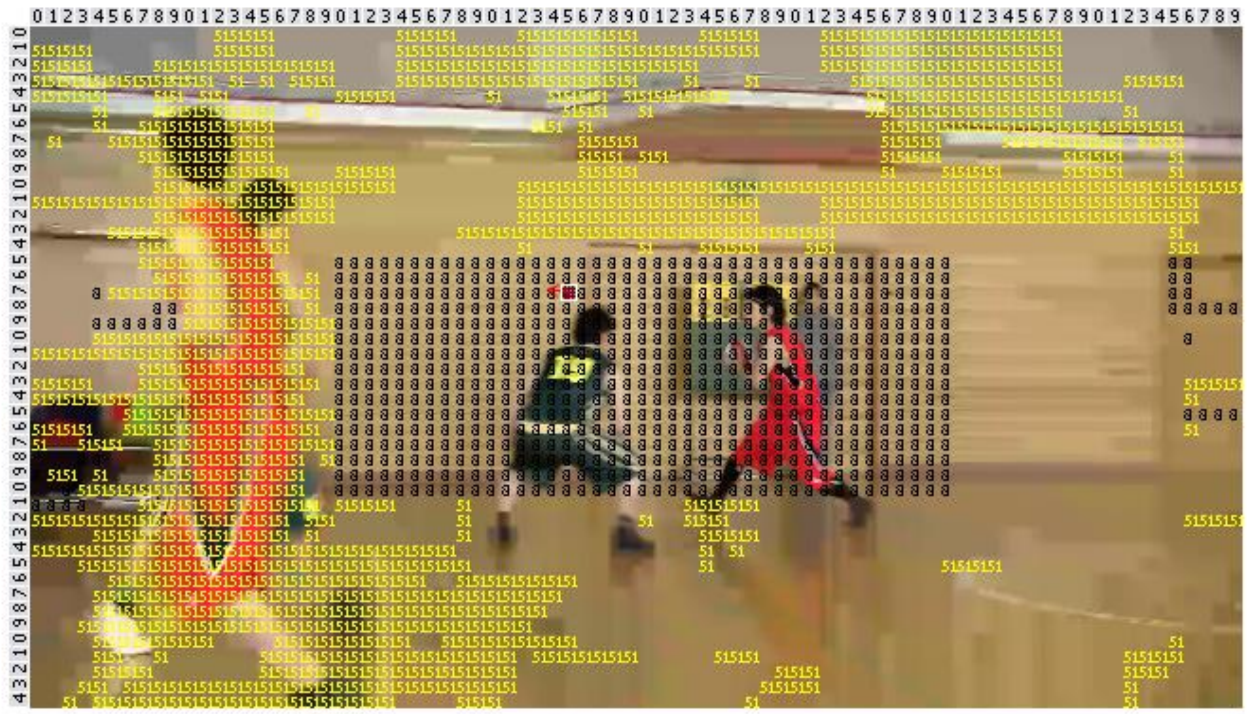
```
// struct describing a Region Of Interest (ROI)
typedef struct _ni_region_of_interest
{
  // self size: must be set to: sizeof(ni_region_of_interest_t)
  uint32_t self_size;

  // ROI rectangle: pixels from the frame's top edge to the top and bottom
edges
  // of the rectangle, from the frame's left edge to the left and right edges
  // of the rectangle.
  int top;
  int bottom;
  int left;
  int right;

  // quantisation offset: [-1, +1], 0 means no quality change; < 0 value asks
  // for better quality (less quantisation), > 0 value asks for worse quality
  // (greater quantisation).
  ni_rational_t qoffset;
} ni_region_of_interest_t;
```

The following code shows an example of applying ROI to a frame: for a 720p frame, the first ROI defined the real region of interest (a window in the middle of image) to be encoded with highest quality (lowest QP), and the second ROI sets the whole image to lowest quality (highest QP).

```
ni_aux_data_t *aux_data = NULL;
ni_region_of_interest_t *my_roi;

aux_data = ni_logan_frame_new_aux_data(frame,
NI_FRAME_AUX_DATA_REGIONS_OF_INTEREST,
                            sizeof(ni_region_of_interest_t) * 2);

my_roi = (ni_region_of_interest_t *)aux_data->data;

my_roi[0].self_size = sizeof(ni_region_of_interest_t);
my_roi[0].top    = 240;
my_roi[0].bottom = 480;
my_roi[0].left   = 320;
my_roi[0].right  = 960;
my_roi[0].qoffset.num = -1;
my_roi[0].qoffset.den = 1;

my_roi[1].self_size = sizeof(ni_region_of_interest_t);
my_roi[1].top    = 0;
my_roi[1].bottom = 720;
my_roi[1].left   = 0;
my_roi[1].right  = 1280;
my_roi[1].qoffset.num = 1;
my_roi[1].qoffset.den = 1;
```

Note that this example is extreme since we are requesting a QP offset of -1 for the region of interest which corresponds to the minimum QP of 0 and a QP offset of +1 for the background which corresponds to the maximum QP of 51. If rate control is disabled, the encoder will use the QP values directly as supplied by the ROI map. If rate control is enables, the QP values will be scaled up or down to meet the rate control objectives.

Note that the ROIs are specified in the order of decreasing importance, so region 1 gets applied before region 0 as shown in the analyzer display with the QP shown below:



Note that with this method of ROI, the ROI map is initialized to qoffset=0. So in the example above, if only my_roi[0] is specified, then the background QP would be the default QP of the encoder.

The formula for how the encoder applies the ROI is as follows:

The QP of an ROI block (16x16 MB for H.264 or 32x32 sub-CTU for H.265) is:

QP = (ROI QP – ROI AVG QP) + QP by RC

where:

ROI QP: the QP specified for the ROI block (1-51) in the ROI map

ROI AVG QP: the average of all the QPs of all ROI blocks

QP by RC: the QP determined by the rate control algorithm

So if ROI QP is larger than average, the rate control QP is increased (poorer quality); if it's smaller than average, the rate control QP is decreased (better quality).

6.5.6.2.3 libxcoder_logan *cacheRoi* Encode Option

The libxcoder_logan ROI API is designed to specify the ROI on a single frame and needs to be specified for each frame. There is a libxcoder_logan encoder configuration option *cacheRoi* that can change this behavior. By using this option, the user can specify the ROI on any frame and it will stay in effect, i.e. cached and applied to the subsequent frames, until a new one is supplied. Its usage is shown in the following ***xcoder_logan*** command example, assuming that ROI maps have been set and applied appropriately:

sudo xcoder_logan -c 0 -i 1280x720p_Basketball.264 -o output.265 -s 1280x720 -m a2h -x ***roiEnable=1****:****cacheRoi=1***

6.5.6.3 Reconfiguration

Encoder reconfiguration is a feature that permits encoding parameters to be changed while encoding is in progress. The types of parameters that can be changed are described in the application note APPS008-netint_apps_note_reconfig[4]. The following is an example of bitrate reconfiguration.

Libxcoder_logan auxiliary data type NI_FRAME_AUX_DATA_BITRATE is used to represent bitrate data. The actual data payload type is int32_t which represents the bitrate to set.

A frame's auxiliary data supporting this feature can be constructed and inserted into a frame as follows:

```
ni_aux_data_t *aux_data = NULL ;

aux_data = ni_logan_frame_new_aux_data(frame, NI_FRAME_AUX_DATA_BITRATE,
                                  sizeof(int32_t));

if (aux_data)
{
  *((int32_t *)aux_data->data) = bitrate;
}
```

6.5.6.4 HLG Preferred Transfer Characteristics

The libxcoder_logan encode option **prefTRC** specifies the preferred transfer characteristics value. 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. This SEI is required by ETSI for HLG and specifies an alternative transfer characteristics from that is provided in the VUI.

The handling of this SEI at encoding is done automatically by libxcoder_logan.

Note that currently, libxcoder_logan does not support this SEI on the decoder side.

6.5.6.5 User Data Unregistered SEI

Handling of User Data Unregistered (UDU) SEI is done automatically by libxcoder_logan. The UDU data is retrieved and stored in the frame as auxiliary data of type

NI_FRAME_AUX_DATA_UDU_SEI after decoding, and it is converted into appropriate form and sent to encoder at encoding. Refer to APPS020[9] for more details.

6.5.6.6 Long Term Reference Frame

The Long Term Reference (LTR) feature involves marking a frame suitable for use as a long term reference and specifying which frames should use the long term reference frame when encoding the current picture for the encoder.

To use this feature, the following steps are required:

- The feature must be enabled by libxcoder_logan encode option **longTermReferenceEnable=1**;
- LTR information needs to be supplied with the desired YUV frame, in the form of auxiliary data.

The struct describing LTR info is as follows:

```
typedef struct _ni_long_term_ref
{
  // A flag for the current picture to be used as a long term reference
  // picture later at other pictures' encoding
  uint8_t use_cur_src_as_long_term_pic;

  // A flag to use a long term reference picture in DPB when encoding the
  // current picture
  uint8_t use_long_term_ref;
} ni_long_term_ref_t;
```

A frame's auxiliary data supporting this feature can be constructed and inserted into a frame as follows:

```
ni_long_term_ref_t *p_ltr ;
ni_aux_data_t *aux_data = NULL ;

aux_data = ni_logan_frame_new_aux_data(frame, NI_FRAME_AUX_DATA_LONG_TERM_REF,
                                 sizeof(ni_long_term_ref_t)) ;

if (aux_data)
{
  p_ltr = (ni_long_term_ref_t *)aux_data->data;

  p_ltr->use_cur_src_as_long_term_pic = ..;

  p_ltr->use_long_term_ref = ..;
}
```

6.6 Transcoding

The transcoding process connects decode to encode in a pipeline: the output of the decoder (YUV raw data and auxiliary data) is fed to the encoder as input. A transcoding flow chart looks as follows:

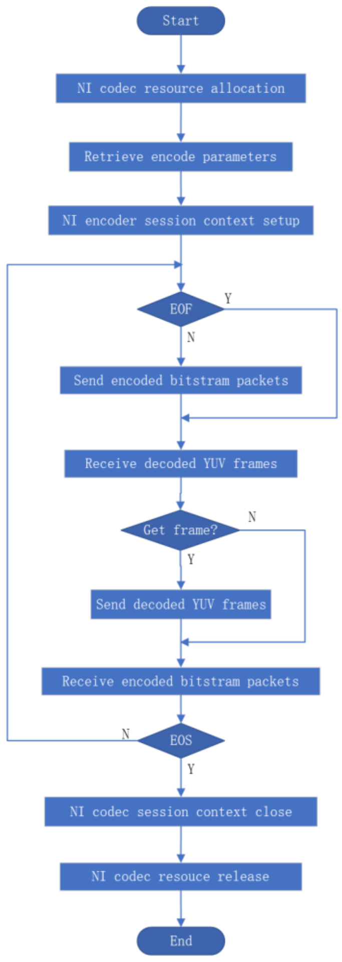

Start
NI codec resource allocation
Retrieve encode parameters
NI encoder session context setup
EOF
Y
N
Send encoded bitstram packets
Receive decoded YUV frames
Get frame?
N
Y
Send decoded YUV frames
Receive encoded bitstram packets
N
EOS
Y
NI codec session context close
NI codec resouce release
End

Figure 14 NETINT Encoding Flow Chart

```
    ni_logan_session_data_io_t in_pkt = {0} ;
    ni_logan_session_data_io_t out_frame = {0} ;
    ni_logan_session_data_io_t enc_in_frame = {0} ;
    ni_logan_session_data_io_t out_packet = {0};

    while (send_fin_flag == 0 || receive_fin_flag == 0 )
    {
      // bitstream Sending
      send_fin_flag = decoder_send_data(
        &dec_ctx, &in_pkt, sos_flag,
        input_video_width, input_video_height, pkt_size, total_file_size,
        &total_bytes_sent, print_time, &xcodeState);

      // YUV Receiving: not writing to file
      receive_fin_flag  = decoder_receive_data(
        &dec_ctx, &out_frame, output_video_width, output_video_height,
        p_file, &total_bytes_received, print_time, 0, &xcodeState);

      if (2 == receive_fin_flag)
      {
        ni_log(NI_LOG_TRACE, "no decoder output, jump to encoder receive !\n");
        goto encode_recv;
      }

      // YUV Sending
      send_fin_flag = encoder_send_data2(
        &enc_ctx, &dec_ctx, &out_frame, &enc_in_frame, sos_flag,
        input_video_width, input_video_height, pfs, total_file_size,
        &total_bytes_sent, &xcodeState);

      ni_logan_decoder_frame_buffer_free(&out_frame.data.frame);

      // encoded bitstream Receiving
encode_recv:
      receive_fin_flag  = encoder_receive_data(
        &enc_ctx, &out_packet, output_video_width, output_video_height,
        p_file, &xcodeRecvTotal, print_time);

      // Error or encoder eos
      if (receive_fin_flag == 2 || out_packet.data.packet.end_of_stream)
      {
        break;
      }
    }
```

The code snippet is stripped from *xcoder_logan* demo and simplified. Check *xcoder_logan* program source code for transcoding flow example, and some practical usage such as encoder reusing the YUV frame struct generated by decoder if only YUV data and no other auxiliary data is sent to encoder for encoding. This reusing can speed up processing and reduce memory copy, if the source and target stream are of the same codec, taking advantage of the same NETINT YUV420p frame layout employed by the decoder and encoder. Care must be taken however, when transcoding between H.264 and H.265 since the alignment requirements are different.

7 Appendix

7.1 API Description Details

Allocate session context & init

```
/*!****************************************************************************
 *  \brief  Allocate and initialize a new ni_logan_session_context_t struct
 *
 *  \return On success returns a valid pointer to newly allocated context. On failure returns NULL
 ****************************************************************************/
 LIB_API ni_logan_session_context_t * ni_logan_device_session_context_alloc_init(void);
```

Init session context

```
/*!****************************************************************************
 *  \brief  Initialize already allocated session context to a known state
 *
 *  \param[in]  p_ctx Pointer to an already allocated ni_logan_session_context_t struct
 ****************************************************************************/
 LIB_API void ni_logan_device_session_context_init(ni_logan_session_context_t *p_ctx);
```

Free session context

```
/*!****************************************************************************
 *  \brief  Frees previously allocated session context
 *
 *  \param[in]  p_ctx Pointer to an already allocated ni_logan_session_context_t struct
 ****************************************************************************/
 LIB_API void ni_logan_device_session_context_free(ni_logan_session_context_t *p_ctx);
```

Create an event handle (Windows only)

```
/*!****************************************************************************
  *  \brief  Create event and returns event handle if successful
  *
  *  \return On success returns a event handle
  *        On failure returns NI_INVALID_EVENT_HANDLE
*****************************************************************************/
```

```
LIB_API ni_event_handle_t ni_logan_create_event();
```

Close and release event resource (Windows only)

```
/*!******************************************************************************
 *  \brief  Closes event and releases resources
 *
 *  \return NONE
 ******************************************************************************/
LIB_API void ni_logan_close_event(ni_event_handle_t event_handle);
```

Open device

```
/*!******************************************************************************
 *  \brief  Opens device and returns device device_handle if successful
 *
 *  \param[in]  p_dev Device name represented as c string. ex: "/dev/nvme0", "/dev/nvme0n1",
 *                   "\\.\PHYSICALDRIVE0".
 *  \param[out] p_max_io_size_out Maximum IO Transfer size supported (Linux only)
 *
 *  \return On success returns a device device_handle
 *              On failure returns NI_INVALID_DEVICE_HANDLE
 ******************************************************************************/
LIB_API ni_device_handle_t ni_logan_device_open(const char* dev, uint32_t *
p_max_io_size_out);
```

Note: On Linux when using regular I/O, it is the block device (not the char device) file that shall be opened for communication.

Close device

```
/*!******************************************************************************
 *  \brief  Closes device and releases resources
 *
 *  \param[in] device_handle Device handle obtained by calling ni_logan_device_open()
 *
 *  \return NONE
 *******************************************************************************/
LIB_API void ni_logan_device_close(ni_device_handle_t dev);
```

Query device capability

```
/*!*****************************************************************************
 *  \brief  Queries device and returns device capability structure
 *
 *  \param[in] device_handle Device handle obtained by calling ni_logan_device_open()
 *  \param[in] p_cap  Pointer to a caller allocated ni_logan_device_capability_t struct
 *  \return On success
 *                  NI_LOGAN_RETCODE_SUCCESS
 *              On failure
                    NI_LOGAN_RETCODE_INVALID_PARAM
                    NI_LOGAN_RETCODE_ERROR_MEM_ALOC
                    NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_device_capability_query(ni_device_handle_t
device_handle, ni_logan_device_capability_t *p_cap);
```

Open device session

```
/*!*****************************************************************************
 *  \brief  Opens a new device session depending on the device_type parameter
 *        If device_type is NI_LOGAN_DEVICE_TYPE_DECODER opens decoding session
 *        If device_type is NI_LOGAN_DEVICE_TYPE_EECODER opens encoding session
 *
 *  \param[in] p_ctx       Pointer to a caller allocated ni_logan_session_context_t struct
 *  \param[in] device_type  NI_LOGAN_DEVICE_TYPE_DECODER or
NI_LOGAN_DEVICE_TYPE_ENCODER
 *  \return On success
 *                      NI_LOGAN_RETCODE_SUCCESS
 *        On failure
 *                      NI_LOGAN_RETCODE_INVALID_PARAM
 *                      NI_LOGAN_RETCODE_ERROR_MEM_ALOC
 *                      NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
 *                      NI_LOGAN_RETCODE_ERROR_INVALID_SESSION
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_device_session_open(ni_logan_session_context_t *p_ctx,
ni_logan_device_type_t device_type);
```

Close device session

```
/*!*****************************************************************************
```

```
 *  \brief  Closes device session that was previously opened by calling
ni_logan_device_session_open()
 *        If device_type is NI_LOGAN_DEVICE_TYPE_DECODER closes decoding session
 *        If device_type is NI_LOGAN_DEVICE_TYPE_EECODER closes encoding session
 *
 *  \param[in] p_ctx       Pointer to a caller allocated ni_logan_session_context_t struct
 *  \param[in] eos_received Flag indicating if End Of Stream indicator was received
 *  \param[in] device_type  NI_LOGAN_DEVICE_TYPE_DECODER or
NI_LOGAN_DEVICE_TYPE_ENCODER
 *  \return On success
 *                    NI_LOGAN_RETCODE_SUCCESS
 *             On failure
 *                    NI_LOGAN_RETCODE_INVALID_PARAM
 *                    NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
 *                    NI_LOGAN_RETCODE_ERROR_INVALID_SESSION
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_device_session_close(ni_logan_session_context_t *p_ctx,
int eos_received, ni_logan_device_type_t device_type);
```

Flush a device session

```
/*!*****************************************************************************
 *  \brief  Sends a flush command to the device
 *        If device_type is NI_LOGAN_DEVICE_TYPE_DECODER sends flush command to decoder
 *        If device_type is NI_LOGAN_DEVICE_TYPE_EECODER sends flush command to decoder
 *
 *  \param[in] p_ctx       Pointer to a caller allocated ni_logan_session_context_t struct
 *  \param[in] device_type  NI_LOGAN_DEVICE_TYPE_DECODER or
NI_LOGAN_DEVICE_TYPE_ENCODER
 *  \return On success
 *                    NI_LOGAN_RETCODE_SUCCESS
 *             On failure
 *                    NI_LOGAN_RETCODE_INVALID_PARAM
 *                    NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
 *                    NI_LOGAN_RETCODE_ERROR_INVALID_SESSION
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_device_session_flush(ni_logan_session_context_t *p_ctx,
ni_logan_device_type_t device_type);
```

Save a stream's headers in a decoder session.

```
/*!****************************************************************************
 *  \brief  Save a stream's headers in a decoder session that can be used later
 *          for continuous decoding from the same source.
 *
 *  \param[in] p_ctx       Pointer to a caller allocated
 *                               ni_logan_session_context_t struct
 *  \param[in] hdr_data    Pointer to header data
 *  \param[in] hdr_size    Size of header data in bytes
 *  \return On success
 *                   NI_LOGAN_RETCODE_SUCCESS
 *          On failure
 *                   NI_LOGAN_RETCODE_INVALID_PARAM
 *                   NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
 *                   NI_LOGAN_RETCODE_ERROR_INVALID_SESSION
 *****************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_device_dec_session_save_hdrs(
 ni_logan_session_context_t *p_ctx, uint8_t *hdr_data, uint8_t hdr_size);
```

Flush a decoder session to get ready to continue decoding.

```
/*!****************************************************************************
 *  \brief  Flush a decoder session to get ready to continue decoding.
 *  Note: this is different from ni_logan_device_session_flush in that it closes the
 *        current decode session and opens a new one for continuous decoding.
 *
 *  \param[in] p_ctx       Pointer to a caller allocated
 *                               ni_logan_session_context_t struct
 *  \return On success
 *                   NI_LOGAN_RETCODE_SUCCESS
 *          On failure
 *                   NI_LOGAN_RETCODE_INVALID_PARAM
 *                   NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
 *                   NI_LOGAN_RETCODE_ERROR_INVALID_SESSION
 *****************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_device_dec_session_flush(ni_logan_session_context_t
*p_ctx);
```

Write to a device session

```
/*!****************************************************************************
```

```
 *  \brief  Sends data to the device
 *        If device_type is NI_LOGAN_DEVICE_TYPE_DECODER sends data packet to decoder
 *        If device_type is NI_LOGAN_DEVICE_TYPE_ENCODER sends data frame to encoder
 *
 *  \param[in] p_ctx       Pointer to a caller allocated  ni_logan_session_context_t struct
 *  \param[in] p_data       Pointer to a caller allocated  ni_logan_session_data_io_t struct which
                        contains either a  ni_logan_frame_t data frame or ni_logan_packet_t data
packet to send
 *  \param[in] device_type  NI_LOGAN_DEVICE_TYPE_DECODER or
NI_LOGAN_DEVICE_TYPE_ENCODER
 *                        If NI_LOGAN_DEVICE_TYPE_DECODER is specified, it is expected
 *                        that the ni_logan_packet_t struct inside the p_data pointer
 *                        contains data to send.
 *                        If NI_LOGAN_DEVICE_TYPE_ENCODER is specified, it is expected
 *                        that the ni_logan_frame_t struct inside the p_data pointer
 *                        contains data to send.
 *  \return On success
 *                        Total number of bytes written
 *              On failure
 *                        NI_LOGAN_RETCODE_INVALID_PARAM
 *                        NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
 *                        NI_LOGAN_RETCODE_ERROR_INVALID_SESSION
******************************************************************************/
LIB_API int ni_logan_device_session_write(ni_logan_session_context_t* p_ctx,
ni_logan_session_data_io_t* p_data, ni_logan_device_type_t device_type);
```

Read from a device session

```
/*!*****************************************************************************
 *  \brief  Reads data from the device
 *        If device_type is NI_LOGAN_DEVICE_TYPE_DECODER reads data frame from decoder
 *        If device_type is NI_LOGAN_DEVICE_TYPE_ENCODER reads data packet from encoder
 *
 *  \param[in] p_ctx       Pointer to a caller allocated  ni_logan_session_context_t struct
 *  \param[in] p_data       Pointer to a caller allocated
 *                        ni_logan_session_data_io_t struct which contains either a
 *                        ni_logan_frame_t data frame or ni_logan_packet_t data packet to receive data
 *  \param[in] device_type  NI_LOGAN_DEVICE_TYPE_DECODER or
NI_LOGAN_DEVICE_TYPE_ENCODER
 *                        If NI_LOGAN_DEVICE_TYPE_DECODER is specified, data that is
 *                   read will be placed into ni_logan_frame_t struct inside the p_data pointer
 *                   If NI_LOGAN_DEVICE_TYPE_ENCODER is specified,  data that is
 *                   read will be placed into ni_logan_packet_t struct inside the p_data pointer
 *  \return On success
 *                   Total number of bytes read
 *            On failure
 *                   NI_LOGAN_RETCODE_INVALID_PARAM
 *                   NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
 *                   NI_LOGAN_RETCODE_ERROR_INVALID_SESSION
******************************************************************************/
LIB_API int ni_logan_device_session_read(ni_logan_session_context_t* p_ctx,
ni_logan_session_data_io_t* p_data, ni_logan_device_type_t device_type);
```

Query a device session

```
/*!****************************************************************************
*  \brief  Query session data from the device
*       If device_type is NI_LOGAN_DEVICE_TYPE_DECODER query session data from decoder
*       If device_type is NI_LOGAN_DEVICE_TYPE_ENCODER query session data from encoder
*
*  \param[in] p_ctx       Pointer to a caller allocated ni_logan_session_context_t struct
*  \param[in] device_type  NI_LOGAN_DEVICE_TYPE_DECODER or
NI_LOGAN_DEVICE_TYPE_ENCODER
*
*  \return On success
*                   NI_LOGAN_RETCODE_SUCCESS
*            On failure
*                   NI_LOGAN_RETCODE_INVALID_PARAM
*                   NI_LOGAN_RETCODE_ERROR_NVME_CMD_FAILED
*                   NI_LOGAN_RETCODE_ERROR_INVALID_SESSION
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_device_session_query(ni_logan_session_context_t* p_ctx,
ni_logan_device_type_t device_type);
```

Allocate a frame buffer

```
/*!****************************************************************************
*  \brief  Allocate preliminary memory for the frame buffer for encoding
*       based on provided parameters. Applicable to YUV420 Planar pixel
*       format only, 8 or 10 bit/pixel.
```

```
 *
 * \param[in] p_frame      Pointer to a caller allocated  ni_logan_frame_t struct
 * \param[in] video_width   Width of the video frame
 * \param[in] video_height  Height of the video frame
 * \param[in] alignment     Alignment requirement
 * \param[in] metadata_flag Flag indicating if space for additional metadata should be allocated
 * \param[in] factor        1 for 8 bits/pixel format, 2 for 10 bits/pixel
 *
 * \return On success
 *                    NI_LOGAN_RETCODE_SUCCESS
 *            On failure
 *                    NI_LOGAN_RETCODE_INVALID_PARAM
 *                    NI_LOGAN_RETCODE_ERROR_MEM_ALOC
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_frame_buffer_alloc(ni_logan_frame_t *pframe, int
video_width, int video_height, int alignment, int metadata_flag, int factor);
```

Allocate a frame buffer for encoding

```
/*!****************************************************************************
 *  \brief  Allocate memory for the frame buffer for encoding based on given
 *        parameters, taking into account pic line size and extra data.
 *        Applicable to YUV420p AVFrame only. 8 or 10 bit/pixel.
 *        Cb/Cr size matches that of Y.
 *
 *  \param[in] p_frame      Pointer to a caller allocated ni_logan_frame_t struct
 *  \param[in] video_width   Width of the video frame
 *  \param[in] video_height  Height of the video frame
 *  \param[in] linesize     Picture line size
 *  \param[in] alignment     Alignment requirement
 *  \param[in] extra_len     Extra data size (incl. meta data, reconf data, SEI data)
 *
 *  \return On success
 *                    NI_LOGAN_RETCODE_SUCCESS
 *            On failure
 *                    NI_LOGAN_RETCODE_INVALID_PARAM
 *                    NI_LOGAN_RETCODE_ERROR_MEM_ALOC
 *****************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_encoder_frame_buffer_alloc(ni_logan_frame_t *pframe,
int video_width, int video_height, int linesize[], int alignment, int extra_len);
```

Allocate memory for a decoder frame buffer

```
/*!*****************************************************************************
 *  \brief  Allocate memory for decoder frame buffer based on provided
 *          parameters; the memory is retrieved from a buffer pool and will be
 *          returned to the same buffer pool by ni_logan_decoder_frame_buffer_free.
 *  Note:   all attributes of ni_logan_frame_t will be set up except for memory and
 *          buffer, which rely on the pool being allocated; the pool will be
 *          allocated only after the frame resolution is known.
 *
 *  \param[in] p_pool       Buffer pool to get the memory from
 *  \param[in] p_frame      Pointer to a caller allocated ni_logan_frame_t struct
 *  \param[in] alloc_mem    Whether to get memory from buffer pool
 *  \param[in] video_width  Width of the video frame
 *  \param[in] video_height Height of the video frame
 *  \param[in] alignment    Allignment requirement
 *  \param[in] factor       1 for 8 bits/pixel format, 2 for 10 bits/pixel
 *
 *  \return On success
 *                      NI_LOGAN_RETCODE_SUCCESS
 *          On failure
 *                      NI_LOGAN_RETCODE_INVALID_PARAM
 *                      NI_LOGAN_RETCODE_ERROR_MEM_ALOC
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_decoder_frame_buffer_alloc(ni_logan_buf_pool_t*
p_pool, ni_logan_frame_t *pframe, int alloc_mem, int video_width, int video_height, int
alignment, int factor);
```

Free a frame buffer

```
/*!*****************************************************************************
 *  \brief  Free frame buffer that was previously allocated with either
 *          ni_logan_frame_buffer_alloc or ni_logan_encoder_frame_buffer_alloc
 *
 *  \param[in] p_frame    Pointer to a previously allocated ni_logan_frame_t struct
 *
 *  \return On success    NI_LOGAN_RETCODE_SUCCESS
 *          On failure    NI_LOGAN_RETCODE_INVALID_PARAM
******************************************************************************/
```

```
LIB_API ni_logan_retcode_t ni_logan_frame_buffer_free(ni_logan_frame_t *pframe);
```

Free a decoder frame buffer

```
/*!*****************************************************************************
 *  \brief  Free decoder frame buffer that was previously allocated with
 *          ni_logan_decoder_frame_buffer_alloc, returning memory to a buffer pool.
 *
 *  \param[in] p_frame    Pointer to a previously allocated ni_logan_frame_t struct
 *
 *  \return On success    NI_LOGAN_RETCODE_SUCCESS
 *          On failure    NI_LOGAN_RETCODE_INVALID_PARAM
 ******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_decoder_frame_buffer_free(ni_logan_frame_t *pframe);
```

Return a decoder frame buffer to buffer pool

```
/*!*****************************************************************************
 *  \brief  Return a memory buffer to memory buffer pool, for a decoder frame.
 *
 *  \param[in] buf            Buffer to be returned.
 *  \param[in] p_buffer_pool  Buffer pool to return buffer to.
 *
 *  \return None
 ******************************************************************************/
LIB_API void ni_logan_decoder_frame_buffer_pool_return_buf(ni_logan_buf_t *buf,
ni_logan_buf_pool_t *p_buffer_pool);
```

Allocate a packet buffer

```
/*!*****************************************************************************
 *  \brief  Allocate memory for the packet buffer based on provided packet size
 *
 *  \param[in] p_packet     Pointer to a caller allocated ni_logan_packet_t struct
 *  \param[in] packet_size  Required allocation size
 *
 *  \return On success
 *                    NI_LOGAN_RETCODE_SUCCESS
 *              On failure
 *                    NI_LOGAN_RETCODE_INVALID_PARAM
 *                  NI_LOGAN_RETCODE_ERROR_MEM_ALOC
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_packet_buffer_alloc(ni_logan_packet_t *ppacket, int
packet_size);
```

Free a packet buffer

```
/*!****************************************************************************
 *  \brief  Free packet buffer that was previously allocated with ni_logan_packet_buffer_alloc
 *
 *  \param[in] p_packet    Pointer to a previously allocated ni_logan_packet_t struct
 *
 *  \return On success    NI_LOGAN_RETCODE_SUCCESS
 *             On failure    NI_LOGAN_RETCODE_INVALID_PARAM
 ******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_packet_buffer_free(ni_logan_packet_t *ppacket);
```

Copy video packet data to a packet buffer

```
/*!******************************************************************************
 *  \brief  Copy video packet accounting for alignment
 *
 *  \param[in] p_destination  Destination to copy to
 *  \param[in] p_source       Source to copy from
 *  \param[in] cur_size       current size
 *  \param[out] p_leftover    Pointer to the data that was left over
 *  \param[out] p_prev_size   Size of the data leftover previously
 *
 *  \return On success       Total number of bytes that were copied
 *             On failure       NI_LOGAN_RETCODE_FAILURE
 ******************************************************************************/
LIB_API int ni_logan_packet_copy(void* p_destination, const void* const p_source, int cur_size,
void* p_leftover, int* p_prev_size);
```

Init default encoder parameters

```
/*!******************************************************************************
 *  \brief  Initialize default encoder parameters
 *
 *  \param[out] p_param     Pointer to a user allocated ni_logan_encoder_params_t
 *                         to initialize to default values
 *  \param[in] fps_num    Frames per second, or FPS numerator
 *  \param[in] fps_denom  FPS denominator
 *  \param[in] bit_rate   bit rate
 *  \param[in] width      width
 *  \param[in] height     height
 *
 *  \return On success
 *                   NI_LOGAN_RETCODE_SUCCESS
 *             On failure
 *                   NI_LOGAN_RETCODE_FAILURE
 *                   NI_LOGAN_RETCODE_INVALID_PARAM
******************************************************************************/
LIB_API ni_logan_retcode_t
ni_logan_encoder_init_default_params(ni_logan_encoder_params_t *p_param, int fps_num, int
fps_denom, long bit_rate, int width, int height);
```

Init default decoder parameters

```
/*!****************************************************************************
 *  \brief  Initialize default decoder parameters
 *
 *  \param[out] param     Pointer to a user allocated ni_logan_encoder_params_t
 *                         to initialize to default values
 *  \param[in] fps_num    Frames per second, FPS numerator
 *  \param[in] fps_denom  FPS denominator
 *  \param[in] bit_rate   bit rate
 *  \param[in] width      width
 *  \param[in] height     height
 *
 *  \return On success
 *                   NI_LOGAN_RETCODE_SUCCESS
 *             On failure
 *                   NI_LOGAN_RETCODE_FAILURE
 *                   NI_LOGAN_RETCODE_INVALID_PARAM
******************************************************************************/
LIB_API ni_logan_retcode_t
ni_logan_decoder_init_default_params(ni_logan_encoder_params_t *p_param, int fps_num, int
fps_denom, long bit_rate, int width, int height);
```

Parse and set value for encoder parameters

```
/*!******************************************************************************
 *  \brief  Set value referenced by name in encoder parameters structure
 *
 *  \param[in] p_params   Pointer to a user allocated ni_logan_encoder_params_t
 *                        to find and set a particular parameter
 *  \param[in] name       String represented parameter name to search
 *  \param[in] value      Parameter value to set
*
 *  \return On success
 *                  NI_LOGAN_RETCODE_SUCCESS
 *          On failure
 *                  NI_LOGAN_RETCODE_FAILURE
 *                  NI_LOGAN_RETCODE_INVALID_PARAM
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_encoder_params_set_value(ni_logan_encoder_params_t *
p_params, const char *name, const char *value);
```

Parse and set value for encoder custom GOP parameters

```
/*!******************************************************************************
 *  \brief  Set gop parameter value referenced by name in encoder parameters structure
 *
 *  \param[in] p_params   Pointer to a user allocated ni_logan_encoder_params_t
 *                        to find and set a particular parameter
 *  \param[in] name       String represented parameter name to search
 *  \param[in] value      Parameter value to set
*
 *  \return On success
 *                  NI_LOGAN_RETCODE_SUCCESS
 *          On failure
 *                  NI_LOGAN_RETCODE_FAILURE
 *                  NI_LOGAN_RETCODE_INVALID_PARAM
 ******************************************************************************/
LIB_API ni_logan_retcode_t
ni_logan_encoder_gop_params_set_value(ni_logan_encoder_params_t * p_params, const char
*name, const char *value);
```

Get GOP's max number of reorder frames

```
/*!*************************************************************************
 *  \brief  Get GOP's max number of reorder frames
 *
 *  \param[in] p_params   Pointer to a user allocated ni_logan_encoder_params_t
 *
 *  \return max number of reorder frames of the GOP
**************************************************************************/
LIB_API int ni_logan_get_num_reorder_of_gop_structure(ni_logan_encoder_params_t *
p_params);
```

Get GOP’s number of reference frames

```
/*!*************************************************************************
 *  \brief  Get GOP's number of reference frames
 *
 *  \param[in] p_params   Pointer to a user allocated ni_logan_encoder_params_t
 *
 *  \return number of reference frames of the GOP
**************************************************************************/
LIB_API int ni_logan_get_num_ref_frame_of_gop_structure(ni_logan_encoder_params_t *
p_params);
```

Add a new auxiliary data to a frame

```
/*!*************************************************************************
 *  \brief  Add a new auxiliary data to a frame
 *
 *  \param[in/out] frame  a frame to which the auxiliary data should be added
 *  \param[in]     type   type of the added auxiliary data
 *  \param[in]     data_size size of the added auxiliary data
 *
 *  \return a pointer to the newly added aux data on success, NULL otherwise
**************************************************************************/
LIB_API ni_aux_data_t *ni_logan_frame_new_aux_data(ni_logan_frame_t *frame,
                                  ni_aux_data_type_t type,
                                  int data_size);
```

Add a new auxiliary data to a frame and copy in the raw data

```
/*!*************************************************************************
```

```
 * \brief  Add a new auxiliary data to a frame and copy in the raw data
 *
 * \param[in/out] frame  a frame to which the auxiliary data should be added
 * \param[in]     type   type of the added auxiliary data
 * \param[in]     raw_data  the raw data of the aux data
 * \param[in]     data_size size of the added auxiliary data
 *
 * \return a pointer to the newly added aux data on success, NULL otherwise
******************************************************************************/
LIB_API ni_aux_data_t *ni_logan_frame_new_aux_data_from_raw_data(
 ni_logan_frame_t *frame,
 ni_aux_data_type_t type,
 const uint8_t* raw_data,
 int data_size);
```

Retrieve from the frame auxiliary data of a given type if exists

```
/*!*****************************************************************************
 * \brief  Retrieve from the frame auxiliary data of a given type if exists
 *
 * \param[in] frame  a frame from which the auxiliary data should be retrieved
 * \param[in] type   type of the auxiliary data to be retrieved
 *
 * \return a pointer to the aux data of a given type on success, NULL otherwise
******************************************************************************/
LIB_API ni_aux_data_t *ni_logan_frame_get_aux_data(const ni_logan_frame_t *frame,
                                    ni_aux_data_type_t type);
```

Free and remove a given type of auxiliary data from the frame if it exists.

```
/*!*****************************************************************************
 * \brief  If auxiliary data of the given type exists in the frame, free it
 *         and remove it from the frame.
 *
 * \param[in/out] frame a frame from which the auxiliary data should be removed
 * \param[in] type   type of the auxiliary data to be removed
 *
 * \return None
******************************************************************************/
LIB_API void ni_logan_frame_free_aux_data(ni_logan_frame_t *frame,
                        ni_aux_data_type_t type);
```

Free and remove all auxiliary data from the frame.

```
/*!****************************************************************************
 *  \brief  Free and remove all auxiliary data from the frame.
 *
 *  \param[in/out] frame a frame from which the auxiliary data should be removed
 *
 *  \return None
******************************************************************************/
LIB_API void ni_logan_frame_wipe_aux_data(ni_logan_frame_t *frame);
```

Set libxcoder_logan log level

```
/*!****************************************************************************
 *  \brief  Set ni_log_level
 *
 *  \param  level log level
 *
 *  \return
******************************************************************************/
LIB_API void ni_log_set_level(ni_log_level_t level);
```

Get libxcoder_logan log level

```
/*!****************************************************************************
 *  \brief Get ni_log_level
 *
 *  \return ni_log_level
******************************************************************************/
LIB_API ni_log_level_t ni_log_get_level(void);
```

Get NETINT HW YUV420p dimension information

```
/*!****************************************************************************
*
 *  \brief  Get dimension information of NETINT HW YUV420p frame to be sent
 *        to encoder for encoding. Caller usually retrieves this info and
 *        uses it in the call to ni_logan_encoder_frame_buffer_alloc for buffer
 *       allocation.
 *
 * \param[in]  width   source YUV frame width
 * \param[in]  height  source YUV frame height
 * \param[in]  bit_depth_factor  1 for 8 bit, 2 for 10 bit
 * \param[in]  is_h264  non-0 for H.264 codec, 0 otherwise (H.265)
 * \param[out] plane_stride  size (in bytes) of each plane width
 * \param[out] plane_height  size of each plane height
 *
 * \return Y/Cb/Cr stride and height info
******************************************************************************/
LIB_API void ni_logan_get_hw_yuv420p_dim(int width, int height, int bit_depth_factor,
                        int is_h264,
                        int plane_stride[NI_LOGAN_MAX_NUM_DATA_POINTERS],
                        int plane_height[NI_LOGAN_MAX_NUM_DATA_POINTERS]);
```

Copy data to NETINT HW YUV420p frame layout

```
/*!*****************************************************************************
*
 * \brief  Copy YUV data to NETINT HW YUV420p frame layout to be sent
 *       to encoder for encoding. Data buffer (dst) is usually allocated by
 *       ni_logan_encoder_frame_buffer_alloc.
 *
 * \param[out] p_dst  pointers of Y/Cb/Cr to which data is copied
 * \param[in]  p_src  pointers of Y/Cb/Cr from which data is copied
 * \param[in]  width  source YUV frame width
 * \param[in]  height source YUV frame height
 * \param[in]  bit_depth_factor  1 for 8 bit, 2 for 10 bit
 * \param[in]  dst_stride  size (in bytes) of each plane width in destination
 * \param[in]  dst_height  size of each plane height in destination
 * \param[in]  src_stride  size (in bytes) of each plane width in source
 * \param[in]  src_height  size of each plane height in source
 *
 * \return Y/Cb/Cr data
******************************************************************************/
LIB_API void ni_logan_copy_hw_yuv420p(uint8_t
*p_dst[NI_LOGAN_MAX_NUM_DATA_POINTERS],
                       uint8_t *p_src[NI_LOGAN_MAX_NUM_DATA_POINTERS],
                       int width, int height, int bit_depth_factor,
                       int dst_stride[NI_LOGAN_MAX_NUM_DATA_POINTERS],
                       int dst_height[NI_LOGAN_MAX_NUM_DATA_POINTERS],
                       int src_stride[NI_LOGAN_MAX_NUM_DATA_POINTERS],
                       int src_height[NI_LOGAN_MAX_NUM_DATA_POINTERS]);
```

Insert emulation prevention byte(s) into data buffer

```
/*!***************************************************************************
*
 *  \brief  Insert emulation prevention byte(s) as needed into the data buffer
 *
 *  \param  buf   data buffer to be worked on - new byte(s) will be inserted
 *         size  number of bytes starting from buf to check
 *
 *  \return the number of emulation prevention bytes inserted into buf, 0 if
 *         none.
 *
 *  Note: caller *MUST* ensure for newly inserted bytes, buf has enough free
 *       space starting from buf + size
****************************************************************************/
LIB_API int insert_emulation_prevent_bytes(uint8_t *buf, int size);
```

Set SPS VUI part of encoded stream header

```
/*!***************************************************************************
 *  \brief  Set SPS VUI part of encoded stream header
 *
 *  \param[in/out]  p_param encoder parameters, its VUI data member will be
 *               updated.
 *  \param[in]  color_primaries color primaries
 *  \param[in]  color_trc color transfer characteristic
 *  \param[in]  color_space YUV colorspace type
 *  \param[in]  video_full_range_flag
 *  \param[in]  sar_num/sar_den sample aspect ration in numerator/denominator
 *  \param[in]  codec_format H.264 or H.265
 *  \param[out]  hrd_params struct for HRD parameters, may be updated.
 *
 *  \return NONE
****************************************************************************/
LIB_API void ni_logan_set_vui(ni_logan_encoder_params_t *p_param,
                    ni_color_primaries_t color_primaries,
                    ni_color_transfer_characteristic_t color_trc,
                    ni_color_space_t color_space,
                    int video_full_range_flag,
                    int sar_num, int sar_den,
                    ni_logan_codec_format_t codec_format,
                    ni_hrd_params_t *hrd_params);
```

Whether SEI should be sent together with this frame to encoder

```
/*!*****************************************************************************
 *  \brief  Whether SEI should be sent together with this frame to encoder
 *
 *  \param[in]  p_enc_ctx encoder session context
 *  \param[in]  pic_type frame type
 *  \param[in]  p_param encoder parameters
 *
 *  \return 1 if yes, 0 otherwise
 ******************************************************************************/
LIB_API int ni_logan_should_send_sei_with_frame(ni_logan_session_context_t* p_enc_ctx,
                                  ni_logan_pic_type_t pic_type,
                                  ni_logan_encoder_params_t *p_param);
```

Retrieve auxiliary data (close caption, various SEI, ROI) associated with this frame returned by decoder.

```
/*!*****************************************************************************
 *  \brief  Retrieve auxiliary data (close caption, various SEI) associated with
 *          this frame that is returned by decoder, convert them to appropriate
 *          format and save them in the frame's auxiliary data storage for
 *          future use by encoding. Usually they would be sent together with
 *          this frame to encoder at encoding.
 *
 *  \param[in/out]  frame that is returned by decoder
 *
 *  \return NONE
 ******************************************************************************/
LIB_API void ni_logan_dec_retrieve_aux_data(ni_logan_frame_t *frame);
```

Prepare auxiliary data that should be sent together with this frame to encoder

```
/*!*****************************************************************************
*
 *  \brief  Prepare SEI that should be sent together with this frame to encoder
 *
 *  \param[in]  p_enc_ctx encoder session context
 *  \param[out]  p_enc_frame frame to be sent to encoder
 *  \param[in]  p_dec_frame frame that is returned by decoder
 *  \param[in]  codec_format H.264 or H.265
 *  \param[in]  should_send_sei_with_frame if need to send a certain type of
 *              SEI with this frame
 *  \param[out]  mdcv_data SEI for HDR mastering display color volume info
 *  \param[out]  cll_data SEI for HDR content light level info
 *  \param[out]  cc_data SEI for close caption
 *  \param[out]  udu_data SEI for User data unregistered
 *  \param[out]  hdrp_data SEI for HDR10+
 *
 *  \return NONE
******************************************************************************/
LIB_API void ni_logan_enc_prep_aux_data(ni_logan_session_context_t* p_enc_ctx,
                        ni_logan_frame_t *p_enc_frame,
                        ni_logan_frame_t *p_dec_frame,
                        ni_logan_codec_format_t codec_format,
                        int should_send_sei_with_frame,
                        uint8_t *mdcv_data,
                        uint8_t *cll_data,
                        uint8_t *cc_data ,
                        uint8_t *udu_data,
                        uint8_t *hdrp_data);
```

Copy auxiliary data that should be sent together with this frame to encoder

```
/*!*****************************************************************************
*
 *  \brief  Copy auxiliary data that should be sent together with this frame to encoder
 *
 *  \param[in]  p_enc_ctx encoder session context
 *  \param[out]  p_enc_frame frame to be sent to encoder
 *  \param[in]  mdcv_data SEI for HDR mastering display color volume info
 *  \param[in]  cll_data SEI for HDR content light level info
 *  \param[in]  cc_data SEI for close caption
 *  \param[in]  udu_data SEI for User data unregistered
 *  \param[in]  hdrp_data SEI for HDR10+
 *
 *  \return NONE
******************************************************************************/
LIB_API void ni_logan_enc_copy_aux_data(ni_logan_session_context_t* p_enc_ctx,
                        ni_logan_frame_t *p_enc_frame,
                        const uint8_t *mdcv_data,
                        const uint8_t *cll_data,
                        const uint8_t *cc_data ,
                        const uint8_t *udu_data,
                        const uint8_t *hdrp_data);
```

Allocate a zero copy frame buffer for encoding

```
/*!*****************************************************************************
 *   \brief  Allocate memory for the frame buffer based on provided parameters
 *           Applicable to use YUV420p AVFrame for zerocopy case.
 *           Allocate extra_len only
 *
 *  \param[in]  p_frame Pointer to a caller allocated ni_logan_frame_t struct
 *  \param[in]  video_width Width of the video frame
 *  \param[in]  video_height Height of the video frame
 *  \param[in]  linesize Picture line size
 *  \param[in]  extra_len Extra data size(incl. Meta data)
 *  \param[in]  factor bytes per pixel.(10bit: factor = 2; 8bit:factor = 1)
 *  \param[in]  buffer[] Buffer address of video frame allocated
 *
 *  \return on success NI_LOGAN_RETCODE_SUCCESS
 *  \return on failure NI_LOGAN_RETCODE_INVALID_PARAM
 *                         NI_LOGAN_RETCODE_ERROR_MEM_ALOC
******************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_frame_zerocopy_buffer_alloc(ni_logan_frame_t* p_frame,
int video_width, int video_height, int linesize[], int extra_len,  int factor,  uint8_t* buffer[]);
```

Check if incoming frame is encoder zero copy compatible or not

```
/*!****************************************************************************
 *
 *   \brief  Check if incoming frame is encoder zero copy compatible or not
 *
 *  \param[in]  width width of the encoder frame.
 *  \param[in]  height Height of the encoder frame.
 *  \param[in]  linesize linesizes (pointer to array).
 *  \param[in]  dst_stride dst_stride(produced by ni_logan_get_hw_yuv420p_dim).
 *  \param[in]  src_height src height of every planar.
 *  \param[in]  dst_height dst height aligned (producd by ni_logan_get_hw_yuv420p_dim).
 *  \param[in]  bit_depth_factor bit depth of frame.
 *  \param[in]  data CPU address of frame planar.
 *
 *  \return on success NI_LOGAN_RETCODE_SUCCESS
 *  \return on failure NI_LOGAN_RETCODE_FAILURE
 *
*****************************************************************************/
LIB_API ni_logan_retcode_t ni_logan_frame_zerocopy_check(const int width, const int height,
const int linesize[], const int dst_stride[], const int src_height[], const int dst_height[], const int
bit_depth_factor, const uint8_t * data[]);
```

Check if logan firmware version is higher than expected firmware api flavor and version.

```
/*!***************************************************************************
 *   \brief  Check if  logan firmware version is higher than expected api flavor and version.
 *
 *  \param[in]  p_ctx Pointer to a caller allocated ni_logan_session_context_t struct.
 *  \param[in]  fw_api_fla expected firmware api flavor.
 *  \param[in]  fw_api_ver expected firmware api version.
 *
 *  \return on success NI_LOGAN_RETCODE_SUCCESS
 *              on failure NI_LOGAN_RETCODE_FAILURE
*****************************************************************************/
LIB_API ni_logan_retcode_t is_logan_fw_rev_higher(ni_logan_session_context_t* p_ctx, int
fw_api_fla, int fw_api_ver);
```

8 Revision History

Version	Change	Create Date	Author
1.0	Document Created	Feb 20, 2020	Zhong Wang
1.1	Updated	Mar 26, 2021	Zhong Wang
2.0	Added Integration sections	June 23, 2021	Zhong Wang
2.1	Added decoder output, encoder input layout sections.	July 29, 2021	Zhong Wang
2.2	Added sections for auxiliary data. Doc review.	September 17, 2021	Neil Gunn, Zhong Wang
2.3	Updates of xcoder command line options, simplified resource allocation, added decoder flush API.	October 29, 2021	Neil Gunn, Steven Zhou, Zhong Wang
3.0	Codec/library renaming, move API details to Appendix	July 20, 2022	Zhong Wang
3.1	Added codec flow chart	Feb 13, 2023	Leo Ma