



# **Denglin Hamming<sup>TM</sup> V2**

## **dIVID Programming Guide**

DL-DG/SW-033A-02

2023-9-18

Copyright©苏州登临科技有限公司，2019 - 2025，版权所有。

未经苏州登临科技有限公司事先书面同意，不得以任何形式或方式复制或传播本文件的任何部分。

## 商标和许可



和其它苏州登临科技有限公司的其它登临科技的图标为苏州登临科技有限公司的商标。本手册中提及的所有其他商标均为其各自所有者的财产。

## 通知

所购买的产品、服务和特性由苏州登临科技有限公司与客户签订的合同规定。本文件中描述的所有或部分产品、服务和特性可能不在采购范围或使用范围内。除非合同中另有规定，本文件中的所有声明、信息和建议均按“原样”提供，无任何明示或暗示的保证或陈述。

本手册中的信息如有更改，恕不另行通知。本文件在编制过程中已尽一切努力确保内容的准确性，本文件中的所有声明、信息和建议不构成任何明示或暗示的保证。

苏州登临科技有限公司

苏州工业园区扬富路11号南岸新地一期商务楼5号1101室，江苏，中国

<http://www.denglin.ai>

Email : support@denglin.ai

## Change History

| Version | Change description                                  |
|---------|---|
| 02      | Add Chapter <b>3.3.7 Skip Frame Output to DDR</b> . |
| 01      | Initial version.                                    |

# Table of Contents

## Table of Contents

### 1. Overview

#### 1.1 Supported Standards

##### 1.1.1 Video Encoding

HEVC (H.265) encoding features

H.264 encoding features

##### 1.1.2 Video Decoding

HEVC (H.265) decoding features

H.264 decoding features

AVS decoding features

VP9 decoding features

VC-1 decoding features

MPEG4 decoding features

MPEG2 decoding features

H.263 decoding features

### 2. Work Flow

#### 2.1 Encode Overview

#### 2.2 Encode Flow

#### 2.3 Decode Overview

#### 2.4 Decode Flow

### 3. Using dVID APIs

#### 3.1 Get Device(s)

##### 3.1.1 Get One Device

##### 3.1.2 Get Multiple Devices

#### 3.2 Session

##### 3.2.1 Create a Session

Session count limit

##### 3.2.2 Activate a Session

##### 3.2.3 Run a Session

##### 3.2.4 Stop a Session

##### 3.2.5 Getting Session count

#### 3.3 Video Buffer

##### 3.3.1 Create/Destroy a Buffer

##### 3.3.2 Allocate/Free Buffer Memory

##### 3.3.3 Data Exchange On a Buffer

##### 3.3.4 Register/Unregister a Buffer to a VPU

##### 3.3.5 Send a Buffer to a VPU

##### 3.3.6 Scaling and Rotation

##### 3.3.7 Skip Frame Output to DDR

##### 3.3.8 Timestamp

##### 3.3.9 EOS

##### 3.3.10 ROI

##### 3.3.11 Buffer Flush

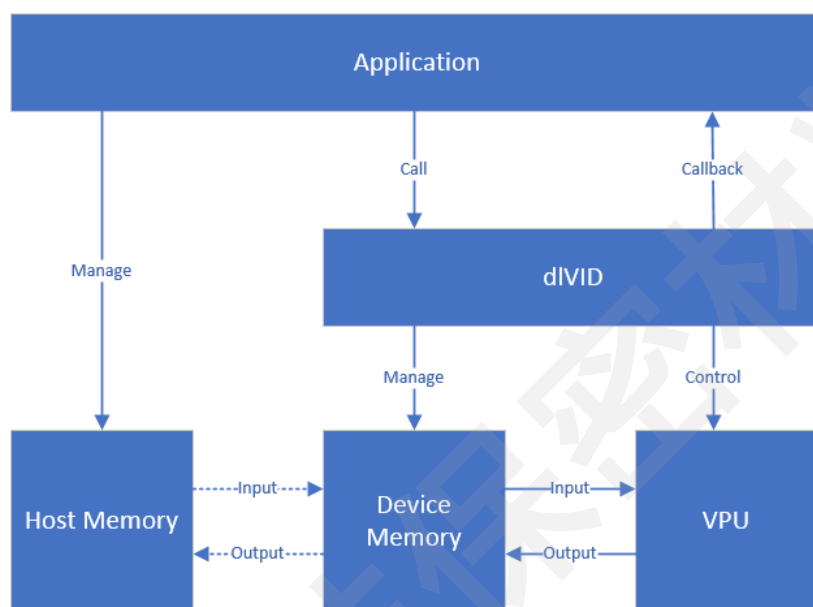
#### 3.4 Callbacks

#### 3.5 Event Loop

# 1. Overview

A DengLin Goldwasser™ AI accelerator card contains multiple video decoder engines and video encoder engines (referred to as VPUs) which provide fully hardware accelerated decode and encode capability on multiple video standards. A decoder engine can do decoding only. An encoder engine can do encoding only. VPUs can do codec simultaneously, and one VPU can do multiple decoding or encoding jobs simultaneously.

dVID library provides software APIs (referred to as dVID APIs) for programming all available VPUs.



Input and output data are put into video buffers. A VPU takes input data from the input buffer and put processed data into the output buffer. For encoding tasks, the input data is YUV/RGB frames, and the output data is compressed video streams. For decoding tasks, the input data is compressed video streams, and the output data is YUV frames. These specific video buffers can be managed through a set of dedicated dVID APIs.

Various events need to be handled while encoding or decoding. Dedicated callbacks must be defined by an application to handle these events. Some callbacks can remain undefined to let the corresponding events to be handled in a default manner.

## 1.1 Supported Standards

The codec standards and frame formats supported are listed below.

### 1.1.1 Video Encoding

The VPU supports encoding the following source-frame input formats:

- 1-plane YUV422, scan-line format, interleaved in YUYV or UYVY order.
- 1-plane RGB, 8-bit byte address order RGBA, BGRA, ARGB, or ABGR
- 2-plane YUV420, scan-line format, chroma interleaved in UV or VU order.
- 3-plane YUV420, scan-line format.

- 3-plane YUV420 10-in-16 bit scan line format.
- YUV420 10-bit, P010, little endian, MSB aligned.
- YUV420 10-bit, 2 × 2 tile format.

The following codec standards are supported for encoding:

- HEVC (H.265) Main.
- HEVC (H.265) Main 10 Profile.
- H.264 Baseline Profile (BP).
- H.264 Main Profile (MP).
- H.264 High Profile (HP).
- H.264 High 10 Profile.

### HEVC (H.265) encoding features

- The encoded bitstream complies with the HEVC (H.265) Main Profile.
- Maximum frame width of 4,096 pixels.
- Maximum frame height of 4,096 pixels.
- 8-bit or 10-bit sample depth with I, P, and B frames.
- Progressive encoding with 64 × 64 CTU size.
- Tiled mode, up to four tiles, horizontal splits only.
- Wavefront parallel encoding.
- The Motion Estimation (ME) search window is ±128 pixels horizontally, ±64 pixels vertically. ME search down to Quarter Picture Element (QPel) resolution.
- 8 × 8, 16 × 16, and 32 × 32 luma intra-modes.
- 4 × 4, 8 × 8, and 16 × 16 chroma intra-modes.
- 8 × 8, 16 × 16, and 32 × 32 inter-modes.
- 8 × 8, 16 × 16, and 32 × 32 transform size for luma.
- 4 × 4, 8 × 8, and 16 × 16 transform size for chromas.
- Skipped CUs, Merge modes.
- Deblocking.
- Constrained intra-prediction selectable.
- Fixed Quantization Parameters (QP) operation or rate-controlled operation.
- Rate controlled based on bitrate and buffer size settings. This is also known as leaky bucket.
- Selectable intra-frame refresh interval.
- Slice insertion on a CTU row granularity.
- Possible to limit the search window and the split options.
- Encoders do not prevent the output from exceeding the maximum number of bits per CTU.

### H.264 encoding features

- The encoded bitstream complies with the Baseline, Main, High, and High 10 Profiles.
- Maximum frame width of 4,096 pixels.
- Maximum frame height of 4,096 pixels.
- I, P, and B frames.
- Progressive encoding.

- Context Adaptive Binary Arithmetic Coding (CABAC) or Context Adaptive Variable Length Coding (CAVLC) entropy coding. (B frames are not supported with CAVLC entropy coding.)
- The Motion Estimation (ME) search window is  $\pm 128$  pixels horizontally,  $\pm 64$  pixels vertically.
- ME search down to Quarter Picture Element (QPEL) resolution.
- All  $4 \times 4$ ,  $8 \times 8$ ,  $16 \times 16$  luma, and  $8 \times 8$  chroma intra-modes evaluated.
- $8 \times 8$ , and  $16 \times 16$  inter-modes.
- $4 \times 4$  and  $8 \times 8$  transform.
- Skipped macroblocks.
- Deblocking.
- Constrained intra-prediction selectable.

### 1.1.2 Video Decoding

The VPU supports the following source-frame output formats when decoding:

- 2-plane YUV420 scan line format, chroma interleaved in UV or VU order.
- 3-plane YUV420 scan line format.
- YUV420 10-bit, P010, little endian, MSB aligned.
- YUV420 10-bit,  $2 \times 2$  tile format.

Note: Support for 3-plane formats is included to help you with testing. Do not use these for maximum performance.

The following codec standards are supported for decoding:

- HEVC (H.265) Main/Main10.
- H.264 Baseline, Main, High, and High10 Progressive Profile.
- AVS Part 2 (Jizhun).
- AVS Part 16 (AVS+, Guangdong).
- VP9 Profile 0 and VP9 Profile 2 at 10-bit depth, not 12-bit.
- VC-1 SP/MP/AP.
- MPEG4 SP/ASP.
- MPEG2 MP.
- H.263 Profile 0.

#### HEVC (H.265) decoding features

- The decoder is fully compliant to the Main and Main10 Profiles.
- Maximum frame width is 4,096 pixels.
- Maximum frame height is 4,096 pixels.
- Error concealment is performed in case of bit errors.
- Stream parameter information is output.

## H.264 decoding features

- The decoder is fully compliant to H.264 Baseline, Main, High, and High 10 progressive Profiles.
- For progressive streams: the maximum frame width is 4,096 pixels, and the maximum frame height is 4,096 pixels.
- For interlaced streams: the maximum frame width is 2,048 pixels, and the maximum frame height is 4,096 pixels.
- Error concealment is performed in case of bit errors.
- Stream parameter information is output.
- Escaping to prevent NAL unit start code emulation is always expected. This is independent of the NAL packet format setting. For more information, see ITU-T H.264 Annex B.

## AVS decoding features

- The decoder is fully compliant to AVS Part 2 (Jizhun) and Part 15 (AVS+, Guangdian) for YUV420 only.
- Maximum frame width is 1,920 pixels.
- Maximum frame height is 1,080 pixels.
- Error concealment is performed in case of bit errors.

## VP8 decoding features

- The decoder is fully compliant to the VP8 Specification.
- Maximum frame width is 2,048 pixels.
- Maximum frame height is 2,048 pixels.
- Error concealment is performed in case of bit errors.

## VP9 decoding features

- The decoder is fully compliant to Profile 0 and Profile 2, 10-bit. Profile 2 at 12-bit is not supported.
- The maximum frame width is 4,096 pixels.
- The maximum frame height is 4,096 pixels.
- Error concealment is performed in case of bit errors.
- Stream parameter information is output.

## VC-1 decoding features

- The decoder is fully compliant to VC-1 Simple, Main, and Advanced Profiles.
- Maximum frame width is 2,048 pixels.
- Maximum frame height is 4,096 pixels.
- Error concealment is performed in case of bit errors.
- Advanced Profile bitstream data is always expected to include the Encapsulation Mechanism. This is independent of the NAL packet format setting. For more information, see SMPTE-421M-2006 Annex E.



### MPEG4 decoding features

- The decoder is compliant to MPEG4 Simple Profile and Advanced Simple Profile.
- Global Motion Compensation (GMC) is limited to a single warp point.
- Maximum frame width is 2,048 pixels.
- Maximum frame height is 2,048 pixels.
- Error concealment is performed in case of bit errors.

### MPEG2 decoding features

- The decoder is compliant to MPEG2 Main Profile.
- Maximum frame width is 4,096 pixels.
- A maximum frame width of 2048 pixels for interlaced streams.
- Maximum frame height is 4,096 pixels.
- Error concealment is performed in case of bit errors.

### H.263 decoding features

- The decoder is compliant to H.263 Profile 0.
- Maximum frame width is 2,048 pixels.
- Maximum frame height is 2,048 pixels.
- Error concealment is performed in case of bit errors.

## 2. Work Flow

### 2.1 Encode Overview

Encoding: encode YUV/RBG frames into a compressed video stream such as H.264 and H.265.

To encode, first of all, get an encoder device. Then, create and activate an encode session on the device.

Buffers are created for input and output. Buffer data memory is allocated on device memory. Buffers are registered to the session. Each input YUV/RBG frame data is put into one input buffer. Generally use two input buffers for ping pong to get good performance. Input frame data is copied to the buffer by `cudaMemcpy()`. Input buffers are sent to the VPU after data is ready.

The session is running after input buffers and output buffers are sent to the VPU. The VPU starts to encode. The VPU returns the input buffer after input data is processed. The returned input buffer should be filled with new input data and sent to the VPU again. The VPU also returns output buffer filled with video stream data. The output buffer is sent to the VPU again after the data is copied out. Buffers are repeatedly sent to the VPU and returned all through the encoding process.

Encoding finishes when there is no more input frame data. Remained buffers in the VPU need to be flushed out. Then, stop the session, free buffer memory, destroy buffers, and destroy the session.

## 2.2 Encode Flow

At a high level, the following steps should be followed for encoding by using dVID APIs.

1. Get an encoder device.
2. Create a session.
3. Activate the session.
4. Set encode configuration if needed.
5. Prepare input and output buffers.
6. Run the session.
7. Run process event function in an infinite loop.
8. Handle events in callbacks
9. Quit the process event loop after all inputs are processed.
10. Clean up.

The above steps are explained in [Using dVID APIs](#) and demonstrated in the sample application included in Hamming™ SDK.

## 2.3 Decode Overview

Decoding: decode compressed video stream into uncompressed YUV frames.

Decoding work flow is just slightly different from encoding. First, user should detect the video stream format and create a corresponding decode session. Video stream format can be detected by demuxer. Second, the output buffer requirement is supplied by the VPU after parsing the input video stream. So the input buffer is sent to the VPU and run the session. Then, get the output buffer information from the VPU. Output buffers can be prepared after getting the information.

Each output buffer contains one frame.

## 2.4 Decode Flow

At a high level, the following steps should be followed for decoding any video content by using dVID APIs.

1. Get a decoder device.
2. Create a session.
3. Activate a session.
4. Prepare input buffers.
5. Run the session.
6. Run process event function in an infinite loop.
7. Prepare output buffers in callback.
8. Handle events in callbacks.
9. Quit the process event loop after all inputs are processed.
10. Clean up.

The above steps are explained in [Using dVID APIs](#) and demonstrated in the sample application included in the DL SDK.

## 3. Using dVID APIs

### 3.1 Get Device(s)

#### 3.1.1 Get One Device

First step for decoding or encoding is to get a `DL_VID_DEVICE`. A `DL_VID_DEVICE` represents one VPU on hardware.

Call **dvidGetDevice()** to get a `DL_VID_DEVICE`. Parameter `DL_VID_DEVICE_INFO` specifies the cluster mask, the channel mask, the device type mask which the VPU matches. If several VPUs match the mask, dVID APIs pick one from them.

```
// Select a GPU device
cudaSetDevice(0);

// Select a video decode device in this GPU device
deviceInfo.clusterMask = 0xf;
deviceInfo.channelMask = 0xf;
deviceInfo.deviceTypeMask = DL_VID_DEVICE_DECODER;
result = dvidGetDevice(&device, &deviceInfo);
```

For decoding, the device type must be `DL_VID_DEVICE_DECODER`.

For encoding, the device type must be `DL_VID_DEVICE_ENCODER`.

**CudaSetDevice()** must be called before **dvidGetDevice()** to decide which GPU the VPU is from.

#### 3.1.2 Get Multiple Devices

Call **dvidGetDevices()** to get multiple `DL_VID_DEVICE` at a time.

Parameter `vidDeviceInfo` can be `nullptr` to get all available devices.

## 3.2 Session

One VPU can process multiple streams concurrently. To protect processing of one video stream from another video stream, video workloads are partitioned into distinct video sessions. Each video session provides a context to encode or decode a single video stream. It is a logical sandbox in which a stream can be encoded or decoded and protected from other streams.

#### 3.2.1 Create a Session

Call **dvidCreateSession()** to create a session. More than one session can be created and run on the same `DL_VID_DEVICE`. The codec format must be the decode format `DL_VID_CODEC_FORMAT_DECODE_XXXX`. Also the codec format must be the encode format `DL_VID_CODEC_FORMAT_ENCODE_XXXX`. Callbacks are bound to the session. Callback will be called when certain codec events need to be handled, for example, the VPU work state

changes, output buffer is ready, input buffer is drained. Callback function can be null if no need to handle that event. For details on callbacks, see [Callbacks](#).

```
// Create video decode session on this decode device
sessionCreateInfo.device = device;
sessionCreateInfo.codecFormat = DL_VID_CODEC_FORMAT_DECODE_H265;
sessionCreateInfo.pfnStateChanged = HandleStateChanged;
sessionCreateInfo.pfnEmptiedInput = HandleEmptiedInput;
sessionCreateInfo.pfnFilledOutput = HandleFilledOutput;
sessionCreateInfo.pfnOutputFlushed = HandleOutputFlushed;
sessionCreateInfo.pfnFrameAllocParams = HandleFrameAllocParams;
sessionCreateInfo.pfnSequenceParams = HandleSequenceParams;
result = dlvidCreateSession(&session, &sessionCreateInfo);
```

### Session count limit

There is a max session count limit on VPU. The max session count is typical 8 or 16 depending on the product.

Creating a session will get an error when the maximum session count is reached on a VPU.

### 3.2.2 Activate a Session

Call **dlvidActivateSession()** to activate a session. Session must be activated before running. Activating a session prepares hardware and puts the VPU into the ready-to-run state.

```
// Activate session
result = dlvidActivateSession(session);
```

### 3.2.3 Run a Session

Call **dlvidRunSession()** to run the session. The VPU starts to work with the input buffer.

```
// Run session, decode start
result = dlvidRunSession(session);
```

### 3.2.4 Stop a Session

Call **dlvidStopSession()** to stop the running session.

Some bitstream may change the frame size. Stop the session, and then flush old buffers and send new-size buffer to the VPU; after that, run the session to continue decoding with the new frame size.

```
// Stop session
dlvidStopSession(session);
```

### 3.2.5 Getting Session count

Call **dlvidGetSessionCount()** to get how many sessions is running on a dVID device.

```
result = dlvidGetSessionCount(&session_count, device);
```

## 3.3 Video Buffer

Buffers are passed between the dVID and the VPU as input and output. Buffers contain video stream or frame data for codec and description for the data. For output buffer, the VPU output format is set by buffer -> format .

An input buffer queue and a output buffer queue are on the VPU for a session. Input buffers and empty output buffers are sent to the input queue and output queue. The VPU uses the buffers as input and output. When a input buffer data is processed, the input callback notifies the user to fill the buffer with new input data. When a output buffer data is ready, the output callback notifies the user to get the output data. The VPU can work on the next buffer in the queue without waiting. If no more input or output buffer is available, the VPU stops codec and waits for a buffer. So lack of buffer may impact the performance.

### 3.3.1 Create/Destroy a Buffer

Call **dlvidCreateBuffer()** to create a buffer. Buffer information such as buffer type, frame height and frame width, is in **DL\_VID\_BUFFER\_CREATE\_INFO** . Most buffer information values cannot be changed after being created.

```
// Create a output buffer
DL_VID_BUFFER_CREATE_INFO bufferCreateInfo = {};
bufferCreateInfo.device = dlvidGetSessionDevice(session);
bufferCreateInfo.format = DL_VID_BUFFER_FORMAT_YUV420P;
bufferCreateInfo.size = frameSize;
bufferCreateInfo.height = frameHeight;
bufferCreateInfo.width = frameWidth;
bufferCreateInfo.stride = frameWidth;
bufferCreateInfo.strideAlign = 1;

dlvidCreateBuffer(&bufferCreateInfo, &outputBuffer);
```

Call **dlvidDestroyBuffer()** to destroy a buffer.

```
// Destroy a buffer
dlvidDestroyBuffer(outputBuffer)
```

### 3.3.2 Allocate/Free Buffer Memory

The buffer data memory is on the device memory because the VPU cannot work with the host memory.

Call **dldvidAllocateBufferMemory()** to allocate the device memory for buffers. The memory size is set by `DL_VID_BUFFER_CREATE_INFO` when a buffer is created.

**Note:** Driver may reserve more device memory for all VPUs in the same cluster when allocating device memory for one VPU. Denglin recommends that you keep VPUs' load balance to avoid wasting device memory.

```
// Allocate buffer memory
dldvidAllocateBufferMemory(outputBuffer);
```

Call **dldvidFreeBufferMemory()** to free the device memory for a buffer. Device memory must be freed before the buffer is destroyed.

```
// Free buffer memory
dldvidFreeBufferMemory(outputBuffer);
```

### 3.3.3 Data Exchange On a Buffer

Input data need to be copied to the buffer data memory on the device memory. And output data also need to be copied from the buffer data memory.

To copy data to or from the buffer data memory, use **cudaMemcpy()**.

Call **dldvidGetBufferMemoryPointer()** to get the CUDA pointer for **cudaMemcpy()**.

Call **dldvidGetBufferFilledLength()** to get the data size. Get the data size when copying data from a buffer.

Call **dldvidSetBufferFilledLength()** to set the data size. Set the data size after copying data to a buffer.

```
// Copy host data to buffer
cudaMemcpy(dldvidGetBufferMemoryPointer(buffer),
           hostBuffer, dataLength, cudaMemcpyHostToDevice);
dldvidSetBufferFilledLength(buffer, dataLength);

// Copy buffer data to host
cudaMemcpy(hostBuffer, dldvidGetBufferMemoryPointer(buffer),
           dataLength, cudaMemcpyDeviceToHost);
```

### 3.3.4 Register/Unregister a Buffer to a VPU

Buffers must be registered to a VPU before being sent to the VPU. Registering buffer lets the VPU know how to use the buffer and do memory mapping. Each buffer only needs to register once. Generally, the user registers enough buffers and reuses them until codec finishes. For example, register 2 input buffers and send them to the VPU. The VPU returns the input buffer one by one after input data is processed. Then, the returned input buffer is filled with new input data and sent to the VPU again.

Call **dldvidRegisterBuffer()** to register a buffer to a VPU.

```
// Register a output buffer
dldvidRegisterBuffer(session, DL_VID_BUFFER_OUT, outputBuffer);
```

Call **dldvidUnregisterBuffer()** to unregister a buffer.

```
// Unregister output buffer
dldvidUnregisterBuffer(session, outputBuffer);
```

### 3.3.5 Send a Buffer to a VPU

Buffers can be sent to a VPU after being registered. Unregistered buffer sent to a VPU causes error.

Call **dldvidEmptyBuffer()** to send input buffer to a VPU.

```
// Send input buffer to VPU
dldvidEmptyBuffer(session, buffer);
```

Call **dldvidFillBuffer()** to send output buffer to a VPU.

```
// Send output buffer to VPU
dldvidFillBuffer(session, buffer);
```

### 3.3.6 Scaling and Rotation

Decode only. One or multiple values of `DL_VID_BUFFER_FRAME_FLAG` can be assigned to `DL_VID_BUFFER_CREATE_INFO::flags`. The created `DL_VID_BUFFER` will get the scaled and rotated output frame affected by the flags when decoding.

### 3.3.7 Skip Frame Output to DDR

Decode only. Decode output buffer can be set not output to DDR to save DDR bandwidth.

To use this feature `DL_VID_BUFFER` must be created with `DL_VID_BUFFER_FRAME_FLAG_SKIPFRAME` assigned to `DL_VID_BUFFER_CREATE_INFO::flags`. And the buffer should not allocate memory by **dldvidAllocateBufferMemory()**.

### 3.3.8 Timestamp

Timestamp can be set on an input buffer, so the corresponding output buffer will have the same timestamp.

**dldvidSetBufferTimestamp()** sets the timestamp on an input buffer.

**dldvidGetBufferTimestamp()** gets the timestamp on an output buffer.

Note: For decoding. The input buffer with timestamp must set flag `DL_VID_BUFFER_FLAG_ENDOFFRAME` if the input buffer is one frame data. Or the output buffer timestamp maybe incorrect.

For example, `dlvidSetBufferFlags(buffer, DL_VID_BUFFER_FLAG_ENDOFFRAME)`

### 3.3.9 EOS

The `DL_VID_BUFFER_FLAG_EOS` flag needs to be set on the last input buffer.

The last output buffer will get the `DL_VID_BUFFER_FLAG_EOS` flag.

Use `dlvidSetBufferFlags()` to set the `DL_VID_BUFFER_FLAG_EOS` flag.

Use `dlvideGetBufferFlags()` to get the `DL_VID_BUFFER_FLAG_EOS` flag.

### 3.3.10 ROI

Encode only. ROI (region of interest) can be set on the input buffer to control the encode quality of each region.

Use `dlvidSetBufferRoiRegions()` to set ROI regions.

### 3.3.11 Buffer Flush

Calling `dlvidFlushOutput()` is only allowed after `pfnSequenceParams()` callback or after the session is stopped.

Calling `dlvidFlushInput()` is only allowed after the session is stopped.

Otherwise, it may cause session error and the session may stop working.

## 3.4 Callbacks

Callbacks are used to handle certain codec events, for example, output buffer is ready, or input buffer is processed.

Callbacks are listed below:

**RpcPrint:** VPU requests printing message.

**Error:** VPU reports an hardware error.

**StateChanged:** Notify session state change.

**EmptiedInput:** Notify an input buffer is processed and returned. User should feed in new input data and send the buffer to the VPU again.

**FilledOutput:** Notify an output buffer is filled and returned. User should get the output data and send the buffer to a VPU again.

**InputFlushed:** Notify input buffer queue on the VPU is flushed and all input buffers are returned. After `dlvidFlushInput()` call.



**OutputFlushed:** Notify output buffer queue on the VPU is flushed and all output buffer is returned. Response to the **dvidFlushOutput()** call.

**FrameAllocParams:** Decode only. Notify the output frame parameters. Triggered after the bitstream header is parsed. The output buffer size is also in the parameters.

**SequenceParams:** Decode only. Notify the sequence parameters. Triggered after the bitstream header is parsed. The minimum number of output buffers required is in the parameters. Must call **dvidFlushOutput()** in this callback and create, send output buffers to the VPU in the **OutputFlushed** callback.

**BufferParam:** Decode only. Optional for some bitstreams. Notify the buffer parameters for bitstreams, such as color description for HDR, and display size for VP9.

## 3.5 Event Loop

Call **dvidProcessSessionEvent()** in an infinite loop to start the event process loop. Callbacks are called for the corresponding event.

```
// Session event process loop
while (!sessionUserData.outputFlushed) {
    result = dvidProcessSessionEvent(session, &sessionUserData, 500);
}
```