Quadra libxcoder API Guide V5.6

1 Legal Notice

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

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

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

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

2 Table of Abbreviations

Abbreviation	Description
AI	Artificial Intelligence
FW	Firmware
GOP	Group of Pictures
HDR	High Dynamic Range
HW	Hardware
LTR	Long Term Reference
NAL	Network Abstraction layer
NHWC	Number of samples x Height x Width x Channels
RBSP	Raw Byte Sequence Payload
RGB	Red x Green x Blue
ROI	Region Of Interest
SEI	Supplemental Enhancement Information
SEI	Supplemental Enhancement Information ni_device_namespace_read
SPS	Sequence Parameter Set
TBC	To Be Completed
VUI	Video Useability Information

3 References

[1] NETINT Quadra 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 User data unregistered SEI passthrough: APPS520

[4] NETINT Long term reference app note : APPS528

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

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

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

4 Background

This document describes in detail the NETINT proprietary libxcoder API. The libxcoder API is provides an interface to Quadra devices, allowing an application to control the NETINT Quadra Video Processing Unit.

4.1 Intended Audience

This document is intended to help engineers/technicians/developers integrate with NETINT Video Processing Units. It can also be used for reference as it provides a detailed explanation for each API function and structure.

4.2 Compatibility

Software Compatibility

This guide is intended to be used with NETINT Quadra Tx Video Processing software Release 5.6.

Hardware Compatibility

Quadra Release Package 5.6 supports all the NETINT Quadra Video Processing devices, including:

- T1M (M.2)
- T1U (U.2)
- T1A (AIC)
- T2A (AIC)
- T1S (SODIMM)

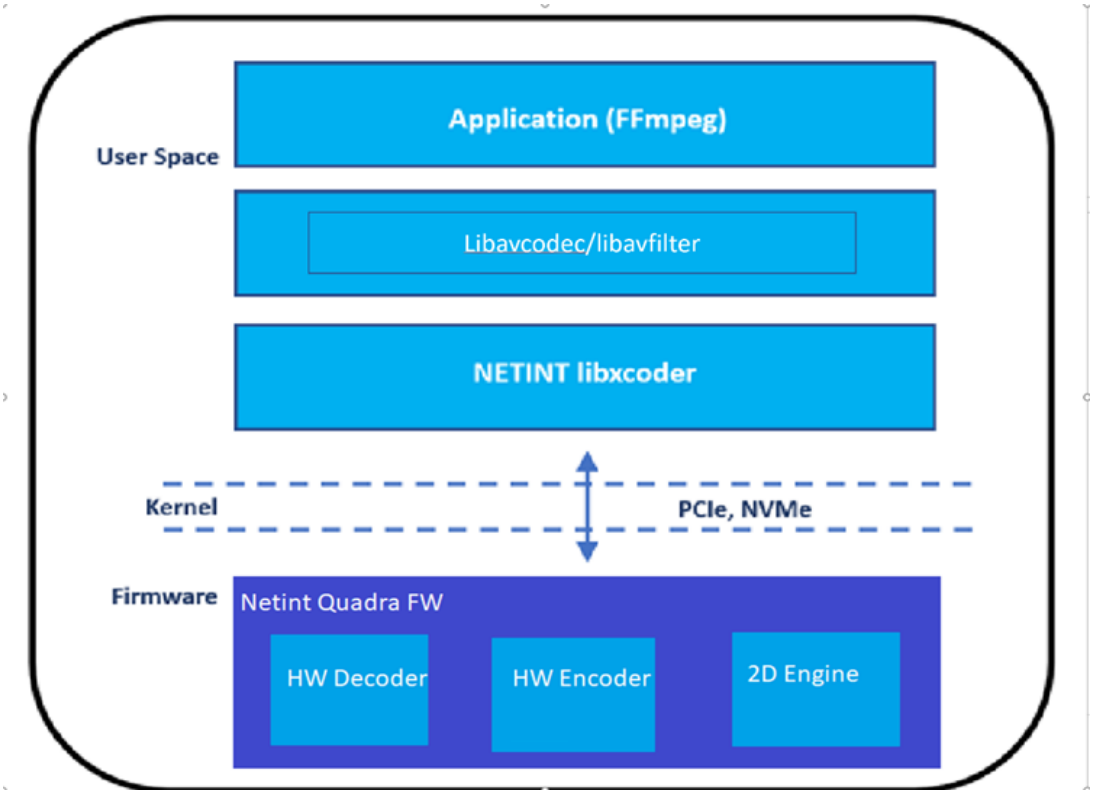
Operation system

Relevant for all Quadra supported Operating Systems.

4.3 Overview

The NETINT libxcoder API is the lowest level at which a user application integrates with the QUADRA Tx Video Processing Unit, as shown in Figure 1.

Figure 1 NETINT video processing unit software framework



Alternatives to directly interfacing an application to the libxcoder API, are to use an Open Source framework such as FFmpeg or GStreamer.

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 below for transcoding.

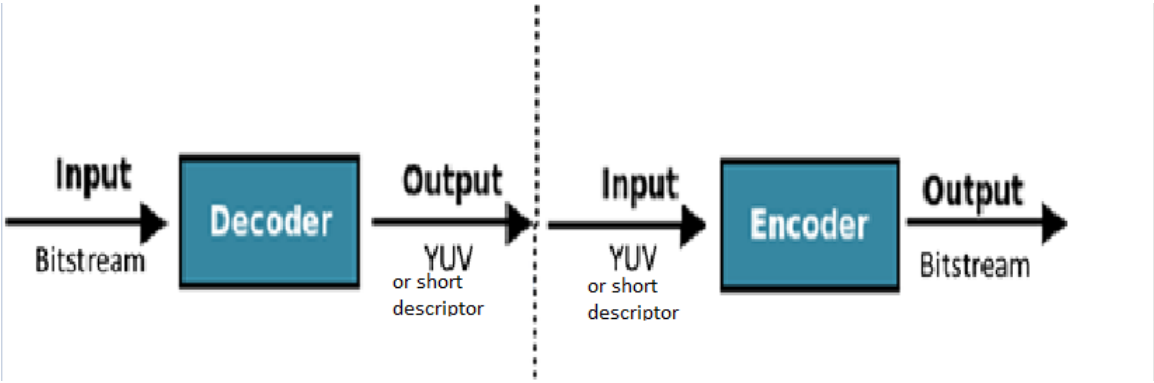


Figure 2 NETINT Decoding and Encoding Pipeline

4.4 Listed Functions

The following table describes each of the libxcoder API functions. For more details please see the **Appendix** Section.

4.4.1 API Description Details

The following tables describes the API functions from the following files

- **ni_device_api.h**
- **ni_av_codec.h**
- **ni_util.h**

API Functions in ni_device_api.h	Description
ni_device_session_context_alloc_init	Allocate and initialize a new **ni_session_context_t** struct
ni_device_session_context_init	Initialize already allocated session context to a known state
ni_device_session_context_clear	Clear already allocated session context
ni_device_session_context_free	Free previously allocated session context
ni_create_event	Create event and return event handle if successful (**Windows only**)
ni_close_event	Close event and releases resources (**Windows only**)
ni_device_open	Open device and return device **device_handle** if successful.It had been deprecated.
ni_device_open2	Open device and return device **device_handle** if successful.
ni_device_close	Close device and release resources
ni_device_capability_query	Query device and return device capability structure

API Functions in ni_device_api.h	Description
ni_device_capability_query2	Query device and return device capability structure
ni_device_session_open	Open a new device session depending on the **device_type** parameter
ni_device_session_close	Close device session that was previously opened by calling **ni_device_session_open**
ni_device_session_flush	Send a flush command to the device
ni_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_device_dec_session_flush**.
ni_device_dec_session_flush	Flush a decoder session to get ready to continue decoding. This is different from **ni_device_session_flush** in that it closes the current decode session and opens a new one for continuous decoding.
ni_device_session_write	Send data to the device
ni_device_session_read	Read data from the device
ni_device_session_query	Query session data from the device
ni_frame_buffer_alloc	Allocate preliminary memory for the frame buffer for **hwdownload** or **HWframe** descriptor storage based on provided parameters. Applicable to YUV420 or nv12 semiplanar pixel format only, 8 or 10 bit/pixel.

ni_frame_buffer_alloc_dl	Allocate preliminary memory for the frame buffer for **hwdownload** or **HWframe** descriptor storage based on provided parameters. Applicable to all supported pixel formats.
ni_decoder_frame_buffer_alloc	Allocate memory for decoder frame buffer based on provided parameters; the memory is retrieved from a memory buffer pool and will be returned to the same buffer pool by **ni_decoder_frame_buffer_free**.
ni_encoder_frame_buffer_alloc	Allocate memory for the frame buffer for encoder input based on provided parameters taking into account pic line size, extra data, and alignment 2 pass requirements. Applicable to YUV420 only. Cb/Cr size matches that of Y.
ni_scaler_dest_frame_alloc	Allocate device destination frame from scaler hwframe pool. Applicable to scale, crop, pad, overlay operations.
ni_scaler_input_frame_alloc	Allocate device input frame by hw descriptor. This API won't actually allocate a frame but sends the incoming hardware frame index to the scaler manager.
ni_scaler_frame_pool_alloc	Init output hardware frames pool for scaler. Default pool size is 1.
ni_scaler_p2p_frame_acquire	Get p2p buffer address of a frame of scaler.
ni_frame_buffer_alloc_nv	Allocate memory for the frame buffer for encoder input based on provided parameters taking into account pic line size, extra data, and alignment 2 pass requirements . Applicable to NV12 only. Cb/Cr size matches that of Y.
ni_encoder_sw_frame_buffer_alloc	This API is a wrapper for **ni_encoder_frame_buffer_alloc()**, used for planar pixel formats, and **ni_frame_buffer_alloc_nv()**, used for semi-planar pixel formats. This API is meant to combine the functionality for both individual format APIs.

ni_frame_buffer_free	Free frame buffer that was previously allocated with **ni_frame_buffer_alloc** or **ni_frame_buffer_alloc_nv** or **ni_encoder_frame_buffer_alloc** or **ni_frame_buffer_alloc_dl**
ni_decoder_frame_buffer_free	Free decoder frame buffer that was previously allocated with **ni_decoder_frame_buffer_alloc**, returning memory to a buffer pool.
ni_decoder_frame_buffer_pool_return_buf	Return a memory buffer to memory buffer pool, for a decoder frame.
ni_packet_buffer_alloc	Allocate memory for the packet buffer based on provided packet size
ni_custom_packet_buffer_alloc	Allocate packet buffer using a user provided pointer
ni_packet_buffer_free	Free packet buffer that was previously allocated with **ni_packet_buffer_alloc**
ni_packet_buffer_free_av1	Free packet buffer that was previously allocated with **ni_packet_buffer_alloc** for AV1 packets merge
ni_packet_copy	Copy video packet accounting for alignment and padding
ni_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_frame_new_aux_data_from_raw_data	Add a new auxiliary data to a frame and copy in the raw data
ni_frame_get_aux_data	Retrieve from the frame auxiliary data of a given type if exists
ni_frame_free_aux_data	If auxiliary data of the given type exists in the frame, free it and remove it from the frame.

ni_frame_wipe_aux_data	Free and remove all auxiliary data from the frame.
ni_encoder_init_default_params	Initialize default encoder parameters
ni_decoder_init_default_params	Initialize default decoder parameters
ni_encoder_params_set_value	Set value referenced by name in encoder parameters structure
ni_decoder_params_set_value	Set value referenced by name in decoder parameters structure
ni_encoder_gop_params_set_value	Set GOP parameter value referenced by name in encoder parameters structure
ni_device_session_copy	Copy source session params into target session
ni_device_session_init_framepool	Send frame pool setup info to device for uploader instance creation
ni_device_session_update_framepool	Adjust uploader instance frame pool
ni_device_session_read_hwdesc	Read HW descriptor from decoder or 2D engine instance.
ni_device_session_hwdl	Read raw YUV data stored on device based on provided HW descriptor
ni_device_session_hwup	Send raw YUV to uploader instance and update the HW descriptor structure with new corresponding descriptor information
ni_frame_buffer_alloc_hwenc	Allocates memory for HW descriptor buffer used for encoder input based on provided parameters
ni_hwframe_buffer_recycle	Free the memory buffer used for HW descriptor buffer and send the command to device to recycle the corresponding memory bin it referred to

ni_hwframe_buffer_recycle2	Provide same function of ni_hwframe_buffer_recycle, but without device handle parameter
ni_scaler_set_params	Set parameters on the device for the 2D engine
ni_device_config_frame	Configure a frame on the device for 2D engine to work on based on provided parameters
ni_device_multi_config_frame	Configure multiple frames on the device for 2D engine to work on based on provided parameters
ni_device_alloc_frame	Allocate an input or output frame on the device for 2D engine based on provided parameters
ni_device_alloc_dst_frame	Allocate a frame on the device and return the frame index with a hw frame surface
ni_device_clone_hwframe	Copy the data of src hw frame to dst hw frame
ni_frame_buffer_alloc_pixfmt	Allocate memory for the frame buffer to be used for HW upload or 2D engine i/o frames based on provided parameters taking into account pixel format, width, height, format, stride, alignment, and extra data
ni_ai_config_network_binary	Configure a network context based with the network binary provided
ni_ai_frame_buffer_alloc	Allocate input layers memory for AI frame buffer based on provided parameters
ni_ai_packet_buffer_alloc	Allocate output layers memory for the packet buffer based on provided network
ni_reconfig_bitrate	Reconfigure bitrate dynamically during encoding.

ni_reconfig_intraprd	Reconfigure intra period dynamically during encoding. NOTE - the frame upon which intra period is reconfigured is encoded as IDR frame NOTE - reconfigure intra period is not allowed if intraRefreshMode is enabled or if gopPresetIdx is 1
ni_reconfig_vui	Reconfigure VUI parameters dynamically during encoding.
ni_force_idr_frame_type	Force next frame to be IDR frame during encoding.
ni_set_ltr	Set a frame's support of Long Term Reference frame during encoding.
ni_set_ltr_interval	Set Long Term Reference interval
ni_set_frame_ref_invalid	Set frame reference invalidation
ni_reconfig_framerate	Reconfigure framerate dynamically during encoding
ni_reconfig_max_frame_size	Reconfigure **maxFrameSize_Bytes** parameter setting during encoding (for details regarding **maxFrameSize_Bytes** parameter, please refer to Integration Programming Guide) NOTE - maxFrameSize_Bytes value less than ((bitrate / 8) / framerate) will be rejected, refer to Integration Programming Guide maxFrameSize_Bytes descriptions for range of supported values
ni_reconfig_crf	Reconfigure crf value dynamically during encoding (for details regarding **crf** parameter and its reconfiguration, please refer to Integration Programming Guide section 8.4 “Encoding Parameters” for **crf** and **ReconfDemoMode** descriptions)
ni_reconfig_vbv_value	Reconfigure vbv buffer size and vbv max rate dynamically during encoding.

ni_reconfig_max_frame_size_ratio	Reconfigure **maxFrameSizeRatio** parameter setting during encoding (for details regarding **maxFrameSizeRatio** parameter, please refer to Integration Programming Guide)
ni_reconfig_slice_arg	Reconfigure sliceArg dynamically during encoding
ni_device_session_acquire	Acquire a frame buffer from a P2P hwupload session
ni_uploader_frame_buffer_lock	Lock a frame from a P2P hwupload session
ni_uploader_frame_buffer_unlock	Unlock a frame from a P2P hwupload session
ni_uploader_p2p_test_send	Special P2P test API call. Copy YUV data from the software frame to the hardware P2P frame on the Quadra device
ni_encoder_set_input_frame_format	Set the input frame format for the encoder
ni_uploader_set_frame_format	Set the frame format for the hwupload session
ni_hwframe_p2p_buffer_recycle	Recycle a frame from the P2P hwupload session
ni_scaler_set_drawbox_params	Send box paramters to drawbox session
ni_device_session_sequence_change	Send sequence / resolution change information to device NOTE - this API is only for changing from large to small resolution – changing from small to large resolution requires application to close existing session and open new session with large resolution
ni_query_nvme_status	Query NVMe and TP load from the device
ni_encoder_session_read_stream_header	A convenience function that may be used to read the initial stream header from the encoder.
ni_get_dma_buf_file_descriptor	Get DMA buffer file descriptor from HW frame (**P2P only**)

ni_device_config_namespace_num	Config Namespace number for different SRIOV
ni_encoder_frame_zerocopy_check	Check if incoming frame is encoder zero copy compatible or not Applicable to YUV420 8 or 10 bit with planar or semi-planar pixel format, or RGBA / BGRA / ABGR / ARGB pixel format
ni_encoder_frame_zerocopy_buffer_alloc	Allocate memory for encoder zero copy (metadata, etc.) based on given parameters, taking into account pic linesize and extra data Applicable to YUV420 8 or 10 bit with planar or semi-planar pixel format, or RGBA / BGRA / ABGR / ARGB pixel format
ni_uploader_frame_zerocopy_check	Check if input frame is hwupload zero copy compatible or not. Zero copy can be enabled if the incoming frame meets the following requirements - Input frame is in YUV420 8 or 10 bit + planar or semi-planar pixel format, or in RGBA / BGRA / ABGR / ARGB pixel format Input frame resolution >= 1080p For YUV420 8 / 10-bit with planar / semi-planar pixel formats: Zero copy is applicable only if frame buffer linesize is 128 bytes aligned For RGBA / BGRA / ABGR / ARGB pixel formats: Zero copy is applicable only if frame buffer linesize is 64 bytes aligned
ni_query_temperature	Query CompositeTemp from device
ni_query_extra_info	Query CompositeTemp and power from device
ni_device_alloc_and_get_firmware_logs	Allocate log buffer if needed and retrieve firmware logs from device. Also write firmware logs to files if requested.
ni_device_alloc_and_get_firmware_logs	Send a p_config command to reconfigure decoding ppu params.

ni_util.h Functions

API Functions in ni_util.h	Description
ni_get_hw_yuv420p_dim	Get dimension information of NETINT HW YUV420p frame to be sent to encoder for encoding.
ni_copy_hw_yuv420p	Copy YUV data to NETINT HW YUV420p frame layout to be sent to encoder for encoding.
ni_copy_yuv_444p_to_420p	Copy yuv444p data to two pieces of NETINT HW yuv420p frame layout to be sent to encoder for encoding
ni_insert_emulation_prevent_bytes	Insert emulation prevention byte(s) as needed into the data buffer
ni_remove_emulation_present_bytes	Remove emulation prevention byte(s) as needed from the data buffer
ni_copy_hw_descriptors	Copy HW descriptor structure from source to destination
ni_get_libxcoder_api_ver	Get libxcoder API version
ni_get_compat_fw_api_ver	Get FW API version libxcoder is compatible with
ni_get_libxcoder_release_ver	Get libxcoder SW release version
ni_network_layer_convert_output	Helper function to convert one layer's output raw data in binary format to data in tensor format.
ni_ai_network_layer_size	Get one layer's raw size in byte
ni_ai_network_layer_dims	Get one layer's number of dimensions
ni_network_layer_convert_tensor	Helper function to convert one layer's input file in tensor format to hardware raw input data in binary format.

API Functions in ni_util.h	Description
ni_network_convert_tensor_to_data	Helper function to convert one layer's input data in tensor format to hardware raw input data in binary format.
ni_network_convert_data_to_tensor	Helper function to convert one layer's output raw data in binary format to data in tensor format with general buffers
ni_get_rc_txt	Get text string for the provided error
ni_param_get_key_value	Retrieve key and value from 'key=value' pair
ni_retrieve_xcoder_params	Retrieve encoder config parameter values
ni_retrieve_decoder_params	Retrieve decoder config parameter values
ni_calculate_sha256	Calculate the SHA256 value from data
ni_gettimeofday	Retrieve time for logs with microsecond timestamps
ni_gettime_ns	Retrieve the system clock time in nanoseconds
ni_usleep	Suspend execution for microsecond intervals
ni_posix_memalign	Allocate aligned memory
ni_pthread_mutex_init	Initialize a mutex which can be recursive.
ni_pthread_mutex_destroy	Destroy a mutex.
ni_pthread_mutex_lock	Thread mutex lock.
ni_pthread_mutex_unlock	Thread mutex unlock.
ni_pthread_create	Create a new thread.
ni_pthread_join	Join with a terminated thread.

API Functions in ni_util.h	Description
ni_pthread_cond_init	Initialize a condition variable.
ni_pthread_cond_destroy	Destroy a condition variable.
ni_pthread_cond_broadcast	Broadcast a condition.
ni_pthread_cond_wait	Wait on a condition.
ni_pthread_cond_signal	Signal a condition.
ni_pthread_cond_timedwait	Wait on a condition with timeout.
ni_pthread_sigmask	Examine and change mask of blocked signals.

ni_av_codec.h Functions

API Functions in ni_av_codec.h	Description
ni_should_send_sei_with_frame	Whether SEI should be sent together with this frame to encoder
ni_dec_retrieve_aux_data	Retrieve auxiliary data (close caption, various SEI) associated with this frame returned by decoder, convert them to appropriate format and save in the frame's auxiliary data storage for future use by encoding.
ni_enc_prep_aux_data	Prepare auxiliary data to send with this frame to the encoder based on the auxiliary data of the decoded frame.
ni_enc_copy_aux_data	Copy auxiliary data that should be sent together with this frame to encoder
ni_enc_insert_timecode	Insert time code info into picture timing SEI (for H264) or time code SEI (for H265) which is sent together with a frame to encoder
ni_enc_write_from_yuv_buffer	Send an input data frame to the encoder with YUV data given in the inputs.
ni_device_session_restart	Send a restart command after flush command. Only support Encoder now.
ni_dec_reconfig_ppu_params	Send a p_config command to reconfigure decoding ppu params.

5 libxcoder Integration

A set of example programs are provided to demonstrate the libxcoder API integration for video decoding, uploading, encoding and transcoding tasks. The source files are located in **libxcoder/source/examples**

When building libxcoder, the following example programs will be built from these source files, located in **libxcoder/build**:

- ni_xcoder_decode
- ni_xcoder_encode
- ni_xcoder_transcode_filter
- ni_xcoder_multithread_transcode

In the following sections, we use these source code as an example for writing our own application for video transcoding tasks, using the libxcoder API.

Note that the example programs are limited in their 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 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 example programs provide frame parsing for H.264, H.265, VP9 and elementary streams only. Customers who would want to use libxcoder for other codec format decoding must provide their own frame parsing logic.

5.1 List of Currently Unsupported Features in Example Programs

This is a list of features that the example programs do not currently support. Support for these could be added in the future if required.

- Audio
- Container, demuxer, muxer
- Custom user SEI
- Decoder/encoder engine reset/recovery
- Dolby Vision
- Decoder PPU output configuration
- Decoder multi-output receiving and splitting
- Semiplanar SW frame transcoding or decoding

5.2 Example Programs Command Line Options

```
ni_xcoder_decode:
Video decoding demo application directly using Netint Libxcoder API
Usage: ni_xcoder_decode [options]
options:
-h | --help                  Show this message.
-v | --version              Print version info.
-i | --input                Required) Input file path.
-o | --output               Required) Output file path.
-m | --dec-codec            Required) Decoder codec format.must match input file.
                            [a|avc, h|hevc, v|vp9]
-l | --loglevel            Set loglevel of this application and libxcoder API.
                            [none, fatal, error, info, debug, trace]
                            (Default: info)
-c | --card                Set card index to use.
                            See `ni_rsrc_mon` for info of cards on system.
                            (Default: 0)
-r | --repeat              To loop input X times. Must be a positive integer
                            (Default: 1)
-d | --decoder-params      Decoding params. See "Decoding Parameters" chapter in
                            QuadraIntegration&ProgrammingGuide*.pdf for help.
                            (Default: "")


ni_xcoder_encode:
Video encoding demo application directly using Netint Libxcoder API
Usage: ni_xcoder_encode [options]


options:
-h | --help                  Show this message.
-v | --version               Print version info.
```

```
-i | --input                (Required) Input file path.
                           Can be specified multiple (max 3) times
                           to concatenate inputs with different resolution
together (sequence change)
-o | --output               (Required) Output file path.
                           Can be specified multiple (max 4) times
                           to run multiple encoding instances simultaneously.
-m | --enc-codec             (Required) Encoder codec format.
                           [a|avc, h|hevc, x|av1, o|obu]
                   (x is in ivf container format, o is output raw AV1 OBU only)
-l | --loglevel             Set loglevel of this application and libxcoder API.
                           [none, fatal, error, info, debug, trace]
                           (Default: info)
-c | --card                 Set card index to use.
                           See `ni_rsrc_mon` for info of cards on system.
                           (Default: 0)
-r | --repeat               To loop input X times. Must be a positive integer
                           (Default: 1)
-k | --readframerate         Read input at specified frame rate.
-p | --pix_fmt              Indicate the pixel format of the input.
    [yuv420p, yuv420p10le, nv12, p010le, rgba, gbra, argb, abgr, bgr0, yuv444p]
                           (Default: yuv420p)
-s | --size                 (Required) Resolution of input file in format
WIDTHxHEIGHT.
                           (eg. '1920x1080')
-e | --encoder-params        Encoding params. See "Encoding Parameters" chapter
in
                           QuadraIntegration&ProgrammingGuide*.pdf for help.
                           Can be specified multiple (max 4) times,
                           must match the number of -o specified.
                           (Default: "")
```

```
-g | --encoder-gop        Custom GOP for encoding. See "Custom Gop Structure"
chapter in

                          QuadraIntegration&ProgrammingGuide*.pdf for help.

                          gopPresetIdx must be set to 0 to be in effect.

                          (Default: "")

-u | --hwupload           (No argument) When enabled upload raw frame to device
                           first before encoding

                          Multiple input files and yuv444p format input are not
                           supported in this mode
```

ni_xcoder_transcode_filter:

```
Video transcoding demo application directly using Netint Libxcoder API

Usage: ni_xcoder_transcode_filter [options]

options:

-h | --help                 Show this message.

-v | --version               Print version info.

-i | --input                (Required) Input file path.

-o | --output               (Required) Output file path.

                          Can be specified multiple (max 4) times

                          to run multiple encoding instances simultaneously.

-m | --dec-codec             (Required) Decoder codec format. Must match the
codec of input file.

                          [a|avc, h|hevc, v|vp9]

-n | --enc-codec          Required) Encoder codec format.

                          [a|avc, h|hevc, x|av1] (x is in ivf container format)

-l | --loglevel           Set loglevel of this application and libxcoder API.

                          [none, fatal, error, info, debug, trace]

                          (Default: info)

-c | --card               Set card index to use.

                          See `ni_rsrc_mon` for info of cards on system.

                          (Default: 0)

-r | --repeat             To loop input X times. Must be a positive integer
```

```
                          (Default: 1)

-d | --decoder-params        Decoding params. See "Decoding Parameters" chapter
in

                          QuadraIntegration&ProgrammingGuide*.pdf for help.

                          (Default: "")

-e | --encoder-params        Encoding params. See "Encoding Parameters" chapter
in

                          QuadraIntegration&ProgrammingGuide*.pdf for help.

                          Can be specified multiple (max 4) times,

                          must match the number of -o specified.

                          (Default: "")

-g | --encoder-gop         Custom GOP for encoding. See "Custom Gop Structure"
                           chapter in

                          QuadraIntegration&ProgrammingGuide*.pdf for help.

                          gopPresetIdx must be set to 0 to be in effect.

                          (Default: "")

-u | --user-data-sei-passthru  (No argument) Enable user data unregistered SEI
passthrough when specified

-f | --vf                  Video filter params. The only supported filters in
                           this demo are:

                          ni_quadra_scale - supported params [width, height,
                           format]

                          e.g.
      ni_quadra_scale=width=1280:height=720:format=yuv420p

                          ni_quadra_drawbox - supported params [x, y, width,
                           height]

                          e.g.
ni_quadra_drawbox=x=300:y=150:width=600:height=400

                          (Default: "")
```

ni_xcoder_multithread_transcode:

```
Multi-threaded video transcoding demo application directly using Netint
Libxcoder API

Usage: ni_xcoder_multithread_transcode [options]

options:
```

```
-h | --help                  Show this message.

-v | --version               Print version info.

-i | --input                 (Required) Input file path.

-o | --output                (Required) Output file path.
                           Can be specified multiple (max 4) times
                           to run multiple encoding instances simultaneously.

-m | --dec-codec             (Required) Decoder codec format. Must match the
codec of input file.
                           [a|avc, h|hevc, v|vp9]

-n | --enc-codec             (Required) Encoder codec format.
                           [a|avc, h|hevc, x|av1] (x is in ivf container format)

-l | --loglevel              Set loglevel of this application and libxcoder API.
                           [none, fatal, error, info, debug, trace]
                           (Default: info)

-c | --card                  Set card index to use.
                           See `ni_rsrc_mon` for info of cards on system.
                           (Default: 0)

-r | --repeat                To loop input X times. Must be a positive integer
                           (Default: 1)

-d | --decoder-params        Decoding params. See "Decoding Parameters" chapter
in
                           QuadraIntegration&ProgrammingGuide*.pdf for help.
                           (Default: "")

-e | --encoder-params        Encoding params. See "Encoding Parameters" chapter
in
                           QuadraIntegration&ProgrammingGuide*.pdf for help.
                           Can be specified multiple (max 4) times,
                           must match the number of -o specified.
                           (Default: "")

-g | --encoder-gop         Custom GOP for encoding. See "Custom Gop Structure"
                     chapter in QuadraIntegration&ProgrammingGuide*.pdf for
help.  gopPresetIdx must be set to 0 to be in effect.
```

```
                            (Default: "")

-u | --user-data-sei-passthru  (No argument) Enable user data unregistered SEI
                                  passthrough when specified

-f | --vf                      Video filter params. The only supported filters in
                               this demo are:

                               ni_quadra_scale - supported params[width,height,format

                               e.g.

ni_quadra_scale=width=1280:height=720:format=yuv420p

                               ni_quadra_drawbox - supported params [x, y, width,
                                height]

                               e.g.
ni_quadra_drawbox=x=300:y=150:width=600:height=400

(Default: "")
```

5.3 Transcoder Resource Allocation

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

The example programs operates on a single NETINT Quadra Video Processing Unit specified by the command line option **-c**, which is the same as the card index shown in the ***ni_rsrc_mon*** output.

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

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


ni_device_session_context_init(&dec_ctx);
ni_device_session_context_init(&enc_ctx);
```

Also for every opened session, a keep-alive heartbeat beats periodically. This keeps the channel open between the libxcoder and the device firmware. If there is no heartbeat from the libxcoder for a specified period of time, (default is 3 seconds, but is configurable by the libxcoder ***keepAliveTimeout*** parameter for both the decoder and encoder configuration), then the device firmware will assume that the application is terminated, and all current sessions will close. For Linux, MacOS and Android platforms, change the name of this thread to KAT+hw_id+session_ id, such as hw_id=0, session_id=0x3ef, thread name is KAT0003ef.

We use the transcoder card index and set it together with the source codec format, (where source is the destination for the encoder), in the session context. This specifies the transcoder to be used for the video coding task. The reservation and allocation of the video resource is automatically handled by the libxcoder at the session opening. This is shown in the following code snippet for the decoder (similar for encoder):

```
dec_ctx.codec_format = dec_codec_format;

dec_ctx.hw_id = iXcoderGUID;
```

We use simple transcoder selection in our example. The transcoder GUID (card index) is specified directly. Section 13.3 of the Integration & Programming Guide[1], describes other resource allocation methods such as - least loaded card. This will allow the libxcoder to automatically pick the least loaded card for use when multiple cards are available on the host, to do this simply set ***hw_id*** to value -1.

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

```
    // default: little endian

    dec_ctx.src_bit_depth = bit_depth;

    dec_ctx.src_endian = NI_FRAME_LITTLE_ENDIAN;

    dec_ctx.bit_depth_factor = 1;

    if (10 == dec_ctx.src_bit_depth)

    {

      dec_ctx.bit_depth_factor = 2;

    }


    ret = ni_device_session_open(&dec_ctx, NI_DEVICE_TYPE_DECODER);
```

Note : The session context also needs to have the **p_session_config** attribute set to appropriate values. For the decoder, the values are used for post processing and output format selection, etc. For the encoder, a number of values related to the encoding configuration need to be set. The decoder parameters and encoder parameters are described in detail in section 9.1 and 8.3 of the Integration and Programming Guide[1] respectively.

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

5.4 HW Frame for Raw Video Transfer

5.4.1 Terminology

The terms for software (SW) frames and hardware (HW) frames correspond to where the frame exists. Simply put, a SW frame is a raw video frame that exists on the host machine in its memory. A HW frame is a raw video frame that exists on the VPU/GPU device and can be referenced through the API.

5.4.2 Pathways and Features

On the host machine, SW frames can be modified and filtered or used freely. However HW frames can only be used in some circumstances. To generate an original HW frame, the uploading instance may be used for SW frame to HW frame conversion (section 5.9). Alternatively, the decoder can produce HW frames as an alternative to SW frame outputs with the decoding parameter:

"out=hw"

With the Quadra T1A, the accepted frame inputs to its functional features are detailed in Figure 3 below:

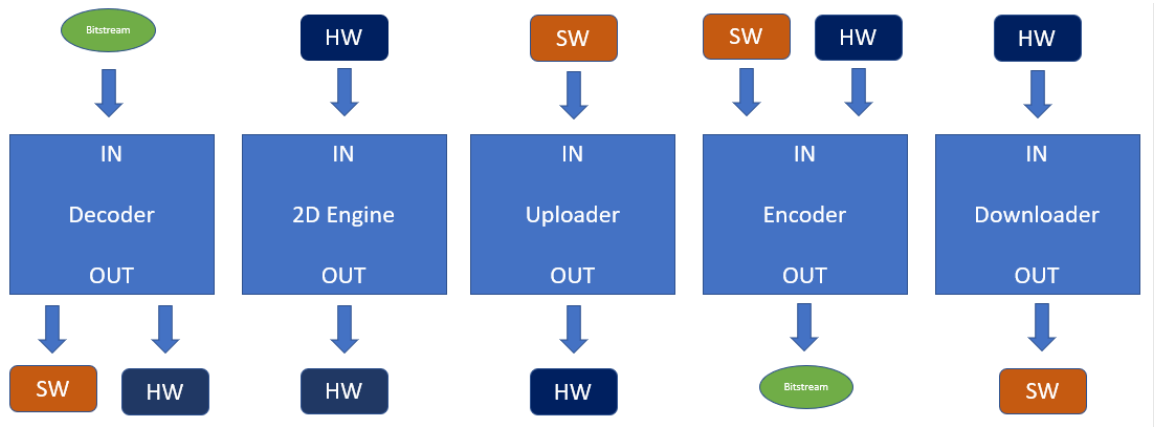


Figure 3 Input output mapping

The question, ***When is it better to use HW frames over SW frames?*** will now be discussed. The benefits for each depend on, whether the YUV needs to be modified off-device, encoding ladder use, available resources, and many other conditions.

In a transcoding scenario, if the intermediate YUV does not need any other processing besides encoding, then bypassing the hefty YUV transfer to the host by using HW frames is preferable and much faster.

HW frames are also more viable when multiple encoding ladders need to operate on the same set of decoded output. A single HW frame can be referenced by X number of encoders simultaneously, which means for the work of generating, sending and storing a single input, we can get X outputs. This can be demonstrated by running the following example HW upload and ladder encoding command:

ni_xcoder_encode -i test/akiyo_352x288p25.yuv -s 352x288 -p yuv420p -m a -u ***-e RcEnable=1:bitrate=20000000 -o lad1.264 -e RcEnable=1:bitrate=400000 -o lad2.264 -e RcEnable=1:bitrate=600000 -o lad3.264***

The above command performs a hardware upload of a given frame, then encode 3 different bitrates of that frame and receive the 3 outputs back on the host.

A downside of utilizing HW frames is that the frames used by the host must be tracked meticulously, they must also be recycled on time, in order to prevent any resource lock-ups or memory leakages during runtime. When a HW frame is no longer in use **ni_hwframe_buffer_recycle2** must be called to let the device know the buffer can be reused, and to free the descriptor on the host.

The structure used to store the details of a HW frame contains this:

```
typedef struct _niFrameSurface1
{
  uint16_t ui16FrameIdx;     //frame location on device
  uint16_t ui16session_ID;    //for instance tracking
  uint16_t ui16width;        // width on device
  uint16_t ui16height;       // height on device
  uint32_t ui32nodeAddress;   //currently not in use, formerly offset
  int32_t device_handle;     //handle to access device
  int8_t bit_depth;          //1 == 8bit per pixel, 2 == 10bit
  int8_t encoding_type;      //0 = semiplanar, 1 = semiplanar, 2 = tiled4x4
  int8_t output_idx;         // 0-2 for decoder output index
  int8_t src_cpu;           // frame origin location
  int32_t dma_buf_fd;        // P2P dma buffer file descriptor
} niFrameSurface1_t;
```

5.4.3 Limitations

Since the HW frame consumes real memory on the Quadra T1 card, there is a maximum limit to how many can co-exist simultaneously. Currently, the minimum bin size is enough to fit **half** a YUV420 8-bit frame at 1080p so any smaller resolution will still consume the same amount of video memory on device.

The **ni_rsrc_mon** tool can be used to check the currently consumed video memory under the “SHARE_MEM” data column. Note that SW frames will also consume this memory resource but its freeing is not controlled by the host application SW.

Another limitation is that all HW frames can only be used if the producer is from the same device. On a host machine with multiple Quadra T1 cards, if device 0 produces HW frame A, then device 2 cannot use HW frame A as an input directly. If the user should want device 2 to process a frame from device 0, the HW frame must first be downloaded and device 2 would work with a SW frame. The resulting SW frame is then available for upload onto device 2.

5.5 Decoding

5.5.1 Decoding Loop

The decoding process involves sending encoded bitstream packets to the decoder and receiving decoded YUV or HW 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 loop is as follows:

```
while (send_rc == NI_TEST_RETCODE_SUCCESS || receive_rc ==
NI_TEST_RETCODE_SUCCESS)

{

    // Sending

    send_rc = decoder_send_data(&ctx, &dec_ctx, &in_pkt, video_width,

                            video_height, p_stream_info);

    if (send_rc < 0)

    {

        ni_log(NI_LOG_ERROR,

               "Error: decoder_send_data() failed, rc: %d\n",

               send_rc);

        break;

    }


    // Receiving

    do

    {

        rx_size = 0;

        receive_rc = decoder_receive_data(&ctx, &dec_ctx, &out_frame,
video_width, video_height,

                                      output_fp, 1, &rx_size);


        if (dec_ctx.hw_action == NI_CODEC_HW_ENABLE)

        {

            if (receive_rc != NI_TEST_RETCODE_EAGAIN &&

                receive_rc != NI_TEST_RETCODE_END_OF_STREAM)

            {

                p_ni_frame = &out_frame.data.frame;

                p_hwframe = (niFrameSurface1_t *)p_ni_frame->p_data[3];
```

```
                ni_log(NI_LOG_DEBUG, "decoder_receive_data HW decode-only. recycle
HW frame idx %u\n", p_hwframe->ui16FrameIdx);

                ni_hwframe_buffer_recycle2(p_hwframe);

                ni_frame_buffer_free(p_ni_frame);

            }

        }

        else

        {

            ni_decoder_frame_buffer_free(&(out_frame.data.frame));

        }


        // Error or eos

        if (receive_rc < 0 || out_frame.data.frame.end_of_stream)

        {

            break;

        }


    } while (!dec_ctx.decoder_low_delay && rx_size > 0); //drain consecutive
outputs


    // Error or eos

    if (receive_rc < 0 || out_frame.data.frame.end_of_stream)

    {

        break;

    }

}
```

The bitstream packet sending involves calling libxcoder API function **ni_device_session_write**, while the YUV frame receiving **ni_device_session_read or ni_device_session_read_hwdesc**. Check the ***ni_xcoder_decode*** program source code for details on preparing the data for 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_packet_t** and **ni_frame_t** struct.

Libxcoder implements a memory buffer pool for storing YUV raw data retrieved from decoder. This mechanism reduces frequent memory allocation/release and speeds up processing and is the recommended way for decoder YUV retrieval. Libxcoder API functions **ni_decoder_frame_buffer_alloc** and **ni_decoder_frame_buffer_free** are provided for this purpose. Check the ***ni_xcoder_decode*** program for their example usage.

As previously detailed in Section 5.4.2, if HW frames are used, the memory footprint on the host is drastically reduced and we would use ni_frame_buffer_alloc and ni_frame_buffer_free as provided for a simpler allocation and freeing method. There is also a critical step for HW frames once the application knows a specific HW frame is no longer needed. As described in the bottom of section 5.4.2 “Pathways and Features”, the HW frame needs to be recirculated to the device with **ni_hwframe_buffer_recycle2.** Check the transcoding, HW upload and encoding parts of the example programs for their example usage.

5.5.1.1 Multithreading

Parts of the decoding cycle can be augmented with multithreading. For low instance counts or smaller resolutions, this often greatly improves performance. The ***ni_xcoder_multithread_transcode*** program demonstrates the multithreaded version of decoding. One thread would handle sending bitstream packets to the decoder, while the other will pull YUV outputs from the decoder.

When taking advantage of multithreading, it is still recommended to keep the decoder open and close in sync, especially when transcoding is involved. This means that for greater reliability, all decoder read/write should be completed before sending a session close.

An example of decoding send and receive thread is shown below:

Decoder Send

```
void *decoder_send_thread(void *args)
{
    dec_send_param_t *p_dec_send_param = args;
    ni_demo_context_t *p_ctx = p_dec_send_param->p_ctx;
    ni_session_context_t *p_dec_ctx = p_dec_send_param->p_dec_ctx;
    ni_session_data_io_t in_pkt = {0};
    int retval = 0;

    ni_log(NI_LOG_INFO, "decoder_send_thread start: decoder_low_delay %d\n",
          p_dec_ctx->decoder_low_delay);
    while (!p_ctx->end_all_threads)
    {
        retval = decoder_send_data(p_ctx, p_dec_ctx, &in_pkt, p_dec_send_param-
>input_width,
                                   p_dec_send_param->input_height, p_dec_send_param-
>p_stream_info);
        if (retval < 0)   // Error
        {
            ni_log(NI_LOG_ERROR, "Error: decoder send packet failed\n");
            break;
        } else if (p_dec_send_param->p_ctx->dec_eos_sent)   //eos
        {
            ni_log(NI_LOG_INFO, "decoder_send_thread reached eos\n");
            break;
        } else if (retval == NI_TEST_RETCODE_EAGAIN) {
            ni_usleep(100);
        }
    }
```

```
    ni_packet_buffer_free(&in_pkt.data.packet);

    // Broadcast all codec threads to quit on exception such as NVMe IO.
    if (retval < 0)
    {
        p_ctx->end_all_threads = 1;
    }

    ni_log(NI_LOG_TRACE, "decoder_send_thread exit\n");
    return (void *)(long)retval;
}
```

Decoder Receive

```
void *decoder_receive_thread(void *args)
{
    dec_recv_param_t *p_dec_recv_param = args;
    ni_demo_context_t *p_ctx = p_dec_recv_param->p_ctx;
    ni_session_context_t *p_dec_ctx = p_dec_recv_param->p_dec_ctx;
    ni_test_frame_list_t *frame_list = p_dec_recv_param->frame_list;
    ni_session_data_io_t *p_out_frame = NULL;
    ni_frame_t *p_ni_frame = NULL;
    niFrameSurface1_t *p_hwframe;
    int retval = 0;
    int rx_size = 0;

    ni_log(NI_LOG_INFO, "decoder_receive_thread start\n");

    for (;;)
    {
        while (frame_list_is_full(frame_list) && !p_ctx->end_all_threads)
        {
            ni_usleep(100);
        }

        if (p_ctx->end_all_threads)
        {
            break;
        }

        p_out_frame = &frame_list->frames[frame_list->tail];
        p_ni_frame = &p_out_frame->data.frame;
```

```
        retval = decoder_receive_data(
            p_ctx, p_dec_ctx, p_out_frame, p_dec_recv_param->input_width,
            p_dec_recv_param->input_height, NULL, 0 /* no save to file */,
&rx_size);
        if (retval < 0) // Error
        {
            if (!p_dec_ctx->hw_action)
            {
                ni_decoder_frame_buffer_free(p_ni_frame);
            } else
            {
                ni_frame_buffer_free(p_ni_frame);
            }
            ni_log(
                NI_LOG_ERROR,
                "Error: decoder_receive_thread break in transcode mode!\n");
            break;
        } else if (p_ni_frame->end_of_stream)
        {
            frame_list_enqueue(frame_list);
            ni_log(NI_LOG_INFO, "decoder_receive_thread reach eos\n");
            retval = 0;
            break;
        } else if (retval == NI_TEST_RETCODE_EAGAIN)
        {
            if (!p_dec_ctx->hw_action)
            {
                ni_decoder_frame_buffer_free(p_ni_frame);
            } else
            {
```

```
                ni_frame_buffer_free(p_ni_frame);
            }
            ni_usleep(100);
        } else
        {
            if (p_dec_ctx->hw_action)
            {
                uint16_t current_hwframe_index = ((niFrameSurface1_t *)p_ni_frame->p_data[3])->ui16FrameIdx;
                ni_log(NI_LOG_DEBUG, "decoder recv:%d, tail:%d\n", current_hwframe_index, frame_list->tail);
            }
            frame_list_enqueue(frame_list);
        }
    }

    ni_frame_buffer_free(&filter_out_frame.data.frame);

    // Broadcast all codec threads to quit on exception such as NVMe IO.
    if (retval < 0)
    {
        p_ctx->end_all_threads = 1;
    }

    ni_log(NI_LOG_TRACE, "decoder_receive_thread exit\n");
    return (void *)(long)retval;
}
```

5.5.2 Decoding Input

The NETINT Quadra decoder engine requires the bitstream to be passed in complete frames so libxcoder expects the same for its input. The example programs provides very simple H.264, H.265 and VP9 parsers to do the frame partitioning.

5.5.3 NETINT Pixel Formats

The NETINT Quadra card supports the following:

- YUV420P(8 and 10-bit) – Decoder output, 2D engine input/output, Upload input, Encoder input
- NV12 (8 and 10-bit) – Decoder output, 2D engine input/output, Upload input, Encoder input
- Tiled4x4 (8 and 10-bit) – Decoder HWframe output, Encoder input
- RGBA (2D engine only)

Note that 10 bit video uses 16 bit words in little endian format. For planar format, the ten least significant bits store the component value with the six most significant bits as don't care. For semi-planar format, the ten most significant bits store the component value and the six least significant bits are not used.

Figure 44 shows the NETINT 8 bit YUV format. The Y components are stored first followed by the U and then the V both subsampled by 2 in each direction. Each component is 8 bits.

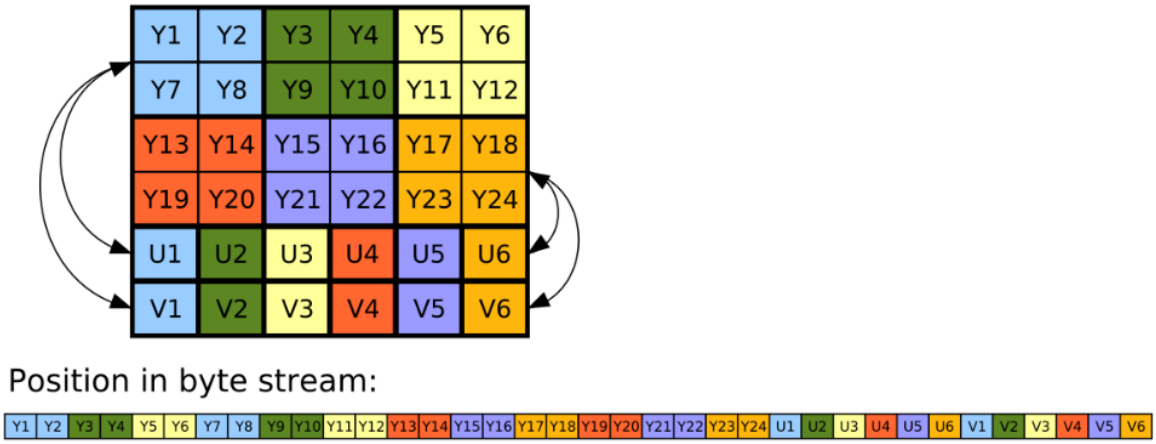


Figure 4 NETINT YUV420 8 bit Format

10 bit pixels are stored as 16 bits as shown in Figure 6.

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

5.5.4 YUV420 Alignment Requirements

The NETINT Quadra hardware places certain restrictions on the height and width of the YUV data. The picture width (stride) must be 128-byte aligned and height (in pixels) must be even. Picture sizes that do not meet these requirements must be padded before encoding. Minimum width and height requirements of 144 and 128 pixels respectively must also be met and aligned to previous rules. Decoded YUV frames will be output aligned.

2-Pass and AV1 encode input have extra restrictions which are described in the Quadra Integration & Programming Guide[1], section 8.4.5 “Encoder Limitations”. They may require extra input padding.

Figure 6 shows how the padding must be added to meet the alignment requirements. For example, to encode a 1916x1076 stream to HEVC, the width of the YUV picture must be padded to 1920 to be divisible by 128. The height can remain the same. Another example would be a 1200x720 picture would require horizontal padding equal to 80 bytes Luma and 40 bytes chroma for H264/H265 encoding.

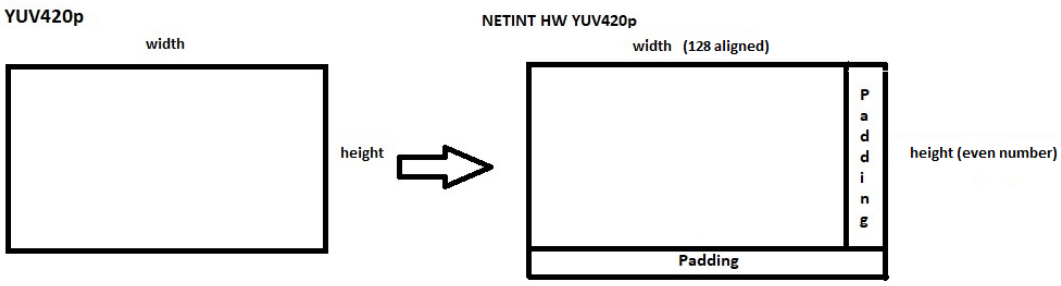


Figure 6 NETINT YUV420 Padding

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

```
typedef struct _ni_frame
{
  uint8_t *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_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 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).

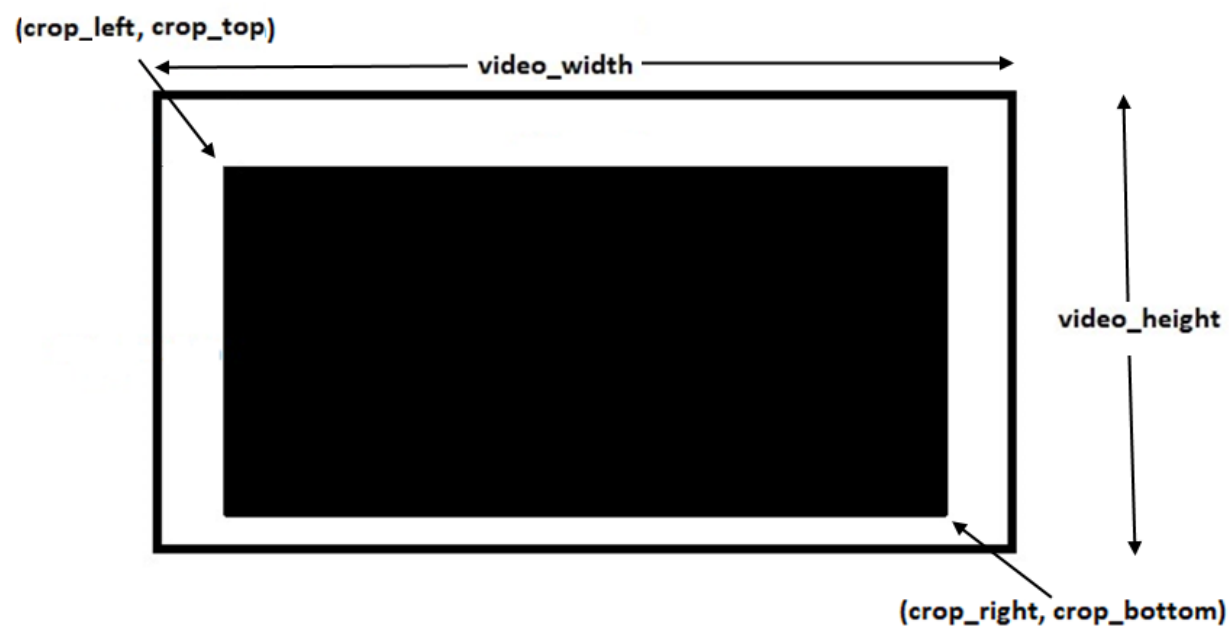


Figure 7 NETINT YUV420 Decoded Frame and Cropping Window

5.5.5 NETINT Decoder Output Data Layout

In addition to the YUV data (or HW descriptors), 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.

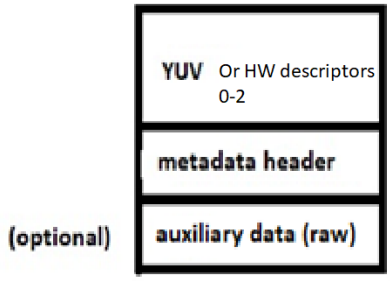


Figure 8 Decoder output layout

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

```
typedef struct _ni_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_metadata_common_t;
```

The cropping and resolution information is used to fill in ni_frame_t attributes presented in section “YUV Alignment Requirements”. Other information (shown as auxiliary data) from supported SEIs, if present, is then used to fill in the remaining attributes of ni_frame_t as follows:

```
typedef struct _ni_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_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. 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 caption data is formatted as per the T35 SEI payload data specified by CEA708[7]. The user data unregistered data includes the UUID and is formatted as per the H.264[5] and H.265[6] standards.

The static HDR metadata is defined in the following structures:

ni_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 data (used for encoding) are called **auxiliary data** in the libxcoder.

The following sections give more details 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.

5.5.6 Auxiliary data

Auxiliary data is information that accompanies a video frame. It is returned by the decoder as a result of the decoding process, e.g. a closed caption that comes with a decoded video frame. It may also 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 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.a
// 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: custom VUI adjustment, 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_VUI,

// 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,
```

```
  // NETINT: long term reference interval adjustment, which takes int32_t
  // type data as payload that indicates the new long term reference interval
  // value.
  NI_FRAME_AUX_DATA_LTR_INTERVAL,

  // NETINT: frame reference invalidation, which takes int32_t type data
  // as payload that indicates the frame number after which all references
  // shall be invalidated.
  NI_FRAME_AUX_DATA_INVALID_REF_FRAME,

  // NETINT: custom framerate adjustment, which takes int32_t type data as
  // payload that indicates the new target framerate numerator and denominator
values.
  NI_FRAME_AUX_DATA_FRAMERATE,

  // NETINT: custom maxFrameSize adjustment, which takes int32_t type data as
  // payload that indicates the new maxFrameSize value
  NI_FRAME_AUX_DATA_MAX_FRAME_SIZE,
} ni_aux_data_type_t;
```

Note that Quadra currently supports decoding a maximum of 1024 bytes of SEI payload.

5.5.6.1 Retrieval and Storage

The auxiliary data returned by the decoder is in a low level format ni_query_session_statistic_info and can be converted into a more useable format by calling the **ni_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 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_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_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.

5.5.6.2 User Data Unregistered SEI

The ***ni_xcoder_transcode_filter*** and ***ni_xcoder_multithread_transcode*** programs with "-u" parameter enables User Data Unregistered SEI passthrough. If the parameter is enabled and the User Data Unregistered SEI is available, the SEI is retrieved and then 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[5] and H.265[6] 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*

5.5.6.3 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[7] 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

5.5.6.4 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;
```

5.5.6.5 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).
  uint16_t max_cll;

  // Max average light level per frame (cd/m^2).
  uint16_t max_fall;
} ni_content_light_level_t;
```

5.5.6.6 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;
```

5.5.7 Decoder Sequence Change

A sequence change is a change in picture size or bit depth of the input bitstream while decoding. To support this with the NETINT decoder, the application must check for differences in the output frame resolution/depth and reallocate frame buffers as needed. The example programs does not currently support this.

5.6 Encoding

5.6.1 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. Alternatively, the HW frame generated by decoder, uploader, or 2D engine can also be accepted as input. It should be noted that the input format may not mix so once encoder gets a raw YUV input, all following inputs must follow that format and the same goes for HW descriptor inputs.

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 loop is as follows:

```
while (!end_of_all_streams &&
    (send_rc == NI_TEST_RETCODE_SUCCESS || receive_rc ==
NI_TEST_RETCODE_SUCCESS))
{
    read_size = read_yuv_from_file(&ctx, input_fp[i_index], yuv_buf,
                            video_width[i_index], video_height[i_index],
                            pix_fmt, sw_pix_fmt, &eos,
                            enc_ctx[0].session_run_state);
    if (read_size < 0)
    {
        break;
    }

    for (i = 0; i < output_total; i++)
    {
        ctx.curr_enc_index = i;
        send_rc = encoder_send_data(&ctx, &enc_ctx[i],
                                &in_frame, eos ? NULL : yuv_buf,
                                video_width[i_index], video_height[i_index],
                                i_index == input_total - 1);

        if (send_rc == NI_TEST_RETCODE_EAGAIN)
        {
            // retry send to same encoder session
            i--;
            continue;
        } else if (send_rc == NI_TEST_RETCODE_NEXT_INPUT)
        {
```

```
            // next input (will trigger sequence change)
             i_index++;
            ctx.total_file_size = get_total_file_size(input_fp[i_index]);
            ctx.curr_file_offset = 0;
            send_rc = NI_TEST_RETCODE_SUCCESS;
        }
    }

    receive_rc = encoder_receive(&ctx, enc_ctx, &in_frame, out_packet,
                                 video_width[0], video_height[0],
                                   output_total, output_fp);
    for (i = 0; receive_rc >= 0 && i < output_total; i++)
    {
        if (!ctx.enc_eos_received[i])
        {
            ni_log(NI_LOG_DEBUG, "enc %d continues to read!\n", i);
            end_of_all_streams = 0;
            break;
        } else
        {
            ni_log(NI_LOG_DEBUG, "enc %d eos !\n", i);
            end_of_all_streams = 1;
        }
    }
}
```

Again the data sending and receiving involve calling libxcoder API function **ni_device_session_write** and **ni_device_session_read**. See the ***ni_xcoder_encode*** program 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_frame_t** and **ni_packet_t** struct.

```
typedef struct _ni_packet
{
  long long dts;
  long long pts;
  long long pos;
  uint32_t end_of_stream;
  uint32_t start_of_stream;
  uint32_t video_width;
  uint32_t video_height;
  uint32_t frame_type; // encoding only 0=I, 1=P, 2=B
  int recycle_index;
  void* p_data;
  uint32_t data_len;
  int sent_size;

  void* p_buffer;
  uint32_t buffer_size;
  uint32_t avg_frame_qp; // average frame QP reported by VPU
  uint8_t *av1_p_buffer[MAX_AV1_ENCODER_GOP_NUM];
  uint8_t *av1_p_data[MAX_AV1_ENCODER_GOP_NUM];
  uint32_t av1_buffer_size[MAX_AV1_ENCODER_GOP_NUM];
  uint32_t av1_data_len[MAX_AV1_ENCODER_GOP_NUM];
  int av1_buffer_index;
```

```
  int av1_show_frame;
  int flags;   // flags of demuxed packet
  ni_custom_sei_set_t *p_custom_sei_set;
  double psnr_y;
  double psnr_u;
  double psnr_v;
  double average_psnr;
  double ssim_y;
  double ssim_u;
  double ssim_v;
} ni_packet_t;
```

5.6.1.1 Multithreading

Parts of the encoding cycle can be augmented with multithreading. For low instance counts or smaller resolutions, this often greatly improves performance. The ***ni_xcoder_multithread_transcode*** program demonstrates the multithreaded version of the encoding. One thread would handle sending YUV frames to the encoder, while the other will pull compressed packets from the encoder.

When taking advantage of multithreading, it is still recommended to keep the encoder open and close in sync, especially when transcoding is involved. This means that for greater reliability, all encoder read/write should be completed before sending a session close.

An example of encoding send and receive thread is shown below:

Encoder Send

```
void *encoder_send_thread(void *args)
{
    enc_send_param_t *p_enc_send_param = args;
    ni_demo_context_t *p_ctx = p_enc_send_param->p_ctx;
    ni_session_context_t *p_enc_ctx_list = p_enc_send_param->p_enc_ctx;
    ni_test_frame_list_t *frame_list = p_enc_send_param->frame_list;
    ni_session_data_io_t *p_dec_frame = NULL;
    ni_session_data_io_t enc_in_frame = {0};
    ni_frame_t *p_ni_frame = NULL;
    niFrameSurface1_t *p_surface;
    int i, ret = 0;
    ni_log(NI_LOG_INFO, "%s start\n", __func__);
    for (;;)
    {
        while (frame_list_is_empty(frame_list) && !p_ctx->end_all_threads)
        {
            ni_usleep(100);
        }
        if (p_ctx->end_all_threads)
        {
            break;
        }
        p_dec_frame = &frame_list->frames[frame_list->head];
        p_ni_frame = &p_dec_frame->data.frame;
        for (i = 0; i < p_enc_send_param->output_total; i++)
        {
            p_ctx->curr_enc_index = i;
```

```
        ret = encoder_send_data2(p_ctx, &p_enc_ctx_list[i], p_dec_frame,
                                 &enc_in_frame,
                                 p_enc_send_param->output_width,
                                 p_enc_send_param->output_height);
        if (ret < 0)   //Error
        {
            if (p_enc_ctx_list[i].hw_action)
            {
                //pre close cleanup will clear it out
                p_surface = (niFrameSurface1_t *)p_ni_frame->p_data[3];
                ni_hw_frame_ref(p_surface);
            } else
            {
                ni_decoder_frame_buffer_free(p_ni_frame);
            }
            frame_list_drain(frame_list);
            ni_log(NI_LOG_ERROR, "Error: encoder send frame failed\n");
            break;
        } else if (ret == NI_TEST_RETCODE_EAGAIN) {
            ni_usleep(100);
            i--;  //resend frame
            continue;
        } else if (p_enc_ctx_list[0].hw_action && !p_ctx->enc_eos_sent[i])
        {
            p_surface = (niFrameSurface1_t *)p_ni_frame->p_data[3];
            ni_hw_frame_ref(p_surface);
        }
    }
    if (ret < 0)
    {
```

```
            break;
        } else if (p_enc_ctx_list[0].hw_action)
        {
            ni_frame_wipe_aux_data(p_ni_frame);   //reuse buffer
        } else
        {
            ni_decoder_frame_buffer_free(p_ni_frame);
        }
        frame_list_drain(frame_list);

        if (p_enc_send_param->p_ctx->enc_eos_sent[0])   // eos
        {
            ni_log(NI_LOG_INFO, "%s EOS sent\n", __func__);
            break;
        }
    }

    ni_frame_buffer_free(&enc_in_frame.data.frame);

    // Broadcast all codec threads to quit on exception such as NVMe IO
    if (ret < 0)
    {
        p_ctx->end_all_threads = 1;
    }

    ni_log(NI_LOG_TRACE, "%s exit\n", __func__);
    return (void *)(long)ret;
}
```

Encoder Receive

```
void *encoder_receive_thread(void *args)
{
    enc_recv_param_t *p_enc_recv_param = args;
    ni_demo_context_t *p_ctx = p_enc_recv_param->p_ctx;
    ni_session_context_t *p_enc_ctx = p_enc_recv_param->p_enc_ctx;
    ni_session_data_io_t out_packet[MAX_OUTPUT_FILES] = {0};
    int i, ret = 0;
    int end_of_all_streams = 0;

    ni_log(NI_LOG_INFO, "encoder_receive_thread start\n");

    while (!end_of_all_streams && ret >= 0 && !p_ctx->end_all_threads)
    {
        ret = encoder_receive(p_ctx, p_enc_ctx,
                              &p_enc_recv_param->frame_list-
>frames[p_enc_recv_param->frame_list->head],
                              out_packet, p_enc_recv_param->output_width,
                              p_enc_recv_param->output_height, p_enc_recv_param-
>output_total,
                              p_enc_recv_param->p_file);
        for (i = 0; ret >= 0 && i < p_enc_recv_param->output_total; i++)
        {
            if (!p_ctx->enc_eos_received[i])
            {
                ni_log(NI_LOG_DEBUG, "enc %d continues to read!\n", i);
                end_of_all_streams = 0;
                break;
            } else
            {
```

```
                ni_log(NI_LOG_DEBUG, "enc %d eos !\n", i);
                end_of_all_streams = 1;
            }
        }
    }

    for (i = 0; i < p_enc_recv_param->output_total; i++)
    {
        ni_packet_buffer_free(&out_packet[i].data.packet);
    }

    // Broadcast all codec threads to quit on exception such as NVMe IO.
    if (ret < 0)
    {
        p_ctx->end_all_threads = 1;
    }

    ni_log(NI_LOG_TRACE, "%s exit\n", __func__);
    return (void *)(long)ret;
}
```

5.6.2 Input YUV Frame Preparation

As described in section 5.5.3, the NETINT Quadra encoder accepts only even width and height pictures, along with some extra resolution required for AV1.

The following is a quick summary of the requirements:

- Horizontal strides must be even
- Vertical alignment must be even
- AV1
    - Width must be 8-pixel aligned
    - Height must be 8-pixel aligned
        - NOTE - non 8-pixel aligned input can be encoded, but the output will be cropped
    - Width <= 4096 pixels
    - Height <= 4352 pixels
    - Width * height <= (4096*2304) pixels
- H265 and AV1 2-pass encoding
    - Input resolution width and height aligned to 32-pixels is preferred for H265 and AV1 2-pass encoding. If input resolution is not 32-pixels aligned, there may be minor quality degradation.

The following is a list of libxcoder API functions for input YUV frame preparation:

- **ni_encoder_frame_zerocopy_check** and **ni_encoder_frame_zerocopy_buffer_alloc** in ni_device_api.c check if zero copy is applicable to the input frame, and prepare input frame for zero copy when applicable. When zero copy is enabled, libxcoder transfers data directly from input frame buffer(s) (luma / chroma buffers can be inconsecutive) to device, which improves performance when encoding high resolution / large frames. Please refer to sample code in section 5.6.2.1 “Zero Copy” to incorporate zero copy feature.

  NOTE – Zero copy is only applicable to input frames which meet the following requirements

  - YUV420 8 or 10 bit with planar or semi-planar pixel format, or RGBA / BGRA / ABGR / ARGB pixel format
  - Input frame width and height are 2-pixels aligned
  - Input frame linesizes (aka strides) of luma / chroma buffers are 2-bytes aligned
  - Input frame resolution >= 144x128
  - Each input frame’s linesize shall be the same throughout the entire encoding session

  NOTE – when ni_encoder_frame_zerocopy_check checks if zero copy is applicable, the decision is also affected by encoder parameter zeroCopyMode, please refer to Integration Programming Guide section 8.4 “Encoding Parameters” for the descriptions regarding this parameter

- If zero copy is not applicable (or disabled by zeroCopyMode), **ni_get_hw_yuv420p_dim**, **ni_encoder_sw_frame_buffer_alloc** and **ni_copy_hw_yuv420p** in ni_util.h calculate linesize, allocate internal buffer, and copy YUV data from input frame buffer(s) to internal buffer, before sending data to the encoder
- The **ni_copy_yuv_444p_to_420p** function can convert SW yuv444p data into two pieces of HW yuv420p frames. 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.h header file and ***ni_xcoder_encode*** program source for their sample usage

5.6.2.1 Zero Copy

The application should incorporate the following steps to utilize the zero copy feature

- Before opening the encoder session, call **ni_encoder_frame_zerocopy_check** with the input frame width, height, and linesize (aka stride).

- Please also note that the set_linesize parameter must be set to ***true***, this allows libxcoder to configure the linesize for this encoder session.

  Example:

Zero Copy (session open example)

```
// config linesize for zero copy (if input resolution is zero copy compatible)
ni_encoder_frame_zerocopy_check(p_enc_ctx, p_enc_params, width, height,
(const int *)src_stride, true);
// open encoder session
ret = ni_device_session_open(p_enc_ctx, NI_DEVICE_TYPE_ENCODER);
```

- Before sending each frame to the encoder, call **ni_encoder_frame_zerocopy_check** with input frame width, height, and linesize (aka stride). Please also note set_linesize parameter must be set to ***false***, so that libxcoder will compare the linesize with the previously configured linesize, in order to decide if the zero copy is applicable to the input frame

- If zero copy is applicable to the input frame then call **ni_encoder_frame_zerocopy_buffer_alloc** with the input frame width, height, linesize (aka stride), pointers to each data buffer (p_src in the example), and also the metadata length (extra_data_len in the example). See the following example.

Zero Copy (send frame example)

```
    // NOTE – Setup array of pointers to each of the luma and chroma data buffers.
FFmpeg / Gstreamer users should use the data pointer array in frame structure instead
of p_src in the following sample code
    if (isrgba)
    {
        src_stride[1] = 0;
        src_stride[2] = 0;

        src_height[1] = 0;
        src_height[2] = 0;

        p_src[1] = NULL;
        p_src[2] = NULL;
    }
    else
    {
        src_stride[1] = is_semiplanar ? src_stride[0] : src_stride[0] / 2;
        src_stride[2] = is_semiplanar ? 0 : src_stride[0] / 2;

        src_height[1] = src_height[0] / 2;
        src_height[2] = is_semiplanar ? 0 : src_height[1];
```

```
            p_src[1] = p_src[0] + src_stride[0] * src_height[0];

            p_src[2] = p_src[1] + src_stride[1] * src_height[1];

        }

        // check input resolution zero copy compatible or not

        if (ni_encoder_frame_zerocopy_check(p_ctx, p_param, frame_width,
frame_height, (const int *)src_stride, false) == NI_RETCODE_SUCCESS)

{

        need_copy = false;

        //alloc metadata buffer etc. (if needed)

        retval = ni_encoder_frame_zerocopy_buffer_alloc(p_enc_frame, frame_width,
frame_height, (const int *)src_stride, (const uint8_t **)p_src, (int)(p_enc_frame-
>extra_data_len));

        if (retval != NI_RETCODE_SUCCESS)

                goto end;

}
```

- Please also refer to the following files / functions for the complete sample code
    - FFmpeg libavcodec/nienc.c xcoder_send_frame()

      Or,

    - libxcoder/source/examples/common/ni_encode_utils.c encoder_open_session()
    - libxcoder/source/ni_av_codec.c ni_enc_write_from_yuv_buffer()

5.6.3 Encoder HW Descriptor Input and Recycling

As described earlier in section 5.4, the encoder may receive HW descriptors as an alternative to large raw YUV inputs. Prerequisite to opening the encoder, the input HW frame must already be available as ui16FrameIdx and device_handle in the HW frame structure is required to properly configure the memory buffers and determine co-location. Encoder opening preparation specific to HW frames will look like this:

```
ni_session_data_io_t out_frame = {0};
decoder_receive_data(
             &dec_ctx, &out_frame, output_video_width, output_video_height,
             p_file, &total_bytes_received, print_time, 0, &xcodeState);
enc_ctx.hw_action = NI_CODEC_HW_ENABLE;
api_param.hwframes = 1;
//to determine if from same device and buffer dimensions in memory
//needs to be done where input frame is available to check
enc_ctx.sender_handle =
    ((niFrameSurface1_t *)((uint8_t *)out_frame.data.frame
                           .p_data[3]))->device_handle;
api_param.rootBufId =
    ((niFrameSurface1_t *)((uint8_t *)out_frame.data.frame
                           .p_data[3]))->ui16FrameIdx;
```

When the descriptor is written to the encoder with **ni_device_session_write** the frame must *not* be recycled immediately. Since this input is currently being referenced internally by the encoder, recycling the input early will cause possible memory corruption as that frame can be overwritten by other entities.

The frame will be ready for recycling upon parsing the output bitstream packet for a non "-1" value on the recycle_index. This index should match the one provided during the input phase and the entire **niFrameSurface1_t** can be passed with the encoder handle to recycle and free the memory with **ni_hwframe_buffer_recycle2**.

A simple example is as follows:

```
niFrameSurface1_t hwframe_pool_tracker[NI_MAX_HWDESC_FRAME_INDEX];

memset(hwframe_pool_tracker, 0, NI_MAX_HWDESC_FRAME_INDEX *
sizeof(niFrameSurface1_t));

.

.

.

send_rc = encoder_send_data2(&ctx, &enc_ctx, &out_frame, &enc_in_frame,

                             output_width, output_height);


if (enc_ctx.hw_action)

{   //track in array with unique index, free when enc read finds

    //this must be implemented in application space for complete

    //tracking of hwframes

    //If encoder write had no buffer avail space the next time we

    //update the tracker will be redundant

    uint16_t current_hwframe_index =

        ((niFrameSurface1_t *)((uint8_t *)

                    out_frame.data.frame.p_data[3]))->ui16FrameIdx;

    memcpy(&hwframe_pool_tracker[current_hwframe_index],

          (uint8_t *)out_frame.data.frame.p_data[3],

          sizeof(niFrameSurface1_t));

    ni_frame_wipe_aux_data(&(out_frame.data.frame));   //reusebuff

} else

{

    ni_decoder_frame_buffer_free(&(out_frame.data.frame));
```

```
}
// encoded bitstream Receiving
encode_recv:
prev_num_pkt = ctx.num_of_packets_received;


receive_fin_flag = encoder_receive_data(
    &ctx, &enc_ctx, &out_packet, output_width, output_height,
    p_file, &enc_in_frame);


if (prev_num_pkt < ctx.num_of_packets_received &&   //skip if nothing read
    enc_ctx.hw_action && out_packet.data.packet.recycle_index > 0 &&
    out_packet.data.packet.recycle_index <
        NI_MAX_HWDESC_FRAME_INDEX)   //encoder only returns valid recycle index
//when there's something to recycle. This range is suitable for all memory bins
{
    if (hwframe_pool_tracker[out_packet.data.packet.recycle_index]
            .ui16FrameIdx)
    {
        ni_hwframe_buffer_recycle2(
            &hwframe_pool_tracker[out_packet.data.packet
                                  .recycle_index]);
    }
}
```

5.6.4 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.h:

```
typedef struct _ni_encoder_cfg_params
{
  …
  // ConformanceWindowOffsets
  int conf_win_top;
  int conf_win_bottom;
  int conf_win_left;
  int conf_win_right;
} ni_encoder_cfg_params_t;


typedef struct _ni_xcoder_params
{
 …
  union
    {
        ni_encoder_cfg_params_t cfg_enc_params;
        ni_decoder_input_params_t dec_input_params;
    };
…
} ni_xcoder_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_MIN_WIDTH x NI_MIN_HEIGHT, i.e. 144 x 128), is implemented in **`ni_get_hw_yuv420p_dim`** and **ni_copy_hw_yuv420p**

For example, to encode to an H.264 128x128 stream, the encoding configuration, api_param.source_width is set to 144 (adjusted from 128->144 to meet minimum width), api_param.source_height 128, conf_win_top = conf_win_left = conf_win_bottom = 0, conf_win_right = 16.

5.6.5 Encoding Formats and Parameters

The encoding formats and a number of encoding parameters provided by the NETINT encoder have been explained in detail in Quadra Integration & Programming Guide[1], section 8.3.

The example programs support the encoding parameters by using a colon separated list of key/value pairs, like the following:

ni_xcoder_encode -c 0 -i input.yuv -o output.265 -s 1920x1080 -p yuv420p -m h -e vbvBufferSize=10:RcEnable=1:bitrate=1000000

Note, when encoding to av1, ni_xcoder_encode can output to ivf (-m x) or obu (-m o) format.

Also note that multiple output and encoding parameters can be specified for a demo of ladder encoding. An example command is shown in section 0.

5.6.5.1 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:

ni_xcoder_transcode_filter -c 0 -i 4kHDR10+.264 -o 4k.xcoder.265 -m a -n h -e ***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.h. Please see the H.265[6] and H.264[5] standards for more details on the VUI.

5.6.5.2 Low Delay Encoding

The encoder can be configured to function in a low latency mode, such that libxcoder 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 section 13.1.3 in the Integration Programming Guide.

To enable this feature, a low delay GOP (where all frames are in sequence) such as gopPresetIdx=9 must be used when lowDelay is enabled, as shown in the following command:

ni_xcoder_transcode_filter -c 0 -i input.264 -o output.265 -m a -n h -e ***gopPresetIdx=9:lowDelay=1:RcEnable=1:bitrate=4000000***

The following FFmpeg and *xcoder* commands would produce identical results using low delay encoding modes:

ffmpeg -vsync 0 -hide_banner -s:v 352x288 -i ../libxcoder/test/akiyo_352x288p25.yuv -c:v h264_ni_enc -xcoder-params lowDelay=1:gopPresetIdx=9:lookAheadDepth=0:

=0:intraQpDelta=3:cuLevelRcEnable=1:vbvBufferSize=33:RcEnable=1:bitrate=1000000 aki-ffmpeg.264 -y

ni_xcoder_encode -i ../libxcoder/test/akiyo_352x288p25.yuv -o aki-xcoder.264 -m a -s 352x288 -p yuv420p -e lowDelay=1:gopPresetIdx=9:lookAheadDepth=0:intraPeriod=0:intraQpDelta=3:cuLevelRcEnable=1:vbvBufferSize=33:RcEnable=1:bitrate=1000000:frameRate=25:colorPri=0:colorTrc=0:colorSpc=0

Since the first packet returned by the encoder contains only the bitstream headers, the application must initially read 2 packets to retrieve the first encoded frame as shown in the following code from **encoder_receive_data** function in **ni_encode_utils.c**:

```
receive_data:
  rc = ni_device_session_read(p_enc_ctx, p_out_data,
                                      NI_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))
    {
      ni_log(NI_LOG_ERROR, "Writing data %d bytes error !\n",
             p_out_pkt->data_len - meta_size);
      ni_log(NI_LOG_ERROR, "ferror rc = %d\n", ferror(p_file));
    }


    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;
    }
```

5.6.5.3 Custom Gop Encoding

The encoder can be configured to use a custom gop structure using a separated list of key=value parameters that can be specified with "--encoder-gop" (or "-g"), where gopPresetIdx must set to 0. Usage is detailed in the Quadra Integration and Programming Guide[1], section 8.4.3.1.

The following sample command uses the specified custom gop structure:

ni_xcoder_encode -i test/akiyo_352x288p25.yuv -s 352x288 -p yuv420p -m a -e RcEnable=1:bitrate=20000000:gopPresetIdx=0 ***-g customGopSize=2:g0pocOffset=1:g0QpOffset=0:g0temporalId=0:g0picType=2:g0numRefPics=2:g0refPic0=-1:g0refPic0Used=1:g0refPic1=-3:g0refPic1Used=1:g1pocOffset=2:g1QpOffset=0:g1temporalId=0:g1picType=2:g1numRefPics=2:g1refPic0=-1:g1refPic0Used=1:g1refPic1=-2:g1refPic1Used=1*** *-o lad1.264*

For ladder encoding, “--xcoder-gop” (or “-g”) can be specified multiple times to apply to each encoder output.

The following xcoder command uses the custom gop structures specified for *lad1.264* and *lad2.264* and the default gop structure for *lad3.264*:

ni_xcoder_encode -i test/akiyo_352x288p25.yuv -s 352x288 -p yuv420p -m a -e RcEnable=1:bitrate=20000000:gopPresetIdx=0 ***-g customGopSize=2:g0pocOffset=1:g0QpOffset=0:g0temporalId=0:g0picType=2:g0numRefPics=2:g0refPic0=-1:g0refPic0Used=1:g0refPic1=-3:g0refPic1Used=1:g1pocOffset=2:g1QpOffset=0:g1temporalId=0:g1picType=2:g1numRefPics=2:g1refPic0=-1:g1refPic0Used=1:g1refPic1=-2:g1refPic1Used=1*** *-o lad1.264 -e RcEnable=1:bitrate=400000:gopPresetIdx=0* ***-g customGopSize=4:g0pocOffset=4:g0QpOffset=0:g0temporalId=0:g0picType=1:g0numRefPics=1:g0refPic0=-4:g0refPic0Used=1:g1pocOffset=2:g1QpOffset=0:g1temporalId=0:g1picType=2:g1numR***

efPics=2:g1refPic0=-2:g1refPic0Used=1:g1refPic1=2:g1refPic1Used=1:g2pocOffset=1:g2QpOffset=0:g2temporalId=0:g2picType=2:g2numRefPics=3:g2refPic0=-1:g2refPic0Used=1:g2refPic1=1:g2refPic1Used=1:g2refPic2=3:g2refPic2Used=0:g3pocOffset=3:g3QpOffset=0:g3temporalId=0:g3picType=2:g3numRefPics=2:g3refPic0=-1:g3refPic0Used=1:g3refPic1=1:g3refPic1Used=1 *-o lad2.264 -e RcEnable=1:bitrate=600000 -o lad3.264*

5.6.6 Encoder Sequence Change

A sequence change is a change in picture size or bit depth of the input YUV while encoding. NETINT encoder provides 2 sequence change methods:

method 1 - If changing from small to large resolution, the application must close the current encoding session and open a new encoding session with large resolution

method 2 - If changing from large to small resolution, the application can call sequence change API to re-config encoding session (without closing / opening session), which has lower latency compared to method 1.

The *ni_xcoder_encode* program supports encoder sequence change when multiple (up to 3) input files with different picture size are specified. The following command would produce one output bitstream containing 3 sequences with resolution 720p -> 1080p -> 720p:

ni_xcoder_encode -c 0 -p yuv420p -i Plaza_1280x720p30_300.yuv -s 1280x720 -i Dinner_1920x1080p30_300.yuv -s 1920x1080 -i Plaza_1280x720p30_300.yuv -s 1280x720 -o xcoder_sequence_change.h264 -m a -e intraPeriod=120

(NOTE – input files in the example above are not included in release package)

In the example above, the *ni_xcoder_encode* program uses method 1 for the first sequence change with the resolution 720p -> 1080p, and uses method 2 for the second sequence change with resolution change 1080p -> 720p.

The following describes the encoder sequence change procedures:

1. Detect sequence change and send end-of-stream to trigger encoder flush.
   By comparing input frame resolution with the resolution of current encoder session, sequence change is detected.

   Then the application should send end-of-stream to trigger an encoder flush. This is done automatically in the **ni_enc_write_from_yuv_buffer** API from **ni_av_codec.h**, as shown in the following code:

```
send_data:
   if (!p_in_frame->start_of_stream &&
       (p_in_frame->video_width != p_enc_ctx->actual_video_width ||
        p_in_frame->video_height != p_enc_ctx->active_video_height ||
        p_in_frame->pixel_format != p_enc_ctx->pixel_format))
   {
      ni_log(NI_LOG_INFO,
            "encoder_send_data: resolution change %dx%d "
            "-> %dx%d or pixel format change %d -> %d\n",
            p_enc_ctx->actual_video_width, p_enc_ctx->active_video_height,
            p_in_frame->video_width, p_in_frame->video_height,
            p_enc_ctx->pixel_format, p_in_frame->pixel_format);
      // change run state
      p_enc_ctx->session_run_state =
         SESSION_RUN_STATE_SEQ_CHANGE_DRAINING;
      ni_log(NI_LOG_DEBUG,
            "encoder_send_data: session_run_state change to %d \n",
            p_enc_ctx->session_run_state);
      // queue frame not needed because frame info stored in p_in_frame
      // (customer application may queue frame here)
      // send EOS
      p_in_frame->end_of_stream = 1;
      goto send_frame;
   }
```

2. Repeat receive_data until all encoder flush completes
   Repeat receiving output data from encoder until encoder flush completes. Then the application should close / open session (method 1) or re-config session (method 2), as shown in the following code from **encoder_receive_data** function in **ni_encode_utils.c**:

```
receive_data:
    rc = ni_device_session_read(p_enc_ctx, p_out_data,
                                    NI_DEVICE_TYPE_ENCODER);


    // end_of_stream flag indicates encoder flush completion
    end_flag = p_out_pkt->end_of_stream;
    rx_size = rc;


    if (rx_size > meta_size)
    {
        // received output from encoder
    } else if (rx_size != 0)
    {
        // received error
    } else if (!end_flag &&
              ((ni_xcoder_params_t *)(p_enc_ctx->p_session_config))
                 ->low_delay_mode)
    {
        // temporarily no output from encoder, continue receive if low delay mode
    } else
    {
        // no output from encoder
        if (end_flag)
        {
```

```
          // encoder flush complete, close / open or re-config session
          if (SESSION_RUN_STATE_SEQ_CHANGE_DRAINING ==
             p_enc_ctx->session_run_state)
          {
            // after sequence change completes, reset codec state
            ni_log(NI_LOG_INFO, "encoder_receive_data: sequence "
                  "change completed, return SEQ_CHANGE_DONE and will reopen "
                  "or reconfig codec!\n");

            rc = encoder_reinit_session(p_enc_ctx, p_in_data, p_out_data);
            ni_log(NI_LOG_TRACE, "encoder_receive_data: encoder_reinit_session
ret %d\n", rc);
            if (rc == NI_RETCODE_SUCCESS)
            {
              return NI_TEST_RETCODE_SEQ_CHANGE_DONE;
            }
            else
            {
              return NI_TEST_RETCODE_FAILURE;
            }
          }
       }
    }
```

3. Close / open session (method 1) or re-config session (method 2)

The application should compare original and new resolutions to decide whether to close / open session (method 1) or re-config session (method 2), as shown in the following code from **ni_encode_utils.c**:

```
int encoder_reinit_session(ni_session_context_t *p_enc_ctx,
                     ni_session_data_io_t *p_in_data,
                     ni_session_data_io_t *p_out_data)
{
   // check resolution change from large to small or small to large, plus other
condition checks - please refer to sample code
  // method 1 - close / open session if new resolution > original resolution
   if ((ori_stride*p_enc_ctx->ori_height < new_stride*new_height) ||
       (p_enc_ctx->ori_pix_fmt != p_buffered_frame->pixel_format) ||
       bIsSmallPicture || (p_enc_ctx->codec_format == NI_CODEC_FORMAT_JPEG)) {
       ni_log(NI_LOG_INFO, "XCoder encode sequence change by close / re-open
session\n");
       encoder_close_session(p_enc_ctx, p_in_data, p_out_data);
       ret = encoder_open_session(p_enc_ctx, p_enc_ctx->codec_format,
                          p_enc_ctx->hw_id, p_api_param, new_width,
                          new_height, new_pix_fmt, true);
       if (NI_RETCODE_SUCCESS != ret)
       {
          ni_log(NI_LOG_ERROR, "Failed to Re-open Encoder Session upon Sequence
Change (status = %d)\n", ret);
          return ret;
       }
       p_out_data->data.packet.end_of_stream = 0;
       p_in_data->data.frame.start_of_stream = 1;
       // clear crop parameters upon sequence change because cropping values may
not be compatible to new resolution
       p_api_param->cfg_enc_params.crop_width = 0;
```

```
        p_api_param->cfg_enc_params.crop_height = 0;
       p_api_param->cfg_enc_params.hor_offset = 0;
        p_api_param->cfg_enc_params.ver_offset = 0;
    }
    else {
    // method 2 - fast sequence change by re-config session (without close / open) only if new resolution < original resolution
     ni_log(NI_LOG_INFO, "XCoder encode sequence change by re-config session (fast path)\n");
     ret = encoder_sequence_change(p_enc_ctx, p_in_data, p_out_data, new_width, new_height, new_bit_depth_factor);
  }
  // set run state
  p_enc_ctx->session_run_state = SESSION_RUN_STATE_SEQ_CHANGE_OPENING; // this state is referenced when sending first frame after sequence change
  ni_log(NI_LOG_DEBUG, "encoder_reinit_session:session_run_state change %d \n",
      p_enc_ctx->session_run_state);
    return ret;
}
```

4. Send first frame with new resolution

The application can now send first frame with new resolution after close / open (method 1) or re-config session (method 2) to new resolution, as shown in the following code from **encoder_send_data** function in **ni_encode_utils.c**:

```
oneSent = ni_enc_write_from_yuv_buffer(p_enc_ctx, p_in_frame, yuv_buf);


if (oneSent < 0)
{
    // send error
} else if (oneSent == 0 && !p_enc_ctx->ready_to_close)
{
    // temporarily no input buffer available in encoder, continue send
} else
{
    // sent first frame with new resolution successfully, reset run state
    if (p_enc_ctx->session_run_state == SESSION_RUN_STATE_SEQ_CHANGE_OPENING)
    {
     p_enc_ctx->session_run_state = SESSION_RUN_STATE_NORMAL;


     ni_log(NI_LOG_DEBUG,
            "encoder_send_data: session_run_state change to %d \n",
            p_enc_ctx->session_run_state);
    }
}
```

5.6.7 NETINT Encoder Input Data Layout

As with the decoder, the YUV frame to be encoded may have auxiliary data as described in section 5.5.5. The layout of the encoder input data is show in Figure 12.

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_metadata_enc_frame
{
  ni_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;
  uint16_t              start_len[3];
  uint8_t               inconsecutive_transfer;
  uint8_t               reserved;
} ni_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.

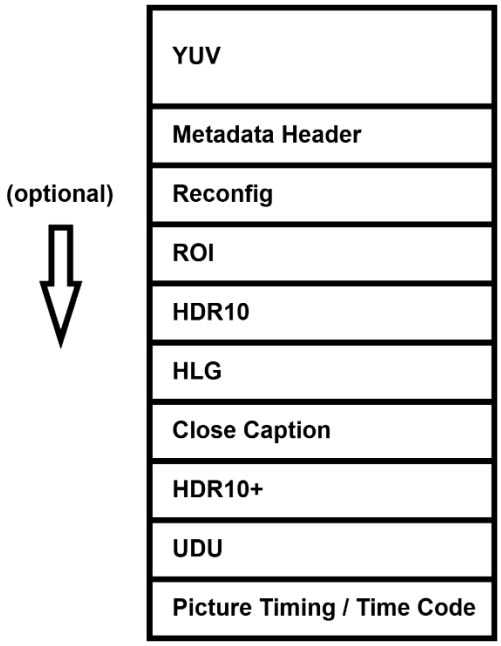


Figure 12 Encoder input layout

The libxcoder API provides a couple of functions to assist in preparing the encoder data, **ni_enc_prep_aux_data** and **ni_enc_copy_aux_data.** These functions convert and copy the auxiliary data into the format required by the encoder, so they must be called for a frame before it is sent to encoder for encoding.

For software frame encoding, the libxcoder API provides the **ni_enc_write_from_yuv_buffer** function. Given YUV data in a caller allocated buffer, this function prepares an encoder input data frame in the layout described above and sends the input frame to the encoder. See the **ni_xcoder_encode** program for an example of how this function can be used.

The retrieval and storage of auxiliary data returned by the decoder is described in section 5.5.6.1.

All SEI data (User data unregistered, HDR10, HDR10+, close caption, HLG preferred characteristics, picture timing / time code) must be 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. Check the syntax definition in the H.264[5] and H.265[6] standards for more detail. The construction of these NALs is hidden from the libxcoder API user and is implemented in the functions **ni_enc_prep_aux_data**, **ni_enc_copy_aux_data** and **ni_enc_insert_timecode**.

Note that the Quadra Tx 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.

5.6.7.1 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.

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

```
ctx->api_fme.data.frame.ni_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_pict_type = PIC_TYPE_IDR;
    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_pict_type = 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 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_pict_type
if (1 == p_ctx->force_frame_type || p_frame->force_key_frame)
{
  if (p_frame->ni_pict_type)
  {
    p_meta->frame_force_type_enable = 1;
    p_meta->frame_force_type = p_frame->ni_pict_type;
  }
}
```

5.6.7.2 ROI

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

- The feature must be enabled by the libxcoder 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.
- NOTE – ROI takes no effects for 2-pass (lookaheadDepth > 0) encode

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

5.6.7.2.1 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 the Integration Programming Guide, section 10.1.7. The coding details of processing a list of rectangular ROI regions are in libxcoder API function **set_roi_map** in file ni_av_codec.c. This method is the most complicated but permits non-rectangular ROI regions to be defined if required.

5.6.7.2.2 libxcoder ROI Struct

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

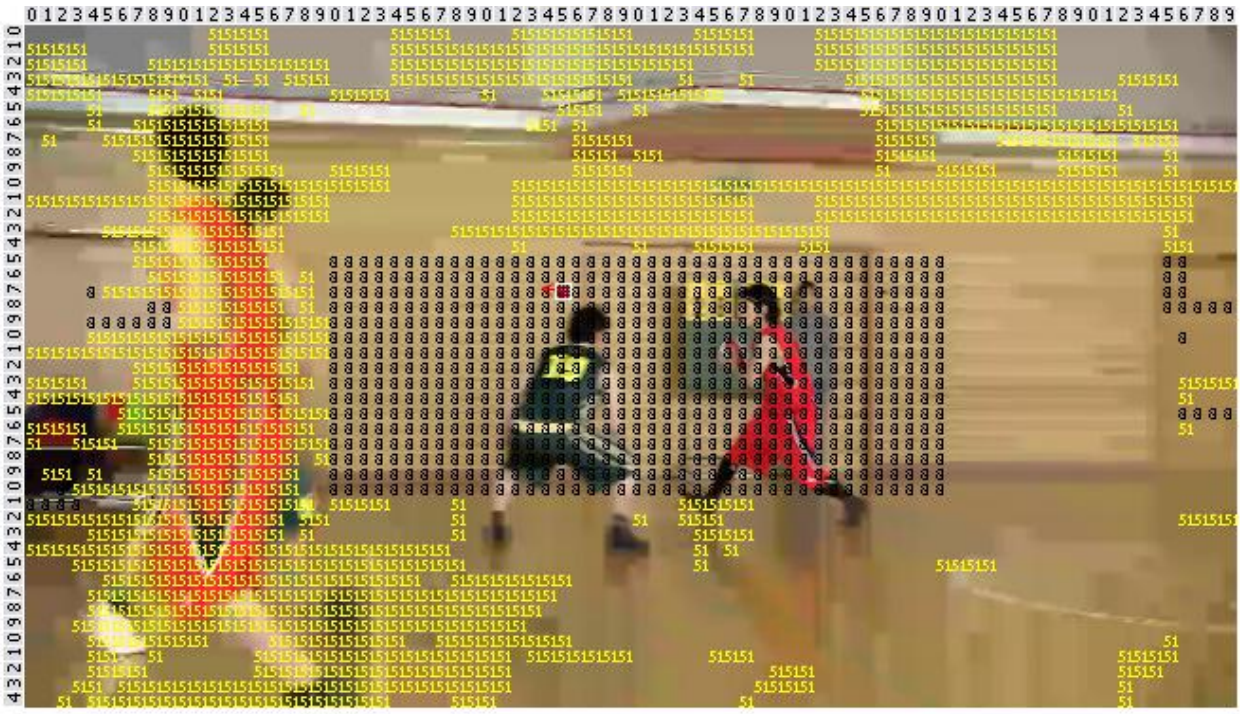
```
// struct describing a Region Of Interest (ROI)
typedef struct _ni_region_of_interest
{
  // self size: must be set to: sizeof(ni_region_of_interest_t)
  uint32_t self_size;

  // 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;

  // quantization offset: [-1, +1], 0 means no quality change; < 0 value asks
  // for better quality (less quantisation), > 0 value asks for worse quality
  // (greater quantization).
  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 defines 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_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 64x64 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).

5.6.7.2.3 libxcoder *cacheRoi* Encode Option

The libxcoder ROI API is designed to specify the ROI on a single frame and needs to be specified for each frame. There is a libxcoder 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 command example, assuming that ROI maps have been set and applied appropriately:

ni_xcoder_transcode_filter -c 0 -i 1280x720p_Basketball.264 -o output.265 -m a -n h -e ***roiEnable=1:cacheRoi=1***

5.6.7.3 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:

- Bitrate
- Intra Period
- VUI
- Long Term Reference (LTR) frame setting
- Long Term Reference (LTR) interval setting
- Frame reference invalidation
- Framerate
- MaxFrameSize parameter setting (for details regarding maxFrameSize parameter, please refer to Integration Programming Guide)
- Max & Min qp setting
- CRF
- CRF Float
- VBVBufferSize & VBVMaxRate
- MaxFrameSizeRatio
- SliceArg

Encoder reconfiguration is not currently supported with JPEG encoding.

The Long Term Reference (LTR) feature has been explained in detail in Quadra Integration & Programming Guide[1], section 8.3.1.

The examples of LTR reconfiguration feature are presented in section 5.6.7.7.2. The same reconfiguration tasks can be accomplished by using the wrapper API without providing a frame.

The following is an example of bitrate reconfiguration using auxiliary data attached to a frame being sent to encoder.

Libxcoder 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_frame_new_aux_data(frame, NI_FRAME_AUX_DATA_BITRATE,
                                 sizeof(int32_t));


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

5.6.7.4 HLG Preferred Transfer Characteristics

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

5.6.7.5 User Data Unregistered SEI

Handling of User Data Unregistered (UDU) SEI is done automatically by libxcoder. 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 APPS520[3] for more details.

5.6.7.6 Picture Timing / Time Code SEI

The **ni_enc_insert_timecode** function (from ni_av_codec.h) provides an interface for inserting arbitrary timecode data into the picture timing SEI (for H264) or time code SEI (for H265) in each frame. The timecode data shall be given to this function in a caller-allocated ni_timecode_t struct. This function must be called after **ni_enc_prep_aux_data** and **ni_enc_copy_aux_data** have handled all the other types of aux data so that the picture timing / time code SEI will be appended to the end of the buffer being sent to encoder.

When using H264 encoder, the enableTimecode encoder configuration parameter must also be enabled to set the pic_struct_present flag in VUI. Otherwise, the timecode data will not be recognized in the encoded stream. See “Encoding Parameters” in the Integration Programming Guide for more information on the enableTimecode option.

The following code provides an example of inserting timecode data using

ni_enc_insert_timecode:

```
// Make sure ni_enc_prep_aux_data and ni_enc_copy_aux_data
// are called before ni_enc_insert_timecode so that the
// frame data layout is correct
ni_enc_prep_aux_data(
       p_enc_ctx, p_enc_frame, p_dec_frame,
       p_enc_ctx->codec_format, should_send_sei_with_frame,
        mdcv_data, cll_data, cc_data, udu_data, hdrp_data);
ni_enc_copy_aux_data(p_enc_ctx, p_enc_frame, p_dec_frame,
       p_enc_ctx->codec_format, mdcv_data, cll_data,
        cc_data, udu_data, hdrp_data, is_hwframe,
        is_semiplanar);
// Allcoate ni_timecode_t and set the fields in it with the
// desired values
ni_timecode_t timecode = {0};
timecode.full_timestamp_flag = 1;
timecode.seconds_value = 42;
timecode.minutes_value = 19;
timecode.hours_value = 7;
. . . (Set all other fields as needed)
// Call the API to format and append the data to the frame
// buffer being sent to the encoder
ni_enc_insert_timecode(p_enc_ctx, p_in_frame, &timecode);
```

5.6.7.7 Long Term Reference Frame

The Long Term Reference (LTR) feature preserves 1 or 2 frames in DPB for long term reference, so that these LTR frames can be used as alternative references when reference invalidation is required.

The following libxcoder APIs allow application to set any frame as LTR, or change LTR interval during run-time.

Prerequisites - To use LTR feature, the following steps are required:

- LTR feature must be enabled by libxcoder encode option **longTermReferenceEnable=1**;
- LTR information needs to be supplied with the desired YUV frame, in the form of auxiliary data.
- (Optional) libxoder encode option **longTermReferenceInterval**=<#> for periodic LTR. If 0, Quadra will not periodically set frame as LTR except for the first frame
    - If **longTermReferenceInterval** > 0, setting current frame as long term reference overrides the interval

5.6.7.7.1 Set Current Frame as LTR

The following struct is required to set current frame as LTR:

```
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;

  // ni_device / no effect - this field is no longer in effect since encoding frame automatically refers to long term reference when short term reference frame(s) invalidated
  uint8_t use_long_term_ref;
} ni_long_term_ref_t;
```

Libxcoder provides two methods for application to choose from - (1) create frame auxiliary data or (2) call wrapper API

Method 1 – Create frame auxiliary data

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_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 = 1;
}
```

Method 2 – Wrapper API

Wrapper API also exists which can set the current frame as LTR. The API is showcased in the example programs in function **prep_reconf_demo_data** (from **ni_encode_utils.c**) as follows:

```
case XCODER_TEST_RECONF_LTR_API:
if (p_enc_ctx->frame_num ==
    api_param->reconf_hash[g_reconfigCount][0])
{
    ni_long_term_ref_t ltr;
    ltr.use_cur_src_as_long_term_pic =
       (uint8_t)api_param->reconf_hash[g_reconfigCount][1];
    ltr.use_long_term_ref =
       (uint8_t)api_param->reconf_hash[g_reconfigCount][2];
    ni_set_ltr(p_enc_ctx, &ltr);
    ni_log(NI_LOG_DEBUG, "%s(): frame #%lu API set LTR\n",
           __func__, p_enc_ctx->frame_num);
    p_ctx->reconfig_count++;
}
break;
```

5.6.7.7.2 Change LTR Interval during run-time

Libxcoder provides two methods for application to choose from - (1) create frame auxiliary data or (2) call wrapper API

Method 1 – Create frame auxiliary data

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_frame_new_aux_data(
    frame, NI_FRAME_AUX_DATA_LTR_INTERVAL,
    sizeof(int32_t));

if (aux_data)
{
 *((int32_t *)aux_data->data) = ..; // new LTR interval
}
```

Method 2 – Wrapper API

Wrapper API also exists which can set the current frame as LTR. The API is showcased in the example programs in function **prep_reconf_demo_data** (from **ni_encode_utils.c**) as follows:

```
case XCODER_TEST_RECONF_LTR_INTERVAL_API:
if (p_enc_ctx->frame_num ==
   api_param->reconf_hash[g_reconfigCount][0])
{
    ni_set_ltr_interval(p_enc_ctx,
       api_param->reconf_hash[g_reconfigCount][1]);
    ni_log(NI_LOG_DEBUG, "%s(): frame #%lu API set LTR
interval %d\n",
          __func__, p_enc_ctx->frame_num,
          api_param->reconf_hash[g_reconfigCount][1]);
    p_ctx->reconfig_count++;
}
break;
```

5.6.7.8 *Reference Invalidation*

Reference invalidation is used by the receiver of the bitstream to invalidate and change the encoding frames' references when the received packet is corrupted (and therefore cannot be referenced).

Libxcoder provides two methods for application to choose from - (1) create frame auxiliary data or (2) call wrapper API

Method 1 – Create frame auxiliary data

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_frame_new_aux_data(
    frame, NI_FRAME_AUX_DATA_INVALID_REF_FRAME,
    sizeof(int32_t));

if (aux_data)
{
 *((int32_t *)aux_data->data) = ..; // frame number to
invalidate
}
```

Method 2 – Wrapper API

Wrapper API also exists which can invalidate reference. The API is showcased in the example programs in function **prep_reconf_demo_data** (from **ni_encode_utils.c**) as follows:

```
case XCODER_TEST_INVALID_REF_FRAME_API:
if (p_enc_ctx->frame_num ==
    api_param->reconf_hash[g_reconfigCount][0])
{
    ni_set_frame_ref_invalid(p_enc_ctx,
        api_param->reconf_hash[g_reconfigCount][1]);
    ni_log(NI_LOG_DEBUG,
        "%s(): frame #%lu API set frame ref invalid %d\n",
        __func__, p_enc_ctx->frame_num,
        api_param->reconf_hash[g_reconfigCount][1]);
    p_ctx->reconfig_count++;
}
break;
```

5.6.7.9 Max&Min qp reconfig

Libxcoder provides two methods for application to choose from – (1) create frame auxiliary data or (2) call wrapper API.

Method 1-Create frame auxiliary data

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

```
aux_data = ni_frame_new_aux_data(&dec_frame,
          NI_FRAME_AUX_DATA_MAX_MIN_QP,
          sizeof(ni_rc_min_max_qp));
if (aux_data) {
     ni_rc_min_max_qp *qp_info = (ni_rc_min_max_qp
*)aux_data->data;
     qp_info->minQpI     = …;
     qp_info->maxQpI     = …;
     qp_info->maxDeltaQp = …;
     qp_info->minQpPB    = …;
     qp_info->maxQpPB    = …;
}
```

Method 2 – Wrapper API

Wrapper API also exists which can reconfig QP. The API is showcased in the example programs in function **prep_reconf_demo_data** (from **ni_encode_utils.c**) as follows:

```
case XCODER_TEST_RECONF_RC_MIN_MAX_QP_API:
if (ctx->api_ctx.frame_num ==
     p_param->reconf_hash[p_ctx->reconfig_count][0]) {
     ni_rc_min_max_qp qp_info;
     qp_info.minQpI = …;
     qp_info.maxQpI = …;
     qp_info.maxDeltaQp = …;
     qp_info.minQpPB = …;
     qp_info.maxQpPB = …;
     ni_reconfig_min_max_qp(p_enc_ctx, &qp_info);
     p_ctx->reconfig_count++;
}
break;
```

5.6.8 Wrapper API

In the previous sections, it is shown that auxiliary data can be used to pass on to encoder information for encoding. The auxiliary data is usually attached to a frame and sent to encoder together with the frame data. Sometimes user needs to send such information to encoder without providing a frame. Wrapper API functions are provided to accomplish this such that the required information will be sent to encoder with the next frame sending out.

The available API functions are:

- Bitrate reconfiguration: ni_reconfig_bitrate
- Intra period reconfiguration: ni_reconfig_intraprd
- VUI reconfiguration: ni_reconf_vui
- IDR frame forcing: ni_force_idr_frame_type
- Long Term Reference frame setting: ni_set_ltr
- Long Term Reference interval setting: ni_set_ltr_interval
- Frame reference invalidation: ni_set_frame_ref_invalid
- Framerate reconfiguration: ni_reconfig_framerate
- maxFrameSize parameter setting reconfiguration: ni_reconfig_max_frame_size (for details regarding **maxFrameSize** parameter, please refer to Integration Programming Guide)
- Max&Min qp setting: ni_reconfig_min_max_qp
- CRF setting: ni_reconfig_crf and ni_reconfig_crf2 (for details regarding **crf** parameter and its reconfiguration, please refer to Integration Programming Guide section 8.4 “Encoding Parameters” for **crf** and **ReconfDemoMode** descriptions)
- VBV value setting: ni_reconfig_vbv_value
- maxFrameSizeRatio parameter setting reconfiguration: ni_reconfig_max_frame_size_ratio (for details regarding **maxFrameSizeRatio** parameter, please refer to Integration Programming Guide)
- SliceArg setting: ni_reconfig_slice_arg

5.6.9 NETINT Encoder Output PSNR and SSIM

Encoder can be configured to output PSNR and SSIM by the following xcoder parameters –

PSNR: **getPsnrMode** and **intervalOfPsnr**

SSIM: **enableSSIM**

Please refer to Quadra Integration and Programming Guide document Section 8.3 “Encoding Parameters” for detailed descriptions about the usages and limitations of the parameters listed above

When PSNR and/or SSIM is enabled, application may retrieve each encoded frame’s PSRN / SSIM via ni_packet_t struct fields `psnr_y, psnr_u, psnr_v, average_psnr, ssim_y, ssim_u, ssim_v.`

Please refer to Section 5.6.1 “Encoding loop” in this document for more descriptions about ni_packet_t struct.

5.7 Scaling

5.7.1 Scaler Demo

A scaler demo is included as part of the **ni_xcoder_transcode_filter** and **ni_xcoder_multithread_transcode** programs to demonstrate how to launch a scaler operation with the libxcoder API. It supports two types of operations – scale and drawbox. These are implemented in the same fashion as the video filters in ffmpeg (ni_quadra_scale, ni_quadra_drawbox) in that they are inserted into the transcoding pipeline and transforms the frame between decoder and encoder.

The following are sample commands for running the scale and drawbox filter demo:

ni_xcoder_transcode_filter -i input.264 -o output.265 -m a -n h -d out=hw ***-f "ni_quadra_scale=width=1280:height=720:format=yuv420p"***

This command scales the input video to 1280x720 after decoding and transforms the pixel format to yuv420p if it's not already in this format. Then the scaled frames are sent encoder for encoding.

ni_xcoder_multithread_transcode -i input.264 -o output.265 -m a -n h -d out=hw ***-f "ni_quadra_drawbox=x=300:y=150:width=600:height=400"***

This command draws a box that's 600 pixels wide and 400 pixels high, with its top left corner located at 300 pixels from the left border and 150 pixels from the top border. The frames are not scaled so the output video resolution will be the same as the input.

Note that the input frame to the scaler must be a hardware frame, this is because the 2D engine only accepts hardware frames. To make the decoder output a hardware frame, "**-d out=hw**" must be specified, as shown in the above commands. When a decoded frame is received, the application needs to make sure that the HW frame buffer of the decoded

frame is properly recycled once scaler output is read, since the scaler output will use another HW frame buffer. The following code snippet from ni_xcoder_transcode_filter shows this behavior:

```
if (scale_params.enabled)
{
    p_hwframe = (niFrameSurface1_t *)out_frame.data.frame.p_data[3];
    ni_hw_frame_ref(p_hwframe);
    scale_filter(&sca_ctx, &out_frame.data.frame, &filter_out_frame,
                 xcoderGUID, scale_params.width, scale_params.height,
                 ni_to_gc620_pix_fmt(dec_ctx.pixel_format),
                 scale_params.format);
    ni_hw_frame_unref(p_hwframe->ui16FrameIdx);  //Recycle decoder output
    frame_to_enc = &filter_out_frame;  //Send scaler output to encoder
}
else if (drawbox_params.enabled)
{
    p_hwframe = (niFrameSurface1_t *)out_frame.data.frame.p_data[3];
    ni_hw_frame_ref(p_hwframe);
    drawbox_filter(&crop_ctx, &pad_ctx, &ovly_ctx, &fmt_ctx,
                   &out_frame.data.frame, &filter_out_frame,
                  &drawbox_params, xcoderGUID,
                   ni_to_gc620_pix_fmt(dec_ctx.pixel_format), GC620_I420);
    ni_hw_frame_unref(p_hwframe->ui16FrameIdx);  //Recycle decoder output
    frame_to_enc = &filter_out_frame;  //Send scaler output to encoder
}
```

5.7.2 Launch a Scaler Operation

Supported scaler operations include SCALE, CROP, PAD, OVERLAY, STACK, and ROTATE. The parameters needed for launching a scaler operation are packed in ni_scaler_input_params_t. Different operations will require different parameters. See init_scaler_params() to learn which parameters are fixed and which parameters can be set by the user for the current operation. After setting the parameters, call launch_scaler_operation():

```
int launch_scaler_operation(ni_session_context_t *p_ctx, int iXcoderGUID,
                        ni_frame_t *p_frame_in_up,
                        ni_frame_t *p_frame_in_bg,
                        ni_session_data_io_t *p_data_out,
                        ni_scaler_input_params_t scaler_params)
{
    // p_frame_in_up and p_frame_in_bg should be the same except it's an
overlay operation
    int ret = 0;
    if (p_ctx->session_id == NI_INVALID_SESSION_ID)
    {
       // If session is not opened yet, open a scaler session
        if (0 != scaler_session_open(p_ctx, iXcoderGUID, scaler_params.op))
      return -1;

        // init scaler hwframe pool where output frames of scaler is put
        if (0 != ni_scaler_frame_pool_alloc(p_ctx, scaler_params))
            return -1;
        }

    // allocate a ni_frame_t structure on the host PC
    if (0 != ni_frame_buffer_alloc_hwenc(&(p_data_out->data.frame),
```

```
                                scaler_params.output_width,
                                scaler_params.output_height, 0))
    { return -1;}
    niFrameSurface1_t *frame_surface_up;
    // input frame to be scaled (overlay frame in overlay case)
    frame_surface_up = (niFrameSurface1_t *)((p_frame_in_up->p_data[3]));
    // allocate scaler input frame
    if (0 != ni_scaler_input_frame_alloc(p_ctx, scaler_params,
        frame_surface_up))
    { return -1;}
    niFrameSurface1_t *frame_surface_bg;
    // input frame to be scaled (main/background frame in overlay case)
    frame_surface_bg = (niFrameSurface1_t *)((p_frame_in_bg->p_data[3]));
    // Allocate scaler destination frame.
    if (0 != ni_scaler_dest_frame_alloc(p_ctx, scaler_params,
        frame_surface_bg))
    { return -1;}


    // Retrieve hardware frame info from 2D engine and put it in the
ni_frame_t structure.
    ret = ni_device_session_read_hwdesc(p_ctx, p_data_out,
          NI_DEVICE_TYPE_SCALER);
    if (ret < 0)
    { // free the input and output frame once failed
       ni_frame_buffer_free(p_frame_in_up);
       ni_frame_buffer_free(p_frame_in_bg);
       ni_frame_buffer_free(&(p_data_out->data.frame));
    }
    return ret;
}
```

Generally, before launching a scaler operation, the hardware frame pool should be initialized. An input hardware frame and destination hardware frame should be allocated. Finally call ni_device_session_read_hwdesc() to get the desired output from the scaler.

5.8 Transcoding

The transcoding process connects decode to encode in a pipeline: the output of the decoder (YUV raw data or HW frame descriptor, and auxiliary data) is fed to the encoder as input. Also, any SEIs received by the decoder will be passed to the encoder. A transcoding loop is like follows:

```
ni_session_data_io_t in_pkt = {0};
ni_session_data_io_t out_frame = {0};
ni_session_data_io_t enc_in_frame = {0};
ni_session_data_io_t out_packet = {0};
while (!end_of_all_streams &&
    (send_rc == NI_TEST_RETCODE_SUCCESS ||
     receive_rc == NI_TEST_RETCODE_SUCCESS))
{
    // Sending bitstream packets to decoder
decode_send:
    send_rc = decoder_send_data(&ctx, &dec_ctx, &in_pkt,
                        input_width, input_height, p_stream_info);
    if (send_rc < 0)
    {
        ni_log(NI_LOG_ERROR, "Error: decoder send packet failed\n");
        ret = send_rc;
        break;
    }

    // YUV Receive
    receive_rc = decoder_receive_data(&ctx, &dec_ctx, &out_frame, input_width,
                                    input_height, NULL, 0, &rx_size);
    frame_to_enc = &out_frame;
```

```
    if (receive_rc < 0)
    {
        ni_log(NI_LOG_ERROR, "Error: decoder receive frame failed\n");
        ret = receive_rc;
        break;
    }
    else if (receive_rc == NI_TEST_RETCODE_EAGAIN)
    {
        if (!dec_ctx.hw_action)
        {
            ni_decoder_frame_buffer_free(&(out_frame.data.frame));
        } else
        {
            ni_frame_buffer_free(&(out_frame.data.frame));
        }

        // use first encode config low delay flag for call flow
        if (p_enc_api_param[0].low_delay_mode <= 0 && encoder_opened)
        {
            ni_log(NI_LOG_DEBUG, "no decoder output, jump to encoder
receive!\n");
            goto encode_recv;
        } else
        {
            ni_log(NI_LOG_DEBUG, "no decoder output, encode low_delay, jump to
decoder send!\n");
            goto decode_send;
        }
    }
    else if (receive_rc != NI_TEST_RETCODE_END_OF_STREAM)
    {
```

```
    //Open ecoder after receiving first frame from decoder
    if (!encoder_opened)
    {
        p_hwframe = dec_ctx.hw_action == NI_CODEC_HW_ENABLE ?
        (niFrameSurface1_t *)out_frame.data.frame.p_data[3] : NULL;

        ret = encoder_open(enc_ctx, p_enc_api_param, output_total,
                           enc_conf_params, enc_gop_params,
                           &out_frame.data.frame, output_width,
                           output_height, fps_num, fps_den, bitrate,
                           enc_codec_format, enc_pix_fmt,
                         out_frame.data.frame.aspect_ratio_idc,
                           xcoderGUID, p_hwframe, 0, false);
        if (ret != 0)
        {
           break;
        }
        encoder_opened = 1;
    }
}

//Send frame to each encoder
for (i = 0; i < output_total; i++)
{
    ctx.curr_enc_index = i;
    send_rc = encoder_send_data2(&ctx, &enc_ctx[i], &out_frame,
                    &enc_in_frame, output_width, output_height);
    if (send_rc < 0)   //Error
    {
        if (dec_ctx.hw_action)
```

```
            {
                p_hwframe = (niFrameSurface1_t *)out_frame.data.frame.p_data[3];
                ni_hw_frame_ref(p_hwframe);
            } else
            {
                ni_decoder_frame_buffer_free(&out_frame.data.frame);
            }
            break;
        } else if (send_rc == NI_TEST_RETCODE_EAGAIN)
        {
            // need to resend
            i--;
            continue;
        } else if (dec_ctx.hw_action && !ctx.enc_eos_sent[i])
        {
            // HW frame send
             // increment reference count of frame buffer for each encoder
            p_hwframe = (niFrameSurface1_t *)out_frame.data.frame.p_data[3];
            ni_hw_frame_ref(p_hwframe);
        }
    }

    if (send_rc < 0)
    {
        break;
    } else if (dec_ctx.hw_action)
    {
        ni_frame_wipe_aux_data(&out_frame.data.frame);
    } else
    {
```

```
        ni_decoder_frame_buffer_free(&out_frame.data.frame);
    }


encode_recv:
    // Receive encodeed packets from all encoders
    // HW frame buffer is unreferenced and recycled in encoder_receive
    receive_rc = encoder_receive(&ctx, enc_ctx, &enc_in_frame, out_packet,
                                 output_width, output_height, output_total,
                                    output_fp);
    for (i = 0; receive_rc >= 0 && i < output_total; i++)
    {
        if (!ctx.enc_eos_received[i])
        {
            ni_log(NI_LOG_DEBUG, "enc %d continues to read!\n", i);
            end_of_all_streams = 0;
            break;
        } else
        {
            ni_log(NI_LOG_DEBUG, "enc %d eos !\n", i);
            end_of_all_streams = 1;
        }
    }
}
```

Check the ***ni_xcoder_transcode_filter*** program source code for transcoding flow example, and some practical usages such as transcoding with HW frames.

5.8.1 Multithreading

The ***ni_xcoder_multithread_transcode*** program provides a demo for the multi-threaded version of the transcoding pipeline. In this scenario, thread one is responsible for feeding the decoder. Thread two is retrieving decoded frames from the decoder. Thread three is sending the retrieved frames from thread two to the encoder. Finally, the last thread is retrieving the compressed bitstream from the encoder. The user application should do best to provide a round robin prioritization for each thread, otherwise if one thread were to get too many cycles, it ends up blocking progress on other entities. All th features supported in the single-threaded demo (***ni_xcoder_transcode_filter***) are supported in the multi-threaded version as well.

5.9 Uploading

5.9.1 Input YUV Frame Preparation / Device Selection

The frame upload preparation for the YUV buffer is the same as what was written in the Encoder section 5.6.2 “Input YUV Frame Preparation”.

Selecting which card to upload to can be done with the “-c" device selection parameter in the ***ni_xcoder_encode*** program. Any number >=0 will directly translate to the device number and a value of -1 will allow the resource allocation to select the least loaded encoder device.

5.9.2 Uploading Loop

The purpose of uploading a YUV frame to the card explicitly is to leverage the advantages of using HW frames. Without an explicit upload, only the decoder would be able to produce HW frames from a source on the host. Data movement on an upload loop is centered around sending a SW frame on the host up to the uploader instance on the device. Outputting a HW frame as a response, this frame is then free to be used and reused among the encoders and 2D engine filters available without extra YUV data movement. For the SW frame upload path, it is very similar to the encoder write path though with some key distinctions.

The uploading loop with output being used to encode is as follows:

```
while (!end_of_all_streams &&
    (send_rc == NI_TEST_RETCODE_SUCCESS ||
     receive_rc== NI_TEST_RETCODE_SUCCESS))
{
    if (first_frame_uploaded && !eos)
    {
        p_hwframe = hwupload_frame(&ctx, &upl_ctx, &sca_ctx, &swin_frame,
                                   &in_frame, &scale_frame, pix_fmt,
                                video_width[0], video_height[0],
                                   input_fp[0], yuv_buf, &eos);
        if (upl_ctx.status == NI_RETCODE_NVME_SC_WRITE_BUFFER_FULL)
        {
            ni_log(NI_LOG_DEBUG, "No space to write to, try to read a packet\n");
            p_in_frame = &in_frame;
            goto receive_pkt;
        } else if (p_hwframe == NULL)
        {
            ret = NI_TEST_RETCODE_FAILURE;
            break;
        }
    }

    // Sending
    p_in_frame = is_ni_enc_pix_fmt(pix_fmt) ? &in_frame : &scale_frame;
    for (i = 0; i < output_total; i++)
    {
        ctx.curr_enc_index = i;
```

```
        send_rc = encoder_send_data3(&ctx, &enc_ctx[i], p_in_frame,
                                     video_width[0], video_height[0], eos);
        first_frame_uploaded = 1;   //since first frame read before while-loop
        if (send_rc < 0)   //Error
        {
            ni_log(NI_LOG_ERROR, "enc %d send error, quit !\n", i);
            ni_hw_frame_ref(p_hwframe);
            end_of_all_streams = 1;
            break;
        }
        if (!ctx.enc_resend[i])
        {
            //Increment reference count of HW frame buffer
             //after sending to encoder
            //It will later be recycled in encoder_receive
            ni_hw_frame_ref(p_hwframe);
        } else
        {
            ni_log(NI_LOG_DEBUG, "enc %d need to re-send !\n", i);
            ni_usleep(500);
            i--;
            continue;
        }
    }
    if (end_of_all_streams)
        break;


receive_pkt:
    receive_rc = encoder_receive(&ctx, enc_ctx, p_in_frame, out_packet,
                                 video_width[0], video_height[0],
```

```
                                output_total, output_fp);
    for (i = 0; receive_rc >= 0 && i < output_total; i++)
    {
        if (!ctx.enc_eos_received[i])
        {
            ni_log(NI_LOG_DEBUG, "enc %d continues to read!\n", i);
            end_of_all_streams = 0;
            break;
        } else
        {
            ni_log(NI_LOG_DEBUG, "enc %d eos !\n", i);
            end_of_all_streams = 1;
        }
    }
}
```

The main libxcoder API functions to complete the YUV conversion from SW frame to HW frame consist of **ni_frame_buffer_alloc_pixfmt** and **ni_frame_buffer_alloc_hwenc** to prepare the buffers, and **ni_device_session_hwup** to make the trade.

It's possible to have a single HW frame being referenced by multiple encoders, essentially creating a ladder encoding with different encoding configurations. An example ***ni_xcoder_encode*** command demonstrating this is shown in section 0. Reference the ***ni_xcoder_encode*** source code for sample implementation details.

5.9.3 Recycling and Implicit Extra Allocation

As always, the HW frame nature of the output requires careful tracking of where the frame goes and when to recycle it with **ni_hwframe_buffer_recycle2**. Please follow the code example above (from ***ni_xcoder_encode***) for reference.

Usually the framepool used during **ni_device_session_init_framepool** does not need to be very big. The reason for this is that any downstream entity will automatically expand the allocation count as needed for its purpose.

The clearest example of this would be if a HW upload instance is paired with an encoding instance having the lookahead depth at 40 frames. Naturally, the encoder instance needs at least a buffer count of 40 and if the upload pool was initialized as 3 frames, it would not seem to be enough. However, the encoder will know that it is receiving HW frame inputs and will allocate additional space during encoder open which are compatible with the original upload instance thus expanding the poolsize to be 3 + 40. No code example of this is needed as this is a FW implementation.

5.9.4 Zero Copy

Application should follow these steps to incorporate zero copy while uploading a YUV frame –

- Before uploading each frame, call **ni_uploader_frame_zerocopy_check** with input frame width, height, linesize (aka stride), and pixel format.
- If zero copy is applicable, call **ni_encoder_frame_zerocopy_buffer_alloc** with input frame width, height, linesize (aka stride), pointers to each data buffer (p_src in the example), and metadata length (extra_data_len in the example) Example:

Zero Copy while uploading a frame

```
// NOTE – The following example is based on FFmpeg hwupload filter, which
receives an AvFrame (“src” in the following example) with an array of pointers
to each of the luma / chroma data buffers, as well as an array of linesizes for
each of the buffers. For users who implements applications based on libxcoder
APIs (and therefore cannot utilitze FFmpeg or Gstreamer frame structure),
please setup linesize array and data buffer pointer array and pass pointers of
the arrays to libxcoder APIs

// NOTE – Please note ni_encoder_frame_zerocopy_buffer_alloc extra_len
parameter must be NI_APP_ENC_FRAME_META_DATA_SIZE



// check input resolution zero copy compatible or not

if (ni_uploader_frame_zerocopy_check(&ctx->api_ctx, src->width, src->height,
(const int *)src->linesize, pixel_format) == NI_RETCODE_SUCCESS)

{

   need_to_copy = 0;

   p_src_session_data->data.frame.extra_data_ln =
NI_APP_ENC_FRAME_META_DATA_SIZE;

   // alloc metadata buffer etc. (if needed)

   ret = ni_encoder_frame_zerocopy_buffer_alloc( &p_src_session_data-
>data.frame, src->width,  src->height, (const int *)src->linesize, (const
uint8_t **)src->data,  (int)p_src_session_data->data.frame.extra_data_len);

   if (ret != NI_RETCODE_SUCCESS)

      return AVERROR(ENOMEM);

}
```

Please also refer to the following file / function for the complete sample code

- FFmpeg libavutil/hwcontext_ni_quad.c ni_hwup_frame ()

5.10 Downloading

5.10.1 Input Preparation, Purpose

The HW download is somewhat unique in its implementation in comparison to the other jobs like decoding, uploading, and encoding. The inputs to the hwdownload must be HW frames from a previous session such as an output from the list below:

1. Decoder in hwframe mode (excluding tiled4x4 format)
2. Scaler
3. Uploader

The purpose of the download is to convert HW frame formats into SW frame formats which are still compatible with with Quadra encoding. A typical scenario would be one such as downloading a HW frame from Quadra device A and sending it to Quadra device B since B does not have access to device A's memory for processing video data.

5.10.2 Downloading Sequence

The downloading sequence is very short and is comprised of these main steps

1. Prepare output buffer to contain raw frame info
2. Download the data using the HW frame as input
3. Freeing buffers where necessary
    a. Example below shows immediate HW frame buffer recycle but that is sequence specific. If there are parallel processes using the HWFrame, the aformentioned frame tracking mechanism is recommeneded and unreferencing the frame is the correct behaviour
    b. The hwdownload output buffer may be reused each time as long as the downstream user is no longer reading from it.

Example:

```
ni_session_data_io_t hwdl_session_data = {0};

ni_session_data_io_t *p_session_data = &hwdl_session_data;

niFrameSurface1_t *src_surf = (niFrameSurface1_t *)(p_src_frame->p_data[3]);

int ret = 0;

int pixel_format = NI_PIX_FMT_YUV420P; //could be others

ret = ni_frame_buffer_alloc_dl(&(p_session_data->data.frame),
src_surf->ui16width, src_surf->ui16height, pixel_format);

if (ret != NI_RETCODE_SUCCESS)

{
       return NI_RETCODE_ERROR_MEM_ALOC;
{

p_ctx->is_auto_dl = false; //for encoder internal download if true

ret = ni_device_session_hwdl(p_ctx, p_session_data, src_surf);

//Unref the hwframe if hwframe tracking used.

//Below is if no other entity is using the hwframe
ni_hwframe_buffer_recycle2(src_surf);


if (ret <= 0)
{
       ni_frame_buffer_free(&p_session_data->data.frame);
        return ret
}

//Do whatever is required with the output frame in p_session_data

//Note, the frame buffer for hwdownload may be reused

ni_frame_buffer_free(&(hwdl_session_data.data.frame));
```

5.11 AI Inference

5.11.1 Open an AI session

Like the encoder and decoder described previously, to run an AI inference, a session needs to be opened. The session id is a reference to the firmware session during the whole inference lifetime.

At the start of the session an ni_session_context_t structure shall be defined and initialized to refer to the AI session. We can specify the card id to choose different cards to start the AI session with. Passing “NI_DEVICE_TYPE_AI” to ni_device_session_open() creates an AI session.

```
ni_session_context_t api_ctx = {0};
ni_retcode_t retval;
retval = ni_device_session_context_init(&api_ctx);
api_ctx.hw_id = devid;
retval = ni_device_session_open(&api_ctx, NI_DEVICE_TYPE_AI);
```

Once an AI session is created successfully, we should pass the model’s network binary file in bytes to the firmware, for it to configure an AI model by calling the function ni_ai_config_network_binary().

On the firmware side, after receiving the whole network binary, it begins to put the content into memory and initializes it by sending init commands to the AI module. When the initialization completes the network layer parameters can be queried out from the firmware and filled in the ni_network_data_t structure.

```
ni_network_data_t network = {0};

retval = ni_ai_config_network_binary(&api_ctx, &network, nb_file);
```

ni_network_data_t defines the number and details of one model's input and output layers. Currently this structure supports models with up to NI_MAX_NETWORK_INPUT_NUM input layers and NI_MAX_NETWORK_OUTPUT_NUM output layers. libxcoder uses the ni_frame_t structure to carry the input data, and the ni_packet_t structure to hold the output data. Since there could be multiple layers for a model, the offset pointers of each layer data in the structure's buffer, will be calculated for future use.

```
typedef struct _ni_network_layer_info
{
    ni_network_layer_params_t in_param[NI_MAX_NETWORK_INPUT_NUM];
    ni_network_layer_params_t out_param[NI_MAX_NETWORK_OUTPUT_NUM];
} ni_network_layer_info_t;


typedef struct _ni_network_data
{
    uint32_t input_num;
    uint32_t output_num;
    ni_network_layer_info_t linfo;
    struct
    {
        int32_t
          offset; /* point to each input layer start offset from p_frame */
    } inset[NI_MAX_NETWORK_INPUT_NUM];
    struct
    {
        int32_t
          offset; /* point to each output layer start offset from p_packet */
    } outset[NI_MAX_NETWORK_OUTPUT_NUM];
} ni_network_data_t;
```

With the ni_network_data_t structure we know the size for the whole input and output data. The helper functions are used to allocate page aligned memory for ni_frame_t and ni_packet_t structures, with sizes based on the structure.

```
ni_session_data_io_t api_src_frame;

ni_session_data_io_t api_dst_packet;

retval = ni_ai_frame_buffer_alloc(&api_src_frame.data.frame, &network);

retval = ni_ai_packet_buffer_alloc(&api_dst_packet.data.packet, &network);
```

The ni_network_layer_params_t contains the network layer parameters. Its main purpose is to show how to interpret the AI hardware input and output data. Each field is described in the comments.

```
ni_session_data_io_t api_src_frame;

Typedef struct _ni_network_layer_params_t

{

   uint32_t num_of_dims; /* The number of dimensions specified in *sizes */

   uint32_t sizes[6];    /* The pointer to an array of dimension */

   int32_t

      data_format; /* Data format for the tensor, see ni_ai_buffer_format_e */

   int32_t

      quant_format; /* Quantized format see ni_ai_buffer_quantize_format_e */

   union

   {

      struct

      {

         int32_t

            fixed_point_pos; /* Specifies the fixed point position when the
input element type is int16, if 0 calculations are performed in integer math */

      } dfp;

      struct

      {

         float scale;      /* Scale value for the quantized value */

         int32_t zeroPoint; /* A 32 bit integer, in range [0, 255] */

      } affine;

   } quant_data; /* The union of quantization information */

   /* The type of this buffer memory. */

   uint32_t memory_type;

} ni_network_layer_params_t;
```

The models will be kept in the firmware if there are enough resources. This way, if there is another session that would like to do inferences with the same model, it can skip passing the network binary to the firmware and just read back the existing network parameters directly. This helps to speed up the configuration process.

5.11.2 Input data inference

In general, there are two types of input data, the first is picture data in the RBG pixel format, and the other is tensor files, which are in text format.

For pictures, if they are the correct shape the model requires, and if the model format is NHWC, we can directly pass the pixel data “NI_DEVICE_TYPE_AI” to perform inference. We restore the layer data to the ni_frame_t buffer using the pre-calculated offsets of each layer.

```
int32_t offset = network.inset[input_layer_idx].offset;
uint8_t *p_data = (uint8_t *)api_src_frame.data.frame.p_data[0] + offset;
linesize = pic_width;
for (w = 0; w < pic_height; w++) {
    memcpy(p_data, pic_data + w * linesize, pic_width);
    p_data += pic_width;
}
```

For tensor files, since each line in the tensor file is float32 data, and the fact the AI hardware accepts only the vendor specific data format, we first need to convert the float32 data to integer data by calling the helper functions ni_network_layer_convert_tensor() or ni_network_convert_tensor_to_data(). The resulting data is stored in the ni_frame_t buffer pointer p_data.

```
ni_retcode_t retval;

int32_t offset = network.inset[input_layer_idx].offset;

uint8_t *p_data = (uint8_t *)api_src_frame.data.frame.p_data[0] + offset;

int input_size =
ni_ai_network_layer_size(&network.linfo.in_param[input_layer_index]);

retval = ni_network_layer_convert_tensor(p_data, input_size, tensor_file,
&network.linfo.in_param[input_layer_index]);
```

5.11.3 Invoke an inference

Once all the layers' input data is prepared in the ni_frame_t structure, we can pass the input data to the firmware to start an inference, we do this by writing the session. The write may not succeed if there is not enough free buffer available. To retry, the system can poll until there is sufficient buffer available.

```
do {
    ret = ni_device_session_write(&api_ctx, &api_src_frame, NI_DEVICE_TYPE_AI);
    if (ret < 0) {
        /* error occurs */
        break;
    } else if (ret >= 0) {
        /* write complete */
        break;
    }
    /* can't write anything. Choose to keep polling the buffer to be written
here. */
} while (ret == 0);
```

After writing the session is complete, we can then poll the output data by reading the session. The read may not succeed immediately if the inference is ongoing. When the inference completes the output data will be stored in the ni_packet_t buffer.

```
do {
    ret = ni_device_session_read(&api_ctx, &api_dst_packet, NI_DEVICE_TYPE_AI);
    if (ret < 0) {
        /* error occurs */
        break;
    } else if (ret >= 0) {
        /* write complete */
        break;
    }
    /* can't read anything. Choose to keep polling the buffer to read here. */
} while (ret == 0);
```

5.11.4 Output data conversion

Any output data from the AI module will be in a hardware vendor specific Integer format. To get the tensor data for postprocessing, we need to convert the output data to the tensor data, based on the output layers network parameters. There are two helper function wrappers to do this conversion, ni_network_layer_convert_output() and ni_network_convert_data_to_tensor(), they accept different input arguments.

Here is an example using ni_network_layer_convert_output(). The first and the second arguments are the output buffer pointer and its size. This buffer is used to store the converted tensor data in the float32 type for the specific layer. The size is equal to the size of float32 multiplied by the ni_ai_network_layer_dims().

```
retval = ni_network_layer_convert_output(output_buffer,output_buffer_size,
    &api_dst_packet.data.packet,
    &network, output_layer_index);
```

5.11.5 Close an AI session

If the inference session is no longer needed, call the ni_device_session_close() to close the session along with which the input and output buffer resources will be reclaimed as well. As mentioned before, releasing the session doesn't mean releasing the model in the firmware. A reference to the model is kept in case the next session would like to infer the same model.

```
retval = ni_device_session_close(&api_ctx, 1, NI_DEVICE_TYPE_AI);
```

5.11.6 AI pre-processing for YUV data

That's a special case about AI processing, the input and output are both HW frame and the output frame can send to encoder for encoding.

Step1. open the AI session and set the network nb file to firmware.

```
ni_session_context_t api_ctx = {0};
ni_retcode_t retval;
retval = ni_device_session_context_init(&api_ctx);
api_ctx.hw_id = devid;
retval = ni_device_session_open(&api_ctx, NI_DEVICE_TYPE_AI);
ni_network_data_t network = {0};
retval = ni_ai_config_network_binary(&api_ctx, &network, nb_file);
```

Step2. Allocate buffer pool for output buffer. The host call ni_device_alloc_frame() to firmware in order to allocate HW frame, the max buffer number is 8. For host side, use ni_frame_buffer_alloc_hwenc() to allocate ni_session_data_t, the width/height/format should be same as input frame.

```
// Create frame pool
  int format = GC620_I420;
  int options = NI_AI_FLAG_IO |  NI_AI_FLAG_PC;


  /* Allocate a pool of frames by the AI */
  retval = ni_device_alloc_frame(&api_ctx,
                          in_frame->width,
                          in_frame->height,
                          format,
                          options,
                          0, // rec width
                          0, // rec height
                          0, // rec X pos
                          0, // rec Y pos
                          8, // frame buffer num
                          0, // frame index
                          NI_DEVICE_TYPE_AI);
   //Allocate dst_frame on host.
   ni_session_data_t dst_frame;
   ni_frame_buffer_alloc_hwenc(&dst_frame.data.frame,
                                      in_frame->width, in_frame->height, 0);
```

Another case is the Model support multi-frame in. The host call ni_device_multi_config_frame() to firmware to set input frames on one api:

```
ni_frame_config_t frame_in[3];
frame_in[0].picture_width  = input_video_width;
frame_in[0].picture_height = input_video_height;
frame_in[0].frame_index    = p_frame1->ui16FrameIdx;
frame_in[1].picture_width  = input_video_width;
frame_in[1].picture_height = input_video_height;
frame_in[1].frame_index    = p_frame2->ui16FrameIdx;
frame_in[2].picture_width  = input_video_width;
frame_in[2].picture_height = input_video_height;
frame_in[2].frame_index    = p_frame3->ui16FrameIdx;


// Config input frames index on this function.
ret = ni_device_multi_config_frame(&watermark_ctx,frame_in,3,NULL);
```

Step3. Prepare output frame buffer and send input HW frame to firmware. Call ni_device_alloc_frame() with NI_AI_FLAG_IO option to prepare the output buffer. Call ni_device_alloc_frame() without option to send input HW frame to firmware.

```
/* Try to get output buffer */
do{
    retval = ni_device_alloc_frame(&api_ctx, in_frame->width,in_frame ->height,
           GC620_I420, NI_AI_FLAG_IO, 0, 0,
           0, 0, 0, -1, NI_DEVICE_TYPE_AI);
   if (retval < NI_RETCODE_SUCCESS) { /* error occurs */
   }
 }while(retval != NI_RETCODE_SUCCESS);

 /* set input buffer */
retval = ni_device_alloc_frame(&api_ctx, 0, 0, 0, 0, 0, 0, 0, 0,
                               frame_surface->ui32nodeAddress,
                               frame_surface->ui16FrameIdx,
                               NI_DEVICE_TYPE_AI);
if (retval != NI_RETCODE_SUCCESS) {  /* error occurs */
}
```

Step4. Read out the HW output frame. Call ni_device_session_read_hwdesc() with NI_DEVICE_TYPE_AI.

```
do{
    retval = ni_device_session_read_hwdesc(
           &api_ctx, &dst_frame, NI_DEVICE_TYPE_AI);
    if(retval < NI_RETCODE_SUCCESS){    /* error occurs */
    }
}while(retval != NI_RETCODE_SUCCESS);
```

5.11.7 Further Information

For further reading, inside the Quadra Release folder, there is a selection of AI documentation, that covers in more detail Quadra's AI systems, and also the AI toolkit for model generation.

5.12 Libxcoder API Version Compatibility

Netint is committed to providing new libxcoder (SW) releases with compatibility for linked applications with code written for a previous version of ibxcoder release. This means, after installing a new version of libxcoder, a linked application does not need code change to work with the new libxcoder. But the linked application will likely need to be recompiled with the new libxcoder's headers.

5.12.1 libxcoder API version

The libxcoder API version number (e.g. 2.0) has two parts, semantic major and semantic minor. It is found in `libxcoder/source/ni_defs.h` as the macro `LIBXCODER_API_VERSION`.

When the semantic major version number changes, a backward incompatible change has been made to libxcoder public API. Linked applications likely need code change to operate with new libxcoder. Netint is unlikely to ever change libxcoder in a backward incompatible manner.

When the semantic minor version number changes, a backward compatible change has been made to libxcoder public API. Linked applications will need to be recompiled to operate with new libxcoder. Linked application code should be changed to cease using any deprecated functions, but this code change is not strictly necessary.

If libxcoder API version does not change in a new Netint SW release, it will not be necessary to recompile application linked to libxcoder.

5.12.2 Reading Various Version Numbers

To read the libxcoder API version number application is compiled with, use the macro `LIBXCODER_API_VERSION` in `ni_defs.h`.

To read the libxcoder API version number application is linked with, use the function `ni_get_libxcoder_api_ver()` in `ni_util.h`.

To read the compatible FW API version number for the libxcoder linked to by application, use the function `ni_get_compat_fw_api_ver()` in `ni_util.h`.

To read the Netint SW release version number for the libxcoder linked to by application, use the function `ni_get_libxcoder_release_ver()` in `ni_util.h`.

5.12.3 Libxcoder Public API Deprecation Warnings

New libxcoder releases may deprecate functions/structs/macros. When compiling an application linked to libxcoder, the use of deprecated elements will cause compiler warnings C4995/C4996. It is recommended, but not necessary, to update application code to not use deprecated elements. Check the code comments beside deprecated elements for suggestions on new elements to use; or, check for an element of similar name but with a number suffix in the name.

To disable warning for deprecation from libxcoder elements, add `-DNI_WARN_AS_ERROR` to your compiler options (e.g. `gcc -DNI_WARN_AS_ERROR`).

5.13 Libxcoder API Error Classification

5.13.1 libxcoder API Error Definition

The libxcoder API error definition is defined in `ni_defs.h` like follow table:

ERROR TYPE	VALUE	ERROR DETIAL
NI_RETCODE_FAILURE	-1	* unrecoverable failure *
NI_RETCODE_INVALID_PARAM	-2	* invalid parameter encountered *
		* uninitialized parameter encounted *
		* null pointer parameter encounted *
NI_RETCODE_ERROR_MEM_ALOC	-3	* host memory allocation failure *
NI_RETCODE_ERROR_NVME_CMD_FAILED	-4	* nvme command failure *
NI_RETCODE_ERROR_INVALID_SESSION	-5	* invalid session ID*
NI_RETCODE_ERROR_RESOURCE_UNAVAILABLE	-6	* HW resource currently unavailable *
NI_RETCODE_PARAM_INVALID_NAME	-7	* invalid parameter name *
NI_RETCODE_PARAM_INVALID_VALUE	-8	* invalid parameter value *
NI_RETCODE_PARAM_ERROR_FRATE	-9	* invalid frame rate parameter value *

NI_RETCODE_PARAM_ERROR_BRATE	-10	* invalid bit rate parameter value *
NI_RETCODE_PARAM_ERROR_TRATE	-11	not used
NI_RETCODE_PARAM_ERROR_VBV_BUFFER_SIZE	-12	* Invalid VBV BufferSize *
NI_RETCODE_PARAM_ERROR_INTRA_PERIOD	-13	* invalid intra period parameter value *
NI_RETCODE_PARAM_ERROR_INTRA_QP	-14	* invalid qp parameter value *
NI_RETCODE_PARAM_ERROR_GOP_PRESET	-15	* invalid gop preset parameter value *
NI_RETCODE_PARAM_ERROR_CU_SIZE_MODE	-16	* Invalid cu_size_mode: out of range *
NI_RETCODE_PARAM_ERROR_MX_NUM_MERGE	-17	not used
NI_RETCODE_PARAM_ERROR_DY_MERGE_8X8_EN	-18	not used
NI_RETCODE_PARAM_ERROR_DY_MERGE_16X16_EN	-19	not used
NI_RETCODE_PARAM_ERROR_DY_MERGE_32X32_EN	-20	not used
NI_RETCODE_PARAM_ERROR_CU_LVL_RC_EN	-21	* invalid cu level rate control parameter value *

NI_RETCODE_PARAM_ERROR_HVS_QP_EN	-22	* Invalid enable_hvs_qp *
NI_RETCODE_PARAM_ERROR_HVS_QP_SCL	-23	not used
NI_RETCODE_PARAM_ERROR_MN_QP	-24	* invalid minimum qp parameter value *
NI_RETCODE_PARAM_ERROR_MX_QP	-25	* invalid maximum qp parameter value *
NI_RETCODE_PARAM_ERROR_MX_DELTA_QP	-26	* invalid maximum delta qp parameter value *
NI_RETCODE_PARAM_ERROR_CONF_WIN_TOP	-27	* invalid top offset of conformance window parameter value *
NI_RETCODE_PARAM_ERROR_CONF_WIN_BOT	-28	* invalid bottom offset of conformance window parameter value *
NI_RETCODE_PARAM_ERROR_CONF_WIN_L	-29	* invalid left offset of conformance window parameter value *
NI_RETCODE_PARAM_ERROR_CONF_WIN_R	-30	* invalid right offset of conformance window parameter value *
NI_RETCODE_PARAM_ERROR_USR_RMD_ENC_PARAM	-31	* invalid user recommended parameter value *
NI_RETCODE_PARAM_ERROR_BRATE_LT_TRATE	-32	not used
NI_RETCODE_PARAM_ERROR_RCENABLE	-33	* Invalid enable_mb_level_rc: out of range *

NI_RETCODE_PARAM_ERROR_MAXNUMMERGE	-34	* Invalid max_num_merge: out of range *
NI_RETCODE_PARAM_ERROR_CUSTOM_GOP	-35	* invalid custom gop preset parameter value *
NI_RETCODE_PARAM_ERROR_PIC_WIDTH	-36	* invalid picture width parameter value *
NI_RETCODE_PARAM_ERROR_PIC_HEIGHT	-37	* invalid picture height parameter value *
NI_RETCODE_PARAM_ERROR_DECODING_REFRESH_TYPE	-38	* invalid decoding refresh type parameter value *
NI_RETCODE_PARAM_ERROR_CUSIZE_MODE_8X8_EN	-39	* Invalid use_recommend_enc_params and cu_size_mode*
NI_RETCODE_PARAM_ERROR_CUSIZE_MODE_16X16_EN	-40	
NI_RETCODE_PARAM_ERROR_CUSIZE_MODE_32X32_EN	-41	
NI_RETCODE_PARAM_ERROR_TOO_BIG	-42	* parameter value is too big *
NI_RETCODE_PARAM_ERROR_TOO_SMALL	-43	* parameter value is too small *
NI_RETCODE_PARAM_ERROR_ZERO	-44	* parameter value is zero *

NI_RETCODE_PARAM_ERROR_OOR	-45	* parameter value is out of range *
NI_RETCODE_PARAM_ERROR_WIDTH_TOO_BIG	-46	* parameter width value is too big *
NI_RETCODE_PARAM_ERROR_WIDTH_TOO_SMALL	-47	* parameter width value is too small *
NI_RETCODE_PARAM_ERROR_HEIGHT_TOO_BIG	-48	* parameter height value is too big *
NI_RETCODE_PARAM_ERROR_HEIGHT_TOO_SMALL	-49	* parameter height value is too small *
NI_RETCODE_PARAM_ERROR_AREA_TOO_BIG	-50	* parameter height x width value is too big *
NI_RETCODE_ERROR_EXCEED_MAX_NUM_SESSIONS	-51	not used
NI_RETCODE_ERROR_GET_DEVICE_POOL	-52	* cannot get info from device *
NI_RETCODE_ERROR_LOCK_DOWN_DEVICE	-53	* cannot obtain the file lock across all the process for query *
NI_RETCODE_ERROR_UNLOCK_DEVICE	-54	* cannot unlock the lock *
NI_RETCODE_ERROR_OPEN_DEVICE	-55	* cannot open the device *
NI_RETCODE_ERROR_INVALID_HANDLE	-56	* device handles that passed in is wrong *

NI_RETCODE_ERROR_INVALID_ALLOCATION_METHOD	-57	not used
NI_RETCODE_PARAM_WARNING_DEPRECATED	-59	* deprecated parameter *
NI_RETCODE_PARAM_ERROR_LOOK_AHEAD_DEPTH	-60	* invalid lookahead depth *
NI_RETCODE_PARAM_ERROR_FILLER	-61	not used
NI_RETCODE_PARAM_ERROR_PICSKIP	-62	not used
NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION	-63	* FW version is not supported *
NI_RETCODE_ERROR_UNSUPPORTED_FEATURE	-64	* feature is not supported *
NI_RETCODE_ERROR_PERMISSION_DENIED	-65	* admin priviledge needed *
NI_RETCODE_ERROR_STREAM_ERROR	-66	* decoder detect stream error *
NI_RETCODE_PARAM_WARN	0x100	* just a warning to print *
NI_RETCODE_EAGAIN	0x101	* memory buffer allocation/query failed*
NI_RETCODE_NVME_SC_RESOURCE_UNAVAILABLE	0x301	* FW subsystem init failed *

NI_RETCODE_NVME_SC_RESOURCE_IS_EMPTY	0x302	* FW subsystem resource is empty *
NI_RETCODE_NVME_SC_RESOURCE_NOT_FOUND	0x303	* FW subsystem Id is not found *
NI_RETCODE_NVME_SC_REQUEST_NOT_COMPLETED	0x304	* FW subsytem request is not completed *
NI_RETCODE_NVME_SC_REQUEST_IN_PROGRESS	0x305	not used
NI_RETCODE_NVME_SC_INVALID_PARAMETER	0x306	* FW subsystem get invalid param *
NI_RETCODE_NVME_SC_STREAM_ERROR	0x307	* FW subsystem detect stream error *
NI_RETCODE_NVME_SC_VPU_RSRC_INSUFFICIENT	0x3FE	* FW subsystem insufficient resource *
NI_RETCODE_NVME_SC_VPU_GENERAL_ERROR	0x3FF	* FW subsystem error occurred *

6 Appendix

6.1 API Description Details

Allocate session context & init

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

Clear already allocated session context

```
/*!*****************************************************************************
******
 *  \brief  Clear already allocated session context
 *
 *  \param[in]  p_ctx Pointer to an already allocated ni_session_context_t
*******************************************************************************
******/
LIB_API void ni_device_session_context_clear(ni_session_context_t *p_ctx);
```

Free session context

```
/*!*****************************************************************************
*******
 *  \brief  Free previously allocated session context
 *
 *  \param[in]  p_ctx Pointer to an already allocated ni_session_context_t struct

*******************************************************************************
*****/
 LIB_API void ni_device_session_context_free(ni_session_context_t *p_ctx);
```

Create an event handle (Windows only)

```
/*!*****************************************************************************
*******
  *  \brief  Create event and return 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_create_event();
```

Close and release event resource (Windows only)

```
/*!*****************************************************************************
 *  \brief  Close event and release resources
 *
 *  \return NONE
 ******************************************************************************/
LIB_API void ni_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_DEPRECATED
 ni_device_handle_t ni_device_open(const char* dev, uint32_t * p_max_io_size_out);
```

Note: This had been deprecated, it had been replaced by ni_device_open2.

```
/*!*****************************************************************************
********
 *   \brief    Open device and return device device_handle if successful
 *
 *   \param[in]    p_dev Device name represented as c string. ex: "/dev/nvme0"
 *   \param[in]    mode Device configuration parameters
 *
 *   \return On success returns a device device_handle
 *                On failure returns NI_INVALID_DEVICE_HANDLE
 ******************************************************************************
*******/
LIB_API ni_device_handle_t ni_device_open2(const char *dev, ni_device_mode_t
mode);
```

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_device_open2()
 *
 *  \return NONE
*******************************************************************************/
LIB_API void ni_device_close(ni_device_handle_t dev);
```

Query device capability

```
/*!*****************************************************************************
 *  \brief  Query device and return device capability structure
 *            This function had been replaced by ni_device_capability_query2
 *            This function can't be callback in multi thread
 *
 *  \param[in] device_handle  Device handle obtained by calling ni_device_open2
 *  \param[in] p_cap  Pointer to a caller allocated ni_device_capability_t
 *                        struct
 *  \return On success
 *                          NI_RETCODE_SUCCESS
 *            On failure
 *                          NI_RETCODE_INVALID_PARAM
 *                          NI_RETCODE_ERROR_MEM_ALOC
 *                          NI_RETCODE_ERROR_NVME_CMD_FAILED
 ******************************************************************************/
LIB_API              ni_retcode_t ni_device_capability_query(ni_device_handle_t
device_handle, ni_device_capability_t *p_cap);
```

```
/*!*****************************************************************************
********
 *   \brief    Query device and return device capability structure
 *             This function had replaced ni_device_capability_query
 *             This function can be callback with multi thread
 *
 *   \param[in] device_handle    Device handle obtained by calling ni_device_open2
 *   \param[in] p_cap    Pointer to a caller allocated ni_device_capability_t
 *                       struct
 *   \param[in] device_in_ctxt If device is in ctx
 *   \return On success
 *                       NI_RETCODE_SUCCESS
 *             On failure
 *                       NI_RETCODE_INVALID_PARAM
 *                       NI_RETCODE_ERROR_MEM_ALOC
 *                       NI_RETCODE_ERROR_NVME_CMD_FAILED
 ******************************************************************************
*******/
LIB_API ni_retcode_t ni_device_capability_query2(ni_device_handle_t device_handle,
                                                 ni_device_capability_t *p_cap, bool device_in_ctxt);
```

Open device session

```
/*!*****************************************************************
********
*   \brief    Open a new device session depending on the device_type parameter
*               If device_type is NI_DEVICE_TYPE_DECODER opens decoding session
*               If device_type is NI_DEVICE_TYPE_ENCODER opens encoding session
*               If device_type is NI_DEVICE_TYPE_SCALER opens scaling session
*
*   \param[in] p_ctx              Pointer to a caller allocated
*                                   ni_session_context_t struct
*   \param[in] device_type    NI_DEVICE_TYPE_DECODER, NI_DEVICE_TYPE_ENCODER,
*                                   or NI_DEVICE_TYPE_SCALER
*   \param[in] xcoder_locked Flag indicating whether xcoder mutex is already
*                                    locked by the caller, this prevents locking twice.
*                                    Generally only set to 1 for internal API calls
*                                    or if user application decides to lock the
*                                    xcoder_mutex outside of API
*   \return On success
*                                   NI_RETCODE_SUCCESS
*               On failure
*                                   NI_RETCODE_INVALID_PARAM
*                                   NI_RETCODE_ERROR_MEM_ALOC
*                                   NI_RETCODE_ERROR_NVME_CMD_FAILED
*                                   NI_RETCODE_ERROR_INVALID_SESSION
*****************************************************************
******/
ni_retcode_t ni_device_session_open(ni_session_context_t *p_ctx,
                                                ni_device_type_t device_type,
                                                uint8_t xcoder_locked)
```

Close device session

```
/*!*****************************************************************
********
*   \brief    Close device session that was previously opened by calling
*               ni_device_session_open()
*               If device_type is NI_DEVICE_TYPE_DECODER closes decoding session
*               If device_type is NI_DEVICE_TYPE_ENCODER closes encoding session
*               If device_type is NI_DEVICE_TYPE_SCALER closes scaling session
*
*   \param[in] p_ctx              Pointer to a caller allocated
*                                   ni_session_context_t struct
*   \param[in] eos_received Flag indicating if End Of Stream indicator was
*                                   received
*   \param[in] device_type    NI_DEVICE_TYPE_DECODER, NI_DEVICE_TYPE_ENCODER,
*                                   or NI_DEVICE_TYPE_SCALER
*   \param[in] xcoder_locked Flag indicating whether xcoder mutex is already
*                                    locked by the caller, this prevents locking twice.
*                                    Generally only set to 1 for internal API calls
*                                    or if user application decides to lock the
*                                    xcoder_mutex outside of API
*   \return On success
*                                   NI_RETCODE_SUCCESS
*               On failure
*                                   NI_RETCODE_INVALID_PARAM
*                                   NI_RETCODE_ERROR_NVME_CMD_FAILED
*                                   NI_RETCODE_ERROR_INVALID_SESSION
*****************************************************************
******/
LIB_API ni_retcode_t ni_device_session_close(ni_session_context_t *p_ctx,
                                             int eos_received,
                                             ni_device_type_t device_type,
                                             uint8_t xcoder_locked);
```

Flush a device session

```
/*!*****************************************************************************
 *  \brief  Sends a flush command to the device
 *          If device_type is NI_DEVICE_TYPE_DECODER sends flush command to decoder
 *          If device_type is NI_DEVICE_TYPE_ENCODER sends flush command to decoder
 *
 *  \param[in] p_ctx          Pointer to a caller allocated ni_session_context_t struct
 *  \param[in] device_type    NI_DEVICE_TYPE_DECODER or NI_DEVICE_TYPE_ENCODER
 *  \return On success
 *                                  NI_RETCODE_SUCCESS
 *                  On failure
 *                                  NI_RETCODE_INVALID_PARAM
 *                                  NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                                  NI_RETCODE_ERROR_INVALID_SESSION
*******************************************************************************/
LIB_API ni_retcode_t ni_device_session_flush(ni_session_context_t *p_ctx,
ni_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_session_context_t struct
*   \param[in] hdr_data          Pointer to header data
*   \param[in] hdr_size          Size of header data in bytes
*   \param[in] xcoder_locked Flag indicating whether xcoder mutex is already
*                                locked by the caller, this prevents locking twice.
*                                Generally only set to 1 for internal API calls
*                                or if user application decides to lock the
*                                xcoder_mutex outside of API
*   \return On success
*                               NI_RETCODE_SUCCESS
*             On failure
*                               NI_RETCODE_INVALID_PARAM
*                               NI_RETCODE_ERROR_NVME_CMD_FAILED
*                               NI_RETCODE_ERROR_INVALID_SESSION
*****************************************************************************
******/
LIB_API ni_retcode_t
ni_device_dec_session_save_hdrs(ni_session_context_t *p_ctx, uint8_t *hdr_data,
                                uint8_t hdr_size, uint8_t xcoder_locked);
```

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_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_session_context_t struct
 *   \return On success
 *                                   NI_RETCODE_SUCCESS
 *               On failure
 *                                   NI_RETCODE_INVALID_PARAM
 *                                   NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                                   NI_RETCODE_ERROR_INVALID_SESSION
*******************************************************************
******/
LIB_API ni_retcode_t ni_device_dec_session_flush(ni_session_context_t *p_ctx);
```

Write to a device session

```
/*!*****************************************************************************
 *  \brief  Send data to the device
 *          If device_type is NI_DEVICE_TYPE_DECODER sends data packet to
 *          decoder
 *          If device_type is NI_DEVICE_TYPE_ENCODER sends data frame to encoder
 *          If device_type is NI_DEVICE_TYPE_AI sends data frame to AI engine
 *
 *  \param[in] p_ctx          Pointer to a caller allocated
 *                            ni_session_context_t struct
 *  \param[in] p_data         Pointer to a caller allocated
 *                            ni_session_data_io_t struct which contains either a
 *                            ni_frame_t data frame or ni_packet_t data packet to
 *                            send
 *  \param[in] device_type    NI_DEVICE_TYPE_DECODER or
NI_DEVICE_TYPE_ENCODER or
 *                            NI_DEVICE_TYPE_AI
 *                            If NI_DEVICE_TYPE_DECODER is specified, it is
 *                            expected that the ni_packet_t struct inside the
 *                            p_data pointer contains data to send.
 *                            If NI_DEVICE_TYPE_ENCODER or NI_DEVICE_TYPE_AI
is
 *                            specified, it is expected that the ni_frame_t
 *                            struct inside the p_data pointer contains data to
 *                            send.
 *  \return On success
 *                            Total number of bytes written
 *          On failure
 *                            NI_RETCODE_INVALID_PARAM
 *                            NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                            NI_RETCODE_ERROR_INVALID_SESSION
```

****
******/ LIB_API int ni_device_session_write(ni_session_context_t* p_ctx,
ni_session_data_io_t* p_data, ni_device_type_t device_type);

Read from a device session

```
/*!*****************************************************************
*******
 *   \brief    Read data from the device
 *             If device_type is NI_DEVICE_TYPE_DECODER reads data packet from
 *             decoder
 *             If device_type is NI_DEVICE_TYPE_ENCODER reads data frame from
 *             encoder
 *             If device_type is NI_DEVICE_TYPE_AI reads data frame from AI engine
 *
 *   \param[in] p_ctx            Pointer to a caller allocated
 *                                 ni_session_context_t struct
 *   \param[in] p_data           Pointer to a caller allocated ni_session_data_io_t
 *                                 struct which contains either a ni_frame_t data frame
 *                                 or ni_packet_t data packet to send
 *   \param[in] device_type    NI_DEVICE_TYPE_DECODER, NI_DEVICE_TYPE_ENCODER,
or
 *                                 NI_DEVICE_TYPE_SCALER
 *                                 If NI_DEVICE_TYPE_DECODER is specified, data that
 *                                 was read will be placed into ni_frame_t struct
 *                                 inside the p_data pointer
 *                                 If NI_DEVICE_TYPE_ENCODER is specified, data that
 *                                 was read will be placed into ni_packet_t struct
 *                                 inside the p_data pointer
 *                                 If NI_DEVICE_TYPE_AI is specified, data that was
 *                                 read will be placed into ni_frame_t struct inside
 *                                 the p_data pointer
 *   \return On success
 *                                 Total number of bytes read
 *             On failure
 *                                 NI_RETCODE_INVALID_PARAM
 *                                 NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                                 NI_RETCODE_ERROR_INVALID_SESSION
```

****
******/ LIB_API int ni_device_session_read(ni_session_context_t* p_ctx,
ni_session_data_io_t* p_data, ni_device_type_t device_type);

Query a device session

```
/*!*****************************************************************************
*******
 *  \brief   Query session data from the device -
 *             If device_type is valid, will query session data
 *             from specified device type
 *
 *  \param[in] p_ctx            Pointer to a caller allocated
 *                                                    ni_session_context_t struct
 *  \param[in] device_type   NI_DEVICE_TYPE_DECODER or
 *                               NI_DEVICE_TYPE_ENCODER or
 *                               NI_DEVICE_TYPE_SCALER or
 *                               NI_DEVICE_TYPE_AI or
 *                               NI_DEVICE_TYPE_UPLOADER
 *
 *  \return On success
 *                               NI_RETCODE_SUCCESS
 *             On failure
 *                               NI_RETCODE_INVALID_PARAM
 *                               NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                               NI_RETCODE_ERROR_INVALID_SESSION
*******************************************************************************
******/
LIB_API ni_retcode_t ni_device_session_query(ni_session_context_t* p_ctx,
ni_device_type_t device_type);
```

Send box paramters to drawbox session

```
/*!*****************************************************************************
*******
   *   \biref Send box paramters to drawbox filter
   *
   *   \param[in] p_ctx      Pointer to an session_context
   *   \param[in] p_params    Structure containing box input params
   *   \return On success
   *                                    NI_RETCODE_SUCCESS
   *              On failure
   *                                    NI_RETCODE_INVALID_PARAM
   *                                    NI_RETCODE_ERROR_INVALID_SESSION
   *                                    NI_RETCODE_ERROR_MEM_ALOC
   *                                    NI_RETCODE_ERROR_NVME_CMD_FAILED

*****************************************************************************
**** /


ni_retcode_t ni_scaler_set_drawbox_params(ni_session_context_t *p_ctx,
ni_scaler_drawbox_params_t *p_params);
```

Config Namespace number for different SRIOV

```
/*!*****************************************************************************
*******
   * \brief Send namespace number and SRIOv index to the device to config the
namespace No.
   *
   * \param[in] device_handle Device handle obtained by calling ni_device_open2
   * \param[in] namespace_num Set the namespace number with designated sriov
   * \param[in] sriov_index Identify which sriov need to be set
   *
   * \return On success
   *                                    NI_RETCODE_SUCCESS
* On failure
*                                   NI_RETCODE_ERROR_MEM_ALOC
*                                   NI_RETCODE_ERROR_NVME_CMD_FAILED
*******************************************************************************
******/

IB_API ni_retcode_t ni_device_config_namespace_num(ni_device_handle_t
device_handle,

uint32_t namespace_num, uint32_t sriov_index);
```

Allocate a frame buffer

```
/*!*****************************************************************************
*******
 *   \brief    Allocate memory for the frame buffer based on provided parameters
 *
 *   \param[in] p_frame            Pointer to a caller allocated
 *                                                          ni_frame_t struct
 *   \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] 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,
 *                                      4 for 32 bits/pixel (RGBA)
 *   \param[in] hw_frame_count Number of hw descriptors stored in lieu of raw YUV
 *   \param[in] is_planar         0 if semiplanar else planar
 *
 *   \return On success
 *                                    NI_RETCODE_SUCCESS
 *               On failure
 *                                    NI_RETCODE_INVALID_PARAM
 *                                    NI_RETCODE_ERROR_MEM_ALOC
*******************************************************************************
******/
LIB_API ni_retcode_t ni_frame_buffer_alloc(ni_frame_t *pframe, int video_width, int
video_height, int alignment, int metadata_flag, int factor, int hw_frame_count, int
is_planar);
```

Allocate memory for a hw frame buffer

```
/*!*****************************************************************************
*******
 *   \brief    Allocate preliminary memory for the frame buffer based on provided
 *             parameters.
 *
 *   \param[in] p_frame          Pointer to a caller allocated
 *                                 ni_frame_t struct
 *   \param[in] video_width      Width of the video frame
 *   \param[in] video_height     Height of the video frame
 *   \param[in] pixel_format     Format for input
 *
 *   \return On success
 *                                 NI_RETCODE_SUCCESS
 *             On failure
 *                                 NI_RETCODE_INVALID_PARAM
 *                                 NI_RETCODE_ERROR_MEM_ALOC
*******************************************************************************
******/
LIB_API ni_retcode_t ni_frame_buffer_alloc_dl(ni_frame_t *p_frame,
                                              int video_width, int video_height,
                                              int pixel_format);
```

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_decoder_frame_buffer_free.
 *   Note:     all attributes of ni_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_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_RETCODE_SUCCESS
 *             On failure
 *                                 NI_RETCODE_INVALID_PARAM
 *                                 NI_RETCODE_ERROR_MEM_ALOC
*******************************************************************
******/
LIB_API ni_retcode_t ni_decoder_frame_buffer_alloc(ni_buf_pool_t* p_pool,
ni_frame_t *pframe, int alloc_mem, int video_width, int video_height, int alignment, int
factor);
```

Allocate a YUV frame buffer for encoder input

```
/*!*****************************************************************
*******
   *   \brief    Allocate memory for the frame buffer based on provided parameters
   *             taking into account pic line size and extra data.
   *             Applicable to YUV420p AVFrame only. Cb/Cr size matches that of Y.
   *
   *   \param[in] p_frame          Pointer to a caller allocated ni_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        Allignment requirement
   *   \param[in] extra_len        Extra data size (incl. meta data)
   *   \param[in] alignment_2pass_wa set alignment to work with 2pass encode
   *
   *   \return On success
   *                                 NI_RETCODE_SUCCESS
   *             On failure
   *                                 NI_RETCODE_INVALID_PARAM
   *                                 NI_RETCODE_ERROR_MEM_ALOC

*****************************************************************
****/
LIB_API ni_retcode_t ni_encoder_frame_buffer_alloc(ni_frame_t *pframe, int
video_width, int video_height, int linesize[], int alignment, int extra_len, bool
alignment_2pass_wa);
```

Allocate device destination frame from scaler hwframe pool

```
/*!*****************************************************************************
********
*   \brief    allocate device destination frame from scaler hwframe pool
*
*   \param [in] p_ctx          pointer to an session_context
*   \param [in] scaler_params    structure containing scaler input params
*   \param [in] p_surface pointer to the hw descriptor of scaler destination frame
*
*   \return 0 if successful, < 0 otherwise
*******************************************************************************
******
LIB_API ni_retcode_t ni_scaler_dest_frame_alloc(
      ni_session_context_t *p_ctx, ni_scaler_input_params_t scaler_params,
      niFrameSurface1_t *p_surface);
```

Allocate device input frame by hw descriptor

```
/*!*****************************************************************************
********
*   \brief    allocate device input frame by hw descriptor. This call won't allocate
                  a frame but sends the incoming hardware frame index to the scaler
*
*   \param [in] p_ctx          pointer to a session_context
*   \param [in] scaler_params    structure containing scaler input params
*   \param [in] p_surface pointer to the hw descriptor of scaler input frame
*
*   \return 0 if successful, < 0 otherwise
*******************************************************************************
******
LIB_API ni_retcode_t ni_scaler_input_frame_alloc(
      ni_session_context_t *p_ctx, ni_scaler_input_params_t scaler_params,
      niFrameSurface1_t *p_src_surface);
```

Init output pool of scaler frames

```
/*!*****************************************************************************
********
*   \brief    init output pool of scaler frames
*
*   \param [in] p_ctx          pointer to an session_context
*   \param [in] scaler_params    structure containing scaler input params
*
*   \return 0 if successful, < 0 otherwise
*******************************************************************************
******
LIB_API ni_retcode_t ni_scaler_frame_pool_alloc(
      ni_session_context_t *p_ctx, ni_scaler_input_params_t scaler_params);
```

Get p2p buffer address of a frame of scaler

```
/*!*****************************************************************************
********
*    \brief    Get p2p buffer address of a frame of scaler.
*
*    \param [in] p_ctx          pointer to an session_context
*    \param [in] p_surface     pointer to niFrameSurface1_t
*    \param [in] data_len        size of data to be read(aligned to 4k)
*
*    \return 0 if successful, < 0 otherwise
*****************************************************************************
******
LIB_API ni_retcode_t ni_scaler_p2p_frame_acquire(ni_session_context_t *p_ctx,
                                                 niFrameSurface1_t
*p_surface,
                                                 int data_len);
```

Allocate a semiplanar YUV frame buffer for encoder input

```
/*!*****************************************************************
*******
   *   \brief    Allocate memory for the frame buffer based on provided parameters
   *             taking into account pic line size and extra data.
   *             Applicable to YUV420p AVFrame only. Cb/Cr size matches that of Y.
   *
   *   \param[in] p_frame          Pointer to a caller allocated ni_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        Allignment requirement
   *   \param[in] extra_len        Extra data size (incl. meta data)
   *   \param[in] alignment_2pass_wa set alignment to work with 2pass encode
   *
   *   \return On success
   *                                 NI_RETCODE_SUCCESS
   *             On failure
   *                                 NI_RETCODE_INVALID_PARAM
   *                                 NI_RETCODE_ERROR_MEM_ALOC

*****************************************************************
****/
LIB_API ni_retcode_t ni_frame_buffer_alloc_nv(ni_frame_t *pframe, int video_width,
int video_height, int linesize[], int alignment, int extra_len, bool alignment_2pass_wa);
```

Wrapper for ni_encoder_frame_buffer_alloc

```
/*!*****************************************************************************
*******
   *   \brief    This API is a wrapper for ni_encoder_frame_buffer_alloc(), used
   *             for planar pixel formats, and ni_frame_buffer_alloc_nv(), used for
   *             semi-planar pixel formats. This API is meant to combine the
   *             functionality for both formats.
   *             Allocate memory for the frame buffer for encoding based on given
   *             parameters, taking into account pic line size and extra data.
   *             Applicable to YUV420p(8 or 10 bit/pixel) or nv12 AVFrame.
   *             Cb/Cr size matches that of Y.
   *
   *   \param[in] planar             true: if planar:
   *                                     pixel_format == (NI_PIX_FMT_YUV420P ||
   *                                           NI_PIX_FMT_YUV420P10LE
||NI_PIX_FMT_RGBA).
   *                                     false: semi-planar:
   *                                     pixel_format == (NI_PIX_FMT_NV12 ||
   *                                           NI_PIX_FMT_P010LE).
   *   \param[in] p_frame            Pointer to a caller allocated ni_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         Allignment requirement. Only used for planar format.
   *   \param[in] extra_len         Extra data size (incl. meta data). < 0 means not
   *                                     to allocate any buffer (zero-copy from existing)
   *   \param[in] alignment_2pass_wa set alignment to work with 2pass encode
   *
   *   \return On success
   *                                   NI_RETCODE_SUCCESS
   *             On failure
   *                                   NI_RETCODE_INVALID_PARAM
   *                                   NI_RETCODE_ERROR_MEM_ALOC
```

```
**********************************************************************
*****/
LIB_API ni_retcode_t ni_encoder_sw_frame_buffer_alloc(bool planar, ni_frame_t
*p_frame,
                                                      int video_width, int
video_height,
                                                          int linesize[], int
alignment,
                                                          int extra_len,
                                                          bool
alignment_2pass_wa);
```

Free a frame buffer

```
/*!*****************************************************************************
*******
 *   \brief    Free frame buffer that was previously allocated with either
 *             ni_frame_buffer_alloc or ni_encoder_frame_buffer_alloc or
 *             ni_frame_buffer_alloc_nv or ni_frame_buffer_alloc_dl
 *
 *   \param[in] p_frame       Pointer to a previously allocated ni_frame_t struct
 *
 *   \return On success       NI_RETCODE_SUCCESS
 *             On failure       NI_RETCODE_INVALID_PARAM
*******************************************************************************
******/ LIB_API ni_retcode_t ni_frame_buffer_free(ni_frame_t *pframe);
```

Free a decoder frame buffer

```
/*!*****************************************************************************
*******
 *   \brief    Free decoder frame buffer that was previously allocated with
 *             ni_decoder_frame_buffer_alloc, returning memory to a buffer pool.
 *
 *   \param[in] p_frame       Pointer to a previously allocated ni_frame_t struct
 *
 *   \return On success       NI_RETCODE_SUCCESS
 *             On failure       NI_RETCODE_INVALID_PARAM
*******************************************************************************
******/
LIB_API ni_retcode_t ni_decoder_frame_buffer_free(ni_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_decoder_frame_buffer_pool_return_buf(ni_buf_t *buf, ni_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_packet_t struct
 *   \param[in] packet_size      Required allocation size
 *
 *   \return On success
 *                                   NI_RETCODE_SUCCESS
 *                       On failure
 *                                   NI_RETCODE_INVALID_PARAM
 *                                   NI_RETCODE_ERROR_MEM_ALOC
*******************************************************************************
******/
LIB_API ni_retcode_t ni_packet_buffer_alloc(ni_packet_t *ppacket, int packet_size);
```

Allocate packet buffer using a user provided buffer

```
/*!*****************************************************************
*******
 *   \brief    Allocate packet buffer using a user provided pointer, the memory
 *     is expected to have already been allocated.
 *
 *     For ideal performance memory should be 4k aligned. If it is not 4K aligned
 *     then a temporary 4k aligned memory will be used to copy data to and from
 *     when writing and reading. This will negatively impact performance.
 *
 *     This API will overwrite p_packet->buffer_size, p_packet->p_buffer and
 *     p_packet->p_data fields in p_packet.
 *
 *     This API will not free any memory associated with p_packet->p_buffer and
 *     p_packet->p_data fields in p_packet.
 *     Common use case could be,
 *           1. Allocate memory to pointer
 *           2. Call ni_custom_packet_buffer_alloc() with allocated pointer.
 *           3. Use p_packet as required.
 *           4. Call ni_packet_buffer_free() to free up the memory.
 *
 *   \param[in] p_buffer          User provided pointer to be used for buffer
 *   \param[in] p_packet          Pointer to a caller allocated
 *                                                          ni_packet_t struct
 *   \param[in] buffer_size      Buffer size
 *   \return On success
 *                                  NI_RETCODE_SUCCESS
 *               On failure
 *                                  NI_RETCODE_INVALID_PARAM
 *                                  NI_RETCODE_ERROR_MEM_ALOC
 ******************************************************************/
ni_retcode_t ni_custom_packet_buffer_alloc(void *p_buffer,
                                           ni_packet_t *p_packet,
                                           int buffer_size);
```

Free a packet buffer

```
/*!*****************************************************************************
 *   \brief    Free packet buffer that was previously allocated with ni_packet_buffer_alloc
 *
 *   \param[in] p_packet       Pointer to a previously allocated ni_packet_t struct
 *
 *   \return On success       NI_RETCODE_SUCCESS
 *                            On failure       NI_RETCODE_INVALID_PARAM
 ******************************************************************************/
LIB_API ni_retcode_t ni_packet_buffer_free(ni_packet_t *ppacket);
```

Free a Packet buffer used for AV1

```
/*!*****************************************************************************
 *******
 *   \brief    Free packet buffer that was previously allocated with either
 *               ni_packet_buffer_alloc for AV1 packets merge
 *
 *   \param[in] p_packet       Pointer to a previously allocated ni_packet_t struct
 *
 *   \return On success       NI_RETCODE_SUCCESS
 *               On failure       NI_RETCODE_INVALID_PARAM
 ******************************************************************************
 ******/LIB_API ni_retcode_t ni_packet_buffer_free_av1(ni_packet_t *ppacket);
```

Copy video packet data to a packet buffer

```
/*!*****************************************************************************
*******
 *  \brief    Copy video packet accounting for alignment
 *
 *  \param[in] p_destination    Destination to where to copy to
 *  \param[in] p_source         Source from where 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
 *
 *  \return On success          Total number of bytes that were copied
 *          On failure          NI_RETCODE_FAILURE
*******************************************************************************
******/ LIB_API int ni_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] param         Pointer to a user allocated ni_xcoder_params_t
 *                               to initialize to default parameters
 *   \param[in] fps_num        Frames per second
 *   \param[in] fps_denom      FPS denomination
 *   \param[in] bit_rate       bit rate
 *   \param[in] width          frame width
 *   \param[in] height         frame height
 *
 *   \return On success
 *                               NI_RETCODE_SUCCESS
 *               On failure
 *                               NI_RETCODE_FAILURE
 *                               NI_RETCODE_INVALID_PARAM
*******************************************************************************
******/ LIB_API ni_retcode_t ni_encoder_init_default_params(ni_xcoder_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_xcoder_params_t
 *                                                 to initialize to default parameters
 *   \param[in] fps_num        Frames per second
 *   \param[in] fps_denom      FPS denomination
 *   \param[in] bit_rate       bit rate
 *   \param[in] width          frame width
 *   \param[in] height         frame height
 *
 *   \return On success
 *                                 NI_RETCODE_SUCCESS
 *               On failure
 *                                 NI_RETCODE_FAILURE
 *                                 NI_RETCODE_INVALID_PARAM
*******************************************************************************
******/ LIB_API ni_retcode_t ni_decoder_init_default_params(ni_xcoder_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_xcoder_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_RETCODE_SUCCESS
 *                        On failure
 *                                  NI_RETCODE_FAILURE
 *                                  NI_RETCODE_INVALID_PARAM
*******************************************************************
******/
LIB_API ni_retcode_t ni_encoder_params_set_value(ni_xcoder_params_t * p_params,
const char *name, const char *value);
```

Parse and set value for decoder parameters

```
/*!*****************************************************************
********
*   \brief    Set value referenced by name in decoder parameters structure
*
*   \param[in] p_params      Pointer to a user allocated ni_xcoder_params_t (used
*                              for decoder too for now ) 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_RETCODE_SUCCESS
*              On failure
*                              NI_RETCODE_FAILURE
*                              NI_RETCODE_INVALID_PARAM
*****************************************************************
******/ LIB_API ni_retcode_t ni_ ni_decoder_params_set_value (ni_xcoder_params_t *
p_params, const char *name, const char *value);
```

Copy Session context

```
/*!*****************************************************************
********
*   \brief    Copy existing decoding session params for hw frame usage
*
*   \param[in] src_p_ctx       Pointer to a caller allocated source session context
*   \param[in] dst_p_ctx       Pointer to a caller allocated destination session
*                                  context
*   \return On success
*                                  NI_RETCODE_SUCCESS
*               On failure
*                                  NI_RETCODE_INVALID_PARAM
*******************************************************************
******/ LIB_API ni_retcode_t ni_device_session_copy(ni_session_context_t *src_p_ctx,
ni_session_context_t *dst_p_ctx);
```

Setup uploader instance frame pool

```
/*!*****************************************************************
********
*   \brief    Send frame pool setup info to device
*
*   \param[in] p_ctx             Pointer to a caller allocated
*                                    ni_session_context_t struct
*   \param[in] pool_size       Upload session initial allocated frames count
*                                    must be > 0,
*   \return On success          Return code
*                On failure
*                                    NI_RETCODE_INVALID_PARAM
*                                    NI_RETCODE_ERROR_NVME_CMD_FAILED
*                                    NI_RETCODE_ERROR_INVALID_SESSION
*                                    NI_RETCODE_ERROR_MEM_ALOC
*******************************************************************
******/ LIB_API int ni_device_session_init_framepool(ni_session_context_t *p_ctx,
uint32_t pool_size);
```

Adjust uploader instance frame pool

```
/*!*****************************************************************
********
*   \brief    Sends frame pool change info to device
*
*   \param[in] p_ctx             Pointer to a caller allocated
*                                       ni_session_context_t struct
*   \param[in] pool_size       if pool_size = 0, free allocated device memory buffers
*                              if pool_size > 0, expand device frame buffer pool of
*                                       current instance with pool_size more frame
buffers
*
*   \return On success          Return code
*               On failure
*                                   NI_RETCODE_FAILURE
*                                   NI_RETCODE_INVALID_PARAM
*                                   NI_RETCODE_ERROR_NVME_CMD_FAILED
*                                   NI_RETCODE_ERROR_INVALID_SESSION
*                                   NI_RETCODE_ERROR_MEM_ALOC
*                                   NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION
*****************************************************************
*******/
ni_retcode_t ni_device_session_update_framepool(ni_session_context_t *p_ctx,
                                                uint32_t pool_size)
```

Read decoder HW descriptor output or 2D engine output

```
/*!*****************************************************************
********
*   \brief    Read data from the device
*             If device_type is NI_DEVICE_TYPE_DECODER reads data hwdesc from
*             decoder
*             If device_type is NI_DEVICE_TYPE_SCALER reads data hwdesc from
*             scaler
*
*   \param[in] p_ctx            Pointer to a caller allocated
*                                 ni_session_context_t struct
*   \param[in] p_data           Pointer to a caller allocated
*                                 ni_session_data_io_t struct which contains either a
*                                 ni_frame_t data frame or ni_packet_t data packet to
*                                 send
*   \param[in] device_type    NI_DEVICE_TYPE_DECODER or NI_DEVICE_TYPE_SCALER
*                                 If NI_DEVICE_TYPE_DECODER or
NI_DEVICE_TYPE_SCALER is specified,
*                                 hw descriptor info will be stored in p_data ni_frame
*   \return On success
*                                 Total number of bytes read
*             On failure
*                                 NI_RETCODE_INVALID_PARAM
*                                 NI_RETCODE_ERROR_NVME_CMD_FAILED
*                                 NI_RETCODE_ERROR_INVALID_SESSION
*****************************************************************
******/LIB_API int ni_device_session_read_hwdesc(ni_session_context_t *p_ctx,
ni_session_data_io_t *p_data, ni_device_type_t device_type);
```

Download YUV frame from provided HW descriptor

```
/*!*****************************************************************
*******
*   \brief    Reads YUV data from hw descriptor stored location on device
*
*   \param[in] p_ctx            Pointer to a caller allocated
*                                                   ni_session_context_t struct
*   \param[in] p_data           Pointer to a caller allocated
*                                  ni_session_data_io_t struct which contains either a
*                                  ni_frame_t data frame or ni_packet_t data packet to
send
*   \param[in] hwdesc           HW descriptor to find frame in XCODER
*   \return On success
*                                  Total number of bytes read
*               On failure
*                                  NI_RETCODE_INVALID_PARAM
*                                  NI_RETCODE_ERROR_NVME_CMD_FAILED
*                                  NI_RETCODE_ERROR_INVALID_SESSION
*****************************************************************************
******/LIB_API int ni_device_session_hwdl(ni_session_context_t* p_ctx,
ni_session_data_io_t *p_data, niFrameSurface1_t* hwdesc);
```

Upload YUV frame from to device and return HW descriptor as token

```
/*!*****************************************************************
*******
*   \brief    Sends raw YUV input to uploader instance and retrieves a HW descriptor
*             to represent it
*
*   \param[in] p_ctx            Pointer to a caller allocated
*                                                    ni_session_context_t struct
*   \param[in] p_src_data     Pointer to a caller allocated
*                                ni_session_data_io_t struct which contains a
*                                ni_frame_t data frame to send to uploader
*   \param[out] hwdesc         HW descriptor to find frame in XCODER
*   \return On success
*                                Total number of bytes read
*             On failure
*                                NI_RETCODE_INVALID_PARAM
*                                NI_RETCODE_ERROR_NVME_CMD_FAILED
*                                NI_RETCODE_ERROR_INVALID_SESSION
*****************************************************************
******/LIB_API int ni_device_session_hwup(ni_session_context_t* p_ctx,
ni_session_data_io_t *p_src_data, niFrameSurface1_t* hwdesc);
```

Allocate HW descriptor buffer for encoder input usage

```
/*!*****************************************************************
********
*   \brief    Allocate memory for the hwDescriptor buffer based on provided
*             parameters taking into account pic size and extra data.
*
*   \param[in] p_frame          Pointer to a caller allocated ni_frame_t struct
*
*   \param[in] video_width      Width of the video frame
*   \param[in] video_height     Height of the video frame
*   \param[in] extra_len        Extra data size (incl. meta data)
*
*   \return On success
*                                 NI_RETCODE_SUCCESS
*             On failure
*                                 NI_RETCODE_INVALID_PARAM
*                                 NI_RETCODE_ERROR_MEM_ALOC
*****************************************************************
*****/ LIB_API ni_retcode_t ni_frame_buffer_alloc_hwenc(ni_frame_t *pframe, int
video_width, int video_height, int extra_len);
```

Recycle memory buffer referred to by HW descriptor

```
/*!*****************************************************************
*******
*   \brief    Recycle a frame buffer on card
*
*   \param[in] surface        Struct containing device and frame location to clear out
*   \param[in] device_handle    handle to access device memory buffer is stored in
*
*   \return On success        NI_RETCODE_SUCCESS
*                On failure        NI_RETCODE_INVALID_PARAM
*****************************************************************
******/ LIB_API ni_retcode_t ni_hwframe_buffer_recycle(niFrameSurface1_t* surface,
ni_device_handle_t device_handle);
```

Recycle memory buffer referred to by HW descriptor

```
/*!*****************************************************************
*******
*    \brief    Recycle a frame buffer on card, this is new version of
ni_hwframe_buffer_recycle
*
*    \param[in] surface        Struct containing device and frame location to clear out
*
*    \return On success        NI_RETCODE_SUCCESS
*                On failure        NI_RETCODE_INVALID_PARAM
*****************************************************************
******/ LIB_API ni_retcode_t ni_hwframe_buffer_recycle2(niFrameSurface1_t*
surface);
```

Set parameters on the device for the 2D engine

```
/*!*****************************************************************************
*******
 *   \brief    Set parameters on the device for the 2D engine
 *
 *   \param[in]    p_ctx           pointer to session context
 *   \param[in]    p_params        pointer to the scaler parameters
 *
 *   \return              NI_RETCODE_INVALID_PARAM
 *                        NI_RETCODE_ERROR_INVALID_SESSION
 *                        NI_RETCODE_ERROR_NVME_CMD_FAILED
*****************************************************************************
******/
LIB_API ni_retcode_t ni_scaler_set_params(ni_session_context_t *p_ctx,
                                          ni_scaler_params_t *p_params);
```

Allocate a frame buffer on device for 2D engine output space

```
/*!*****************************************************************************
*******
 *   \brief    Allocate a frame on the device for 2D engine or AI engine
 *               to work on based on provided parameters
 *   \param[in]    p_ctx            pointer to session context
 *   \param[in]    width            width, in pixels
 *   \param[in]    height           height, in pixels
 *   \param[in]    format           pixel format
 *   \param[in]    options          options bitmap flags, bit 0 (NI_SCALER_FLAG_IO) is
 *                   0=input frame or 1=output frame. Bit 1 (NI_SCALER_FLAG_PC) is
 *                   0=single allocation, 1=create pool. Bit 2 (NI_SCALER_FLAG_PA) is
 *                   0=straight alpha, 1=premultiplied alpha
 *   \param[in]    rectangle_width         clipping rectangle width
 *   \param[in]    rectangle_height        clipping rectangle height
 *   \param[in]    rectangle_x             horizontal position of clipping rectangle
 *   \param[in]    rectangle_y             vertical position of clipping rectangle
 *   \param[in]    rgba_color              RGBA fill colour (for padding only)
 *   \param[in]    frame_index             input hwdesc index
 *   \param[in]    device_type             only NI_DEVICE_TYPE_SCALER
 *                   and NI_DEVICE_TYPE_AI (only needs p_ctx and frame_index)
 *
 *   \return             NI_RETCODE_INVALID_PARAM
 *                         NI_RETCODE_ERROR_INVALID_SESSION
 *                         NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                         NI_RETCODE_ERROR_MEM_ALOC
*******************************************************************************
******/LIB_API ni_retcode_t ni_device_alloc_frame(ni_session_context_t* p_ctx,
                                                int width,
                                                int height,
                                                int format,
                                                int options,
                                                int rectangle_width,
                                                int rectangle_height,
```

```
                                                int rectangle_x,
                                                int rectangle_y,
                                                int rgba_color,
                                                int frame_index,
                                                ni_device_type_t device_type);
```

Allocate a frame on the device and return the frame index

```
/*!*****************************************************************************
********
 *   \brief    Allocate a frame on the device and return the frame index
 *
 *   \param[in]    p_ctx    pointer to session context
 *   \param[in]    p_out_surface    pointer to output frame surface
 *   \param[in]    device_type       currently only NI_DEVICE_TYPE_AI
 *
 *   \return              NI_RETCODE_INVALID_PARAM
 *                          NI_RETCODE_ERROR_INVALID_SESSION
 *                          NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION
 *                          NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                          NI_RETCODE_ERROR_MEM_ALOC

*****************************************************************************
******/
LIB_API ni_retcode_t ni_device_alloc_dst_frame(ni_session_context_t *p_ctx,
                                               niFrameSurface1_t *p_out_surface,
                                               ni_device_type_t device_type);
```

Copy the data of src hwframe to dst hwframe

```
/*!*****************************************************************************
********
 *   \brief    Copy the data of src hwframe to dst hwframe
 *
 *   \param[in]    p_ctx    pointer to session context
 *   \param[in]    p_frameclone_desc    pointer to the frameclone descriptor
 *
 *   \return               NI_RETCODE_INVALID_PARAM
 *                          NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION
 *                          NI_RETCODE_ERROR_INVALID_SESSION
 *                          NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                          NI_RETCODE_ERROR_MEM_ALOC

*******************************************************************************
******/
LIB_API ni_retcode_t ni_device_clone_hwframe(ni_session_context_t *p_ctx,
                                             ni_frameclone_desc_t
*p_frameclone_desc);
```

Configure a frame on the device for 2D engine to work on

```
/*!*****************************************************************************
*******
 *   \brief    Config a frame on the device for 2D engined
 *               to work on based on provided parameters
 *
 *   \param[in]    p_ctx           pointer to session context
 *   \param[in]    p_cfg           pointer to frame config
 *
 *   \return             NI_RETCODE_INVALID_PARAM
 *                         NI_RETCODE_ERROR_INVALID_SESSION
 *                         NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                         NI_RETCODE_ERROR_MEM_ALOC
*******************************************************************************
******/
LIB_API ni_retcode_t ni_device_config_frame(ni_session_context_t *p_ctx,
                                            ni_frame_config_t *p_cfg);
```

Configure multiple frames on the device for 2D engine to work on

```
/*!*****************************************************************
*******
 *   \brief    Config multiple frame on the device for 2D engined
 *               to work on based on provided parameters
 *
 *   \param[in]   p_ctx            pointer to session context
 *   \param[in]   p_cfg_in         input frame config array
 *   \param[in]   numInCfgs        number of frame config entries in the p_cfg_in array
 *   \param[in]   p_cfg_out        output frame config
 *
 *   \return              NI_RETCODE_INVALID_PARAM
 *                          NI_RETCODE_ERROR_INVALID_SESSION
 *                          NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                          NI_RETCODE_ERROR_MEM_ALOC
*****************************************************************
******/
LIB_API ni_retcode_t ni_device_multi_config_frame(ni_session_context_t *p_ctx,
                                                  ni_frame_config_t
p_cfg_in[],
                                                  int numInCfgs,
                                                  ni_frame_config_t
*p_cfg_out);
```

Allocate frame buffer to be used for sending input YUV to uploader

```
/*!*****************************************************************
*******
 *   \brief     Allocate memory for the frame buffer based on provided parameters
 *                taking into account the pixel format, width, height, stride,
 *                alignment, and extra data
 *   \param[in]   p_frame              Pointer to caller allocated ni_frame_t
 *   \param[in]   pixel_format       a pixel format in ni_pix_fmt_t enum
 *   \param[in]   video_width         width, in pixels
 *   \param[in]   video_height       height, in pixels
 *   \param[in]   linesize             horizontal stride
 *   \param[in]   alignment           apply a 16 pixel height alignment (T408 only)
 *   \param[in]   extra_len           meta data size
 *
 *   \return        NI_RETCODE_SUCCESS
 *                    NI_RETCODE_INVALID_PARAM
 *                    NI_RETCODE_ERROR_MEM_ALOC
 *
*******************************************************************************
******/LIB_API ni_retcode_t ni_frame_buffer_alloc_pixfmt(ni_frame_t *pframe,
                                                       int pixel_format,
                                                       int video_width,
                                                       int video_height,
                                                       int linesize[],
                                                       int alignment,
                                                       int extra_len);
```

Config AI session from file

```
/*!*****************************************************************************
*******
 *   \brief      configure a network context based with the network binary
 *
 *   \param[in]    p_ctx                 Pointer to caller allocated ni_session_context_t
 *   \param[in]    file                  Pointer to caller network binary file path
 *
 *   \return       NI_RETCODE_SUCCESS
 *                   NI_RETCODE_INVALID_PARAM
 *                   NI_RETCODE_ERROR_MEM_ALOC
 *                   NI_RETCODE_ERROR_INVALID_SESSION
 *                   NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                   NI_RETCODE_FAILURE
 *
******************************************************************************
******/LIB_API ni_retcode_t ni_ai_config_network_binary(ni_session_context_t
*p_ctx,
                                                      ni_network_data_t
*p_network,
                                                      const char *file);
```

Allocate input frame buffer for AI

```
/*!*****************************************************************************
 *    \brief      Allocate input layers memory for AI frame buffer based on provided parameters
 *                taking into account width, height, format defined by network.
 *
 *    \param[out] p_frame              Pointer to caller allocated ni_frame_t
 *    \param[in]    p_network          Pointer to caller allocated ni_network_data_t
 *
 *    \return         NI_RETCODE_SUCCESS
 *                    NI_RETCODE_INVALID_PARAM
 *                    NI_RETCODE_ERROR_MEM_ALOC
 *
*******************************************************************************
******/LIB_API ni_retcode_t ni_ai_frame_buffer_alloc(ni_frame_t *p_frame,
                                                   ni_network_data_t
*p_network);
```

Allocate output buffer for AI

```
/*!*****************************************************************************
*******
 *   \brief    Allocate output layers memory for the packet buffer based on provided
network
 *
 *   \param[out] p_packet         Pointer to a caller allocated
 *                                                    ni_packet_t struct
 *   \param[in] p_network         Pointer to a caller allocated
 *                                                    ni_network_data_t struct
 *
 *   \return On success
 *                                NI_RETCODE_SUCCESS
 *               On failure
 *                                NI_RETCODE_INVALID_PARAM
 *                                NI_RETCODE_ERROR_MEM_ALOC
*******************************************************************************
******/LIB_API ni_retcode_t ni_ai_packet_buffer_alloc(ni_packet_t *p_packet,
                                                    ni_network_data_t
*p_network);
```

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_xcoder_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_RETCODE_SUCCESS
 *                        On failure
 *                                  NI_RETCODE_FAILURE
 *                                  NI_RETCODE_INVALID_PARAM

*****************************************************************
*****/
LIB_API ni_retcode_t ni_encoder_gop_params_set_value(ni_xcoder_params_t *
p_params, const char *name, char *value);
```

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_frame_new_aux_data(ni_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_frame_new_aux_data_from_raw_data(
   ni_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_frame_get_aux_data(const ni_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_frame_free_aux_data(ni_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_frame_wipe_aux_data(ni_frame_t *frame);
```

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_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_get_hw_yuv420p_dim(int width, int height, int bit_depth_factor,
                                   int is_h264,
                                   int
plane_stride[NI_MAX_NUM_DATA_POINTERS],
                                   int
plane_height[NI_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_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_copy_hw_yuv420p(uint8_t *p_dst[NI_MAX_NUM_DATA_POINTERS],
                                uint8_t *p_src[NI_MAX_NUM_DATA_POINTERS],
                                int width, int height, int bit_depth_factor,
                                int dst_stride[NI_MAX_NUM_DATA_POINTERS],
                                int dst_height[NI_MAX_NUM_DATA_POINTERS],
                                int src_stride[NI_MAX_NUM_DATA_POINTERS],
                                int src_height[NI_MAX_NUM_DATA_POINTERS]);
```

Copy yuv444p data to NETINT two pieces of HW YUV420p frames layout

```
/*!*****************************************************************
********
*   \brief    Copy yuv444p data to yuv420p frame layout to be sent
*             to encoder for encoding. Data buffer (dst) is usually allocated by
*             ni_encoder_frame_buffer_alloc.
*
*   \param[out]    p_dst0 pointers of Y/Cb/Cr as yuv420p output0
*   \param[out]    p_dst1 pointers of Y/Cb/Cr as yuv420p output1
*   \param[in]       p_src pointers of Y/Cb/Cr as yuv444p intput
*   \param[in]       width    source YUV frame width
*   \param[in]       height source YUV frame height
*   \param[in]       factor    1 for 8 bit, 2 for 10 bit
*   \param[in]       mode 0 for
*                      out0 is Y+1/2V, with the original input as the out0, 1/4V
*                      copy to data[1] 1/4V copy to data[2]
*                      out1 is U+1/2V, U copy to data[0], 1/4V copy to data[1], 1/4V
*                      copy to data[2]
*                      mode 1 for
*                      out0 is Y+1/2u+1/2v, with the original input as the output0,
*                      1/4U copy to data[1] 1/4V copy to data[2]
*                      out1 is (1/2U+1/2V)+1/4U+1/4V, 1/2U & 1/2V copy to data[0],
*                      1/4U copy to data[1], 1/4V copy to data[2]
*
*   \return Y/Cb/Cr data
*
*****************************************************************
******/
LIB_API void ni_copy_yuv_444p_to_420p(uint8_t
*p_dst0[NI_MAX_NUM_DATA_POINTERS],
uint8_t *p_dst1[NI_MAX_NUM_DATA_POINTERS],
uint8_t *p_src[NI_MAX_NUM_DATA_POINTERS],

int width, int height, int factor, int mode);
```

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 ni_insert_emulation_prevent_bytes(uint8_t *buf, int size);
```

Remove emulation prevention byte(s) into data buffer

```
/*!*****************************************************************
*   \brief    Remove emulation prevention byte(s) as needed from the data buffer
*
*   \param    buf      data buffer to be worked on - emu prevent byte(s) will be
*                      removed from.
*             size     number of bytes starting from buf to check
*
*   \return the number of emulation prevention bytes removed from buf, 0 if
*             none.
*
*   Note: buf will be modified if emu prevent byte(s) found and removed.
*****************************************************************
******/
LIB_API int ni_remove_emulation_prevent_bytes(uint8_t *buf, int size);
```

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_should_send_sei_with_frame(ni_session_context_t* p_enc_ctx,
                                          ni_pic_type_t pic_type,
                                          ni_xcoder_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_dec_retrieve_aux_data(ni_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_enc_prep_aux_data(ni_session_context_t* p_enc_ctx,
                                  ni_frame_t *p_enc_frame,
                                  ni_frame_t *p_dec_frame,
                                  ni_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+
*   \param[in]    is_hwframe, must be 0 (sw frame) or 1 (hw frame)
 *
 *   \return NONE
*******************************************************************************
******/
LIB_API void ni_enc_copy_aux_data(ni_session_context_t* p_enc_ctx,
                                  ni_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,
                                  int is_hwframe,
                                  int is_nv12frame);
```

Insert timecode info into picture timing SEI (for H264) or time code SEI (for H265)

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

* \brief Insert timecode data into picture timing SEI (H264) or time code SEI (H265)

*

* \note This function must be callled after all other aux data has been processed by

* ni_enc_prep_aux_data and ni_enc_copy_aux_data. Otherwise the timecode SEI data

* might be overwritten

*

* \param[in] p_enc_ctx encoder session context

* \param[out] p_enc_frame frame to be sent to encoder

* \param[in] p_timecode the timecode data to be written along with the frame

*

* \return NI_RETCODE_SUCCESS on success, NI_RETCODE_FAILURE on failure

*****************************************************************
******/

LIB_API int ni_enc_insert_timecode(ni_session_context_t *p_enc_ctx,

                                   ni_frame_t *p_enc_frame,

                                   ni_timecode_t *p_timecode);
```

Reconfigure bitrate dynamically during encoding

```
/*!*****************************************************************************
*******
*    \brief    Reconfigure bitrate dynamically during encoding.
*
*    \param[in] p_ctx          Pointer to caller allocated ni_session_context_t
*    \param[in] bitrate        Target bitrate to set
*
*    \return On success        NI_RETCODE_SUCCESS
*                On failure        NI_RETCODE_INVALID_PARAM
******************************************************************************
******/
LIB_API ni_retcode_t ni_reconfig_bitrate(ni_session_context_t *p_ctx,
                                                        int32_t bitrate);
```

Reconfigure intra period dynamically during encoding

```
/*!*****************************************************************************
*******
*    \brief    Reconfigure intraPeriod dynamically during encoding.
*
*    \param[in] p_ctx          Pointer to caller allocated ni_session_context_t
*    \param[in] intra_period       Target intra period to set
*
*    \return On success        NI_RETCODE_SUCCESS
*                On failure        NI_RETCODE_INVALID_PARAM
*
*    NOTE - the frame upon which intra period is reconfigured is encoded as IDR frame
*    NOTE - reconfigure intra period is not allowed if intraRefreshMode is enabled or if
gopPresetIdx is 1
*
*****************************************************************************
******/
LIB_API ni_retcode_t r(ni_session_context_t *p_ctx,
                                                    int32_t intra_period);
```

Reconfigure VUI parameters dynamically during encoding

```
/*!*****************************************************************
*******
*   \brief    Reconfigure VUI parameters dynamically during encoding.
*
*   \param[in] p_ctx          Pointer to caller allocated ni_session_context_t
*   \param[in] vui                Pointer to struct specifying vui parameters
*
*   \return On success       NI_RETCODE_SUCCESS
*               On failure       NI_RETCODE_INVALID_PARAM
*******************************************************************
******/
LIB_API ni_retcode_t ni_reconfig_vui(ni_session_context_t *p_ctx,
                                                    ni_vui_hrd_t *vui);
```

Force next frame to be IDR frame during encoding

```
/*!*****************************************************************
*******
* \brief Force next frame to be IDR frame during encoding.
*
* \param[in] p_ctx Pointer to caller allocated ni_session_context_t
*
* \return On success NI_RETCODE_SUCCESS
*******************************************************************
******/
LIB_API ni_retcode_t ni_force_idr_frame_type(ni_session_context_t *p_ctx);
```

Set a frame’s support of Long Term Reference frame during encoding

```
/*!*****************************************************************
*******
* \brief Set a frame's support of Long Term Reference frame during encoding.
*
* \param[in] p_ctx Pointer to caller allocated ni_session_context_t
* \param[in] ltr Pointer to struct specifying LTR support
*
* \return On success NI_RETCODE_SUCCESS
* On failure NI_RETCODE_INVALID_PARAM
*****************************************************************
******/
LIB_API ni_retcode_t ni_set_ltr(ni_session_context_t *p_ctx,
ni_long_term_ref_t *ltr);
```

Set Long Term Reference interval

```
/*!*****************************************************************************
*******
* \brief Set Long Term Reference interval
*
* \param[in] p_ctx Pointer to caller allocated ni_session_context_t
* \param[in] ltr_interval the new long term reference inteval value
*
* \return On success NI_RETCODE_SUCCESS
* On failure NI_RETCODE_INVALID_PARAM
******************************************************************************
******/
LIB_API ni_retcode_t ni_set_ltr_interval(ni_session_context_t *p_ctx,
int32_t ltr_interval);
```

Set frame reference invalidation

/*!***
****

* \brief Set frame reference invalidation

* \param[in] p_ctx Pointer to caller allocated ni_session_context_t

* \param[in] frame_num frame number after which all references shall be

* invalidated

* NOTE – frame number should be incremented for every frame sent to the encoder (regardless * if encoder drops the output due to picSkip / maxFrameSize settings or not), and should not be * reset to 0 upon I-frame (unlike poc)

* \return On success NI_RETCODE_SUCCESS

* On failure NI_RETCODE_INVALID_PARAM

****
******/

LIB_API ni_retcode_t ni_set_frame_ref_invalid(ni_session_context_t *p_ctx,

int32_t frame_num);

Reconfigure framerate dynamically during encoding

```
/*!*****************************************************************************
*******
*    \brief    Reconfigure framerate dynamically during encoding.
*
*    \param[in] p_ctx          Pointer to caller allocated ni_session_context_t
*    \param[in] framerate      Pointer to struct specifying framerate numerator /
denominator
*
*    \return On success       NI_RETCODE_SUCCESS
*                On failure       NI_RETCODE_INVALID_PARAM
*****************************************************************************
******/
LIB_API ni_retcode_t ni_reconfig_framerate(ni_session_context_t *p_ctx,
                                           ni_framerate_t *framerate);
```

Reconfigure maxFrameSize parameter setting during encoding

```
/*!*****************************************************************************
*******
*    \brief    Reconfigure maxFrameSize parameter setting during encoding.
*
*    \param[in] p_ctx          Pointer to caller allocated ni_session_context_t
*    \param[in] max_frame_size              the new maxFrameSize value
*
*    \return On success       NI_RETCODE_SUCCESS
*                On failure       NI_RETCODE_INVALID_PARAM
*
*    NOTE - maxFrameSize_Bytes value less than ((bitrate / 8) / framerate) will be
rejected
*
******************************************************************************
******
LIB_API ni_retcode_t ni_reconfig_max_frame_size(ni_session_context_t *p_ctx,
                                                int32_t max_frame_size);
```

Reconfigure crf setting during encoding

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

 *   \brief    Reconfigure crf value dynamically during encoding.

 *

 *   \param[in] p_ctx              Pointer to caller allocated ni_session_context_t

 *   \param[in] crf                  crf value to reconfigure

 *

 *   \return On success       NI_RETCODE_SUCCESS

 *                On failure       NI_RETCODE_INVALID_PARAM

*******************************************************************************
******/ LIB_API ni_retcode_t ni_reconfig_crf(ni_session_context_t *p_ctx,

                                           int32_t crf);
```

Reconfigure float point crf setting during encoding

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

*    \brief    Reconfigure crf float point value dynamically during encoding.

*

*    \param[in] p_ctx              Pointer to caller allocated ni_session_context_t

*    \param[in] crf                  float point crf value to reconfigure

*

*    \return On success       NI_RETCODE_SUCCESS

*                On failure       NI_RETCODE_INVALID_PARAM

*****************************************************************************
******/

ni_retcode_t ni_reconfig_crf2(ni_session_context_t *p_ctx,

                                        float crf)
```

Reconfigure VBV buffer size and VBV max rate setting during encoding

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

*    \brief    Reconfigure vbv buffer size and vbv max rate dynamically during encoding.

*

*    \param[in] p_ctx           Pointer to caller allocated ni_session_context_t

*    \param[in] vbvBufferSize Target vbvBufferSize to set

*    \param[in] vbvMaxRate       Target vbvMaxRate to set

*

*    \return On success       NI_RETCODE_SUCCESS

*                 On failure       NI_RETCODE_INVALID_PARAM

*****************************************************************************
******/

LIB_API ni_retcode_t ni_reconfig_vbv_value(ni_session_context_t *p_ctx,

                                                    int32_t vbvMaxRate, int32_t
vbvBufferSize);
```

Reconfigure maxFrameSizeRatio parameter setting during encoding

```
/*!*************************************************************************
*******
*   \brief    Reconfigure maxFrameSizeRatio parameter setting during encoding.
*
*   \param[in] p_ctx          Pointer to caller allocated ni_session_context_t
*   \param[in] max_frame_size_ratio              the new maxFrameSizeRatio value
*
*   \return On success       NI_RETCODE_SUCCESS
*               On failure       NI_RETCODE_INVALID_PARAM
*************************************************************************
LIB_API ni_retcode_t ni_reconfig_max_frame_size_ratio(ni_session_context_t *p_ctx,
                                                      int32_t max_frame_size_ratio);
/*!*************************************************************************
*   \brief    Reconfigure sliceArg dynamically during encoding.
*
*   \param[in] p_ctx          Pointer to caller allocated ni_session_context_t
*   \param[in] sliceArg              the new sliceArg value
*
*   \return On success       NI_RETCODE_SUCCESS
*               On failure       NI_RETCODE_INVALID_PARAM
*************************************************************************/
LIB_API ni_retcode_t ni_reconfig_slice_arg(ni_session_context_t *p_ctx,
                                           int16_t sliceArg);
```

Acquire a frame buffer from a P2P hwupload session

```
/*!*****************************************************************
*******
*   \brief    Acquire a frame buffer from the hwupload session
*
*   \param[in] p_upl_ctx       Pointer to a caller allocated
*                                                    ni_session_context_t struct
*   \param[out] p_frame        Pointer to a caller allocated hw frame
*
*   \return On success
*                                NI_RETCODE_SUCCESS
*              On failure
*                                NI_RETCODE_INVALID_PARAM
*                                NI_RETCODE_ERROR_NVME_CMD_FAILED
*                                NI_RETCODE_ERROR_INVALID_SESSION
*****************************************************************
******/
LIB_API int ni_device_session_acquire(ni_session_context_t *p_upl_ctx, ni_frame_t
*p_frame);
```

Lock a hardware P2P frame

```
/*!*****************************************************************************
*******
*   \brief    Lock a hardware P2P frame prior to encoding
*
*   \param[in] p_upl_ctx       pointer to caller allocated upload context
*   \param[in] p_frame          pointer to caller allocated hardware P2P frame
*
*   \return On success
*                                   NI_RETCODE_SUCCESS
*                       On failure
*                                   NI_RETCODE_FAILURE
*                                   NI_RETCODE_INVALID_PARAM
*****************************************************************************
******/
LIB_API ni_retcode_t ni_uploader_frame_buffer_lock(ni_session_context_t *p_upl_ctx,
ni_frame_t *p_frame);
```

Unlock a hardware P2P frame

```
/*!*****************************************************************************
*******
*   \brief    Unlock a hardware P2P frame after encoding
*
*   \param[in] p_upl_ctx        pointer to caller allocated upload context
*   \param[in] p_frame          pointer to caller allocated hardware P2P frame
*
*   \return On success
*                                   NI_RETCODE_SUCCESS
*                       On failure
*                                   NI_RETCODE_FAILURE
*                                   NI_RETCODE_INVALID_PARAM
*****************************************************************************
******/
LIB_API ni_retcode_t ni_uploader_frame_buffer_unlock(
      ni_session_context_t *p_upl_ctx,
      ni_frame_t *p_frame);
```

Special P2P test API call. Copy YUV data from the software frame to the hardware P2P frame

```
/*!*****************************************************************************
 *  \brief  Special P2P test API call. Copies YUV data from the software
 *          frame to the hardware P2P frame on the Quadra device
 *
 *  \param[in] p_upl_ctx      pointer to caller allocated uploader session
 *                            context
 *         [in] p_swframe     pointer to a caller allocated software frame
 *         [in] p_hwframe     pointer to a caller allocated hardware frame
 *
 *  \return On success
 *                            NI_RETCODE_SUCCESS
 *          On failure
 *                            NI_RETCODE_FAILURE
 *                            NI_RETCODE_INVALID_PARAM
 ******************************************************************************/
LIB_API ni_retcode_t ni_uploader_p2p_test_send(ni_session_context_t *p_upl_ctx,
                                               uint8_t *p_data, uint32_t len,
                                               ni_frame_t *p_hwframe);
```

Set the incoming frame format for encoder

```
/*!*****************************************************************************
*   \brief    Set the incoming frame format for the encoder
*
*   \param[in] p_enc_ctx       pointer to encoder context
*   \param[in] p_enc_params pointer to encoder parameters
*   \param[in] width                  input width
*   \param[in] height                 input height
*   \param[in] bit_depth       8 for 8-bit YUV, 10 for 10-bit YUV
*   \param[in] src_endian    NI_FRAME_LITTLE_ENDIAN or NI_FRAME_BIG_ENDIAN
*   \param[in] planar                 0 for semi-planar YUV, 1 for planar YUV
*   \return On success
*                                NI_RETCODE_SUCCESS
*                          On failure
*                                NI_RETCODE_INVALID_PARAM
******************************************************************************/
LIB_API ni_retcode_t ni_encoder_set_input_frame_format(
      ni_session_context_t *p_enc_ctx,
      ni_xcoder_params_t *p_enc_params,
      int width,
      int height,
      int bit_depth,
      int src_endian,
     int planar);
```

Set the frame format for the uploader session

```
/*!*****************************************************************************
*******
*   \brief    Set the frame format for the uploader
*
*   \param[in]    p_upl_ctx           pointer to uploader context
*   \param[in]    width                    width
*   \param[in]    height                   height
*   \param[in]    pixel_format    pixel format
*   \param[in]    isP2P                    0 = normal, 1 = P2P
*
*   \return On success
*                           NI_RETCODE_SUCCESS
*
*                       On failure
*                           NI_RETCODE_INVALID_PARAM
*****************************************************************************
******/
LIB_API ni_retcode_t ni_uploader_set_frame_format(
      ni_session_context_t *p_upl_ctx,
      int width,
      int height,
      ni_pix_fmt_t pixel_format,
      int isP2P);
```

Recycle hardware P2P frames

```
/*!*****************************************************************************
*******
*   \brief    Recycle hw P2P frames
*
*   \param [in] p_frame         pointer to an acquired P2P hw frame
*
*   \return On success
*                               NI_RETCODE_SUCCESS
*
*                       On failure
*                               NI_RETCODE_INVALID_PARAM
******************************************************************************
******/
LIB_API ni_retcode_t ni_hwframe_p2p_buffer_recycle(ni_frame_t *p_frame);
```

Read initial stream header from the encoder

This is a convenience function that can be used instead of ni_device_session_read() to read the initial stream header from the encoder session. This function should be called exactly once after which the caller should use ni_device_session_read() to retrieve packets from the encoder session.

```
/*!*****************************************************************************
 *  \brief  Read encoder stream header from the device
 *  \param[in] p_ctx            Pointer to a caller allocated
 *                                  ni_session_context_t struct from encoder
 *  \param[in] p_data       Pointer to a caller allocated ni_session_data_io_t
 *                                  struct which contains a ni_packet_t data packet to
 *                                  receive
 *  \return On success
 *                                  Total number of bytes read
 *              On failure
 *                                  NI_RETCODE_INVALID_PARAM
 *                                  NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                                  NI_RETCODE_ERROR_INVALID_SESSION
******************************************************************************/
LIB_API int ni_encoder_session_read_stream_header(ni_session_context_t *p_ctx,
                                                  ni_session_data_io_t *p_data);
```

Get the DMA buffer file descriptor from the P2P frame

```
/*!*****************************************************************************
********
*   \brief    Get the DMA buffer file descriptor from the P2P frame
*
*   \param[in]    p_frame         pointer to a P2P frame
*
*   \return         On success
*                                   DMA buffer file descriptor
*                    On failure
*                                   NI_RETCODE_INVALID_PARAM
******************************************************************************
******/
LIB_API int32_t ni_get_dma_buf_file_descriptor(const ni_frame_t* p_frame);
```

Send sequence / resolution change information to device

NOTE - this API is for large to small resolution change only – small to large resolution change requires session close and session open

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

*   \brief    Send sequence change information to device

*

*   \param[in] p_ctx              Pointer to a caller allocated

*                                     ni_session_context_t struct

*   \param[in] width              input width

*   \param[in] height             input height

*   \param[in] bit_depth_factor       1 for 8-bit YUV, 2 for 10-bit YUV

    *   \param[in] device_type       device type (must be encoder)

*   \return On success

*                                     NI_RETCODE_SUCCESS

*                 On failure

*                                     NI_RETCODE_INVALID_PARAM

*                                     NI_RETCODE_ERROR_MEM_ALOC

*                                     NI_RETCODE_ERROR_NVME_CMD_FAILED

*                                     NI_RETCODE_ERROR_INVALID_SESSION

*****************************************************************
******/

LIB_API ni_retcode_t ni_device_session_sequence_change(ni_session_context_t *p_ctx,

                                                       int width, int height, int
bit_depth_factor, ni_device_type_t device_type);
```

Query NVMe/TP load from the device

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

* \brief Query NVMe load from the device

*

* \param[in] p_ctx Pointer to a caller allocated

* ni_session_context_t struct

* \param[in] p_load_query Pointer to a caller allocated

* ni_load_query_t struct

*

* \return On success

* NI_RETCODE_SUCCESS

* On failure

* NI_RETCODE_INVALID_PARAM

* NI_RETCODE_ERROR_NVME_CMD_FAILED

* NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION

* NI_RETCODE_ERROR_MEM_ALOC

*****************************************************************
******/

LIB_API ni_retcode_t ni_query_nvme_status(

ni_session_context_t *p_ctx, ni_load_query_t *p_load_query);
```

Send an input data frame to the encoder with YUV data given in the inputs.

```
/*!*****************************************************************************
*******
 *  \brief  Send an input data frame to the encoder with YUV data given in
 *   the inputs.
 *
 *   For ideal performance memory should be 4k aligned. If it is not 4K aligned
 *   then a temporary 4k aligned memory will be used to copy data to and from
 *   when writing and reading. This will negatively impact performance.
 *   Any metadata to be sent with the frame should be attached to p_enc_frame
 *   as aux data (e.g. using ni_frame_new_aux_data()).
 *  \param[in] p_ctx                    Encoder session context
 *  \param[in] p_enc_frame              Struct information about the
frame
 *                                      to be sent to the encoder
 *  \param[in] p_yuv_buffer             Caller allocated buffer holding YUV
data
 *                                      for the frame
 *  \return On success
 *                              Total number of bytes written
 *          On failure
 *                              NI_RETCODE_INVALID_PARAM
 *                              NI_RETCODE_ERROR_MEM_ALOC
 *                              NI_RETCODE_ERROR_NVME_CMD_FAILED
 *                              NI_RETCODE_ERROR_INVALID_SESSION
```

```
*****************************************************************************
*****/
int ni_enc_write_from_yuv_buffer(ni_session_context_t *p_ctx,
                                 ni_frame_t *p_enc_frame, uint8_t
*p_yuv_buffer)
```

Calculate the total number of dimensions of network layers.

```
/*!*****************************************************************************
********
*   \brief    Calculate the total dimension of network layers
*
*   \param[in] p_param                    Network layer parameter structure
*   \return
*                                         Total number of dimensions
*
*****************************************************************************
******/
LIB_API uint32_t ni_ai_network_layer_dims(ni_network_layer_params_t *p_param);
```

Convert tensors in the tensor files to format data accepted by networks.

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

*    \brief    Convert tensors in the tensor files to format data accepted by networks

*

*    \param[in] dst                                format data buffer pointer

*    \param[in] dst_len                            format data buffer size

*    \param[in] tensor_file                        tensor file path and file name

*    \param[in] p_param                            network layer parameter structure

*    \return On success

*                                                  NI_RETCODE_SUCCESS

*              On failure

*                                                  NI_RETCODE_INVALID_PARAM

*                                                  NI_RETCODE_FAILURE

*

*****************************************************************
******/

LIB_API ni_retcode_t ni_network_layer_convert_tensor(

                              uint8_t *dst, uint32_t dst_len, const char *tensor_file,

                              ni_network_layer_params_t *p_param);
```

Convert tensors in the tensor buffer to format data accepted by networks.

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

*   \brief  Convert tensors in the tensor buffer to format data accepted by networks.

*

*

*   \param[in] dst                          format data buffer pointer

*   \param[in] dst_len                      format data buffer size

*   \param[in] src                          tensor buffer pointer

*   \param[in] src_len                      tensor buffer size

*   \param[in] p_param                      network layer parameter structure

*

*   \return On success

*                                           NI_RETCODE_SUCCESS

*           On failure

*                                           NI_RETCODE_INVALID_PARAM

*                                           NI_RETCODE_FAILURE

*

*****************************************************************
******/

LIB_API ni_retcode_t ni_network_layer_convert_tensor(

                        uint8_t *dst, uint32_t dst_len, float *src, uint32_t src_len,

                        ni_network_layer_params_t *p_param);
```

Calculate the SHA256 value from data.

```
/*!*****************************************************************************
********
*   \brief    Calculate the SHA256 value from data
*
*   \param[in] aui8Data                         input data to be calculated with
*   \param[in] ui32DataLength                    input data length to be calculated with
*   \param[in] aui8Hash                         output SHA256 hash value
*   \return
*
*****************************************************************************
******/
LIB_API void ni_calculate_sha256(const uint8_t aui8Data[],
                                 size_t ui32DataLength, uint8_t aui8Hash[]);
```

Copy descriptor data to Netint HW descriptor frame layout to be sent to encoder for encoding.

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

*   \brief    Copy Descriptor data to Netint HW descriptor frame layout to be sent

*             to encoder for encoding. Data buffer (dst) is usually allocated by

*             ni_encoder_frame_buffer_alloc. Only necessary when metadata size in

*             source is insufficient

*

*   \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

*

*   \return descriptor data

*

*******************************************************************************
******/

LIB_API void ni_copy_hw_descriptors(uint8_t *p_dst[NI_MAX_NUM_DATA_POINTERS],

                                    uint8_t
*p_src[NI_MAX_NUM_DATA_POINTERS]);
```

Get libxcoder API version

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

 *  \brief  Get libxcoder API version

 *

 *  \return char pointer to libxcoder API version

*******************************************************************************
*****/

LIB_API char* ni_get_libxcoder_api_ver(void);
```

Get FW API version libxcoder is compatible with

```
/*!*****************************************************************************
 *  \brief  Get FW API version libxcoder is compatible with
 *
 *  \return char pointer to FW API version libxcoder is compatible with
 ******************************************************************************/
LIB_API char* ni_get_compat_fw_api_ver(void);
```

Get libxcoder SW release version

```
/*!*****************************************************************************
 *  \brief  Get libxcoder SW release version
 *
 *  \return char pointer to libxcoder SW release version
 ******************************************************************************/
LIB_API char* ni_get_libxcoder_release_ver(void);
```

Retrieve time for logs with microsecond timestamps.

```
/*!*****************************************************************************
 *    \brief Retrieve time for logs with microsecond timestamps
 *
 *    \param[in/out] p_tp      timeval struct
 *    \param[in] p_tzp                timezone info, can be NULL
 *
 *    \return On success
 *                                                    0
 *                        On failure
 *                                                    -1
 ******************************************************************************/
LIB_API int ni_gettimeofday(struct timeval *p_tp, void *p_tzp);
```

Retrieve the system time in nanoseconds.

```
/*!*****************************************************************************
*********
*   \brief    Retrieve the system clock time in nanoseconds
*
*   \param
*
*   \return
*                                                   system clock time in nanoseconds
******************************************************************************
******/
LIB_API uint64_t ni_gettime_ns(void);
```

Suspend execution for microsecond intervals.

```
/*!*****************************************************************************
*********
*   \brief    Suspend execution for microsecond intervals
*
*   \param[in] usec         sleep time intervals in microsecond
*
*   \return
******************************************************************************
******/
LIB_API void_t ni_usleep(uint64_t usec);
```

Allocate aligned memory.

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

*    \brief Allocate aligned memory

*

*    \param[in/out] memptr    The address of the allocated memory will be a multiple of
alignment,

*                                                    which must be a power of
two and a multiple of sizeof(void *). If size

*                                                    is 0, then    the value placed
is either NULL, or a unique pointer value

*                                                    that can later be successfully
passed to free.

*    \param[in] alignment          The alignment value of the allocated value.

*    \param[in] size                    The allocated memory size.

*

*    \return On success

*                                                    0

*                    On failure

*                                                    ENOMEM

*****************************************************************************
******/

LIB_API ni_posix_memalign(void **memptr, size_t alignment, size_t size);
```

Initialize a mutex.

```
/*!*****************************************************************************
*********
*   \brief    Initialize a mutex
*
*   \param[in] mutex          thread mutex
*
*   \return On success
*                                                   NI_RETCODE_SUCCESS
*               On failure
*                                                   NI_RETCODE_FAILURE
******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_mutex_init(ni_pthread_mutex_t *mutex);
```

Destroy a mutex.

```
/*!*****************************************************************************
*********
*   \brief    Destroy a mutex
*
*   \param[in] mutex       thread mutex
*
*   \return On success
*                                                   NI_RETCODE_SUCCESS
*               On failure
*                                                   NI_RETCODE_FAILURE
******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_mutex_destroy(ni_pthread_mutex_t *mutex);
```

Lock thread mutex.

```
/*!*****************************************************************************
*********
*   \brief    Thread mutex lock
*
*   \param[in] mutex       thread mutex to be locked
*
*   \return On success
*                                                   NI_RETCODE_SUCCESS
*               On failure
*                                                   NI_RETCODE_FAILURE
*****************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_mutex_lock(ni_pthread_mutex_t *mutex);
```

Unlock thread mutex.

```
/*!*****************************************************************************
*********
*   \brief    Thread mutex unlock
*
*   \param[in] mutex       thread mutex to be unlocked
*
*   \return On success
*                                                   NI_RETCODE_SUCCESS
*               On failure
*                                                   NI_RETCODE_FAILURE
*****************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_mutex_unlock(ni_pthread_mutex_t *mutex);
```

Create a new thread.

```
/*!*****************************************************************************
*********
*   \brief    Create a new thread
*
*   \param[in] thread                    thread id
*   \param[in] attr                      attributes to the new thread
*   \param[in] start_routine       entry of the thread routine
*   \param[in] arg                       sole argument of the routine
*
*   \return On success
*                                        NI_RETCODE_SUCCESS
*               On failure
*                                        NI_RETCODE_FAILURE
*******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_create(ni_pthread_t *thread,
                                       const ni_pthread_attr_t *attr,
                                       void *(*start_routine)(void *), void *arg);
```

Join with a terminated thread.

```
/*!*****************************************************************************
*********
*    \brief    Join with a terminated thread
*
*    \param[in] thread       thread id
*    \param[out] retval      return status
*
*    \return On success
*                                                    NI_RETCODE_SUCCESS
*                On failure
*                                                    NI_RETCODE_FAILURE
*****************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_join(ni_pthread_t thread, void **retval);
```

Initialize condition variable.

```
/*!*****************************************************************************
*********
*    \brief    Initialize condition variable
*
*    \param[in] cond              condition variable
*    \param[in] attr               attribute to the condvar
*
*    \return On success
*                                                    NI_RETCODE_SUCCESS
*                On failure
*                                                    NI_RETCODE_FAILURE
******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_cond_init(ni_pthread_cond_t *cond,
                                                    const ni_pthread_condattr_t *attr);
```

Destroy a condition variable

```
/*!*****************************************************************************
*********
*   \brief    Destroy condition variable
*
*   \param[in] cond              condition variable
*
*   \return On success
*                                                   NI_RETCODE_SUCCESS
*               On failure
*                                                   NI_RETCODE_FAILURE
******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_cond_destroy(ni_pthread_cond_t *cond);
```

Broadcast a condition.

```
/*!*****************************************************************************
*********
*   \brief    Broadcast a condition
*
*   \param[in] cond              condition variable
*
*   \return On success
*                                                    NI_RETCODE_SUCCESS
*               On failure
*                                                    NI_RETCODE_FAILURE
******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_cond_broadcast(ni_pthread_cond_t *cond);
```

Wait on a condition.

```
/*!*****************************************************************************
*********
*   \brief    Wait on a condition
*
*   \param[in] cond              condition variable
*   \param[in] mutex         mutex related to the condvar
*
*   \return On success
*                                                   NI_RETCODE_SUCCESS
*                On failure
*                                                     NI_RETCODE_FAILURE
******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_cond_wait(ni_pthread_cond_t *cond,

ni_pthread_mutex_t *mutex);
```

Signal a condition.

```
/*!*****************************************************************************
*********
*   \brief    Signal a condition
*
*   \param[in] cond       condition variable
*
*   \return On success
*                                                   NI_RETCODE_SUCCESS
*                On failure
*                                                   NI_RETCODE_FAILURE
******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_cond_signal(ni_pthread_cond_t *cond);
```

Wait on a condition with timeout.

```
/*!*****************************************************************************
*********
*   \brief    Wait on a condition with timeout.
*
*   \param[in] cond              condition variable
*   \param[in] mutex             mutex related to the condvar
*   \param[in] abstime      abstract value of timeout for waiting
*
*   \return On success
*                                                   NI_RETCODE_SUCCESS
*               On failure
*                                                   NI_RETCODE_FAILURE
*******************************************************************************
******/
LIB_API ni_retcode_t ni_pthread_cond_timedwait(ni_pthread_cond_t *cond,
                                               ni_pthread_mutex_t *mutex,
                                               const struct timespec *abstime);
```

Examine and change mask of blocked signals.

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

*    \brief    examine and change mask of blocked signals

*

*    \param[in] how          behavior of this call, can be value of SIG_BLOCK, SIG_UNBLOCK
and    SIG_SETMASK

*    \param[in] set          current value of the signal mask. If NULL, the mask keeps
unchanged.

*    \param[in] old_set    previous value of the signal mask, can be NULL.

*

*    \return On success

*                                                    NI_RETCODE_SUCCESS

*                On failure

*                                                    NI_RETCODE_FAILURE

*****************************************************************************
******/

LIB_API ni_retcode_t ni_pthread_sigmask(int how, const ni_sigset_t *set, ni_sigset_t
*old_set);
```

Get text string for the provided error

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

 *  \brief  Get text string for the provided error

 *

 *  \return char pointer for the provided error

*******************************************************************************
*****/

LIB_API const char *ni_get_rc_txt(ni_retcode_t rc);
```

Retrieve key and value from 'key=value' pair

```
/*!*****************************************************************************
********
 *  \brief  Retrieve key and value from 'key=value' pair
 *
 *  \param[in]    p_str      pointer to string to extract pair from
 *  \param[out]   key        pointer to key
 *  \param[out]   value      pointer to value
 *
 *  \return return 0 if successful, otherwise 1
 *

*******************************************************************************
*****/
LIB_API int ni_param_get_key_value(char *p_str, char *key, char *value);
```

Retrieve encoder config parameter values from --xcoder-params

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

 *   \brief    Retrieve encoder config parameter values from --xcoder-params

 *

 *   \param[in]      xcoderParams        pointer to string containing xcoder params

 *   \param[out]     params               pointer to xcoder params to fill out

 *   \param[out]     ctx                  pointer to session context

 *

 *   \return return 0 if successful, -1 otherwise

 *


*****************************************************************************
*****/

LIB_API int ni_retrieve_xcoder_params(char xcoderParams[],
                                      ni_xcoder_params_t *params,
                                      ni_session_context_t *ctx);
```

Retrieve custom gop config values from --xcoder-gop

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

 *   \brief    Retrieve custom gop config values from --xcoder-gop

 *

 *   \param[in]     xcoderGop       pointer to string containing xcoder gop

 *   \param[out]    params                pointer to xcoder params to fill out

 *   \param[out]    ctx                   pointer to session context

 *

 *   \return return 0 if successful, -1 otherwise

 *


*****************************************************************************
*****/

LIB_API int ni_retrieve_xcoder_gop(char xcoderGop[],

                                                    ni_xcoder_params_t *params,

                                                    ni_session_context_t *ctx);
```

Retrieve decoder config parameter values

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

 *   \brief    Retrieve decoder config parameter values from --decoder-params

 *

 *   \param[in]      xcoderParams        pointer to string containing xcoder params

 *   \param[out]     params               pointer to xcoder params to fill out

 *   \param[out]     ctx                  pointer to session context

 *

 *   \return return 0 if successful, -1 otherwise

 *


*****************************************************************************
*****/

LIB_API int ni_retrieve_decoder_params(char xcoderParams[],
                                       ni_xcoder_params_t *params,
                                       ni_session_context_t *ctx);
```

Check if encoder input frame is zero copy compatible or not (Also set encoder frame stride according to the input frame)

```
/*!*****************************************************************************
*   \brief    Check if incoming frame is encoder zero copy compatible or not
*   \param[in] p_enc_ctx       pointer to encoder context
*              [in] p_enc_params pointer to encoder parameters
*              [in] width        input width
*              [in] height       input height
*              [in] linesize     input linesizes (pointer to array)
*              [in] set_linesize setup linesizes 0 means not setup linesizes, 1 means setup linesizes (before encoder open)
*   \return on success and can do zero copy
*               NI_RETCODE_SUCCESS
*               cannot do zero copy
*               NI_RETCODE_ERROR_UNSUPPORTED_FEATURE
*               NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION
*               NI_RETCODE_INVALID_PARAM
******************************************************************************/
LIB_API ni_retcode_t ni_encoder_frame_zerocopy_check(ni_session_context_t *p_enc_ctx,
                                                     ni_xcoder_params_t *p_enc_params,
                                                     int width, int height,
                                                     const int linesize[],
                                                     bool set_linesize);
```

Allocate memory for encoder zero copy (metadata, etc.)

```
/*!*****************************************************************************
********
   *   \brief    Allocate memory for encoder zero copy (metadata, etc.)
   *             for encoding based on given
   *             parameters, taking into account pic linesize and extra data.
   *             Applicable to YUV planr / semi-planar 8 or 10 bit and RGBA pixel
formats.
   *
   *
   *   \param[in] p_frame          Pointer to a caller allocated ni_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] data             Picture data pointers (for each of YUV planes)
   *   \param[in] extra_len        Extra data size (incl. meta data)
   *
   *   \return On success
   *                                 NI_RETCODE_SUCCESS
   *             On failure
   *                                 NI_RETCODE_INVALID_PARAM
   *                                 NI_RETCODE_ERROR_MEM_ALOC

*****************************************************************************
*****/
```

```
LIB_API ni_retcode_t ni_encoder_frame_zerocopy_buffer_alloc(ni_frame_t *p_frame,
                                                            int video_width, int video_height,
                                                            const int linesize[], const uint8_t
*data[],
                                                            int extra_len);
```

Check if hwupload input frame is zero copy compatible or not

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

*   \brief    Check if incoming frame is hwupload zero copy compatible or not

*

*   \param[in] p_upl_ctx       pointer to uploader context

*              [in] width              input width

*              [in] height             input height

*              [in] linesize           input linesizes (pointer to array)

*              [in] pixel_format       input pixel format

*   \return on success and can do zero copy

*              NI_RETCODE_SUCCESS

*              cannot do zero copy

*              NI_RETCODE_ERROR_UNSUPPORTED_FEATURE

*              NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION

*              NI_RETCODE_INVALID_PARAM

*****************************************************************
*******/

LIB_API ni_retcode_t ni_uploader_frame_zerocopy_check(ni_session_context_t
*p_upl_ctx,

                                                      int width, int height,

                                                      const int linesize[], int
pixel_format);
```

Query Composite Temperature from device

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

*   \brief    Query CompositeTemp from device

*

*   \param[in] device_handle Device handle obtained by calling ni_device_open2()

*   \param[in] p_dev_temp       Pointer to device temperature

*   \param[in] fw_rev[]         Fw version to check if this function is supported

*

*   \return On success

*                                   NI_RETCODE_SUCCESS

*               On failure

*                                   NI_RETCODE_INVALID_PARAM

*                                   NI_RETCODE_ERROR_NVME_CMD_FAILED

*                                   NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION

*                                   NI_RETCODE_ERROR_MEM_ALOC

*****************************************************************
******/

LIB_API ni_retcode_t ni_query_temperature(ni_device_handle_t device_handle,

                                          ni_device_temp_t *p_dev_temp,

                                          uint8_t fw_rev[]);
```

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

*   \brief    Query CompositeTemp from device

*

*   \param[in] device_handle            Device handle obtained by calling
ni_device_open2()

*   \param[in] p_dev_extra_info       Pointer to device extra info

*   \param[in] fw_rev[]                 Fw version to check if this function is supported

*

*   \return On success

*                                   NI_RETCODE_SUCCESS

*               On failure

*                                   NI_RETCODE_INVALID_PARAM

*                                   NI_RETCODE_ERROR_NVME_CMD_FAILED

*                                   NI_RETCODE_ERROR_UNSUPPORTED_FW_VERSION

*                                   NI_RETCODE_ERROR_MEM_ALOC

******************************************************************************
******/

LIB_API ni_retcode_t ni_query_extra_info(ni_device_handle_t device_handle,

                                                    ni_device_extra_info_t
*p_dev_extra_info,

                                                    uint8_t fw_rev[]);
```

Retrieve firmware logs and write logs to files if requested

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

 *   \brief    Allocate log buffer if needed and retrieve firmware logs from device.
 *             Also write firmware logs to files if requested.
 *
 *   \param[in] p_ctx              Pointer to a caller allocated
 *                                 ni_session_context_t struct
 *   \param[in] p_log_buffer           Reference to pointer to a log buffer
 *                                 If log buffer pointer is NULL, this function will allocate
log buffer
 *                                 NOTE caller is responsible for freeing log buffer after
calling this function
 *   \param[in] gen_log_file           Indicating whether it is required to generate log
files
 *   \return on success
 *                   NI_RETCODE_SUCCESS
 *
 *               on failure
 *                   NI_RETCODE_ERROR_MEM_ALOC
 *                   NI_RETCODE_INVALID_PARAM
*****************************************************************
*******/

LIB_API ni_retcode_t ni_device_alloc_and_get_firmware_logs(ni_session_context_t
*p_ctx, void** p_log_buffer, bool gen_log_file);
```

Send a p_config command to reconfigure decoding ppu params

```
/*!*****************************************************************************
*********
*   \brief    Send a p_config command to reconfigure decoding ppu params.
*
*   \param      ni_session_context_t p_session_ctx - xcoder Context
*   \param      ni_xcoder_params_t p_param - xcoder Params
*   \param      ni_ppu_config_t p_ppu_config - Struct ni_ppu_config
*
*   \return - NI_RETCODE_SUCCESS on success, NI_RETCODE_ERROR_INVALID_SESSION,
NI_RETCODE_ERROR_NVME_CMD_FAILED on failure
*****************************************************************************
*******/
LIB_API ni_retcode_t ni_dec_reconfig_ppu_params(ni_session_context_t
*p_session_ctx,
                                                ni_xcoder_params_t *p_param,
                                                ni_ppu_config_t *p_ppu_config);
```

Create an instance of hw_device_info_coder_param_t used for ni_check_hw_info()

```
/*!*****************************************************************************
*********
* \brief Create a pointer to hw_device_info_coder_param_t instance .This instance will
be created
*and set to default vaule by param mode.You may change the resolution ,fps, bit_8_10
or other *vaule which is diferent from default value.
```

```
*
* \param[in] mode:0:create instance with decoder_param ,encoder_param will be set to
NULL
* 1:create instance with encoder_param ,decoder_param will be set to NULL
* 2:create instance with both decoder_param and encoder_param for
ni_check_hw_info() hw_mode
*
* \return NULL-error,pointer to an instance when success
**********************************************************************
*******/
LIB_API          ni_hw_device_info_quadra_coder_param_t*
ni_create_hw_device_info_quadra_coder_param(int mode);
```

Release resource in a pointer of hw_device_info_coder_param_t

```
/*!*****************************************************************
*********
* \brief Release a pointer to hw_device_info_coder_param_t instance created by
* create_hw_device_info_coder_param
*
*
* \param[in] p_hw_device_info_coder_param:pointer to a
hw_device_info_coder_param_t instance
* created by create_hw_device_info_coder_param
*
*\return
*****************************************************************
*******/
LIB_API void
ni_destory_hw_device_info_quadra_coder_param(ni_hw_device_info_quadra_coder_p
aram_t
                                             *p_hw_device_info_quadra_coder_param);
```

Alloc an instance of ni_hw_device_info_quadra_t used for ni_check_hw_info()

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

* \brief Create a pointer to ni_hw_device_info_quadra_t instance .

*

* \param[in] device_type_num:number of device type to be allocated in this function

*

* \param[in] avaliable_card_num:number of avaliable card per device to be allocated in
this function

*

* \return NULL-error,pointer to an instance when success

*****************************************************************
*******/

LIB_API ni_hw_device_info_quadra_t *ni_hw_device_info_alloc_quadra(int
device_type_num,int

                                                    avaliable_card_num);
```

Release resource in a pointer of ni_hw_device_info_quadra_t

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

* \brief Release a pointer to ni_hw_device_info_quadra_t instance created by

* create_hw_device_info_coder_param

*

* \param[in] p_hw_device_info:pointer to a ni_hw_device_info_quadra_t instance
created by

*create_hw_device_info_coder_param

*

*****************************************************************************
*******/

LIB_API void ni_hw_device_info_free_quadra(ni_hw_device_info_quadra_t
*p_hw_device_info);
```

Check card information and select a card by parameter

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

* \brief check hw info, return the appropriate card number to use depends on the

* load&task_num&used resource

*

* \param[out] pointer_to_p_hw_device_info : pointer to user-supplied
ni_hw_device_info_quadra_t

* (allocated by ni_hw_device_info_alloc).

* May be a ponter to NULL ,in which case a ni_hw_device_info_quadra_coder_param_t
is allocated

* by this function and written to pointer_to_p_hw_device_info.

* record the device info, including available card num and which card to select,

* and each card's informaton, such as, the load, task num, device type

*

* \param[in] task_mode: affect the scheduling strategy,

* 1 - both the load_num and task_num should consider, usually applied to live scenes

* 0 - only consider the task_num, don not care the load_num

*

* \param[in] hw_info_threshold_param : an array of threshold including device type task
threshold

* and load threshold

* in hw_mode fill the arry with both encoder and decoder threshold or

* fill the arry with preferential device type threshold when don not in hw_mode

* load threshold in range[0:100] task num threshold in range [0:32]

*

* \param[in] preferential_device_type : which device type is preferential 0:decode
1:encode .
```

* This need to set to encoder/decoder even if in sw_mode to check whether coder_param is wrong.

*

* \param[in] coder_param : encoder and decoder information that helps to choose card .This

*coder_param can be created and set to default value by function *create_hw_device_info_coder_param().

* You may change the resolution fps bit_8_10 or other vaule which is different from default value.

*

* \param[in] hw_mode:Set 1 then this function will choose encoder and decoder in just one card .

* When no card meets the conditions ,NO card will be choosed.

* You can try to use set hw_mode 0 to use sw_mode to do encoder/decoder in different card when

* hw_mode reports an error

* In hw_mode set both encoder_param and decoder_param in coder_param.

* Set 0 then just consider sw_mode to choose which card to do encode/decode,

* In sw_mode set one param in coder_param the other one will be set to NULL.

*

* \param[in] consider_mem : set 1 this function will consider memory usage extra

* set 0 this function will not consider memory usage

*

* \return 0-error 1-success

**********/

```
LIB_API int ni_check_hw_info(
                ni_hw_device_info_quadra_t    **pointer_to_p_hw_device_info,
                int task_mode,
                ni_hw_device_info_quadra_threshold_param_t
                *hw_info_threshold_param,
                ni_device_type_t preferential_device_type,
                ni_hw_device_info_quadra_coder_param_t * coder_param,
                int hw_mode,
                int consider_mem);
```

Print logs with additional information

```
/*!*****************************************************************************
******
*   \brief    print log message and additional information using ni_log_callback,
*   \param[in] p_context a pointer to ni_session_context_t if p_context != NULL
*                                  session_id/E2EID will be printed as extra information
*   \param[in] level    log level, if log_level == NI_LOG_ERROR timastamp will be
*                               printed as extra information
*   \param[in] format printf format specifier
*   \param[in] ...        additional arguments
*
*   \return
*****************************************************************************
******
LIB_API_LOG void ni_log2(const void *p_context, ni_log_level_t level, const char *fmt,
...)
```

Set whether to use a lock or not in ni_log2

```
/*!*****************************************************************************
********
*   \brief set whether to use a lock or not in ni_log2
*
*   \param[in] on whether to use a lock,
*                          1-->use a lock,
*                          0-->use extra buf(no lock)
*   \return
*****************************************************************************
******
void ni_log2_with_mutex(int on);
```

6.2 Quadra Pixel Formats

Quadra supports five pixel formats: 8-bit YUV420 planar, 8-bit YUV420 semi-planar, 10-bit YUV420 planar, 10-bit YUV420 semi-planar, RGBA, and Tiled4x4.

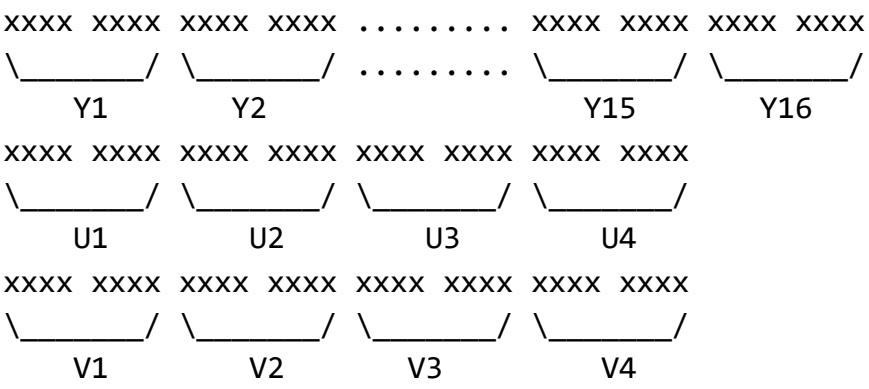
In YUV420, there is 2:1 chroma subsampling both vertically and horizontally. For every 2x2 grid of pixels, there are four luma (Y) components but only a single U and single V chroma are shared amongst the four pixels.

Below is a simple example of a 4 x 4 pixel image. Sections 7.2.1 through 7.2.4 will show the memory layout of these formats.

```
4 x 4 pixel image
-----------------  Pixel 1: Y1-U1-V1  Pixel 9:   Y9-U3-V3
| 1 | 2 | 3 | 4 |  Pixel 2: Y2-U1-V1  Pixel 10: Y10-U3-V3
|---+---+---+---|  Pixel 3: Y3-U2-V2  Pixel 11: Y11-U4-V4
| 5 | 6 | 7 | 8 |  Pixel 4: Y4-U2-V2  Pixel 12: Y12-U4-V4
|---+---+---+---|  Pixel 5: Y5-U1-V1  Pixel 13: Y13-U3-V3
| 9 | 10| 11| 12|  Pixel 6: Y6-U1-V1  Pixel 14: Y14-U3-V3
|---+---+---+---|  Pixel 7: Y7-U2-V2  Pixel 15: Y15-U4-V4
| 13| 14| 15| 16|  Pixel 8: Y8-U2-V2  Pixel 16: Y16-U4-V4
-----------------
```

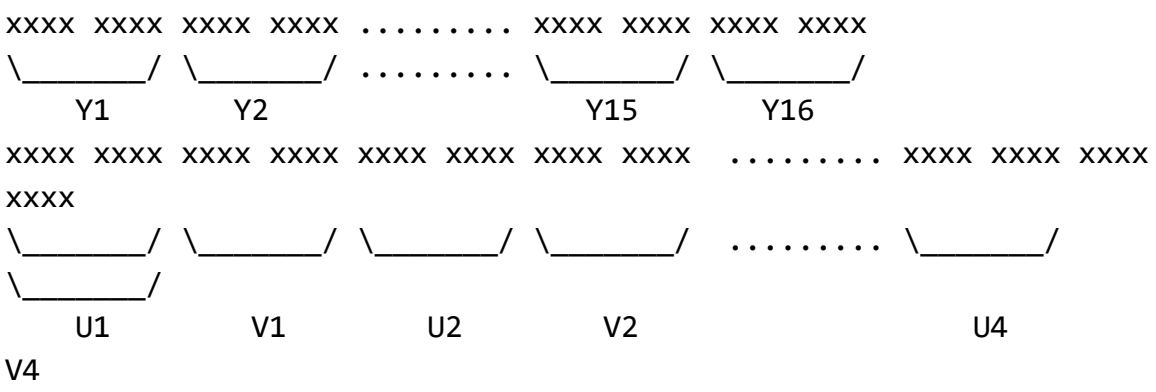
6.2.1 8-bit YUV420 planar (I420)

This is a common YUV420 format having eight bits per component. The components are divided into three planes. The Y components are followed by the U components which are subsequently followed by the V components.



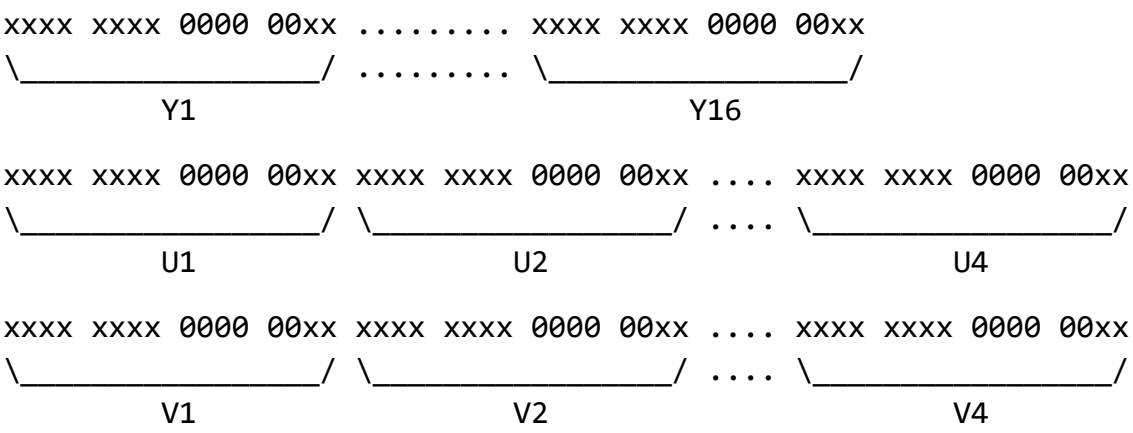
6.2.2 8-bit YUV420 semi-planar (NV12)

In the NV12 semi-planar format, the components are divided into two planes. The Y components are followed by a UV interleave with the U component first, V component second.



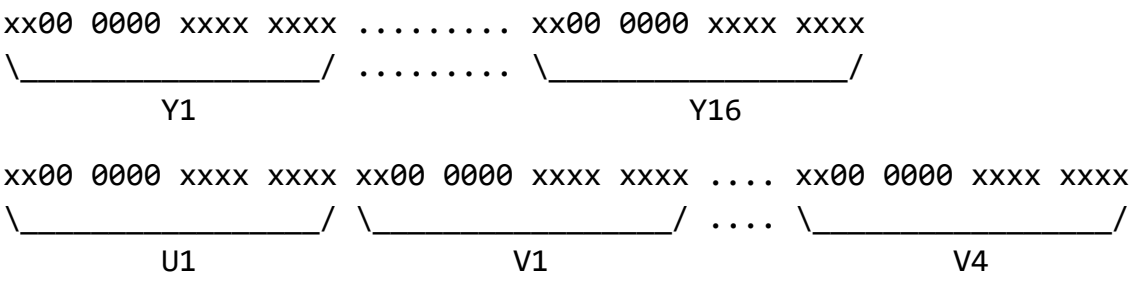
6.2.3 10-bit YUV420 planar

In the 10-bit YUV420 planar format, each component occupies sixteen bits of which ten of the sixteen bits are used. The least significant ten bits are used while the upper six bits are set to zero. The format is little-endian. The order of the planes is Y followed by U followed by V.



6.2.4 10-bit YUV420 semi-planar

In the 10-bit YUV420 semi-planar format, each component occupies sixteen bits of which ten of the sixteen bits are used. The most significant ten bits are used while the lower six bits are set to zero. The format is little-endian. The UV interleave is U first, V second.



6.2.5 32-bit RGBA

This is a packed 32-bit RGBA format where each pixel is represented by a four-byte value.

```
xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
\_____/ \_____/ \_____/ \_____/
   R       G       B       A
```

6.2.6 8 and 10-bit Tiled4x4

The tiled4x4 format is completely internal to Quadra in the sense that it cannot be hwdownloaded out of the device nor uploaded to into it. It is a compressed format and purely meant for increased memory efficiency during transcoding of large resolution inputs, ideally 10-bit.

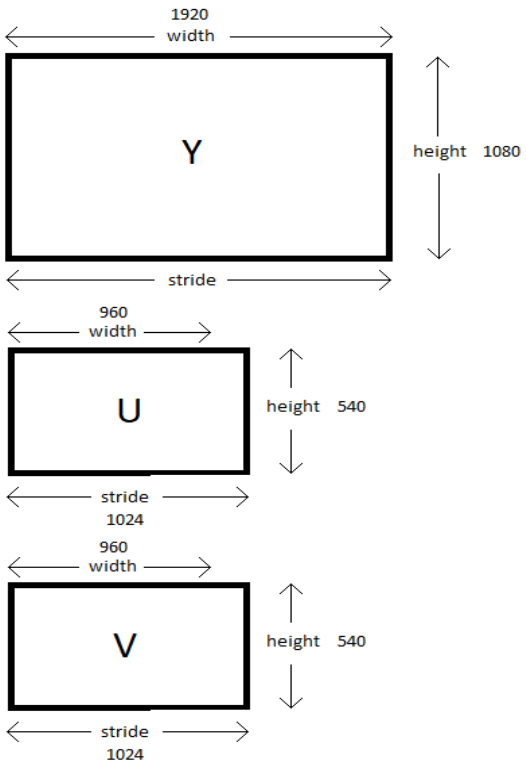
The layout of the pixels can be simplified as lossless compressed windows of the luma and chroma plane with a small table to decode the compressed data.

6.2.7 YUV420 stride and memory alignment requirements in Quadra

Quadra has the following requirements regarding the frame layout in memory.

- The frame pixel width and pixel height must be even
- The frame pixel width must be a minimum of 144 pixels
- The frame pixel height must be a minimum of 128 pixels
- The horizontal stride must be a multiple of 128 bytes for **both** the luma and chroma planes
- The starting memory address for each luma and chroma plane must be 128-byte aligned

Note that the requirement that the stride must be a multiple of 128 bytes for both the luma and chroma planes. For example, a 1920x1080 8-bit YUV420 planar frame will have both a width of 920 pixels and a stride of 1920 bytes for the luma plane. But the chroma



planes U and V will have a width of 960 pixels and a stride of 1024 bytes.

6.2.8 Device memory allocation and update

Quadra's device memory usage can be detected with ni_rsrc_mon. SHARE_MEM shows the percentage of memory in use.

For decoder instance, the number of buffers allocated is determined by input stream's header info(SPS, PPS, VPS, etc.) and decoding parameters. If a decoder instance is in hw mode(i.e, out=hw specified), 3 extra hw buffers are allocated. If a decoder instance has multiple outputs by enabling multiple ppu outputs, for example, 3 outputs, the total buffer count would be 3 * (buffer count calculated from header info). The number of buffers allocated by decoder instance can't be adjusted by host once open.

For encoder instance, the number of buffers allocated by it is determined by encoding parameters, for example, gop structure, lookahead depth, etc. The number of buffers allocated by encoder instance can't be adjusted by host once open.

For uploader instance, the number of buffers allocated by it is passed from libxcoder API ni_device_session_init_framepool. Before Quadra Release 4.7.0, it's not allowed to adjust uploader frame buffer pool once inited. **From Quadra Release 4.7.0(>=4.7.0)**, libxcoder could adjust uploader frame buffer pool by calling ni_device_session_update_framepool. Here is a use case:

- Host application calls ni_device_session_init_framepool to allocate frame pool with size 3. Currently, frame pool size = 3 frames.
- Host application calls ni_device_session_update_framepool to expand frame pool with size 5. At this time, frame pool size = 8 frames.
- Host application calls ni_device_session_update_framepool to free frame pool with size **0.** At this time, frame pool size = 0 frames.

NOTE: the frame pool size could be changed from Quadra Release 4.7.0, the frame pool type could **not** be changed, i.e, if the original frame pool type is

p2p(NI_POOL_TYPE_P2P, this type of frames data could be transferred via p2p, see section-6.7.4), expanding uploader frame pool with normal type(NI_POOL_TYPE_NORMAL) is not allowed, and vice versa.

For scaler instance, the number of buffers allocated by it is passed from libxcoder API (ni_device_alloc_frame). Default scaler frame pool size is NI_MAX_FILTER_POOL_SIZE. Before Quadra Release 4.7.0, adjusting scaler frame buffer pool size is not allowed, so allocating scaler frame buffer pool more than once will get error. **From Quadra Release 4.7.0(>=4.7.0)**, libxcoder could adjust scaler frame buffer pool by calling ni_device_alloc_frame. If frame pool size passed by the API is 0, allocated but not acquired frame buffers of the scaler instance will be freed. If frame pool size passed by the API is greater than 0, the frame buffer pool will expand specified count more buffers. Here is a use case:

- host application calls ni_device_alloc_frame() to allocate frame pool with size 4. Currently, frame pool size = 4 frames.
- host application calls ni_device_alloc_frame() again to allocate frame pool with size 3. At this time, frame pool size = 7 frames.
- host application calls ni_device_alloc_frame() again to free frame pool with size **0.** At this time, frame pool size = 0 frames.

NOTE: the frame pool size could be changed from Quadra Release 4.7.0, the frame pool type could **not** be changed, i.e, if the original frame pool type is p2p(NI_POOL_TYPE_P2P, this type of frames data could be transferred via p2p, see section-6.7.4), expanding scaler frame pool with normal type(NI_POOL_TYPE_NORMAL) is not allowed, and vice versa.