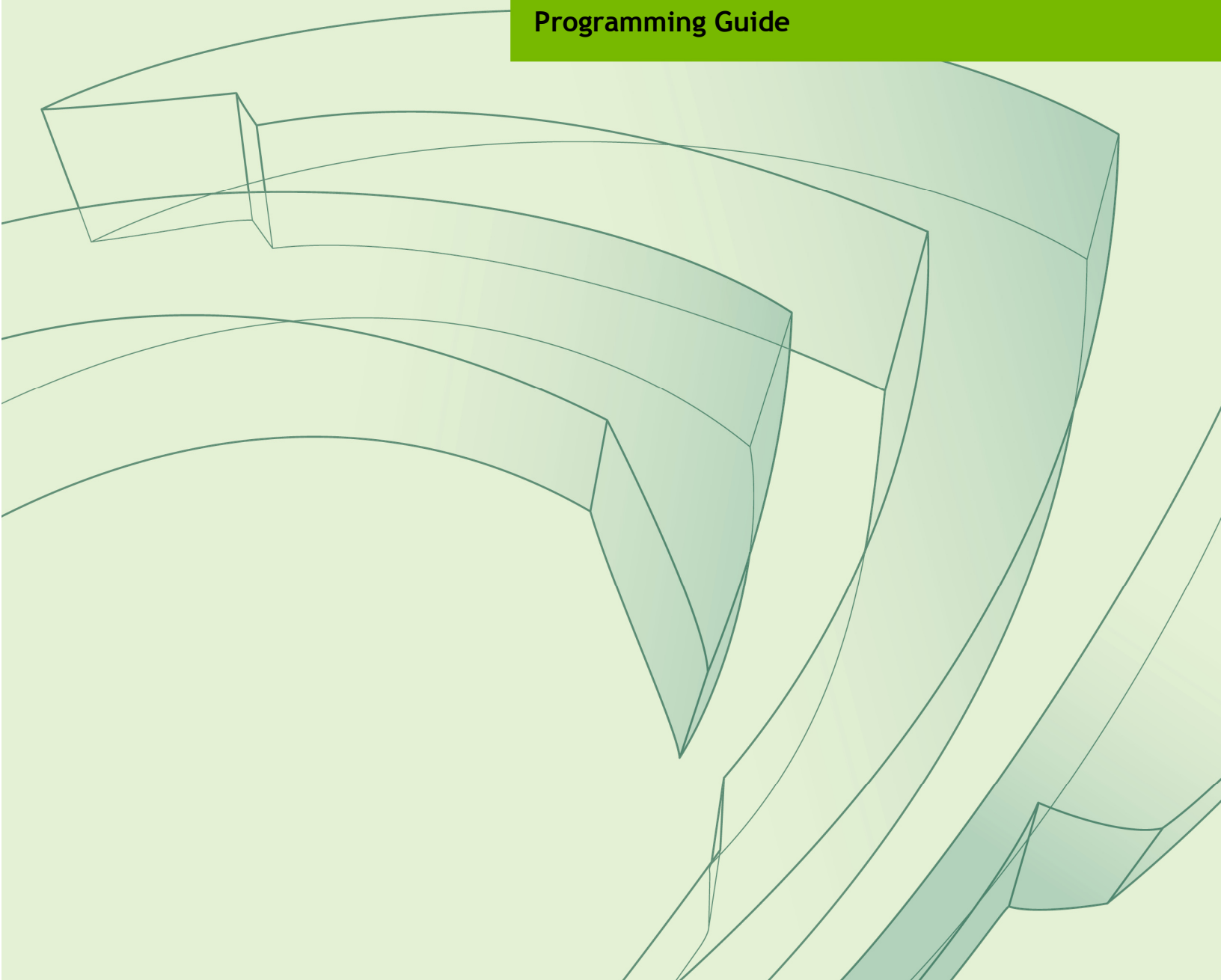




NVENC - NVIDIA VIDEO ENCODER INTERFACE

NVENC_VideoEncoder_API_PG-06155-001_v03 | August 2013

Programming Guide



REVISION HISTORY

Revision	Date	Author	Description
1.0	2011/12/29	SD/CC	Initial release.
1.1	2012/05/04	SD	Update for Version 1.1
2.0	2012/10/12	SD	Update for Version 2.0
3.0	2013/07/25	AG	Update for Version 3.0

TABLE OF CONTENTS

NVIDIA Video ENCODER Interface	1
1. Initializing NVIDIA Video Encoder Interface	1
2. Setting up The Hardware for encoding.....	2
2.1 Opening an Encode Session	2
2.1.1 Using the License client Key GUID:	2
2.1.2 Initializing the Encode Device	2
2.2 Selecting an Encoder GUID	3
2.3 Querying Capability Values	3
2.4 Getting Preset Encoder Configurations.....	3
2.4.1 Enumerating Preset GUIDs.....	4
2.4.2 Fetching Preset Encoder Configuration.....	4
2.5 Selecting an Encoder Profile.....	4
2.6 Getting Supported Input Format list.....	5
2.7 Initializing the Hardware Encoder Session.....	5
2.7.1 Configuring Encode Session Attributes	5
2.7.2 Finalizing the Codec Configuration for Encoding	6
2.7.3 Setting Encode Session Attributes	8
2.7 Creating Resources Required to Hold Input/Output Data	8
2.8 Retrieving Sequence Parameters	9
3. Encoding The Video Stream	10
3.1 Preparing Input Buffer for Encoding:.....	10
3.1.1 Input Buffers Allocated through the NVIDIA Video Encoder Interface:	10
3.1.2 Input Buffers Allocated Externally.....	10
3.2 Configuring Per-Frame Encode Parameters	11
3.2.1 Forcing the Current Frame to be Used as Intra-Predicted Reference Frame	11
3.2.2 Forcing the Current Frame To Be Used as a Reference Frame [H.264 only]	11
3.2.3 Forcing current frame to be used as an IDR frame [H.264 only]	11
3.2.4 Requesting Generation of Sequence parameters	11
3.2.6 Specifying QP Hints for the Current Input Picture	11
3.3 Submitting Input for Encoding	12
3.4 Consuming Encoded Output.....	12
4. End of Encoding	13
4.1 Notifying the End of Input Stream.....	13
4.2 Releasing the Created Resources	13
4.3 Closing the Encode Session.....	13
5. Modes of Operation.....	14
5.1 Asynchronous Mode	14
5.2 Synchronous Mode	17
6. Threading Model	18

7. Using Additional Features	19
7.1 Dynamic Output Resolution Change.....	19
7.2 Dynamic Reconfiguration of Encoder Rate Control Parameters.....	20
7.3 Low Latency Encoding	20
7.3.1 Single Frame VBV	20
7.3.2. Two-Pass Rate Control.....	20
7.3.3. Infinite GOP Length	21
7.3.4. Invalidate Reference Frame.....	21
The time-stamps passed must be same as NV_ENC_PIC_PARAMS::inputTimeStamp value sent as part of the NvEncEncodePicture API while encoding those frames.	22
7.4.5. Low latency Presets	22
7.4 Reconfigure API.....	22

NVIDIA VIDEO ENCODER INTERFACE

1. INITIALIZING NVIDIA VIDEO ENCODER INTERFACE

The client should start by linking the nvEncodeAPI dll. The client can choose to link at compile time or link at run-time using LoadLibrary.

In either case, the client's first interaction with the NVIDIA Video Encoder Interface is to call `NvEncodeAPICreateInstance`. This will give the client API function pointers to proceed with configuring and using the NVIDIA Video Encoder Interface.

2. SETTING UP THE HARDWARE FOR ENCODING

2.1 Opening an Encode Session

After loading the NVENC Interface, the client should first call `NvEncOpenEncodeSession` to open an encoding session. The NVENC Interface will provide a encode session handle to the client, which must be used for all further API calls in the current session.

2.1.1 Using the License client Key GUID:

The client should pass a pointer to the key GUID that has been delivered with this SDK or has been purchased as part of a license separately, as

```
NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::clientKeyPtr
```

2.1.2 Initializing the Encode Device

The NVIDIA Video Encoder Interface supports the use of the following types of Devices:

1) DirectX 9:

The client should create a DirectX 9 device with behavior flags including:

`D3DCREATE_FPU_PRESERVE`

`D3DCREATE_MULTITHREADED`

`D3DCREATE_HARDWARE_VERTEXPROCESSING`

The client should pass a pointer to the IUnknown interface of the created device [typecast to `void *`] as `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device`, and set `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` to `NV_ENC_DEVICE_TYPE_DIRECTX`. Use of DirectX devices is supported only on Windows 7 and later OS.

2) DirectX 10:

The client should pass a pointer to the IUnknown interface of the created device [typecast to `void *`] as `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device`, and set `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` to `NV_ENC_DEVICE_TYPE_DIRECTX`. Use of DirectX devices is supported only on Windows 7 and later OS.

3) DirectX 11:

The client should pass a pointer to the IUnknown interface of the created device [typecast to `void *`] as `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device`, and set `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` to `NV_ENC_DEVICE_TYPE_DIRECTX`. Use of DirectX devices is supported only on Windows 7 and later OS.

4) CUDA:

The client should create a floating CUDA context, and pass the CUDA device pointer [typecast to void *] as `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device`, and set `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` to `NV_ENC_DEVICE_TYPE_CUDA`. Use of CUDA device for Encoding is supported on Windows XP and Linux, in addition to Windows 7 and later, from NVIDIA Video Encoder Interface version 2.0 onwards.

2.2 Selecting an Encoder GUID

The negotiation process starts with selecting an Encoding GUID that represents the desired Codec for encoding the video sequence.

- i) The client should call `NvEncGetEncodeGUIDCount` to get the number of supported Encoder GUIDs from the NVIDIA Video Encoder Interface.
- ii) The client should use this count to allocate a buffer of sufficient size to hold the supported Encoder GUIDS.
- iii) The client should then call `NvEncGetEncodeGUIDs` to populate this list.

The client should select a GUID that matches its requirement from this list and use that as the `encodeGUID` for the remainder of the encoding session.

2.3 Querying Capability Values

The client may need to explicitly query the capability of the encoder to support certain features or certain encoding configuration parameters. For this, the client should do the following:

- i) The client is required to specify the capability attribute it wants to query through the `NV_ENC_CAPS_PARAM::capsToQuery` parameter. This should be a member of the `NV_ENC_CAPS` enum.
- ii) The client should call `NvEncGetEncoderCaps` to determine support for the required attribute.

Refer to the API reference `NV_ENC_CAPS` enum definition for interpretation of individual capability attributes.

2.4 Getting Preset Encoder Configurations

The client may not want to fine tune each parameter and set up an encoder configuration from scratch. In that case, the client should select a preset encoder configuration.

2.4.1 Enumerating Preset GUIDs

The client should enumerate supported Preset GUIDs for the selected encodeGUID as follows:

- i) The client should call `NvEncGetEncodePresetCount` to get the number of supported Encoder GUIDs from the NVIDIA Video Encoder Interface.
- ii) The client should use this count to allocate a buffer of sufficient size to hold the supported Preset GUIDS.
- iii) The client should then call `NvEncGetEncodePresetGUIDs` to populate this list.

2.4.2 Fetching Preset Encoder Configuration

The client can either use just the preset GUID for configuring the encode session or it can fine-tune the encoder configuration corresponding to a preset GUID.

The client should follow these steps to fetch a Preset Encode configuration:

- i) The client should enumerate the supported presets as described above, in the section “2.4.1 Enumerating Preset GUIDs”
- ii) The client should select a Preset GUID for which the encode configuration is to be fetched.
- iii) The client should call `NvEncGetPresetConfig` with the selected EncodeGUID and PresetGUID as inputs
- iv) The required preset encoder configuration can be retrieved through `NV_ENC_PRESET_CONFIG::presetCfg`.

2.5 Selecting an Encoder Profile

The client might need to explicitly specify the encoder profile to be used in some cases.

Eg: encoding video for playback on iPod, encoding video for BD authoring, etc.

The client should do the following to retrieve a list of supported encoder profiles:

- i) The client should call `NvEncGetEncodeProfileGUIDCount` to get the number of supported Encoder GUIDs from the NVIDIA Video Encoder Interface.
- ii) The client should use this count to allocate a buffer of sufficient size to hold the supported Encode Profile GUIDS.
- iii) The client should then call `NvEncGetEncodeProfileGUIDs` to populate this list.

The client should select the profile GUID that best matches its requirement..

2.6 Getting Supported Input Format list

The client should follow these steps to retrieve the list of supported input formats:

- i) The client should call `NvEncGetInputFormatCount`.
- ii) The Client should use the count retrieved from `NvEncGetInputFormatCount` to allocate a buffer to hold the list of supported input buffer formats [list elements of type `NV_ENC_BUFFER_FORMAT`].
- iii) The client should then populate this list by calling `NvEncGetInputFormats`.

The client should select a format enumerated in this list for creating input buffers.

2.7 Initializing the Hardware Encoder Session

The client needs to call `NvEncInitializeEncoder` with a valid encoder configuration specified through `NV_ENC_INITIALIZE_PARAMS`, and a pointer to a DirectX9 device to initialize the Encoder session.

2.7.1 Configuring Encode Session Attributes

Encode session Configuration is divided into three parts:

2.7.1.1 Session Parameters

Common parameters like output dimensions, input format, display aspect ratio, frame rate, average bitrate, etc. are available in `NV_ENC_INITIALIZE_PARAMS` structure. The client should use an instance of this structure as input to `NvEncInitializeEncoder`.

The Client must populate the following members of the `NV_ENC_INITIALIZE_PARAMS` struct for the Encode session to be successfully initialized:

- i) `NV_ENC_INITIALIZE_PARAMS::encodeGUID`: The client must select a suitable codec GUID as described in section “2.2 Selecting an Encoder GUID”
- ii) `NV_ENC_INITIALIZE_PARAMS::encodeWidth`: The client must specify the desired width of the encoded video.
- iii) `NV_ENC_INITIALIZE_PARAMS::encodeHeight`: The client must specify the desired height of the encoded video.

2.7.1.2 Advanced Codec-level parameters

Parameters dealing with the encoded bitstream such as GOP length, encoder profile, initial QP hints, Rate Control mode, etc are exposed through the `NV_ENC_CONFIG` structure. The client can pass codec level parameters through `NV_ENC_INITIALIZE_PARAMS::codecConfig`.

2.7.1.3 Advanced Codec-specific parameters

Advanced codec-specific parameters are available in `NV_ENC_CONFIG_XXXX` structures. This is for extensibility of the API to accommodate more codecs in the future.

Eg: H.264-specific parameters are available in `NV_ENC_CONFIG_H264`.

The client can pass codec-specific parameters through the `NV_ENC_CONFIG::encodeCodecConfig` union.

2.7.2 Finalizing the Codec Configuration for Encoding

2.7.2.1 High-level Control Using Presets

This is the simplest method of configuring the NVIDIA Video Encoder Interface, and involves minimal setup steps to be performed by the client. This is intended for use cases where the client does not need to fine-tune any codec level parameters.

In this case, the client should follow these steps:

- i) The client should specify the session parameters as described in section “2.7.1.1 Session Parameters”
- ii) Optionally, the client can enumerate and select Preset GUID that best suites the current use case, as described in section “2.4.1 Enumerating Preset GUIDs” The client should then pass the selected Preset GUID using `NV_ENC_INITIALIZE_PARAMS::presetGUID`.

This helps the NVIDIA Video Encoder interface to correctly configure the encoder session based on the `encodeGUID` and `presetGUID` provided. Hence, this step is recommended.
- iii) The client should set the advanced codec-level parameter pointer `NV_ENC_INITIALIZE_PARAMS::codecParams` to `NULL`

2.7.2.2 Coarse Control by overriding Preset Parameters

The client can choose to edit some encoding parameters, but not want to populate the complete encode configuration from scratch. In this case, the client is recommended to follow these steps:

- i) The client should specify the session parameters as described in section “[2.7.1.1 Session Parameters](#)”
- ii) The client should enumerate and select a Preset GUID that best suites the current use case, as described in section “[2.4.1 Enumerating Preset GUIDs](#)” The client should retrieve a Preset encode configuration as described in section “[2.4.2 Fetching Preset Encoder Configuration](#)”
- iii) The client may need to explicitly query the capability of the encoder to support certain features or certain encoding configuration parameters. For this, the client should do the following:

- iv) The client is required to specify the capability attribute it wants to query through the `NV_ENC_CAPS_PARAM::capsToQuery` parameter. This should be a member of the `NV_ENC_CAPS` enum.
- v) The client should call `NvEncGetEncoderCaps` to determine support for the required attribute. Refer to `NV_ENC_CAPS` enum definition in the API reference for interpretation of individual capability attributes.
- vi) The client should then select a desired Preset GUID and fetch the corresponding Preset Encode Configuration as described in: “2.4 Getting Preset Encoder Configurations”.
- vii) The client can override any parameters from the preset `NV_ENC_CONFIG` according to its requirements. The client should pass the fine-tuned `NV_ENC_CONFIG` structure using `NV_ENC_INITIALIZE_PARAMS::codecConfig` pointer.
- viii) Additionally, the client should also pass the selected preset GUID through `NV_ENC_INITIALIZE_PARAMS::presetGUID`. This is to allow the NVIDIA Video Encoder interface to program internal parameters associated with the encoding session to ensure that the encoded output conforms to the client’s request. Passing the preset GUID will not override the fine-tuned parameters.

2.7.2.3 Fine-Grain Control by building a custom preset

There might be cases where the client would like to build an encode configuration from scratch. In this case, the client is recommended to take the following measures:

- i) The client should specify the session parameters as described in section “[2.7.1.1 Session Parameters](#)”
- ii) The client should ensure that it sets `NV_ENC_INITIALIZE_PARAMS::codecConfig` pointer to a valid `NV_ENC_CONFIG` struct.
- iii) The client should ensure that the `NV_ENC_CONFIG` struct members have valid values based on the encoder’s capabilities as described in section “[2.3 Querying Capability Values](#)”.

Eg: attributes like rate control mode, no. of B-frames, frame/field encoding, monochrome encoding, etc. should be validated beforehand.
- iv) The client should ensure that it either sets `NV_ENC_CONFIG::profileGUID` to a valid value or sets it to `NV_ENC_CODEC_PROFILE_UNKNOWN_GUID` in order to allow the NVIDIA Video Encoder Interface to select the most appropriate encoder profile. Selecting a valid encoder profile is described in section “2.5 Selecting an Encoder Profile”. It is recommended that the client set `profileGUID` to `NV_ENC_CODEC_PROFILE_UNKNOWN_GUID` if the client does not require the output to be constrained to a particular profile. In such a case, the NVIDIA Video Encoder Interface will select the appropriate profile based on the custom preset provided.
- v) Additionally, the client can also pass the selected preset GUID through `NV_ENC_INITIALIZE_PARAMS::presetGUID`. This is to allow the NVIDIA Video Encoder interface to program internal parameters associated with the encoder session to ensure that the encoded output conforms to the client’s request. Passing the preset GUID will not override the fine-tuned parameters.

Note: The client may need to explicitly specify the encoder profile to be used in some cases, such as encoding video for playback on iPod, encoding video for BD authoring, etc.

In order to conform to such constraints as may be imposed on the encoded bitstream by such use cases, the NVIDIA Video Encoder Interface will always honor the encoder profile set by the client through `NV_ENC_CONFIG::profileGUID`. It may override some of the parameters provided in order to generate a profile-compliant bitstream.

2.7.3 Setting Encode Session Attributes

Once all Encoder settings have been finalized, the client should populate a `NV_ENC_CONFIG` struct, and use it as an input to `NvEncInitializeEncoder` in order to freeze the Encode settings for the current encodes session. Some settings such as Rate Control mode, Average Bitrate, and QP hints can be changed on-the-fly.

The client is required to explicitly specify the following while initializing the Encode Session:

2.7.3.1 Mode of Operation

The client should set `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` to 1 if it wants to operate in Asynchronous mode. Set it to 0 for operating in Synchronous mode. Asynchronous mode encoding is only supported on Windows 7 and later Windows OS.

2.7.3.2 Picture-type Decision

The client should set `NV_ENC_INITIALIZE_PARAMS::enablePTD` to 1 for allowing the HW encoder to take Picture-type decision. Set it to 0 if client wants to decide picture type for encoded output.

Note: `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync == 0` and `NV_ENC_INITIALIZE_PARAMS::enablePTD == 1` are not a compatible combination.

If the client needs to use synchronous mode of operation, the client should also take care of picture-type decision; i.e. if `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync == 0` then `NV_ENC_INITIALIZE_PARAMS::enablePTD` should also be 0.

2.7 Creating Resources Required to Hold Input/Output Data

Once the Encode session is initialized, the client should allocate buffers to hold the input/output data. The input buffer width and height should be 32-aligned.

The client should allocate buffers to hold the output encoded bitstream using the `NvCreateBitstreamBuffer` API. It is the client's responsibility to destroy these buffers before closing the Encode Session.

The client may choose to allocate input buffers through the NVIDIA Video Encoder Interface, by calling `NvEncCreateInputBuffer` API. In this case, the client is responsible to destroy the allocated input buffers before closing the Encode Session. It is also the client's responsibility to fill the input buffer with valid input data according to the chosen input buffer format.

Alternatively, in scenarios where the client cannot or does not want to allocate input buffers through the NVIDIA Video Encoder Interface, it can use any externally allocated DirectX resource as an input buffer. However, the client has to perform some simple processing to map these resources to resource handles that are recognized by the NVIDIA Video Encoder Interface before use. The translation procedure is explained in section [“3.1.2 Input Buffers Allocated Externally”](#)

If the client has used a CUDA device to initialize the encoder session, and wishes to use input buffers NOT allocated through the NVIDIA Video Encoder Interface, the client is required to use buffers allocated using the `cuMemAlloc` family of APIs. The NVIDIA Video Encoder Interface version 2.0 only supports `CUdevicePtr` as input. Support for `CUarray` inputs will be added future version. Even in this case, the client is required to perform the

Note: The client should allocate at least $[4 + \text{No. of B-Frames}]$ Input/Output buffers.

2.8 Retrieving Sequence Parameters

After configuring the Encode Session, the client can retrieve Sequence parameter information at any time by calling `NvEncGetSequenceParams`. The client must allocate a buffer of size `NV_MAX_SEQ_HDR_LEN` to hold the Sequence parameters. It is the client's responsibility to manage this memory.

By default, SPS PPS data will be attached to every IDR frame. However, the client can request the NVIDIA Video Encoder Interface to generate SPS PPS data on the fly as well. For this, the client must set `NV_ENC_PIC_PARAMS::encodeFlags` to `NV_ENC_FLAGS_OUTPUT_SPSPPS`. The output frame generated for the current input will then contain SPS PPS data attached to it.

The client can call `NvEncGetSequenceParams` at any time during the encoding session, after it has called `NvEncInitializeEncoder`.

3. ENCODING THE VIDEO STREAM

Once the Encode Session is configured and input/output buffers are allocated, the client can start streaming the input data for encoding. The client is required to pass a handle to a valid input buffer and a valid bitstream buffer to the NVIDIA Video Encoder Interface for encoding an input picture.

3.1 Preparing Input Buffer for Encoding:

3.1.1 Input Buffers Allocated through the NVIDIA Video Encoder Interface:

If the client has allocated input buffers through `NvEncCreateInputBuffer`, the client needs to fill valid input data before using the buffer as input for encoding. For this, the client should call `NvEncLockInputBuffer` to get a CPU pointer to the input buffer. Once the client has filled input data, it should call `NvUnlockInputBuffer`. The client should use the input buffer for encode only after unlocking it. The client must also take care to unlock any locked input buffer before destroying it.

3.1.2 Input Buffers Allocated Externally

If the client is using externally allocated buffers as input, the client is required to call `NVEncRegisterResource` before use with the NVIDIA Video Encoder Interface. The client should also explicitly call `NvEncUnregisterResource` with this handle before destroying the resource. The client should call `NvEncMapInputResource` to retrieve a handle to the resource that is understandable by the NVIDIA Video Encoder Interface. Note that the client is required to pass the registered handle as

`NV_ENC_MAP_INPUT_RESOURCE::inputRegisteredPtr`. The mapped handle will be made available in `NV_ENC_MAP_INPUT_RESOURCE::outputResourcePtr`. The client should use this mapped handle as the input buffer handle in `NV_ENC_PIC_PARAM`. The client should call `NvEncUnmapInputResource` after it has finished using the resource as an input to the NVIDIA Video Encoder Interface. The resource should not be used for any other purpose outside the NVIDIA Video Encoder Interface while it is in 'mapped' state. Such usage is not supported and may lead to undefined behavior.

3.2 Configuring Per-Frame Encode Parameters

The client should populate a `NV_ENC_PIC_PARAMS` struct with the parameters it requires to be applied to the current input picture. The client can do the following on a per-frame basis:

3.2.1 Forcing the Current Frame to be used as Intra-Predicted Reference Frame

The client should set `NV_ENC_PIC_PARAMS::encodeFlags` to `NV_ENC_FLAGS_FORCEINTRA`.

3.2.2 Forcing the Current Frame to be used as a Reference Frame [H.264 only]

The client should set `NV_ENC_PIC_PARAMS_H264::refPicFlag` to 1

3.2.3 Forcing current frame to be used as an IDR frame [H.264 only]

The client should set `NV_ENC_PIC_PARAMS_H264::forceIDR` to 1.

3.2.4 Requesting Generation of Sequence parameters

The client should set `NV_ENC_PIC_PARAMS::encodeFlags` to `NV_ENC_FLAGS_OUTPUT_SPSPPS`.

3.2.6 Specifying QP Hints for the Current Input Picture

The client should set the flag `NV_ENC_PIC_PARAMS::userForcedConstQP` to 1 and specify the required RateControl QP value in `NV_ENC_PIC_PARAMS::userFrameQP`. This request will be honored only if the Encode Session is already running with rate control mode `NV_ENC_PARAMS_RC_CONSTQP`, or the rate control mode is being set to `NV_ENC_PARAMS_RC_CONSTQP` in the current operation.

3.3 Submitting Input for Encoding

The client should call `NvEncEncodePicture` to perform encoding.

The input picture data will be taken from the specified input buffer, and the encoded bitstream will be available in the specified bitstream buffer once the encoding process completes.

Common parameters such as timestamp, duration, input buffer pointer, etc are available in `NV_ENC_PIC_PARAMS` while Codec-specific parameters are available in `NV_ENC_PIC_PARAMS_XXXX` structures.

For example, H.264-specific parameters are available in `NV_ENC_PIC_PARAMS_H264`.

The client should specify the codec specific structure to `NV_ENC_PIC_PARAMS` using the `NV_ENC_PIC_PARAMS::codecPicParams` member.

3.4 Consuming Encoded Output

Upon completion of the encoding process for an input picture, the client is required to call `NvEncLockBitstream` to get a CPU pointer to the encoded bitstream. The client can choose to make a local copy of the encoded data or pass on the CPU pointer to a media file writer.

The CPU pointer will remain valid until the client calls `NvUnlockBitstreamBuffer`. The client should call `NvUnlcokBitstreamBuffer` after it completes processing the output data.

The client must ensure that all bitstream buffers are unlocked before destroying them (while closing an encode session).

4. END OF ENCODING

4.1 Notifying the End of Input Stream

To notify the end of input stream, the client must call `NvEncEncodePicture`, with the flag `NV_ENC_PIC_PARAMS::encodeFlags` set to `NV_ENC_FLAGS_EOS`. This must be done before closing the Encode Session.

When notifying End of Stream, the client should set all other members of `NV_ENC_PIC_PARAMS` to 0. No input buffer is required in case of EOS notification.

EOS notification effectively flushes the encoder. This can be called multiple times in a single encode session.

4.2 Releasing the Created Resources

Once encoding completes, the client should destroy all allocated resources.

The client should call `NvEncDestroyInputBuffer` if it had allocated input buffers through the NVIDIA Video Encoder Interface. The client must be sure not to destroy a buffer while it is locked.

The client should call `NvEncDestroyBitStreamBuffer` to destroy each bitstream buffer it had allocated. The client must be sure not to destroy a bitstream buffer while it is locked.

4.3 Closing the Encode Session

The client should call `NvEncDestroyEncodeSession` to close the encoding session. The client should ensure that all resources tied to the encode session being closed have been destroyed before calling `NvEncDestroyEncodeSession`. These include Input buffers, bitstream buffers, SPSPS buffer, etc.

It must also ensure that all registered events are unregistered, and all mapped input buffer handles are unmapped.

5. MODES OF OPERATION

The NVIDIA Video Encoder Interface supports the following two modes of operation:

5.1 Asynchronous Mode

This mode of operation is used for asynchronous output buffer processing and is event driven. In this mode, the client typically allocates an event object and associates the event with an allocated output buffer. This event object is passed to NVIDIA Video Encoder Interface as part of the `NvEncEncodePicture` API. The client can wait on the event on a separate thread and when the event is signaled, the client can call NVIDIA Video Encoder Interface to copy bitstream output produced by the HW encoder. Note that the NVIDIA Video Encoder Interface versions 1.0, 2.0 and 3.0 support asynchronous mode of operation on Windows only. In Linux, only synchronous mode is supported (described in Section “5.2 Synchronous Mode”)

The client should set the flag `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` to 1 to indicate that it wants to operate in asynchronous mode. After creating the Event objects [one corresponding to each bitstream buffer it creates], the client needs to register them with the NVIDIA Video Encoder Interface through the `NvEncRegisterAsyncEvent` API. The client is required to pass a bitstream buffer handle and an event handle as input to `NvEncEncodePicture`. The NVIDIA Video Encoder Interface will signal this event when the HW finishes encoding the current input data. The client can then call `NvEncLockBitstream` in non-blocking mode [`NV_ENC_LOCK_BITSTREAM::doNotWait` flag set to 1] to fetch the output data.

The client should call `NvEncUnregisterAsyncEvent` to unregister the Event handles before destroying the Event objects. This is the preferred mode of operation.

A step-by-step control flow would be:

- i) When working in asynchronous mode, the output sample must consist of an event + output buffer and clients must work in multi-threaded manner (D3D9 device should be created with `MULTITHREADED` flag).
- ii) The output buffers are allocated using `NvEncCreateBitstream` API. The NVIDIA Video Encoder Interface will return an opaque pointer to the output memory in `NV_ENC_CREATE_BITSTREAM_BUFFER::bitstreambuffer`. This opaque output pointer should be used in `NvEncEncodePicture` and `NvEncLockBitstream` / `NvEncUnlockBitstream` calls. For accessing the output memory using CPU, client must call `NvEncLockBitstream` API. The number of IO buffers should be at least 4 + number of B frames.
- iii) The events are windows event handles allocated using `win32 CreateEvent` API and registered with `NvEncodeAPI` SW using `NvEncRegisterAsyncEvent` before encoding. The registering of events is required only once per encoding session. Clients must unregister the events using `NvEncUnregisterAsyncEvent` before destroying the event

handles. The number of event handles must be same as number of output buffers as each output buffer is associated with an event.

- iv) Client must create a secondary thread in which it can wait on the completion event and copy the bitstream data from the output sample. Client will have 2 threads, one is the main application thread which submits encoding work to `NvEncodeAPI` while secondary thread waits on the completion events and copies the compressed bitstream data from the output buffer.
- v) Client must send both the event and output buffer in `NV_ENC_PICTURE_PARAMS::outputBitstream` and `NV_ENC_PICTURE_PARAMS::completionEvent` fields as part of `NvEncEncodePicture` API call.
- vi) Client should then wait on the event on the secondary thread in the same order in which they have called `NvEncEncodePicture` calls irrespective of input buffer re-ordering (encode order != display order). `NvEncodeAPI` takes care of the reordering in case of B frames and should be transparent to the encoder clients.
- vii) When the event gets signalled then client must send down the output buffer of the output sample on whose event it was waiting on in `NV_ENC_LOCK_BITSTREAM::outputBitstream` field as part of `NvEncLockBitstream`.
- viii) NVIDIA Video Encoder Interface will return a CPU pointer and bitstream size in bytes as part of the `NV_ENC_LOCK_BITSTREAM`.
- ix) After copying the bitstream data, client must call `NvEncUnlockBitstream` for the locked output bitstream buffer.

The example below shows how reordering happens in case of encoding with 1 B-frame

Let's say client allocates 4 input buffers (I1,I2..) and 4 output samples (4 buffers + 4 events O1, O2... + E1 , E2). The NVIDIA Video Encoder Interface will need to keep a copy of the input buffers for reordering and it allocates following internal buffers (NvI1, NvI2...). These internal buffers (NvI1, NvI2....) are managed by NVIDIA Video Encoder Interface. The client is not responsible for the allocating or freeing the memory.

a) Client main thread will queue the following encode frame calls. Note the picture type is unknown to the client, the decision is being taken by NVIDIA Video Encoder Interface. The client should pass parameter sets consisting of allocated input buffer, output buffer and output events in successive `NvEncEncodePicture` API calls. For example:

1st EncodePicture parameters - (I1, O1, E1)

2nd EncodePicture parameters - (I2, O2, E2)

3rd EncodePicture parameters - (I3, O3, E3)

b) NVIDIA Video Encoder Interface will receive the following encode Commands from the client. The left side shows input from client in the form (Input buffer, Output Buffer, Output Event). The right hand side shows a possible picture type decision of the NVIDIA Video Encoder Interface layer.

(I1, O1, E1) ---I Frame

(I2, O2, E2) ---B Frame

(I3, O3, E3) ---P Frame

c) NVIDIA Video Encoder Interface will do a copy (copy is done on GPU) from the input buffers to `NvEncEncodePicture` internal buffers for reordering. These copies are done as part of `NvEncEncodePicture` function call from the client and NVIDIA Video Encoder Interface is responsible for synchronization of these copies with the encoding operation.

I1 → NvI1

I2 → NvI2

I3 → NvI3

d) After returning from `NvEncEncodePicture` call , the client can queue the output bitstream operation(waiting on the event and copying the bitstream) to the secondary thread. This unblocks the main thread to submit more work to the NVIDIA Video Encoder Interface. The work queued to the secondary thread is in the following order

(I1, O1, E1)

(I2, O2, E2)

(I3, O3, E3)

Note they are in the same order in which client calls `NvEncEncodePicture` API in step a).

e) NVIDIA Video Encoder Interface will do the reordering such that Encoder HW will receive the following encode Commands

(NvI1, O1, E1) ---I Frame

(NvI3, O2, E2) ---P Frame

(NvI2, O3, E3) ---B frame

f) After the encoding operations are completed , the events will be signalled by NVIDIA Video Encoder Interface in the following order

(O1, E1) ---I Frame encoding completed and event signalled.

(O2, E2) ---P Frame encoding completed and event signalled.

(O3, E3) ---B Frame encoding completed and event signalled.

g) The client must lock the bitstream data using `NvEncLockBitstream` API in the order O1, O2, O3 and copy the encoded data to the final bitstream.

Note:

- The client will receive the events signal and output buffer in the same order in which they have been queued.
- The `LockBitstream` parameter struct has a picture type field which will notify the output picture type to the clients.
- Both, the input and output sample (output buffer and the output completion event) are free to be reused once the NVIDIA Video Encoder Interface has signalled the event and the client has copied the data from the output buffer.

5.2 Synchronous Mode

This mode of operation is used for synchronous output buffer processing. In this mode the client needs to make a blocking call to NVIDIA Video Encoder Interface to retrieve the output bitstream data produced by the encoder. The client should set the flag `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` to 0 to indicate that it wants to operate in synchronous mode. To operate in synchronous mode, the client should call `NvEncEncodePicture` without setting a completion Event handle. It should instead call `NvEncLockBitstream` with flag `NV_ENC_LOCK_BITSTREAM::doNotWait` set to 1, so that the lock call blocks until the HW Encoder finishes writing the output bitstream. The client can then operate on the generated bitstream data and call `NvEncUnlockBitstream` as it would normally.

It is important to note that if the client wants to use Synchronous mode for encoding with B-frames, the client must also set `NV_ENC_CONFIG::enablePTD` to 0, i.e. the client must also handle the picture-type decision, and send a valid picture input to the NVIDIA Encoder Interface.

6. THREADING MODEL

In order to get maximum performance for encoding, the encoder client should create a separate thread to wait on events or when making any blocking calls to the encoder interface.

The client should avoid making any blocking calls from the main encoder processing thread. The main encoder thread should be used only for encoder initialization and to submit work to the HW Encoder using `NvEncEncodePicture` API, which is non-blocking.

Output buffer processing, such as waiting on the completion event in asynchronous mode or calling the `NvEncLockBitstream/NvEncUnlockBitstream` blocking API in synchronous mode, should be done on the secondary thread. This ensures the main encoder thread is never blocked except when the encoder client runs out of resources.

7. USING ADDITIONAL FEATURES

7.1 Dynamic Output Resolution Change

The NVIDIA Video Encoder Interface versions 2.0 and above support changing encode output resolution on the fly, without re-negotiating a new encode session.

To make use of this feature, the client must do the following:

- 1) Check for dynamic resolution change support using capability query for `NV_ENC_CAPS_SUPPORT_DYN_RES_CHANGE`.
- 2) Set `NV_ENC_INITIALIZE_PARAMS::maxEncodeWidth` and `NV_ENC_INITIALIZE_PARAMS::maxEncodeHeight` to the maximum encoding width and maximum encoding height that the client foresees to be used in the current encoding session. The client must check for `NV_ENC_CAPS_WIDTH_MAX` and `NV_ENC_CAPS_HEIGHT_MAX` before setting the maximum and initial encoding dimensions.
- 3) Allocate Input buffers using the max dimensions specified above
- 4) During encoding, when there is a output resolution change required, the client must:
 - a. Set `NV_ENC_PIC_PARAMS::encodePicFlags` to include `NV_ENC_PIC_FLAG_DYN_RES_CHANGE`. If Picture Type Decision is being handled by the Client [`NV_ENC_INITIALIZE_PARAMS::enablePTD == 0`], the client should set the `NV_ENC_PIC_PARAMS::pictureType` as `NV_ENC_PIC_TYPE_IDR`.
 - b. Set `NV_ENC_PIC_PARAMS::newEncodeWidth` and `NV_ENC_PIC_PARAMS::newEncodeHeight` to the new desired encode width and height respectively.

Important constraints:

- 1) The new encoding dimensions must not exceed the maximum dimensions set at the time of encoder initialization; otherwise the attempt to change encoding dimensions will fail.
- 2) The NVIDIA Video Encoder Interface will not perform any scaling even when the Dynamic Output Resolution Change feature is used. The client must ensure that the input YUV surface has the same dimensions as the desired encoded output resolution. Failure to meet this constraint can lead to cropped or corrupt output.

Please note that this method of dynamically changing output resolution will be deprecated in future releases of NV Encode API. To change the resolution dynamically, clients are advised to use the API `NvEncReconfigureEncoder`, explained in Section 7.4 Reconfigure API.

7.2 Dynamic Reconfiguration of Encoder Rate Control Parameters

Client can change the rate control parameters such as bitrate, VBV size, etc. can be changed dynamically per encoded picture, using the variables in structure

`NV_ENC_PIC_PARAMS::rcParams` and setting the flag
`NV_ENC_PIC_FLAG_DYN_BITRATE_CHANGE` in `NV_ENC_PIC_PARAMS::encodePicFlags`.

Please note that this method of dynamically changing bitrate parameters will be deprecated in future releases of NV Encode API. To change bitrate parameters dynamically, clients are advised to use the API `NvEncReconfigureEncoder`, explained in Section 7.4 Reconfigure API.

7.3 Low Latency Encoding

The following section describes the NVIDIA recommended settings for H.264 encoder for low-latency applications such as cloud gaming, real-time streaming etc.

7.3.1 Single Frame VBV

NVIDIA recommends setting the VBV buffer size equal to single frame size. This is very helpful in low latency applications, where network bandwidth is a concern. Single frame VBV allows users to enable capped frame size encoding. In single frame VBV, VBV buffer size must be set to maximum frame size which is equal to channel bitrate divided by frame rate. With this setting, every frame can be sent to client immediately upon encoding and the decoder can also decode without any buffering.

For example, if you have a channel bitrate of B bits/sec and you are encoding at N fps, the following settings are recommended to enable single frame VBV buffer size.

```
uint32_t maxFrameSize = B/N;
NV_ENC_RC_PARAMS::vbvBufferSize= maxFrameSize;
NV_ENC_RC_PARAMS::vbvInitialDelay= maxFrameSize;
NV_ENC_RC_PARAMS::maxBitRate= NV_ENC_CONFIG::vbvBufferSize *N; // where
N is the encoding frame rate.
NV_ENC_RC_PARAMS::averageBitRate=
    NV_ENC_RC_PARAMS::vbvBufferSize *N; // where N is the encoding frame
rate.
NV_ENC_RC_PARAMS::rateControlMode= NV_ENC_PARAMS_RC_TWOPASS_CBR;
```

7.3.2. Two-Pass Rate Control

For low latency encoding, as well for generally better quality, it is recommended to use two-pass rate control (`NV_ENC_PARAMS_RC_TWOPASS_CBR`), which allows better-quality

encoding compared to single pass rate control (`NV_ENC_PARAMS_RC_CBR`) with a small drop in performance.

Applications using strict single frame VBV and single pass rate control will result in poor quality scene-cuts and lower quality for frames with high complexity. Using two-pass rate control helps to mitigate these issues. It should be set as follows:

```
NV_ENC_RC_PARAMS::rateControlMode= NV_ENC_PARAMS_RC_TWOPASS_CBR;
```

7.3.3. Infinite GOP Length

To maintain uniform quality at all times, both I and P-frames must be encoded with the same quality. However, encoding I frames with quality comparable to P frame requires significantly more bits. With single-frame VBV buffer size, this is not possible, and results in poor quality I-frames.

Therefore, for low-latency applications, NVIDIA recommends using infinite GOP length while encoding. This is achieved by setting `NV_ENC_CONFIG::gopLength` to `0xffffffff`. Infinite GOP length disables automatic insertion of I frames. The client is recommended to avoid sending I frames unless it is necessary for error recovery.

Additionally, clients can also use two pass rate control (`NV_ENC_PARAMS_RC_TWOPASS_CBR`), which will allow encoder to detect and perform additional processing to encode scene-cuts at better quality.

```
NV_ENC_CONFIG::gopLength = 0xffffffff;
```

The client can force individual frame to be encoded as intra-coded frame by setting `NV_ENC_PIC_PARAMS::encodePicFlags` to include `NV_ENC_PIC_FLAG_FORCEINTRA` while calling `NvEncEncodePicture` API.

7.3.4. Invalidate Reference Frame

This is the recommended error resiliency feature for low latency applications. When a client decoder detects a lost packet or corrupt frame, it can signal the server side encoder to invalidate that frame and all the frames which have been constructed using the corrupt frame. If no frames are left for motion estimation, then the current frame will be encoded as an I frame. The client must set a large DPB size (`NV_ENC_CONFIG_H264::maxNumRefFrames`) to allow encoder to store a large number of backup reference frames, in case encoder has to fallback to use older reference frames for motion estimation when recent reference frames have been invalidated. Note that using a large DPB size doesn't increase the actual number of frames used for motion estimation; so there is no drop in performance. The recommended DPB size is 16 frames. This achieved by setting the following parameter while calling `NvEncInitializeEncoder` :

```
NV_ENC_CONFIG_H264::maxNumRefFrames = 16;
```

Let's say server receives a request from the decoder to invalidate N reference frames with capture time-stamps of $a, b, c, \dots$. Then, the `NvEncInvalidateRefFrames` API must be called once with each time-stamp that needs to be invalidated.

The time-stamps passed must be same as `NV_ENC_PIC_PARAMS::inputTimeStamp` value sent as part of the `NvEncEncodePicture` API while encoding those frames.

7.4.5. Low latency Presets

The client application can use either of the following three encoding presets provided by NVENC interface which can be specified by `NV_ENC_CONFIG` parameter. This controls various encoding parameters for motion estimation (ME) and mode decision in the H.264 encoder.

NV_ENC_PRESET_DEFAULT_GUID: This is the default low latency preset.

NV_ENC_PRESET_CLOUD_GAMING_720p60_GUID: This preset is designed for cloud gaming when encoding a 1280×720 capture at 60 frames per second. This uses a larger ME search range than the one used in default preset and hence there is a small performance drop compared to default preset.

NV_ENC_PRESET_CLOUD_GAMING_720p30_GUID: This preset is designed for cloud gaming when encoding a 1280×720 capture at 30 frames per second. This uses larger ME search range than those in the above two presets and hence provides better ME result but has a larger performance penalty compared to earlier two presets.

7.4 Reconfigure API

`NvEncReconfigureEncoder` API allows changing of encoder initialization parameters in `NV_ENC_INITIALIZE_PARAMS` without closing existing encoder session. The reconfigured parameters are passed via `NV_ENC_RECONFIGURE_PARAMS::reInitEncodeParams`. It is much more efficient than recreating a new encode session.

Features like dynamic resolution change, bitrate change, resetting encoder are now supported using this API. This API was introduced in NV Encode API version 3.0.

Currently reconfiguration of following parameters is not supported via this method:

1. Change in GOP structure (NV_ENC_CONFIG_H264::idrPeriod, NV_ENC_CONFIG::gopLength, NV_ENC_CONFIG:: frameIntervalP)
2. Change in sync-async mode (NV_ENC_INITIALIZE_PARAMS:: enableEncodeAsync)
3. Change in MaxWidth & MaxHeight (NV_ENC_INITIALIZE_PARAMS:: maxEncodeWidth, NV_ENC_INITIALIZE_PARAMS::maxEncodeHeight)
4. Change in PTDmode (NV_ENC_INITIALIZE_PARAMS::enablePTD)

Support for changing these parameters *may be* added in the future versions of the API.

Resolution change is possible only if maxEncodeWidth & maxEncodeHeight of NV_ENC_INITIALIZE_PARAMS is set while creating encoder session.

If the client wishes to change the resolution using this API, it is advisable to force the next frame following the reconfiguration as an IDR frame by setting NV_ENC_RECONFIGURE_PARAMS:: forceIDR to 1.

If the client wishes to reset the internal rate control state, set NV_ENC_RECONFIGURE_PARAMS::resetEncoder to 1.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2011-2013 NVIDIA Corporation. All rights reserved.