# oneVPL Specification

Intel

Nov 16, 2023

# CONTENTS

The oneAPI Video Processing Library (oneVPL) is a programming interface for video decoding, encoding, and processing to build portable media pipelines on CPUs, GPUs, and other accelerators. It provides device discovery and selection in media centric and video analytics workloads and API primitives for zero-copy buffer sharing. oneVPL is backwards and cross-architecture compatible to ensure optimal execution on current and next generation hardware without source code changes.

## oneVPL Specification Version

This document contains oneVPL specification version of 2.10.0.

Latest published version of oneVPL specification: https://spec.oneapi.com/onevpl/latest/index.html.

CHAPTER

ONE

# ONEVPL FOR INTEL® MEDIA SOFTWARE DEVELOPMENT KIT USERS

oneVPL is source compatible with Intel® Media Software Development Kit. Applications can use Intel® Media Software Development Kit to target older hardware and oneVPL to target everything else. Some obsolete features of Intel® Media Software Development Kit have been omitted from oneVPL. Hereinafter the term "Legacy" will be used to describe a behavior when oneVPL is called by Intel® Media Software Development Kit applications.

## 1.1 oneVPL Ease of Use Enhancements

oneVPL provides improved ease of use compared to Intel® Media Software Development Kit. Ease of use enhancements include the following:

- Smart dispatcher with discovery of implementation capabilities. See *oneVPL Session* for more details.
- Simplified decoder initialization. See *Decoding Procedures* for more details.
- New memory management and components (session) interoperability. See *Internal Memory Management* and *Decoding Procedures* for more details.

## 1.2 New APIs in oneVPL

oneVPL introduces new functions that are not available in Intel® Media Software Development Kit.

New oneVPL dispatcher functions:

- `MFXLoad()`
- `MFXUnload()`
- `MFXCreateConfig()`
- `MFXSetConfigFilterProperty()`
- `MFXEnumImplementations()`
- `MFXCreateSession()`
- `MFXDispReleaseImplDescription()`

New oneVPL memory management functions:

- `MFXMemory_GetSurfaceForVPP()`
- `MFXMemory_GetSurfaceForVPPOut()`

- *MFXMemory_GetSurfaceForEncode()*
- *MFXMemory_GetSurfaceForDecode()*

New oneVPL implementation capabilities retrieval functions:

- *MFXQueryImplsDescription()*
- *MFXReleaseImplDescription()*

New oneVPL session initialization:

- *MFXInitialize()*

## 1.3 Intel® Media Software Development Kit Feature Removals

The following Intel® Media Software Development Kit features are considered obsolete and are not included in oneVPL:

- **Audio support.** oneVPL is intended for video processing. Audio APIs that duplicate functionality from other audio libraries such as Sound Open Firmware have been removed.
- **ENC and PAK interfaces.** Part of the Flexible Encode Infrastructure (FEI) and plugin interfaces which provide additional control over the encoding process for AVC and HEVC encoders. This feature was removed because it is not widely used by customers.
- **User plugins architecture.** oneVPL enables robust video acceleration through API implementations of many different video processing frameworks. Support of a Intel® Media Software Development Kit user plugin framework is obsolete. Intel® Media Software Development Kit RAW acceleration (Camera API) which is implemented as plugin is also obsolete, oneVPL enables RAW acceleration (Camera API) through oneVPL runtime such as oneVPL GPU runtime.
- **External buffer memory management.** A set of callback functions to replace internal memory allocation is obsolete.
- **Video Processing extended runtime functionality.** Video processing function MFXVideoVPP_RunFrameVPPAsyncEx is used for plugins only and is obsolete.
- **External threading.** The new threading model makes the MFXDoWork function obsolete.
- **Multi-frame encode.** A set of external buffers to combine several frames into one encoding call. This feature was removed because it is device specific and not commonly used.
- **Surface Type Neutral Transcoding.** Opaque memory support is removed and replaced with internal memory allocation concept.
- **Raw Acceleration.** Intel® Media Software Development Kit RAW acceleration (Camera API) which is implemented as plugin is obsolete, replaced by oneVPL and oneVPL runtime implementation. oneVPL reused most of Intel® Media Software Development Kit Camera API, but oneVPL camera API is not backward compatible with Intel® Media Software Development Kit camera API due to obsolete plugin mechanism in oneVPL and a few difference between oneVPL and Intel® Media Software Development Kit. The major difference between oneVPL and Intel® Media Software Development Kit are listed: 1) mfxCamGammaParam and mfxExtCamGammaCorrection are removed in oneVPL; 2) Added reserved bytes in mfxExtCamHotPixelRemoval, mfxCamVignetteCorrectionParam and mfxCamVignetteCorrectionElement for future extension; 3) Changed CCM from mfxF64 to mfxF32 in mfxExtCamColorCorrection3x3 and added more reserved bytes.

## 1.4 Intel® Media Software Development Kit API Removals

The following Intel® Media Software Development Kit functions are not included in oneVPL:

- **Audio related functions**
  - MFXAudioCORE_SyncOperation()
  - MFXAudioDECODE_Close()
  - MFXAudioDECODE_DecodeFrameAsync()
  - MFXAudioDECODE_DecodeHeader()
  - MFXAudioDECODE_GetAudioParam()
  - MFXAudioDECODE_Init()
  - MFXAudioDECODE_Query()
  - MFXAudioDECODE_QueryIOSize()
  - MFXAudioDECODE_Reset()
  - MFXAudioENCODE_Close()
  - MFXAudioENCODE_EncodeFrameAsync()
  - MFXAudioENCODE_GetAudioParam()
  - MFXAudioENCODE_Init()
  - MFXAudioENCODE_Query()
  - MFXAudioENCODE_QueryIOSize()
  - MFXAudioENCODE_Reset()
- **Flexible encode infrastructure functions**
  - MFXVideoENC_Close()
  - MFXVideoENC_GetVideoParam()
  - MFXVideoENC_Init()
  - MFXVideoENC_ProcessFrameAsync()
  - MFXVideoENC_Query()
  - MFXVideoENC_QueryIOSurf()
  - MFXVideoENC_Reset()
  - MFXVideoPAK_Close()
  - MFXVideoPAK_GetVideoParam()
  - MFXVideoPAK_Init()
  - MFXVideoPAK_ProcessFrameAsync()
  - MFXVideoPAK_Query()
  - MFXVideoPAK_QueryIOSurf()
  - MFXVideoPAK_Reset()
- **User plugin functions**

- MFXAudioUSER_ProcessFrameAsync()
- MFXAudioUSER_Register()
- MFXAudioUSER_Unregister()
- MFXVideoUSER_GetPlugin()
- MFXVideoUSER_ProcessFrameAsync()
- MFXVideoUSER_Register()
- MFXVideoUSER_Unregister()
- MFXVideoUSER_Load()
- MFXVideoUSER_LoadByPath()
- MFXVideoUSER_UnLoad()
- MFXDoWork()

- **Memory functions**
  - MFXVideoCORE_SetBufferAllocator()
- **Video processing functions**
  - MFXVideoVPP_RunFrameVPPAsyncEx()
- **Memory type and IOPattern enumerations**
  - MFX_IOPATTERN_IN_OPAQUE_MEMORY
  - MFX_IOPATTERN_OUT_OPAQUE_MEMORY
  - MFX_MEMTYPE_OPAQUE_FRAME

**Important:** Corresponding extension buffers are also removed.

The following behaviors occur when attempting to use a Intel® Media Software Development Kit API that is not supported by oneVPL:

- Code compiled with the oneVPL API headers will generate a compile and/or link error when attempting to use a removed API.
- Code previously compiled with Intel® Media Software Development Kit and executed using a oneVPL runtime will generate an *MFX_ERR_NOT_IMPLEMENTED* error when calling a removed function.

## 1.5 Intel® Media Software Development Kit Legacy API

oneVPL contains following header files from Intel® Media Software Development Kit included for the simplification of existing applications migration to oneVPL:

- mfxvideo++.h

**Important:** Intel® Media Software Development Kit obsolete API removed from those header files. Code compiled with the oneVPL API headers will generate a compile and/or link error when attempting to use a removed API.

CHAPTER

TWO

# ARCHITECTURE

oneVPL functions fall into the following categories:

**DECODE**
: Functions that decode compressed video streams into raw video frames

**ENCODE**
: Functions that encode raw video frames into compressed bitstreams

**VPP**
: Functions that perform video processing on raw video frames

**DECODE_VPP**
: Functions that perform combined operations of decoding and video processing

**CORE**
: Auxiliary functions for synchronization

**Misc**
: Global auxiliary functions

With the exception of the global auxiliary functions, oneVPL functions are named after their functioning domain and category. oneVPL exposes video domain functions.

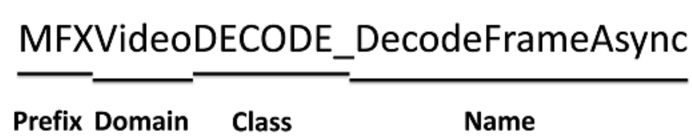



Fig. 1: oneVPL function name notation

Applications use oneVPL functions by linking with the oneVPL dispatcher library.

The dispatcher library identifies the hardware acceleration device on the running platform, determines the most suitable platform library for the identified hardware acceleration, and then redirects function calls accordingly.

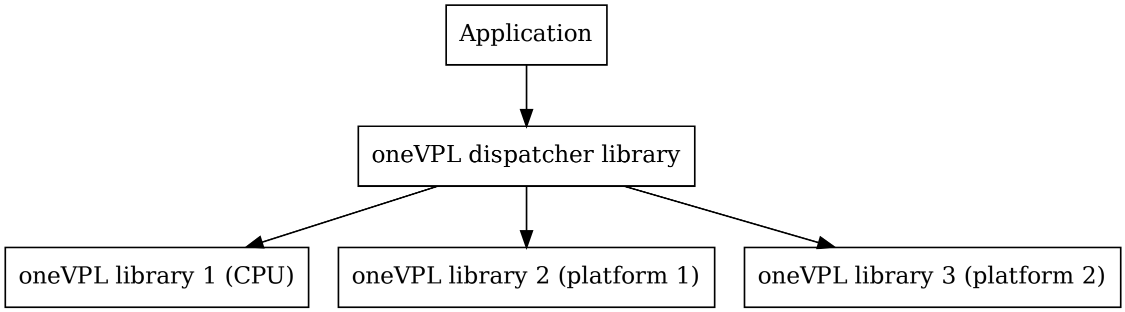



Fig. 2: oneVPL dispatching mechanism

## 2.1 Video Decoding

The *DECODE* class of functions take a compressed bitstream as input and converts it to raw frames as output.

DECODE processes only pure or elementary video streams with the exception of AV1/VP9/VP8 decoders, which accept the IVF format. The library can process bitstreams that reside in an IVF container but cannot process bitstreams that reside in any other container format, such as MP4 or MPEG.

The application must first demultiplex the bitstreams. Demultiplexing extracts pure video streams out of the container format. The application can provide the input bitstream as one complete frame of data, a partial frame (less than one complete frame), or as multiple frames. If only a partial frame is provided, DECODE internally constructs one frame of data before decoding it.

The time stamp of a bitstream buffer must be accurate to the first byte of the frame data. For H.264 the first byte of the frame data comes from the NAL unit in the video coding layer. For MPEG-2 or VC-1 the first byte of the frame data comes from the picture header. DECODE passes the time stamp to the output surface for audio and video multiplexing or synchronization.

Decoding the first frame is a special case because DECODE does not provide enough configuration parameters to correctly process the bitstream. DECODE searches for the sequence header (a sequence parameter set in H.264 or a sequence header in MPEG-2 and VC-1) that contains the video configuration parameters used to encode subsequent video frames. The decoder skips any bitstream prior to the sequence header. In the case of multiple sequence headers in the bitstream, DECODE adopts the new configuration parameters, ensuring proper decoding of subsequent frames.

DECODE supports repositioning of the bitstream at any time during decoding. Because there is no way to obtain the correct sequence header associated with the specified bitstream position after a position change, the application must supply DECODE with a sequence header before the decoder can process the next frame at the new position. If the sequence header required to correctly decode the bitstream at the new position is not provided by the application, DECODE treats the new location as a new “first frame” and follows the procedure for decoding first frames.

## 2.2 Video Encoding

The *ENCODE* class of functions take raw frames as input and compresses them into a bitstream.

Input frames usually come encoded in a repeated pattern called the Group of Picture (GOP) sequence. For example, a GOP sequence can start with an I-frame followed by a few B-frames, a P-frame, and so on. ENCODE uses an MPEG-2 style GOP sequence structure that can specify the length of the sequence and the distance between two keyframes: I- or P-frames. A GOP sequence ensures that the segments of a bitstream do not completely depend upon each other. It also enables decoding applications to reposition the bitstream.

ENCODE processes input frames in two ways;

- **Display order:** ENCODE receives input frames in the display order. GOP structure parameters specify the GOP sequence during ENCODE initialization. Scene changes resulting from the video processing stage of a pipeline can alter the GOP sequence.
- **Encoded order:** ENCODE receives input frames in their encoding order. The application must specify the exact input frame type for encoding. ENCODE references GOP parameters to determine when to insert information, such as an end-of-sequence, into the bitstream.

An ENCODE output consists of one frame of a bitstream with the time stamp passed from the input frame. The time stamp is used for multiplexing subsequent video with other associated data such as audio. oneVPL provides only pure video stream encoding. The application must provide its own multiplexing.

ENCODE supports the following bitrate control algorithms: constant bitrate, variable bitrate (VBR), and constant quantization parameter (QP). In the constant bitrate mode, ENCODE performs stuffing when the size of the least-compressed frame is smaller than what is required to meet the hypothetical reference decoder (HRD) buffer requirements (or VBR requirements). (Stuffing is a process that appends zeros to the end of encoded frames.)

## 2.3 Video Processing

Video processing functions (*VPP*) take raw frames as input and provide raw frames as output.

The actual conversion process is a chain operation with many single-function filters.

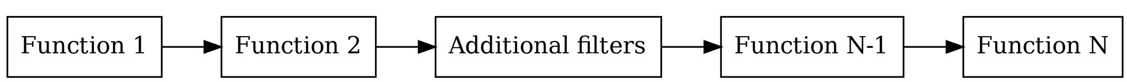



Fig. 3: Video processing operation pipeline

The application specifies the input and output format; oneVPL configures the pipeline according to the specified input and output formats. The application can also attach one or more hint structures to configure individual filters or turn them on and off. Unless specifically instructed, oneVPL builds the pipeline in a way that best utilizes hardware acceleration or generates the best video processing quality.

The *Video Processing Features table* shows oneVPL video processing features. The application can configure supported video processing features through the video processing I/O parameters. The application can also configure optional features through hints. See *Video Processing Procedures* for more details on how to configure optional filters.

Table 1: Video Processing Features

| Video Processing Features | Configuration |
|---|---|
| Convert color format from input to output | I/O parameters |
| De-interlace to produce progressive frames at the output | I/O parameters |
| Crop and resize the input frames | I/O parameters |
| Convert input frame rate to match the output | I/O parameters |
| Perform inverse telecine operations | I/O parameters |
| Fields weaving | I/O parameters |
| Fields splitting | I/O parameters |
| Remove noise | Hint (optional feature) |
| Enhance picture details/edges | Hint (optional feature) |
| Adjust the brightness, contrast, saturation, and hue settings | Hint (optional feature) |
| Perform image stabilization | Hint (optional feature) |
| Convert input frame rate to match the output, based on frame interpolation | Hint (optional feature) |
| Perform detection of picture structure | Hint (optional feature) |

## 2.4 Video Decoding with multiple video processing

The *DECODE_VPP* class of functions take a compressed bitstream as input, converts it to raw frames and applies video processing filters to raw frames. Users can set several output channels where each channel represents a list of video processing filters applied for decoded frames.

The *DECODE_VPP* supports only internal allocation.

CHAPTER

THREE

# PROGRAMMING GUIDE

This chapter describes the concepts used in programming with oneVPL.

The application must use the include file `mfx.h` for C/C++ programming and link the oneVPL dispatcher library `libvpl.so`.

Include these files:

```
#include "mfx.h"    /* oneVPL include file */
```

Link this library:

```
libvpl.so               /* oneVPL dynamic dispatcher library (Linux\*) */
```

## 3.1 Status Codes

The oneVPL functions are organized into categories for easy reference. The categories include *ENCODE* (encoding functions), *DECODE* (decoding functions), and *VPP* (video processing functions).

**Init**, **Reset**, and **Close** are member functions within the ENCODE, DECODE, and VPP classes that initialize, restart, and deinitialize specific operations defined for the class. Call all member functions of a given class within the **Init - Reset - Close** sequence, except **Query** and **QueryIOSurf**. **Reset** functions are optional within the sequence.

The **Init** and **Reset** member functions set up necessary internal structures for media processing. **Init** functions allocate memory and **Reset** functions only reuse allocated internal memory. If oneVPL needs to allocate additional memory, **Reset** can fail. **Reset** functions can also fine-tune ENCODE and VPP parameters during those processes or reposition a bitstream during DECODE.

All oneVPL functions return status codes to indicate if an operation succeeded or failed. The `mfxStatus::MFX_ERR_NONE` status code indicates that the function successfully completed its operation. Error status codes are less than `mfxStatus::MFX_ERR_NONE` and warning status codes are greater than `mfxStatus::MFX_ERR_NONE`. See the `mfxStatus` enumerator for all defined status codes.

If a oneVPL function returns a warning, it has sufficiently completed its operation. Note that the output of the function might not be strictly reliable. The application must check the validity of the output generated by the function.

If a oneVPL function returns an error (except `mfxStatus::MFX_ERR_MORE_DATA`, `mfxStatus::MFX_ERR_MORE_SURFACE`, or `mfxStatus::MFX_ERR_MORE_BITSTREAM`), the function aborts the operation. The application must call either the **Reset** function to reset the class back to a clean state or the **Close** function to terminate the operation. The behavior is undefined if the application continues to call any class member functions without a **Reset** or **Close**. To avoid memory leaks, always call the **Close** function after **Init**.

## 3.2 oneVPL Session

Before calling any oneVPL functions, the application must initialize the library and create a oneVPL session. A oneVPL session maintains context for the use of any of *DECODE*, *ENCODE*, *VPP*, *DECODE_VPP* functions.

### 3.2.1 Intel® Media Software Development Kit Dispatcher (Legacy)

The `MFXInit()` or `MFXInitEx()` function starts (initializes) a session. The `MFXClose()` function closes (de-initializes) the session. To avoid memory leaks, always call `MFXClose()` after `MFXInit()`.

**Important:** `MFXInit()` and `MFXInitEx()` are deprecated starting from API 2.0. Applications must use `MFXLoad()` and `MFXCreateSession()` to initialize implementations.

**Important:** For backward compatibility with existent Intel® Media Software Development Kit applications oneVPL session can be created and initialized by the legacy dispatcher through `MFXInit()` or `MFXInitEx()` calls. In this case, the reported API version will be 1.255 on Intel® platforms with $X^e$ architecture.

The application can initialize a session as a software-based session (`MFX_IMPL_SOFTWARE`) or a hardware-based session (`MFX_IMPL_HARDWARE`). In a software-based session, the SDK functions execute on a CPU. In a hardware-base session, the SDK functions use platform acceleration capabilities. For platforms that expose multiple graphic devices, the application can initialize a session on any alternative graphic device using the `MFX_IMPL_HARDWARE`, `MFX_IMPL_HARDWARE2`, `MFX_IMPL_HARDWARE3`, or `MFX_IMPL_HARDWARE4` values of `mfxIMPL`.

The application can also initialize a session to be automatic ( `MFX_IMPL_AUTO` or `MFX_IMPL_AUTO_ANY`), instructing the dispatcher library to detect the platform capabilities and choose the best SDK library available. After initialization, the SDK returns the actual implementation through the `MFXQueryIMPL()` function.

Internally, the dispatcher works as follows:

1. Dispatcher searches for the shared library with the specific name:

| OS | Name | Description |
|---|---|---|
| Linux* | libmfxsw64.so.1 | 64-bit software-based implementation |
| Linux | libmfxsw32.so.1 | 32-bit software-based implementation |
| Linux | libmfxhw64.so.1 | 64-bit hardware-based implementation |
| Linux | libmfxhw64.so.1 | 32-bit hardware-based implementation |
| Windows* | libmfxsw32.dll | 64-bit software-based implementation |
| Windows | libmfxsw32.dll | 32-bit software-based implementation |
| Windows | libmfxhw64.dll | 64-bit hardware-based implementation |
| Windows | libmfxhw64.dll | 32-bit hardware-based implementation |

2. Once the library is loaded, the dispatcher obtains addresses for each SDK function. See the *Exported Functions/API Version table* for the list of functions to expose.

How the shared library is identified using the implementation search strategy will vary according to the OS.

- On Windows, the dispatcher searches the following locations, in the specified order, to find the correct implementation library:
    1. The `Driver Store` directory for the current adapter. All types of graphics drivers can install libraries in this directory. Learn more about Driver Store.

2. The directory specified for the current hardware under the registry key `HKEY_CURRENT_USER\Software\Intel\MediaSDK\Dispatch`.
3. The directory specified for the current hardware under the registry key `HKEY_LOCAL_MACHINE\Software\Intel\MediaSDK\Dispatch`.
4. The directory that is stored in these registry keys: `C:Program FilesIntelMedia SDK`. This directory is where legacy graphics drivers install libraries.
5. The directory where the current module (the module that links the dispatcher) is located (only if the current module is a dll).

After the dispatcher completes the main search, it additionally checks:

1. The directory of the exe file of the current process, where it looks for software implementation only, regardless of which implementation the application requested.
2. Default dll search. This provides loading from the directory of the application's exe file and from the `System32` and `SysWOW64` directories. Learn more about default dll search order.
3. The `System32` and `SysWOW64` directories, which is where DCH graphics drivers install libraries.

- On Linux, the dispatcher searches the following locations, in the specified order, to find the correct implementation library:
  1. Directories provided by the environment variable `LD_LIBRARY_PATH`.
  2. Content of the `/etc/ld.so.cache` cache file.
  3. Default path `/lib`, then `/usr/lib` or `/lib64`, and then `/usr/lib64` on some 64 bit OSs. On Debian: `/usr/lib/x86_64-linux-gnu`.
  4. SDK installation folder.

### 3.2.2 oneVPL Dispatcher

The oneVPL dispatcher extends the legacy dispatcher by providing additional ability to select the appropriate implementation based on the implementation capabilities. Implementation capabilities include information about supported decoders, encoders, and VPP filters. For each supported encoder, decoder, and filter, capabilities include information about supported memory types, color formats, and image (frame) size in pixels.

The recommended approach to configure the dispatcher's capabilities search filters and to create a session based on a suitable implementation is as follows:

1. Create loader with *`MFXLoad()`*.
2. Create loader's configuration with *`MFXCreateConfig()`*.
3. Add configuration properties with *`MFXSetConfigFilterProperty()`*.
4. Explore available implementations with *`MFXEnumImplementations()`*.
5. Create a suitable session with *`MFXCreateSession()`*.

The procedure to terminate an application is as follows:

1. Destroy session with *`MFXClose()`*.
2. Destroy loader with *`MFXUnload()`*.

**Note:** Multiple loader instances can be created.

**Note:** Each loader may have multiple configuration objects associated with it. When a configuration object is modified through *MFXSetConfigFilterProperty()* it implicitly impacts the state and configuration of the associated loader.

**Important:** One configuration object can handle only one filter property.

**Note:** Multiple sessions can be created by using one loader object.

When the dispatcher searches for the implementation, it uses the following priority rules:

1. Hardware implementation has priority over software implementation.
2. General hardware implementation has priority over VSI hardware implementation.
3. Highest API version has higher priority over lower API version.

**Note:** Implementation has priority over the API version. In other words, the dispatcher must return the implementation with the highest API priority (greater than or equal to the implementation requested).

How the shared library is identified using the implementation search strategy will vary according to the OS.

- On Windows, the dispatcher searches the following locations, in the specified order, to find the correct implementation library:
  1. The `Driver Store` directory for all available adapters. All types of graphics drivers can install libraries in this directory. Learn more about Driver Store. Applicable only for Intel implementations.
  2. The directory of the exe file of the current process.
  3. *PATH* environmental variable.
  4. For backward compatibility with older spec versions, dispatcher also checks user-defined search folders which are provided by *ONEVPL_SEARCH_PATH* environmental variable.
- On Linux, the dispatcher searches the following locations, in the specified order, to find the correct implementation library:
  1. Directories provided by the environment variable `LD_LIBRARY_PATH`.
  2. Default path `/lib`, then `/usr/lib` or `/lib64`, and then `/usr/lib64` on some 64 bit OSs. On Debian: `/usr/lib/x86_64-linux-gnu`.
  3. For backward compatibility with older spec versions, dispatcher also checks user-defined search folders which are provided by *ONEVPL_SEARCH_PATH* environmental variable.

**Important:** To prioritize loading of custom oneVPL library, users may set environment variable *ONEVPL_PRIORITY_PATH* with the path to the user-defined folder. All libraries found in the ONEVPL_PRIORITY_PATH have the same priority (higher than any others, and HW/SW or API version rules are not applied) and should be loaded/filtered according to *MFXSetConfigFilterProperty()*.

When oneVPL dispatcher searchers for the legacy Intel® Media Software Development Kit implementation it uses *legacy dispatcher search order*, excluding the current working directory and `/etc/ld.so.cache`.

The dispatcher supports different software implementations. The user can use the `mfxImplDescription::VendorID` field, the `mfxImplDescription::VendorImplID` field, or the `mfxImplDescription::ImplName` field to search for the specific implementation.

Internally, the dispatcher works as follows:

1. Dispatcher loads all shared libraries in the given search folders, whose names match any of the patterns in the following table:

| Windows 64-bit | Windows 32-bit | Linux 64-bit | Description |
|---|---|---|---|
| libvpl*.dll | libvpl*.dll | libvpl*.so* | Runtime library for any platform |
| libmfx64-gen.dll | libmfx32-gen.dll | libmfx-gen.so.1.2 | Runtime library for oneVPL for Intel® platforms with $X^e$ architecture |
| libmfxhw64.dll | libmfxhw32.dll | libmfxhw64.so.1 | Runtime library for Intel® Media Software Development Kit |

2. For each loaded library, the dispatcher tries to resolve address of the `MFXQueryImplsDescription()` function to collect the implementation's capabilities.
3. Once the user has requested to create the session based on this implementation, the dispatcher obtains addresses of each oneVPL function. See the *Exported Functions/API Version table* for the list of functions to export.

### 3.2.3 oneVPL Dispatcher Configuration Properties

The *Dispatcher Configuration Properties Table* shows property strings supported by the dispatcher. Table organized in the hierarchy way, to create the string, go from the left to right from column to column and concatenate strings by using . (dot) as the separator.

Table 1: Dispatcher Configuration Properties

| Structure name | Property | Value Data Type | Comment |
|---|---|---|---|
| `mfxImplDescription` | mfxImplDescription<br>.Impl | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription<br>.AccelerationMode | MFX_VARIANT_TYPE_U32 | The mode will be used for session initialization |
| | mfxImplDescription<br>.ApiVersion<br>.Version | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription<br>.ApiVersion<br>.Major | MFX_VARIANT_TYPE_U16 | |

continues on next page

Table 1 – continued from previous page

| Structure name | Property | Value Data Type | Comment |
|---|---|---|---|
| | mfxImplDescription .ApiVersion .Minor | MFX_VARIANT_TYPE_U16 | |
| | mfxImplDescription .ImplName | MFX_VARIANT_TYPE_PTR | Pointer to the null-terminated string. |
| | mfxImplDescription .License | MFX_VARIANT_TYPE_PTR | Pointer to the null-terminated string. |
| | mfxImplDescription .Keywords | MFX_VARIANT_TYPE_PTR | Pointer to the null-terminated string. |
| | mfxImplDescription .VendorID | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .VendorImplID | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxSurfacePoolMode | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxDeviceDescription .device .DeviceID | MFX_VARIANT_TYPE_PTR | Pointer to the null-terminated string. |
| | mfxImplDescription .mfxDeviceDescription .device .MediaAdapterType | MFX_VARIANT_TYPE_U16 | |
| | mfxImplDescription .mfxDecoderDescription .decoder .CodecID | MFX_VARIANT_TYPE_U32 | |

continues on next page

Table 1 – continued from previous page

| Structure name | Property | Value Data Type | Comment |
|---|---|---|---|
| | mfxImplDescription .mfxDecoderDescription .decoder .MaxcodecLevel | MFX_VARIANT_TYPE_U16 | |
| | mfxImplDescription .mfxDecoderDescription .decoder .decprofile .Profile | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxDecoderDescription .decoder .decprofile .Profile .decmemdesc .MemHandleType | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxDecoderDescription .decoder .decprofile .Profile .decmemdesc .Width | MFX_VARIANT_TYPE_PTR | Pointer to the *mfxRange32U* object |
| | mfxImplDescription .mfxDecoderDescription .decoder .decprofile .Profile .decmemdesc .Height | MFX_VARIANT_TYPE_PTR | Pointer to the *mfxRange32U* object |

continues on next page

Table 1 – continued from previous page

| Structure name | Property | Value Data Type | Comment |
|---|---|---|---|
| | mfxImplDescription .mfxDecoderDescription .decoder .decprofile .Profile .decmemdesc .ColorFormats | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxEncoderDescription .encoder .CodecID | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxEncoderDescription .encoder .MaxcodecLevel | MFX_VARIANT_TYPE_U16 | |
| | mfxImplDescription .mfxEncoderDescription .encoder .BiDirectionalPrediction | MFX_VARIANT_TYPE_U16 | |
| | mfxImplDescription .mfxEncoderDescription .encoder .ReportedStats | MFX_VARIANT_TYPE_U16 | |
| | mfxImplDescription .mfxEncoderDescription .encoder .encprofile .Profile | MFX_VARIANT_TYPE_U32 | |

continues on next page

Table 1 – continued from previous page

| Structure name | Property | Value Data Type | Comment |
|---|---|---|---|
| | mfxImplDescription .mfxEncoderDescription .encoder .encprofile .Profile .encmemdesc .MemHandleType | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxEncoderDescription .encoder .encprofile .Profile .encmemdesc .Width | MFX_VARIANT_TYPE_PTR | Pointer to the `mfxRange32U` object |
| | mfxImplDescription .mfxEncoderDescription .encoder .encprofile .Profile .encmemdesc .Height | MFX_VARIANT_TYPE_PTR | Pointer to the `mfxRange32U` object |
| | mfxImplDescription .mfxEncoderDescription .encoder .encprofile .Profile .encmemdesc .ColorFormats | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxVPPDescription .filter .FilterFourCC | MFX_VARIANT_TYPE_U32 | |

continues on next page

Table 1 – continued from previous page

| Structure name | Property | Value Data Type | Comment |
|---|---|---|---|
| | mfxImplDescription .mfxVPPDescription .filter .MaxDelayInFrames | MFX_VARIANT_TYPE_U16 | |
| | mfxImplDescription .mfxVPPDescription .filter .memdesc .MemHandleType | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxVPPDescription .filter .memdesc .Width | MFX_VARIANT_TYPE_PTR | Pointer to the *mfxRange32U* object |
| | mfxImplDescription .mfxVPPDescription .filter .memdesc .Height | MFX_VARIANT_TYPE_PTR | Pointer to the *mfxRange32U* object |
| | mfxImplDescription .mfxVPPDescription .filter .memdesc .format .InFormat | MFX_VARIANT_TYPE_U32 | |
| | mfxImplDescription .mfxVPPDescription .filter .memdesc .format .OutFormats | MFX_VARIANT_TYPE_U32 | |

continues on next page

Table 1 – continued from previous page

| Structure name | Property | Value Data Type | Comment |
|---|---|---|---|
| *mfxImplementedFunctio* | mfxImplementedFunctions .FunctionsName | MFX_VARIANT_TYPE_PTR | Pointer to the buffer with string |
| N/A | DXGIAdapterIndex | MFX_VARIANT_TYPE_U32 | Adapter index according to IDXGIFactory::EnumAdapters |
| N/A | AutoSelectImpl | MFX_VARIANT_TYPE_PTR | Pointer to a struct for automatic implementation *selection* |

**Important:** DXGIAdapterIndex property is available for Windows only and filters only hardware implementations.

Examples of the property name strings:

- mfxImplDescription.mfxDecoderDescription.decoder.decprofile.Profile
- mfxImplDescription.mfxDecoderDescription.decoder.decprofile.decmemdesc.MemHandleType
- mfxImplementedFunctions.FunctionsName

Following properties are supported in a special manner: they are used to send additional data to the implementation through the dispatcher. Application needs to use `MFXSetConfigFilterProperty()` to set them up but they don't influence on the implementation selection. They are used during the `MFXCreateSession()` function call to fine tune the implementation.

Table 2: Dispatcher's Special Properties

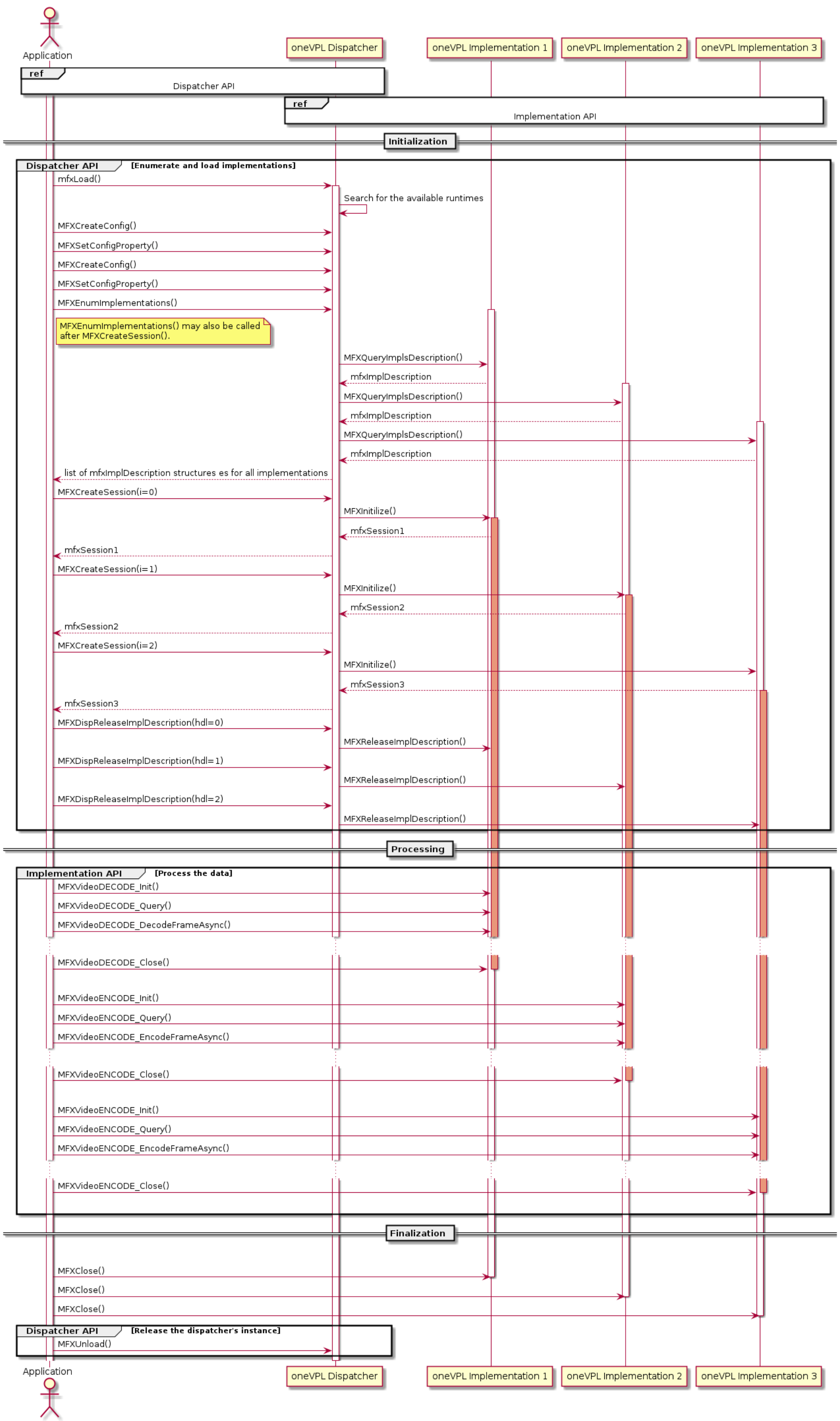
| Property Name | Property Value | Value data type |
|---|---|---|
| mfxHandleType | `mfxHandleType` | `mfxVariantType::MFX_VARIANT_TYPE_U32` |
| mfxHDL | `mfxHDL` | `mfxVariantType::MFX_VARIANT_TYPE_PTR` |
| NumThread | Unsigned fixed-point integer value | `mfxVariantType::MFX_VARIANT_TYPE_U32` |
| DeviceCopy | *Device copy* | `mfxVariantType::MFX_VARIANT_TYPE_U16` |
| ExtBuffer | Pointer to the extension buffer | `mfxVariantType::MFX_VARIANT_TYPE_PTR` |

### 3.2.4 oneVPL Dispatcher Interactions

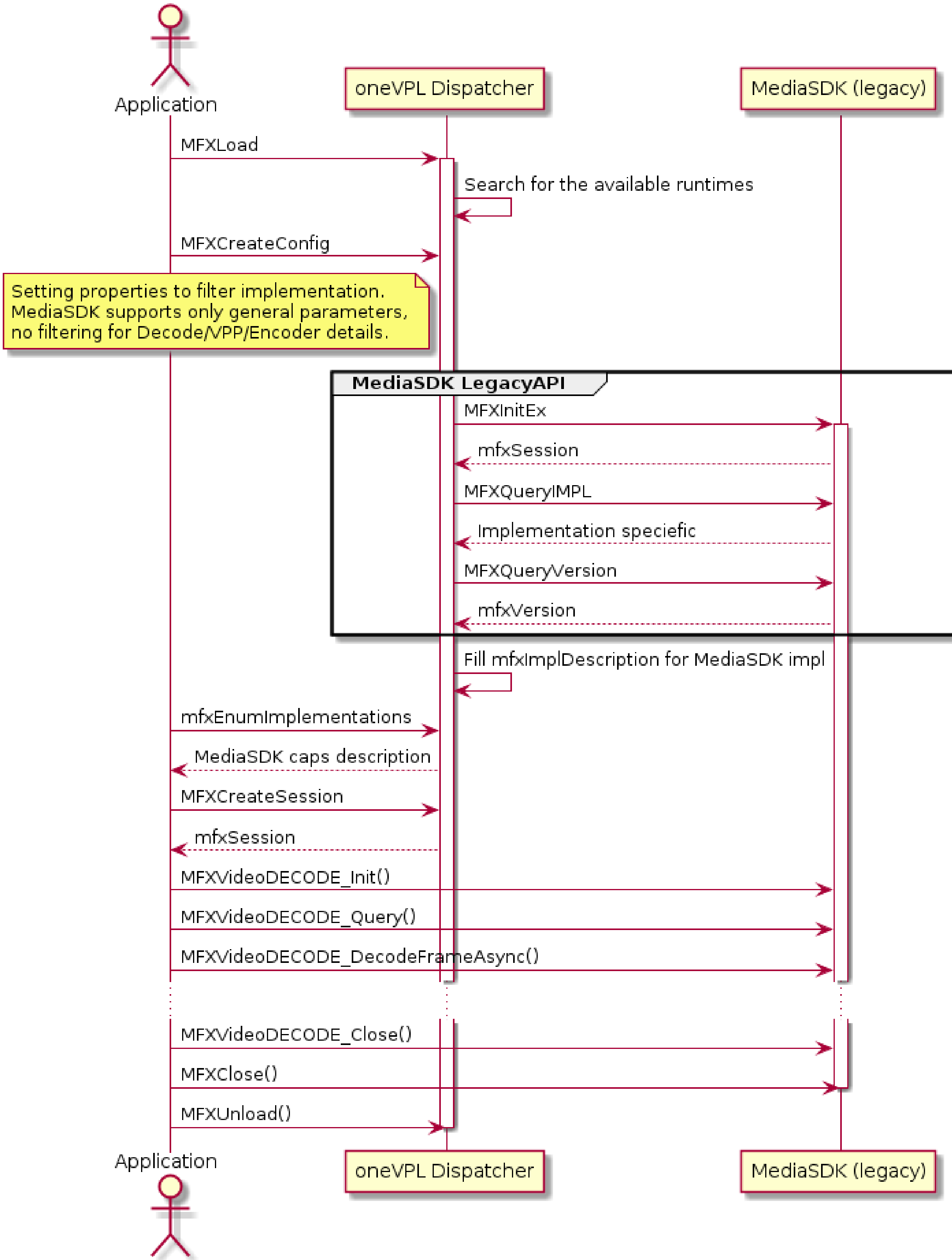
This sequence diagram visualize how application communicates with implementations via the dispatcher.

**Dispatcher API**
This API is implemented in the dispatcher.

**Implementation API**
This API is provided by the any implementation.


Application
oneVPL Dispatcher
oneVPL Implementation 1
oneVPL Implementation 2
oneVPL Implementation 3
ref
Dispatcher API
ref
Implementation API
Initialization
Dispatcher API
[Enumerate and load implementations]
mfxLoad()
Search for the available runtimes
MFXCreateConfig()
MFXSetConfigProperty()
MFXCreateConfig()
MFXSetConfigProperty()
MFXEnumImplementations()
MFXEnumImplementations() may also be called after MFXCreateSession().
MFXQueryImplsDescription()
mfxImplDescription
MFXQueryImplsDescription()
mfxImplDescription
MFXQueryImplsDescription()
mfxImplDescription
list of mfxImplDescription structures es for all implementations
MFXCreateSession(i=0)
MFXInitilize()
mfxSession1
mfxSession1
MFXCreateSession(i=1)
MFXInitilize()
mfxSession2
mfxSession2
MFXCreateSession(i=2)
MFXInitilize()
mfxSession3
mfxSession3
MFXDispReleaseImplDescription(hdl=0)
MFXReleaseImplDescription()
MFXDispReleaseImplDescription(hdl=1)
MFXReleaseImplDescription()
MFXDispReleaseImplDescription(hdl=2)
MFXReleaseImplDescription()
Processing
Implementation API
[Process the data]
MFXVideoDECODE_Init()
MFXVideoDECODE_Query()
MFXVideoDECODE_DecodeFrameAsync()
MFXVideoDECODE_Close()
MFXVideoENCODE_Init()
MFXVideoENCODE_Query()
MFXVideoENCODE_EncodeFrameAsync()
MFXVideoENCODE_Close()
MFXVideoENCODE_Init()
MFXVideoENCODE_Query()
MFXVideoENCODE_EncodeFrameAsync()
MFXVideoENCODE_Close()
Finalization
MFXClose()
MFXClose()
MFXClose()
Dispatcher API
[Release the dispatcher's instance]
MFXUnload()
Application
oneVPL Dispatcher
oneVPL Implementation 1
oneVPL Implementation 2
oneVPL Implementation 3

The oneVPL dispatcher is capable to load and initialize Intel® Media Software Development Kit legacy library. The sequence diagram below demonstrates the approach.



**Important:** The dispatcher doesn't filter and report *mfxDeviceDescription*, *mfxDecoderDescription*, *mfxEncoderDescription*, *mfxVPPDescription* when enumerates or creates Intel® Media Software Development Kit implementation. Once Intel® Media Software Development Kit is loaded applications have to use legacy approach to query capabilities.

### 3.2.5 oneVPL Dispatcher Debug Log

The debug output of the dispatcher is controlled with the *ONEVPL_DISPATCHER_LOG* environment variable. To enable log output, set the *ONEVPL_DISPATCHER_LOG* environment variable value equals to "ON".

By default, oneVPL dispatcher prints all log messages to the console. To redirect log output to the desired file, set the *ONEVPL_DISPATCHER_LOG_FILE* environmental variable with the file name of the log file.

### 3.2.6 Examples of Dispatcher's Usage

This code illustrates simple usage of dispatcher to load first available library:

```
mfxLoader loader = MFXLoad();
MFXCreateSession(loader,0,&session);
```

This code illustrates simple usage of dispatcher to load first available HW accelerated library:

```
mfxLoader loader = MFXLoad();
mfxConfig cfg = MFXCreateConfig(loader);
mfxVariant ImplValue;
ImplValue.Type = MFX_VARIANT_TYPE_U32;
ImplValue.Data.U32 = MFX_IMPL_TYPE_HARDWARE;
MFXSetConfigFilterProperty(cfg,(const mfxU8 *)"mfxImplDescription.Impl",ImplValue);
MFXCreateSession(loader,0,&session);
```

This code illustrates how multiple sessions from multiple loaders can be created:

```
// Create session with software based implementation
mfxLoader loader1 = MFXLoad();
mfxConfig cfg1 = MFXCreateConfig(loader1);
mfxVariant ImplValueSW;
ImplValueSW.Type = MFX_VARIANT_TYPE_U32;
ImplValueSW.Data.U32 = MFX_IMPL_TYPE_SOFTWARE;
MFXSetConfigFilterProperty(cfg1,(const mfxU8 *)"mfxImplDescription.Impl",ImplValueSW);
MFXCreateSession(loader1,0,&sessionSW);

// Create session with hardware based implementation
mfxLoader loader2 = MFXLoad();
mfxConfig cfg2 = MFXCreateConfig(loader2);
mfxVariant ImplValueHW;
ImplValueHW.Type = MFX_VARIANT_TYPE_U32;
ImplValueHW.Data.U32 = MFX_IMPL_TYPE_HARDWARE;
MFXSetConfigFilterProperty(cfg2,(const mfxU8 *)"mfxImplDescription.Impl",ImplValueHW);
MFXCreateSession(loader2,0,&sessionHW);

```

(continues on next page)

(continued from previous page)

```
// use both sessionSW and sessionHW
// ...
// Close everything
MFXClose(sessionSW);
MFXClose(sessionHW);
MFXUnload(loader1); // cfg1 will be destroyed here.
MFXUnload(loader2); // cfg2 will be destroyed here.
```

This code illustrates how multiple decoders from single loader can be created:

```
mfxLoader loader = MFXLoad();

// We want to have AVC decoder supported.
mfxConfig cfg1 = MFXCreateConfig(loader);
mfxVariant ImplValue;
ImplValue.Type = MFX_VARIANT_TYPE_U32;
ImplValue.Data.U32 = MFX_CODEC_AVC;
MFXSetConfigFilterProperty(cfg1,
            (const mfxU8 *)"mfxImplDescription.mfxDecoderDescription.decoder.CodecID", ImplValue);

// And we want to have HEVC encoder supported by the same implementation.
mfxConfig cfg2 = MFXCreateConfig(loader);
ImplValue.Type = MFX_VARIANT_TYPE_U32;
ImplValue.Data.U32 = MFX_CODEC_HEVC;
MFXSetConfigFilterProperty(cfg2,
            (const mfxU8 *)"mfxImplDescription.mfxEncoderDescription.encoder.CodecID", ImplValue);

// To create single session with both capabilities.
MFXCreateSession(loader,0,&session);
```

### 3.2.7 How To Check If Function is Implemented

There are two ways to check if particular function is implemented or not by the implementation.

This code illustrates how application can iterate through the whole list of implemented functions:

```
mfxHDL h;
// request pointer to the list. Assume that implementation supports that.
// Assume that `loader` is configured before.
mfxStatus sts = MFXEnumImplementations(loader, idx, MFX_IMPLCAPS_IMPLEMENTEDFUNCTIONS, &h);
// break if no idx
if (sts != MFX_ERR_NOT_FOUND) {
  // Cast typeless handle to structure pointer
  mfxImplementedFunctions *implemented_functions = (mfxImplementedFunctions*)h;

  // print out list of functions' name
  std::for_each(implemented_functions->FunctionsName, implemented_functions->FunctionsName +
```

(continues on next page)

(continued from previous page)

```
                                                        implemented_functions->NumFunctions,
          [](mfxChar*  functionName) {
              std::cout << functionName << " is implemented" << std::endl;
          });
  // Release resource
  MFXDispReleaseImplDescription(loader, h);
}
```

This code illustrates how application can check that specific functions are implemented:

```
mfxSession session_handle;
loader = mfxLoader();

// We want to search for the implementation with Decode+VPP domain functions support.
// i.e we search for the MFXVideoDECODE_VPP_Init and MFXVideoDECODE_VPP_DecodeFrameAsync
// implemented functions
mfxConfig init_funct_prop = MFXCreateConfig(loader);
mfxVariant value;

// Filter property for the Init function
value.Type = mfxVariantType::MFX_VARIANT_TYPE_PTR;
value.Data.Ptr = (mfxHDL)"MFXVideoDECODE_VPP_Init";
MFXSetConfigFilterProperty(init_funct_prop, (const mfxU8*)"mfxImplementedFunctions.FunctionsName",
                           value);

// Filter property for the Process function
mfxConfig process_func_prop = MFXCreateConfig(loader);
value.Data.Ptr = (mfxHDL)"MFXVideoDECODE_VPP_DecodeFrameAsync";
MFXSetConfigFilterProperty(process_func_prop, (const mfxU8*)"mfxImplementedFunctions.FunctionsName",
                           value);

// create session from first matched implementation
MFXCreateSession(loader, 0, &session_handle);
```

### 3.2.8 How To Search For The Available encoder/decoder implementation

The *CodecFormatFourCC* enum specifies codec's FourCC values. Application needs to assign this value to the field of *mfxDecoderDescription::decoder::CodecID* to search for the decoder or *mfxEncoderDescription::encoder::CodecID* to search for the encoder.

This code illustrates decoder's implementation search procedure:

```
mfxSession hevc_session_handle;
loader = mfxLoader();

// We want to search for the HEVC decoder implementation
mfxConfig hevc_decoder_config = MFXCreateConfig(loader);
mfxVariant value;
```

(continues on next page)

(continued from previous page)

```

// Filter property for the implementations with HEVC decoder
value.Type     = MFX_VARIANT_TYPE_U32;
value.Data.U32 = MFX_CODEC_HEVC;

MFXSetConfigFilterProperty(hevc_decoder_config
      , (const mfxU8*)"mfxImplDescription.mfxDecoderDescription.decoder.CodecID"
      , value);

// create session from first matched implementation
MFXCreateSession(loader, 0, &hevc_session_handle);
```

### 3.2.9 How To Search For The Available VPP Filter implementation

Each VPP filter identified by the filter ID. Filter ID is defined by corresponding to the filter extension buffer ID value which is defined in a form of FourCC value. Filter ID values are subset of the general *ExtendedBufferID* enum. The *table* references available IDs of VPP filters to search. Application needs to assign this value to the field of *mfxVPPDescription::filter::FilterFourCC* to search for the needed VPP filter.

Table 3: VPP Filters ID

| Filter ID | Description |
|---|---|
| *MFX_EXTBUFF_VPP_DENOISE2* | Denoise filter |
| *MFX_EXTBUFF_VPP_MCTF* | Motion-Compensated Temporal Filter (MCTF). |
| *MFX_EXTBUFF_VPP_DETAIL* | Detail/edge enhancement filter. |
| *MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION* | Frame rate conversion filter |
| *MFX_EXTBUFF_VPP_IMAGE_STABILIZATION* | Image stabilization filter |
| *MFX_EXTBUFF_VPP_PROCAMP* | ProcAmp filter |
| *MFX_EXTBUFF_VPP_FIELD_PROCESSING* | Field processing filter |
| *MFX_EXTBUFF_VPP_COLOR_CONVERSION* | Color Conversion filter |
| *MFX_EXTBUFF_VPP_SCALING* | Resize filter |
| *MFX_EXTBUFF_VPP_COMPOSITE* | Surfaces composition filter |
| *MFX_EXTBUFF_VPP_DEINTERLACING* | Deinterlace filter |
| *MFX_EXTBUFF_VPP_ROTATION* | Rotation filter |
| *MFX_EXTBUFF_VPP_MIRRORING* | Mirror filter |
| *MFX_EXTBUFF_VPP_COLORFILL* | ColorFill filter |

This code illustrates VPP mirror filter implementation search procedure:

```
mfxSession mirror_session_handle;
loader = mfxLoader();

// We want to search for the VPP mirror implementation
mfxConfig  mirror_flt_config = MFXCreateConfig(loader);
mfxVariant value;

// Filter property for the implementations with VPP mirror
value.Type     = MFX_VARIANT_TYPE_U32;
value.Data.U32 = MFX_EXTBUFF_VPP_MIRRORING;
```

(continues on next page)

(continued from previous page)

```

MFXSetConfigFilterProperty(mirror_flt_config
      , (const mfxU8*)"mfxImplDescription.mfxVPPDescription.filter.FilterFourCC"
      , value);

// create session from first matched implementation
MFXCreateSession(loader, 0, &mirror_session_handle);
```

### 3.2.10 How To Select Implementation Automatically From Device Handle

Starting from API 2.9 applications may request the dispatcher to load an implementation corresponding to a hardware acceleration device which has already been initialized by the application. The application must initialize a structure of type *mfxAutoSelectImplDeviceHandle* with the appropriate acceleration mode, handle type, and hardware device handle. Then this structure must be passed to the dispatcher by calling *MFXSetConfigFilterProperty()* with property name 'AutoSelectImpl' and property value of type MFX_VARIANT_TYPE_PTR, pointing to the *mfxAutoSelectImplDeviceHandle* structure.

This is currently an experimental API. Backward compatibility and future presence is not guaranteed. Applications should check the error codes returned from *MFXSetConfigFilterProperty()* and *MFXCreateSession()* to check whether the feature is supported and a suitable implementation was found.

This code illustrates automatic implementation selection using an application-provided hardware device handle:

```
mfxLoader loader = MFXLoad();

// In actual code, application should initialize deviceHandle to a
// hardware device handle of the type indicated in the following table.
//
// AccelMode                  DeviceHandleType                DeviceHandle native type
// -------------------------------------------------------------------------------
// MFX_ACCEL_MODE_VIA_D3D9    MFX_HANDLE_D3D9_DEVICE_MANAGER  IDirect3DDeviceManager9*
// MFX_ACCEL_MODE_VIA_D3D11   MFX_HANDLE_D3D11_DEVICE         ID3D11Device*
// MFX_ACCEL_MODE_VIA_VAAPI   MFX_HANDLE_VA_DISPLAY           VADisplay
//
// Example:
// ID3D11Device *pD3D11Device;
// D3D11CreateDevice(... , &pD3D11Device , ...);
// mfxHDL deviceHandle = (mfxHDL)pD3D11Device;

mfxHDL deviceHandle = NULL;

mfxAutoSelectImplDeviceHandle autoSelectStruct;
autoSelectStruct.AutoSelectImplType = MFX_AUTO_SELECT_IMPL_TYPE_DEVICE_HANDLE;
autoSelectStruct.AccelMode          = MFX_ACCEL_MODE_VIA_D3D11;
autoSelectStruct.DeviceHandleType   = MFX_HANDLE_D3D11_DEVICE;
autoSelectStruct.DeviceHandle       = deviceHandle;

mfxConfig cfg1       = MFXCreateConfig(loader);
mfxVariant ImplValue;
ImplValue.Type       = MFX_VARIANT_TYPE_PTR;
ImplValue.Data.Ptr   = &autoSelectStruct;

```

(continues on next page)

(continued from previous page)

```
MFXSetConfigFilterProperty(cfg1, (const mfxU8 *)"AutoSelectImpl", ImplValue);

// Create session with implementation corresponding to deviceHandle.
// It is not required to call MFXVideoCORE_SetHandle() in this case,
// since the implementation already has the necessary deviceHandle.
MFXCreateSession(loader, 0, &session);
```

### 3.2.11 How To Get Path to the Shared Library With the Implementation

Sessions can be created from different implementations, each implementations can be located in different shared libraries. To get path of the shared library with the implementation from which session can be or was created, application can use *MFXEnumImplementations()* and pass *MFX_IMPLCAPS_IMPLPATH* value as the output data request.

This code illustrates collection and print out path of implementations's shared library:

```
mfxHDL h;
mfxSession def_session;

loader = mfxLoader();

// Create session from the first available implementation.
// That's why we no any filters need to be set.
// First available implementation has index equal to the 0.
MFXCreateSession(loader, 0, &def_session);

// Get and print out OS path to the loaded shared library
// with the implementation. It is absolutely OK to call
// MFXEnumImplementations after session creation just need to make
// sure that the same index of implementation is provided to the
// function call.
MFXEnumImplementations(loader, 0, MFX_IMPLCAPS_IMPLPATH, &h);
mfxChar* path = reinterpret_cast<mfxChar*>(h);

// Print out the path
std::cout << "Loaded shared library: " << path << std::endl;

// Release the memory for the string with path.
MFXDispReleaseImplDescription(loader, h);
```

### 3.2.12 oneVPL implementation on Intel® platforms with $X^e$ architecture and Intel® Media Software Development Kit Coexistence

oneVPL supersedes Intel® Media Software Development Kit and is partially binary compatible with Intel® Media Software Development Kit. Both oneVPL and Intel® Media Software Development Kit includes own dispatcher and implementation. Coexistence of oneVPL and Intel® Media Software Development Kit dispatchers and implementations on single system is allowed until Intel® Media Software Development Kit is not EOL.

Usage of the following combinations of dispatchers and implementations within the single application is permitted for the legacy purposes only. In that scenario legacy applications developed with Intel® Media Software Development Kit will continue to work on any HW supported either by Intel® Media Software Development Kit or by the oneVPL.

> **Attention:** Any application to work with the oneVPL API starting from version 2.0 must use only oneVPL dispatcher.

**Intel® Media Software Development Kit API**
Intel® Media Software Development Kit API of 1.x version.

**Removed API**
Intel® Media Software Development Kit *API* which is removed from oneVPL.

**Core API**
Intel® Media Software Development Kit API without removed API.

**oneVPL API**
New *API* introduced in oneVPL only started from API 2.0 version.

**oneVPL Dispatcher API**
Dispatcher *API* introduced in oneVPL in 2.0 API version. This is subset of oneVPL API.

Table 4: oneVPL for Intel® platforms with $X^e$ architecture and Intel® Media Software Development Kit

| Dispatcher | Installed on the device | Loaded | Allowed API |
|---|---|---|---|
| oneVPL | oneVPL for Intel® platforms with $X^e$ architecture | oneVPL for Intel® platforms with $X^e$ architecture | Usage of any API except removed API is allowed. |
| oneVPL | Intel® Media Software Development Kit | Intel® Media Software Development Kit | Usage of core API plus dispatcher API is allowed only. |
| oneVPL | oneVPL for Intel® platforms with $X^e$ architecture and Intel® Media Software Development Kit | oneVPL for Intel® platforms with $X^e$ architecture | Usage of any API except removed API is allowed. |
| Intel® Media Software Development Kit | oneVPL for Intel® platforms with $X^e$ architecture | oneVPL for Intel® platforms with $X^e$ architecture | Usage of core API is allowed only. |
| Intel® Media Software Development Kit | oneVPL for Intel® platforms with $X^e$ architecture and Intel® Media Software Development Kit | Intel® Media Software Development Kit | Usage of Intel® Media Software Development Kit API is allowed. |
| Intel® Media Software Development Kit | Intel® Media Software Development Kit | Intel® Media Software Development Kit | Usage of Intel® Media Software Development Kit API is allowed. |

**Note:** if system has multiple devices the logic of selection and loading implementations will be applied to each device accordingly to the system enumeration.

### 3.2.13 Multiple Sessions

Each oneVPL session can run exactly one instance of the DECODE, ENCODE, and VPP functions. This is adequate for a simple transcoding operation. If the application needs more than one instance of DECODE, ENCODE, or VPP in a complex transcoding setting or needs more simultaneous transcoding operations, the application can initialize multiple oneVPL sessions created from one or several oneVPL implementations.

The application can use multiple oneVPL sessions independently or run a "joined" session. To join two sessions together, the application can use the function *MFXJoinSession()*. Alternatively, the application can use the *MFXCloneSession()* function to duplicate an existing session. Joined oneVPL sessions work together as a single session, sharing all session resources, threading control, and prioritization operations except hardware acceleration devices and external allocators. When joined, the first session (first join) serves as the parent session and will schedule execution resources with all other child sessions. Child sessions rely on the parent session for resource management.

**Important:** Applications can join sessions created from the same oneVPL implementation only.

With joined sessions, the application can set the priority of session operations through the *MFXSetPriority()* function. A lower priority session receives fewer CPU cycles. Session priority does not affect hardware accelerated processing.

After the completion of all session operations, the application can use the *MFXDisjoinSession()* function to remove the joined state of a session. Do not close the parent session until all child sessions are disjoined or closed.

## 3.3 Frame and Fields

In oneVPL terminology, a frame (also referred to as frame surface) contains either a progressive frame or a complementary field pair. If the frame is a complementary field pair, the odd lines of the surface buffer store the top fields and the even lines of the surface buffer store the bottom fields.

### 3.3.1 Frame Surface Management

During encoding, decoding, or video processing, cases arise that require reserving input or output frames for future use. For example, when decoding, a frame that is ready for output must remain as a reference frame until the current sequence pattern ends. The usual method to manage this is to cache the frames internally. This method requires a copy operation, which can significantly reduce performance.

oneVPL has two approaches to avoid the need for copy operations. The legacy approach uses a frame-locking mechanism that works as follows:

1. The application allocates a pool of frame surfaces large enough to include oneVPL function I/O frame surfaces and internal cache needs. Each frame surface maintains a `Locked` counter, which is part of the *mfxFrameData* structure. The `Locked` counter is initially set to zero.
2. The application calls a oneVPL function with frame surfaces from the pool whose `Locked` counter is set as appropriate for the desired operation. For decoding or video processing operations, where oneVPL uses the surfaces to write, the `Locked` counter should be equal to zero. If the oneVPL function needs to reserve any frame surface, the oneVPL function increases the `Locked` counter of the frame surface. A non-zero `Locked` counter indicates that the calling application must treat the frame surface as "in use." When the frame surface is in use, the application can read but cannot alter, move, delete, or free the frame surface.
3. In subsequent oneVPL executions, if the frame surface is no longer in use, oneVPL decreases the `Locked` counter. When the `Locked` counter reaches zero, the application is free to do as it wishes with the frame surface.

In general, the application should not increase or decrease the Locked counter since oneVPL manages this field. If, for some reason, the application needs to modify the Locked counter, the operation must be atomic to avoid a race condition.

oneVPL API version 2.0 introduces the *mfxFrameSurfaceInterface* structure which provides a set of callback functions for the *mfxFrameSurface1* structure to work with frame surfaces. This interface defines *mfxFrameSurface1* as a reference counted object which can be allocated by oneVPL or the application. The application must follow the general rules of operation with reference counted objects. For example, when surfaces are allocated by oneVPL during *MFXVideoDECODE_DecodeFrameAsync()* or with the help of *MFXMemory_GetSurfaceForVPP()* or *MFXMemory_GetSurfaceForVPPOut()* or *MFXMemory_GetSurfaceForEncode()*, the application must call the corresponding *mfxFrameSurfaceInterface::Release* function for the surfaces that are no longer in use.

> **Attention:** Note that the Locked counter defines read/write access policies and the reference counter is responsible for managing a frame's lifetime.

The second approach to avoid the need for copy operations is based on the *mfxFrameSurfaceInterface* and works as follows:

1. oneVPL or the application allocates a frame surface and the application stores a value of reference counter obtained through *mfxFrameSurfaceInterface::GetRefCounter*.
2. The application calls a oneVPL function with the frame surface. If oneVPL needs to reserve the frame surface it increments the reference counter through the *mfxFrameSurfaceInterface::AddRef* call. When the frame surface is no longer in use by the oneVPL it decrements reference counter through the *mfxFrameSurfaceInterface::Release* call which returns the reference counter to the original value.
3. The application checks the reference counter of the frame surface and when it is equal to the original value after allocation, it can reuse the reference counter for subsequent operations.

**Note:** All *mfxFrameSurface1* structures starting from mfxFrameSurface1::mfxStructVersion = {1,1} support the *mfxFrameSurfaceInterface*.

## 3.4 Decoding Procedures

There are several approaches to decode video frames. The first one is based on the internal allocation mechanism presented here:

```
MFXVideoDECODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
      append_more_bitstream(bitstream);
   bits=(end_of_stream())?NULL:bitstream;
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,NULL,&disp,&syncp);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_
↪SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
      disp->FrameInterface->Release(disp);
```

(continues on next page)

(continued from previous page)

```
    }
}
MFXVideoDECODE_Close(session);
```

Note the following key points about the example:

- The application calls the *MFXVideoDECODE_DecodeFrameAsync()* function for a decoding operation with the bitstream buffer (bits), frame surface is allocated internally by the library.

> **Attention:** As shown in the example above starting with API version 2.0, the application can provide NULL as the working frame surface that leads to internal memory allocation.

- If decoding output is not available, the function returns a status code requesting additional bitstream input as follows:
  - *mfxStatus::MFX_ERR_MORE_DATA*: The function needs additional bitstream input. The existing buffer contains less than a frame's worth of bitstream data.
- Upon successful decoding, the *MFXVideoDECODE_DecodeFrameAsync()* function returns *mfxStatus::MFX_ERR_NONE*. However, the decoded frame data (identified by the surface_out pointer) is not yet available because the *MFXVideoDECODE_DecodeFrameAsync()* function is asynchronous. The application must use the *MFXVideoCORE_SyncOperation()* or *mfxFrameSurfaceInterface::Synchronize* to synchronize the decoding operation before retrieving the decoded frame data.
- At the end of the bitstream, the application continuously calls the *MFXVideoDECODE_DecodeFrameAsync()* function with a NULL bitstream pointer to drain any remaining frames cached within the decoder until the function returns *mfxStatus::MFX_ERR_MORE_DATA*.
- When application completes the work with frame surface, it must call release to avoid memory leaks.

The next example demonstrates how applications can use internally pre-allocated chunk of video surfaces:

```
MFXVideoDECODE_QueryIOSurf(session, &init_param, &request);
MFXVideoDECODE_Init(session, &init_param);
for (int i = 0; i < request.NumFrameSuggested; i++) {
    MFXMemory_GetSurfaceForDecode(session, &work);
    add_surface_to_pool(work);
}
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
      append_more_bitstream(bitstream);
   bits=(end_of_stream())?NULL:bitstream;
   // application logic to distinguish free and busy surfaces
   find_free_surface_from_the_pool(&work);
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,work,&disp,&syncp);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_
↪SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
      disp->FrameInterface->Release(disp);
   }
```

(continues on next page)

(continued from previous page)

```
}
for (int i = 0; i < request.NumFrameSuggested; i++) {
    get_next_surface_from_pool(&work);
    work->FrameInterface->Release(work);
}
MFXVideoDECODE_Close(session);
```

Here the application should use the *MFXVideoDECODE_QueryIOSurf()* function to obtain the number of working frame surfaces required to reorder output frames. It is also required that *MFXMemory_GetSurfaceForDecode()* call is done after decoder initialization. In the *MFXVideoDECODE_DecodeFrameAsync()* the oneVPL library increments reference counter of incoming surface frame so it is required that the application releases frame surface after the call.

Another approach to decode frames is to allocate video frames on-fly with help of *MFXMemory_GetSurfaceForDecode()* function, feed the library and release working surface after *MFXVideoDECODE_DecodeFrameAsync()* call.

> **Attention:** Please pay attention on two release calls for surfaces: after *MFXVideoDECODE_DecodeFrameAsync()* to decrease reference counter of working surface returned by *MFXMemory_GetSurfaceForDecode()*. After *MFXVideoCORE_SyncOperation()* to decrease reference counter of output surface returned by *MFXVideoDECODE_DecodeFrameAsync()*.

```
MFXVideoDECODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
      append_more_bitstream(bitstream);
   bits=(end_of_stream())?NULL:bitstream;
   MFXMemory_GetSurfaceForDecode(session, &work);
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,work,&disp,&syncp);
   work->FrameInterface->Release(work);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_
↪SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
      disp->FrameInterface->Release(disp);
   }
}
MFXVideoDECODE_Close(session);
```

The following pseudo code shows the decoding procedure according to the legacy mode with external video frames allocation:

```
MFXVideoDECODE_DecodeHeader(session, bitstream, &init_param);
MFXVideoDECODE_QueryIOSurf(session, &init_param, &request);
allocate_pool_of_frame_surfaces(request.NumFrameSuggested);
MFXVideoDECODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
```

(continues on next page)

(continued from previous page)

```
      append_more_bitstream(bitstream);
   find_free_surface_from_the_pool(&work);
   bits=(end_of_stream())?NULL:bitstream;
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,work,&disp,&syncp);
   if (sts==MFX_ERR_MORE_SURFACE) continue;
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   if (sts==MFX_ERR_REALLOC_SURFACE) {
      MFXVideoDECODE_GetVideoParam(session, &param);
      realloc_surface(work, param.mfx.FrameInfo);
      continue;
   }
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
   }
}
MFXVideoDECODE_Close(session);
free_pool_of_frame_surfaces();
```

Note the following key points about the example:

- The application can use the *MFXVideoDECODE_DecodeHeader()* function to retrieve decoding initialization parameters from the bitstream. This step is optional if the data is retrievable from other sources such as an audio/video splitter.
- The *MFXVideoDECODE_DecodeFrameAsync()* function can return following status codes in addition to the described above:
  - *mfxStatus::MFX_ERR_MORE_SURFACE*: The function needs one more frame surface to produce any output.
  - *mfxStatus::MFX_ERR_REALLOC_SURFACE*: Dynamic resolution change case - the function needs a bigger working frame surface (work).

The following pseudo code shows the simplified decoding procedure:

```
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
      append_more_bitstream(bitstream);
   bits=(end_of_stream())?NULL:bitstream;
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,NULL,&disp,&syncp);
   if (sts==MFX_ERR_MORE_SURFACE) continue;
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
      disp->FrameInterface->Release(disp);
   }
}
```

oneVPL API version 2.0 introduces a new decoding approach. For simple use cases, when the user wants to decode a stream and does not want to set additional parameters, a simplified procedure for the decoder's initialization has been proposed. In this scenario it is possible to skip explicit stages of a stream's header decoding and the decoder's initialization and instead to perform these steps implicitly during decoding of the first frame. This change also requires setting the additional field `mfxBitstream::CodecId` to indicate codec type. In this mode the decoder allocates `mfxFrameSurface1` internally, so users should set the input surface to zero.

### 3.4.1 Bitstream Repositioning

The application can use the following procedure for bitstream reposition during decoding:

1. Use the `MFXVideoDECODE_Reset()` function to reset the oneVPL decoder.
2. Optional: If the application maintains a sequence header that correctly decodes the bitstream at the new position, the application may insert the sequence header to the bitstream buffer.
3. Append the bitstream from the new location to the bitstream buffer.
4. Resume the decoding procedure. If the sequence header is not inserted in the previous steps, the oneVPL decoder searches for a new sequence header before starting decoding.

### 3.4.2 Broken Streams Handling

Robustness and the capability to handle a broken input stream is an important part of the decoder.

First, the start code prefix (ITU-T* H.264 3.148 and ITU-T H.265 3.142) is used to separate NAL units. Then all syntax elements in the bitstream are parsed and verified. If any of the elements violate the specification, the input bitstream is considered invalid and the decoder tries to re-sync (find the next start code). Subsequent decoder behavior is dependent on which syntax element is broken:

- SPS header is broken: return `mfxStatus::MFX_ERR_INCOMPATIBLE_VIDEO_PARAM` (HEVC decoder only, AVC decoder uses last valid).
- PPS header is broken: re-sync, use last valid PPS for decoding.
- Slice header is broken: skip this slice, re-sync.
- Slice data is broken: corruption flags are set on output surface.

Many streams have IDR frames with `frame_num != 0` while the specification says that "If the current picture is an IDR picture, frame_num shall be equal to 0" (ITU-T H.265 7.4.3).

VUI is also validated, but errors do not invalidate the whole SPS. The decoder either does not use the corrupted VUI (AVC) or resets incorrect values to default (HEVC).

**Note:** Some requirements are relaxed because there are many streams which violate the strict standard but can be decoded without errors.

Corruption at the reference frame is spread over all inter-coded pictures that use the reference frame for prediction. To cope with this problem you must either periodically insert I-frames (intra-coded) or use the intra-refresh technique. The intra-refresh technique allows recovery from corruptions within a predefined time interval. The main point of intra-refresh is to insert a cyclic intra-coded pattern (usually a row) of macroblocks into the inter-coded pictures, restricting motion vectors accordingly. Intra-refresh is often used in combination with recovery point SEI, where the `recovery_frame_cnt` is derived from the intra-refresh interval. The recovery point SEI message is well described at ITU-T H.264 D.2.7 and ITU-T H.265 D.2.8. If decoding starts from AU associated with this SEI message, then the message can be used by the decoder to determine from which picture all subsequent pictures have no errors. In comparison to IDR, the recovery point message does not mark reference pictures as "unused for reference".

Besides validation of syntax elements and their constraints, the decoder also uses various hints to handle broken streams:

- If there are no valid slices for the current frame, then the whole frame is skipped.
- The slices which violate slice segment header semantics (ITU-T H.265 7.4.7.1) are skipped. Only the `slice_temporal_mvp_enabled_flag` is checked for now.
- Since LTR (Long Term Reference) stays at DPB until it is explicitly cleared by IDR or MMCO, the incorrect LTR could cause long standing visual artifacts. AVC decoder uses the following approaches to handle this:
  - When there is a DPB overflow in the case of an incorrect MMCO command that marks the reference picture as LT, the operation is rolled back.
  - An IDR frame with `frame_num != 0` can't be LTR.
- If the decoder detects frame gapping, it inserts "fake"" (marked as non-existing) frames, updates FrameNumWrap (ITU-T H.264 8.2.4.1) for reference frames, and applies the Sliding Window (ITU-T H.264 8.2.5.3) marking process. Fake frames are marked as reference, but since they are marked as non-existing, they are not used for inter-prediction.

### 3.4.3 VP8 Specific Details

Unlike other oneVPL supported decoders, VP8 can accept only a complete frame as input. The application should provide the complete frame accompanied by the *MFX_BITSTREAM_COMPLETE_FRAME* flag. This is the single specific difference.

### 3.4.4 JPEG

The application can use the same decoding procedures for JPEG/motion JPEG decoding, as shown in the following pseudo code:

```
// optional; retrieve initialization parameters
MFXVideoDECODE_DecodeHeader(...);
// decoder initialization
MFXVideoDECODE_Init(...);
// single frame/picture decoding
MFXVideoDECODE_DecodeFrameAsync(...);
MFXVideoCORE_SyncOperation(...);
// optional; retrieve meta-data
MFXVideoDECODE_GetUserData(...);
// close
MFXVideoDECODE_Close(...);
```

The *MFXVideoDECODE_Query()* function will return *mfxStatus::MFX_ERR_UNSUPPORTED* if the input bitstream contains unsupported features.

For still picture JPEG decoding, the input can be any JPEG bitstreams that conform to the ITU-T Recommendation T.81 with an EXIF or JFIF header. For motion JPEG decoding, the input can be any JPEG bitstreams that conform to the ITU-T Recommendation T.81.

Unlike other oneVPL decoders, JPEG decoding supports three different output color formats: *NV12*, *YUY2*, and *RGB32*. This support sometimes requires internal color conversion and more complicated initialization. The color format of the input bitstream is described by the *mfxInfoMFX::JPEGChromaFormat* and *mfxInfoMFX::JPEGColorFormat* fields. The *MFXVideoDECODE_DecodeHeader()* function usually fills them in. If the JPEG bitstream does not contains color format information, the application should provide it. Output color format is described by general oneVPL parameters: the *mfxFrameInfo::FourCC* and *mfxFrameInfo::ChromaFormat* fields.

Motion JPEG supports interlaced content by compressing each field (a half-height frame) individually. This behavior is incompatible with the rest of the oneVPL transcoding pipeline, where oneVPL requires fields to be in odd and even lines of the same frame surface. The decoding procedure is modified as follows:

- The application calls the *MFXVideoDECODE_DecodeHeader()* function with the first field JPEG bitstream to retrieve initialization parameters.
- The application initializes the oneVPL JPEG decoder with the following settings:
  - The `PicStruct` field of the *mfxVideoParam* structure set to the correct interlaced type, *MFX_PICSTRUCT_FIELD_TFF* or *MFX_PICSTRUCT_FIELD_BFF*, from the motion JPEG header.
  - Double the `Height` field in the *mfxVideoParam* structure as the value returned by the *MFXVideoDECODE_DecodeHeader()* function describes only the first field. The actual frame surface should contain both fields.
- During decoding, the application sends both fields for decoding in the same *mfxBitstream*. The application should also set *mfxBitstream::DataFlag* to *MFX_BITSTREAM_COMPLETE_FRAME*. oneVPL decodes both fields and combines them into odd and even lines according to oneVPL convention.

By default, the *MFXVideoDECODE_DecodeHeader()* function returns the `Rotation` parameter so that after rotation, the pixel at the first row and first column is at the top left. The application can overwrite the default rotation before calling *MFXVideoDECODE_Init()*.

The application may specify Huffman and quantization tables during decoder initialization by attaching *mfxExtJPEGQuantTables* and *mfxExtJPEGHuffmanTables* buffers to the *mfxVideoParam* structure. In this case, the decoder ignores tables from bitstream and uses the tables specified by the application. The application can also retrieve these tables by attaching the same buffers to *mfxVideoParam* and calling *MFXVideoDECODE_GetVideoParam()* or *MFXVideoDECODE_DecodeHeader()* functions.

### 3.4.5 Multi-view Video Decoding

The oneVPL MVC decoder operates on complete MVC streams that contain all view and temporal configurations. The application can configure the oneVPL decoder to generate a subset at the decoding output. To do this, the application must understand the stream structure and use the stream information to configure the decoder for target views.

The decoder initialization procedure is as follows:

1. The application calls the *MFXVideoDECODE_DecodeHeader()* function to obtain the stream structural information. This is done in two steps:
   1. The application calls the *MFXVideoDECODE_DecodeHeader()* function with the *mfxExtMVCSeqDesc* structure attached to the *mfxVideoParam* structure. At this point, do not allocate memory for the arrays in the *mfxExtMVCSeqDesc* structure. Set the `View`, `ViewId`, and `OP` pointers to NULL and set `NumViewAlloc`, `NumViewIdAlloc`, and `NumOPAlloc` to zero. The function parses the bitstream and returns *mfxStatus::MFX_ERR_NOT_ENOUGH_BUFFER* with the correct values for `NumView`, `NumViewId`, and `NumOP`. This step can be skipped if the application is able to obtain the `NumView`, `NumViewId`, and `NumOP` values from other sources.
   2. The application allocates memory for the `View`, `ViewId`, and `OP` arrays and calls the *MFXVideoDECODE_DecodeHeader()* function again. The function returns the MVC structural information in the allocated arrays.
2. The application fills the *mfxExtMVCTargetViews* structure to choose the target views, based on information described in the *mfxExtMVCSeqDesc* structure.
3. The application initializes the oneVPL decoder using the *MFXVideoDECODE_Init()* function. The application must attach both the *mfxExtMVCSeqDesc* structure and the *mfxExtMVCTargetViews* structure to the *mfxVideoParam* structure.

In the above steps, do not modify the values of the *mfxExtMVCSeqDesc* structure after the *MFXVideoDECODE_DecodeHeader()* function, as the oneVPL decoder uses the values in the structure for internal memory allocation. Once the application configures the oneVPL decoder, the rest of the decoding procedure remains unchanged. As shown in the pseudo code below, the application calls the *MFXVideoDECODE_DecodeFrameAsync()* function multiple times to obtain all target views of the current frame picture, one target view at a time. The target view is identified by the `FrameID` field of the *mfxFrameInfo* structure.

```
mfxExtBuffer *eb[2];
mfxExtMVCSeqDesc  seq_desc;
mfxVideoParam init_param;

init_param.ExtParam=(mfxExtBuffer **)&eb;
init_param.NumExtParam=1;
eb[0]=(mfxExtBuffer *)&seq_desc;
MFXVideoDECODE_DecodeHeader(session, bitstream, &init_param);

/* select views to decode */
mfxExtMVCTargetViews tv;
init_param.NumExtParam=2;
eb[1]=(mfxExtBuffer *)&tv;

/* initialize decoder */
MFXVideoDECODE_Init(session, &init_param);

/* perform decoding */
for (;;) {
    MFXVideoDECODE_DecodeFrameAsync(session, bits, work, &disp, &syncp);
    disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_SyncOperation(session,syncp,INFINITE)
}

/* close decoder */
MFXVideoDECODE_Close(session);
```

### 3.4.6 Combined Decode with Multi-channel Video Processing

The oneVPL exposes interface for making decode and video processing operations in one call. Users can specify a number of output processing channels and multiple video filters per each channel. This interface supports only internal memory allocation model and returns array of processed frames through *mfxSurfaceArray* reference object as shown by the example:

```
num_channel_par = 2;
// first video processing channel with resize
vpp_par_array[0]->VPP.Width = 400;
vpp_par_array[0]->VPP.Height = 400;

// second video channel with color conversion filter
vpp_par_array[1]->VPP.FourCC = MFX_FOURCC_UYVY;

sts = MFXVideoDECODE_VPP_Init(session, decode_par, vpp_par_array, num_channel_par);

sts = MFXVideoDECODE_VPP_DecodeFrameAsync(session, bitstream, NULL, 0, &surf_array_out);
```

(continues on next page)

(continued from previous page)

```

//surf_array_out layout is
do_smth(surf_array_out->Surfaces[0]); //The first channel contains decoded frames.
do_smth(surf_array_out->Surfaces[1]); //The second channel contains resized frames after decode.
do_smth(surf_array_out->Surfaces[2]); //The third channel contains color converted frames after decode.
```

It's possible that different video processing channels may have different latency:

```
//1st call
sts = MFXVideoDECODE_VPP_DecodeFrameAsync(session, bitstream, NULL, 0, &surf_array_out);
//surf_array_out layout is
do_smth(surf_array_out->Surfaces[0]); //decoded frame
do_smth(surf_array_out->Surfaces[1]); //resized frame (ChannelId = 1). The first frame from channel with resize available
// no output from channel with ADI output since it has one frame delay

//2nd call
sts = MFXVideoDECODE_VPP_DecodeFrameAsync(session, bitstream, NULL, 0, &surf_array_out);
//surf_array_out layout is
do_smth(surf_array_out->Surfaces[0]); //decoded frame
do_smth(surf_array_out->Surfaces[1]); //resized frame (ChannelId = 1)
do_smth(surf_array_out->Surfaces[2]); //ADI output (ChannelId = 2). The first frame from ADI channel
```

Application can match decoded frame w/ specific VPP channels using *mfxFrameData::TimeStamp*, :cpp:member:mfxFrameData::FrameOrder` and *mfxFrameInfo::ChannelId*.

Application can skip some or all channels including decoding output with help of *skip_channels* and *num_skip_channels* parameters as follows: application fills *skip_channels* array with *ChannelId`s to disable output of correspondent channels. In that case :cpp:member:`surf_array_out* would contain only surfaces for the remaining channels. If the decoder's channel and/or impacted VPP channels don't have output frame(s) for the current call (for instance, input bitstream doesn't contain complete frame or deinterlacing/FRC filter have delay) *skip_channels* parameter is ignored for this channel.

If application disables all channels the SDK returns NULL as *mfxSurfaceArray*.

If application doesn't need to disable any channels it sets *num_skip_channels* to zero, *skip_channels* is ignored when *num_skip_channels* is zero.

If application doesn't need to make scaling or cropping operations it has to set the following fields *mfxFrameInfo::Width*, *mfxFrameInfo::Height*, *mfxFrameInfo::CropX*, *mfxFrameInfo::CropY* *mfxFrameInfo::CropW*, *mfxFrameInfo::CropH* of the VPP channel to zero. In that case output surfaces have the original decoded resolution and cropping. The operation supports bitstreams with resolution change without need of *MFXVideoDECODE_VPP_Reset()* call.

**Note:** Even if more than one input compressed frame is consumed, the *MFXVideoDECODE_VPP_DecodeFrameAsync()* produces only one decoded frame and correspondent frames from VPP channels.

## 3.5 Encoding Procedures

There are two methods for shared memory allocation and handling in oneVPL: external and internal.

### 3.5.1 External Memory

The following pseudo code shows the encoding procedure with external memory (legacy mode):

```
MFXVideoENCODE_QueryIOSurf(session, &init_param, &request);
allocate_pool_of_frame_surfaces(request.NumFrameSuggested);
MFXVideoENCODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream()) {
      find_unlocked_surface_from_the_pool(&surface);
      fill_content_for_encoding(surface);
   }
   surface2=end_of_stream()?NULL:surface;
   sts=MFXVideoENCODE_EncodeFrameAsync(session,NULL,surface2,bits,&syncp);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // Skipped other error handling
   if (sts==MFX_ERR_NONE) {
      MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
      do_something_with_encoded_bits(bits);
   }
}
MFXVideoENCODE_Close(session);
free_pool_of_frame_surfaces();
```

Note the following key points about the example:

- The application uses the *MFXVideoENCODE_QueryIOSurf()* function to obtain the number of working frame surfaces required for reordering input frames.
- The application calls the *MFXVideoENCODE_EncodeFrameAsync()* function for the encoding operation. The input frame must be in an unlocked frame surface from the frame surface pool. If the encoding output is not available, the function returns the *mfxStatus::MFX_ERR_MORE_DATA* status code to request additional input frames.
- Upon successful encoding, the *MFXVideoENCODE_EncodeFrameAsync()* function returns *mfxStatus::MFX_ERR_NONE*. At this point, the encoded bitstream is not yet available because the *MFXVideoENCODE_EncodeFrameAsync()* function is asynchronous. The application must use the *MFXVideoCORE_SyncOperation()* function to synchronize the encoding operation before retrieving the encoded bitstream.
- At the end of the stream, the application continuously calls the *MFXVideoENCODE_EncodeFrameAsync()* function with a NULL surface pointer to drain any remaining bitstreams cached within the oneVPL encoder, until the function returns *mfxStatus::MFX_ERR_MORE_DATA*.

**Note:** It is the application's responsibility to fill pixels outside of the crop window when it is smaller than the frame to be encoded, especially in cases when crops are not aligned to minimum coding block size (16 for AVC and 8 for HEVC and VP9).

### 3.5.2 Internal Memory

The following pseudo code shows the encoding procedure with internal memory:

```
MFXVideoENCODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream()) {
      MFXMemory_GetSurfaceForEncode(session,&surface);
      fill_content_for_encoding(surface);
   }
   surface2=end_of_stream()?NULL:surface;
   sts=MFXVideoENCODE_EncodeFrameAsync(session,NULL,surface2,bits,&syncp);
   if (surface2) surface->FrameInterface->Release(surface2);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // Skipped other error handling
   if (sts==MFX_ERR_NONE) {
      MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
      do_something_with_encoded_bits(bits);
   }
}
MFXVideoENCODE_Close(session);
```

There are several key differences in this example, compared to external memory (legacy mode):

- The application does not need to call the *MFXVideoENCODE_QueryIOSurf()* function to obtain the number of working frame surfaces since allocation is done by oneVPL.
- The application calls the *MFXMemory_GetSurfaceForEncode()* function to get a free surface for the subsequent encode operation.
- The application must call the *mfxFrameSurfaceInterface::Release* function to decrement the reference counter of the obtained surface after the call to the *MFXVideoENCODE_EncodeFrameAsync()* function.

### 3.5.3 Configuration Change

The application changes configuration during encoding by calling the *MFXVideoENCODE_Reset()* function. Depending on the difference in configuration parameters before and after the change, the oneVPL encoder will either continue the current sequence or start a new one. If the encoder starts a new sequence, it completely resets internal state and begins a new sequence with the IDR frame.

The application controls encoder behavior during parameter change by attaching the *mfxExtEncoderResetOption* structure to the *mfxVideoParam* structure during reset. By using this structure, the application instructs the encoder to start or not start a new sequence after reset. In some cases, the request to continue the current sequence cannot be satisfied and the encoder will fail during reset. To avoid this scenario, the application may query the reset outcome before the actual reset by calling the *MFXVideoENCODE_Query()* function with the *mfxExtEncoderResetOption* attached to the *mfxVideoParam* structure.

The application uses the following procedure to change encoding configurations:

1. The application retrieves any cached frames in the oneVPL encoder by calling the *MFXVideoENCODE_EncodeFrameAsync()* function with a NULL input frame pointer until the function returns *mfxStatus::MFX_ERR_MORE_DATA*.
2. The application calls the *MFXVideoENCODE_Reset()* function with the new configuration:
   - If the function successfully sets the configuration, the application can continue encoding as usual.

- If the new configuration requires a new memory allocation, the function returns *mfxStatus::MFX_ERR_INCOMPATIBLE_VIDEO_PARAM*. The application must close the oneVPL encoder and reinitialize the encoding procedure with the new configuration.

### 3.5.4 External Bitrate Control

The application can make the encoder use the external Bitrate Control (BRC) instead of the native bitrate control. To make the encoder use the external BRC, the application should attach the *mfxExtCodingOption2* structure with `ExtBRC = MFX_CODINGOPTION_ON` and the *mfxExtBRC* callback structure to the *mfxVideoParam* structure during encoder initialization. The **Init**, **Reset**, and **Close** callbacks will be invoked inside their corresponding functions: *MFXVideoENCODE_Init()*, *MFXVideoENCODE_Reset()*, and *MFXVideoENCODE_Close()*. The following figure shows asynchronous encoding flow with external BRC (using `GetFrameCtrl` and `Update`):

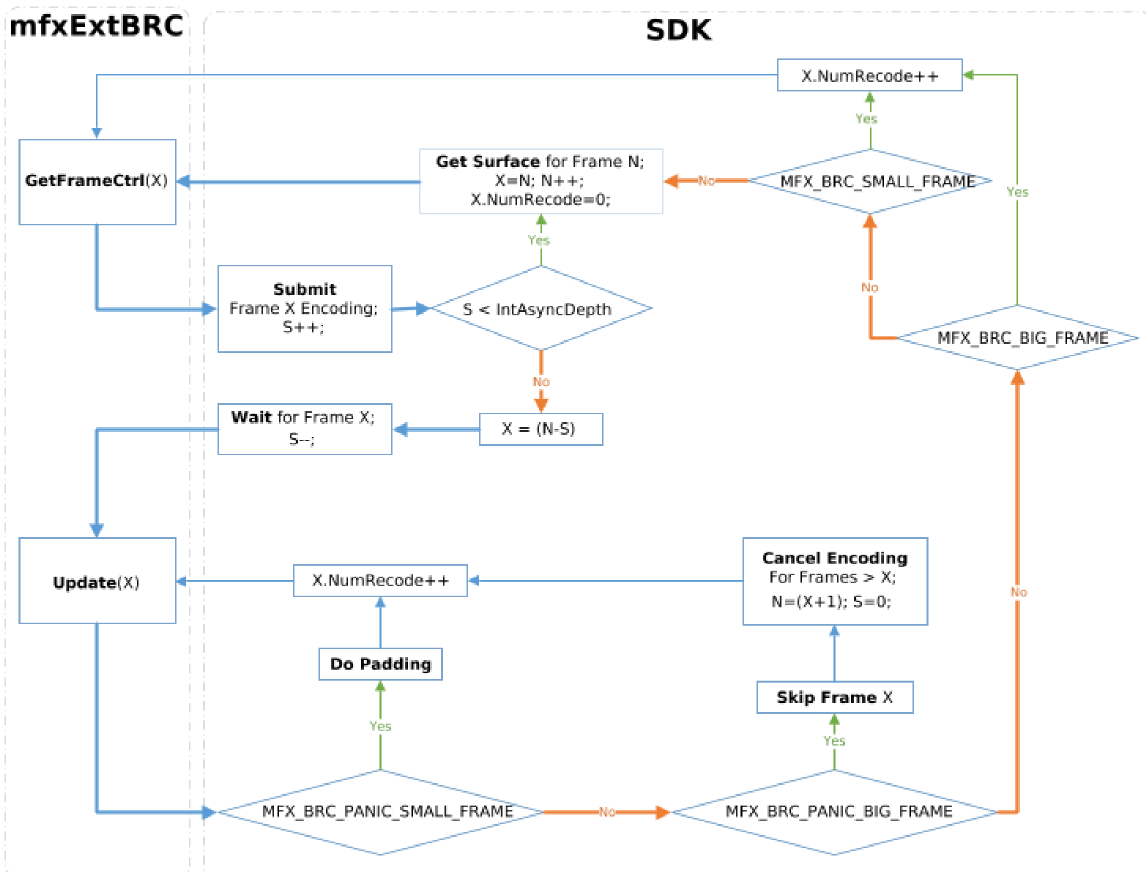



Fig. 1: Asynchronous encoding flow with external BRC

---

**Note:** `IntAsyncDepth` is the oneVPL max internal asynchronous encoding queue size. It is always less than or equal to *mfxVideoParam::AsyncDepth*.

---

The following pseudo code shows use of the external BRC:

```
#include "mfxvideo.h"
#include "mfxbrc.h"

typedef struct {
   mfxU32 EncodedOrder;
   mfxI32 QP;
   mfxU32 MaxSize;
   mfxU32 MinSize;
   mfxU16 Status;
   mfxU64 StartTime;
   // ... skipped
} MyBrcFrame;

typedef struct {
   MyBrcFrame* frame_queue;
   mfxU32 frame_queue_size;
   mfxU32 frame_queue_max_size;
   mfxI32 max_qp[3]; //I,P,B
   mfxI32 min_qp[3]; //I,P,B
   // ... skipped
} MyBrcContext;

void* GetExtBuffer(mfxExtBuffer** ExtParam, mfxU16 NumExtParam, mfxU32 bufferID)
{
    int i=0;
    for(i = 0; i < NumExtParam; i++) {
        if(ExtParam[i]->BufferId == bufferID) return ExtParam[i];
    }
    return NULL;
}

static int IsParametersSupported(mfxVideoParam *par)
{
    UNUSED_PARAM(par);
    // do some checks
    return 1;
}

static int IsResetPossible(MyBrcContext* ctx, mfxVideoParam *par)
{
    UNUSED_PARAM(ctx);
    UNUSED_PARAM(par);
    // do some checks
    return 1;
}

static MyBrcFrame* GetFrame(MyBrcFrame *frame_queue, mfxU32 frame_queue_size, mfxU32 EncodedOrder)
{
    UNUSED_PARAM(EncodedOrder);
    //do some logic
    if(frame_queue_size) return &frame_queue[0];
```

(continues on next page)

(continued from previous page)

```
        return NULL;
    }

    static mfxU32 GetFrameCost(mfxU16 FrameType, mfxU16 PyramidLayer)
    {
        UNUSED_PARAM(FrameType);
        UNUSED_PARAM(PyramidLayer);
        // calculate cost
        return 1;
    }

    static mfxU32 GetMinSize(MyBrcContext *ctx, mfxU32 cost)
    {
        UNUSED_PARAM(ctx);
        UNUSED_PARAM(cost);
        // do some logic
        return 1;
    }

    static mfxU32 GetMaxSize(MyBrcContext *ctx, mfxU32 cost)
    {
        UNUSED_PARAM(ctx);
        UNUSED_PARAM(cost);
        // do some logic
        return 1;
    }

    static mfxI32 GetInitQP(MyBrcContext *ctx, mfxU32 MinSize, mfxU32 MaxSize, mfxU32 cost)
    {
        UNUSED_PARAM(ctx);
        UNUSED_PARAM(MinSize);
        UNUSED_PARAM(MaxSize);
        UNUSED_PARAM(cost);
        // do some logic
        return 1;
    }

    static mfxU64 GetTime()
    {
        mfxU64 wallClock = 0xFFFF;
        return wallClock;
    }

    static void UpdateBRCState(mfxU32 CodedFrameSize, MyBrcContext *ctx)
    {
        UNUSED_PARAM(CodedFrameSize);
        UNUSED_PARAM(ctx);
        return;
    }

    static void RemoveFromQueue(MyBrcFrame* frame_queue, mfxU32 frame_queue_size,
```

(continues on next page)

(continued from previous page)

```
MyBrcFrame* frame)
{
    UNUSED_PARAM(frame_queue);
    UNUSED_PARAM(frame_queue_size);
    UNUSED_PARAM(frame);
    return;
}

static mfxU64 GetMaxFrameEncodingTime(MyBrcContext *ctx)
{
    UNUSED_PARAM(ctx);
    return 2;
}

mfxStatus MyBrcInit(mfxHDL pthis, mfxVideoParam* par) {
   MyBrcContext* ctx = (MyBrcContext*)pthis;
   mfxI32 QpBdOffset;
   mfxExtCodingOption2* co2;
   mfxI32 defaultQP = 4;

   if (!pthis || !par)
      return MFX_ERR_NULL_PTR;

   if (!IsParametersSupported(par))
      return MFX_ERR_UNSUPPORTED;

   ctx->frame_queue_max_size = par->AsyncDepth;
   ctx->frame_queue = (MyBrcFrame*)malloc(sizeof(MyBrcFrame) * ctx->frame_queue_max_size);

   if (!ctx->frame_queue)
      return MFX_ERR_MEMORY_ALLOC;

   co2 = (mfxExtCodingOption2*)GetExtBuffer(par->ExtParam, par->NumExtParam, MFX_EXTBUFF_CODING_OPTION2);
   QpBdOffset = (par->mfx.FrameInfo.BitDepthLuma > 8) ? (6 * (par->mfx.FrameInfo.BitDepthLuma - 8)) : 0;

   ctx->max_qp[0] = (co2 && co2->MaxQPI) ? (co2->MaxQPI - QpBdOffset) : defaultQP;
   ctx->min_qp[0] = (co2 && co2->MinQPI) ? (co2->MinQPI - QpBdOffset) : defaultQP;

   ctx->max_qp[1] = (co2 && co2->MaxQPP) ? (co2->MaxQPP - QpBdOffset) : defaultQP;
   ctx->min_qp[1] = (co2 && co2->MinQPP) ? (co2->MinQPP - QpBdOffset) : defaultQP;

   ctx->max_qp[2] = (co2 && co2->MaxQPB) ? (co2->MaxQPB - QpBdOffset) : defaultQP;
   ctx->min_qp[2] = (co2 && co2->MinQPB) ? (co2->MinQPB - QpBdOffset) : defaultQP;

   // skipped initialization of other other BRC parameters

   ctx->frame_queue_size = 0;

   return MFX_ERR_NONE;
```

(continues on next page)

(continued from previous page)

```
    }

    mfxStatus MyBrcReset(mfxHDL pthis, mfxVideoParam* par) {
       MyBrcContext* ctx = (MyBrcContext*)pthis;

       if (!pthis || !par)
          return MFX_ERR_NULL_PTR;

       if (!IsParametersSupported(par))
          return MFX_ERR_UNSUPPORTED;

       if (!IsResetPossible(ctx, par))
          return MFX_ERR_INCOMPATIBLE_VIDEO_PARAM;

       // reset here BRC parameters if required

       return MFX_ERR_NONE;
    }

    mfxStatus MyBrcClose(mfxHDL pthis) {
       MyBrcContext* ctx = (MyBrcContext*)pthis;

       if (!pthis)
          return MFX_ERR_NULL_PTR;

       if (ctx->frame_queue) {
          free(ctx->frame_queue);
          ctx->frame_queue = NULL;
          ctx->frame_queue_max_size = 0;
          ctx->frame_queue_size = 0;
       }

       return MFX_ERR_NONE;
    }

    mfxStatus MyBrcGetFrameCtrl(mfxHDL pthis, mfxBRCFrameParam* par, mfxBRCFrameCtrl* ctrl) {
       MyBrcContext* ctx = (MyBrcContext*)pthis;
       MyBrcFrame* frame = NULL;
       mfxU32 cost;

       if (!pthis || !par || !ctrl)
          return MFX_ERR_NULL_PTR;

       if (par->NumRecode > 0)
          frame = GetFrame(ctx->frame_queue, ctx->frame_queue_size, par->EncodedOrder);
       else if (ctx->frame_queue_size < ctx->frame_queue_max_size)
          frame = &ctx->frame_queue[ctx->frame_queue_size++];

       if (!frame)
          return MFX_ERR_UNDEFINED_BEHAVIOR;

```

(continues on next page)

(continued from previous page)

```
    if (par->NumRecode == 0) {
        frame->EncodedOrder = par->EncodedOrder;
        cost = GetFrameCost(par->FrameType, par->PyramidLayer);
        frame->MinSize = GetMinSize(ctx, cost);
        frame->MaxSize = GetMaxSize(ctx, cost);
        frame->QP = GetInitQP(ctx, frame->MinSize, frame->MaxSize, cost); // from QP/size stat
        frame->StartTime = GetTime();
    }

    ctrl->QpY = frame->QP;

    return MFX_ERR_NONE;
}

#define DEFAULT_QP_INC 4
#define DEFAULT_QP_DEC 4

mfxStatus MyBrcUpdate(mfxHDL pthis, mfxBRCFrameParam* par, mfxBRCFrameCtrl* ctrl, mfxBRCFrameStatus* status) {
    MyBrcContext* ctx = (MyBrcContext*)pthis;
    MyBrcFrame* frame = NULL;
    mfxU32 panic = 0;

    if (!pthis || !par || !ctrl || !status)
        return MFX_ERR_NULL_PTR;

    frame = GetFrame(ctx->frame_queue, ctx->frame_queue_size, par->EncodedOrder);
    if (!frame)
        return MFX_ERR_UNDEFINED_BEHAVIOR;

    // update QP/size stat here

    if (   frame->Status == MFX_BRC_PANIC_BIG_FRAME
      || frame->Status == MFX_BRC_PANIC_SMALL_FRAME)
        panic = 1;

    if (panic || (par->CodedFrameSize >= frame->MinSize && par->CodedFrameSize <= frame->MaxSize)) {
        UpdateBRCState(par->CodedFrameSize, ctx);
        RemoveFromQueue(ctx->frame_queue, ctx->frame_queue_size, frame);
        ctx->frame_queue_size--;
        status->BRCStatus = MFX_BRC_OK;

        // Here update Min/MaxSize for all queued frames

        return MFX_ERR_NONE;
    }

    panic = ((GetTime() - frame->StartTime) >= GetMaxFrameEncodingTime(ctx));

    if (par->CodedFrameSize > frame->MaxSize) {
```

(continues on next page)

(continued from previous page)

```
        if (panic || (frame->QP >= ctx->max_qp[0])) {
            frame->Status = MFX_BRC_PANIC_BIG_FRAME;
        } else {
            frame->Status = MFX_BRC_BIG_FRAME;
            frame->QP = DEFAULT_QP_INC;
        }
    }

    if (par->CodedFrameSize < frame->MinSize) {
        if (panic || (frame->QP <= ctx->min_qp[0])) {
            frame->Status = MFX_BRC_PANIC_SMALL_FRAME;
            status->MinFrameSize = frame->MinSize;
        } else {
            frame->Status = MFX_BRC_SMALL_FRAME;
            frame->QP = DEFAULT_QP_DEC;
        }
    }

    status->BRCStatus = frame->Status;

    return MFX_ERR_NONE;
}

void EncoderInit()
{
     //initialize encoder
     MyBrcContext brc_ctx;
     mfxExtBRC ext_brc;
     mfxExtCodingOption2 co2;
     mfxExtBuffer* ext_buf[2] = {&co2.Header, &ext_brc.Header};
     mfxVideoParam vpar;

     memset(&brc_ctx, 0, sizeof(MyBrcContext));
     memset(&ext_brc, 0, sizeof(mfxExtBRC));
     memset(&co2, 0, sizeof(mfxExtCodingOption2));

     vpar.ExtParam = ext_buf;
     vpar.NumExtParam = sizeof(ext_buf) / sizeof(ext_buf[0]);

     co2.Header.BufferId = MFX_EXTBUFF_CODING_OPTION2;
     co2.Header.BufferSz = sizeof(mfxExtCodingOption2);
     co2.ExtBRC = MFX_CODINGOPTION_ON;

     ext_brc.Header.BufferId = MFX_EXTBUFF_BRC;
     ext_brc.Header.BufferSz = sizeof(mfxExtBRC);
     ext_brc.pthis           = &brc_ctx;
     ext_brc.Init            = MyBrcInit;
     ext_brc.Reset           = MyBrcReset;
     ext_brc.Close           = MyBrcClose;
     ext_brc.GetFrameCtrl    = MyBrcGetFrameCtrl;
     ext_brc.Update          = MyBrcUpdate;

```

(continues on next page)

(continued from previous page)

```
        sts = MFXVideoENCODE_Query(session, &vpar, &vpar);
        if (sts == MFX_ERR_UNSUPPORTED || co2.ExtBRC != MFX_CODINGOPTION_ON)
            // unsupported case
            sts = sts;
        else
            sts = MFXVideoENCODE_Init(session, &vpar);
    }
```

### 3.5.5 JPEG

The application can use the same encoding procedures for JPEG/motion JPEG encoding, as shown in the following pseudo code:

```
// encoder initialization
MFXVideoENCODE_Init (...);
// single frame/picture encoding
MFXVideoENCODE_EncodeFrameAsync (...);
MFXVideoCORE_SyncOperation(...);
// close down
MFXVideoENCODE_Close(...);
```

The application may specify Huffman and quantization tables during encoder initialization by attaching *mfxExtJPEGQuantTables* and *mfxExtJPEGHuffmanTables* buffers to the *mfxVideoParam* structure. If the application does not define tables, then the oneVPL encoder uses tables recommended in ITU-T* Recommendation T.81. If the application does not define a quantization table it must specify the *mfxInfoMFX::Quality* parameter. In this case, the oneVPL encoder scales the default quantization table according to the specified *mfxInfoMFX::Quality* parameter value.

The application should properly configure chroma sampling format and color format using the *mfxFrameInfo::FourCC* and *mfxFrameInfo::ChromaFormat* fields. For example, to encode a 4:2:2 vertically sampled YCbCr picture, the application should set *mfxFrameInfo::FourCC* to *MFX_FOURCC_YUY2* and *mfxFrameInfo::ChromaFormat* to *MFX_CHROMAFORMAT_YUV422V*. To encode a 4:4:4 sampled RGB picture, the application should set *mfxFrameInfo::FourCC* to *MFX_FOURCC_RGB4* and *mfxFrameInfo::ChromaFormat* to *MFX_CHROMAFORMAT_YUV444*.

The oneVPL encoder supports different sets of chroma sampling and color formats on different platforms. The application must call the *MFXVideoENCODE_Query()* function to check if the required color format is supported on a given platform and then initialize the encoder with proper values of *mfxFrameInfo::FourCC* and *mfxFrameInfo::ChromaFormat*.

The application should not define the number of scans and number of components. These numbers are derived by the oneVPL encoder from the `mfxInfoMFx::Interleaved` flag and from chroma type. If interleaved coding is specified, then one scan is encoded that contains all image components. Otherwise, the number of scans is equal to number of components. The encoder uses the following component IDs: “1” for luma (Y), “2” for chroma Cb (U), and “3” for chroma Cr (V).

The application should allocate a buffer that is big enough to hold the encoded picture. A rough upper limit may be calculated using the following equation where **Width** and **Height** are width and height of the picture in pixel and **BytesPerPx** is the number of bytes for one pixel:

```
BufferSizeInKB = 4 + (Width * Height * BytesPerPx + 1023) / 1024;
```

The equation equals 1 for a monochrome picture, 1.5 for NV12 and YV12 color formats, 2 for YUY2 color format, and 3 for RGB32 color format (alpha channel is not encoded).

### 3.5.6 Multi-view Video Encoding

Similar to the decoding and video processing initialization procedures, the application attaches the *mfxExtMVCSeqDesc* structure to the *mfxVideoParam* structure for encoding initialization. The *mfxExtMVCSeqDesc* structure configures the oneVPL MVC encoder to work in three modes:

- **Default dependency mode:** The application specifies *mfxExtMVCSeqDesc::NumView* and all other fields to zero. The oneVPL encoder creates a single operation point with all views (view identifier 0 : NumView-1) as target views. The first view (view identifier 0) is the base view. Other views depend on the base view.
- **Explicit dependency mode:** The application specifies *mfxExtMVCSeqDesc::NumView* and the view dependency array, and sets all other fields to zero. The oneVPL encoder creates a single operation point with all views (view identifier View[0 : NumView-1].ViewId) as target views. The first view (view identifier View[0].ViewId) is the base view. View dependencies are defined as *mfxMVCViewDependency* structures.
- **Complete mode:** The application fully specifies the views and their dependencies. The oneVPL encoder generates a bitstream with corresponding stream structures.

During encoding, the oneVPL encoding function *MFXVideoENCODE_EncodeFrameAsync()* accumulates input frames until encoding of a picture is possible. The function returns *mfxStatus::MFX_ERR_MORE_DATA* for more data at input or *mfxStatus::MFX_ERR_NONE* if it successfully accumulated enough data for encoding a picture. The generated bitstream contains the complete picture (multiple views). The application can change this behavior and instruct the encoder to output each view in a separate bitstream buffer. To do so, the application must turn on the *mfxExtCodingOption::ViewOutput* flag. In this case, the encoder returns *mfxStatus::MFX_ERR_MORE_BITSTREAM* if it needs more bitstream buffers at output and *mfxStatus::MFX_ERR_NONE* when processing of the picture (multiple views) has been finished. It is recommended that the application provide a new input frame each time the oneVPL encoder requests a new bitstream buffer. The application must submit view data for encoding in the order they are described in the *mfxExtMVCSeqDesc* structure. Particular view data can be submitted for encoding only when all views that it depends upon have already been submitted.

The following pseudo code shows the encoding procedure:

```
mfxExtBuffer *eb;
mfxExtMVCSeqDesc  seq_desc;
mfxVideoParam init_param;

init_param.ExtParam=(mfxExtBuffer **)&eb;
init_param.NumExtParam=1;
eb=(mfxExtBuffer *)&seq_desc;

/* init encoder */
MFXVideoENCODE_Init(session, &init_param);

/* perform encoding */
for (;;) {
    MFXVideoENCODE_EncodeFrameAsync(session, NULL, surface2, bits,
                                    &syncp);
    MFXVideoCORE_SyncOperation(session,syncp,INFINITE);
}

/* close encoder */
MFXVideoENCODE_Close(session);
}
```

## 3.6 Video Processing Procedures

The following pseudo code shows the video processing procedure:

```
MFXVideoVPP_QueryIOSurf(session, &init_param, response);
allocate_pool_of_surfaces(in_pool, response[0].NumFrameSuggested);
allocate_pool_of_surfaces(out_pool, response[1].NumFrameSuggested);
MFXVideoVPP_Init(session, &init_param);
mfxFrameSurface1 *in=find_unlocked_surface_and_fill_content(in_pool);
mfxFrameSurface1 *out=find_unlocked_surface_from_the_pool(out_pool);
for (;;) {
   sts=MFXVideoVPP_RunFrameVPPAsync(session,in,out,NULL,&syncp);
   if (sts==MFX_ERR_MORE_SURFACE || sts==MFX_ERR_NONE) {
      MFXVideoCORE_SyncOperation(session,syncp,INFINITE);
      process_output_frame(out);
      out=find_unlocked_surface_from_the_pool(out_pool);
   }
   if (sts==MFX_ERR_MORE_DATA && in==NULL)
      break;
   if (sts==MFX_ERR_NONE || sts==MFX_ERR_MORE_DATA) {
      in=find_unlocked_surface_from_the_pool(in_pool);
      fill_content_for_video_processing(in);
      if (end_of_stream())
         in=NULL;
   }
}
MFXVideoVPP_Close(session);
free_pool_of_surfaces(in_pool);
free_pool_of_surfaces(out_pool);
```

Note the following key points about the example:

- The application uses the *MFXVideoVPP_QueryIOSurf()* function to obtain the number of frame surfaces needed for input and output. The application must allocate two frame surface pools: one for the input and one for the output.
- The video processing function *MFXVideoVPP_RunFrameVPPAsync()* is asynchronous. The application must use the *MFXVideoCORE_SyncOperation()* function to synchronize in order to make the output result ready.
- The body of the video processing procedure covers the following three scenarios:
  - If the number of frames consumed at input is equal to the number of frames generated at output, *VPP* returns *mfxStatus::MFX_ERR_NONE* when an output is ready. The application must process the output frame after synchronization, as the *MFXVideoVPP_RunFrameVPPAsync()* function is asynchronous. The application must provide a NULL input at the end of the sequence to drain any remaining frames.
  - If the number of frames consumed at input is more than the number of frames generated at output, VPP returns *mfxStatus::MFX_ERR_MORE_DATA* for additional input until an output is ready. When the output is ready, VPP returns *mfxStatus::MFX_ERR_NONE*. The application must process the output frame after synchronization and provide a NULL input at the end of the sequence to drain any remaining frames.
  - If the number of frames consumed at input is less than the number of frames generated at output, VPP returns either *mfxStatus::MFX_ERR_MORE_SURFACE* (when more than one output is ready), or *mfxStatus::MFX_ERR_NONE* (when one output is ready and VPP expects new input). In both cases, the application must process the output frame after synchronization and provide a NULL input at the end of the sequence to drain any remaining frames.

### 3.6.1 Configuration

oneVPL configures the video processing pipeline operation based on the difference between the input and output formats, specified in the *mfxVideoParam* structure. The following list shows several examples:

- When the input color format is *YUY2* and the output color format is *NV12*, oneVPL enables color conversion from YUY2 to NV12.
- When the input is interleaved and the output is progressive, oneVPL enables deinterlacing.
- When the input is single field and the output is interlaced or progressive, oneVPL enables field weaving, optionally with deinterlacing.
- When the input is interlaced and the output is single field, oneVPL enables field splitting.

In addition to specifying the input and output formats, the application can provide hints to fine-tune the video processing pipeline operation. The application can disable filters in the pipeline by using the *mfxExtVPPDoNotUse* structure, enable filters by using the *mfxExtVPPDoUse* structure, and configure filters by using dedicated configuration structures. See the *Configurable VPP Filters table* for a complete list of configurable video processing filters, their IDs, and configuration structures. See the *ExtendedBufferID enumerator* for more details.

oneVPL ensures that all filters necessary to convert the input format to the output format are included in the pipeline. oneVPL may skip some optional filters even if they are explicitly requested by the application, for example due to limitations of the underlying hardware. To notify the application about skipped optional filters, oneVPL returns the *mfxStatus::MFX_WRN_FILTER_SKIPPED* warning. The application can retrieve the list of active filters by attaching the *mfxExtVPPDoUse* structure to the *mfxVideoParam* structure and calling the *MFXVideoVPP_GetVideoParam()* function. The application must allocate enough memory for the filter list.

See the *Configurable VPP Filters table* for a full list of configurable filters.

Table 5: Configurable VPP Filters

| Filter ID | Configuration Structure |
|---|---|
| MFX_EXTBUFF_VPP_DENOISE2 | mfxExtVPPDenoise2 |
| MFX_EXTBUFF_VPP_MCTF | mfxExtVppMctf |
| MFX_EXTBUFF_VPP_DETAIL | mfxExtVPPDetail |
| MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION | mfxExtVPPFrameRateConversion |
| MFX_EXTBUFF_VPP_IMAGE_STABILIZATION | mfxExtVPPImageStab |
| MFX_EXTBUFF_VPP_PROCAMP | mfxExtVPPProcAmp |
| MFX_EXTBUFF_VPP_FIELD_PROCESSING | mfxExtVPPFieldProcessing |
| MFX_EXTBUFF_VPP_3DLUT | mfxExtVPP3DLut |

The following example shows video processing configuration:

```
/* enable image stabilization filter with default settings */
mfxExtVPPDoUse du;
mfxU32 al=MFX_EXTBUFF_VPP_IMAGE_STABILIZATION;

du.Header.BufferId=MFX_EXTBUFF_VPP_DOUSE;
du.Header.BufferSz=sizeof(mfxExtVPPDoUse);
du.NumAlg=1;
du.AlgList=&al;

/* configure the mfxVideoParam structure */
mfxVideoParam conf;
mfxExtBuffer *eb=(mfxExtBuffer *)&du;
```

(continues on next page)

(continued from previous page)

```

memset(&conf,0,sizeof(conf));
conf.IOPattern=MFX_IOPATTERN_IN_SYSTEM_MEMORY | MFX_IOPATTERN_OUT_SYSTEM_MEMORY;
conf.NumExtParam=1;
conf.ExtParam=&eb;

conf.vpp.In.FourCC=MFX_FOURCC_YV12;
conf.vpp.Out.FourCC=MFX_FOURCC_NV12;
conf.vpp.In.Width=conf.vpp.Out.Width=1920;
conf.vpp.In.Height=conf.vpp.Out.Height=1088;

/* video processing initialization */
MFXVideoVPP_Init(session, &conf);
```

### 3.6.2 Region of Interest

During video processing operations, the application can specify a region of interest for each frame as shown in the following figure:

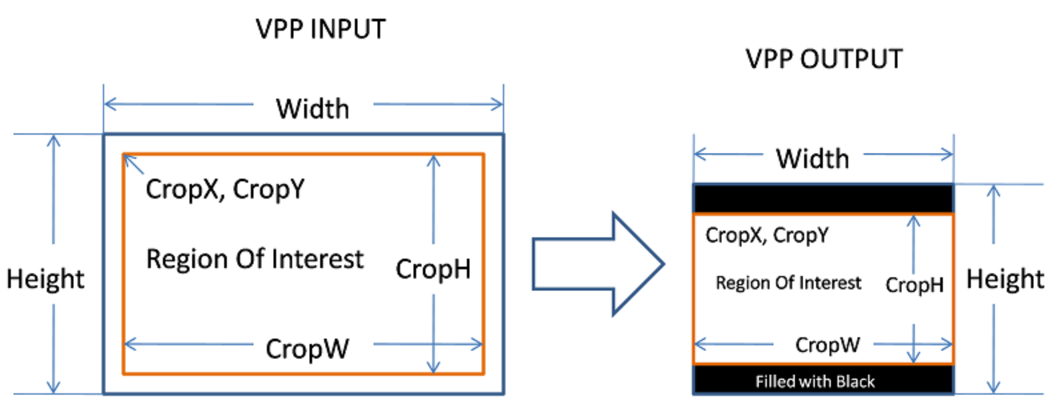



Fig. 2: VPP region of interest operation

Specifying a region of interest guides the resizing function to achieve special effects, such as resizing from 16:9 to 4:3, while keeping the aspect ratio intact. Use the `CropX`, `CropY`, `CropW`, and `CropH` parameters in the *`mfxVideoParam`* structure to specify a region of interest for each frame when calling *`MFXVideoVPP_RunFrameVPPAsync()`*. Note: For per-frame dynamic change, the application should set the `CropX`, `CropY`, `CropW`, and `CropH` parameters when calling *`MFXVideoVPP_RunFrameVPPAsync()`* per frame.

The *VPP Region of Interest Operations table* shows examples of VPP operations applied to a region of interest.

Table 6: VPP Region of Interest Operations

| **Operation** | **VPP Input** *Width X Height* | **VPP Input** *CropX, CropY, CropW, CropH* | **VPP Output** *Width X Height* | **VPP Output** *CropX, CropY, CropW, CropH* |
|---|---|---|---|---|
| Cropping | 720 x 480 | 16, 16, 688, 448 | 720 x 480 | 16, 16, 688, 448 |
| Resizing | 720 x 480 | 0, 0, 720, 480 | 1440 x 960 | 0, 0, 1440, 960 |
| Horizontal stretching | 720 x 480 | 0, 0, 720, 480 | 640 x 480 | 0, 0, 640, 480 |
| 16:9 4:3 with letter boxing at the top and bottom | 1920 x 1088 | 0, 0, 1920, 1088 | 720 x 480 | 0, 36, 720, 408 |
| 4:3 16:9 with pillar boxing at the left and right | 720 x 480 | 0, 0, 720, 480 | 1920 x 1088 | 144, 0, 1632, 1088 |

### 3.6.3 Multi-view Video Processing

oneVPL video processing supports processing multiple views. For video processing initialization, the application needs to attach the *mfxExtMVCSeqDesc* structure to the *mfxVideoParam* structure and call the *MFXVideoVPP_Init()* function. The function saves the view identifiers. During video processing, oneVPL processes each view individually. oneVPL refers to the `FrameID` field of the *mfxFrameInfo* structure to configure each view according to its processing pipeline. If the video processing source frame is not the output from the oneVPL MVC decoder, then the application needs to fill the the `FrameID` field before calling the *MFXVideoVPP_RunFrameVPPAsync()* function. This is shown in the following pseudo code:

```
mfxExtBuffer *eb;
mfxExtMVCSeqDesc  seq_desc;
mfxVideoParam init_param;

init_param.ExtParam = &eb;
init_param.NumExtParam=1;
eb=(mfxExtBuffer *)&seq_desc;

/* init VPP */
MFXVideoVPP_Init(session, &init_param);

/* perform processing */
for (;;) {
    MFXVideoVPP_RunFrameVPPAsync(session,in,out,NULL,&syncp);
    MFXVideoCORE_SyncOperation(session,syncp,INFINITE);
}

/* close VPP */
MFXVideoVPP_Close(session);
```

### 3.6.4 Video Processing 3DLUT

oneVPL video processing supports 3DLUT with Intel HW specific memory layout. The following pseudo code shows how to create a *MFX_3DLUT_MEMORY_LAYOUT_INTEL_65LUT* 3DLUT surface.

```
VADisplay va_dpy = 0;
VASurfaceID surface_id = 0;

vaInitialize(va_dpy, NULL, NULL);

// MFX_3DLUT_MEMORY_LAYOUT_INTEL_65LUT indicate 65*65*128*8bytes.
mfxU32 seg_size = 65, mul_size = 128;
mfxMemId memId = 0;

// create 3DLUT surface (MFX_3DLUT_MEMORY_LAYOUT_INTEL_65LUT)
VASurfaceAttrib    surface_attrib = {};
surface_attrib.type =  VASurfaceAttribPixelFormat;
surface_attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;
surface_attrib.value.type = VAGenericValueTypeInteger;
surface_attrib.value.value.i = VA_FOURCC_RGBA;

vaCreateSurfaces(va_dpy,
                 VA_RT_FORMAT_RGB32,   // 4 bytes
                 seg_size * mul_size,  // 65*128
                 seg_size * 2,         // 65*2
                 &surface_id,
                 1,
                 &surface_attrib,
                 1);

*((VASurfaceID*)memId) = surface_id;

// configure 3DLUT parameters
mfxExtVPP3DLut lut3DConfig;
memset(&lut3DConfig, 0, sizeof(lut3DConfig));
lut3DConfig.Header.BufferId         = MFX_EXTBUFF_VPP_3DLUT;
lut3DConfig.Header.BufferSz         = sizeof(mfxExtVPP3DLut);
lut3DConfig.ChannelMapping          = MFX_3DLUT_CHANNEL_MAPPING_RGB_RGB;
lut3DConfig.BufferType              = MFX_RESOURCE_VA_SURFACE;
lut3DConfig.VideoBuffer.DataType    = MFX_DATA_TYPE_U16;
lut3DConfig.VideoBuffer.MemLayout   = MFX_3DLUT_MEMORY_LAYOUT_INTEL_65LUT;
lut3DConfig.VideoBuffer.MemId       = memId;

// release 3DLUT surface
vaDestroySurfaces(va_dpy, &surface_id, 1);
```

The following pseudo code shows how to create a system memory *mfx3DLutSystemBuffer* 3DLUT surface.

```
// 64 size 3DLUT(3 dimension look up table)
// The buffer size(in bytes) for every channel is 64*64*64*sizeof(DataType)
mfxU16 dataR[64*64*64], dataG[64*64*64], dataB[64*64*64];
mfxChannel channelR, channelG, channelB;
channelR.DataType = MFX_DATA_TYPE_U16;
```

(continues on next page)

(continued from previous page)

```
channelR.Size = 64;
channelR.Data16 = dataR;
channelG.DataType = MFX_DATA_TYPE_U16;
channelG.Size = 64;
channelG.Data16 = dataG;
channelB.DataType = MFX_DATA_TYPE_U16;
channelB.Size = 64;
channelB.Data16 = dataB;

// configure 3DLUT parameters
mfxExtVPP3DLut lut3DConfig;
memset(&lut3DConfig, 0, sizeof(lut3DConfig));
lut3DConfig.Header.BufferId          = MFX_EXTBUFF_VPP_3DLUT;
lut3DConfig.Header.BufferSz          = sizeof(mfxExtVPP3DLut);
lut3DConfig.ChannelMapping           = MFX_3DLUT_CHANNEL_MAPPING_RGB_RGB;
lut3DConfig.BufferType               = MFX_RESOURCE_SYSTEM_SURFACE;
lut3DConfig.SystemBuffer.Channel[0] = channelR;
lut3DConfig.SystemBuffer.Channel[1] = channelG;
lut3DConfig.SystemBuffer.Channel[2] = channelB;
```

### 3.6.5 HDR Tone Mapping

oneVPL video processing supports HDR Tone Mapping with Intel HW. The following pseudo code shows how to perform HDR Tone Mapping.

The following pseudo code shows HDR to SDR.

```
// HDR to SDR (e.g P010 HDR signal -> NV12 SDR signal) in transcoding pipeline
// Attach input external buffers as the below for HDR input. SDR is by default, hence no
// extra output external buffer.
// The input Video Signal Information
mfxExtVideoSignalInfo inSignalInfo   = {};
inSignalInfo.Header.BufferId         = MFX_EXTBUFF_VIDEO_SIGNAL_INFO_IN;
inSignalInfo.Header.BufferSz         = sizeof(mfxExtVideoSignalInfo);
inSignalInfo.VideoFullRange          = 0; // Limited range P010
inSignalInfo.ColourPrimaries         = 9; // BT.2020
inSignalInfo.TransferCharacteristics = 16; // ST2084

// The content Light Level Information
mfxExtContentLightLevelInfo inContentLight = {};
inContentLight.Header.BufferId         = MFX_EXTBUFF_CONTENT_LIGHT_LEVEL_INFO;
inContentLight.Header.BufferSz         = sizeof(mfxExtContentLightLevelInfo);
inContentLight.MaxContentLightLevel    = 4000; // nits
inContentLight.MaxPicAverageLightLevel = 1000; // nits

// The mastering display colour volume
mfxExtMasteringDisplayColourVolume inColourVolume = {};
inColourVolume.Header.BufferId = MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME_IN;
inColourVolume.Header.BufferSz = sizeof(mfxExtMasteringDisplayColourVolume);
// Based on the needs, Please set DisplayPrimaryX/Y[3], WhitePointX/Y, and␣
↪MaxDisplayMasteringLuminance,
```

(continues on next page)

(continued from previous page)

```
// MinDisplayMasteringLuminance

mfxExtBuffer *ExtBufferIn[3];
ExtBufferIn[0] = (mfxExtBuffer *)&inSignalInfo;
ExtBufferIn[1] = (mfxExtBuffer *)&inContentLight;
ExtBufferIn[2] = (mfxExtBuffer *)&inColourVolume;

mfxSession session      = (mfxSession)0;
mfxVideoParam VPPParams = {};
VPPParams.NumExtParam   = 3;
VPPParams.ExtParam      = (mfxExtBuffer **)&ExtBufferIn[0];
MFXVideoVPP_Init(session, &VPPParams);
```

The following pseudo code shows SDR to HDR.

```
// SDR to HDR (e.g NV12 SDR signal -> P010 HDR signal) in transcoding pipeline
// Attach output external buffers as the below for HDR output. SDR is by default, hence␣
↪no
// extra input external buffer.
// The output Video Signal Information
mfxExtVideoSignalInfo outSignalInfo   = {};
outSignalInfo.Header.BufferId        = MFX_EXTBUFF_VIDEO_SIGNAL_INFO_OUT;
outSignalInfo.Header.BufferSz         = sizeof(mfxExtVideoSignalInfo);
outSignalInfo.VideoFullRange          = 0; // Limited range P010
outSignalInfo.ColourPrimaries         = 9; // BT.2020
outSignalInfo.TransferCharacteristics = 16; // ST2084

// The mastering display colour volume
mfxExtMasteringDisplayColourVolume outColourVolume = {};
outColourVolume.Header.BufferId = MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME_OUT;
outColourVolume.Header.BufferSz = sizeof(mfxExtMasteringDisplayColourVolume);
// Based on the needs, Please set DisplayPrimaryX/Y[3], WhitePointX/Y, and␣
↪MaxDisplayMasteringLuminance,
// MinDisplayMasteringLuminance

mfxExtBuffer *ExtBufferOut[2];
ExtBufferOut[0] = (mfxExtBuffer *)&outSignalInfo;
ExtBufferOut[2] = (mfxExtBuffer *)&outColourVolume;

mfxSession session      = (mfxSession)0;
mfxVideoParam VPPParams = {};
VPPParams.NumExtParam   = 2;
VPPParams.ExtParam      = (mfxExtBuffer **)&ExtBufferOut[0];
MFXVideoVPP_Init(session, &VPPParams);
```

The following pseudo code shows HDR to HDR.

```
// HDR to HDR (e.g P010 HDR signal -> P010 HDR signal) in transcoding pipeline
// Attach in/output external buffers as the below for HDR input/output.
// The input Video Signal Information
mfxExtVideoSignalInfo inSignalInfo   = {};
inSignalInfo.Header.BufferId         = MFX_EXTBUFF_VIDEO_SIGNAL_INFO_IN;
```

(continues on next page)

(continued from previous page)

```
inSignalInfo.Header.BufferSz         = sizeof(mfxExtVideoSignalInfo);
inSignalInfo.VideoFullRange          = 0; // Limited range P010
inSignalInfo.ColourPrimaries         = 9; // BT.2020
inSignalInfo.TransferCharacteristics = 16; // ST2084

// The content Light Level Information
mfxExtContentLightLevelInfo inContentLight = {};
inContentLight.Header.BufferId          = MFX_EXTBUFF_CONTENT_LIGHT_LEVEL_INFO;
inContentLight.Header.BufferSz          = sizeof(mfxExtContentLightLevelInfo);
inContentLight.MaxContentLightLevel     = 4000; // nits
inContentLight.MaxPicAverageLightLevel = 1000; // nits

// The mastering display colour volume
mfxExtMasteringDisplayColourVolume inColourVolume = {};
inColourVolume.Header.BufferId = MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME_IN;
inColourVolume.Header.BufferSz = sizeof(mfxExtMasteringDisplayColourVolume);
// Based on the needs, Please set DisplayPrimaryX/Y[3], WhitePointX/Y, and␣
↪MaxDisplayMasteringLuminance,
// MinDisplayMasteringLuminance

mfxExtVideoSignalInfo outSignalInfo   = {};
outSignalInfo.Header.BufferId         = MFX_EXTBUFF_VIDEO_SIGNAL_INFO_OUT;
outSignalInfo.Header.BufferSz         = sizeof(mfxExtVideoSignalInfo);
outSignalInfo.VideoFullRange          = 0; // Limited range P010
outSignalInfo.ColourPrimaries         = 9; // BT.2020
outSignalInfo.TransferCharacteristics = 16; // ST2084

// The mastering display colour volume
mfxExtMasteringDisplayColourVolume outColourVolume = {};
outColourVolume.Header.BufferId = MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME_OUT;
outColourVolume.Header.BufferSz = sizeof(mfxExtMasteringDisplayColourVolume);
// Based on the needs, Please set DisplayPrimaryX/Y[3], WhitePointX/Y, and␣
↪MaxDisplayMasteringLuminance,
// MinDisplayMasteringLuminance

mfxExtBuffer *ExtBuffer[5];
ExtBuffer[0] = (mfxExtBuffer *)&inSignalInfo;
ExtBuffer[1] = (mfxExtBuffer *)&inContentLight;
ExtBuffer[2] = (mfxExtBuffer *)&inColourVolume;
ExtBuffer[3] = (mfxExtBuffer *)&outSignalInfo;
ExtBuffer[4] = (mfxExtBuffer *)&outColourVolume;

mfxSession session      = (mfxSession)0;
mfxVideoParam VPPParams = {};
VPPParams.NumExtParam   = 5;
VPPParams.ExtParam      = (mfxExtBuffer **)&ExtBuffer[0];
MFXVideoVPP_Init(session, &VPPParams);
```

### 3.6.6 Camera RAW acceleration

oneVPL supports camera raw format processing with Intel HW. The following pseudo code shows how to perform camera raw hardware acceleration. For pipeline processing initialization, the application needs to attach the camera structures to the *mfxVideoParam* structure and call the *MFXVideoVPP_Init()* function.

The following pseudo code shows camera raw processing.

```
#ifdef ONEVPL_EXPERIMENTAL
// Camera Raw Format
mfxExtCamPipeControl pipeControl = {};
pipeControl.Header.BufferId      = MFX_EXTBUF_CAM_PIPECONTROL;
pipeControl.Header.BufferSz      = sizeof(mfxExtCamPipeControl);
pipeControl.RawFormat            = (mfxU16)MFX_CAM_BAYER_BGGR;

// Black level correction
mfxExtCamBlackLevelCorrection blackLevelCorrection   = {};
blackLevelCorrection.Header.BufferId                 = MFX_EXTBUF_CAM_BLACK_LEVEL_CORRECTION;
blackLevelCorrection.Header.BufferSz                 = sizeof(mfxExtCamBlackLevelCorrection);
mfxU16 black_level_B = 16, black_level_G0 = 16, black_level_G1 = 16, black_level_R = 16;
// Initialize the value for black level B, G0, G1, R as needed
blackLevelCorrection.B  = black_level_B;
blackLevelCorrection.G0 = black_level_G0;
blackLevelCorrection.G1 = black_level_G1;
blackLevelCorrection.R  = black_level_R;

mfxExtBuffer *ExtBufferIn[2];
ExtBufferIn[0] = (mfxExtBuffer *)&pipeControl;
ExtBufferIn[1] = (mfxExtBuffer *)&blackLevelCorrection;

mfxSession session      = (mfxSession)0;
mfxVideoParam VPPParams = {};
VPPParams.NumExtParam   = 2;
VPPParams.ExtParam      = (mfxExtBuffer **)&ExtBufferIn[0];
MFXVideoVPP_Init(session, &VPPParams);
#endif
```

### 3.6.7 Task submission synchronization

oneVPL can return synchronization object - syncpoint to notify application about submission a task to the GPU. The following example demonstrates the approach.

```
#ifdef ONEVPL_EXPERIMENTAL
mfxExtSyncSubmission syncSubmit = {};
syncSubmit.Header.BufferId      = MFX_EXTBUFF_SYNCSUBMISSION;
syncSubmit.Header.BufferSz      = sizeof(mfxExtSyncSubmission);

mfxFrameSurface1 in;
mfxFrameSurface1 out;
out.Data.ExtParam = (mfxExtBuffer **)&syncSubmit;
```

(continues on next page)

(continued from previous page)

```
out.Data.NumExtParam=1;
sts=MFXVideoVPP_RunFrameVPPAsync(session,&in,&out,NULL,&syncp);
if (MFX_ERR_NONE == sts) {
    MFXVideoCORE_SyncOperation(session, *syncSubmit.SubmissionSyncPoint, INFINITE);
    run_your_GPU_kernel(&out);
}
#endif
```

## 3.7 Transcoding Procedures

The application can use oneVPL encoding, decoding, and video processing functions together for transcoding operations. This section describes the key aspects of connecting two or more oneVPL functions together.

### 3.7.1 Asynchronous Pipeline

The application passes the output of an upstream oneVPL function to the input of the downstream oneVPL function to construct an asynchronous pipeline. Pipeline construction is done at runtime and can be dynamically changed, as shown in the following example:

```
mfxSyncPoint sp_d, sp_e;
MFXVideoDECODE_DecodeFrameAsync(session,bs,work,&vin, &sp_d);
if (going_through_vpp) {
   MFXVideoVPP_RunFrameVPPAsync(session,vin,vout, NULL, &sp_d);
   MFXVideoENCODE_EncodeFrameAsync(session,NULL,vout,bits2,&sp_e);
} else {
   MFXVideoENCODE_EncodeFrameAsync(session,NULL,vin,bits2,&sp_e);
}
MFXVideoCORE_SyncOperation(session,sp_e,INFINITE);
```

oneVPL simplifies the requirements for asynchronous pipeline synchronization. The application only needs to synchronize after the last oneVPL function. Explicit synchronization of intermediate results is not required and may slow performance.

oneVPL tracks dynamic pipeline construction and verifies dependency on input and output parameters to ensure the execution order of the pipeline function. In the previous example, oneVPL will ensure *MFXVideoENCODE_EncodeFrameAsync()* does not begin its operation until *MFXVideoDECODE_DecodeFrameAsync()* or *MFXVideoVPP_RunFrameVPPAsync()* has finished.

During the execution of an asynchronous pipeline, the application must consider the input data as "in use" and must not change it until the execution has completed. The application must also consider output data unavailable until the execution has finished. In addition, for encoders, the application must consider extended and payload buffers as "in use" while the input surface is locked.

oneVPL checks dependencies by comparing the input and output parameters of each oneVPL function in the pipeline. Do not modify the contents of input and output parameters before the previous asynchronous operation finishes. Doing so will break the dependency check and can result in undefined behavior. An exception occurs when the input and output parameters are structures, in which case overwriting fields in the structures is allowed.

**Note:** The dependency check works on the pointers to the structures only.

There are two exceptions with respect to intermediate synchronization:

- If the input is from any asynchronous operation, the application must synchronize any input before calling the oneVPL *MFXVideoDECODE_DecodeFrameAsync()* function.
- When the application calls an asynchronous function to generate an output surface in video memory and passes that surface to a non-oneVPL component, it must explicitly synchronize the operation before passing the surface to the non-oneVPL component.

### 3.7.2 Surface Pool Allocation

When connecting API function **A** to API function **B**, the application must take into account the requirements of both functions to calculate the number of frame surfaces in the surface pool. Typically, the application can use the formula **Na+Nb**, where **Na** is the frame surface requirements for oneVPL function **A** output, and **Nb** is the frame surface requirements for oneVPL function **B** input.

For performance considerations, the application must submit multiple operations and delay synchronization as much as possible, which gives oneVPL flexibility to organize internal pipelining. For example, compare the following two operation sequences, where the first sequence is the recommended order:

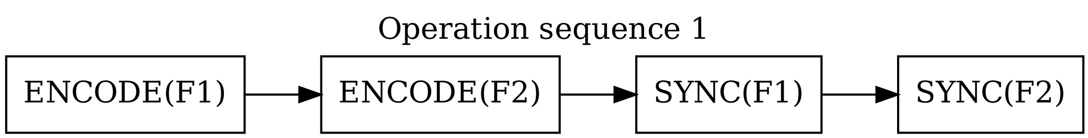



Fig. 3: Recommended operation sequence

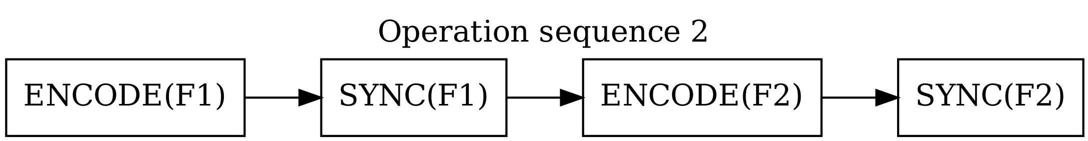



Fig. 4: Operation sequence - not recommended

In this example, the surface pool needs additional surfaces to take into account multiple asynchronous operations before synchronization. The application can use the *mfxVideoParam::AsyncDepth* field to inform a oneVPL function of the number of asynchronous operations the application plans to perform before synchronization. The corresponding oneVPL **QueryIOSurf** function will reflect this number in the *mfxFrameAllocRequest::NumFrameSuggested* value. The following example shows a way of calculating the surface needs based on *mfxFrameAllocRequest::NumFrameSuggested* values:

```
mfxVideoParam init_param_v, init_param_e;
mfxFrameAllocRequest response_v[2], response_e;
```

(continues on next page)

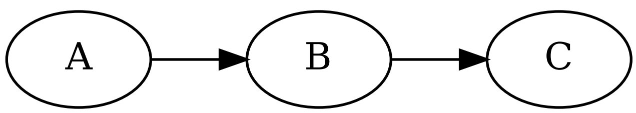
(continued from previous page)

```

// Desired depth
mfxU16 async_depth=4;

init_param_v.AsyncDepth=async_depth;
MFXVideoVPP_QueryIOSurf(session, &init_param_v, response_v);
init_param_e.AsyncDepth=async_depth;
MFXVideoENCODE_QueryIOSurf(session, &init_param_e, &response_e);
mfxU32 num_surfaces = response_v[1].NumFrameSuggested
         + response_e.NumFrameSuggested
         - async_depth; /* double counted in ENCODE & VPP */

allocate_surfaces(num_surfaces);
```

### 3.7.3 Pipeline Error Reporting

During asynchronous pipeline construction, each pipeline stage function will return a synchronization point (sync point). These synchronization points are useful in tracking errors during the asynchronous pipeline operation.

For example, assume the following pipeline:




The application synchronizes on sync point **C**. If the error occurs in function **C**, then the synchronization returns the exact error code. If the error occurs before function **C**, then the synchronization returns *mfxStatus::MFX_ERR_ABORTED*. The application can then try to synchronize on sync point **B**. Similarly, if the error occurs in function **B**, the synchronization returns the exact error code, or else *mfxStatus:: MFX_ERR_ABORTED*. The same logic applies if the error occurs in function **A**.

## 3.8 Parameter Configuration

oneVPL API 2.10 introduces a new interface for configuring VPL for encode, decode, or video processing. Applications may optionally use the new function *mfxConfigInterface::SetParameter* to fill in data structures used for initialization, including *mfxVideoParam* and extension buffers of type *mfxExtBuffer*.

*mfxConfigInterface::SetParameter* accepts as input key-value pairs of `char *` strings, converts these strings to appropriate C data types, and writes the results into the application-provided initialization structures. This can provide a simpler and more flexible initialization method for applications which accept user input in the form of strings, or which store configuration information in formats such as XML, YAML, or JSON.

Applications may freely mix use of *mfxConfigInterface::SetParameter* with standard C-style initialization of the same structures. Additionally, the use of *mfxConfigInterface::SetParameter* facilitates support of new parameters which may be added to VPL API in the future. When new extension buffers are added to VPL API, *mfxConfigInterface::SetParameter* will enable the application to allocate buffers of the appropriate size, and initialize them with the required values, without recompiling the application.

### 3.8.1 Setting a parameter in mfxVideoParam

The following code snippet shows an example of setting a parameter in the structure *mfxVideoParam*.

```
// call MFXVideoCORE_GetHandle to obtain mfxConfigInterface
mfxConfigInterface *iface = nullptr;
sts = MFXVideoCORE_GetHandle(session, MFX_HANDLE_CONFIG_INTERFACE, (mfxHDL *)(&iface));

// Alternately, we could use the following alias:
// sts = MFXGetConfigInterface(session, &iface);

// pass string parameter which maps to mfxVideoParam
mfxVideoParam par   = {};
mfxExtBuffer extBuf = {};
sts = iface->SetParameter(iface, (mfxU8 *)"TargetKbps", (mfxU8 *)"1650", MFX_STRUCTURE_TYPE_VIDEO_PARAM, &par, &extBuf);
```

### 3.8.2 Setting a parameter in an extension buffer

The following code snippet shows an example of setting a parameter in the extension buffer *mfxExtHEVCParam* and attaching it to the structure *mfxVideoParam*.

When setting a parameter which maps to an extension buffer, the function first checks whether the required extension buffer has been attached to the provided *mfxVideoParam*. If so, VPL will update the corresponding field in the extension buffer and return MFX_ERR_NONE.

If the required extension buffer is not attached, VPL will instead return MFX_ERR_MORE_EXTBUFFER. If this happens, the application is required to allocate an extension buffer whose size and buffer ID are indicated by the ext_buffer parameter returned from the call to *mfxConfigInterface::SetParameter*. This extension buffer must then be attached to *mfxVideoParam*, then *mfxConfigInterface::SetParameter* should be called again with the same arguments. If the key and value strings represent a valid parameter in the newly-attached extension buffer, the function will now return MFX_ERR_NONE.

```
// call MFXGetConfigInterface() to obtain mfxConfigInterface
mfxConfigInterface *iface = nullptr;
sts = MFXGetConfigInterface(session, &iface);

// pass string parameter which maps to a VPL extension buffer
mfxVideoParam par   = {};
mfxExtBuffer extBuf = {};
sts = iface->SetParameter(iface, (mfxU8 *)"mfxExtHEVCParam.PicWidthInLumaSamples", (mfxU8 *)"640", MFX_STRUCTURE_TYPE_VIDEO_PARAM, &par, &extBuf);

// if extension buffer has not already been attached, allocate it and call again
if (sts == MFX_ERR_MORE_EXTBUFFER) {
    // the first call to SetParameter filled in extBuf with the buffer ID and size to allocate
    mfxExtBuffer *extBufNew = (mfxExtBuffer *)calloc(extBuf.BufferSz, 1);
    if (!extBufNew)
```

(continues on next page)

(continued from previous page)

```
        return MFX_ERR_MEMORY_ALLOC;

    extBufNew->BufferId = extBuf.BufferId;
    extBufNew->BufferSz = extBuf.BufferSz;

    extBufVector.push_back(extBufNew);
    par.NumExtParam = static_cast<mfxU16>(extBufVector.size());
    par.ExtParam    = extBufVector.data();

    // the correct extension buffer is now attached, so the call should succeed this time
    sts = iface->SetParameter(iface, (mfxU8 *)"mfxExtHEVCParam.PicWidthInLumaSamples",␣
↪(mfxU8 *)"640", MFX_STRUCTURE_TYPE_VIDEO_PARAM, &par, &extBuf);

    if (sts != MFX_ERR_NONE)
        return sts;
    }

return MFX_ERR_NONE;
```

## 3.9 Hardware Acceleration

oneVPL provides a new model for working with hardware acceleration while continuing to support hardware acceleration in legacy mode.

### 3.9.1 New Model to Work with Hardware Acceleration

oneVPL API version 2.0 introduces a new memory model: internal allocation where oneVPL is responsible for video memory allocation. In this mode, an application is not dependent on a low-level video framework API, such as DirectX* or the VA API, and does not need to create and set corresponding low-level oneVPL primitives such as *ID3D11Device* or *VADisplay*. Instead, oneVPL creates all required objects to work with hardware acceleration and video surfaces internally. An application can get access to these objects using `MFXVideoCORE_GetHandle()` or with help of the `mfxFrameSurfaceInterface` interface.

This approach simplifies the oneVPL initialization, making calls to the `MFXVideoENCODE_QueryIOSurf()`, `MFXVideoDECODE_QueryIOSurf()`, or `MFXVideoVPP_QueryIOSurf()` functions optional. See *Internal Memory Management*.

**Note:** Applications can set device handle before session creation through `MFXSetConfigFilterProperty()` like shown in the code below:

```
mfxLoader loader = MFXLoad();
mfxConfig config1 = MFXCreateConfig(loader);
mfxConfig config2 = MFXCreateConfig(loader);
mfxSession session;

mfxVariant HandleType;
HandleType.Type = MFX_VARIANT_TYPE_U32;
HandleType.Data.U32 = MFX_HANDLE_VA_DISPLAY;
```

(continues on next page)

(continued from previous page)

```
MFXSetConfigFilterProperty(config1, (mfxU8*)"mfxHandleType", HandleType);

mfxVariant DisplayHandle;
DisplayHandle.Type = MFX_VARIANT_TYPE_PTR;
HandleType.Data.Ptr = vaDisplay;
MFXSetConfigFilterProperty(config2, (mfxU8*)"mfxHDL", DisplayHandle);

MFXCreateSession(loader, 0, &session);
```

### 3.9.2 Work with Hardware Acceleration in Legacy Mode

#### Work with Multiple Media Devices

If your system has multiple graphics adapters, you may need hints on which adapter is better suited to process a particular workload. The legacy mode of oneVPL provides a helper API to select the most suitable adapter for your workload based on the provided workload description.

**Important:** *MFXQueryAdapters()*, *MFXQueryAdaptersDecode()*, and *MFXQueryAdaptersNumber()* are deprecated starting from API 2.9. Applications should use *MFXEnumImplementations()* and *MFXSetConfigFilterProperty()* to query adapter capabilities and to select a suitable adapter for the input workload.

The following example shows workload initialization on a discrete adapter in legacy mode:

```
mfxU32 num_adapters_available;
mfxIMPL impl;

// Query number of graphics adapters available on system
mfxStatus sts = MFXQueryAdaptersNumber(&num_adapters_available);
MSDK_CHECK_STATUS(sts, "MFXQueryAdaptersNumber failed");

// Allocate memory for response
std::vector<mfxAdapterInfo> displays_data(num_adapters_available);
mfxAdaptersInfo adapters = { displays_data.data(), mfxU32(displays_data.size()), 0u, {0}};

// Query information about all adapters (mind that first parameter is NULL)
sts = MFXQueryAdapters(nullptr, &adapters);
MSDK_CHECK_STATUS(sts, "MFXQueryAdapters failed");

// Find dGfx adapter in list of adapters
auto idx_d = std::find_if(adapters.Adapters, adapters.Adapters + adapters.NumActual,
    [](const mfxAdapterInfo info)
{
   return info.Platform.MediaAdapterType == mfxMediaAdapterType::MFX_MEDIA_DISCRETE;
});

// No dGfx in list
if (idx_d == adapters.Adapters + adapters.NumActual)
```

(continues on next page)

(continued from previous page)

```
{
   printf("Warning: No dGfx detected on machine\n");
   return -1;
}

mfxU32 idx = static_cast<mfxU32>(std::distance(adapters.Adapters, idx_d));

// Choose correct implementation for discrete adapter
switch (adapters.Adapters[idx].Number)
{
case 0:
   impl = MFX_IMPL_HARDWARE;
   break;
case 1:
   impl = MFX_IMPL_HARDWARE2;
   break;
case 2:
   impl = MFX_IMPL_HARDWARE3;
   break;
case 3:
   impl = MFX_IMPL_HARDWARE4;
   break;

default:
   // Try searching on all display adapters
   impl = MFX_IMPL_HARDWARE_ANY;
   break;
}
printf("Choosen implementation: %d\n", impl);
// Initialize mfxSession in regular way with obtained implementation.
```

The example shows that after obtaining the adapter list with *MFXQueryAdapters()*, further initialization of *mfxSession* is performed in the regular way. The specific adapter is selected using the *MFX_IMPL_HARDWARE*, *MFX_IMPL_HARDWARE2*, *MFX_IMPL_HARDWARE3*, or *MFX_IMPL_HARDWARE4* values of *mfxIMPL*.

The following example shows the use of *MFXQueryAdapters()* for querying the most suitable adapter for a particular encode workload:

```
mfxU32 num_adapters_available;
mfxIMPL impl;
mfxVideoParam Encode_mfxVideoParam;

// Query number of graphics adapters available on system
mfxStatus sts = MFXQueryAdaptersNumber(&num_adapters_available);
MSDK_CHECK_STATUS(sts, "MFXQueryAdaptersNumber failed");

// Allocate memory for response
std::vector<mfxAdapterInfo> displays_data(num_adapters_available);
mfxAdaptersInfo adapters = { displays_data.data(), mfxU32(displays_data.size()), 0u, {0}};

// Fill description of Encode workload
```

(continues on next page)

(continued from previous page)

```
mfxComponentInfo interface_request = { MFX_COMPONENT_ENCODE, Encode_mfxVideoParam, {0} };

// Query information about suitable adapters for Encode workload described by Encode_mfxVideoParam
sts = MFXQueryAdapters(&interface_request, &adapters);

if (sts == MFX_ERR_NOT_FOUND)
{
   printf("Error: No adapters on machine capable to process desired workload\n");
   return -1;
}

MSDK_CHECK_STATUS(sts, "MFXQueryAdapters failed");

// Choose correct implementation for discrete adapter. Mind usage of index 0, this is best suitable adapter from MSDK perspective
switch (adapters.Adapters[0].Number)
{
case 0:
   impl = MFX_IMPL_HARDWARE;
   break;
case 1:
   impl = MFX_IMPL_HARDWARE2;
   break;
case 2:
   impl = MFX_IMPL_HARDWARE3;
   break;
case 3:
   impl = MFX_IMPL_HARDWARE4;
   break;

default:
   // Try searching on all display adapters
   impl = MFX_IMPL_HARDWARE_ANY;
   break;
}

printf("Choosen implementation: %d\n", impl);

// Initialize mfxSession in regular way with obtained implementation
```

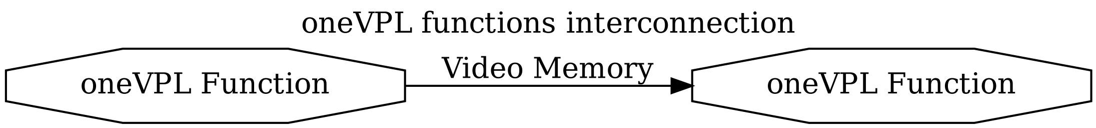
See the *MFXQueryAdapters()* description for adapter priority rules.

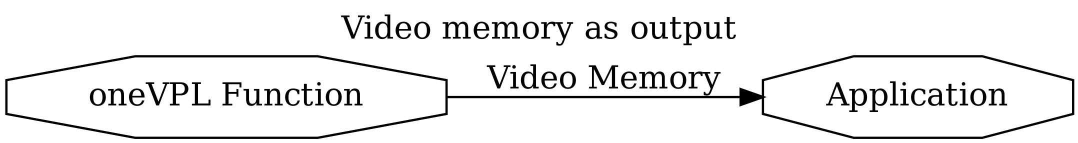
### Work with Video Memory

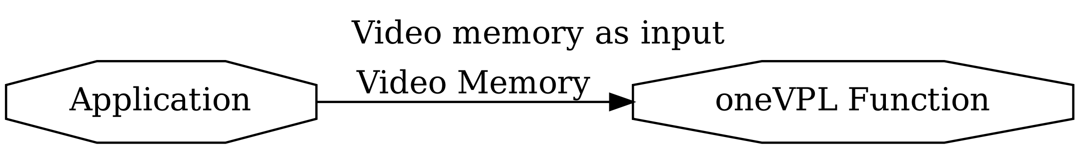
To fully utilize the oneVPL acceleration capability, the application should support OS specific infrastructures. If using Microsoft* Windows*, the application should support Microsoft DirectX*. If using Linux*, the application should support the VA API for Linux.

The hardware acceleration support in an application consists of video memory support and acceleration device support.

Depending on the usage model, the application can use video memory at different stages in the pipeline. Three major scenarios are shown in the following diagrams:



The application must use the *mfxVideoParam::IOPattern* field to indicate the I/O access pattern during initialization. Subsequent function calls must follow this access pattern. For example, if a function operates on video memory surfaces at both input and output, the application must specify the access pattern *IOPattern* at initialization in *MFX_IOPATTERN_IN_VIDEO_MEMORY* for input and *MFX_IOPATTERN_OUT_VIDEO_MEMORY* for output. This particular I/O access pattern must not change inside the **Init - Close** sequence.

Initialization of any hardware accelerated oneVPL component requires the acceleration device handle. This handle is also used by the oneVPL component to query hardware capabilities. The application can share its device with oneVPL by passing the device handle through the *MFXVideoCORE_SetHandle()* function. It is recommended to share the handle before any actual usage of oneVPL.

### Work with Microsoft DirectX* Applications

oneVPL supports two different infrastructures for hardware acceleration on the Microsoft Windows OS: the Direct3D* 9 DXVA2 and Direct3D 11 Video API. If Direct3D 9 DXVA2 is used for hardware acceleration, the application should use the *IDirect3DDeviceManager9* interface as the acceleration device handle. If the Direct3D 11 Video API is used for hardware acceleration, the application should use the *ID3D11Device* interface as the acceleration device handle.

The application should share one of these interfaces with oneVPL through the *MFXVideoCORE_SetHandle()* function. If the application does not provide the interface, then oneVPL creates its own internal acceleration device. As a result, oneVPL input and output will be limited to system memory only for the external allocation mode, which will reduce oneVPL performance. If oneVPL fails to create a valid acceleration device, then oneVPL cannot proceed with hardware acceleration and returns an error status to the application.

---

**Note:** It is recommended to work in the internal allocation mode if the application does not provide the *IDirect3DDeviceManager9* or *ID3D11Device* interface.

---

The application must create the Direct3D 9 device with the flag `D3DCREATE_MULTITHREADED`. The flag `D3DCREATE_FPU_PRESERVE` is also recommended. This influences floating-point calculations, including PTS values.

The application must also set multi-threading mode for the Direct3D 11 device. The following example shows how to set multi-threading mode for a Direct3D 11 device:

```
ID3D11Device            *pD11Device;
ID3D11DeviceContext     *pD11Context;
ID3D10Multithread       *pD10Multithread;

pD11Device->GetImmediateContext(&pD11Context);
pD11Context->QueryInterface(IID_ID3D10Multithread, &pD10Multithread);
pD10Multithread->SetMultithreadProtected(true);
```

During hardware acceleration, if a Direct3D "device lost" event occurs, the oneVPL operation terminates with the *mfxStatus::MFX_ERR_DEVICE_LOST* return status. If the application provided the Direct3D device handle, the application must reset the Direct3D device.

When the oneVPL decoder creates auxiliary devices for hardware acceleration, it must allocate the list of Direct3D surfaces for I/O access, also known as the surface chain, and pass the surface chain as part of the device creation command. In most cases, the surface chain is the frame surface pool mentioned in the *Frame Surface Locking* section.

The application passes the surface chain to the oneVPL component **Init** function through a oneVPL external allocator callback. See the *Memory Allocation and External Allocators* section for details.

Only the decoder **Init** function requests the external surface chain from the application and uses it for auxiliary device creation. Encoder and VPP **Init** functions may only request internal surfaces. See the *ExtMemFrameType enumerator* for more details about different memory types.

Depending on configuration parameters, oneVPL requires different surface types. It is strongly recommended to call the *MFXVideoENCODE_QueryIOSurf()* function, the *MFXVideoDECODE_QueryIOSurf()* function, or the *MFXVideoVPP_QueryIOSurf()* function to determine the appropriate type in the external allocation mode.

### Work with VA API Applications

oneVPL supports the VA API infrastructure for hardware acceleration on Linux. The application should use the *VADisplay* interface as the acceleration device handle for this infrastructure and share it with oneVPL through the *MFXVideoCORE_SetHandle()* function.

The following example shows how to obtain the VA display from the X Window System:

```
Display   *x11_display;
VADisplay va_display;

x11_display = XOpenDisplay(current_display);
va_display  = vaGetDisplay(x11_display);

MFXVideoCORE_SetHandle(session, MFX_HANDLE_VA_DISPLAY, (mfxHDL) va_display);
```

The following example shows how to obtain the VA display from the Direct Rendering Manager:

```
int card;
VADisplay va_display;

card = open("/dev/dri/card0", O_RDWR); /* primary card */
va_display = vaGetDisplayDRM(card);
vaInitialize(va_display, &major_version, &minor_version);

MFXVideoCORE_SetHandle(session, MFX_HANDLE_VA_DISPLAY, (mfxHDL) va_display);
```

When the oneVPL decoder creates a hardware acceleration device, it must allocate the list of video memory surfaces for I/O access, also known as the surface chain, and pass the surface chain as part of the device creation command. The application passes the surface chain to the oneVPL component **Init** function through a oneVPL external allocator callback. See the *Memory Allocation and External Allocators* section for details. Starting from oneVPL API version 2.0, oneVPL creates its own surface chain if an external allocator is not set. See the :ref`New Model to work with Hardware Acceleration <hw-acceleration>` section for details.

**Note:** The VA API does not define any surface types and the application can use either *MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET* or *MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET* to indicate data in video memory.

## 3.10 Memory Allocation and External Allocators

There are two models of memory management in oneVPL: internal and external.

### 3.10.1 External Memory Management

In the external memory model, the application must allocate sufficient memory for input and output parameters and buffers and deallocate it when oneVPL functions complete their operations. During execution, the oneVPL functions use callback functions to the application to manage memory for video frames through the external allocator interface *mfxFrameAllocator*.

If an application needs to control the allocation of video frames, it can use callback functions through the *mfxFrameAllocator* interface. If an application does not specify an allocator, an internal allocator is used. However, if an application uses video memory surfaces for input and output, it must specify the hardware acceleration device and an external frame allocator using *mfxFrameAllocator*.

The external frame allocator can allocate different frame types:

- In-system memory.
- In-video memory, as 'Decoder Render Targets' or 'Processor Render Targets.' See *Working with Hardware Acceleration* for additional details.

The external frame allocator responds only to frame allocation requests for the requested memory type and returns *mfxStatus::MFX_ERR_UNSUPPORTED* for all other types. The allocation request uses flags (part of the memory type field) to indicate which oneVPL class initiated the request so that the external frame allocator can respond accordingly.

The following example shows a simple external frame allocator:

```
#define ALIGN32(X) (((mfxU32)((X)+31)) & (~ (mfxU32)31))

typedef struct {
   mfxU16 width, height;
   mfxU8 *base;
} mid_struct;

mfxStatus fa_alloc(mfxHDL pthis, mfxFrameAllocRequest *request, mfxFrameAllocResponse
↪*response) {
   UNUSED_PARAM(pthis);
   if (!(request->Type&MFX_MEMTYPE_SYSTEM_MEMORY))
      return MFX_ERR_UNSUPPORTED;
   if (request->Info.FourCC!=MFX_FOURCC_NV12)
      return MFX_ERR_UNSUPPORTED;
   response->NumFrameActual=request->NumFrameMin;
   for (int i=0;i<request->NumFrameMin;i++) {
      mid_struct *mmid=(mid_struct *)malloc(sizeof(mid_struct));
      mmid->width=ALIGN32(request->Info.Width);
      mmid->height=ALIGN32(request->Info.Height);
      mmid->base=(mfxU8*)malloc(mmid->width*mmid->height*3/2);
      response->mids[i]=mmid;
   }
   return MFX_ERR_NONE;
}

mfxStatus fa_lock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
   UNUSED_PARAM(pthis);
   mid_struct *mmid=(mid_struct *)mid;
   ptr->Pitch=mmid->width;
   ptr->Y=mmid->base;
   ptr->U=ptr->Y+mmid->width*mmid->height;
```

(continues on next page)

(continued from previous page)

```
   ptr->V=ptr->U+1;
   return MFX_ERR_NONE;
}

mfxStatus fa_unlock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
   UNUSED_PARAM(pthis);
   UNUSED_PARAM(mid);
   if (ptr) ptr->Y=ptr->U=ptr->V=ptr->A=0;
   return MFX_ERR_NONE;
}

mfxStatus fa_gethdl(mfxHDL pthis, mfxMemId mid, mfxHDL *handle) {
   UNUSED_PARAM(pthis);
   UNUSED_PARAM(mid);
   UNUSED_PARAM(handle);
   return MFX_ERR_UNSUPPORTED;
}

mfxStatus fa_free(mfxHDL pthis, mfxFrameAllocResponse *response) {
   UNUSED_PARAM(pthis);
   for (int i=0;i<response->NumFrameActual;i++) {
      mid_struct *mmid=(mid_struct *)response->mids[i];
      free(mmid->base); free(mmid);
   }
   return MFX_ERR_NONE;
}
```

For system memory, it is highly recommended to allocate memory for all planes of the same frame as a single buffer (using one single malloc call).

### 3.10.2 Internal Memory Management

In the internal memory management model, oneVPL provides interface functions for frames allocation:

- *MFXMemory_GetSurfaceForVPP()*
- *MFXMemory_GetSurfaceForVPPOut()*
- *MFXMemory_GetSurfaceForEncode()*
- *MFXMemory_GetSurfaceForDecode()*

These functions are used together with *mfxFrameSurfaceInterface* for surface management. The surface returned by these functions is a reference counted object and the application must call *mfxFrameSurfaceInterface::Release* after finishing all operations with the surface. In this model the application does not need to create and set the external allocator to oneVPL.

Another method to obtain an internally allocated surface is to call *MFXVideoDECODE_DecodeFrameAsync()* with a working surface equal to NULL (see *Simplified decoding procedure*). In this scenario, the decoder will allocate a new refcountable *mfxFrameSurface1* and return it to the user. All assumed contracts with the user are similar to the **MFXMemory_GetSurfaceForXXX** functions.

### 3.10.3 mfxFrameSurfaceInterface

oneVPL API version 2.0 introduces *mfxFrameSurfaceInterface*. This interface is a set of callback functions to manage the lifetime of allocated surfaces, get access to pixel data, and obtain native handles and device abstractions (if suitable). Instead of directly accessing *mfxFrameSurface1* structure members, it's recommended to use the *mfxFrameSurfaceInterface* if present or call external allocator callback functions if set.

The following pseudo code shows the usage of *mfxFrameSurfaceInterface* for memory sharing:

```
// lets decode frame and try to access output in an optimal way.
sts = MFXVideoDECODE_DecodeFrameAsync(session, NULL, NULL, &outsurface, &syncp);
if (MFX_ERR_NONE == sts)
{
    mfxStatus s = outsurface->FrameInterface->GetDeviceHandle(outsurface,
                                                  &device_handle, &device_type);
    // if application or component is familar with mfxHandleType and it's
    // possible to share memory created by device_handle.
    if (MFX_ERR_NONE == s && isDeviceTypeCompatible(device_type)
                          && isPossibleForMemorySharing(device_handle)) {
        // get native handle and type
        outsurface->FrameInterface->GetNativeHandle(outsurface,
                                                      &resource, &resource_type);
        if (isResourceTypeCompatible(resource_type)) {
            //use memory directly
            ProcessNativeMemory(resource);
            outsurface->FrameInterface->Release(outsurface);
        }
    } else {
      // Application or component is not aware about such DeviceHandle or
      // Resource type need to map to system memory.
      outsurface->FrameInterface->Map(outsurface, MFX_MAP_READ);
      ProcessSystemMemory(outsurface);
      outsurface->FrameInterface->Unmap(outsurface);
      outsurface->FrameInterface->Release(outsurface);
    }
}
```

## 3.11 Importing and Exporting Shared Surfaces

oneVPL API 2.10 introduces new interfaces for sharing surfaces between VPL runtime and other applications, frameworks, and graphics APIs. This functionality is only supported when using the *internal memory management* model.

**Importing** a surface enables VPL to access raw video data as input to encode or VPP operations without first mapping the data to system memory and then copying it to a surface allocated by VPL runtime using *mfxFrameSurfaceInterface::Map*.

**Exporting** a surface similarly enables the application to access raw video data which is the output of decode or VPP operations and which was allocated by VPL runtime, without first mapping to system memory using *mfxFrameSurfaceInterface::Map*.

### 3.11.1 Import Example

The following code snippet shows an example of importing a shared surface.

```
// get interface with import function
mfxMemoryInterface *memoryInterface = nullptr;
MFXGetMemoryInterface(session, &memoryInterface);

// capture desktop as a D3D11 texture using an OS-specific capture API
ID3D11Texture2D *pTexture2D;
CaptureFrame(&pTexture2D);

// import D3D11 texture into VPL, zero-copy (shared) is preferred, copy is permitted if zero-copy is not supported
mfxSurfaceD3D11Tex2D d3d11_surface                          = {};
d3d11_surface.SurfaceInterface.Header.SurfaceType  = MFX_SURFACE_TYPE_D3D11_TEX2D;
d3d11_surface.SurfaceInterface.Header.SurfaceFlags = (MFX_SURFACE_FLAG_IMPORT_SHARED | MFX_SURFACE_FLAG_IMPORT_COPY);
d3d11_surface.SurfaceInterface.Header.StructSize   = sizeof(mfxSurfaceD3D11Tex2D);

// pass the pointer to the shared D3D11 texture
d3d11_surface.texture2D = pTexture2D;

// external_surface is a pointer to mfxSurfaceHeader but points to a complete structure of type mfxSurfaceD3D11Tex2D
mfxSurfaceHeader *external_surface = reinterpret_cast<mfxSurfaceHeader *>(&d3d11_surface);

// ImportFrameSurface() will return a VPL surface which may then be used as input for encode or VPP
mfxFrameSurface1 *imported_surface = nullptr;
memoryInterface->ImportFrameSurface(memoryInterface, MFX_SURFACE_COMPONENT_ENCODE, external_surface, &imported_surface);

// encode the surface
MFXVideoENCODE_EncodeFrameAsync(session, nullptr, imported_surface, &bitstream, &syncp);

// release imported surface
imported_surface->FrameInterface->Release(imported_surface);
```

### 3.11.2 Export Example

The following code snippet shows an example of exporting a shared surface.

```
// decode frame
mfxFrameSurface1 *decoded_surface = nullptr;
MFXVideoDECODE_DecodeFrameAsync(session, &bitstream, nullptr, &decoded_surface, &syncp);

// run VPP on frame
mfxFrameSurface1 *vpp_out_surface = nullptr;
MFXVideoVPP_ProcessFrameAsync(session, decoded_surface, &vpp_out_surface);

```

(continues on next page)

(continued from previous page)

```
// release decoded frame (decrease reference count) after passing to VPP
decoded_surface->FrameInterface->Release(decoded_surface);

// export mfxFrameSurface1 from VPL to a shared D3D11 texture, zero-copy (shared) is enabled
mfxSurfaceHeader export_header = {};
export_header.SurfaceType  = MFX_SURFACE_TYPE_D3D11_TEX2D;
export_header.SurfaceFlags = MFX_SURFACE_FLAG_EXPORT_SHARED;

// exported_surface is a pointer to mfxSurfaceHeader but will point to a complete structure of type mfxSurfaceD3D11Tex2D
mfxSurfaceHeader *exported_surface = nullptr;
vpp_out_surface->FrameInterface->Export(vpp_out_surface, export_header, &exported_surface);

// get pointer to the shared D3D11 texture
mfxSurfaceD3D11Tex2D *d3d11_surface = reinterpret_cast<mfxSurfaceD3D11Tex2D *>(exported_surface);
ID3D11Texture2D *pTexture2D = reinterpret_cast<ID3D11Texture2D *>(d3d11_surface->texture2D);

// render the D3D11 texture to screen
RenderFrame(pTexture2D);

// release exported surface
mfxSurfaceInterface *exported_surface_interface = reinterpret_cast<mfxSurfaceInterface *>(exported_surface);
exported_surface_interface->Release(exported_surface_interface);

// release VPP output frame
vpp_out_surface->FrameInterface->Release(vpp_out_surface);
```

## 3.12 Hardware Device Error Handling

For implementations that accelerate decoding, encoding, and video processing through a hardware device, API functions may return errors or warnings if the hardware device encounters errors. See the *Hardware Device Errors and Warnings table* for detailed information about the errors and warnings.

Table 7: Hardware Device Errors and Warnings

| Status | Description |
|---|---|
| *mfxStatus::MFX_ERR_DEVICE_FAILED* | Hardware device returned unexpected errors. oneVPL was unable to restore operation. |
| *mfxStatus::MFX_ERR_DEVICE_LOST* | Hardware device was lost due to system lock or shutdown. |
| *mfxStatus::MFX_WRN_PARTIAL_ACCELERATION* | The hardware does not fully support the specified configuration. The encoding, decoding, or video processing operation may be partially accelerated. |
| *mfxStatus::MFX_WRN_DEVICE_BUSY* | Hardware device is currently busy. |

oneVPL **Query**, **QueryIOSurf**, and **Init** functions return *mfxStatus::MFX_WRN_PARTIAL_ACCELERATION* to indicate that the encoding, decoding, or video processing operation can be partially hardware accelerated or not hardware accelerated at all. The application can ignore this warning and proceed with the operation. (Note that oneVPL functions may return errors or other warnings overwriting *mfxStatus::MFX_WRN_PARTIAL_ACCELERATION*, as it is a lower priority warning.)

oneVPL functions return *mfxStatus::MFX_WRN_DEVICE_BUSY* to indicate that the hardware device is busy and unable to receive commands at this time. The recommended approach is:

- If the asynchronous operation returns synchronization point along with *mfxStatus::MFX_WRN_DEVICE_BUSY* - call the *MFXVideoCORE_SyncOperation()* with it.
- If application has buffered synchronization point(s) obtained from previous asynchronous operations - call *MFXVideoCORE_SyncOperation()* with the oldest one.
- If no synchronization point(s) available - wait for a few milliseconds.
- Resume the operation by resubmitting the request.

```
mfxStatus sts=MFX_ERR_NONE;
for (;;) {
   // do something
   sts=MFXVideoDECODE_DecodeFrameAsync(session, bitstream,  surface_work, &surface_disp, &syncp);
   if (sts == MFX_ERR_NONE) buffered_syncp = syncp;
   else if (sts == MFX_WRN_DEVICE_BUSY) prg_handle_device_busy(session, syncp ? syncp : buffered_syncp);

}
```

The same procedure applies to encoding and video processing.

oneVPL functions return *mfxStatus::MFX_ERR_DEVICE_LOST* or *mfxStatus::MFX_ERR_DEVICE_FAILED* to indicate that there is a complete failure in hardware acceleration. The application must close and reinitialize the oneVPL function class. If the application has provided a hardware acceleration device handle to oneVPL, the application must reset the device.

CHAPTER

FOUR

# MANDATORY APIS AND FUNCTIONS

## 4.1 Disclaimer

Developers can implement any subset of the oneVPL API. The specification makes no claim about what encoder, decoder, VPP filter, or any other underlying features are mandatory for the implementation. The oneVPL API is designed such that users have several options to discover capabilities exposed by the implementation:

1. Before or after session creation: Users can get a list of supported encoders, decoders, VPP filters, correspondent color formats, and memory types with the help of the `MFXEnumImplementations()` function. For more details, see *oneVPL Dispatcher Interactions*.
2. After session is created: Users can call **Query** functions to obtain low level implementation capabilities.

**Attention:** The legacy Intel® Media Software Development Kit implementation does not support the first approach to obtain capabilities.

## 4.2 Exported Functions

The *Exported Functions table* lists all functions that must be exposed by any oneAPI Video Processing Library implementation. The realization of all listed functions is mandatory; most functions may return `mfxStatus::MFX_ERR_NOT_IMPLEMENTED`.

**Note:** Functions `MFXInit()` and `MFXInitEx()` are not required to be exported.

See *Mandatory APIs* for details about which functions, in which conditions, must not return `mfxStatus::MFX_ERR_NOT_IMPLEMENTED`.

Table 1: Exported Functions

| Function | API Version |
| --- | --- |
| `MFXClose()` | 1.0 |
| `MFXQueryIMPL()` | 1.0 |
| `MFXQueryVersion()` | 1.0 |
| `MFXJoinSession()` | 1.1 |
| `MFXDisjoinSession()` | 1.1 |
| `MFXCloneSession()` | 1.1 |
| `MFXSetPriority()` | 1.1 |

continues on next page

Table 1 – continued from previous page

| Function | API Version |
|---|---|
| MFXGetPriority() | 1.1 |
| MFXVideoCORE_SetFrameAllocator() | 1.0 |
| MFXVideoCORE_SetHandle() | 1.0 |
| MFXVideoCORE_GetHandle() | 1.0 |
| MFXVideoCORE_SyncOperation() | 1.0 |
| MFXVideoENCODE_Query() | 1.0 |
| MFXVideoENCODE_QueryIOSurf() | 1.0 |
| MFXVideoENCODE_Init() | 1.0 |
| MFXVideoENCODE_Reset() | 1.0 |
| MFXVideoENCODE_Close() | 1.0 |
| MFXVideoENCODE_GetVideoParam() | 1.0 |
| MFXVideoENCODE_GetEncodeStat() | 1.0 |
| MFXVideoENCODE_EncodeFrameAsync() | 1.0 |
| MFXVideoDECODE_Query() | 1.0 |
| MFXVideoDECODE_DecodeHeader() | 1.0 |
| MFXVideoDECODE_QueryIOSurf() | 1.0 |
| MFXVideoDECODE_Init() | 1.0 |
| MFXVideoDECODE_Reset() | 1.0 |
| MFXVideoDECODE_Close() | 1.0 |
| MFXVideoDECODE_GetVideoParam() | 1.0 |
| MFXVideoDECODE_GetDecodeStat() | 1.0 |
| MFXVideoDECODE_SetSkipMode() | 1.0 |
| MFXVideoDECODE_GetPayload() | 1.0 |
| MFXVideoDECODE_DecodeFrameAsync() | 1.0 |
| MFXVideoVPP_Query() | 1.0 |
| MFXVideoVPP_QueryIOSurf() | 1.0 |
| MFXVideoVPP_Init() | 1.0 |
| MFXVideoVPP_Reset() | 1.0 |
| MFXVideoVPP_Close() | 1.0 |
| MFXVideoVPP_GetVideoParam() | 1.0 |
| MFXVideoVPP_GetVPPStat() | 1.0 |
| MFXVideoVPP_RunFrameVPPAsync() | 1.0 |
| MFXVideoCORE_QueryPlatform() | 1.19 |
| MFXQueryAdapters() | 1.31 |
| MFXQueryAdaptersDecode() | 1.31 |
| MFXQueryAdaptersNumber() | 1.31 |
| MFXMemory_GetSurfaceForVPP() | 2.0 |
| MFXMemory_GetSurfaceForEncode() | 2.0 |
| MFXMemory_GetSurfaceForDecode() | 2.0 |
| MFXQueryImplsDescription() | 2.0 |
| MFXReleaseImplDescription() | 2.0 |
| MFXInitialize() | 2.0 |
| MFXMemory_GetSurfaceForVPPOut() | 2.1 |
| MFXVideoVPP_ProcessFrameAsync() | 2.1 |
| MFXVideoDECODE_VPP_Init() | 2.1 |
| MFXVideoDECODE_VPP_DecodeFrameAsync() | 2.1 |
| MFXVideoDECODE_VPP_Reset() | 2.1 |
| MFXVideoDECODE_VPP_GetChannelParam() | 2.1 |
| MFXVideoDECODE_VPP_Close() | 2.1 |

## 4.3 Mandatory APIs

All implementations must implement the APIs listed in the *Mandatory APIs table*:

Table 2: Mandatory APIs

| Functions | Description |
|---|---|
| *MFXInitialize()*<br>*MFXClose()* | Required functions for the dispatcher to create a session. |
| *MFXQueryImplsDescription()*<br>*MFXReleaseImplDescription()* | Required functions for the dispatcher to return implementation capabilities. |
| *MFXVideoCORE_SyncOperation()* | Required function for synchronization of asynchronous operations. |

If the implementation exposes any encoder, decoder, or VPP filter, it must implement the corresponding mandatory APIs, as described in the *Mandatory Encode*, *Decode*, *VPP* and *Decode+VPP* APIs tables:

Table 3: Mandatory Encode APIs

| Functions | Description |
|---|---|
| *MFXVideoENCODE_Init()*<br>*MFXVideoENCODE_Close()*<br>*MFXVideoENCODE_Query()*<br>*MFXVideoENCODE_EncodeFrameAsync()* | Required functions if the implementation implements any encoder. |

Table 4: Mandatory Decode APIs

| Functions | Description |
|---|---|
| *MFXVideoDECODE_Init()*<br>*MFXVideoDECODE_Close()*<br>*MFXVideoDECODE_Query()*<br>*MFXVideoDECODE_DecodeFrameAsync()* | Required functions if the implementation implements any decoder. |

Table 5: Mandatory VPP APIs

| Functions | Description |
|---|---|
| *MFXVideoVPP_Init()*<br>*MFXVideoVPP_Close()*<br>*MFXVideoVPP_Query()*<br>*MFXVideoVPP_RunFrameVPPAsync()* or *MFXVideoVPP_ProcessFrameAsync()* | Required functions if the implementation implements any VPP filter. |

Table 6: Mandatory Decode+VPP APIs

| Functions | Description |
|---|---|
| *MFXVideoDECODE_VPP_Init()*<br>*MFXVideoDECODE_VPP_DecodeFrameAsync()*<br>*MFXVideoDECODE_VPP_Close()* | Required functions if the implementation implements any Decode+VPP component. |

**Note:** Mandatory functions must not return the *MFX_ERR_NOT_IMPLEMENTED* status.

If at least one of the encoder, decoder, or VPP filter functions is implemented, the *MFXQueryImplsDescription()* function must return a valid `mfxImplDescription` structure instance with mandatory capabilities of the implementation, including decoder, encoder, or VPP capabilities information.

If the implementation supports internal memory allocation by exposing at least one of the function from that family: *internal memory allocation and management API* then implementation of whole scope of the `mfxFrameSurfaceInterface` structure as a part of the `mfxFrameSurface1` is mandatory.

Any other functions or extension buffers are optional for the implementation.

CHAPTER

FIVE

# ONEVPL API REFERENCE

## 5.1 Function Reference

*VideoDECODE*
Functions that implement a complete decoder that decompresses input bitstreams directly to output frame surfaces.

*VideoENCODE*
Functions that perform the entire encoding pipeline from the input video frames to the output bitstream.

*VideoVPP*
Functions that perform video processing before encoding, rendering, or standalone.

*VideoCORE*
Functions to perform external device and memory management and synchronization.

*Session Management*
Functions to manage sessions.

*Memory*
Functions for internal memory allocation and management.

*Implementation Capabilities*
Functions to report capabilities of available implementations and create user-requested library implementations.

*Adapters*
Functions that identify graphics adapters for Microsoft* DirectX* video processing, encoding, and decoding.

*VideoDECODE_VPP*
Functions that implement combined operation of decoding and video processing with multiple output frame surfaces.

### 5.1.1 VideoDECODE

Functions that implement a complete decoder that decompresses input bitstreams directly to output frame surfaces.

## API

- *MFXVideoDECODE_Query*
- *MFXVideoDECODE_DecodeHeader*
- *MFXVideoDECODE_QueryIOSurf*
- *MFXVideoDECODE_Init*
- *MFXVideoDECODE_Reset*
- *MFXVideoDECODE_Close*
- *MFXVideoDECODE_GetVideoParam*
- *MFXVideoDECODE_GetDecodeStat*
- *MFXVideoDECODE_SetSkipMode*
- *MFXVideoDECODE_GetPayload*
- *MFXVideoDECODE_DecodeFrameAsync*

### MFXVideoDECODE_Query

*mfxStatus* **MFXVideoDECODE_Query**(*mfxSession* session, *mfxVideoParam* *in, *mfxVideoParam* *out)

Works in one of two modes:

- If the `in` parameter is zero, the function returns the class configurability in the output structure. A non-zero value in each field of the output structure indicates that the field is configurable by the implementation with the MFXVideoDECODE_Init function.
- If the `in` parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values to the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeros the fields. This feature can verify whether the implementation supports certain profiles, levels, or bitrates.

The application can call this function before or after it initializes the decoder. The `CodecId` field of the output structure is a mandated field (to be filled by the application) to identify the coding standard.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **in** – **[in]** Pointer to the *mfxVideoParam* structure as input.
- **out** – **[out]** Pointer to the *mfxVideoParam* structure as output.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_UNSUPPORTED The function failed to identify a specific implementation for the required features.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The decoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

**Important:** The `MFXVideoDECODE_Query()` is mandatory when implementing a decoder.

### MFXVideoDECODE_DecodeHeader

*mfxStatus* **`MFXVideoDECODE_DecodeHeader`**(*mfxSession* session, *mfxBitstream* *bs, *mfxVideoParam* *par)

Parses the input bitstream and fills the *mfxVideoParam* structure with appropriate values, such as resolution and frame rate, for the Init API function.

The application can then pass the resulting structure to the MFXVideoDECODE_Init function for decoder initialization.

An application can call this API function at any time before or after decoder initialization. If the library finds a sequence header in the bitstream, the function moves the bitstream pointer to the first bit of the sequence header. Otherwise, the function moves the bitstream pointer close to the end of the bitstream buffer but leaves enough data in the buffer to avoid possible loss of start code.

The `CodecId` field of the *mfxVideoParam* structure is a mandated field (to be filled by the application) to identify the coding standard.

The application can retrieve a copy of the bitstream header, by attaching the *mfxExtCodingOptionSPSPPS* structure to the *mfxVideoParam* structure.

**Since**

This function is available since API version 1.0.

**Parameters**

- **`session`** – **[in]** Session handle.
- **`bs`** – **[in]** Pointer to the bitstream.
- **`par`** – **[in]** Pointer to the *mfxVideoParam* structure.

**Returns**

- MFX_ERR_NONE The function successfully filled the structure. It does not mean that the stream can be decoded by the library. The application should call MFXVideoDECODE_Query function to check if decoding of the stream is supported.
- MFX_ERR_MORE_DATA The function requires more bitstream data.
- MFX_ERR_UNSUPPORTED `CodecId` field of the *mfxVideoParam* structure indicates some unsupported codec.
- MFX_ERR_INVALID_HANDLE Session is not initialized.
- MFX_ERR_NULL_PTR `bs` or `par` pointer is NULL.

### MFXVideoDECODE_QueryIOSurf

*mfxStatus* **MFXVideoDECODE_QueryIOSurf**(*mfxSession* session, *mfxVideoParam* *par, *mfxFrameAllocRequest* *request)

Returns minimum and suggested numbers of the output frame surfaces required for decoding initialization and their type.

Init will call the external allocator for the required frames with the same set of numbers. The use of this function is recommended. For more information, see the *Working with Hardware Acceleration section*.

The `CodecId` field of the *mfxVideoParam* structure is a mandated field (to be filled by the application) to identify the coding standard. This function does not validate I/O parameters except those used in calculating the number of output surfaces.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the *mfxVideoParam* structure as input.
- **request** – **[in]** Pointer to the *mfxFrameAllocRequest* structure as output.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

### MFXVideoDECODE_Init

*mfxStatus* **MFXVideoDECODE_Init**(*mfxSession* session, *mfxVideoParam* *par)

Allocates memory and prepares tables and necessary structures for encoding.

This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the *mfxVideoParam* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX_ERR_UNDEFINED_BEHAVIOR The function is called twice without a close.

**Important:** The `MFXVideoDECODE_Init()` is mandatory when implementing a decoder.

### MFXVideoDECODE_Reset

*mfxStatus* **MFXVideoDECODE_Reset**(*mfxSession* session, *mfxVideoParam* *par)

Stops the current decoding operation and restores internal structures or parameters for a new decoding operation.

Reset serves two purposes:

- It recovers the decoder from errors.
- It restarts decoding from a new position

The function resets the old sequence header (sequence parameter set in H.264, or sequence header in MPEG-2 and VC-1). The decoder will expect a new sequence header before it decodes the next frame and will skip any bitstream before encountering the new sequence header.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the *mfxVideoParam* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected that video parameters are wrong or they conflict with initialization parameters. Reset is impossible.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

### MFXVideoDECODE_Close

*mfxStatus* **MFXVideoDECODE_Close**(*mfxSession* session)

Terminates the current decoding operation and de-allocates any internal tables or structures.

**Since**

This function is available since API version 1.0.

**Parameters**

**session** – **[in]** Session handle.

**Returns**

MFX_ERR_NONE The function completed successfully.

**Important:** The *MFXVideoDECODE_Close()* is mandatory when implementing a decoder.

### MFXVideoDECODE_GetVideoParam

*mfxStatus* **MFXVideoDECODE_GetVideoParam**(*mfxSession* session, *mfxVideoParam* *par)

Retrieves current working parameters to the specified output structure.

If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure. The application can retrieve a copy of the bitstream header, by attaching the *mfxExtCodingOptionSPSPPS* structure to the *mfxVideoParam* structure.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the corresponding parameter structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXVideoDECODE_GetDecodeStat

*mfxStatus* **MFXVideoDECODE_GetDecodeStat**(*mfxSession* session, *mfxDecodeStat* *stat)

Obtains statistics collected during decoding.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **stat** – **[in]** Pointer to the *mfxDecodeStat* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXVideoDECODE_SetSkipMode

*mfxStatus* **MFXVideoDECODE_SetSkipMode**(*mfxSession* session, *mfxSkipMode* mode)

Sets the decoder skip mode.

The application may use this API function to increase decoding performance by sacrificing output quality. Increasing the skip level first results in skipping of some decoding operations like deblocking and then leads to frame skipping; first B, then P. Particular details are platform dependent.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **mode** – **[in]** Decoder skip mode. See the mfxSkipMode enumerator for details.

**Returns**

MFX_ERR_NONE The function completed successfully and the output surface is ready for decoding

MFX_WRN_VALUE_NOT_CHANGED The skip mode is not affected as the maximum or minimum skip range is reached.

### MFXVideoDECODE_GetPayload

*mfxStatus* **MFXVideoDECODE_GetPayload**(*mfxSession* session, *mfxU64* *ts, *mfxPayload* *payload)

Extracts user data (MPEG-2) or SEI (H.264) messages from the bitstream.

Internally, the decoder implementation stores encountered user data or SEI messages. The application may call this API function multiple times to retrieve the user data or SEI messages, one at a time.

If there is no payload available, the function returns with payload->NumBit=0.

**Since**

This function is available since API version 1.0.

**Parameters**

- `session` – **[in]** Session handle.
- `ts` – **[in]** Pointer to the user data time stamp in units of 90 KHz; divide ts by 90,000 (90 KHz) to obtain the time in seconds; the time stamp matches the payload with a specific decoded frame.
- `payload` – **[in]** Pointer to the *mfxPayload* structure; the payload contains user data in MPEG-2 or SEI messages in H.264.

**Returns**

MFX_ERR_NONE The function completed successfully and the output buffer is ready for decoding.

MFX_ERR_NOT_ENOUGH_BUFFER The payload buffer size is insufficient.

### MFXVideoDECODE_DecodeFrameAsync

*mfxStatus* **MFXVideoDECODE_DecodeFrameAsync**(*mfxSession* session, *mfxBitstream* *bs, *mfxFrameSurface1* *surface_work, *mfxFrameSurface1* **surface_out, *mfxSyncPoint* *syncp)

Decodes the input bitstream to a single output frame.

The `surface_work` parameter provides a working frame buffer for the decoder. The application should allocate the working frame buffer, which stores decoded frames. If the function requires caching frames after decoding, it locks the frames and the application must provide a new frame buffer in the next call.

If, and only if, the function returns MFX_ERR_NONE, the pointer `surface_out` points to the output frame in the display order. If there are no further frames, the function will reset the pointer to zero and return the appropriate status code.

Before decoding the first frame, a sequence header (sequence parameter set in H.264 or sequence header in MPEG-2 and VC-1) must be present. The function skips any bitstreams before it encounters the new sequence header.

The input bitstream `bs` can be of any size. If there are not enough bits to decode a frame, the function returns MFX_ERR_MORE_DATA, and consumes all input bits except if a partial start code or sequence header is at the end of the buffer. In this case, the function leaves the last few bytes in the bitstream buffer. If there is more incoming bitstream, the application should append the incoming bitstream to the bitstream buffer. Otherwise, the application should ignore the remaining bytes in the bitstream buffer and apply the end of stream procedure described below.

The application must set `bs` to NULL to signal end of stream. The application may need to call this API function several times to drain any internally cached frames until the function returns MFX_ERR_MORE_DATA.

If more than one frame is in the bitstream buffer, the function decodes until the buffer is consumed. The decoding process can be interrupted for events such as if the decoder needs additional working buffers, is readying a frame for retrieval, or encountering a new header. In these cases, the function returns appropriate status code and moves the bitstream pointer to the remaining data.

The decoder may return MFX_ERR_NONE without taking any data from the input bitstream buffer. If the application appends additional data to the bitstream buffer, it is possible that the bitstream buffer may contain more than one frame. It is recommended that the application invoke the function repeatedly until the function returns MFX_ERR_MORE_DATA, before appending any more data to the bitstream buffer.

Starting from API 2.0 it is possible to pass NULL instead of surface_work. In such case runtime will allocate output frames internally.

This function is asynchronous.

**Since**

This function is available since API version 1.0.

**Parameters**

- `session` – **[in]** Session handle.
- `bs` – **[in]** Pointer to the input bitstream.
- `surface_work` – **[in]** Pointer to the working frame buffer for the decoder.
- `surface_out` – **[out]** Pointer to the output frame in the display order.
- `syncp` – **[out]** Pointer to the sync point associated with this operation.

**Returns**

MFX_ERR_NONE The function completed successfully and the output surface is ready for decoding.

MFX_ERR_MORE_DATA The function requires more bitstream at input before decoding can proceed.

MFX_ERR_MORE_SURFACE The function requires more frame surface at output before decoding can proceed.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again after MFXVideoCORE_SyncOperation or in a few milliseconds.

MFX_WRN_VIDEO_PARAM_CHANGED The decoder detected a new sequence header in the bitstream. Video parameters may have changed.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The decoder detected incompatible video parameters in the bitstream and failed to follow them.

MFX_ERR_REALLOC_SURFACE Bigger surface_work required. May be returned only if *mfxInfoMFX::EnableReallocRequest* was set to ON during initialization.

MFX_WRN_ALLOC_TIMEOUT_EXPIRED Timeout expired for internal output frame allocation (if set with *mfxExtAllocationHints* and NULL passed as surface_work). Repeat the call in a few milliseconds or re-initialize decoder with higher surface limit.

**Important:** The `MFXVideoDECODE_DecodeFrameAsync()` is mandatory when implementing a decoder.

### 5.1.2 VideoENCODE

Functions that perform the entire encoding pipeline from the input video frames to the output bitstream.

#### API

- *MFXVideoENCODE_Query*
- *MFXVideoENCODE_QueryIOSurf*
- *MFXVideoENCODE_Init*
- *MFXVideoENCODE_Reset*
- *MFXVideoENCODE_Close*
- *MFXVideoENCODE_GetVideoParam*
- *MFXVideoENCODE_GetEncodeStat*
- *MFXVideoENCODE_EncodeFrameAsync*

#### MFXVideoENCODE_Query

*mfxStatus* **MFXVideoENCODE_Query**(*mfxSession* session, *mfxVideoParam* *in, *mfxVideoParam* *out)

Works in either of four modes:

- If the `in` parameter is zero, the function returns the class configurability in the output structure. The application must set to zero the fields it wants to check for support. If the field is supported, function sets non-zero value to this field, otherwise it would be ignored. It indicates that the SDK implementation can configure the field with Init.
- If the `in` parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values in the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeroes the fields. This feature can verify whether the implementation supports certain profiles, levels or bitrates.
- If the `in` parameter is non-zero and *mfxExtEncoderResetOption* structure is attached to it, the function queries for the outcome of the MFXVideoENCODE_Reset function and returns it in the *mfxExtEncoderResetOption* structure attached to out. The query function succeeds if a reset is possible and returns an error otherwise. Unlike other modes that are independent of the encoder state, this one checks if reset is possible in the present encoder state. This mode also requires a completely defined *mfxVideoParam* structure, unlike other modes that support partially defined configurations. See *mfxExtEncoderResetOption* description for more details.
- If the `in` parameter is non-zero and *mfxExtEncoderCapability* structure is attached to it, the function returns encoder capability in the *mfxExtEncoderCapability* structure attached to out. It is recommended to fill in the *mfxVideoParam* structure and set the hardware acceleration device handle before calling the function in this mode.

The application can call this function before or after it initializes the encoder. The `CodecId` field of the output structure is a mandated field (to be filled by the application) to identify the coding standard.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **in** – **[in]** Pointer to the *mfxVideoParam* structure as input.
- **out** – **[out]** Pointer to the *mfxVideoParam* structure as output.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_UNSUPPORTED The function failed to identify a specific implementation for the required features.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

**Important:** The *MFXVideoENCODE_Query()* function is mandatory when implementing an encoder.

### MFXVideoENCODE_QueryIOSurf

*mfxStatus* **MFXVideoENCODE_QueryIOSurf**(*mfxSession* session, *mfxVideoParam* *par, *mfxFrameAllocRequest* *request)

Returns minimum and suggested numbers of the input frame surfaces required for encoding initialization and their type.

Init will call the external allocator for the required frames with the same set of numbers. This function does not validate I/O parameters except those used in calculating the number of input surfaces.

The use of this function is recommended. For more information, see the *Working with Hardware Acceleration section*.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the *mfxVideoParam* structure as input.
- **request** – **[in]** Pointer to the *mfxFrameAllocRequest* structure as output.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

### MFXVideoENCODE_Init

*mfxStatus* **`MFXVideoENCODE_Init`**(*mfxSession* session, *mfxVideoParam* *par)

Allocates memory and prepares tables and necessary structures for encoding.

This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

**Since**

This function is available since API version 1.0.

**Parameters**

- **`session`** – **[in]** Session handle.
- **`par`** – **[in]** Pointer to the *mfxVideoParam* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX_ERR_UNDEFINED_BEHAVIOR The function is called twice without a close;

**Important:** The *`MFXVideoENCODE_Init()`* function is mandatory when implementing an encoder.

### MFXVideoENCODE_Reset

*mfxStatus* **MFXVideoENCODE_Reset**(*mfxSession* session, *mfxVideoParam* *par)

Stops the current encoding operation and restores internal structures or parameters for a new encoding operation, possibly with new parameters.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the *mfxVideoParam* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

### MFXVideoENCODE_Close

*mfxStatus* **MFXVideoENCODE_Close**(*mfxSession* session)

Terminates the current encoding operation and de-allocates any internal tables or structures.

**Since**

This function is available since API version 1.0.

**Parameters**

**session** – **[in]** Session handle.

**Returns**

MFX_ERR_NONE The function completed successfully.

**Important:** The *MFXVideoENCODE_Close()* function is mandatory when implementing an encoder.

### MFXVideoENCODE_GetVideoParam

*mfxStatus* **MFXVideoENCODE_GetVideoParam**(*mfxSession* session, *mfxVideoParam* *par)

Retrieves current working parameters to the specified output structure.

If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure. The application can retrieve a copy of the bitstream header by attaching the *mfxExtCodingOptionSPSPPS* structure to the *mfxVideoParam* structure.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the corresponding parameter structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXVideoENCODE_GetEncodeStat

*mfxStatus* **MFXVideoENCODE_GetEncodeStat**(*mfxSession* session, *mfxEncodeStat* *stat)

Obtains statistics collected during encoding.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **stat** – **[in]** Pointer to the *mfxEncodeStat* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXVideoENCODE_EncodeFrameAsync

*mfxStatus* **MFXVideoENCODE_EncodeFrameAsync**(*mfxSession* session, *mfxEncodeCtrl* *ctrl, *mfxFrameSurface1* *surface, *mfxBitstream* *bs, *mfxSyncPoint* *syncp)

Takes a single input frame in either encoded or display order and generates its output bitstream.

In the case of encoded ordering, the *mfxEncodeCtrl* structure must specify the explicit frame type. In the case of display ordering, this function handles frame order shuffling according to the GOP structure parameters specified during initialization.

Since encoding may process frames differently from the input order, not every call of the function generates output and the function returns MFX_ERR_MORE_DATA. If the encoder needs to cache the frame, the function locks the frame. The application should not alter the frame until the encoder unlocks the frame. If there is output (with return status MFX_ERR_NONE), the return is a frame's worth of bitstream.

It is the calling application's responsibility to ensure that there is sufficient space in the output buffer. The value `BufferSizeInKB` in the *mfxVideoParam* structure at encoding initialization specifies the maximum possible size for any compressed frames. This value can also be obtained from the MFXVideoENCODE_GetVideoParam function after encoding initialization.

To mark the end of the encoding sequence, call this function with a NULL surface pointer. Repeat the call to drain any remaining internally cached bitstreams (one frame at a time) until MFX_ERR_MORE_DATA is returned.

This function is asynchronous.

**Since**

This function is available since API version 1.0.

**Parameters**

- **`session`** – **[in]** Session handle.
- **`ctrl`** – **[in]** Pointer to the *mfxEncodeCtrl* structure for per-frame encoding control; this parameter is optional (it can be NULL) if the encoder works in the display order mode. ctrl can be freed right after successful MFXVideoENCODE_EncodeFrameAsync (it is copied inside), but not the ext buffers attached to this ctrl. If the ext buffers are allocated by the user, do not move, alter or delete unless surface.Data.Locked is zero.
- **`surface`** – **[in]** Pointer to the frame surface structure. For surfaces allocated by VPL RT it is safe to call *mfxFrameSurface1::FrameInterface*->Release after successful MFXVideoENCODE_EncodeFrameAsync. If it is allocated by user, do not move, alter or delete unless surface.Data.Locked is zero.
- **`bs`** – **[out]** Pointer to the output bitstream.
- **`syncp`** – **[out]** Pointer to the returned sync point associated with this operation.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NOT_ENOUGH_BUFFER The bitstream buffer size is insufficient.

MFX_ERR_MORE_DATA The function requires more data to generate any output.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again after MFXVideoCORE_SyncOperation or in a few milliseconds.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM Inconsistent parameters detected not conforming to Configuration Parameter Constraints.

**Important:** The *`MFXVideoENCODE_EncodeFrameAsync()`* function is mandatory when implementing an encoder.

### 5.1.3 VideoVPP

Functions that perform video processing before encoding, rendering, or standalone.

#### API

- *MFXVideoVPP_Query*
- *MFXVideoVPP_QueryIOSurf*
- *MFXVideoVPP_Init*
- *MFXVideoVPP_Reset*
- *MFXVideoVPP_Close*
- *MFXVideoVPP_GetVideoParam*
- *MFXVideoVPP_GetVPPStat*
- *MFXVideoVPP_RunFrameVPPAsync*
- *MFXVideoVPP_ProcessFrameAsync*

#### MFXVideoVPP_Query

*mfxStatus* **MFXVideoVPP_Query**(*mfxSession* session, *mfxVideoParam* *in, *mfxVideoParam* *out)

Works in one of two modes:

- If the `in` pointer is zero, the function returns the class configurability in the output structure. A non-zero value in a field indicates that the implementation can configure it with Init.
- If the `in` parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values to the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeroes the fields.

The application can call this function before or after it initializes the preprocessor.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **in** – **[in]** Pointer to the *mfxVideoParam* structure as input.
- **out** – **[out]** Pointer to the *mfxVideoParam* structure as output.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_UNSUPPORTED The implementation does not support the specified configuration.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

**Important:** The *MFXVideoVPP_Query()* function is mandatory when implementing a VPP filter.

### MFXVideoVPP_QueryIOSurf

*mfxStatus* **MFXVideoVPP_QueryIOSurf**(*mfxSession* session, *mfxVideoParam* *par, *mfxFrameAllocRequest* request[2])

Returns minimum and suggested numbers of the input frame surfaces required for video processing initialization and their type.

The parameter `request[0]` refers to the input requirements; `request[1]` refers to output requirements. Init will call the external allocator for the required frames with the same set of numbers. This function does not validate I/O parameters except those used in calculating the number of input surfaces.

The use of this function is recommended. For more information, see the *Working with Hardware Acceleration section*.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the *mfxVideoParam* structure as input.
- **request** – **[in]** Pointer to the *mfxFrameAllocRequest* structure; use `request[0]` for input requirements and `request[1]` for output requirements for video processing.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

### MFXVideoVPP_Init

*mfxStatus* **MFXVideoVPP_Init**(*mfxSession* session, *mfxVideoParam* *par)

Allocates memory and prepares tables and necessary structures for video processing.

This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the *mfxVideoParam* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX_ERR_UNDEFINED_BEHAVIOR The function is called twice without a close.

MFX_WRN_FILTER_SKIPPED The VPP skipped one or more filters requested by the application.

**Important:** The `MFXVideoVPP_Init()` function is mandatory when implementing a VPP filter.

### MFXVideoVPP_Reset

*mfxStatus* **MFXVideoVPP_Reset**(*mfxSession* session, *mfxVideoParam* *par)

Stops the current video processing operation and restores internal structures or parameters for a new operation.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **par** – **[in]** Pointer to the *mfxVideoParam* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected that video parameters are wrong or they conflict with initialization parameters. Reset is impossible.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

### MFXVideoVPP_Close

*mfxStatus* `MFXVideoVPP_Close`(*mfxSession* session)

Terminates the current video processing operation and de-allocates any internal tables or structures.

**Since**

This function is available since API version 1.0.

**Parameters**

`session` – **[in]** Session handle.

**Returns**

MFX_ERR_NONE The function completed successfully.

---

**Important:** The *`MFXVideoVPP_Close()`* function is mandatory when implementing a VPP filter.

---

### MFXVideoVPP_GetVideoParam

*mfxStatus* `MFXVideoVPP_GetVideoParam`(*mfxSession* session, *mfxVideoParam* *par)

Retrieves current working parameters to the specified output structure.

If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure.

**Since**

This function is available since API version 1.0.

**Parameters**

- `session` – **[in]** Session handle.
- `par` – **[in]** Pointer to the corresponding parameter structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXVideoVPP_GetVPPStat

*mfxStatus* **MFXVideoVPP_GetVPPStat**(*mfxSession* session, *mfxVPPStat* *stat)

Obtains statistics collected during video processing.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **stat** – **[in]** Pointer to the *mfxVPPStat* structure.

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXVideoVPP_RunFrameVPPAsync

*mfxStatus* **MFXVideoVPP_RunFrameVPPAsync**(*mfxSession* session, *mfxFrameSurface1* *in, *mfxFrameSurface1* *out, *mfxExtVppAuxData* *aux, *mfxSyncPoint* *syncp)

Processes a single input frame to a single output frame.

Retrieval of the auxiliary data is optional; the encoding process may use it. The video processing process may not generate an instant output given an input. See the *Video Processing Procedures section* for details on how to correctly send input and retrieve output.

At the end of the stream, call this function with the input argument `in=NULL` to retrieve any remaining frames, until the function returns MFX_ERR_MORE_DATA. This function is asynchronous.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **in** – **[in]** Pointer to the input video surface structure.
- **out** – **[out]** Pointer to the output video surface structure.
- **aux** – **[in]** Optional pointer to the auxiliary data structure.
- **syncp** – **[out]** Pointer to the output sync point.

**Returns**

MFX_ERR_NONE The output frame is ready after synchronization.

MFX_ERR_MORE_DATA Need more input frames before VPP can produce an output.

MFX_ERR_MORE_SURFACE The output frame is ready after synchronization. Need more surfaces at output for additional output frames available.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again after MFXVideoCORE_SyncOperation or in a few milliseconds.

### MFXVideoVPP_ProcessFrameAsync

*mfxStatus* **MFXVideoVPP_ProcessFrameAsync**(*mfxSession* session, *mfxFrameSurface1* *in, *mfxFrameSurface1* **out)

The function processes a single input frame to a single output frame with internal allocation of output frame.

At the end of the stream, call this function with the input argument `in=NULL` to retrieve any remaining frames, until the function returns MFX_ERR_MORE_DATA. This function is asynchronous.

**Since**

This function is available since API version 2.1.

**Parameters**

- **session** – **[in]** Session handle.
- **in** – **[in]** Pointer to the input video surface structure.
- **out** – **[out]** Pointer to the output video surface structure which is reference counted object allocated by the library.

**Returns**

MFX_ERR_NONE The output frame is ready after synchronization.

MFX_ERR_MORE_DATA Need more input frames before VPP can produce an output.

MFX_ERR_MEMORY_ALLOC The function failed to allocate output video frame.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again after MFXVideoCORE_SyncOperation or in a few milliseconds.

MFX_WRN_ALLOC_TIMEOUT_EXPIRED Timeout expired for internal output frame allocation (if set with *mfxExtAllocationHints*). Repeat the call in a few milliseconds or reinitialize VPP with higher surface limit.

**Important:** Either *MFXVideoVPP_RunFrameVPPAsync()* or *MFXVideoVPP_ProcessFrameAsync()* function is mandatory when implementing a VPP filter.

### 5.1.4 VideoCORE

Functions to perform external device and memory management and synchronization.

#### API

- *MFXVideoCORE_SetFrameAllocator*
- *MFXVideoCORE_SetHandle*
- *MFXVideoCORE_GetHandle*
- *MFXVideoCORE_QueryPlatform*
- *MFXVideoCORE_SyncOperation*

#### MFXVideoCORE_SetFrameAllocator

*mfxStatus* **MFXVideoCORE_SetFrameAllocator**(*mfxSession* session, *mfxFrameAllocator* *allocator)

Sets the external allocator callback structure for frame allocation.

If the allocator argument is NULL, the library uses the default allocator, which allocates frames from system memory or hardware devices. The behavior of the API is undefined if it uses this function while the previous allocator is in use. A general guideline is to set the allocator immediately after initializing the session.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **allocator** – **[in]** Pointer to the *mfxFrameAllocator* structure

**Returns**

MFX_ERR_NONE The function completed successfully.

#### MFXVideoCORE_SetHandle

*mfxStatus* **MFXVideoCORE_SetHandle**(*mfxSession* session, *mfxHandleType* type, *mfxHDL* hdl)

Sets any essential system handle that the library might use.

If the specified system handle is a COM interface, the reference counter of the COM interface will increase. The counter will decrease when the session closes.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.

- `type` – **[in]** Handle type
- `hdl` – **[in]** Handle to be set

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_UNDEFINED_BEHAVIOR The same handle is redefined. For example, the function has been called twice with the same handle type or an internal handle has been created before this function call. MFX_ERR_DEVICE_FAILED The SDK cannot initialize using the handle.

### MFXVideoCORE_GetHandle

*mfxStatus* **MFXVideoCORE_GetHandle**(*mfxSession* session, *mfxHandleType* type, *mfxHDL* *hdl)

Obtains system handles previously set by the MFXVideoCORE_SetHandle function.

If the handler is a COM interface, the reference counter of the interface increases. The calling application must release the COM interface.

**Since**

This function is available since API version 1.0.

**Parameters**

- `session` – **[in]** Session handle.
- `type` – **[in]** Handle type
- `hdl` – **[in]** Pointer to the handle to be set

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_UNDEFINED_BEHAVIOR Specified handle type not found.

### MFXVideoCORE_QueryPlatform

*mfxStatus* **MFXVideoCORE_QueryPlatform**(*mfxSession* session, *mfxPlatform* *platform)

Returns information about current hardware platform in the Legacy mode.

**Since**

This function is available since API version 1.19.

**Parameters**

- `session` – **[in]** Session handle.
- `platform` – **[out]** Pointer to the *mfxPlatform* structure

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXVideoCORE_SyncOperation

*mfxStatus* **MFXVideoCORE_SyncOperation**(*mfxSession* session, *mfxSyncPoint* syncp, *mfxU32* wait)

Initiates execution of an asynchronous function not already started and returns the status code after the specified asynchronous operation completes. If wait is zero, the function returns immediately.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **syncp** – **[in]** Sync point
- **wait** – **[in]** wait time in milliseconds

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NONE_PARTIAL_OUTPUT The function completed successfully, bitstream contains a portion of the encoded frame according to required granularity.

MFX_WRN_IN_EXECUTION The specified asynchronous function is in execution.

MFX_ERR_ABORTED The specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

**Important:** The `MFXVideoCORE_SyncOperation()` function is mandatory for any implementation.

## 5.1.5 Session Management

Functions to manage sessions.

### API

- *MFXInit*
- *MFXInitEx*
- *MFXInitialize*
- *MFXClose*
- *MFXQueryIMPL*
- *MFXQueryVersion*
- *MFXJoinSession*
- *MFXDisjoinSession*
- *MFXCloneSession*

- *MFXSetPriority*
- *MFXGetPriority*

### MFXInit

*mfxStatus* **MFXInit**(*mfxIMPL* impl, *mfxVersion* *ver, *mfxSession* *session)

Creates and initializes a session in the legacy mode for compatibility with Intel(r) Media SDK applications. This function is deprecated starting from API version 2.0, applications must use MFXLoad with mfxCreateSession to select the implementation and initialize the session.

Call this function before calling any other API function. If the desired implementation specified by `impl` is MFX_IMPL_AUTO, the function will search for the platform-specific implementation. If the function cannot find the platform-specific implementation, it will use the software implementation instead.

The `ver` argument indicates the desired version of the library implementation. The loaded implementation will have an API version compatible to the specified version (equal in the major version number, and no less in the minor version number.) If the desired version is not specified, the default is to use the API version from the library release with which an application is built.

Production applications should always specify the minimum API version that meets the functional requirements. For example, if an application uses only H.264 decoding as described in API v1.0, the application should initialize the library with API v1.0. This ensures backward compatibility.

*Deprecated:*

Deprecated in API version 2.3. Use MFXLoad and MFXCreateSession to initialize the session. Use MFX_DEPRECATED_OFF macro to turn off the deprecation message visualization.

**Since**

This function is available since API version 1.0.

**Parameters**

- **impl** – **[in]** mfxIMPL enumerator that indicates the desired legacy Intel(r) Media SDK implementation.
- **ver** – **[in]** Pointer to the minimum library version or zero, if not specified.
- **session** – **[out]** Pointer to the legacy Intel(r) Media SDK session handle.

**Returns**

MFX_ERR_NONE The function completed successfully. The output parameter contains the handle of the session.

MFX_ERR_UNSUPPORTED The function cannot find the desired legacy Intel(r) Media SDK implementation or version.

### MFXInitEx

*mfxStatus* **MFXInitEx**(*mfxInitParam* par, *mfxSession* *session)

Creates and initializes a session in the legacy mode for compatibility with Intel(r) Media SDK applications. This function is deprecated starting from API version 2.0, applications must use MFXLoad with mfxCreateSession to select the implementation and initialize the session.

Call this function before calling any other API functions. If the desired implementation specified by `par` is MFX_IMPL_AUTO, the function will search for the platform-specific implementation. If the function cannot find the platform-specific implementation, it will use the software implementation instead.

The argument `par.Version` indicates the desired version of the implementation. The loaded implementation will have an API version compatible to the specified version (equal in the major version number, and no less in the minor version number.) If the desired version is not specified, the default is to use the API version from the library release with which an application is built.

Production applications should always specify the minimum API version that meets the functional requirements. For example, if an application uses only H.264 decoding as described in API v1.0, the application should initialize the library with API v1.0. This ensures backward compatibility.

The argument `par.ExternalThreads` specifies threading mode. Value 0 means that the implementation should create and handle work threads internally (this is essentially the equivalent of the regular MFXInit).

*Deprecated:*

Deprecated in API version 2.3. Use MFXLoad and MFXCreateSession to initialize the session. Use MFX_DEPRECATED_OFF macro to turn off the deprecation message visualization.

**Since**

This function is available since API version 1.14.

**Parameters**

- **par** – **[in]** *mfxInitParam* structure that indicates the desired implementation, minimum library version and desired threading mode.
- **session** – **[out]** Pointer to the session handle.

**Returns**

MFX_ERR_NONE The function completed successfully. The output parameter contains the handle of the session.

MFX_ERR_UNSUPPORTED The function cannot find the desired implementation or version.

### MFXInitialize

*mfxStatus* **MFXInitialize**(*mfxInitializationParam* par, *mfxSession* *session)

Creates and initializes a session starting from API version 2.0. This function is used by the dispatcher. The dispatcher creates and fills the *mfxInitializationParam* structure according to mfxConfig values set by an application. Calling this function directly is not recommended. Instead, applications must call the mfxCreateSession function.

**Since**

This function is available since API version 2.0.

**Parameters**

- **par** – **[in]** *mfxInitializationParam* structure that indicates the minimum library version and acceleration type.
- **session** – **[out]** Pointer to the session handle.

**Returns**

MFX_ERR_NONE The function completed successfully. The output parameter contains the handle of the session.

MFX_ERR_UNSUPPORTED The function cannot find the desired implementation or version.

**Important:** The *MFXInitialize()* function is mandatory for any implementation.

### MFXClose

*mfxStatus* **MFXClose**(*mfxSession* session)

Completes and deinitializes a session. Any active tasks in execution or in queue are aborted. The application cannot call any API function after calling this function.

All child sessions must be disjoined before closing a parent session.

**Since**

This function is available since API version 1.0.

**Parameters**

**session** – **[in]** session handle.

**Returns**

MFX_ERR_NONE The function completed successfully.

**Important:** The *MFXClose()* function is mandatory for any implementation.

### MFXQueryIMPL

*mfxStatus* **MFXQueryIMPL**(*mfxSession* session, *mfxIMPL* *impl)

Returns the implementation type of a given session.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **impl** – **[out]** Pointer to the implementation type

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXQueryVersion

*mfxStatus* **MFXQueryVersion**(*mfxSession* session, *mfxVersion* *version)

Returns the implementation version.

**Since**

This function is available since API version 1.0.

**Parameters**

- **session** – **[in]** Session handle.
- **version** – **[out]** Pointer to the returned implementation version.

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXJoinSession

*mfxStatus* **MFXJoinSession**(*mfxSession* session, *mfxSession* child)

Joins the child session to the current session.

After joining, the two sessions share thread and resource scheduling for asynchronous operations. However, each session still maintains its own device manager and buffer/frame allocator. Therefore, the application must use a compatible device manager and buffer/frame allocator to share data between two joined sessions.

The application can join multiple sessions by calling this function multiple times. When joining the first two sessions, the current session becomes the parent responsible for thread and resource scheduling of any later joined sessions.

Joining of two parent sessions is not supported.

**Since**

This function is available since API version 1.1.

**Parameters**

- **session** – **[inout]** The current session handle.
- **child** – **[in]** The child session handle to be joined

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_WRN_IN_EXECUTION Active tasks are executing or in queue in one of the sessions. Call this function again after all tasks are completed.

MFX_ERR_UNSUPPORTED The child session cannot be joined with the current session.

### MFXDisjoinSession

*mfxStatus* **MFXDisjoinSession**(*mfxSession* session)

Removes the joined state of the current session.

```
After disjoining, the current session becomes independent. The application must␣
↪ensure there is no active task running in the session before calling this API␣
↪function.
```

**Since**
This function is available since API version 1.1.

**Parameters**
**session** – **[inout]** The current session handle.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_WRN_IN_EXECUTION Active tasks are executing or in queue in one of the sessions. Call this function again after all tasks are completed.

MFX_ERR_UNDEFINED_BEHAVIOR The session is independent, or this session is the parent of all joined sessions.

### MFXCloneSession

*mfxStatus* **MFXCloneSession**(*mfxSession* session, *mfxSession* *clone)

Creates a clean copy of the current session.

```
The cloned session is an independent session and does not inherit any user-
↪defined buffer, frame allocator, or device manager handles from the current␣
↪session.
This function is a light-weight equivalent of MFXJoinSession after MFXInit.
```

**Since**
This function is available since API version 1.1.

**Parameters**

- **session** – **[in]** The current session handle.
- **clone** – **[out]** Pointer to the cloned session handle.

**Returns**
MFX_ERR_NONE The function completed successfully.

### MFXSetPriority

*mfxStatus* **MFXSetPriority**(*mfxSession* session, *mfxPriority* priority)

Sets the current session priority.

**Since**

This function is available since API version 1.1.

**Parameters**

- **session** – **[in]** The current session handle.
- **priority** – **[in]** Priority value.

**Returns**

MFX_ERR_NONE The function completed successfully.

### MFXGetPriority

*mfxStatus* **MFXGetPriority**(*mfxSession* session, *mfxPriority* *priority)

Returns the current session priority.

**Since**

This function is available since API version 1.1.

**Parameters**

- **session** – **[in]** The current session handle.
- **priority** – **[out]** Pointer to the priority value.

**Returns**

MFX_ERR_NONE The function completed successfully.

## 5.1.6 Memory

Functions for internal memory allocation and management.

### API

- *MFXMemory_GetSurfaceForVPP*
- *MFXMemory_GetSurfaceForVPPOut*
- *MFXMemory_GetSurfaceForEncode*
- *MFXMemory_GetSurfaceForDecode*

### MFXMemory_GetSurfaceForVPP

*mfxStatus* **MFXMemory_GetSurfaceForVPP**(*mfxSession* session, *mfxFrameSurface1* **surface)

Returns surface which can be used as input for VPP.

VPP should be initialized before this call. Surface should be released with mfxFrameSurface1::FrameInterface.Release(…) after usage. The value of mfxFrameSurface1::Data.Locked for the returned surface is 0.

**Since**

This function is available since API version 2.0.

**Parameters**

- **session** – **[in]** Session handle.
- **surface** – **[out]** Pointer is set to valid *mfxFrameSurface1* object.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR If double-pointer to the `surface` is NULL.

MFX_ERR_INVALID_HANDLE If `session` was not initialized.

MFX_ERR_NOT_INITIALIZED If VPP was not initialized (allocator needs to know surface size from somewhere).

MFX_ERR_MEMORY_ALLOC In case of any other internal allocation error.

MFX_WRN_ALLOC_TIMEOUT_EXPIRED In case of waiting timeout expired (if set with *mfxExtAllocationHints*).

Alias below, can be used as well:

**MFXMemory_GetSurfaceForVPPIn**

Alias for MFXMemory_GetSurfaceForVPP function.

### MFXMemory_GetSurfaceForVPPOut

*mfxStatus* **MFXMemory_GetSurfaceForVPPOut**(*mfxSession* session, *mfxFrameSurface1* **surface)

Returns surface which can be used as output of VPP.

VPP should be initialized before this call. Surface should be released with mfxFrameSurface1::FrameInterface.Release(…) after usage. The value of mfxFrameSurface1::Data.Locked for the returned surface is 0.

**Since**

This function is available since API version 2.1.

**Parameters**

- **session** – **[in]** Session handle.
- **surface** – **[out]** Pointer is set to valid *mfxFrameSurface1* object.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR If double-pointer to the `surface` is NULL.

MFX_ERR_INVALID_HANDLE If `session` was not initialized.

MFX_ERR_NOT_INITIALIZED If VPP was not initialized (allocator needs to know surface size from somewhere).

MFX_ERR_MEMORY_ALLOC In case of any other internal allocation error.

MFX_WRN_ALLOC_TIMEOUT_EXPIRED In case of waiting timeout expired (if set with *mfxExtAllocationHints*).

### MFXMemory_GetSurfaceForEncode

*mfxStatus* **MFXMemory_GetSurfaceForEncode**(*mfxSession* session, *mfxFrameSurface1* **surface)

Returns a surface which can be used as input for the encoder.

Encoder should be initialized before this call. Surface should be released with mfxFrameSurface1::FrameInterface.Release(…) after usage. The value of mfxFrameSurface1::Data.Locked for the returned surface is 0.

**Since**

This function is available since API version 2.0.

**Parameters**

- **session** – **[in]** Session handle.
- **surface** – **[out]** Pointer is set to valid *mfxFrameSurface1* object.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If session was not initialized.

MFX_ERR_NOT_INITIALIZED If the encoder was not initialized (allocator needs to know surface size from somewhere).

MFX_ERR_MEMORY_ALLOC In case of any other internal allocation error.

MFX_WRN_ALLOC_TIMEOUT_EXPIRED In case of waiting timeout expired (if set with *mfxExtAllocationHints*).

### MFXMemory_GetSurfaceForDecode

*mfxStatus* **MFXMemory_GetSurfaceForDecode**(*mfxSession* session, *mfxFrameSurface1* **surface)

Returns a surface which can be used as output of the decoder.

Decoder should be initialized before this call. Surface should be released with mfxFrameSurface1::FrameInterface.Release(…) after usage. The value of mfxFrameSurface1::Data.Locked for the returned surface is 0.’

**Since**

This function is available since API version 2.0.

**Note:** This function was added to simplify transition from legacy surface management to the proposed internal allocation approach. Previously, the user allocated surfaces for the working pool and fed them to the decoder using DecodeFrameAsync calls. With MFXMemory_GetSurfaceForDecode it is possible to change the existing pipeline by just changing the source of work surfaces. Newly developed applications should prefer direct usage of DecodeFrameAsync with internal allocation.

**Parameters**

- **session** – **[in]** Session handle.
- **surface** – **[out]** Pointer is set to valid *mfxFrameSurface1* object.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If session was not initialized.

MFX_ERR_NOT_INITIALIZED If the decoder was not initialized (allocator needs to know surface size from somewhere).

MFX_ERR_MEMORY_ALLOC Other internal allocation error.

MFX_WRN_ALLOC_TIMEOUT_EXPIRED In case of waiting timeout expired (if set with *mfxExtAllocationHints*).

## 5.1.7 Implementation Capabilities

Functions to report capabilities of available implementations and create user-requested library implementations.

### API

- *MFXQueryImplsDescription*
- *MFXReleaseImplDescription*

### MFXQueryImplsDescription

*mfxHDL* ***MFXQueryImplsDescription**(*mfxImplCapsDeliveryFormat* format, *mfxU32* *num_impls)

Delivers implementation capabilities in the requested format according to the format value.

**Since**

This function is available since API version 2.0.

**Parameters**

- **format** – **[in]** Format in which capabilities must be delivered. See mfxImplCapsDeliveryFormat for more details.
- **num_impls** – **[out]** Number of the implementations.

**Returns**

Array of handles to the capability report or NULL in case of unsupported format or NULL num_impls pointer. Length of array is equal to num_impls.

**Important:** The *MFXQueryImplsDescription()* function is mandatory for any implementation.

### MFXReleaseImplDescription

*mfxStatus* **MFXReleaseImplDescription**(*mfxHDL* hdl)

Destroys the handle allocated by the MFXQueryImplsDescription function. Implementation must remember which handles are released. Once the last handle is released, this function must release memory allocated for the array of handles.

**Since**

This function is available since API version 2.0.

**Parameters**

**hdl** – **[in]** Handle to destroy. Can be equal to NULL.

**Returns**

MFX_ERR_NONE The function completed successfully.

**Important:** The *MFXReleaseImplDescription()* function is mandatory for any implementation.

### 5.1.8 Adapters

Functions that identify graphics adapters for Microsoft* DirectX* video processing, encoding, and decoding.

#### API



#### MFXQueryAdapters

*mfxStatus* **MFXQueryAdapters**(*mfxComponentInfo* *input_info, *mfxAdaptersInfo* *adapters)

Returns a list of adapters that are suitable to handle workload `input_info`. The list is sorted in priority order, with iGPU given the highest precedence. This rule may change in the future. If the `input_info` pointer is NULL, the list of all available adapters will be returned.

*Deprecated:*

Deprecated in API version 2.9. Use MFXEnumImplementations and MFXSetConfigFilterProperty to query adapter capabilities and to select a suitable adapter for the input workload. Use MFX_DEPRECATED_OFF macro to turn off the deprecation message visualization.

**Since**

This function is available since API version 1.31.

**Parameters**

- **input_info** – **[in]** Pointer to workload description. See *mfxComponentInfo* description for details.
- **adapters** – **[out]** Pointer to output description of all suitable adapters for input workload. See *mfxAdaptersInfo* description for details.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR `input_info` or adapters pointer is NULL.

MFX_ERR_NOT_FOUND No suitable adapters found.

MFX_WRN_OUT_OF_RANGE Not enough memory to report back entire list of adapters. In this case as many adapters as possible will be returned.

### MFXQueryAdaptersDecode

*mfxStatus* **MFXQueryAdaptersDecode**(*mfxBitstream* *bitstream, *mfxU32* codec_id, *mfxAdaptersInfo* *adapters)

Returns list of adapters that are suitable to decode the input bitstream. The list is sorted in priority order, with iGPU given the highest precedence. This rule may change in the future. This function is a simplification of MFXQueryAdapters, because bitstream is a description of the workload itself.

*Deprecated:*

Deprecated in API version 2.9. Use MFXEnumImplementations and MFXSetConfigFilterProperty to query adapter capabilities and to select a suitable adapter for the input workload. Use MFX_DEPRECATED_OFF macro to turn off the deprecation message visualization.

**Since**

This function is available since API version 1.31.

**Parameters**

- `bitstream` – **[in]** Pointer to bitstream with input data.
- `codec_id` – **[in]** Codec ID to determine the type of codec for the input bitstream.
- `adapters` – **[out]** Pointer to the output list of adapters. Memory should be allocated by user. See *mfxAdaptersInfo* description for details.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR bitstream or `adapters` pointer is NULL.

MFX_ERR_NOT_FOUND No suitable adapters found.

MFX_WRN_OUT_OF_RANGE Not enough memory to report back entire list of adapters. In this case as many adapters as possible will be returned.

### MFXQueryAdaptersNumber

*mfxStatus* **MFXQueryAdaptersNumber**(*mfxU32* *num_adapters)

Returns the number of detected graphics adapters. It can be used before calling MFXQueryAdapters to determine the size of input data that the user will need to allocate.

*Deprecated:*

Deprecated in API version 2.9. Use MFXEnumImplementations and MFXSetConfigFilterProperty to query adapter capabilities and to select a suitable adapter for the input workload. Use MFX_DEPRECATED_OFF macro to turn off the deprecation message visualization.

**Since**

This function is available since API version 1.31.

**Parameters**

`num_adapters` – **[out]** Pointer for the output number of detected graphics adapters.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR num_adapters pointer is NULL.

### 5.1.9 VideoDECODE_VPP

Functions that implement combined operation of decoding and video processing with multiple output frame surfaces.

#### API



#### MFXVideoDECODE_VPP_Init

*mfxStatus* **MFXVideoDECODE_VPP_Init**(*mfxSession* session, *mfxVideoParam* *decode_par, *mfxVideoChannelParam* **vpp_par_array, *mfxU32* num_vpp_par)

Initialize the SDK in (decode + vpp) mode. The logic of this function is similar to MFXVideoDECODE_Init, but application has to provide array of pointers to *mfxVideoChannelParam* and num_channel_param - number of channels. Application is responsible for

memory allocation for *mfxVideoChannelParam* parameters and for each channel it should specify channel IDs:

mfxVideoChannelParam::mfxFrameInfo::ChannelId. ChannelId should be unique value within one session. ChannelID equals to the 0 is reserved for the original decoded frame. The application can attach *mfxExtInCrops* to *mfxVideoChannelParam::ExtParam* to annotate input video frame if it wants to enable letterboxing operation.

**Since**

This function is available since API version 2.1.

**Parameters**

- **session** – **[in]** SDK session handle.
- **decode_par** – **[in]** Pointer to the *mfxVideoParam* structure which contains initialization parameters for decoder.
- **vpp_par_array** – **[in]** Array of pointers to `mfxVideoChannelParam`structures. Each *mfxVideoChannelParam* contains initialization parameters for each VPP channel.
- **num_vpp_par** – **[in]** Size of array of pointers to *mfxVideoChannelParam* structures.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX_ERR_UNDEFINED_BEHAVIOR The component is already initialized.

MFX_WRN_FILTER_SKIPPED The VPP skipped one or more filters requested by the application.

**Important:** The `MFXVideoDECODE_VPP_Init()` is mandatory when implementing a combined decode plus vpp.

### MFXVideoDECODE_VPP_Reset

*mfxStatus* **MFXVideoDECODE_VPP_Reset**(*mfxSession* session, *mfxVideoParam* *decode_par, *mfxVideoChannelParam* **vpp_par_array, *mfxU32* num_vpp_par)

This function is similar to MFXVideoDECODE_Reset and stops the current decoding and vpp operation, and restores internal structures or parameters for a new decoding plus vpp operation. It resets the state of the decoder and/or all initialized vpp channels. Applications have to care about draining of buffered frames for decode and all vpp channels before call this function. The application can attach *mfxExtInCrops* to *mfxVideoChannelParam::ExtParam* to annotate input video frame if it wants to enable letterboxing operation.

**Since**

This function is available since API version 2.1.

**Parameters**

- **session** – **[in]** Session handle.
- **decode_par** – **[in]** Pointer to the `mfxVideoParam` structure which contains new initialization parameters for decoder. Might be NULL if application wants to Reset only VPP channels.
- **vpp_par_array** – **[in]** Array of pointers to *mfxVideoChannelParam* structures. Each *mfxVideoChannelParam* contains new initialization parameters for each VPP channel.
- **num_vpp_par** – **[in]** Size of array of pointers to *mfxVideoChannelParam* structures.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_INVALID_VIDEO_PARAM The function detected that video parameters are wrong or they conflict with initialization parameters. Reset is impossible.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved. MFX_ERR_NULL_PTR Both pointers decode_par and vpp_par_array` equal to zero.

### MFXVideoDECODE_VPP_GetChannelParam

*mfxStatus* **MFXVideoDECODE_VPP_GetChannelParam**(*mfxSession* session, *mfxVideoChannelParam* *par, *mfxU32* channel_id)

Returns actual VPP parameters for selected channel which should be specified by application through

mfxVideoChannelParam::mfxFrameInfo::ChannelId.

**Since**
This function is available since API version 2.1.

**Parameters**

- `session` – **[in]** Session handle.
- `par` – **[in]** Pointer to the *mfxVideoChannelParam* structure which allocated by application
- `channel_id` – **[in]** specifies the requested channel's info

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR par pointer is NULL.

MFX_ERR_NOT_FOUND the library is not able to find VPP channel with such channel_id.

### MFXVideoDECODE_VPP_DecodeFrameAsync

*mfxStatus* **MFXVideoDECODE_VPP_DecodeFrameAsync**(*mfxSession* session, *mfxBitstream* *bs, *mfxU32* *skip_channels, *mfxU32* num_skip_channels, *mfxSurfaceArray* **surf_array_out)

This function is similar to MFXVideoDECODE_DecodeFrameAsync and inherits all bitstream processing logic. As output, it allocates and returns `surf_array_out` array of processed surfaces according to the chain of filters specified by application in MFXVideoDECODE_VPP_Init, including original decoded frames. In the `surf_array_out`, the original decoded frames are returned through surfaces with *mfxFrameInfo::ChannelId* == 0, followed by each of the subsequent frame surfaces for each of the requested *mfxVideoChannelParam* entries provided to the MFXVideoCECODE_VPP_Init function. At maximum, the number of frame surfaces return is 1 + the value of `num_vpp_par` to the MFXVideoDECODE_VPP_Init function, but the application must be prepared to the case when some particular filters are not ready to output surfaces, so the length of `surf_array_out` will be less. Application should use *mfxFrameInfo::ChannelId* parameter to match output surface against configured filter.

An application must synchronize each output surface from the `surf_array_out` surface array independently.

**Since**
This function is available since API version 2.1.

**Parameters**

- **session** – **[in]** SDK session handle.
- **bs** – **[in]** Pointer to the input bitstream.
- **skip_channels** – **[in]** Pointer to the array of ChannelIds which specifies channels with skip output frames. Memory for the array is allocated by application.
- **num_skip_channels** – **[in]** Number of channels addressed by skip_channels.
- **surf_array_out** – **[out]** The address of a pointer to the structure with frame surfaces.

**Returns**

MFX_ERR_NONE The function completed successfully and the output surface is ready for decoding.

MFX_ERR_MORE_DATA The function requires more bitstream at input before decoding can proceed.

MFX_ERR_MORE_SURFACE The function requires more frame surface at output before decoding can proceed.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again after MFXVideoCORE_SyncOperation or in a few milliseconds.

MFX_WRN_VIDEO_PARAM_CHANGED The decoder detected a new sequence header in the bitstream. Video parameters may have changed.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The decoder detected incompatible video parameters in the bitstream and failed to follow them.

MFX_ERR_NULL_PTR num_skip_channels doesn't equal to 0 when skip_channels is NULL.

---

**Important:** The *MFXVideoDECODE_VPP_DecodeFrameAsync()* is mandatory when implementing a combined decode plus vpp.

---

### MFXVideoDECODE_VPP_Close

*mfxStatus* **MFXVideoDECODE_VPP_Close**(*mfxSession* session)

This function is similar to MFXVideoDECODE_Close. It terminates the current decoding and vpp operation and de-allocates any internal tables or structures.

**Since**

This function is available since API version 2.1.

**Parameters**

**session** – **[in]** Session handle.

**Returns**

MFX_ERR_NONE The function completed successfully.

**Important:** The *MFXVideoDECODE_VPP_Close()* is mandatory when implementing a combined decode plus vpp.

## 5.2 Structure Reference

*Type Definitions*
Structures used for type definitions.

*Memory Structures*
Structures used for memory.

*Implementation Management*
Structures used for implementation management.

*Cross-component Structures*
Structures used across library components.

*Decode Structures*
Structures used by Decode only.

*Encode Structures*
Structures used by Encode only.

*VPP Structures*
Structures used by VPP only.

*Protected Structures*
Protected structures.

*DECODE_VPP Structures*
Structures used by *DECODE_VPP* only.

*Camera Structures*
Structures used by Camera Raw Acceleration Processing.

### 5.2.1 Type Definitions

Structures used for type definitions.

#### API

- *mfxExtBuffer*
- *mfxHDLPair*
- *mfxI16Pair*
- *mfxRange32U*
- *mfxStructVersion*

### mfxExtBuffer

struct **mfxExtBuffer**

The common header definition for external buffers and video processing hints.

#### Public Members

*mfxU32* **BufferId**

Identifier of the buffer content. See the ExtendedBufferID enumerator for a complete list of extended buffers.

*mfxU32* **BufferSz**

Size of the buffer.

### mfxHDLPair

struct **mfxHDLPair**

Represents pair of handles of type mfxHDL.

#### Public Members

*mfxHDL* **first**

First handle.

*mfxHDL* **second**

Second handle.

### mfxI16Pair

struct **mfxI16Pair**

Represents a pair of numbers of type mfxI16.

#### Public Members

*mfxI16* **x**

First number.

*mfxI16* **y**

Second number.

### mfxRange32U

struct **mfxRange32U**

Represents a range of unsigned values.

#### Public Members

*mfxU32* **Min**

Minimal value of the range.

*mfxU32* **Max**

Maximal value of the range.

*mfxU32* **Step**

Value increment.

### mfxStructVersion

union **mfxStructVersion**

*#include <mfxdefs.h>* Introduce the field Version for any structure. Assumed that any structure changes are backward binary compatible. *mfxStructVersion* starts from {1,0} for any new API structures. If *mfxStructVersion* is added to the existent legacy structure (replacing reserved fields) it starts from {1, 1}.

#### Major and Minor fields

Anonymous structure with Major and Minor fields. Minor number is incremented when reserved fields are used. Major number is incremented when the size of structure is increased.

*mfxU8* **Minor**

Minor number of the correspondent structure.

*mfxU8* **Major**

Major number of the correspondent structure.

#### Public Members

struct *mfxStructVersion*::[anonymous] **[anonymous]**

*mfxU16* **Version**

Structure version number.

### 5.2.2 Memory Structures

Structures used for memory.

#### API

- *mfxBitstream*
- *mfxFrameAllocator*
- *mfxFrameAllocRequest*
- *mfxFrameAllocResponse*
- *mfxFrameData*
- *mfxFrameInfo*
- *mfxFrameSurface1*
- *mfxFrameSurfaceInterface*
- *mfxSurfacePoolInterface*
- *mfxMemoryInterface*
- *mfxSurfaceTypesSupported*
- *mfxSurfaceHeader*
- *mfxSurfaceInterface*
- *mfxSurfaceD3D11Tex2D*
- *mfxSurfaceVAAPI*
- *mfxSurfaceOpenCLImg2D*
- *mfxExtSurfaceOpenCLImg2DExportDescription*

#### mfxBitstream

struct **mfxBitstream**

Defines the buffer that holds compressed video data.

##### Public Members

mfxEncryptedData ***EncryptedData**

Reserved and must be zero.

*mfxExtBuffer* ****ExtParam**

Array of extended buffers for additional bitstream configuration. See the ExtendedBufferID enumerator for a complete list of extended buffers.

*mfxU16* **NumExtParam**

The number of extended buffers attached to this structure.

*mfxU32* **CodecId**

Specifies the codec format identifier in the FourCC code. See the CodecFormatFourCC enumerator for details. This optional parameter is required for the simplified decode initialization.

*mfxI64* **DecodeTimeStamp**

Decode time stamp of the compressed bitstream in units of 90KHz. A value of MFX_TIMESTAMP_UNKNOWN indicates that there is no time stamp.

This value is calculated by the encoder from the presentation time stamp provided by the application in the *mfxFrameSurface1* structure and from the frame rate provided by the application during the encoder initialization.

*mfxU64* **TimeStamp**

Time stamp of the compressed bitstream in units of 90KHz. A value of MFX_TIMESTAMP_UNKNOWN indicates that there is no time stamp.

*mfxU8* ***Data**

Bitstream buffer pointer, 32-bytes aligned.

*mfxU32* **DataOffset**

Next reading or writing position in the bitstream buffer.

*mfxU32* **DataLength**

Size of the actual bitstream data in bytes.

*mfxU32* **MaxLength**

Allocated bitstream buffer size in bytes.

*mfxU16* **PicStruct**

Type of the picture in the bitstream. Output parameter.

*mfxU16* **FrameType**

Frame type of the picture in the bitstream. Output parameter.

*mfxU16* **DataFlag**

Indicates additional bitstream properties. See the BitstreamDataFlag enumerator for details.

*mfxU16* **reserved2**

Reserved for future use.

### mfxFrameAllocator

struct **mfxFrameAllocator**

Describes the API callback functions Alloc, Lock, Unlock, GetHDL, and Free that the implementation might use for allocating internal frames. Applications that operate on OS-specific video surfaces must implement these API callback functions.

Using the default allocator implies that frame data passes in or out of functions through pointers, as opposed to using memory IDs.

Behavior is undefined when using an incompletely defined external allocator. See the *Memory Allocation and External Allocators section* for additional information.

#### Public Members

*mfxHDL* **pthis**

Pointer to the allocator object.

*mfxStatus* (***Alloc**)(*mfxHDL* pthis, *mfxFrameAllocRequest* *request, *mfxFrameAllocResponse* *response)

Allocates surface frames. For decoders, MFXVideoDECODE_Init calls Alloc only once. That call includes all frame allocation requests. For encoders, MFXVideoENCODE_Init calls Alloc twice: once for the input surfaces and again for the internal reconstructed surfaces.

If two library components must share DirectX* surfaces, this function should pass the pre-allocated surface chain to the library instead of allocating new DirectX surfaces. See the *Surface Pool Allocation section* for additional information.

**Param pthis**
**[in]** Pointer to the allocator object.

**Param request**
**[in]** Pointer to the *mfxFrameAllocRequest* structure that specifies the type and number of required frames.

**Param response**
**[out]** Pointer to the *mfxFrameAllocResponse* structure that retrieves frames actually allocated.

**Return**

MFX_ERR_NONE The function successfully allocated the memory block.

MFX_ERR_MEMORY_ALLOC The function failed to allocate the video frames.

MFX_ERR_UNSUPPORTED The function does not support allocating the specified type of memory.

*mfxStatus* (***Lock**)(*mfxHDL* pthis, *mfxMemId* mid, *mfxFrameData* *ptr)

Locks a frame and returns its pointer.

**Param pthis**
**[in]** Pointer to the allocator object.

**Param mid**
**[in]** Memory block ID.

**Param ptr**
**[out]** Pointer to the returned frame structure.

**Return**

MFX_ERR_NONE The function successfully locked the memory block.

MFX_ERR_LOCK_MEMORY This function failed to lock the frame.

*mfxStatus* (***Unlock**)(*mfxHDL* pthis, *mfxMemId* mid, *mfxFrameData* *ptr)

Unlocks a frame and invalidates the specified frame structure.

**Param pthis**
**[in]** Pointer to the allocator object.

**Param mid**
**[in]** Memory block ID.

**Param ptr**
**[out]** Pointer to the frame structure. This pointer can be NULL.

**Return**
MFX_ERR_NONE The function successfully locked the memory block.

*mfxStatus* (***GetHDL**)(*mfxHDL* pthis, *mfxMemId* mid, *mfxHDL* *handle)

Returns the OS-specific handle associated with a video frame. If the handle is a COM interface, the reference counter must increase. The library will release the interface afterward.

**Param pthis**
**[in]** Pointer to the allocator object.

**Param mid**
**[in]** Memory block ID.

**Param handle**
**[out]** Pointer to the returned OS-specific handle.

**Return**

MFX_ERR_NONE The function successfully returned the OS-specific handle.

MFX_ERR_UNSUPPORTED The function does not support obtaining OS-specific handle..

*mfxStatus* (***Free**)(*mfxHDL* pthis, *mfxFrameAllocResponse* *response)

De-allocates all allocated frames.

**Param pthis**
**[in]** Pointer to the allocator object.

**Param response**
**[in]** Pointer to the *mfxFrameAllocResponse* structure returned by the Alloc function.

**Return**
MFX_ERR_NONE The function successfully de-allocated the memory block.

### mfxFrameAllocRequest

struct **mfxFrameAllocRequest**

Describes multiple frame allocations when initializing encoders, decoders, and video preprocessors. A range specifies the number of video frames. Applications are free to allocate additional frames. In all cases, the minimum number of frames must be at least NumFrameMin or the called API function will return an error.

#### Public Members

*mfxU32* **AllocId**

Unique (within the session) ID of component requested the allocation.

*mfxFrameInfo* **Info**

Describes the properties of allocated frames.

*mfxU16* **Type**

Allocated memory type. See the ExtMemFrameType enumerator for details.

*mfxU16* **NumFrameMin**

Minimum number of allocated frames.

*mfxU16* **NumFrameSuggested**

Suggested number of allocated frames.

### mfxFrameAllocResponse

struct **mfxFrameAllocResponse**

Describes the response to multiple frame allocations. The calling API function returns the number of video frames actually allocated and pointers to their memory IDs.

#### Public Members

*mfxU32* **AllocId**

Unique (within the session) ID of component requested the allocation.

*mfxMemId* ***mids**

Pointer to the array of the returned memory IDs. The application allocates or frees this array.

*mfxU16* **NumFrameActual**

Number of frames actually allocated.

### mfxFrameData

struct **mfxY410**

Specifies "pixel" in Y410 color format.

#### Public Members

*mfxU32* **U**

U component.

*mfxU32* **Y**

Y component.

*mfxU32* **V**

V component.

*mfxU32* **A**

A component.

struct **mfxY416**

Specifies "pixel" in Y416 color format.

#### Public Members

*mfxU32* **U**

U component.

*mfxU32* **Y**

Y component.

*mfxU32* **V**

V component.

*mfxU32* **A**

A component.

struct **mfxA2RGB10**

Specifies "pixel" in A2RGB10 color format

### Public Members

*mfxU32* **B**

B component.

*mfxU32* **G**

G component.

*mfxU32* **R**

R component.

*mfxU32* **A**

A component.

struct **mfxFrameData**

Describes frame buffer pointers.

### Extension Buffers

*mfxU16* **NumExtParam**

The number of extra configuration structures attached to this structure.

### General members

*mfxU16* **reserved**[9]

Reserved for future use.

*mfxU16* **MemType**

Allocated memory type. See the ExtMemFrameType enumerator for details. Used for better integration of 3rd party plugins into the pipeline.

*mfxU16* **PitchHigh**

Distance in bytes between the start of two consecutive rows in a frame.

*mfxU64* **TimeStamp**

Time stamp of the video frame in units of 90KHz. Divide TimeStamp by 90,000 (90 KHz) to obtain the time in seconds. A value of MFX_TIMESTAMP_UNKNOWN indicates that there is no time stamp.

*mfxU32* **FrameOrder**

Current frame counter for the top field of the current frame. An invalid value of MFX_FRAMEORDER_UNKNOWN indicates that API functions that generate the frame output do not use this frame.

*mfxU16* **Locked**

Counter flag for the application. If Locked is greater than zero then the application locks the frame or field pair. Do not move, alter or delete the frame.

### Color Planes

Data pointers to corresponding color channels (planes). The frame buffer pointers must be 16-byte aligned. The application has to specify pointers to all color channels even for packed formats. For example, for YUY2 format the application must specify Y, U, and V pointers. For RGB32 format, the application must specify R, G, B, and A pointers.

*mfxU8* ***A**

A channel.

*mfxMemId* **MemId**

Memory ID of the data buffers. Ignored if any of the preceding data pointers is non-zero.

### Additional Flags

*mfxU16* **Corrupted**

Some part of the frame or field pair is corrupted. See the Corruption enumerator for details.

*mfxU16* **DataFlag**

Additional flags to indicate frame data properties. See the FrameDataFlag enumerator for details.

### Public Members

*mfxExtBuffer* ****ExtParam**

Points to an array of pointers to the extra configuration structures. See the ExtendedBufferID enumerator for a list of extended configurations.

*mfxU16* **PitchLow**

Distance in bytes between the start of two consecutive rows in a frame.

*mfxU8* ***Y**

Y channel.

*mfxU16* ***Y16**

Y16 channel.

*mfxU8* ***R**

R channel.

*mfxU8* ***UV***

UV channel for UV merged formats.

*mfxU8* ***VU***

YU channel for VU merged formats.

*mfxU8* ***CbCr***

CbCr channel for CbCr merged formats.

*mfxU8* ***CrCb***

CrCb channel for CrCb merged formats.

*mfxU8* ***Cb***

Cb channel.

*mfxU8* ***U***

U channel.

*mfxU16* ***U16***

U16 channel.

*mfxU8* ***G***

G channel.

*mfxY410* ***Y410***

T410 channel for Y410 format (merged AVYU).

*mfxY416* ***Y416***

This format is a packed 16-bit representation that includes 16 bits of alpha.

*mfxU8* ***Cr***

Cr channel.

*mfxU8* ***V***

V channel.

*mfxU16* ***V16***

V16 channel.

*mfxU8* ***B***

B channel.

*mfxA2RGB10* ***A2RGB10***

A2RGB10 channel for A2RGB10 format (merged ARGB).

mfxABGR16FP ***ABGRFP16**

ABGRFP16 channel for half float ARGB format (use this merged one due to no separate FP16 Alpha Channel).

## mfxFrameInfo

struct **mfxFrameInfo**

Specifies properties of video frames. See also "Configuration Parameter Constraints" chapter.

### FrameRate

Specify the frame rate with the following formula: FrameRateExtN / FrameRateExtD.

For encoding, frame rate must be specified. For decoding, frame rate may be unspecified (FrameRateExtN and FrameRateExtD are all zeros.) In this case, the frame rate is defaulted to 30 frames per second.

*mfxU32* **FrameRateExtN**

Frame rate numerator.

*mfxU32* **FrameRateExtD**

Frame rate denominator.

### AspectRatio

AspectRatioW and AspectRatioH are used to specify the sample aspect ratio. If sample aspect ratio is explicitly defined by the standards (see Table 6-3 in the MPEG-2 specification or Table E-1 in the H.264 specification), AspectRatioW and AspectRatioH should be the defined values. Otherwise, the sample aspect ratio can be derived as follows:

- `AspectRatioW=display_aspect_ratio_width*display_height`
- `AspectRatioH=display_aspect_ratio_height*display_width`

For MPEG-2, the above display aspect ratio must be one of the defined values in Table 6-3 in the MPEG-2 specification. For H.264, there is no restriction on display aspect ratio values.

If both parameters are zero, the encoder uses the default value of sample aspect ratio.

*mfxU16* **AspectRatioW**

Aspect Ratio for width.

*mfxU16* **AspectRatioH**

Aspect Ratio for height.

### ROI

The region of interest of the frame. Specify the display width and height in *mfxVideoParam*.

*mfxU16* **CropX**

X coordinate. In case of fused operation of decode plus VPP it can be set to zero to signalize that cropping operation is not requested.

*mfxU16* **CropY**

Y coordinate. In case of fused operation of decode plus VPP it can be set to zero to signalize that cropping operation is not requested.

*mfxU16* **CropW**

Width in pixels. In case of fused operation of decode plus VPP it can be set to zero to signalize that cropping operation is not requested.

*mfxU16* **CropH**

Height in pixels. In case of fused operation of decode plus VPP it can be set to zero to signalize that cropping operation is not requested.

### Public Members

*mfxU32* **reserved**[4]

Reserved for future use.

*mfxU16* **ChannelId**

The unique ID of each VPP channel set by application. It's required that during Init/Reset application fills ChannelId for each *mfxVideoChannelParam* provided by the application and the SDK sets it back to the correspondent mfxSurfaceArray::mfxFrameSurface1 to distinguish different channels. It's expected that surfaces for some channels might be returned with some delay so application has to use *mfxFrameInfo::ChannelId* to distinguish what returned surface belongs to what VPP channel. Decoder's initialization parameters are always sent through channel with *mfxFrameInfo::ChannelId* equals to zero. It's allowed to skip setting of decoder's parameters for simplified decoding procedure

*mfxU16* **BitDepthLuma**

Number of bits used to represent luma samples.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU16* **BitDepthChroma**

Number of bits used to represent chroma samples.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU16* **Shift**

When the value is not zero, indicates that values of luma and chroma samples are shifted. Use BitDepthLuma and BitDepthChroma to calculate shift size. Use zero value to indicate absence of shift. See example data alignment below.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxFrameId* **FrameId**

Describes the view and layer of a frame picture.

*mfxU32* **FourCC**

FourCC code of the color format. See the ColorFourCC enumerator for details.

*mfxU16* **Width**

Width of the video frame in pixels. Must be a multiple of 16. In case of fused operation of decode plus VPP it can be set to zero to signalize that scaling operation is not requested.

*mfxU16* **Height**

Height of the video frame in pixels. Must be a multiple of 16 for progressive frame sequence and a multiple of 32 otherwise. In case of fused operation of decode plus VPP it can be set to zero to signalize that scaling operation is not requested.

*mfxU64* **BufferSize**

Size of frame buffer in bytes. Valid only for plain formats (when FourCC is P8). In this case, Width, Height, and crop values are invalid.

*mfxU16* **PicStruct**

Picture type as specified in the PicStruct enumerator.

*mfxU16* **ChromaFormat**

Color sampling method. Value is the same as that of ChromaFormatIdc. ChromaFormat is not defined if FourCC is zero.

**Note:** Example data alignment for Shift = 0:

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | 0 | 0 | 0 | 0 | 0 | 0 | Valid data | | | | | | | | | |

Example data alignment for Shift != 0:

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | Valid data | | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 |

---

### mfxFrameSurface1

struct **mfxFrameSurface1**

Defines the uncompressed frames surface information and data buffers. The frame surface is in the frame or complementary field pairs of pixels up to four color-channels, in two parts: *mfxFrameInfo* and *mfxFrameData*.

#### Public Members

struct *mfxFrameSurfaceInterface* ***FrameInterface**

Specifies interface to work with surface.

*mfxFrameInfo* **Info**

Specifies surface properties.

*mfxFrameData* **Data**

Describes the actual frame buffer.

### mfxFrameSurfaceInterface

struct **mfxFrameSurfaceInterface**

### Public Members

*mfxHDL* **Context**

The context of the memory interface. User should not touch (change, set, null) this pointer.

*mfxStructVersion* **Version**

The version of the structure.

*mfxStatus* (***AddRef**)(*mfxFrameSurface1* *surface)

Increments the internal reference counter of the surface. The surface is not destroyed until the surface is released using the *mfxFrameSurfaceInterface::Release* function. *mfxFrameSurfaceInterface::AddRef* should be used each time a new link to the surface is created (for example, copy structure) for proper surface management.

**Param surface**
**[in]** Valid surface.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Release**)(*mfxFrameSurface1* *surface)

Decrements the internal reference counter of the surface. *mfxFrameSurfaceInterface::Release* should be called after using the *mfxFrameSurfaceInterface::AddRef* function to add a surface or when allocation logic requires it. For example, call *mfxFrameSurfaceInterface::Release* to release a surface obtained with the GetSurfaceForXXX function.

**Param surface**
**[in]** Valid surface.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNDEFINED_BEHAVIOR If Reference Counter of surface is zero before call.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetRefCounter**)(*mfxFrameSurface1* *surface, *mfxU32* *counter)

Returns current reference counter of *mfxFrameSurface1* structure.

**Param surface**
**[in]** Valid surface.

**Param counter**
**[out]** Sets counter to the current reference counter value.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface or counter is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Map**)(*mfxFrameSurface1* *surface, *mfxU32* flags)

Sets pointers of surface->Info.Data to actual pixel data, providing read-write access.

In case of video memory, the surface with data in video memory becomes mapped to system memory. An application can map a surface for read access with any value of mfxFrameSurface1::Data::Locked, but can map a surface for write access only when mfxFrameSurface1::Data::Locked equals to 0.

Note: A surface allows shared read access, but exclusive write access. Consider the following cases:

- Map with Write or Read|Write flags. A request during active another read or write access returns MFX_ERR_LOCK_MEMORY error immediately, without waiting. MFX_MAP_NOWAIT does not impact behavior. This type of request does not lead to any implicit synchronizations.
- Map with Read flag. A request during active write access will wait for resource to become free, or exits immediately with error if MFX_MAP_NOWAIT flag was set. This request may lead to the implicit synchronization (with same logic as Synchronize call) waiting for surface to become ready to use (all dependencies should be resolved and upstream components finished writing to this surface).

It is guaranteed that read access will be acquired right after synchronization without allowing another thread to acquire this surface for writing.

If MFX_MAP_NOWAIT was set and the surface is not ready yet (for example the surface has unresolved data dependencies or active processing), the read access request exits immediately with error.

Read-write access with MFX_MAP_READ_WRITE provides exclusive simultaneous reading and writing access.

**Note:** Bitwise copying of *mfxFrameSurface1* object between map / unmap calls may result in having dangling data pointers in copies.

**Param surface**
**[in]** Valid surface.

**Param flags**
**[out]** Specify mapping mode.

**Param surface->Info.Data**
**[out]** Pointers set to actual pixel data.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNSUPPORTED If flags are invalid.

MFX_ERR_LOCK_MEMORY If user wants to map the surface for write and surface->Data.Locked does not equal to 0.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Unmap**)(*mfxFrameSurface1* *surface)

Invalidates pointers of surface->Info.Data and sets them to NULL. In case of video memory, the underlying texture becomes unmapped after last reader or writer unmap.

**Param surface**
**[in]** Valid surface.

**Param surface->Info.Data**
**[out]** Pointers set to NULL.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNSUPPORTED If surface is already unmapped.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetNativeHandle**)(*mfxFrameSurface1* *surface, *mfxHDL* *resource, *mfxResourceType* *resource_type)

Returns a native resource's handle and type. The handle is returned *as-is*, meaning that the reference counter of base resources is not incremented. The native resource is not detached from surface and the library still owns the resource. User must not destroy the native resource or assume that the resource will be alive after *mfxFrameSurfaceInterface::Release*.

**Param surface**
**[in]** Valid surface.

**Param resource**
**[out]** Pointer is set to the native handle of the resource.

**Param resource_type**
**[out]** Type of native resource. See mfxResourceType enumeration).

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If any of surface, resource or resource_type is NULL.

MFX_ERR_INVALID_HANDLE If any of surface, resource or resource_type is not valid object (no native resource was allocated).

MFX_ERR_UNSUPPORTED If surface is in system memory.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetDeviceHandle**)(*mfxFrameSurface1* *surface, *mfxHDL* *device_handle, *mfxHandleType* *device_type)

Returns a device abstraction that was used to create that resource. The handle is returned *as-is*, meaning that the reference counter for the device abstraction is not incremented. The native resource is not detached from the surface and the library still has a reference to the resource. User must not destroy the device or assume that the device will be alive after *mfxFrameSurfaceInterface::Release*.

**Param surface**
**[in]** Valid surface.

**Param device_handle**
**[out]** Pointer is set to the device which created the resource

**Param device_type**
**[out]** Type of device (see mfxHandleType enumeration).

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If any of surface, device_handle or device_type is NULL.

MFX_ERR_INVALID_HANDLE If any of surface, resource or resource_type is not valid object (no native resource was allocated).

MFX_ERR_UNSUPPORTED If surface is in system memory.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Synchronize**)(*mfxFrameSurface1* *surface, *mfxU32* wait)

Guarantees readiness of both the data (pixels) and any frame's meta information (for example corruption flags) after a function completes.

Instead of MFXVideoCORE_SyncOperation, users may directly call the *mfxFrameSurfaceInterface::Synchronize* function after the corresponding Decode or VPP function calls (MFXVideoDECODE_DecodeFrameAsync or MFXVideoVPP_RunFrameVPPAsync). The prerequisites to call the functions are:

- The main processing functions return MFX_ERR_NONE.
- A valid *mfxFrameSurface1* object.

**Param surface**
**[in]** Valid surface.

**Param wait**
**[out]** Wait time in milliseconds.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If any of surface is not valid object .

MFX_WRN_IN_EXECUTION If the given timeout is expired and the surface is not ready.

MFX_ERR_ABORTED If the specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

MFX_ERR_UNKNOWN Any internal error.

void (*OnComplete)(*mfxStatus* sts)

The library calls the function after complete of associated video operation notifying the application that frame surface is ready.

It is expected that the function is low-intrusive designed otherwise it may impact performance.

**Attention**
This is callback function and intended to be called by the library only.

**Note:** The library calls this callback only when this surface is used as the output surface.

**Param sts**
**[in]** The status of completed operation.

*mfxStatus* (*QueryInterface)(*mfxFrameSurface1* *surface, *mfxGUID* guid, *mfxHDL* *iface)

Returns an interface defined by the GUID. If the returned interface is a reference counted object the caller should release the obtained interface to avoid memory leaks.

**Param surface**
**[in]** Valid surface.

**Param guid**
**[in]** GUID of the requested interface.

**Param iface**
**[out]** Interface.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If interface or surface is NULL.

MFX_ERR_UNSUPPORTED If requested interface is not supported.

MFX_ERR_NOT_IMPLEMENTED If requested interface is not implemented.

MFX_ERR_NOT_INITIALIZED If requested interface is not available (not created or already deleted).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (*Export)(*mfxFrameSurface1* *surface, *mfxSurfaceHeader* export_header, *mfxSurfaceHeader* **exported_surface)

If successful returns an exported surface, which is a refcounted object allocated by runtime. It could be exported with or without copy, depending on export flags and the possibility of such export. Exported surface is valid throughout the session, as long as the original *mfxFrameSurface1* object is not closed and the refcount of exported surface is not zero.

**Param surface**
**[in]** Valid surface.

**Param export_header**
**[in]** Description of export: caller should fill in SurfaceType (type to export to) and SurfaceFlags (allowed export modes).

**Param exported_surface**
**[out]** Exported surface, allocated by runtime, user needs to decrement refcount after usage for object release. After successful export, the value of *mfxSurfaceHeader::SurfaceFlags* will contain the actual export mode.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If export surface or surface is NULL.

MFX_ERR_UNSUPPORTED If requested export is not supported.

MFX_ERR_NOT_IMPLEMENTED If requested export is not implemented.

MFX_ERR_UNKNOWN Any internal error.

### mfxSurfacePoolInterface

struct **mfxSurfacePoolInterface**

Specifies the surface pool interface.

#### Public Members

*mfxHDL* **Context**

The context of the surface pool interface. User should not touch (change, set, null) this pointer.

*mfxStatus* (***AddRef**)(struct *mfxSurfacePoolInterface* *pool)

Increments the internal reference counter of the *mfxSurfacePoolInterface*. The *mfxSurfacePoolInterface* is not destroyed until the *mfxSurfacePoolInterface* is destroyed with *mfxSurfacePoolInterface::Release* function. *mfxSurfacePoolInterface::AddRef* should be used each time a new link to the *mfxSurfacePoolInterface* is created for proper management.

**Param pool**
**[in]** Valid pool.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If pool is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfacePoolInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Release**)(struct *mfxSurfacePoolInterface* *pool)

Decrements the internal reference counter of the *mfxSurfacePoolInterface*. *mfxSurfacePoolInterface::Release* should be called after using the *mfxSurfacePoolInterface::AddRef* function to add a *mfxSurfacePoolInterface* or when allocation logic requires it. For example, call *mfxSurfacePoolInterface::Release* to release a *mfxSurfacePoolInterface* obtained with the *mfxFrameSurfaceInterface::QueryInterface* function.

**Param pool**
**[in]** Valid pool.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If pool is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfacePoolInterface->Context is invalid (for example NULL).

MFX_ERR_UNDEFINED_BEHAVIOR If Reference Counter of *mfxSurfacePoolInterface* is zero before call.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetRefCounter**)(struct *mfxSurfacePoolInterface* *pool, *mfxU32* *counter)

Returns current reference counter of *mfxSurfacePoolInterface* structure.

**Param pool**
**[in]** Valid pool.

**Param counter**
**[out]** Sets counter to the current reference counter value.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If pool or counter is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfacePoolInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***SetNumSurfaces**)(struct *mfxSurfacePoolInterface* *pool, *mfxU32* num_surfaces)

The function should be called by oneAPI Video Processing Library (oneVPL) components or application to specify how many surfaces it will use concurrently. Internally, oneVPL allocates surfaces in the shared pool according to the component's policy set by mfxPoolAllocationPolicy. The exact moment of surfaces allocation is defined by the component and generally independent from that call.

**Param pool**
**[in]** Valid pool.

**Param num_surfaces**
**[in]** The number of surfaces required by the component.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If pool is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfacePoolInterface->Context is invalid (for example NULL).

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM If pool has MFX_ALLOCATION_UNLIMITED or MFX_ALLOCATION_LIMITED policy.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***RevokeSurfaces**)(struct *mfxSurfacePoolInterface* *pool, *mfxU32* num_surfaces)

The function should be called by oneVPL components when component is closed or reset and doesn't need to use pool more. It helps to manage memory accordingly and release redundant memory. Important to specify the same number of surfaces which is requested during SetNumSurfaces call, otherwise it may lead to the pipeline stalls.

**Param pool**
**[in]** Valid pool.

**Param num_surfaces**
**[in]** The number of surfaces used by the component.

**Return**

MFX_ERR_NONE If no error.

MFX_WRN_OUT_OF_RANGE If num_surfaces doesn't equal to num_surfaces requested during SetNumSurfaces call.

MFX_ERR_NULL_PTR If pool is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfacePoolInterface->Context is invalid (for example NULL).

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM If pool has MFX_ALLOCATION_UNLIMITED or MFX_ALLOCATION_LIMITED policy.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetAllocationPolicy**)(struct *mfxSurfacePoolInterface* *pool, *mfxPoolAllocationPolicy* *policy)

Returns current allocation policy.

**Param pool**
**[in]** Valid pool.

**Param policy**
**[out]** Sets policy to the current allocation policy value.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If pool or policy is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfacePoolInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetMaximumPoolSize**)(struct *mfxSurfacePoolInterface* *pool, *mfxU32* *size)

Returns maximum pool size. In case of mfxPoolAllocationPolicy::MFX_ALLOCATION_UNLIMITED policy 0xFFFFFFFF will be returned.

**Param pool**
**[in]** Valid pool.

**Param size**
**[out]** Sets size to the maximum pool size value.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If pool or size is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfacePoolInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (*`GetCurrentPoolSize`)(struct *mfxSurfacePoolInterface* *pool, *mfxU32* *size)

Returns current pool size.

**Param pool**
**[in]** Valid pool.

**Param size**
**[out]** Sets size to the current pool size value.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If pool or size is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfacePoolInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxHDL* `reserved`[4]

Reserved for future use.

### mfxMemoryInterface

struct `mfxMemoryInterface`

#### Public Members

*mfxHDL* `Context`

The context of the memory interface. User should not touch (change, set, null) this pointer.

*mfxStructVersion* `Version`

The version of the structure.

*mfxStatus* (*`ImportFrameSurface`)(struct *mfxMemoryInterface* *memory_interface, *mfxSurfaceComponent* surf_component, *mfxSurfaceHeader* *external_surface, *mfxFrameSurface1* **imported_surface)

Imports an application-provided surface into *mfxFrameSurface1* which may be used as input for encoding or video processing.

**Since**
This function is available since API version 2.10.

**Param memory_interface**
**[in]** Valid memory interface.

**Param surf_component**
**[in]** Surface component type. Required for allocating new surfaces from the appropriate pool.

**Param external_surface**
**[inout]** Pointer to the mfxSurfaceXXX object describing the surface to be imported. All fields in *mfxSurfaceHeader* must be set by the application. *mfxSurfaceHeader::SurfaceType* is read by VPL runtime to determine which particular mfxSurfaceXXX structure is supplied. For example, if mfxSurfaceXXX::SurfaceType == MFX_SURFACE_TYPE_D3D11_TEX2D, then the handle will be interpreted as an object of type *mfxSurfaceD3D11Tex2D*. The application should set or clear other fields as specified in the corresponding structure description. After successful import, the value of *mfxSurfaceHeader::SurfaceFlags* will be replaced with the actual import type. It can be used to determine which import type (with or without copy) took place in the case of initial default setting, or if multiple import flags were OR'ed. All external sync operations on the ext_surface must be completed before calling this function.

**Param imported_surface**
**[out]** Pointer to a valid *mfxFrameSurface1* object containing the imported frame. imported_surface may be passed as an input to Encode or VPP processing operations.

**Return**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR If ext_surface or imported_surface are NULL.

MFX_ERR_INVALID_HANDLE If the corresponding session was not initialized.

MFX_ERR_UNSUPPORTED If surf_component is not one of [MFX_SURFACE_COMPONENT_ENCODE, MFX_SURFACE_COMPONENT_VPP_INPUT], or if *mfxSurfaceHeader::SurfaceType* is not supported by VPL runtime for this operation.

### mfxSurfaceTypesSupported

struct **mfxSurfaceTypesSupported**

This structure describes the supported surface types and modes.

#### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **NumSurfaceTypes**

Number of supported surface types.

*mfxU32* **reserved**[4]

Reserved for future use.

struct **surftype**

#### Public Members

*mfxSurfaceType* **SurfaceType**

Supported surface type.

*mfxU32* **reserved**[6]

Reserved for future use.

*mfxU16* **NumSurfaceComponents**

Number of supported surface components.

struct **surfcomp**

#### Public Members

*mfxSurfaceComponent* **SurfaceComponent**

Supported surface component.

*mfxU32* **SurfaceFlags**

Supported surface flags for this component (may be OR'd).

*mfxU32* **reserved**[7]

Reserved for future use.

### mfxSurfaceHeader

struct **mfxSurfaceHeader**

#### Public Members

*mfxSurfaceType* **SurfaceType**

Set to the MFX_SURFACE_TYPE enum corresponding to the specific structure.

*mfxU32* **SurfaceFlags**

Set to the MFX_SURFACE_FLAG enum (or combination) corresponding to the allowed import / export mode(s). Multiple flags may be combined with OR. Upon a successful Import or Export operation, this field will indicate the actual mode used.

*mfxU32* **StructSize**

Size in bytes of the complete mfxSurfaceXXX structure.

*mfxU16* **NumExtParam**

The number of extra configuration structures attached to the structure.

*mfxExtBuffer* ****ExtParam**

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

### mfxSurfaceInterface

struct **mfxSurfaceInterface**

Contains *mfxSurfaceHeader* and the callback functions AddRef, Release and GetRefCounter that the application may use to manage access to exported surfaces. These interfaces are only valid for surfaces obtained by *mfxFrameSurfaceInterface::Export*. They are not used for surface descriptions passed to function *mfxMemoryInterface::ImportFrameSurface*.

#### Public Members

*mfxSurfaceHeader* **Header**

Exported surface header. Contains description of current surface.

*mfxStructVersion* **Version**

The version of the structure.

*mfxHDL* **Context**

The context of the exported surface interface. User should not touch (change, set, null) this pointer.

*mfxStatus* (***AddRef**)(struct *mfxSurfaceInterface* *surface)

Increments the internal reference counter of the surface. The surface is not destroyed until the surface is released using the *mfxSurfaceInterface::Release* function. *mfxSurfaceInterface::AddRef* should be used each time a new link to the surface is created (for example, copy structure) for proper surface management.

**Param surface**
**[in]** Valid surface.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Release**)(struct *mfxSurfaceInterface* *surface)

Decrements the internal reference counter of the surface. *mfxSurfaceInterface::Release* should be called after using the *mfxSurfaceInterface::AddRef* function to add a surface or when allocation logic requires it. For example, call *mfxSurfaceInterface::Release* to release a surface obtained with the *mfxFrameSurfaceInterface::Export* function.

**Param surface**
**[in]** Valid surface.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNDEFINED_BEHAVIOR If Reference Counter of surface is zero before call.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetRefCounter**)(struct *mfxSurfaceInterface* *surface, *mfxU32* *counter)

Returns current reference counter of exported surface.

**Param surface**
**[in]** Valid surface.

**Param counter**
**[out]** Sets counter to the current reference counter value.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface or counter is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Synchronize**)(struct *mfxSurfaceInterface* *surface, *mfxU32* wait)

This function is only valuable for surfaces which were exported in sharing mode (without a copy). Guarantees readiness of both the data (pixels) and any original *mfxFrameSurface1* frame's meta information (for example corruption flags) after a function completes.

Instead of MFXVideoCORE_SyncOperation, users may directly call the *mfxSurfaceInterface::Synchronize* function after the corresponding Decode or VPP function calls (MFXVideoDECODE_DecodeFrameAsync or MFXVideoVPP_RunFrameVPPAsync). The prerequisites to call the functions are:

- The main processing functions return MFX_ERR_NONE.
- A valid surface object.

**Param surface**
**[in]** Valid surface.

**Param wait**
**[out]** Wait time in milliseconds.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If any of surface is not valid object .

MFX_WRN_IN_EXECUTION If the given timeout is expired and the surface is not ready.

MFX_ERR_ABORTED If the specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

MFX_ERR_UNKNOWN Any internal error.

### mfxSurfaceD3D11Tex2D

struct **mfxSurfaceD3D11Tex2D**

#### Public Members

*mfxHDL* **texture2D**

Pointer to texture, type ID3D11Texture2D

### mfxSurfaceVAAPI

struct **mfxSurfaceVAAPI**

#### Public Members

*mfxHDL* **vaDisplay**

Object of type VADisplay.

*mfxU32* **vaSurfaceID**

Object of type VASurfaceID.

### mfxSurfaceOpenCLImg2D

struct **mfxSurfaceOpenCLImg2D**

#### Public Members

*mfxHDL* **ocl_context**

Pointer to OpenCL context, type cl_context

*mfxHDL* **ocl_command_queue**

Pointer to OpenCL command queue, type cl_command_queue

*mfxHDL* **ocl_image**[4]

Pointer to OpenCL 2D images, type cl_mem *mfxU32* **ocl_image_num**

Number of valid images (planes), depends on color format

### mfxExtSurfaceOpenCLImg2DExportDescription

struct **mfxExtSurfaceOpenCLImg2DExportDescription**

Optional extension buffer, which can be attached to *mfxSurfaceHeader::ExtParam* (second parameter of *mfxFrameSurfaceInterface::Export*) in order to pass OCL parameters during *mfxFrameSurface1* exporting to OCL surface. If buffer is not provided all resources will be created by VPL RT internally.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_EXPORT_SHARING_DESC_OCL.

*mfxHDL* **ocl_context**

Pointer to OpenCL context, type cl_context

*mfxHDL* **ocl_command_queue**

Pointer to OpenCL command queue, type cl_command_queue

## 5.2.3 Implementation Management

Structures used for implementation management.

### API

- *mfxAdapterInfo*
- *mfxAdaptersInfo*
- *mfxExtThreadsParam*
- *mfxInitParam*
- *mfxPlatform*
- *mfxVersion*
- *mfxExtDeviceAffinityMask*
- *mfxInitializationParam*
- *mfxAutoSelectImplDeviceHandle*

### mfxAdapterInfo

struct **mfxAdapterInfo**

Contains a description of the graphics adapter for the Legacy mode.

#### Public Members

*mfxPlatform* **Platform**

Platform type description. See *mfxPlatform* for details.

*mfxU32* **Number**

Value which uniquely characterizes media adapter. On Windows* this number can be used for initialization through DXVA interface (see example).

### mfxAdaptersInfo

struct **mfxAdaptersInfo**

Contains description of all graphics adapters available on the current system.

#### Public Members

*mfxAdapterInfo* ***Adapters**

Pointer to array of *mfxAdapterInfo* structs allocated by user.

*mfxU32* **NumAlloc**

Length of Adapters array.

*mfxU32* **NumActual**

Number of Adapters entries filled by MFXQueryAdapters.

### mfxExtThreadsParam

struct **mfxExtThreadsParam**

Specifies options for threads created by this session. Attached to the *mfxInitParam* structure during legacy Intel(r) Media SDK session initialization or to *mfxInitializationParam* by the dispatcher in MFXCreateSession function.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_THREADS_PARAM.

*mfxU16* **NumThread**

The number of threads.

*mfxI32* **SchedulingType**

Scheduling policy for all threads.

*mfxI32* **Priority**

Priority for all threads.

*mfxU16* **reserved**[55]

Reserved for future use.

### mfxInitParam

struct **mfxInitParam**

Specifies advanced initialization parameters. A zero value in any of the fields indicates that the corresponding field is not explicitly specified.

#### Public Members

*mfxIMPL* **Implementation**

Enumerator that indicates the desired legacy Intel(r) Media SDK implementation.

*mfxVersion* **Version**

Structure which specifies minimum library version or zero, if not specified.

*mfxU16* **ExternalThreads**

Desired threading mode. Value 0 means internal threading, 1 - external.

*mfxExtBuffer* ****ExtParam**

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

*mfxU16* **NumExtParam**

The number of extra configuration structures attached to this structure.

*mfxU16* **GPUCopy**

Enables or disables GPU accelerated copying between video and system memory in legacy Intel(r) Media SDK components. See the GPUCopy enumerator for a list of valid values.

### mfxPlatform

struct **mfxPlatform**

Contains information about hardware platform for the Legacy mode.

#### Public Members

*mfxU16* **CodeName**

Deprecated.

*mfxU16* **DeviceId**

Unique identifier of graphics device.

*mfxU16* **MediaAdapterType**

Description of graphics adapter type. See the mfxMediaAdapterType enumerator for a list of possible values.

*mfxU16* **reserved**[13]

Reserved for future use.

### mfxVersion

union **mfxVersion**

*#include <mfxcommon.h>* The *mfxVersion* union describes the version of the implementation.

#### Major and Minor fields

Anonymous structure with Major and Minor fields.

*mfxU16* **Minor**

Minor number of the implementation.

*mfxU16* **Major**

Major number of the implementation.

#### Public Members

struct *mfxVersion*::[anonymous] **[anonymous]**

*mfxU32* **Version**

Implementation version number.

### mfxExtDeviceAffinityMask

struct **mfxExtDeviceAffinityMask**

The *mfxExtDeviceAffinityMask* structure is used by the application to specify affinity mask for the device with given device ID. See *mfxDeviceDescription* for the device ID definition and sub device indexes. If the implementation manages CPU threads for some purpose, the user can set the CPU thread affinity mask by using this structure with DeviceID set to "CPU".

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DEVICE_AFFINITY_MASK.

*mfxChar* **DeviceID**[MFX_STRFIELD_LEN]

Null terminated string with device ID. In case of CPU affinity mask it must be equal to "CPU".

*mfxU32* **NumSubDevices**

Number of sub devices or threads in case of CPU in the mask.

*mfxU8* ***Mask**

Mask array. Every bit represents sub-device (or thread for CPU). "1" means execution is allowed. "0" means that execution is prohibited on this sub-device (or thread). Length of the array is equal to the: "NumSubDevices / 8" and rounded to the closest (from the right) integer. Bits order within each entry of the mask array is LSB: bit 0 holds data for sub device with index 0 and bit 8 for sub device with index 8. Index of sub device is defined by the *mfxDeviceDescription* structure.

### mfxInitializationParam

struct **mfxInitializationParam**

Specifies initialization parameters for API version starting from 2.0.

#### Public Members

*mfxAccelerationMode* **AccelerationMode**

Hardware acceleration stack to use. OS dependent parameter. Use VA for Linux*, DX* for Windows* or HDDL.

*mfxU16* **DeviceCopy**

Enables or disables device's accelerated copying between device and host. See the GPUCopy enumerator for a list of valid values. This parameter is the equivalent of *mfxInitParam::GPUCopy*.

*mfxU16* **reserved**[2]

Reserved for future use.

*mfxU16* **NumExtParam**

The number of extra configuration structures attached to this structure.

*mfxExtBuffer* ****ExtParam**

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

*mfxU32* **VendorImplID**

Vendor specific number with given implementation ID. Represents the same field from *mfxImplDescription*.

*mfxU32* **reserved2**[3]

Reserved for future use.

### mfxAutoSelectImplDeviceHandle

struct **mfxAutoSelectImplDeviceHandle**

Specifies that an implementation should be selected which matches the device handle provided by the application.

#### Public Members

*mfxAutoSelectImplType* **AutoSelectImplType**

Must be set to MFX_AUTO_SELECT_IMPL_TYPE_DEVICE_HANDLE.

*mfxAccelerationMode* **AccelMode**

Hardware acceleration mode of provided device handle.

*mfxHandleType* **DeviceHandleType**

Type of provided device handle.

*mfxHDL* **DeviceHandle**

System handle to hardware device.

*mfxU16* **reserved**[8]

Reserved for future use.

## 5.2.4 Cross-component Structures

Structures used across library components.

### API

- *mfxComponentInfo*
- *mfxExtHEVCParam*
- *mfxExtJPEGHuffmanTables*
- *mfxExtJPEGQuantTables*
- *mfxExtMVCSeqDesc*
- *mfxExtMVCTargetViews*
- *mfxExtVideoSignalInfo*
- *mfxExtVP9Param*
- *mfxFrameId*
- *mfxInfoMFX*
- *mfxMVCOperationPoint*
- *mfxMVCViewDependency*
- *mfxPayload*
- *mfxVideoParam*
- *mfxVP9SegmentParam*
- *mfxExtAV1FilmGrainParam*
- *mfxAV1FilmGrainPoint*
- *mfxRect*
- *mfxExtHyperModeParam*
- *mfxGUID*
- *mfxExtAllocationHints*
- *mfxRefInterface*
- *mfxExtMasteringDisplayColourVolume*
- *mfxExtContentLightLevelInfo*
- *mfxExtSyncSubmission*
- *mfxExtTuneEncodeQuality*
- *mfxConfigInterface*

### mfxComponentInfo

struct **mfxComponentInfo**

Contains workload description, which is accepted by MFXQueryAdapters function.

#### Public Members

*mfxComponentType* **Type**

Type of workload: Encode, Decode, VPP. See mfxComponentType enumerator for values.

*mfxVideoParam* **Requirements**

Detailed description of workload. See *mfxVideoParam* for details.

### mfxExtHEVCParam

struct **mfxExtHEVCParam**

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_HEVC_PARAM.

*mfxU16* **PicWidthInLumaSamples**

Specifies the width of each coded picture in units of luma samples.

*mfxU16* **PicHeightInLumaSamples**

Specifies the height of each coded picture in units of luma samples.

*mfxU64* **GeneralConstraintFlags**

Additional flags to specify exact profile and constraints. See the GeneralConstraintFlags enumerator for values of this field.

*mfxU16* **SampleAdaptiveOffset**

Controls SampleAdaptiveOffset encoding feature. See the SampleAdaptiveOffset enumerator for supported values (bit-ORed). Valid during encoder Init and Runtime.

*mfxU16* **LCUSize**

Specifies largest coding unit size (max luma coding block). Valid during encoder Init.

### mfxExtJPEGHuffmanTables

struct **mfxExtJPEGHuffmanTables**

Specifies Huffman tables. The application may specify up to 2 quantization table pairs for baseline process. The encoder assigns an ID to each table. That ID is equal to the table index in the DCTables and ACTables arrays. Table "0" is used for encoding of the Y component and table "1" is used for encoding of the U and V component. The application may specify only one table, in which case the table will be used for all components in the image. The following table illustrates this behavior.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_JPEG_HUFFMAN.

*mfxU16* **NumDCTable**

Number of DC quantization table in DCTables array.

*mfxU16* **NumACTable**

Number of AC quantization table in ACTables array.

*mfxU8* **Bits**[16]

Number of codes for each code length.

*mfxU8* **Values**[12]

List of the 8-bit symbol values.

Array of AC tables.

struct *mfxExtJPEGHuffmanTables*::[anonymous] **DCTables**[4]

Array of DC tables.

struct *mfxExtJPEGHuffmanTables*::[anonymous] **ACTables**[4]

List of the 8-bit symbol values.

| Table ID<br>Number of tables | 0 | 1 |
|---|---|---|
| 0 | Y, U, V | |
| 1 | Y | U, V |

## mfxExtJPEGQuantTables

struct **mfxExtJPEGQuantTables**

Specifies quantization tables. The application may specify up to 4 quantization tables. The encoder assigns an ID to each table. That ID is equal to the table index in the Qm array. Table "0" is used for encoding of the Y component, table "1" for the U component, and table "2" for the V component. The application may specify fewer tables than the number of components in the image. If two tables are specified, then table "1" is used for both U and V components. If only one table is specified then it is used for all components in the image. The following table illustrates this behavior.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_JPEG_QT.

*mfxU16* **NumTable**

Number of quantization tables defined in Qm array.

*mfxU16* **Qm**[4][64]

Quantization table values.

| Table ID<br>Number of tables | 0 | 1 | 2 |
|---|---|---|---|
| 0 | Y, U, V | | |
| 1 | Y | U, V | |
| 2 | Y | U | V |

## mfxExtMVCSeqDesc

struct **mfxExtMVCSeqDesc**

Describes the MVC stream information of view dependencies, view identifiers, and operation points. See the ITU*-T H.264 specification chapter H.7.3.2.1.4 for details.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MVC_SEQUENCE_DESCRIPTION.

*mfxU32* **NumView**

Number of views.

*mfxU32* **NumViewAlloc**

The allocated view dependency array size.

*mfxMVCViewDependency* ***View**

Pointer to a list of the *mfxMVCViewDependency*.

*mfxU32* **NumViewId**

Number of view identifiers.

*mfxU32* **NumViewIdAlloc**

The allocated view identifier array size.

*mfxU16* ***ViewId**

Pointer to the list of view identifier.

*mfxU32* **NumOP**

Number of operation points.

*mfxU32* **NumOPAlloc**

The allocated operation point array size.

*mfxMVCOperationPoint* ***OP**

Pointer to a list of the *mfxMVCOperationPoint* structure.

*mfxU16* **NumRefsTotal**

Total number of reference frames in all views required to decode the stream. This value is returned from the MFXVideoDECODE_Decodeheader function. Do not modify this value.

### mfxExtMVCTargetViews

struct **mfxExtMVCTargetViews**

Configures views for the decoding output.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MVC_TARGET_VIEWS.

*mfxU16* **TemporalId**

The temporal identifier to be decoded.

*mfxU32* **NumView**

The number of views to be decoded.

*mfxU16* **ViewId**[1024]

List of view identifiers to be decoded.

### mfxExtVideoSignalInfo

struct **mfxExtVideoSignalInfo**

Defines the video signal information.

For H.264, see Annex E of the ISO/IEC 14496-10 specification for the definition of these parameters.

For MPEG-2, see section 6.3.6 of the ITU* H.262 specification for the definition of these parameters. The field VideoFullRange is ignored.

For VC-1, see section 6.1.14.5 of the SMPTE* 421M specification. The fields VideoFormat and VideoFullRange are ignored.

**Note:** If ColourDescriptionPresent is zero, the color description information (including ColourPrimaries, TransferCharacteristics, and MatrixCoefficients) does not present in the bitstream.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VIDEO_SIGNAL_INFO.

*mfxU16* **VideoFormat**

*mfxU16* **VideoFullRange**

*mfxU16* **ColourDescriptionPresent**

*mfxU16* **ColourPrimaries**

*mfxU16* **TransferCharacteristics**

*mfxU16* **MatrixCoefficients**

### mfxExtVP9Param

struct **mfxExtVP9Param**

Structure attached to the *mfxVideoParam* structure. Extends the *mfxVideoParam* structure with VP9-specific parameters. Used by both decoder and encoder.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VP9_PARAM.

*mfxU16* **FrameWidth**

Width of the coded frame in pixels.

*mfxU16* **FrameHeight**

Height of the coded frame in pixels.

*mfxU16* **WriteIVFHeaders**

Set this option to ON to make the encoder insert IVF container headers to the output stream. The NumFrame field of the IVF sequence header will be zero. It is the responsibility of the application to update the NumFrame field with the correct value. See the CodingOptionValue enumerator for values of this option.

*mfxI16* **QIndexDeltaLumaDC**

Specifies an offset for a particular quantization parameter.

*mfxI16* **QIndexDeltaChromaAC**

Specifies an offset for a particular quantization parameter.

*mfxI16* **QIndexDeltaChromaDC**

Specifies an offset for a particular quantization parameter.

*mfxU16* **NumTileRows**

Number of tile rows. Should be power of two. The maximum number of tile rows is 4, per the VP9 specification. In addition, the maximum supported number of tile rows may depend on the underlying library implementation.

Use the Query API function to check if a particular pair of values (NumTileRows, NumTileColumns) is supported. In VP9, tile rows have dependencies and cannot be encoded or decoded in parallel. Therefore, tile rows are always encoded by the library in serial mode (one-by-one).

*mfxU16* **NumTileColumns**

Number of tile columns. Should be power of two. Restricted with maximum and minimum tile width in luma pixels, as defined in the VP9 specification (4096 and 256 respectively). In addition, the maximum supported number of tile columns may depend on the underlying library implementation.

Use the Query API function to check if a particular pair of values (NumTileRows, NumTileColumns) is supported. In VP9, tile columns do not have dependencies and can be encoded/decoded in parallel. Therefore, tile columns can be encoded by the library in both parallel and serial modes.

Parallel mode is automatically utilized by the library when NumTileColumns exceeds 1 and does not exceed the number of tile coding engines on the platform. In other cases, serial mode is used. Parallel mode is capable of encoding more than 1 tile row (within limitations provided by VP9 specification and particular platform). Serial mode supports only tile grids 1xN and Nx1.

### mfxFrameId

struct **mfxFrameId**

Describes the view and layer of a frame picture.

#### Public Members

*mfxU16* **TemporalId**

The temporal identifier as defined in the annex H of the ITU*-T H.264 specification.

*mfxU16* **PriorityId**

Reserved and must be zero.

*mfxU16* **DependencyId**

Reserved for future use.

*mfxU16* **QualityId**

Reserved for future use.

*mfxU16* **ViewId**

The view identifier as defined in the annex H of the ITU-T H.264 specification.

### mfxInfoMFX

struct **mfxInfoMFX**

Specifies configurations for decoding, encoding, and transcoding processes. A zero value in any of these fields indicates that the field is not explicitly specified.

#### Public Members

*mfxU32* **reserved**[7]

Reserved for future use.

*mfxU16* **LowPower**

Hint to enable low power consumption mode for encoders. See the CodingOptionValue enumerator for values of this option. Use the Query API function to check if this feature is supported.

*mfxU16* **BRCParamMultiplier**

Specifies a multiplier for bitrate control parameters. Affects the following variables: InitialDelayInKB, BufferSizeInKB, TargetKbps, MaxKbps. If this value is not equal to zero, the encoder calculates BRC parameters as `value * BRCParamMultiplier`.

*mfxFrameInfo* **FrameInfo**

*mfxFrameInfo* structure that specifies frame parameters.

*mfxU32* **CodecId**

Specifies the codec format identifier in the FourCC code; see the CodecFormatFourCC enumerator for details. This is a mandated input parameter for the QueryIOSurf and Init API functions.

*mfxU16* **CodecProfile**

Specifies the codec profile; see the CodecProfile enumerator for details. Specify the codec profile explicitly or the API functions will determine the correct profile from other sources, such as resolution and bitrate.

*mfxU16* **CodecLevel**

Codec level; see the CodecLevel enumerator for details. Specify the codec level explicitly or the functions will determine the correct level from other sources, such as resolution and bitrate.

*mfxU16* **TargetUsage**

Target usage model that guides the encoding process; see the TargetUsage enumerator for details.

*mfxU16* **GopPicSize**

Number of pictures within the current GOP (Group of Pictures); if GopPicSize = 0, then the GOP size is unspecified. If GopPicSize = 1, only I-frames are used. The following pseudo-code that shows how the library uses this parameter:

```
mfxU16 get_gop_sequence (...) {
   pos=display_frame_order;
   if (pos == 0)
       return MFX_FRAMETYPE_I | MFX_FRAMETYPE_IDR | MFX_FRAMETYPE_REF;

   If (GopPicSize == 1) // Only I-frames
       return MFX_FRAMETYPE_I | MFX_FRAMETYPE_REF;

   if (GopPicSize == 0)
               frameInGOP = pos;    //Unlimited GOP
           else
               frameInGOP = pos%GopPicSize;

   if (frameInGOP == 0)
       return MFX_FRAMETYPE_I | MFX_FRAMETYPE_REF;

   if (GopRefDist == 1 || GopRefDist == 0)    // Only I,P frames
               return MFX_FRAMETYPE_P | MFX_FRAMETYPE_REF;

   frameInPattern = (frameInGOP-1)%GopRefDist;
   if (frameInPattern == GopRefDist - 1)
       return MFX_FRAMETYPE_P | MFX_FRAMETYPE_REF;

   return MFX_FRAMETYPE_B;
 }
```

*mfxU16* **GopRefDist**

Distance between I- or P (or GPB) - key frames; if it is zero, the GOP structure is unspecified. Note: If GopRefDist = 1, there are no regular B-frames used (only P or GPB); if *mfxExtCodingOption3::GPB* is ON, GPB frames (B without backward references) are used instead of P.

*mfxU16* **GopOptFlag**

ORs of the GopOptFlag enumerator indicate the additional flags for the GOP specification.

*mfxU16* **IdrInterval**

For H.264, specifies IDR-frame interval in terms of I-frames. For example:

- If IdrInterval = 0, then every I-frame is an IDR-frame.
- If IdrInterval = 1, then every other I-frame is an IDR-frame.

For HEVC, if IdrInterval = 0, then only first I-frame is an IDR-frame. For example:

- If IdrInterval = 1, then every I-frame is an IDR-frame.
- If IdrInterval = 2, then every other I-frame is an IDR-frame.

For MPEG2, IdrInterval defines sequence header interval in terms of I-frames. For example:

- If IdrInterval = 0 (default), then the sequence header is inserted once at the beginning of the stream.
- If IdrInterval = N, then the sequence header is inserted before every Nth I-frame.

If GopPicSize or GopRefDist is zero, IdrInterval is undefined.

*mfxU16* **InitialDelayInKB**

Initial size of the Video Buffering Verifier (VBV) buffer.

**Note:** In this context, KB is 1000 bytes and Kbps is 1000 bps.

*mfxU16* **QPI**

Quantization Parameter (QP) for I-frames for constant QP mode (CQP). Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPI might be clipped to supported QPI range.

**Note:** In the HEVC design, a further adjustment to QPs can occur based on bit depth. Adjusted QPI value = QPI - (6 * (BitDepthLuma - 8)) for BitDepthLuma in the range [8,14]. For HEVC_MAIN10, we minus (6*(10-8)=12) on our side and continue.

**Note:** Default QPI value is implementation dependent and subject to change without additional notice in this document.

*mfxU16* **Accuracy**

Specifies accuracy range in the unit of tenth of percent.

*mfxU16* **BufferSizeInKB**

Represents the maximum possible size of any compressed frames.

*mfxU16* **TargetKbps**

Constant bitrate TargetKbps. Used to estimate the targeted frame size by dividing the frame rate by the bitrate.

*mfxU16* **QPP**

Quantization Parameter (QP) for P-frames for constant QP mode (CQP). Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPP might be clipped to supported QPI range.

---

**Note:** In the HEVC design, a further adjustment to QPs can occur based on bit depth. Adjusted QPP value = QPP - (6 * (BitDepthLuma - 8)) for BitDepthLuma in the range [8,14]. For HEVC_MAIN10, we minus (6*(10-8)=12) on our side and continue.

---

**Note:** Default QPP value is implementation dependent and subject to change without additional notice in this document.

---

*mfxU16* **ICQQuality**

Used by the Intelligent Constant Quality (ICQ) bitrate control algorithm. Values are in the 1 to 51 range, where 1 corresponds the best quality.

*mfxU16* **MaxKbps**

The maximum bitrate at which the encoded data enters the Video Buffering Verifier (VBV) buffer.

*mfxU16* **QPB**

Quantization Parameter (QP) for B-frames for constant QP mode (CQP). Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPI might be clipped to supported QPB range.

---

**Note:** In the HEVC design, a further adjustment to QPs can occur based on bit depth. Adjusted QPB value = QPB - (6 * (BitDepthLuma - 8)) for BitDepthLuma in the range [8,14]. For HEVC_MAIN10, we minus (6*(10-8)=12) on our side and continue.

---

**Note:** Default QPB value is implementation dependent and subject to change without additional notice in this document.

---

*mfxU16* **Convergence**

Convergence period in the unit of 100 frames.

*mfxU16* **NumSlice**

Number of slices in each video frame. Each slice contains one or more macro-block rows. If NumSlice equals zero, the encoder may choose any slice partitioning allowed by the codec standard. See also *mfxExtCodingOption2::NumMbPerSlice*.

*mfxU16* **NumRefFrame**

Max number of all available reference frames (for AVC/HEVC, NumRefFrame defines DPB size). If NumRefFrame = 0, this parameter is not specified. See also NumRefActiveP, NumRefActiveBL0, and NumRefActiveBL1 in the *mfxExtCodingOption3* structure, which set a number of active references.

*mfxU16* `EncodedOrder`

If not zero, specifies that ENCODE takes the input surfaces in the encoded order and uses explicit frame type control. The application must still provide GopRefDist and *mfxExtCodingOption2::BRefType* so the library can pack headers and build reference lists correctly.

*mfxU16* `DecodedOrder`

For AVC and HEVC, used to instruct the decoder to return output frames in the decoded order. Must be zero for all other decoders. When enabled, correctness of *mfxFrameData::TimeStamp* and FrameOrder for output surface is not guaranteed, the application should ignore them.

*mfxU16* `ExtendedPicStruct`

Instructs DECODE to output extended picture structure values for additional display attributes. See the PicStruct description for details.

*mfxU16* `TimeStampCalc`

Time stamp calculation method. See the TimeStampCalc description for details.

*mfxU16* `SliceGroupsPresent`

Nonzero value indicates that slice groups are present in the bitstream. Used only by AVC decoder.

*mfxU16* `MaxDecFrameBuffering`

Nonzero value specifies the maximum required size of the decoded picture buffer in frames for AVC and HEVC decoders.

*mfxU16* `EnableReallocRequest`

For decoders supporting dynamic resolution change (VP9), set this option to ON to allow MFXVideoDECODE_DecodeFrameAsync return MFX_ERR_REALLOC_SURFACE. See the CodingOptionValue enumerator for values of this option. Use the Query API function to check if this feature is supported.

*mfxU16* `FilmGrain`

Special parameter for AV1 decoder. Indicates presence/absence of film grain parameters in bitstream. Also controls decoding behavior for streams with film grain parameters. MFXVideoDECODE_DecodeHeader returns nonzero FilmGrain for streams with film grain parameters and zero for streams w/o them. Decoding with film grain requires additional output surfaces. If FilmGrain` is non-zero then MFXVideoDECODE_QueryIOSurf will request more surfaces in case of external allocated video memory at decoder output. FilmGrain is passed to MFXVideoDECODE_Init function to control decoding operation for AV1 streams with film grain parameters. If FilmGrain is nonzero decoding of each frame require two output surfaces (one for reconstructed frame and one for output frame with film grain applied). The decoder returns MFX_ERR_MORE_SURFACE from MFXVideoDECODE_DecodeFrameAsync if it has insufficient output surfaces to decode frame. Application can forcibly disable the feature passing zero value of `FilmGrain` to `MFXVideoDECODE_Init`. In this case the decoder will output reconstructed frames w/o film grain applied. Application can retrieve film grain parameters for a frame by attaching extended buffer *mfxExtAV1FilmGrainParam* to *mfxFrameSurface1*. If stream has no film grain parameters `FilmGrain` passed to `MFXVideoDECODE_Init` is ignored by the decoder.

*mfxU16* `IgnoreLevelConstrain`

If not zero, it forces SDK to attempt to decode bitstream even if a decoder may not support all features associated with given CodecLevel. Decoder may produce visual artifacts. Only AVC decoder supports this field.

*mfxU16* `SkipOutput`

This flag is used to disable output of main decoding channel. When it's ON SkipOutput = MFX_CODINGOPTION_ON decoder outputs only video processed channels. For pure decode this flag should be always disabled.

*mfxU16* `JPEGChromaFormat`

Specify the chroma sampling format that has been used to encode a JPEG picture. See the ChromaFormat enumerator for details.

*mfxU16* `Rotation`

Rotation option of the output JPEG picture. See the Rotation enumerator for details.

*mfxU16* `JPEGColorFormat`

Specify the color format that has been used to encode a JPEG picture. See the JPEGColorFormat enumerator for details.

*mfxU16* `InterleavedDec`

Specify JPEG scan type for decoder. See the JPEGScanType enumerator for details.

*mfxU8* `SamplingFactorH`[4]

Horizontal sampling factor.

*mfxU8* `SamplingFactorV`[4]

Vertical sampling factor.

*mfxU16* `Interleaved`

Specify interleaved or non-interleaved scans. If it is equal to MFX_SCANTYPE_INTERLEAVED then the image is encoded as interleaved, all components are encoded in one scan. See the JPEG Scan Type enumerator for details.

*mfxU16* `Quality`

Specifies the image quality if the application does not specified quantization table. The value is from 1 to 100 inclusive. "100" is the best quality.

*mfxU16* `RestartInterval`

Specifies the number of MCU in the restart interval. "0" means no restart interval.

**Note:** The *`mfxInfoMFX::InitialDelayInKB`*, *`mfxInfoMFX::TargetKbps`*, *`mfxInfoMFX::MaxKbps`* parameters are used by the constant bitrate (CBR), variable bitrate control (VBR), and CQP HRD algorithms.

Encoders follow the Hypothetical Reference Decoding (HRD) model. The HRD model assumes that data flows into a buffer of the fixed size BufferSizeInKB with a constant bitrate of TargetKbps. (Estimate the targeted frame size by dividing frame rate by bitrate.)

The decoder starts decoding after the buffer reaches the initial size InitialDelayInKB, which is equivalent to reaching an initial delay of InitialDelayInKB*8000/TargetKbpsms. *In this context, KB is 1000 bytes and Kbps is 1000 bps.*

If InitialDelayInKB or BufferSizeInKB is equal to zero, the value is calculated using bitrate, frame rate, profile, level, and so on.

TargetKbps must be specified for encoding initialization.

For variable bitrate control, the MaxKbps parameter specifies the maximum bitrate at which the encoded data enters the Video Buffering Verifier (VBV) buffer. If MaxKbps is equal to zero, the value is calculated from bitrate, frame rate, profile, and level.

---

**Note:** The *mfxInfoMFX::TargetKbps*, *mfxInfoMFX::Accuracy*, *mfxInfoMFX::Convergence* parameters are used by the average variable bitrate control (AVBR) algorithm. The algorithm focuses on overall encoding quality while meeting the specified bitrate, TargetKbps, within the accuracy range, Accuracy, after a Convergence period. This method does not follow HRD and the instant bitrate is not capped or padded.

---

### mfxMVCOperationPoint

struct **mfxMVCOperationPoint**

Describes the MVC operation point.

#### Public Members

*mfxU16* **TemporalId**

Temporal identifier of the operation point.

*mfxU16* **LevelIdc**

Level value signaled for the operation point.

*mfxU16* **NumViews**

Number of views required for decoding the target output views that correspond to the operation point.

*mfxU16* **NumTargetViews**

Number of target output views for the operation point.

*mfxU16* ***TargetViewId**

Target output view identifiers for operation point.

### mfxMVCViewDependency

struct **mfxMVCViewDependency**

Describes MVC view dependencies.

#### Public Members

*mfxU16* **ViewId**

View identifier of this dependency structure.

*mfxU16* **NumAnchorRefsL0**

Number of view components for inter-view prediction in the initial reference picture list RefPicList0 for anchor view components.

*mfxU16* **NumAnchorRefsL1**

Number of view components for inter-view prediction in the initial reference picture list RefPicList1 for anchor view components.

*mfxU16* **AnchorRefL0**[16]

View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList0 for anchor view components.

*mfxU16* **AnchorRefL1**[16]

View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList1 for anchor view components.

*mfxU16* **NumNonAnchorRefsL0**

Number of view components for inter-view prediction in the initial reference picture list RefPicList0 for non-anchor view components.

*mfxU16* **NumNonAnchorRefsL1**

Number of view components for inter-view prediction in the initial reference picture list RefPicList1 for non-anchor view components.

*mfxU16* **NonAnchorRefL0**[16]

View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList0 for non-anchor view components.

### mfxPayload

struct **mfxPayload**

Describes user data payload in MPEG-2 or SEI message payload in H.264.

For encoding, these payloads can be inserted into the bitstream. The payload buffer must contain a valid formatted payload.

For H.264, this is the sei_message() as specified in the section 7.3.2.3.1 'Supplemental enhancement information message syntax' of the ISO/IEC 14496-10 specification.

For MPEG-2, this is the section 6.2.2.2.2 'User data' of the ISO/IEC 13818-2 specification, excluding the user data start_code.

For decoding, these payloads can be retrieved as the decoder parses the bitstream and caches them in an internal buffer.

#### Public Members

*mfxU32* **CtrlFlags**

Additional payload properties. See the PayloadCtrlFlags enumerator for details.

*mfxU8* ***Data**

Pointer to the actual payload data buffer.

*mfxU32* **NumBit**

Number of bits in the payload data

*mfxU16* **Type**

MPEG-2 user data start code or H.264 SEI message type.

*mfxU16* **BufSize**

Payload buffer size in bytes.

| Codec | Supported Types |
|---|---|
| MPEG2 | 0x01B2 //User Data |
| AVC | 02 //pan_scan_rect<br>03 //filler_payload<br>04 //user_data_registered_itu_t_t35<br>05 //user_data_unregistered<br>06 //recovery_point<br>09 //scene_info<br>13 //full_frame_freeze<br>14 //full_frame_freeze_release<br>15 //full_frame_snapshot<br>16 //progressive_refinement_segment_start<br>17 //progressive_refinement_segment_end<br>19 //film_grain_characteristics<br>20 //deblocking_filter_display_preference<br>21 //stereo_video_info<br>45 //frame_packing_arrangement |
| HEVC | All |

### mfxVideoParam

struct **mfxVideoParam**

Configuration parameters for encoding, decoding, transcoding, and video processing.

#### Public Members

*mfxU32* **AllocId**

Unique component ID that will be passed by the library to *mfxFrameAllocRequest*. Useful in pipelines where several components of the same type share the same allocator.

*mfxU16* **AsyncDepth**

Specifies how many asynchronous operations an application performs before the application explicitly synchronizes the result. If zero, the value is not specified.

*mfxInfoMFX* **mfx**

Configurations related to encoding, decoding, and transcoding. See the definition of the *mfxInfoMFX* structure for details.

*mfxInfoVPP* **vpp**

Configurations related to video processing. See the definition of the *mfxInfoVPP* structure for details.

*mfxU16* **Protected**

Specifies the content protection mechanism. See the Protected enumerator for a list of supported protection schemes.

*mfxU16* **IOPattern**

Input and output memory access types for functions. See the enumerator IOPattern for details. The Query API functions return the natively supported IOPattern if the Query input argument is NULL. This parameter is a mandated input for QueryIOSurf and Init API functions. The output pattern must be specified for DECODE. The input pattern must be specified for ENCODE. Both input and output pattern must be specified for VPP.

*mfxExtBuffer* ****ExtParam**

Points to an array of pointers to the extra configuration structures. See the ExtendedBufferID enumerator for a list of extended configurations. The list of extended buffers should not contain duplicated entries, such as entries of the same type. If the *mfxVideoParam* structure is used to query library capability, then the list of extended buffers attached to the input and output *mfxVideoParam* structure should be equal, that is, it should contain the same number of extended buffers of the same type.

*mfxU16* **NumExtParam**

The number of extra configuration structures attached to this structure.

### mfxVP9SegmentParam

struct **mfxVP9SegmentParam**

Contains features and parameters for the segment.

#### Public Members

*mfxU16* **FeatureEnabled**

Indicates which features are enabled for the segment. See the SegmentFeature enumerator for values for this option. Values from the enumerator can be bit-OR'ed. Support of a particular feature depends on underlying hardware platform. Application can check which features are supported by calling Query.

*mfxI16* **QIndexDelta**

Quantization index delta for the segment. Ignored if MFX_VP9_SEGMENT_FEATURE_QINDEX isn't set in FeatureEnabled. Valid range for this parameter is [-255, 255]. If QIndexDelta is out of this range, it will be ignored. If QIndexDelta is within valid range, but sum of base quantization index and QIndexDelta is out of [0, 255], QIndexDelta will be clamped.

*mfxI16* **LoopFilterLevelDelta**

Loop filter level delta for the segment. Ignored if MFX_VP9_SEGMENT_FEATURE_LOOP_FILTER is not set in FeatureEnabled. Valid range for this parameter is [-63, 63]. If LoopFilterLevelDelta is out of this range, it will be ignored. If LoopFilterLevelDelta is within valid range, but sum of base loop filter level and LoopFilterLevelDelta is out of [0, 63], LoopFilterLevelDelta will be clamped.

*mfxU16* **ReferenceFrame**

Reference frame for the segment. See VP9ReferenceFrame enumerator for values for this option. Ignored if MFX_VP9_SEGMENT_FEATURE_REFERENCE isn't set in FeatureEnabled.

### mfxExtAV1FilmGrainParam

struct **mfxExtAV1FilmGrainParam**

The structure is used by AV-1 decoder to report film grain parameters for decoded frame.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AV1_FILM_GRAIN_PARAM.

*mfxU16* **FilmGrainFlags**

Bit map with bit-ORed flags from FilmGrainFlags enum.

*mfxU16* **GrainSeed**

Starting value for pseudo-random numbers used during film grain synthesis.

*mfxU8* **RefIdx**

Indicate which reference frame contains the film grain parameters to be used for this frame.

*mfxU8* **NumYPoints**

The number of points for the piece-wise linear scaling function of the luma component.

*mfxU8* **NumCbPoints**

The number of points for the piece-wise linear scaling function of the Cb component.

*mfxU8* **NumCrPoints**

The number of points for the piece-wise linear scaling function of the Cr component.

*mfxAV1FilmGrainPoint* **PointY**[14]

The array of points for luma component.

*mfxAV1FilmGrainPoint* **PointCb**[10]

The array of points for Cb component.

*mfxAV1FilmGrainPoint* **PointCr**[10]

The array of points for Cr component.

*mfxU8* **GrainScalingMinus8**

The shift - 8 applied to the values of the chroma component. The grain_scaling_minus_8 can take values of 0..3 and determines the range and quantization step of the standard deviation of film grain.

*mfxU8* **ArCoeffLag**

The number of auto-regressive coefficients for luma and chroma.

*mfxU8* **ArCoeffsYPlus128**[24]

Auto-regressive coefficients used for the Y plane.

*mfxU8* **ArCoeffsCbPlus128**[25]

Auto-regressive coefficients used for the Cb plane.

*mfxU8* **ArCoeffsCrPlus128**[25]

The number of points for the piece-wise linear scaling function of the Cr component.

*mfxU8* **ArCoeffShiftMinus6**

The range of the auto-regressive coefficients. Values of 0, 1, 2, and 3 correspond to the ranges for auto-regressive coefficients of [-2, 2), [-1, 1), [-0.5, 0.5) and [-0.25, 0.25) respectively.

*mfxU8* **GrainScaleShift**

Downscaling factor of the grain synthesis process for the Gaussian random numbers .

*mfxU8* **CbMult**

The multiplier for the Cb component used in derivation of the input index to the Cb component scaling function.

*mfxU8* **CbLumaMult**

The multiplier for the average luma component used in derivation of the input index to the Cb component scaling function.

*mfxU16* **CbOffset**

The offset used in derivation of the input index to the Cb component scaling function.

*mfxU8* **CrMult**

The multiplier for the Cr component used in derivation of the input index to the Cr component scaling function.

*mfxU8* **CrLumaMult**

The multiplier for the average luma component used in derivation of the input index to the Cr component scaling function.

*mfxU16* **CrOffset**

The offset used in derivation of the input index to the Cr component scaling function.

### mfxAV1FilmGrainPoint

struct **mfxAV1FilmGrainPoint**

Defines film grain point.

#### Public Members

*mfxU8* **Value**

The x coordinate for the i-th point of the piece-wise linear scaling function for luma/Cb/Cr component.

*mfxU8* **Scaling**

The scaling (output) value for the i-th point of the piecewise linear scaling function for luma/Cb/Cr component.

### mfxRect

struct **mfxRect**

The structure describes rectangle coordinates that can be used for ROI or for Cropping.

#### Public Members

*mfxU16* **Left**

X coordinate of region of top-left corner of rectangle.

*mfxU16* **Top**

Y coordinate of region of top-left corner of rectangle.

*mfxU16* **Right**

X coordinate of region of bottom-right corner of rectangle.

*mfxU16* **Bottom**

Y coordinate of region of bottom-right corner of rectangle.

### mfxExtHyperModeParam

struct **mfxExtHyperModeParam**

The structure is used for HyperMode initialization.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. BufferId must be equal to MFX_EXTBUFF_HYPER_MODE_PARAM.

*mfxHyperMode* **Mode**

HyperMode implementation behavior.

## mfxGUID

struct **mfxGUID**

Represents Globally Unique Identifier (GUID) with memory layout compliant to RFC 4122. See https://www.rfc-editor.org/info/rfc4122 for details.

### Public Members

*mfxU8* **Data**[16]

Array to keep GUID.

## mfxExtAllocationHints

struct **mfxExtAllocationHints**

The extension buffer specifies surface pool management policy. Absence of the attached buffer means MFX_ALLOCATION_UNLIMITED policy: each call of GetSurfaceForXXX leads to surface allocation.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ALLOCATION_HINTS.

*mfxPoolAllocationPolicy* **AllocationPolicy**

Allocation policy.

*mfxU32* **NumberToPreAllocate**

How many surfaces to allocate during Init. It's applicable for any polices set by mfxPoolAllocationPolicy::AllocationPolicy even if the requested number exceeds recommended size of the pool.

*mfxU32* **DeltaToAllocateOnTheFly**

DeltaToAllocateOnTheFly specifies how many surfaces are allocated in addition to NumberToPreAllocate in MFX_ALLOCATION_LIMITED mode. Maximum number of allocated frames will be NumberToPreAllocate + DeltaToAllocateOnTheFly.

*mfxVPPPoolType* **VPPPoolType**

Defines what VPP pool is targeted - input or output. Ignored for other components.

*mfxU32* **Wait**

Time in milliseconds for GetSurfaceForXXX() and DecodeFrameAsync functions to wait until surface will be available.

*mfxU32* **reserved1**[4]

Reserved for future use

### mfxRefInterface

struct **mfxRefInterface**

The structure represents reference counted interface structure. The memory is allocated and released by the implementation.

#### Public Members

*mfxHDL* **Context**

The context of the container interface. User should not touch (change, set, null) this pointer.

*mfxStructVersion* **Version**

The version of the structure.

*mfxStatus* (***AddRef**)(struct *mfxRefInterface* *ref_interface)

Increments the internal reference counter of the container. The container is not destroyed until the container is released using the *mfxRefInterface::Release* function. *mfxRefInterface::AddRef* should be used each time a new link to the container is created (for example, copy structure) for proper management.

**Param ref_interface**
**[in]** Valid interface.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If interface is NULL.

MFX_ERR_INVALID_HANDLE If mfxRefInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Release**)(struct *mfxRefInterface* *ref_interface)

Decrements the internal reference counter of the container. *mfxRefInterface::Release* should be called after using the *mfxRefInterface::AddRef* function to add a container or when allocation logic requires it.

**Param ref_interface**
**[in]** Valid interface.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If interface is NULL.

MFX_ERR_INVALID_HANDLE If mfxRefInterface->Context is invalid (for example NULL).

MFX_ERR_UNDEFINED_BEHAVIOR If Reference Counter of container is zero before call.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetRefCounter**)(struct *mfxRefInterface* *ref_interface, *mfxU32* *counter)

Returns current reference counter of *mfxRefInterface* structure.

**Param ref_interface**
**[in]** Valid interface.

**Param counter**
**[out]** Sets counter to the current reference counter value.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If interface or counter is NULL.

MFX_ERR_INVALID_HANDLE If mfxRefInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

### mfxExtMasteringDisplayColourVolume

struct **mfxExtMasteringDisplayColourVolume**

Handle the HDR SEI message.

During encoding: If the application attaches this structure to the *mfxEncodeCtrl* structure at runtime, the encoder inserts the HDR SEI message for the current frame and ignores InsertPayloadToggle. If the application attaches this structure to the *mfxVideoParam* structure during initialization or reset, the encoder inserts the HDR SEI message based on InsertPayloadToggle.

During video processing: If the application attaches this structure for video processing, InsertPayloadToggle will be ignored. And DisplayPrimariesX[3], DisplayPrimariesY[3] specify the color primaries where 0,1,2 specifies Red, Green, Blue respectively.

During decoding: If the application attaches this structure to the *mfxFrameSurface1* structure at runtime which will seed to the MFXVideoDECODE_DecodeFrameAsync() as surface_work parameter, the decoder will parse the HDR SEI message if the bitstream include HDR SEI message per frame. The parsed HDR SEI will be attached to the ExtendBuffer of surface_out parameter of MFXVideoDECODE_DecodeFrameAsync() with flag `InsertPayloadToggle` to indicate if there is valid HDR SEI message in the clip. `InsertPayloadToggle` will be set to `MFX_PAYLOAD_IDR` if oneAPI Video Processing Library (oneVPL) gets valid HDR SEI, otherwise it will be set to `MFX_PAYLOAD_OFF`. This function is support for HEVC only now.

Encoding or Decoding, Field semantics are defined in ITU-T* H.265 Annex D, AV1 6.7.4 Metadata OBU semantics.

Video processing, DisplayPrimariesX[3] and WhitePointX are in increments of 0.00002, in the range of [5, 37000]. DisplayPrimariesY[3] and WhitePointY are in increments of 0.00002, in the range of [5, 42000]. MaxDisplayMasteringLuminance is in units of 1 candela per square meter. MinDisplayMasteringLuminance is in units of 0.0001 candela per square meter.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUM

*mfxU16* **InsertPayloadToggle**

InsertHDRPayload enumerator value.

*mfxU16* **DisplayPrimariesX**[3]

Color primaries for a video source. Consist of RGB x coordinates and define how to convert colors from RGB color space to CIE XYZ color space.

*mfxU16* **DisplayPrimariesY**[3]

Color primaries for a video source. Consists of RGB y coordinates and defines how to convert colors from RGB color space to CIE XYZ color space.

*mfxU16* **WhitePointX**

White point X coordinate.

*mfxU16* **WhitePointY**

White point Y coordinate.

*mfxU32* **MaxDisplayMasteringLuminance**

Specify maximum luminance of the display on which the content was authored.

*mfxU32* **MinDisplayMasteringLuminance**

Specify minimum luminance of the display on which the content was authored.

## mfxExtContentLightLevelInfo

struct **mfxExtContentLightLevelInfo**

Handle the HDR SEI message.

During encoding: If the application attaches this structure to the *mfxEncodeCtrl* structure at runtime, the encoder inserts the HDR SEI message for the current frame and ignores InsertPayloadToggle. If the application attaches this structure to the *mfxVideoParam* structure during initialization or reset, the encoder inserts the HDR SEI message based on InsertPayloadToggle.

During video processing: If the application attaches this structure for video processing, InsertPayloadToggle will be ignored.

During decoding: If the application attaches this structure to the *mfxFrameSurface1* structure at runtime which will seed to the MFXVideoDECODE_DecodeFrameAsync() as surface_work parameter, the decoder will parse the HDR SEI message if the bitstream include HDR SEI message per frame. The parsed HDR SEI will be attached to the ExtendBuffer of surface_out parameter of MFXVideoDECODE_DecodeFrameAsync() with flag `InsertPayloadToggle` to indicate if there is valid HDR SEI message in the clip. `InsertPayloadToggle` will be set to `MFX_PAYLOAD_IDR` if oneVPL gets valid HDR SEI, otherwise it will be set to `MFX_PAYLOAD_OFF`. This function is support for HEVC only now.

Field semantics are defined in ITU-T* H.265 Annex D, AV1 6.7.3 Metadata high dynamic range content light level semantics.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to EXTBUFF_CONTENT_LIGHT_LEVEL_INFO.

*mfxU16* **InsertPayloadToggle**

InsertHDRPayload enumerator value.

*mfxU16* **MaxContentLightLevel**

Maximum luminance level of the content. Field range is 1 to 65535.

*mfxU16* **MaxPicAverageLightLevel**

Maximum average per-frame luminance level of the content. Field range is 1 to 65535.

### mfxExtSyncSubmission

struct **mfxExtSyncSubmission**

The structure is used to get a synchronization object which signalizes about submission of a task to GPU.

#### Public Members

*mfxSyncPoint* ***SubmissionSyncPoint**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_SYNCSUBMISSION. SyncPoint object to get a moment of a submission task to GPU.

*mfxU32* **reserved1**[8]

Reserved for future use.

### mfxExtTuneEncodeQuality

struct **mfxExtTuneEncodeQuality**

The structure specifies type of quality optimization used by the encoder. The buffer can also be attached for VPP functions to make correspondent pre-filtering.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_TUNE_ENCODE_QUALITY.

*mfxU32* **TuneQuality**

The control to specify type of encode quality metric(s) to optimize; See correspondent enum.

*mfxExtBuffer* ****ExtParam**

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

*mfxU16* **NumExtParam**

The number of extra configuration structures attached to the structure.

## mfxConfigInterface

struct **mfxConfigInterface**

### Public Members

*mfxHDL* **Context**

The context of the config interface. User should not touch (change, set, null) this pointer.

*mfxStructVersion* **Version**

The version of the structure.

*mfxStatus* (***SetParameter**)(struct *mfxConfigInterface* *config_interface, const *mfxU8* *key, const *mfxU8* *value, *mfxStructureType* struct_type, *mfxHDL* structure, *mfxExtBuffer* *ext_buffer)

Sets a parameter to specified value in the current session. If a parameter already has a value, the new value will overwrite the existing value.

**Since**

This function is available since API version 2.10.

**Param config_interface**

**[in]** The valid interface returned by calling MFXQueryInterface().

**Param key**

**[in]** Null-terminated string containing parameter to set. The string length must be < MAX_PARAM_STRING_LENGTH bytes.

**Param value**

**[in]** Null-terminated string containing value to which key should be set. The string length must be < MAX_PARAM_STRING_LENGTH bytes. value will be converted from a string to the expected data type for the given key, or return an error if conversion fails.

**Param struct_type**
**[in]** Type of structure pointed to by structure.

**Param structure**
**[out]** If and only if SetParameter returns MFX_ERR_NONE, the contents of structure (including any attached extension buffers) will be updated according to the provided key and value. If key modifies a field in an extension buffer which is not already attached, the function will return MFX_ERR_MORE_EXTBUFFER and fill ext_buffer with the header for the required *mfxExtBuffer* type.

**Param ext_buffer**
**[out]** If and only if SetParameter returns MFX_ERR_MORE_EXTBUFFER, ext_buffer will contain the header for a buffer of type *mfxExtBuffer*. The caller should allocate a buffer of the size ext_buffer.BufferSz, copy the header in ext_buffer to the start of this new buffer, attach this buffer to videoParam, then call SetParameter again. Otherwise, the contents of ext_buffer will be cleared.

**Return**
MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR If key, value, videoParam, and/or ext_buffer is NULL. MFX_ERR_NOT_FOUND If key contains an unknown parameter name. MFX_ERR_UNSUPPORTED If value is of the wrong format for key (for example, a string is provided where an integer is required) or if value cannot be converted into any valid data type. MFX_ERR_INVALID_VIDEO_PARAM If length of key or value is >= MAX_PARAM_STRING_LENGTH or is zero (empty string). MFX_ERR_MORE_EXTBUFFER If key requires modifying a field in an *mfxExtBuffer* which is not attached. Caller must allocate and attach the buffer type provided in ext_buffer then call the function again.

### 5.2.5 Decode Structures

Structures used by Decode only.

#### API

- *mfxDecodeStat*
- *mfxExtDecodeErrorReport*
- *mfxExtDecodedFrameInfo*
- *mfxExtTimeCode*

#### mfxDecodeStat

struct **mfxDecodeStat**

Returns statistics collected during decoding.

#### Public Members

*mfxU32* **NumFrame**

Number of total decoded frames.

*mfxU32* **NumSkippedFrame**

Number of skipped frames.

*mfxU32* **NumError**

Number of errors recovered.

*mfxU32* **NumCachedFrame**

Number of internally cached frames.

### mfxExtDecodeErrorReport

struct **mfxExtDecodeErrorReport**

Used by the decoders to report bitstream error information right after DecodeHeader or DecodeFrameAsync. The application can attach this extended buffer to the *mfxBitstream* structure at runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DECODE_ERROR_REPORT.

*mfxU32* **ErrorTypes**

Bitstream error types (bit-ORed values). See ErrorTypes enumerator for the list of types.

### mfxExtDecodedFrameInfo

struct **mfxExtDecodedFrameInfo**

Used by the decoders to report additional information about a decoded frame. The application can attach this extended buffer to the mfxFrameSurface1::mfxFrameData structure at runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DECODED_FRAME_INFO.

*mfxU16* **FrameType**

Frame type. See FrameType enumerator for the list of types.

### mfxExtTimeCode

struct **mfxExtTimeCode**

Used by the library to pass MPEG 2 specific timing information.

See ISO/IEC 13818-2 and ITU-T H.262, MPEG-2 Part 2 for the definition of these parameters.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_TIME_CODE.

*mfxU16* **DropFrameFlag**

Indicated dropped frame.

*mfxU16* **TimeCodeHours**

Hours.

*mfxU16* **TimeCodeMinutes**

Minutes.

*mfxU16* **TimeCodeSeconds**

Seconds.

*mfxU16* **TimeCodePictures**

Pictures.

## 5.2.6 Encode Structures

Structures used by Encode only.

### API

- *mfxBRCFrameCtrl*
- *mfxBRCFrameParam*
- *mfxBRCFrameStatus*
- *mfxEncodeCtrl*
- *mfxEncodedUnitInfo*
- *mfxEncodeStat*
- *mfxExtAVCEncodedFrameInfo*
- *mfxExtAVCRefListCtrl*

- *mfxExtAVCRefLists*
- *mfxExtAVCRoundingOffset*
- *mfxExtAvcTemporalLayers*
- *mfxExtBRC*
- *mfxExtChromaLocInfo*
- *mfxExtCodingOption*
- *mfxExtCodingOption2*
- *mfxExtCodingOption3*
- *mfxExtCodingOptionSPSPPS*
- *mfxExtCodingOptionVPS*
- *mfxExtDirtyRect*
- *mfxExtEncodedUnitsInfo*
- *mfxExtEncoderCapability*
- *mfxExtEncoderIPCMArea*
- *mfxExtEncoderResetOption*
- *mfxExtEncoderROI*
- *mfxExtHEVCRegion*
- *mfxExtHEVCTiles*
- *mfxExtInsertHeaders*
- *mfxExtMBDisableSkipMap*
- *mfxExtMBForceIntra*
- *mfxExtMBQP*
- *mfxExtMoveRect*
- *mfxExtMVOverPicBoundaries*
- *mfxExtPartialBitstreamParam*
- *mfxExtPictureTimingSEI*
- *mfxExtPredWeightTable*
- *mfxExtVP8CodingOption*
- *mfxExtVP9Segmentation*
- *mfxExtVP9TemporalLayers*
- *mfxQPandMode*
- *mfxVP9TemporalLayer*
- *mfxTemporalLayer*
- *mfxExtTemporalLayers*
- *mfxExtAV1BitstreamParam*

- *mfxExtAV1ResolutionParam*
- *mfxExtAV1TileParam*
- *mfxExtAV1Segmentation*
- *mfxCTUHeader*
- *mfxCUInfo*
- *mfxCTUInfo*
- *mfxMBInfo*
- *mfxEncodeBlkStats*
- *mfxEncodeHighLevelStats*
- *mfxEncodeFrameStats*
- *mfxEncodeSliceStats*
- *mfxEncodeTileStats*
- *mfxEncodeStatsContainer*
- *mfxExtEncodeStatsOutput*
- *mfxExtHEVCRefListCtrl*
- *mfxExtHEVCRefLists*
- *mfxExtHEVCTemporalLayers*

## mfxBRCFrameCtrl

struct **mfxBRCFrameCtrl**

Specifies controls for next frame encoding provided by external BRC functions.

### Public Members

*mfxI32* **QpY**

Frame-level Luma QP.

*mfxU32* **InitialCpbRemovalDelay**

See initial_cpb_removal_delay in codec standard. Ignored if no HRD control: *mfxExtCodingOption::VuiNalHrdParameters* = MFX_CODINGOPTION_OFF. Calculated by encoder if initial_cpb_removal_delay==0 && initial_cpb_removal_offset == 0 && HRD control is switched on.

*mfxU32* **InitialCpbRemovalOffset**

See initial_cpb_removal_offset in codec standard. Ignored if no HRD control: *mfxExtCodingOption::VuiNalHrdParameters* = MFX_CODINGOPTION_OFF. Calculated by encoder if initial_cpb_removal_delay==0 && initial_cpb_removal_offset == 0 && HRD control is switched on.

*mfxU32* **MaxFrameSize**

Max frame size in bytes. Option for repack feature. Driver calls PAK until current frame size is less than or equal to MaxFrameSize, or number of repacking for this frame is equal to MaxNumRePak. Repack is available if there is driver support, MaxFrameSize !=0, and MaxNumRePak != 0. Ignored if MaxNumRePak == 0.

*mfxU8* **DeltaQP**[8]

Option for repack feature. Ignored if MaxNumRePak == 0 or MaxNumRePak==0. If current frame size > MaxFrameSize and/or number of repacking (nRepack) for this frame <= MaxNumRePak, PAK is called with QP = *mfxBRCFrameCtrl::QpY* + Sum(DeltaQP[i]), where i = [0,nRepack]. Non zero DeltaQP[nRepack] are ignored if nRepack > MaxNumRePak. If repacking feature is on ( MaxFrameSize & MaxNumRePak are not zero), it is calculated by the encoder.

*mfxU16* **MaxNumRepak**

Number of possible repacks in driver if current frame size > MaxFrameSize. Ignored if MaxFrameSize==0. See MaxFrameSize description. Possible values are in the range of 0 to 8.

*mfxU16* **NumExtParam**

Reserved for future use.

*mfxExtBuffer* ****ExtParam**

Reserved for future use.

### mfxBRCFrameParam

struct **mfxBRCFrameParam**

Describes frame parameters required for external BRC functions.

#### Public Members

*mfxU16* **SceneChange**

Frame belongs to a new scene if non zero.

*mfxU16* **LongTerm**

Frame is a Long Term Reference frame if non zero.

*mfxU32* **FrameCmplx**

Frame Complexity Frame spatial complexity if non zero. Zero if complexity is not available.

*mfxU32* **EncodedOrder**

The frame number in a sequence of reordered frames starting from encoder Init.

*mfxU32* **DisplayOrder**

The frame number in a sequence of frames in display order starting from last IDR.

*mfxU32* **CodedFrameSize**

Size of the frame in bytes after encoding.

*mfxU16* **FrameType**

Frame type. See FrameType enumerator for possible values.

*mfxU16* **PyramidLayer**

B-pyramid or P-pyramid layer that the frame belongs to.

*mfxU16* **NumRecode**

Number of recodings performed for this frame.

*mfxU16* **NumExtParam**

Reserved for future use.

*mfxExtBuffer* ****ExtParam**

Reserved for future use.

Frame spatial complexity is calculated according to the following formula:

$$R = \frac{16}{WH} \sum_{k=0}^{\frac{W}{4}-1} \sum_{l=0}^{\frac{H}{4}-1} \left[ \frac{\sum_{i=0}^{3} \sum_{j=0}^{3} |P[k*4+i][l*4+j] - P[k*4+i-1][l*4+j]|}{16} \right]$$

$$C = \frac{16}{WH} \sum_{k=0}^{\frac{W}{4}-1} \sum_{l=0}^{\frac{H}{4}-1} \left[ \frac{\sum_{i=0}^{3} \sum_{j=0}^{3} |P[k*4+i][l*4+j] - P[k*4+i][l*4+j-1]|}{16} \right]$$

$$FrameCmplx = \sqrt{R^2 + C^2}$$

## mfxBRCFrameStatus

struct **mfxBRCFrameStatus**

Specifies instructions for the encoder provided by external BRC after each frame encoding. See the BRCStatus enumerator for details.

### Public Members

*mfxU32* **MinFrameSize**

Size in bytes, coded frame must be padded to when Status = MFX_BRC_PANIC_SMALL_FRAME.

*mfxU16* **BRCStatus**

BRC status. See the BRCStatus enumerator for possible values.

### mfxEncodeCtrl

struct **mfxEncodeCtrl**

Contains parameters for per-frame based encoding control.

#### Public Members

*mfxExtBuffer* **Header**

This extension buffer doesn't have assigned buffer ID. Ignored.

*mfxU16* **MfxNalUnitType**

Type of NAL unit that contains encoding frame. All supported values are defined by MfxNalUnitType enumerator. Other values defined in ITU-T H.265 specification are not supported.

The encoder uses this field only if application sets *mfxExtCodingOption3::EnableNalUnitType* option to ON during encoder initialization.

**Note:** Only encoded order is supported. If application specifies this value in display order or uses value inappropriate for current frame or invalid value, then the encoder silently ignores it.

*mfxU16* **SkipFrame**

Indicates that current frame should be skipped or the number of missed frames before the current frame. See *mfxExtCodingOption2::SkipFrame* for details.

*mfxU16* **QP**

If nonzero, this value overwrites the global QP value for the current frame in the constant QP mode.

*mfxU16* **FrameType**

Encoding frame type. See the FrameType enumerator for details. If the encoder works in the encoded order, the application must specify the frame type. If the encoder works in the display order, only key frames are enforceable.

*mfxU16* **NumExtParam**

Number of extra control buffers.

*mfxU16* **NumPayload**

Number of payload records to insert into the bitstream.

*mfxExtBuffer* ****ExtParam**

Pointer to an array of pointers to external buffers that provide additional information or control to the encoder for this frame or field pair. A typical use is to pass the VPP auxiliary data generated by the video processing pipeline to the encoder. See the ExtendedBufferID for the list of extended buffers.

*mfxPayload* ****Payload**

Pointer to an array of pointers to user data (MPEG-2) or SEI messages (H.264) for insertion into the bitstream. For field pictures, odd payloads are associated with the first field and even payloads are associated with the second field. See the *mfxPayload* structure for payload definitions.

### mfxEncodedUnitInfo

struct **mfxEncodedUnitInfo**

Used to report encoded unit information.

#### Public Members

*mfxU16* **Type**

Codec-dependent coding unit type (NALU type for AVC/HEVC, start_code for MPEG2 etc).

*mfxU32* **Offset**

Offset relative to the associated *mfxBitstream::DataOffset*.

*mfxU32* **Size**

Unit size, including delimiter.

### mfxEncodeStat

struct **mfxEncodeStat**

Returns statistics collected during encoding.

#### Public Members

*mfxU32* **NumFrame**

Number of encoded frames.

*mfxU64* **NumBit**

Number of bits for all encoded frames.

*mfxU32* **NumCachedFrame**

Number of internally cached frames.

### mfxExtAVCEncodedFrameInfo

struct **mfxExtAVCEncodedFrameInfo**

Used by the encoder to report additional information about the encoded picture. The application can attach this buffer to the *mfxBitstream* structure before calling MFXVideoENCODE_EncodeFrameAsync function. For interlaced content the encoder requires two such structures. They correspond to fields in encoded order.

**Note:** Not all implementations of the encoder support this extended buffer. The application must use query mode 1 to determine if the functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE then the functionality is supported.

#### Reference Lists

The following structure members are used by the reference lists contained in the parent structure.

*mfxU32* **FrameOrder**

Frame order of encoded picture.

Frame order of reference picture.

*mfxU16* **PicStruct**

Picture structure of encoded picture.

Picture structure of reference picture.

*mfxU16* **LongTermIdx**

Long term index of encoded picture if applicable.

Long term index of reference picture if applicable.

*mfxU16* **reserved**[2]

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODED_FRAME_INFO.

*mfxU32* **MAD**

Mean Absolute Difference between original pixels of the frame and motion compensated (for inter macroblocks) or spatially predicted (for intra macroblocks) pixels. Only luma component, Y plane, is used in calculation.

*mfxU16* **BRCPanicMode**

Bitrate control was not able to allocate enough bits for this frame. Frame quality may be unacceptably low.

*mfxU16* **QP**

Luma QP.

*mfxU32* **SecondFieldOffset**

Offset to second field. Second field starts at *mfxBitstream::Data* + *mfxBitstream::DataOffset* + *mfxExtAVCEncodedFrameInfo::SecondFieldOffset*.

struct *mfxExtAVCEncodedFrameInfo*::[anonymous] **UsedRefListL0**[32]

Reference list that has been used to encode picture.

struct *mfxExtAVCEncodedFrameInfo*::[anonymous] **UsedRefListL1**[32]

Reference list that has been used to encode picture.

### mfxExtAVCRefListCtrl

struct **mfxExtAVCRefListCtrl**

Configures reference frame options for the H.264 encoder. See the *Reference List Selection* and *Long Term Reference Frame* sections for more details.

---

**Note:** Not all implementations of the encoder support LongTermIdx and ApplyLongTermIdx fields in this structure. The application must use query mode 1 to determine if such functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE and these fields were set to one, then the functionality is supported. If the function fails or sets fields to zero, then the functionality is not supported.

---

#### Reference Lists

The following structure members are used by the reference lists contained in the parent structure.

*mfxU32* **FrameOrder**

Together FrameOrder and PicStruct fields are used to identify reference picture. Use FrameOrder = MFX_FRAMEORDER_UNKNOWN to mark unused entry.

*mfxU16* **PicStruct**

Together FrameOrder and PicStruct fields are used to identify reference picture. Use FrameOrder = MFX_FRAMEORDER_UNKNOWN to mark unused entry.

*mfxU16* **ViewId**

Reserved and must be zero.

*mfxU16* **LongTermIdx**

Index that should be used by the encoder to mark long-term reference frame.

*mfxU16* **reserved**[3]

Reserved

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AVC_REFLIST_CTRL.

*mfxU16* **NumRefIdxL0Active**

Specify the number of reference frames in the active reference list L0. This number should be less or equal to the NumRefFrame parameter from encoding initialization.

*mfxU16* **NumRefIdxL1Active**

Specify the number of reference frames in the active reference list L1. This number should be less or equal to the NumRefFrame parameter from encoding initialization.

struct *mfxExtAVCRefListCtrl*::[anonymous] **PreferredRefList**[32]

Reference list that specifies the list of frames that should be used to predict the current frame.

struct *mfxExtAVCRefListCtrl*::[anonymous] **RejectedRefList**[16]

Reference list that specifies the list of frames that should not be used for prediction.

struct *mfxExtAVCRefListCtrl*::[anonymous] **LongTermRefList**[16]

Reference list that specifies the list of frames that should be marked as long-term reference frame.

*mfxU16* **ApplyLongTermIdx**

If it is equal to zero, the encoder assigns long-term index according to internal algorithm. If it is equal to one, the encoder uses LongTermIdx value as long-term index.

## mfxExtAVCRefLists

struct **mfxExtAVCRefLists**

Specifies reference lists for the encoder. It may be used together with the *mfxExtAVCRefListCtrl* structure to create customized reference lists. If both structures are used together, then the encoder takes reference lists from the *mfxExtAVCRefLists* structure and modifies them according to the *mfxExtAVCRefListCtrl* instructions. In case of interlaced coding, the first *mfxExtAVCRefLists* structure affects TOP field and the second - BOTTOM field.

**Note:** Not all implementations of the encoder support this structure. The application must use the Query API function to determine if it is supported.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AVC_REFLISTS.

*mfxU16* **NumRefIdxL0Active**

Specify the number of reference frames in the active reference list L0. This number should be less than or equal to the NumRefFrame parameter from encoding initialization.

*mfxU16* **NumRefIdxL1Active**

Specify the number of reference frames in the active reference list L1. This number should be less than or equal to the NumRefFrame parameter from encoding initialization.

struct *mfxExtAVCRefLists*::*mfxRefPic* **RefPicList0**[32]

Specify L0 reference list.

struct *mfxExtAVCRefLists*::*mfxRefPic* **RefPicList1**[32]

Specify L1 reference list.

struct **mfxRefPic**

Used by the reference lists contained in the parent structure. Together these fields are used to identify reference picture.

#### Public Members

*mfxU32* **FrameOrder**

Use FrameOrder = MFX_FRAMEORDER_UNKNOWN to mark unused entry.

*mfxU16* **PicStruct**

Use PicStruct = MFX_PICSTRUCT_FIELD_TFF for TOP field, PicStruct = MFX_PICSTRUCT_FIELD_BFF for BOTTOM field.

### mfxExtAVCRoundingOffset

struct **mfxExtAVCRoundingOffset**

Used by encoders to set rounding offset parameters for quantization. It is per-frame based encoding control, and can be attached to some frames and skipped for others. When the extension buffer is set the application can attach it to the *mfxEncodeCtrl* during runtime.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AVC_ROUNDING_OFFSET.

*mfxU16* **EnableRoundingIntra**

Enable rounding offset for intra blocks. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **RoundingOffsetIntra**

Intra rounding offset. Value must be in the range of 0 to 7, inclusive.

*mfxU16* **EnableRoundingInter**

Enable rounding offset for inter blocks. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **RoundingOffsetInter**

Inter rounding offset. Value must be in the range of 0 to 7, inclusive.

## mfxExtAvcTemporalLayers

struct **mfxExtAvcTemporalLayers**

Configures the H.264 temporal layers hierarchy.

If the application attaches it to the *mfxVideoParam* structure during initialization, the encoder generates the temporal layers and inserts the prefix NAL unit before each slice to indicate the temporal and priority IDs of the layer.

This structure can be used with the display-order encoding mode only.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AVC_TEMPORAL_LAYERS.

*mfxU16* **BaseLayerPID**

The priority ID of the base layer. The encoder increases the ID for each temporal layer and writes to the prefix NAL unit.

*mfxU16* **Scale**

The ratio between the frame rates of the current temporal layer and the base layer.

### mfxExtBRC

struct **mfxExtBRC**

Contains a set of callbacks to perform external bitrate control. Can be attached to the *mfxVideoParam* structure during encoder initialization. Set the *mfxExtCodingOption2::ExtBRC* option to ON to make the encoder use the external BRC instead of the native one.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_BRC.

*mfxHDL* **pthis**

Pointer to the BRC object.

*mfxStatus* (***Init**)(*mfxHDL* pthis, *mfxVideoParam* *par)

Initializes the BRC session according to parameters from input *mfxVideoParam* and attached structures. It does not modify the input *mfxVideoParam* and attached structures. Invoked during MFXVideoENCODE_Init.

**Param pthis**
**[in]** Pointer to the BRC object.

**Param par**
**[in]** Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

**Return**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_UNSUPPORTED The function detected unsupported video parameters.

*mfxStatus* (***Reset**)(*mfxHDL* pthis, *mfxVideoParam* *par)

Resets BRC session according to new parameters. It does not modify the input *mfxVideoParam* and attached structures. Invoked during MFXVideoENCODE_Reset.

**Param pthis**
**[in]** Pointer to the BRC object.

**Param par**
**[in]** Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

**Return**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_UNSUPPORTED The function detected unsupported video parameters.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that the video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed.

*mfxStatus* (*`Close`)(*mfxHDL* pthis)

Deallocates any internal resources acquired in Init for this BRC session. Invoked during MFXVideoENCODE_Close.

**Param pthis**
**[in]** Pointer to the BRC object.

**Return**
MFX_ERR_NONE The function completed successfully.

*mfxStatus* (*`GetFrameCtrl`)(*mfxHDL* pthis, *mfxBRCFrameParam* *par, *mfxBRCFrameCtrl* *ctrl)

Returns controls (`ctrl`) to encode next frame based on info from input *mfxBRCFrameParam* structure (`par`) and internal BRC state. Invoked asynchronously before each frame encoding or recoding.

**Param pthis**
**[in]** Pointer to the BRC object.

**Param par**
**[in]** Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

**Param ctrl**
**[out]** Pointer to the output *mfxBRCFrameCtrl* structure.

**Return**
MFX_ERR_NONE The function completed successfully.

*mfxStatus* (*`Update`)(*mfxHDL* pthis, *mfxBRCFrameParam* *par, *mfxBRCFrameCtrl* *ctrl, *mfxBRCFrameStatus* *status)

Updates internal BRC state and returns status to instruct encoder whether it should recode the previous frame, skip the previous frame, do padding, or proceed to next frame based on info from input *mfxBRCFrameParam* and *mfxBRCFrameCtrl* structures. Invoked asynchronously after each frame encoding or recoding.

**Param pthis**
**[in]** Pointer to the BRC object.

**Param par**
**[in]** Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

**Param ctrl**
**[in]** Pointer to the output *mfxBRCFrameCtrl* structure.

**Param status**
**[in]** Pointer to the output *mfxBRCFrameStatus* structure.

**Return**
MFX_ERR_NONE The function completed successfully.

### mfxExtChromaLocInfo

struct **mfxExtChromaLocInfo**

Members of this structure define the location of chroma samples information.

See Annex E of the ISO*\/IEC* 14496-10 specification for the definition of these parameters.

**Note:** Not all implementations of the encoder support this structure. The application must use the Query API function to determine if it is supported.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CHROMA_LOC_INFO.

*mfxU16* **ChromaLocInfoPresentFlag**

*mfxU16* **ChromaSampleLocTypeTopField**

*mfxU16* **ChromaSampleLocTypeBottomField**

*mfxU16* **reserved**[9]

### mfxExtCodingOption

struct **mfxExtCodingOption**

Specifies additional options for encoding.

The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION.

*mfxU16* **RateDistortionOpt**

Set this flag if rate distortion optimization is needed. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **MECostType**

Motion estimation cost type. This value is reserved and must be zero.

*mfxU16* **MESearchType**

Motion estimation search algorithm. This value is reserved and must be zero.

*mfxI16Pair* **MVSearchWindow**

Rectangular size of the search window for motion estimation. This parameter is reserved and must be (0, 0).

*mfxU16* **FramePicture**

Set this flag to encode interlaced fields as interlaced frames. This flag does not affect progressive input frames. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **CAVLC**

If set, CAVLC is used; if unset, CABAC is used for encoding. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **RecoveryPointSEI**

Set this flag to insert the recovery point SEI message at the beginning of every intra refresh cycle. See the description of IntRefType in *mfxExtCodingOption2* structure for details on how to enable and configure intra refresh.

If intra refresh is not enabled then this flag is ignored.

See the CodingOptionValue enumerator for values of this option.

*mfxU16* **ViewOutput**

Set this flag to instruct the MVC encoder to output each view in separate bitstream buffer. See the CodingOptionValue enumerator for values of this option and the Multi-View Video Coding section for more details about usage of this flag.

*mfxU16* **NalHrdConformance**

If this option is turned ON, then AVC encoder produces an HRD conformant bitstream. If it is turned OFF, then the AVC encoder may (but not necessarily) violate HRD conformance. That is, this option can force the encoder to produce an HRD conformant stream, but cannot force it to produce a non-conformant stream.

See the CodingOptionValue enumerator for values of this option.

*mfxU16* **SingleSeiNalUnit**

If set, encoder puts all SEI messages in the singe NAL unit. It includes messages provided by application and created by encoder. It is a three-states option. See CodingOptionValue enumerator for values of this option. The three states are:

- UNKNOWN Put each SEI in its own NAL unit.
- ON Put all SEI messages in the same NAL unit.
- OFF The same as unknown.

*mfxU16* **VuiVclHrdParameters**

If set and VBR rate control method is used, then VCL HRD parameters are written in bitstream with values identical to the values of the NAL HRD parameters. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **RefPicListReordering**

Set this flag to activate reference picture list reordering. This value is reserved and must be zero.

*mfxU16* **ResetRefList**

Set this flag to reset the reference list to non-IDR I-frames of a GOP sequence. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **RefPicMarkRep**

Set this flag to write the reference picture marking repetition SEI message into the output bitstream. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **FieldOutput**

Set this flag to instruct the AVC encoder to output bitstreams immediately after the encoder encodes a field, in the field-encoding mode. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **IntraPredBlockSize**

Minimum block size of intra-prediction. This value is reserved and must be zero.

*mfxU16* **InterPredBlockSize**

Minimum block size of inter-prediction. This value is reserved and must be zero.

*mfxU16* **MVPrecision**

Specify the motion estimation precision. This parameter is reserved and must be zero.

*mfxU16* **MaxDecFrameBuffering**

Specifies the maximum number of frames buffered in a DPB. A value of zero means unspecified.

*mfxU16* **AUDelimiter**

Set this flag to insert the Access Unit Delimiter NAL. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **PicTimingSEI**

Set this flag to insert the picture timing SEI with pic_struct syntax element. See sub-clauses D.1.2 and D.2.2 of the ISO/IEC 14496-10 specification for the definition of this syntax element. See the CodingOptionValue enumerator for values of this option. The default value is ON.

*mfxU16* **VuiNalHrdParameters**

Set this flag to insert NAL HRD parameters in the VUI header. See the CodingOptionValue enumerator for values of this option.

### mfxExtCodingOption2

struct **mfxExtCodingOption2**

Used with the *mfxExtCodingOption* structure to specify additional options for encoding.

The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and to the *mfxEncodeCtrl* during runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION2.

*mfxU16* **IntRefType**

Specifies intra refresh type. See the IntraRefreshTypes. The major goal of intra refresh is improvement of error resilience without significant impact on encoded bitstream size caused by I-frames. The encoder achieves this by encoding part of each frame in the refresh cycle using intra MBs.

This parameter is valid during initialization and runtime. When used with temporal scalability, intra refresh applied only to base layer.

MFX_REFRESH_NO No refresh.

MFX_REFRESH_VERTICAL Vertical refresh, by column of MBs.

MFX_REFRESH_HORIZONTAL Horizontal refresh, by rows of MBs.

MFX_REFRESH_SLICE Horizontal refresh by slices without overlapping.

MFX_REFRESH_SLICE Library ignores IntRefCycleSize (size of refresh cycle equals number slices).

*mfxU16* **IntRefCycleSize**

Specifies number of pictures within refresh cycle starting from 2. 0 and 1 are invalid values. This parameter is valid only during initialization.

*mfxI16* **IntRefQPDelta**

Specifies QP difference for inserted intra MBs. Signed values are in the -51 to 51 range. This parameter is valid during initialization and runtime.

*mfxU32* **MaxFrameSize**

Specify maximum encoded frame size in byte. This parameter is used in VBR based bitrate control modes and ignored in others. The encoder tries to keep frame size below specified limit but minor overshoots are possible to preserve visual quality. This parameter is valid during initialization and runtime. It is recommended to set MaxFrameSize to 5x-10x target frame size ((TargetKbps*1000)/(8* FrameRateExtN/FrameRateExtD)) for I-frames and 2x-4x target frame size for P- and B-frames.

*mfxU32* **MaxSliceSize**

Specify maximum slice size in bytes. If this parameter is specified other controls over number of slices are ignored.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU16* **BitrateLimit**

Modifies bitrate to be in the range imposed by the encoder. The default value is ON, that is, bitrate is limited. Setting this flag to OFF may lead to violation of HRD conformance.Specifying bitrate below the encoder range might significantly affect quality.

If set to ON, this option takes effect in non CQP modes: if TargetKbps is not in the range imposed by the encoder, it will be changed to be in the range.

This parameter is valid only during initialization. Flag works with MFX_CODEC_AVC only, it is ignored with other codecs. See the CodingOptionValue enumerator for values of this option.

*Deprecated:*

Deprecated in API version 2.9

*mfxU16* **MBBRC**

Setting this flag enables macroblock level bitrate control that generally improves subjective visual quality. Enabling this flag may have negative impact on performance and objective visual quality metric. See the CodingOptionValue enumerator for values of this option. The default value depends on target usage settings.

*mfxU16* **ExtBRC**

Set this option to ON to enable external BRC. See the CodingOptionValue enumerator for values of this option. Use the Query API function to check if this feature is supported.

*mfxU16* **LookAheadDepth**

Specifies the depth of the look ahead rate control algorithm. The depth value is the number of frames that the encoder analyzes before encoding. Values are in the 10 to 100 range, inclusive. To instruct the encoder to use the default value the application should zero this field.

*mfxU16* **Trellis**

Used to control trellis quantization in AVC encoder. See TrellisControl enumerator for values of this option. This parameter is valid only during initialization.

*mfxU16* **RepeatPPS**

Controls picture parameter set repetition in AVC encoder. Set this flag to ON to repeat PPS with each frame. See the CodingOptionValue enumerator for values of this option. The default value is ON. This parameter is valid only during initialization.

*mfxU16* **BRefType**

Controls usage of B-frames as reference. See BRefControl enumerator for values of this option. This parameter is valid only during initialization.

*mfxU16* **AdaptiveI**

Controls insertion of I-frames by the encoder. Set this flag to ON to allow changing of frame type from P and B to I. This option is ignored if GopOptFlag in *mfxInfoMFX* structure is equal to MFX_GOP_STRICT.

See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

*mfxU16* **AdaptiveB**

Controls changing of frame type from B to P. Set this flag to ON enable changing of frame type from B to P. This option is ignored if GopOptFlag in *mfxInfoMFX* structure is equal to MFX_GOP_STRICT. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

*mfxU16* **LookAheadDS**

Controls down sampling in look ahead bitrate control mode. See LookAheadDownSampling enumerator for values of this option. This parameter is valid only during initialization.

*mfxU16* **NumMbPerSlice**

Specifies suggested slice size in number of macroblocks. The library can adjust this number based on platform capability. If this option is specified, that is, if it is not equal to zero, the library ignores *mfxInfoMFX::NumSlice* parameter.

*mfxU16* **SkipFrame**

Enables usage of *mfxEncodeCtrl::SkipFrame* parameter. See the SkipFrame enumerator for values of this option.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU8* **MinQPI**

Minimum allowed QP value for I-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU8* **MaxQPI**

Maximum allowed QP value for I-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU8* **MinQPP**

Minimum allowed QP value for P-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU8* **MaxQPP**

Maximum allowed QP value for P-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU8* **MinQPB**

Minimum allowed QP value for B-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU8* **MaxQPB**

Maximum allowed QP value for B-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU16* **FixedFrameRate**

Sets fixed_frame_rate_flag in VUI.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU16* **DisableDeblockingIdc**

Disables deblocking.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU16* **DisableVUI**

Completely disables VUI in the output bitstream.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU16* **BufferingPeriodSEI**

Controls insertion of buffering period SEI in the encoded bitstream. It should be one of the following values:

MFX_BPSEI_DEFAULT Encoder decides when to insert BP SEI,

MFX_BPSEI_IFRAME BP SEI should be inserted with every I-frame.

*mfxU16* **EnableMAD**

Set this flag to ON to enable per-frame reporting of Mean Absolute Difference. This parameter is valid only during initialization.

*mfxU16* **UseRawRef**

Set this flag to ON to use raw frames for reference instead of reconstructed frames. This parameter is valid during initialization and runtime (only if was turned ON during initialization).

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

### mfxExtCodingOption3

struct **mfxExtCodingOption3**

Used with *mfxExtCodingOption* and *mfxExtCodingOption2* structures to specify additional options for encoding. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and to the *mfxEncodeCtrl* during runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION3.

*mfxU16* **NumSliceI**

The number of slices for I-frames.

**Note:** Not all codecs and implementations support these values. Use the Query API function to check if this feature is supported

*mfxU16* **NumSliceP**

The number of slices for P-frames.

**Note:** Not all codecs and implementations support these values. Use the Query API function to check if this feature is supported *mfxU16* **NumSliceB**

The number of slices for B-frames.

---

**Note:** Not all codecs and implementations support these values. Use the Query API function to check if this feature is supported

---

*mfxU16* **WinBRCMaxAvgKbps**

When rate control method is MFX_RATECONTROL_VBR, MFX_RATECONTROL_LA, MFX_RATECONTROL_LA_HRD, or MFX_RATECONTROL_QVBR this parameter specifies the maximum bitrate averaged over a sliding window specified by WinBRCSize. For MFX_RATECONTROL_CBR this parameter is ignored and equals TargetKbps.

*mfxU16* **WinBRCSize**

When rate control method is MFX_RATECONTROL_CBR, MFX_RATECONTROL_VBR, MFX_RATECONTROL_LA, MFX_RATECONTROL_LA_HRD, or MFX_RATECONTROL_QVBR this parameter specifies sliding window size in frames. Set this parameter to zero to disable sliding window.

*mfxU16* **QVBRQuality**

When rate control method is MFX_RATECONTROL_QVBR, this parameter specifies quality factor. Values are in the 1 to 51 range, where 1 corresponds to the best quality.

*mfxU16* **EnableMBQP**

Set this flag to ON to enable per-macroblock QP control. Rate control method must be MFX_RATECONTROL_CQP. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

*mfxU16* **IntRefCycleDist**

Distance between the beginnings of the intra-refresh cycles in frames. Zero means no distance between cycles.

*mfxU16* **DirectBiasAdjustment**

Set this flag to ON to enable the ENC mode decision algorithm to bias to fewer B Direct/Skip types. Applies only to B-frames, all other frames will ignore this setting. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **GlobalMotionBiasAdjustment**

Enables global motion bias. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **MVCostScalingFactor**

Values are:

- 0: Set MV cost to be 0.
- 1: Scale MV cost to be 1/2 of the default value.

- 2: Scale MV cost to be 1/4 of the default value.
- 3: Scale MV cost to be 1/8 of the default value.

*mfxU16* **MBDisableSkipMap**

Set this flag to ON to enable usage of *mfxExtMBDisableSkipMap*. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

*mfxU16* **WeightedPred**

Weighted prediction mode. See the WeightedPred enumerator for values of these options.

*mfxU16* **WeightedBiPred**

Weighted prediction mode. See the WeightedPred enumerator for values of these options.

*mfxU16* **AspectRatioInfoPresent**

Instructs encoder whether aspect ratio info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **OverscanInfoPresent**

Instructs encoder whether overscan info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **OverscanAppropriate**

ON indicates that the cropped decoded pictures output are suitable for display using overscan. OFF indicates that the cropped decoded pictures output contain visually important information in the entire region out to the edges of the cropping rectangle of the picture. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **TimingInfoPresent**

Instructs encoder whether frame rate info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **BitstreamRestriction**

Instructs encoder whether bitstream restriction info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **LowDelayHrd**

Corresponds to AVC syntax element low_delay_hrd_flag (VUI). See the CodingOptionValue enumerator for values of this option.

*mfxU16* **MotionVectorsOverPicBoundaries**

When set to OFF, no sample outside the picture boundaries and no sample at a fractional sample position for which the sample value is derived using one or more samples outside the picture boundaries is used for inter prediction of any sample.

When set to ON, one or more samples outside picture boundaries may be used in inter prediction.

See the CodingOptionValue enumerator for values of this option.

*mfxU16* **ScenarioInfo**

Provides a hint to encoder about the scenario for the encoding session. See the ScenarioInfo enumerator for values of this option.

*mfxU16* **ContentInfo**

Provides a hint to encoder about the content for the encoding session. See the ContentInfo enumerator for values of this option.

*mfxU16* **PRefType**

When GopRefDist=1, specifies the model of reference list construction and DPB management. See the PRefType enumerator for values of this option.

*mfxU16* **FadeDetection**

Instructs encoder whether internal fade detection algorithm should be used for calculation of weigh/offset values for pred_weight_table unless application provided *mfxExtPredWeightTable* for this frame. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **GPB**

Set this flag to OFF to make HEVC encoder use regular P-frames instead of GPB. See the CodingOption-Value enumerator for values of this option.

*mfxU32* **MaxFrameSizeI**

Same as *mfxExtCodingOption2::MaxFrameSize* but affects only I-frames. MaxFrameSizeI must be set if MaxFrameSizeP is set. If MaxFrameSizeI is not specified or greater than spec limitation, spec limitation will be applied to the sizes of I-frames.

*mfxU32* **MaxFrameSizeP**

Same as *mfxExtCodingOption2::MaxFrameSize* but affects only P/B-frames. If MaxFrameSizeP equals 0, the library sets MaxFrameSizeP equal to MaxFrameSizeI. If MaxFrameSizeP is not specified or greater than spec limitation, spec limitation will be applied to the sizes of P/B-frames.

*mfxU16* **EnableQPOffset**

Enables QPOffset control. See the CodingOptionValue enumerator for values of this option.

*mfxI16* **QPOffset**[8]

Specifies QP offset per pyramid layer when EnableQPOffset is set to ON and RateControlMethod is CQP.

For B-pyramid, B-frame QP = QPB + QPOffset[layer].

For P-pyramid, P-frame QP = QPP + QPOffset[layer].

*mfxU16* **NumRefActiveP**[8]

Max number of active references for P-frames. Array index is pyramid layer.

*mfxU16* **NumRefActiveBL0**[8]

Max number of active references for B-frames in reference picture list 0. Array index is pyramid layer.

*mfxU16* **NumRefActiveBL1**[8]

Max number of active references for B-frames in reference picture list 1. Array index is pyramid layer.

*mfxU16* **TransformSkip**

For HEVC if this option is turned ON, the transform_skip_enabled_flag will be set to 1 in PPS. OFF specifies that transform_skip_enabled_flag will be set to 0.

*mfxU16* **TargetChromaFormatPlus1**

Minus 1 specifies target encoding chroma format (see ChromaFormatIdc enumerator). May differ from the source format. TargetChromaFormatPlus1 = 0 specifies the default target chroma format which is equal to source (mfxVideoParam::mfx::FrameInfo::ChromaFormat + 1), except RGB4 source format. In case of RGB4 source format default target , chroma format is 4:2:0 (instead of 4:4:4) for the purpose of backward compatibility.

*mfxU16* **TargetBitDepthLuma**

Target encoding bit-depth for luma samples. May differ from source bit-depth. 0 specifies a default target bit-depth that is equal to source (mfxVideoParam::mfx::FrameInfo::BitDepthLuma).

*mfxU16* **TargetBitDepthChroma**

Target encoding bit-depth for chroma samples. May differ from source bit-depth. 0 specifies a default target bit-depth that is equal to source (mfxVideoParam::mfx::FrameInfo::BitDepthChroma).

*mfxU16* **BRCPanicMode**

Controls panic mode in AVC and MPEG2 encoders.

*mfxU16* **LowDelayBRC**

When rate control method is MFX_RATECONTROL_VBR, MFX_RATECONTROL_QVBR or MFX_RATECONTROL_VCM this parameter specifies frame size tolerance. Set this parameter to MFX_CODINGOPTION_ON to allow strictly obey average frame size set by MaxKbps, for example cases when MaxFrameSize == (MaxKbps*1000)/(8* FrameRateExtN/FrameRateExtD). Also MaxFrameSizeI and MaxFrameSizeP can be set separately.

*mfxU16* **EnableMBForceIntra**

Set this flag to ON to enable usage of *mfxExtMBForceIntra* for AVC encoder. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

*mfxU16* **AdaptiveMaxFrameSize**

If this flag is set to ON, BRC may decide a larger P- or B-frame size than what MaxFrameSizeP dictates when the scene change is detected. It may benefit the video quality. AdaptiveMaxFrameSize feature is not supported with LowPower ON or if the value of MaxFrameSizeP = 0.

*mfxU16* **RepartitionCheckEnable**

Controls AVC encoder attempts to predict from small partitions. Default value allows encoder to choose preferred mode. MFX_CODINGOPTION_ON forces encoder to favor quality and MFX_CODINGOPTION_OFF forces encoder to favor performance.

*mfxU16* **EncodedUnitsInfo**

Set this flag to ON to make encoded units info available in *mfxExtEncodedUnitsInfo*.

*mfxU16* **EnableNalUnitType**

If this flag is set to ON, the HEVC encoder uses the NAL unit type provided by the application in the *mfxEncodeCtrl::MfxNalUnitType* field. This parameter is valid only during initialization.

**Note:** Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

*mfxU16* **AdaptiveLTR**

If this flag is set to ON, encoder will mark, modify, or remove LTR frames based on encoding parameters and content properties. Turn OFF to prevent Adaptive marking of Long Term Reference Frames.

*mfxU16* **AdaptiveCQM**

If this flag is set to ON, encoder adaptively selects one of implementation-defined quantization matrices for each frame. Non-default quantization matrices aim to improve subjective visual quality under certain conditions. Their number and definitions are API implementation specific. If this flag is set to OFF, default quantization matrix is used for all frames. This parameter is valid only during initialization.

*mfxU16* **AdaptiveRef**

If this flag is set to ON, encoder adaptively selects list of reference frames to improve encoding quality. Enabling of the flag can increase computation complexity and introduce additional delay. If this flag is set to OFF, regular reference frames are used for encoding.

### mfxExtCodingOptionSPSPPS

struct **mfxExtCodingOptionSPSPPS**

Attach this structure as part of the extended buffers to configure the encoder during MFXVideoENCODE_Init. The sequence or picture parameters specified by this structure overwrite any parameters specified by the structure or any other attached extended buffers attached.

For H.264, SPSBuffer and PPSBuffer must point to valid bitstreams that contain the sequence parameter set and picture parameter set, respectively.

For MPEG-2, SPSBuffer must point to valid bitstreams that contain the sequence header followed by any sequence header extension. The PPSBuffer pointer is ignored.

The encoder imports parameters from these buffers. If the encoder does not support the specified parameters, the encoder does not initialize and returns the status code MFX_ERR_INCOMPATIBLE_VIDEO_PARAM.

Check with the MFXVideoENCODE_Query function for the support of this multiple segment encoding feature. If this feature is not supported, the query returns MFX_ERR_UNSUPPORTED.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION_SPSPPS.

*mfxU8* ***SPSBuffer**

Pointer to a valid bitstream that contains the SPS (sequence parameter set for H.264 or sequence header followed by any sequence header extension for MPEG-2) buffer. Can be NULL to skip specifying the SPS.

*mfxU8* ***PPSBuffer**

Pointer to a valid bitstream that contains the PPS (picture parameter set for H.264 or picture header followed by any picture header extension for MPEG-2) buffer. Can be NULL to skip specifying the PPS.

*mfxU16* **SPSBufSize**

Size of the SPS in bytes.

*mfxU16* **PPSBufSize**

Size of the PPS in bytes.

*mfxU16* **SPSId**

SPS identifier. The value is reserved and must be zero.

*mfxU16* **PPSId**

PPS identifier. The value is reserved and must be zero.

## mfxExtCodingOptionVPS

struct **mfxExtCodingOptionVPS**

Attach this structure as part of the extended buffers to configure the encoder during MFXVideoENCODE_Init. The sequence or picture parameters specified by this structure overwrite any parameters specified by the structure or any other attached extended buffers attached.

If the encoder does not support the specified parameters, the encoder does not initialize and returns the status code MFX_ERR_INCOMPATIBLE_VIDEO_PARAM.

Check with the MFXVideoENCODE_Query function for the support of this multiple segment encoding feature. If this feature is not supported, the query returns MFX_ERR_UNSUPPORTED.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION_VPS.

*mfxU8* ***VPSBuffer**

Pointer to a valid bitstream that contains the VPS (video parameter set for HEVC) buffer.

*mfxU16* **VPSBufSize**

Size of the VPS in bytes.

*mfxU16* **VPSId**

VPS identifier; the value is reserved and must be zero.

## mfxExtDirtyRect

struct **mfxExtDirtyRect**

Used by the application to specify dirty regions within a frame during encoding. It may be used at initialization or at runtime.

Dirty rectangle definition is using end-point exclusive notation. In other words, the pixel with (Right, Bottom) coordinates lies immediately outside of the dirty rectangle. Left, Top, Right, Bottom should be aligned by codec-specific block boundaries (should be dividable by 16 for AVC, or by block size (8, 16, 32 or 64, depends on platform) for HEVC).

Every dirty rectangle with unaligned coordinates will be expanded to a minimal-area block-aligned dirty rectangle, enclosing the original one. For example, a (5, 5, 15, 31) dirty rectangle will be expanded to (0, 0, 16, 32) for AVC encoder, or to (0, 0, 32, 32) for HEVC, if block size is 32.

Dirty rectangle (0, 0, 0, 0) is a valid dirty rectangle and means that the frame is not changed.

### Dirty rectangle coordinates

The following structure members are used by the Rect array contained in the parent structure.

*mfxU32* **Left**

Dirty region left coordinate.

*mfxU32* **Top**

Dirty region top coordinate.

*mfxU32* **Right**

Dirty region right coordinate.

*mfxU32* **Bottom**

Dirty region bottom coordinate.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DIRTY_RECTANGLES.

*mfxU16* **NumRect**

Number of dirty rectangles.

struct *mfxExtDirtyRect*::[anonymous] **Rect**[256]

Array of dirty rectangles.

### mfxExtEncodedUnitsInfo

struct **mfxExtEncodedUnitsInfo**

If *mfxExtCodingOption3::EncodedUnitsInfo* was set to MFX_CODINGOPTION_ON during encoder initialization, the *mfxExtEncodedUnitsInfo* structure is attached to the *mfxBitstream* structure during encoding. It is used to report information about coding units in the resulting bitstream.

The number of filled items in UnitInfo is min(NumUnitsEncoded, NumUnitsAlloc).

For counting a minimal amount of encoded units you can use the following algorithm:

```
nSEI = amountOfApplicationDefinedSEI;
if (CodingOption3.NumSlice[IPB] != 0 || mfxVideoParam.mfx.NumSlice != 0)
  ExpectedAmount = 10 + nSEI + Max(CodingOption3.NumSlice[IPB], mfxVideoParam.mfx.
↪NumSlice);
else if (CodingOption2.NumMBPerSlice != 0)
  ExpectedAmount = 10 + nSEI + (FrameWidth * FrameHeight) / (256 * CodingOption2.
↪NumMBPerSlice);
else if (CodingOption2.MaxSliceSize != 0)
  ExpectedAmount = 10 + nSEI + Round(MaxBitrate / (FrameRate*CodingOption2.
↪MaxSliceSize));
else
  ExpectedAmount = 10 + nSEI;

if (mfxFrameInfo.PictStruct != MFX_PICSTRUCT_PROGRESSIVE)
  ExpectedAmount = ExpectedAmount * 2;

if (temporalScaleabilityEnabled)
  ExpectedAmount = ExpectedAmount * 2;
```

**Note:** Only supported by the AVC encoder.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODED_UNITS_INFO.

*mfxEncodedUnitInfo* ***UnitInfo**

Pointer to an array of mfxEncodedUnitsInfo structures whose size is equal to or greater than NumUnitsAlloc.

*mfxU16* **NumUnitsAlloc**

UnitInfo array size.

*mfxU16* **NumUnitsEncoded**

Output field. Number of coding units to report. If NumUnitsEncoded is greater than NumUnitsAlloc, the UnitInfo array will contain information only for the first NumUnitsAlloc units. User may consider reallocating the UnitInfo array to avoid this for subsequent frames.

### mfxExtEncoderCapability

struct **mfxExtEncoderCapability**

Used to retrieve encoder capability. See the description of mode 4 of the MFXVideoENCODE_Query function for details on how to use this structure.

---

**Note:** Not all implementations of the encoder support this extended buffer. The application must use query mode 1 to determine if the functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE then the functionality is supported.

---

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODER_CAPABILITY.

*mfxU32* **MBPerSec**

Specify the maximum processing rate in macro blocks per second.

### mfxExtEncoderIPCMArea

struct **mfxExtEncoderIPCMArea**

Specifies rectangle areas for IPCM coding mode.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODER_IPCM_AREA.

struct *mfxExtEncoderIPCMArea*::*area* ***Areas**

Array of areas.

struct **area**

Number of areas

### Public Members

*mfxU32* **Left**

Left area coordinate.

*mfxU32* **Top**

Top area coordinate.

*mfxU32* **Right**

Right area coordinate.

*mfxU32* **Bottom**

Bottom area coordinate.

## mfxExtEncoderResetOption

struct **mfxExtEncoderResetOption**

Used to control the encoder behavior during reset. By using this structure, the application instructs the encoder to start a new coded sequence after reset or to continue encoding of the current sequence.

This structure is also used in mode 3 of the MFXVideoENCODE_Query function to check for reset outcome before actual reset. The application should set StartNewSequence to the required behavior and call the query function. If the query fails (see status codes below), then reset is not possible in current encoder state. If the application sets StartNewSequence to MFX_CODINGOPTION_UNKNOWN, then the query function replaces the coding option with the actual reset type: MFX_CODINGOPTION_ON if the encoder will begin a new sequence after reset or MFX_CODINGOPTION_OFF if the encoder will continue the current sequence.

Using this structure may cause one of the following status codes from the MFXVideoENCODE_Reset and MFXVideoENCODE_Queryfunctions:

- MFX_ERR_INVALID_VIDEO_PARAM If a reset is not possible. For example, the application sets StartNewSequence to off and requests resolution change.
- MFX_ERR_INCOMPATIBLE_VIDEO_PARAM If the application requests change that leads to memory allocation. For example, the application sets StartNewSequence to on and requests resolution change to greater than the initialization value.
- MFX_ERR_NONE If reset is possible.

The following limited list of parameters can be changed without starting a new coded sequence:

- The bitrate parameters, TargetKbps and MaxKbps, in the *mfxInfoMFX* structure.
- The number of slices, NumSlice, in the *mfxInfoMFX* structure. Number of slices should be equal to or less than the number of slices during initialization.
- The number of temporal layers in the *mfxExtAvcTemporalLayers* structure. Reset should be called immediately before encoding of frame from base layer and number of reference frames should be large enough for the new temporal layers structure.

- The quantization parameters, QPI, QPP and QPB, in the *mfxInfoMFX* structure.

The application should retrieve all cached frames before calling reset. When the Query API function checks for reset outcome, it expects that this requirement be satisfied. If it is not true and there are some cached frames inside the encoder, then the query result may differ from the reset result, because the encoder may insert an IDR frame to produce valid coded sequence. See the *Configuration Change* section for more information.

See the *Streaming and Video Conferencing Features* section for more information.

---

**Note:** Not all implementations of the encoder support this extended buffer. The application must use query mode 1 to determine if the functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE, then the functionality is supported.

---

#### Public Members

*mfxExtBuffer* **`Header`**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODER_RESET_OPTION.

*mfxU16* **`StartNewSequence`**

Instructs encoder to start new sequence after reset. Use one of the CodingOptionValue options:

- MFX_CODINGOPTION_ON The encoder completely reset internal state and begins new coded sequence after reset, including insertion of IDR frame, sequence, and picture headers.
- MFX_CODINGOPTION_OFF The encoder continues encoding of current coded sequence after reset, without insertion of IDR frame.
- MFX_CODINGOPTION_UNKNOWN Depending on the current encoder state and changes in configuration parameters, the encoder may or may not start new coded sequence. This value is also used to query reset outcome.

### mfxExtEncoderROI

struct **`mfxExtEncoderROI`**

Used by the application to specify different Region Of Interests during encoding. It may be used at initialization or at runtime.

### ROI location rectangle

The ROI rectangle definition uses end-point exclusive notation. In other words, the pixel with (Right, Bottom) coordinates lies immediately outside of the ROI. Left, Top, Right, Bottom should be aligned by codec-specific block boundaries (should be dividable by 16 for AVC, or by 32 for HEVC). Every ROI with unaligned coordinates will be expanded by the library to minimal-area block-aligned ROI, enclosing the original one. For example (5, 5, 15, 31) ROI will be expanded to (0, 0, 16, 32) for AVC encoder, or to (0, 0, 32, 32) for HEVC.

*mfxU32* **`Left`**

Left ROI's coordinate.

*mfxU32* **`Top`**

Top ROI's coordinate.

*mfxU32* **`Right`**

Right ROI's coordinate.

*mfxU32* **`Bottom`**

Bottom ROI's coordinate.

### Public Members

*mfxExtBuffer* **`Header`**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODER_ROI.

*mfxU16* **`NumROI`**

Number of ROI descriptions in array. The Query API function mode 2 returns maximum supported value (set it to 256 and query will update it to maximum supported value).

*mfxU16* **`ROIMode`**

QP adjustment mode for ROIs. Defines if Priority or DeltaQP is used during encoding.

*mfxI16* **`Priority`**

Priority of ROI. Used if ROIMode = MFX_ROI_MODE_PRIORITY.This is an absolute value in the range of -3 to 3, which will be added to the MB QP. Priority is deprecated mode and is used only for backward compatibility. Bigger value produces better quality.

*mfxI16* **`DeltaQP`**

Delta QP of ROI. Used if ROIMode = MFX_ROI_MODE_QP_DELTA. This is an absolute value in the range of -51 to 51, which will be added to the MB QP. Lesser value produces better quality.

struct *mfxExtEncoderROI*::[anonymous] **`ROI`**[256]

Array of ROIs. Different ROI may overlap each other. If macroblock belongs to several ROI, Priority from ROI with lowest index is used.

### mfxExtHEVCRegion

struct **mfxExtHEVCRegion**

Attached to the *mfxVideoParam* structure during HEVC encoder initialization. Specifies the region to encode.

**Note:** Not all implementations of the encoder support this structure. The application must use the Query API function to determine if it is supported.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_HEVC_REGION.

*mfxU32* **RegionId**

ID of region.

*mfxU16* **RegionType**

Type of region. See HEVCRegionType enumerator for the list of types.

*mfxU16* **RegionEncoding**

Set to MFX_HEVC_REGION_ENCODING_ON to encode only specified region.

### mfxExtHEVCTiles

struct **mfxExtHEVCTiles**

Configures tiles options for the HEVC encoder. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_HEVC_TILES.

*mfxU16* **NumTileRows**

Number of tile rows.

*mfxU16* **NumTileColumns**

Number of tile columns.

### mfxExtInsertHeaders

struct **mfxExtInsertHeaders**

Runtime ctrl buffer for SPS/PPS insertion with current encoding frame.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_INSERT_HEADERS.

*mfxU16* **SPS**

Tri-state option to insert SPS.

*mfxU16* **PPS**

Tri-state option to insert PPS.

*mfxU16* **reserved**[8]

### mfxExtMBDisableSkipMap

struct **mfxExtMBDisableSkipMap**

Specifies macroblock map for current frame which forces specified macroblocks to be non-skip if *mfxExtCodingOption3::MBDisableSkipMap* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* structure during runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MB_DISABLE_SKIP_MAP.

*mfxU32* **MapSize**

Macroblock map size.

*mfxU8* ***Map**

Pointer to a list of non-skip macroblock flags in raster scan order. Each flag is one byte in map. Set flag to 1 to force corresponding macroblock to be non-skip. In case of interlaced encoding, the first half of map affects the top field and the second half of map affects the bottom field.

### mfxExtMBForceIntra

struct **mfxExtMBForceIntra**

Specifies macroblock map for current frame which forces specified macroblocks to be encoded as intra if *mfxExtCodingOption3::EnableMBForceIntra* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* structure during runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MB_FORCE_INTRA.

*mfxU32* **MapSize**

Macroblock map size.

*mfxU8* ***Map**

Pointer to a list of force intra macroblock flags in raster scan order. Each flag is one byte in map. Set flag to 1 to force corresponding macroblock to be encoded as intra. In case of interlaced encoding, the first half of map affects top field and the second half of map affects the bottom field.

### mfxExtMBQP

struct **mfxExtMBQP**

Specifies per-macroblock QP for current frame if *mfxExtCodingOption3::EnableMBQP* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* structure during runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MBQP.

*mfxU32* **Pitch**

Distance in bytes between the start of two consecutive rows in the QP array.

*mfxU16* **Mode**

Defines QP update mode. See MBQPMode enumerator for more details.

*mfxU16* **BlockSize**

QP block size, valid for HEVC only during Init and Runtime.

*mfxU32* **NumQPAlloc**

Size of allocated by application QP or DeltaQP array.

*mfxU8* ***QP**

Pointer to a list of per-macroblock QP in raster scan order. In case of interlaced encoding the first half of QP array affects the top field and the second half of QP array affects the bottom field. Valid when Mode = MFX_MBQP_MODE_QP_VALUE.

For AVC, the valid range is 1 to 51.

For HEVC, the valid range is 1 to 51. Application's provided QP values should be valid. Otherwise invalid QP values may cause undefined behavior. MBQP map should be aligned for 16x16 block size. The alignment rule is (width +15 /16) && (height +15 /16).

For MPEG2, QP corresponds to quantizer_scale of the ISO*\IEC* 13818-2 specification and has a valid range of 1 to 112.

*mfxI8* ***DeltaQP**

Pointer to a list of per-macroblock QP deltas in raster scan order. For block i: QP[i] = BrcQP[i] + DeltaQP[i]. Valid when Mode = MFX_MBQP_MODE_QP_DELTA.

*mfxQPandMode* ***QPmode**

Block-granularity modes when MFX_MBQP_MODE_QP_ADAPTIVE is set.

### mfxExtMoveRect

struct **mfxExtMoveRect**

Used by the application to specify moving regions within a frame during encoding.

Destination rectangle location should be aligned to MB boundaries (should be dividable by 16). If not, the encoder truncates it to MB boundaries, for example, both 17 and 31 will be truncated to 16.

#### Destination and source rectangle location

The following structure members are used by the Rect array contained in the parent structure.

*mfxU32* **DestLeft**

Destination rectangle location.

*mfxU32* **DestTop**

Destination rectangle location.

*mfxU32* **DestRight**

Destination rectangle location.

*mfxU32* **DestBottom**

Destination rectangle location.

*mfxU32* **SourceLeft**

Source rectangle location.

*mfxU32* **SourceTop**

Source rectangle location.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MOVING_RECTANGLE.

*mfxU16* **NumRect**

Number of moving rectangles.

struct *mfxExtMoveRect*::[anonymous] **Rect**[256]

Array of moving rectangles.

## mfxExtMVOverPicBoundaries

struct **mfxExtMVOverPicBoundaries**

Instructs encoder to use or not use samples over specified picture border for inter prediction. Attached to the *mfxVideoParam* structure.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MV_OVER_PIC_BOUNDARIES.

*mfxU16* **StickTop**

When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **StickBottom**

When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **StickLeft**

When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **StickRight**

When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

### mfxExtPartialBitstreamParam

struct **mfxExtPartialBitstreamParam**

Used by an encoder to output parts of the bitstream as soon as they are ready. The application can attach this extended buffer to the *mfxVideoParam* structure at initialization. If this option is turned ON (Granularity != MFX_PARTIAL_BITSTREAM_NONE), then the encoder can output bitstream by part based on the required granularity.

This parameter is valid only during initialization and reset. Absence of this buffer means default or previously configured bitstream output behavior.

**Note:** Not all codecs and implementations support this feature. Use the Query API function to check if this feature is supported.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_PARTIAL_BITSTREAM_PARAM.

*mfxU32* **BlockSize**

Output block granularity for PartialBitstreamGranularity. Valid only for MFX_PARTIAL_BITSTREAM_BLOCK.

*mfxU16* **Granularity**

Granularity of the partial bitstream: slice/block/any, all types of granularity state in PartialBitstreamOutput enum.

### mfxExtPictureTimingSEI

struct **mfxExtPictureTimingSEI**

Configures the H.264 picture timing SEI message. The encoder ignores it if HRD information in the stream is absent and the PicTimingSEI option in the *mfxExtCodingOption* structure is turned off. See *mfxExtCodingOption* for details.

If the application attaches this structure to the *mfxVideoParam* structure during initialization, the encoder inserts the picture timing SEI message based on provided template in every access unit of coded bitstream.

If application attaches this structure to the *mfxEncodeCtrl* structure at runtime, the encoder inserts the picture timing SEI message based on provided template in access unit that represents current frame.

These parameters define the picture timing information. An invalid value of 0xFFFF indicates that application does not set the value and encoder must calculate it.

See Annex D of the ISO*/IEC* 14496-10 specification for the definition of these parameters.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_PICTURE_TIMING_SEI.

*mfxU32* **reserved**[14]

*mfxU16* **ClockTimestampFlag**

*mfxU16* **CtType**

*mfxU16* **NuitFieldBasedFlag**

*mfxU16* **CountingType**

*mfxU16* **FullTimestampFlag**

*mfxU16* **DiscontinuityFlag**

*mfxU16* **CntDroppedFlag**

*mfxU16* **NFrames**

*mfxU16* **SecondsFlag**

*mfxU16* **MinutesFlag**

*mfxU16* **HoursFlag**

*mfxU16* **SecondsValue**

*mfxU16* **MinutesValue**

*mfxU16* **HoursValue**

*mfxU32* **TimeOffset**

struct *mfxExtPictureTimingSEI*::[anonymous] **TimeStamp**[3]

### mfxExtPredWeightTable

struct **mfxExtPredWeightTable**

Specifies weighted prediction table for current frame when all of the following conditions are met:

- *mfxExtCodingOption3::WeightedPred* was set to explicit during encoder Init or Reset .
- The current frame is P-frame or *mfxExtCodingOption3::WeightedBiPred* was set to explicit during encoder Init or Reset.
- The current frame is B-frame and is attached to the *mfxEncodeCtrl* structure.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_PRED_WEIGHT_TABLE.

*mfxU16* **LumaLog2WeightDenom**

Base 2 logarithm of the denominator for all luma weighting factors. Value must be in the range of 0 to 7, inclusive.

*mfxU16* **ChromaLog2WeightDenom**

Base 2 logarithm of the denominator for all chroma weighting factors. Value must be in the range of 0 to 7, inclusive.

*mfxU16* **LumaWeightFlag**[2][32]

LumaWeightFlag[L][R] equal to 1 specifies that the weighting factors for the luma component are specified for R's entry of RefPicList L.

*mfxU16* **ChromaWeightFlag**[2][32]

ChromaWeightFlag[L][R] equal to 1 specifies that the weighting factors for the chroma component are specified for R's entry of RefPicList L.

*mfxI16* **Weights**[2][32][3][2]

The values of the weights and offsets used in the encoding processing. The value of Weights[i][j][k][m] is interpreted as: i refers to reference picture list 0 or 1; j refers to reference list entry 0-31; k refers to data for the luma component when it is 0, the Cb chroma component when it is 1 and the Cr chroma component when it is 2; m refers to weight when it is 0 and offset when it is 1

### mfxExtVP8CodingOption

struct **mfxExtVP8CodingOption**

Describes VP8 coding options.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VP8_CODING_OPTION.

*mfxU16* **Version**

Determines the bitstream version. Corresponds to the same VP8 syntax element in frame_tag.

*mfxU16* **EnableMultipleSegments**

Set this option to ON to enable segmentation. This is tri-state option. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **LoopFilterType**

Select the type of filter (normal or simple). Corresponds to VP8 syntax element filter_type.

*mfxU16* **LoopFilterLevel**[4]

Controls the filter strength. Corresponds to VP8 syntax element loop_filter_level.

*mfxU16* **SharpnessLevel**

Controls the filter sensitivity. Corresponds to VP8 syntax element sharpness_level.

*mfxU16* **NumTokenPartitions**

Specifies number of token partitions in the coded frame.

*mfxI16* **LoopFilterRefTypeDelta**[4]

Loop filter level delta for reference type (intra, last, golden, altref).

*mfxI16* **LoopFilterMbModeDelta**[4]

Loop filter level delta for MB modes.

*mfxI16* **SegmentQPDelta**[4]

QP delta for segment.

*mfxI16* **CoeffTypeQPDelta**[5]

QP delta for coefficient type (YDC, Y2AC, Y2DC, UVAC, UVDC).

*mfxU16* **WriteIVFHeaders**

Set this option to ON to enable insertion of IVF container headers into bitstream. This is tri-state option. See the CodingOptionValue enumerator for values of this option *mfxU32* **NumFramesForIVFHeader**

Specifies number of frames for IVF header when WriteIVFHeaders is ON.

### mfxExtVP9Segmentation

struct **mfxExtVP9Segmentation**

In the VP9 encoder it is possible to divide a frame into up to 8 segments and apply particular features (like delta for quantization index or for loop filter level) on a per-segment basis. "Uncompressed header" of every frame indicates if segmentation is enabled for the current frame, and (if segmentation enabled) contains full information about features applied to every segment. Every "Mode info block" of a coded frame has segment_id in the range of 0 to 7.

To enable Segmentation, the *mfxExtVP9Segmentation* structure with correct settings should be passed to the encoder. It can be attached to the *mfxVideoParam* structure during initialization or the MFXVideoENCODE_Reset call (static configuration). If the *mfxExtVP9Segmentation* buffer isn't attached during initialization, segmentation is disabled for static configuration. If the buffer isn't attached for the Reset call, the encoder continues to use static configuration for segmentation which was the default before this Reset call. If the *mfxExtVP9Segmentation* buffer with NumSegments=0 is provided during initialization or Reset call, segmentation becomes disabled for static configuration.

The buffer can be attached to the *mfxEncodeCtrl* structure during runtime (dynamic configuration). Dynamic configuration is applied to the current frame only. After encoding of the current frame, the encoder will switch to the next dynamic configuration or to static configuration if dynamic configuration is not provided for next frame).

The SegmentIdBlockSize, NumSegmentIdAlloc, and SegmentId parameters represent a segmentation map. Here, the segmentation map is an array of segment_ids (one byte per segment_id) for blocks of size NxN in raster scan order. The size NxN is specified by the application and is constant for the whole frame. If *mfxExtVP9Segmentation* is attached during initialization and/or during runtime, all three parameters should be set to proper values that do not conflict with each other and with NumSegments. If any of the parameters are not set or any conflict or error in these parameters is detected by the library, the segmentation map will be discarded.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VP9_SEGMENTATION.

*mfxU16* **NumSegments**

Number of segments for frame. Value 0 means that segmentation is disabled. Sending 0 for a particular frame will disable segmentation for this frame only. Sending 0 to the Reset API function will disable segmentation permanently. Segmentation can be enabled again by a subsequent Reset call.

*mfxVP9SegmentParam* **Segment**[8]

Array of *mfxVP9SegmentParam* structures containing features and parameters for every segment. Entries with indexes bigger than NumSegments-1 are ignored. See the *mfxVP9SegmentParam* structure for definitions of segment features and their parameters.

*mfxU16* **SegmentIdBlockSize**

Size of block (NxN) for segmentation map. See SegmentIdBlockSize enumerator for values for this option.

An encoded block that is bigger than SegmentIdBlockSize uses segment_id taken from it's top-left sub-block from the segmentation map. The application can check if a particular block size is supported by calling Query.

*mfxU32* **NumSegmentIdAlloc**

Size of buffer allocated for segmentation map (in bytes). Application must assure that NumSegmentIdAlloc is large enough to cover frame resolution with blocks of size SegmentIdBlockSize. Otherwise the segmentation map will be discarded.

*mfxU8* ***SegmentId**

Pointer to the segmentation map buffer which holds the array of segment_ids in raster scan order. The application is responsible for allocation and release of this memory. The buffer pointed to by SegmentId, provided during initialization or Reset call should be considered in use until another SegmentId is provided via Reset call (if any), or until MFXVideoENCODE_Close is called. The buffer pointed to by SegmentId provided with *mfxEncodeCtrl* should be considered in use while the input surface is locked by the library. Every segment_id in the map should be in the range of 0 to NumSegments-1. If some segment_id is out of valid range, the segmentation map cannot be applied. If the *mfxExtVP9Segmentation* buffer is attached to the *mfxEncodeCtrl* structure in runtime, SegmentId can be zero. In this case, the segmentation map from static configuration will be used.

### mfxExtVP9TemporalLayers

struct **mfxExtVP9TemporalLayers**

API allows the encoding of VP9 bitstreams that contain several subset bitstreams that differ in frame rates, also called "temporal layers".

When decoding, each temporal layer can be extracted from the coded stream and decoded separately. The *mfxExtVP9TemporalLayers* structure configures the temporal layers for the VP9 encoder. It can be attached to the *mfxVideoParam* structure during initialization or the MFXVideoENCODE_Reset call. If the *mfxExtVP9TemporalLayers* buffer isn't attached during initialization, temporal scalability is disabled. If the buffer isn't attached for the Reset call, the encoder continues to use the temporal scalability configuration that was defined before the Reset call.

In the API, temporal layers are ordered by their frame rates in ascending order. Temporal layer 0 (having the lowest frame rate) is called the base layer. Each subsequent temporal layer includes all previous layers.

The temporal scalability feature requires a minimum number of allocated reference frames (controlled by the NumRefFrame parameter). If the NumRefFrame value set by the application isn't enough to build the reference structure for the requested number of temporal layers, the library corrects the NumRefFrame value. The temporal layer structure is reset (re-started) after key-frames.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VP9_TEMPORAL_LAYERS.

*mfxVP9TemporalLayer* **Layer**[8]

The array of temporal layers. Layer[0] specifies the base layer.

The library reads layers from the array when they are defined (FrameRateScale > 0). All layers starting from first layer with FrameRateScale = 0 are ignored. The last layer that is not ignored is considered the "highest layer".

The frame rate of the highest layer is specified in the *mfxVideoParam* structure. Frame rates of lower layers are calculated using their FrameRateScale.

TargetKbps of the highest layer should be equal to the TargetKbps value specified in the *mfxVideoParam* structure. If it is not true, TargetKbps of highest temporal layers has priority.

If there are no defined layers in the Layer array, the temporal scalability feature is disabled. For example, to disable temporal scalability in runtime, the application should pass *mfxExtVP9TemporalLayers* buffer to Reset with all FrameRateScales set to 0.

### mfxQPandMode

struct **mfxQPandMode**

Specifies per-MB or per-CU mode and QP or DeltaQP value depending on the mode type.

#### Public Members

*mfxU8* **QP**

QP for MB or CU. Valid when Mode = MFX_MBQP_MODE_QP_VALUE.

For AVC, the valid range is 1 to 51.

For HEVC, the valid range is 1 to 51. The application's provided QP values should be valid, otherwise invalid QP values may cause undefined behavior.

MBQP map should be aligned for 16x16 block size. The align rule is: (width +15 /16) && (height +15 /16).

For MPEG2, the valid range is 1 to 112. QP corresponds to quantizer_scale of the ISO*\IEC* 13818-2 specification.

*mfxI8* **DeltaQP**

Per-macroblock QP delta. Valid when Mode = MFX_MBQP_MODE_QP_DELTA.

*mfxU16* **Mode**

Defines QP update mode. Can be equal to MFX_MBQP_MODE_QP_VALUE or MFX_MBQP_MODE_QP_DELTA.

### mfxVP9TemporalLayer

struct **mfxVP9TemporalLayer**

Specifies temporal layer.

### Public Members

*mfxU16* **FrameRateScale**

The ratio between the frame rates of the current temporal layer and the base layer. The library treats a particular temporal layer as "defined" if it has FrameRateScale > 0. If the base layer is defined, it must have FrameRateScale = 1. FrameRateScale of each subsequent layer (if defined) must be a multiple of and greater than the FrameRateScale value of previous layer.

*mfxU16* **TargetKbps**

Target bitrate for the current temporal layer. Ignored if RateControlMethod is CQP. If RateControlMethod is not CQP, the application must provide TargetKbps for every defined temporal layer. TargetKbps of each subsequent layer (if defined) must be greater than the TargetKbps value of the previous layer.

## mfxTemporalLayer

struct **mfxTemporalLayer**

The structure is used for universal temporal layer description.

### Public Members

*mfxU16* **FrameRateScale**

The ratio between the frame rates of the current temporal layer and the base layer. The library treats a particular temporal layer as "defined" if it has FrameRateScale > 0. If the base layer is defined, it must have FrameRateScale = 1. FrameRateScale of each subsequent layer (if defined) must be a multiple of and greater than the FrameRateScale value of previous layer.

*mfxU16* **reserved**[3]

Reserved for future use.

*mfxU32* **InitialDelayInKB**

Initial size of the Video Buffering Verifier (VBV) buffer for the current temporal layer.

**Note:** In this context, KB is 1000 bytes and Kbps is 1000 bps.

*mfxU32* **BufferSizeInKB**

Represents the maximum possible size of any compressed frames for the current temporal layer.

*mfxU32* **TargetKbps**

Target bitrate for the current temporal layer. If RateControlMethod is not CQP, the application can provide TargetKbps for every defined temporal layer. If TargetKbps per temporal layer is not set then encoder doesn't apply any special bitrate limitations for the layer.

*mfxU32* **MaxKbps**

The maximum bitrate at which the encoded data enters the Video Buffering Verifier (VBV) buffer for the current temporal layer.

*mfxU32* **reserved1**[16]

Reserved for future use.

*mfxI32* **QPI**

Quantization Parameter (QP) for I-frames for constant QP mode (CQP) for the current temporal layer. Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPI might be clipped to supported QPI range.

**Note:** Default QPI value is implementation dependent and subject to change without additional notice in this document.

*mfxI32* **QPP**

Quantization Parameter (QP) for P-frames for constant QP mode (CQP) for the current temporal layer. Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPP might be clipped to supported QPI range.

**Note:** Default QPP value is implementation dependent and subject to change without additional notice in this document.

*mfxI32* **QPB**

Quantization Parameter (QP) for B-frames for constant QP mode (CQP) for the current temporal layer. Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPI might be clipped to supported QPB range.

**Note:** Default QPB value is implementation dependent and subject to change without additional notice in this document.

*mfxU16* **reserved2**[4]

Reserved for future use.

### mfxExtTemporalLayers

struct **mfxExtTemporalLayers**

The structure is used for universal temporal layers description.

#### Public Members

*mfxU16* **NumLayers**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_UNIVERSAL_TEMPORAL_LAYERS. The number of temporal layers.

*mfxU16* **BaseLayerPID**

The priority ID of the base layer. The encoder increases the ID for each temporal layer and writes to the prefix NAL unit for AVC and HEVC.

*mfxU16* **reserved**[2]

Reserved for future use.

*mfxTemporalLayer* ***Layers**

The array of temporal layers.

*mfxU16* **reserved1**[8]

Reserved for future use.

### mfxExtAV1BitstreamParam

struct **mfxExtAV1BitstreamParam**

The structure is used by AV1 encoder with more parameter control to encode frame.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AV1_BITSTREAM_PARAM.

*mfxU16* **WriteIVFHeaders**

Tri-state option to control IVF headers insertion, default is ON. Writing IVF headers is enabled in the encoder when *mfxExtAV1BitstreamParam* is attached and its value is ON or zero. Writing IVF headers is disabled by default in the encoder when *mfxExtAV1BitstreamParam* is not attached.

### mfxExtAV1ResolutionParam

struct **mfxExtAV1ResolutionParam**

The structure is used by AV1 encoder with more parameter control to encode frame.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AV1_RESOLUTION_PARAM.

*mfxU32* **FrameWidth**

Width of the coded frame in pixels, default value is from *mfxFrameInfo*.

*mfxU32* **FrameHeight**

Height of the coded frame in pixels, default value is from *mfxFrameInfo*.

### mfxExtAV1TileParam

struct **mfxExtAV1TileParam**

The structure is used by AV1 encoder with more parameter control to encode frame.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AV1_TILE_PARAM.

*mfxU16* **NumTileRows**

Number of tile rows, default value is 1.

*mfxU16* **NumTileColumns**

Number of tile columns, default value is 1.

*mfxU16* **NumTileGroups**

Number of tile groups, it will be ignored if the tile groups num is invalid, default value is 1.

### mfxExtAV1Segmentation

struct **mfxExtAV1Segmentation**

In the AV1 encoder it is possible to divide a frame into up to 8 segments and apply particular features (like delta for quantization index or for loop filter level) on a per-segment basis. "Uncompressed header" of every frame indicates if segmentation is enabled for the current frame, and (if segmentation enabled) contains full information about features applied to every segment. Every "Mode info block" of a coded frame has segment_id in the range of 0 to 7. To enable Segmentation, the *mfxExtAV1Segmentation* structure with correct settings should be passed to the encoder. It can be attached to the *mfxVideoParam* structure during initialization or the MFXVideoENCODE_Reset call (static configuration). If the *mfxExtAV1Segmentation* buffer isn't attached during initialization, segmentation is disabled for static configuration. If the buffer isn't attached for the Reset call, the encoder continues to use static configuration for segmentation which was the default before this Reset call. If the *mfxExtAV1Segmentation* buffer with NumSegments=0 is provided during initialization or Reset call, segmentation becomes disabled for static configuration. The buffer can be attached to the *mfxEncodeCtrl* structure during runtime (dynamic configuration). Dynamic configuration is applied to the current frame only. After encoding of the current frame, the encoder will switch to the next dynamic configuration or to static configuration if dynamic configuration is not provided for next frame). The SegmentIdBlockSize, NumSegmentIdAlloc, and SegmentId parameters represent a segmentation map. Here, the segmentation map is an array of segment_ids (one byte per segment_id) for blocks of size NxN in raster scan order. The size NxN is specified by the application and is constant for the whole frame. If *mfxExtAV1Segmentation* is attached during initialization and/or during runtime, all three parameters should be set to proper values that do not conflict with each other and with NumSegments. If any of the parameters are not set or any conflict or error in these parameters is detected by the library, the segmentation map will be discarded.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AV1_SEGMENTATION.

*mfxU8* **NumSegments**

Number of segments for frame. Value 0 means that segmentation is disabled. Sending 0 for a particular frame will disable segmentation for this frame only. Sending 0 to the Reset API function will disable segmentation permanently. Segmentation can be enabled again by a subsequent Reset call.

mfxAV1SegmentParam **Segment**[8]

Array of mfxAV1SegmentParam structures containing features and parameters for every segment. Entries with indexes bigger than NumSegments-1 are ignored. See the mfxAV1SegmentParam structure for definitions of segment features and their parameters.

*mfxU16* **SegmentIdBlockSize**

Size of block (NxN) for segmentation map. See AV1 SegmentIdBlockSize enumerator for values for this option. An encoded block that is bigger than AV1 SegmentIdBlockSize uses segment_id taken from it's top-left sub-block from the segmentation map. The application can check if a particular block size is supported by calling Query.

*mfxU32* **NumSegmentIdAlloc**

Size of buffer allocated for segmentation map (in bytes). Application must assure that NumSegmentIdAlloc is large enough to cover frame resolution with blocks of size SegmentIdBlockSize. Otherwise the segmentation map will be discarded.

*mfxU8* ***SegmentIds**

Pointer to the segmentation map buffer which holds the array of segment_ids in raster scan order. The application is responsible for allocation and release of this memory. The buffer pointed to by SegmentId, provided during initialization or Reset call should be considered in use until another SegmentId is provided via Reset call (if any), or until MFXVideoENCODE_Close is called. The buffer pointed to by SegmentId provided with *mfxEncodeCtrl* should be considered in use while the input surface is locked by the library. Every segment_id in the map should be in the range of 0 to NumSegments-1. If some segment_id is out of valid range, the segmentation map cannot be applied. If the *mfxExtAV1Segmentation* buffer is attached to the *mfxEncodeCtrl* structure in runtime, SegmentId can be zero. In this case, the segmentation map from static configuration will be used.

### mfxCTUHeader

struct **mfxCTUHeader**

#### Public Members

*mfxU32* **CUcountminus1**

Number of CU per CTU.

*mfxU32* **MaxDepth**

Max quad-tree depth of CU in CTU.

*mfxU16* **CurrXAddr**

Horizontal address of CTU.

*mfxU16* **CurrYAddr**

Vertical address of CTU.

### mfxCUInfo

struct **mfxCUInfo**

#### Public Members

*mfxU32* **CU_Size**

indicates the CU size of the current CU. 0: 8x8 1: 16x16 2: 32x32 3: 64x64

*mfxU32* **CU_pred_mode**

indicates the prediction mode for the current CU. 0: intra 1: inter *mfxU32* **CU_part_mode**

indicates the PU partition mode for the current CU. 0: 2Nx2N 1: 2NxN (inter) 2: Nx2N (inter) 3: NXN (intra only, CU Size=8x8 only. Luma Intra Mode indicates the intra prediction mode for 4x4_0. The additional prediction modes are overloaded on 4x4_1, 4x4_2, 4x4_3 below) 4: 2NxnT (inter only) 5: 2NxnB (inter only) 6: nLx2N (inter only) 7: nRx2N (inter only).

*mfxU32* **InterPred_IDC_MV0**

indicates the prediction direction for PU0 of the current CU. 0: L0 1: L1 2: Bi 3: reserved

*mfxU32* **InterPred_IDC_MV1**

indicates the prediction direction for PU1 of the current CU. 0: L0 1: L1 2: Bi 3: reserved

*mfxU32* **LumaIntraMode**

Final explicit Luma Intra Mode 4x4_0 for NxN. Valid values 0..34 Note: CU_part_mode==NxN.

*mfxU32* **ChromaIntraMode**

indicates the final explicit Luma Intra Mode for the CU. 0: DM (use Luma mode, from block 0 if NxN) 1: reserved 2: Planar 3: Vertical 4: Horizontal 5: DC

*mfxU32* **LumaIntraMode4x4_1**

Final explicit Luma Intra Mode 4x4_1. Valid values 0..34 Note: CU_part_mode==NxN.

*mfxU32* **LumaIntraMode4x4_2**

Final explicit Luma Intra Mode 4x4_2. Valid values 0..34 Note: CU_part_mode==NxN.

*mfxU32* **LumaIntraMode4x4_3**

Final explicit Luma Intra Mode 4x4_3. Valid values 0..34 Note: CU_part_mode==NxN.

*mfxU32* **SAD**

distortion measure, approximation to SAD.

Will deviate significantly (pre, post reconstruction) and due to variation in algorithm.

*mfxI16Pair* **MV**[2][2]

These parameters indicate motion vectors that are associated with the PU0/PU1 winners range [-2048.00..2047.75]. L0/PU0 - MV[0][0] L0/PU1 - MV[0][1] L1/PU0 - MV[1][0] L1/PU1 - MV[1][1]

*mfxU32* **L0_MV0_RefID**

This parameter indicates the reference index associated with the MV X/Y that is populated in the L0_MV0.X and L0_MV0.Y fields.

*mfxU32* **L0_MV1_RefID**

This parameter indicates the reference index associated with the MV X/Y that is populated in the L0_MV1.X and L0_MV1.Y fields.

*mfxU32* **L1_MV0_RefID**

This parameter indicates the reference index associated with the MV X/Y that is populated in the L1_MV0.X and L1_MV0.Y fields.

*mfxU32* **L1_MV1_RefID**

This parameter indicates the reference index associated with the MV X/Y that is populated in the L1_MV1.X and L1_MV1.Y fields.

### mfxCTUInfo

struct **mfxCTUInfo**

#### Public Members

*mfxCTUHeader* **CtuHeader**

H.265 CTU header.

*mfxCUInfo* **CuInfo**[64]

Array of CU.

### mfxMBInfo

struct **mfxMBInfo**

The structure describes H.264 stats per MB.

#### Public Members

*mfxU32* **MBType**

Together with `IntraMbFlag` this parameter specifies macroblock type according to the ISO*\IEC* 14496-10 with the following difference - it stores either intra or inter values according to `IntraMbFlag`, but not intra after inter. Values for P-slices are mapped to B-slice values. For example P_16x8 is coded with B_FWD_16x8 value.

*mfxU32* **InterMBMode**

This field specifies inter macroblock mode and is ignored for intra MB. It is derived from `MbType` and has next values:

- 0 - 16x16 mode
- 1 - 16x8 mode
- 2 - 8x16 mode
- 3 - 8x8 mode

*mfxU32* **IntraMBMode**

This field specifies intra macroblock mode and is ignored for inter MB. It is derived from `MbType` and has next values:

- 0 - 16x16 mode
- 1 - 8x8 mode
- 2 - 4x4 mode
- 3 - PCM

*mfxU32* **IntraMBFlag**

This flag specifies intra/inter MB type and has next values: 0 - Inter prediction MB type 1 - Intra prediction MB type

*mfxU32* **SubMBShapes**

This field specifies subblock shapes for the current MB. Each block is described by 2 bits starting from lower bits for block 0.

- 0 - 8x8
- 1 - 8x4
- 2 - 4x8
- 3 - 4x4

*mfxU32* **SubMBShapeMode**

This field specifies prediction modes for the current MB partition blocks. Each block is described by 2 bits starting from lower bits for block 0.

- 0 - Pred_L0
- 1 - Pred_L1
- 2 - BiPred
- 3 - reserved

Only one prediction value for partition is reported, the rest values are set to zero. For example:

- 16x16 Pred_L1 - 0x01 (only 2 lower bits are used)
- 16x8 Pred_L1 / BiPred - 0x09 (1001b)
- 8x16 BiPred / BiPred - 0x0a (1010b)

For P MBs this value is always zero.

*mfxU32* **ChromaIntraPredMode**

This value specifies chroma intra prediction mode.

- 0 - DC
- 1 - Horizontal
- 2 - Vertical
- 3 - Plane

*mfxU32* **SAD**

Distortion measure, approximation to SAD.

Deviate significantly (pre, post reconstruction) and due to variation in algorithm.

*mfxI8* **Qp**

MB QP.

*mfxU16* **LumaIntraMode**[4]

These values specify luma intra prediction modes for current MB. Each element of the array corresponds to 8x8 block and each holds prediction modes for four 4x4 subblocks. Four bits per mode, lowest bits for left top subblock. All 16 prediction modes are always specified. For 8x8 case, block prediction mode is populated to all subblocks of the 8x8 block. For 16x16 case - to all subblocks of the MB.

Prediction directions for 4x4 and 8x8 blocks:

- 0 - Vertical
- 1 - Horizontal
- 2 - DC
- 3 - Diagonal Down Left
- 4 - Diagonal Down Right
- 5 - Vertical Right
- 6 - Horizontal Down
- 7 - Vertical Left
- 8 - Horizontal Up

Prediction directions for 16x16 blocks:

- 0 - Vertical
- 1 - Horizontal
- 2 - DC
- 3 - Plane

### mfxEncodeBlkStats

struct **mfxEncodeBlkStats**

The structure describes H.264 and H.265 stats per MB or CTUs.

### Public Members

*mfxU32* **NumMB**

Number of MBs per frame for H.264.

*mfxU32* **NumCTU**

number of CTUs per frame for H.265.

*mfxCTUInfo* ***HEVCCTUArray**

Array of CTU statistics.

*mfxMBInfo* ***AVCMBArray**

Array of MB statistics.

## mfxEncodeHighLevelStats

struct **mfxEncodeHighLevelStats**

The structure describes H.264/H.265 frame/slice/tile level statistics.

### Public Members

*mfxF32* **PSNRLuma**

PSNR for LUMA samples.

*mfxF32* **PSNRCb**

PSNR for Chroma (Cb) samples.

*mfxF32* **PSNRCr**

PSNR for Chroma (Cr) samples.

*mfxU64* **SADLuma**

distortion measure, approximation to SAD.

Will deviate significantly (pre, post reconstruction) and due to variation in algorithm.

*mfxU32* **NumMB**

Number of MBs per frame for H.264.

*mfxU32* **NumCTU**

number of CTUs per frame for H.265.

*mfxU32* **NumIntraBlock**

For H.264 it is always 16x16 corresponding to MB size. In H.265 it's normalized to 4x4, so for each CU we calculate number of 4x4 which belongs to the block.

*mfxU32* **NumInterBlock**

Number of intra blocks in the frame. The size of block is defined by BlockSize. For H.265 it can be more than number of intra CU.

*mfxU32* **NumSkippedBlock**

Number of inter blocks in the frame. The size of block is defined by BlockSize. For H.265 it can be more than number of inter CU.

*mfxU32* **reserved**[8]

Number of skipped blocks in the frame. The size of block is defined by BlockSize. For H.265 it can be more than number of skipped CU.

### mfxEncodeFrameStats

typedef *mfxEncodeHighLevelStats* **mfxEncodeFrameStats**

Alias for the structure to describe H.264 and H.265 frame level stats.

### mfxEncodeSliceStats

struct **mfxEncodeSliceStats**

The structure describes H.264 and H.265 stats per Slice or Tile.

#### Public Members

*mfxU32* **NumElements**

Number of Slices or Tiles per frame for H.264/H.265.

*mfxEncodeHighLevelStats* ***HighLevelStatsArray**

Array of CTU statistics.

### mfxEncodeTileStats

typedef *mfxEncodeSliceStats* **mfxEncodeTileStats**

Alias for the structure to describe H.264 and H.265 tile level stats.

### mfxEncodeStatsContainer

struct **mfxEncodeStatsContainer**

The structure represents reference counted container for output after encoding operation which includes statistics and synchronization primitive for compressed bitstream. The memory is allocated and released by the library.

#### Public Members

*mfxStructVersion* **Version**

The version of the structure.

*mfxStatus* (***SynchronizeStatistics**)(*mfxRefInterface* *ref_interface, *mfxU32* wait)

Guarantees readiness of the statistics after a function completes. Instead of MFXVideoCORE_SyncOperation which leads to the synchronization of all output objects, users may directly call the *mfxEncodeStatsContainer::SynchronizeStatistics* function to get output statistics.

< Reference counting interface.

**Param ref_interface**
**[in]** Valid interface.

**Param wait**
**[out]** Wait time in milliseconds.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If interface is NULL.

MFX_ERR_INVALID_HANDLE If any of container is not valid object .

MFX_WRN_IN_EXECUTION If the given timeout is expired and the container is not ready.

MFX_ERR_ABORTED If the specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***SynchronizeBitstream**)(*mfxRefInterface* *ref_interface, *mfxU32* wait)

Guarantees readiness of associated compressed bitstream after a function completes. Instead of MFXVideoCORE_SyncOperation which leads to the synchronization of all output objects, users may directly call the *mfxEncodeStatsContainer::SynchronizeStatistics* function to get output bitstream.

**Param ref_interface**
**[in]** Valid interface.

**Param wait**
**[out]** Wait time in milliseconds.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If interface is NULL.

MFX_ERR_INVALID_HANDLE If any of container is not valid object .

MFX_WRN_IN_EXECUTION If the given timeout is expired and the container is not ready.

MFX_ERR_ABORTED If the specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

MFX_ERR_UNKNOWN Any internal error.

### mfxExtEncodeStatsOutput

struct **mfxExtEncodeStatsOutput**

The extension buffer which should be attached by application for *mfxBitstream* buffer before encode operation. As result the encoder will allocate memory for statistics and fill appropriate structures.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODESTATS_BLK.

*mfxEncodeStatsMode* **Mode**

What statistics is required: block/slice/tile/frame level or any combinations. In case of slice or tile output statistics for one slice or tile will be available only. What encoding mode should be used to gather statistics.

*mfxEncodeStatsContainer* ***EncodeStatsContainer**

encode output, filled by the implementation.

### mfxExtHEVCRefListCtrl

typedef *mfxExtAVCRefListCtrl* **mfxExtHEVCRefListCtrl**

### mfxExtHEVCRefLists

typedef *mfxExtAVCRefLists* **mfxExtHEVCRefLists**

### mfxExtHEVCTemporalLayers

typedef *mfxExtAvcTemporalLayers* **mfxExtHEVCTemporalLayers**

### 5.2.7 VPP Structures

Structures used by VPP only.

#### API

- *mfxExtColorConversion*
- *mfxExtDecVideoProcessing*
- *mfxExtEncodedSlicesInfo*
- *mfxExtVppAuxData*
- *mfxExtVPPColorFill*
- *mfxExtVPPComposite*
- *mfxExtVPPDeinterlacing*
- *mfxExtVPPDenoise*
- *mfxExtVPPDenoise2*
- *mfxExtVPPDetail*
- *mfxExtVPPDoNotUse*
- *mfxExtVPPDoUse*
- *mfxExtVPPFieldProcessing*
- *mfxExtVPPFrameRateConversion*
- *mfxExtVPPImageStab*
- *mfxExtVppMctf*
- *mfxExtVPPMirroring*
- *mfxExtVPPProcAmp*
- *mfxExtVPPRotation*
- *mfxExtVPPScaling*
- *mfxChannel*
- *mfx3DLutSystemBuffer*
- *mfx3DLutVideoBuffer*
- *mfxExtVPP3DLut*
- *mfxExtVPPVideoSignalInfo*
- *mfxInfoVPP*
- *mfxVPPCompInputStream*
- *mfxVPPStat*
- *mfxExtVPPPercEncPrefilter*

### mfxExtColorConversion

struct **mfxExtColorConversion**

A hint structure that tunes the VPP Color Conversion algorithm when attached to the *mfxVideoParam* structure during VPP Init.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_COLOR_CONVERSION.

*mfxU16* **ChromaSiting**

See ChromaSiting enumerator for details.

ChromaSiting is applied on input or output surface depending on the scenario:

| VPP Input | VPP Output | ChromaSiting Indicates |
|---|---|---|
| MFX_CHROMAFORMAT_YUV420 MFX_CHROMAFORMAT_YUV422 | MFX_CHROMAFORMAT_YUV444 | Chroma location for input |
| MFX_CHROMAFORMAT_YUV444 | MFX_CHROMAFORMAT_YUV420 MFX_CHROMAFORMAT_YUV422 | Chroma location for output |
| MFX_CHROMAFORMAT_YUV420 | MFX_CHROMAFORMAT_YUV420 | Chroma location for input and output |
| MFX_CHROMAFORMAT_YUV420 | MFX_CHROMAFORMAT_YUV422 | Horizontal location for input and output, vertical location for input |

### mfxExtDecVideoProcessing

struct **mfxExtDecVideoProcessing**

If attached to the *mfxVideoParam* structure during the Init stage, this buffer will instruct the decoder to resize output frames via the fixed function resize engine (if supported by hardware), utilizing direct pipe connection and bypassing intermediate memory operations. The main benefits of this mode of pipeline operation are offloading resize operation to a dedicated engine, thus reducing power consumption and memory traffic.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DEC_VIDEO_PROCESSING.

struct *mfxExtDecVideoProcessing*::*mfxIn* **In**

Input surface description.

struct *mfxExtDecVideoProcessing*::*mfxOut* **Out**

Output surface description.

struct **mfxIn**

Input surface description.

### Public Members

*mfxU16* **CropX**

X coordinate of region of interest of the input surface.

*mfxU16* **CropY**

Y coordinate of region of interest of the input surface.

*mfxU16* **CropW**

Width coordinate of region of interest of the input surface.

*mfxU16* **CropH**

Height coordinate of region of interest of the input surface.

struct **mfxOut**

Output surface description.

### Public Members

*mfxU32* **FourCC**

FourCC of output surface Note: Should be MFX_FOURCC_NV12.

*mfxU16* **ChromaFormat**

Chroma Format of output surface.

**Note:** Should be MFX_CHROMAFORMAT_YUV420

*mfxU16* **Width**

Width of output surface.

*mfxU16* **Height**

Height of output surface.

*mfxU16* **CropX**

X coordinate of region of interest of the output surface.

*mfxU16* **CropY**

Y coordinate of region of interest of the output surface.

*mfxU16* **CropW**

Width coordinate of region of interest of the output surface.

*mfxU16* **CropH**

Height coordinate of region of interest of the output surface.

### mfxExtEncodedSlicesInfo

struct **mfxExtEncodedSlicesInfo**

Used by the encoder to report additional information about encoded slices. The application can attach this buffer to the *mfxBitstream* structure before calling the MFXVideoENCODE_EncodeFrameAsync function.

**Note:** Not all implementations of the encoder support this extended buffer. The application must use query mode 1 to determine if the functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE, then the functionality is supported.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODED_SLICES_INFO.

*mfxU16* **SliceSizeOverflow**

When *mfxExtCodingOption2::MaxSliceSize* is used, indicates the requested slice size was not met for one or more generated slices.

*mfxU16* **NumSliceNonCopliant**

When *mfxExtCodingOption2::MaxSliceSize* is used, indicates the number of generated slices exceeds specification limits.

*mfxU16* **NumEncodedSlice**

Number of encoded slices.

*mfxU16* **NumSliceSizeAlloc**

SliceSize array allocation size. Must be specified by application.

*mfxU16* ***SliceSize**

Slice size in bytes. Array must be allocated by application.

### mfxExtVppAuxData

struct **mfxExtVppAuxData**

Returns auxiliary data generated by the video processing pipeline. The encoding process may use the auxiliary data by attaching this structure to the *mfxEncodeCtrl* structure.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_AUXDATA.

*mfxU16* **PicStruct**

Detected picture structure - top field first, bottom field first, progressive or unknown if video processor cannot detect picture structure. See the PicStruct enumerator for definition of these values.

*mfxU16* **RepeatedFrame**

The flag signalizes that the frame is identical to the previous one.

### mfxExtVPPColorFill

struct **mfxExtVPPColorFill**

Configures the VPP ColorFill filter algorithm.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_COLORFILL.

*mfxU16* **Enable**

Set to ON makes VPP fill the area between Width/Height and Crop borders. See the CodingOptionValue enumerator for values of this option.

### mfxExtVPPComposite

struct **mfxExtVPPComposite**

Used to control composition of several input surfaces in one output. In this mode, the VPP skips any other filters. The VPP returns an error if any mandatory filter is specified and returns the filter skipped warning if an optional filter is specified. The only supported filters are deinterlacing and interlaced scaling. The only supported combinations of input and output color formats are:

- RGB to RGB,
- NV12 to NV12,

- RGB and NV12 to NV12, for per the pixel alpha blending use case.

The VPP returns MFX_ERR_MORE_DATA for additional input until an output is ready. When the output is ready, the VPP returns MFX_ERR_NONE. The application must process the output frame after synchronization.

The composition process is controlled by:

- mfxFrameInfo::CropXYWH in the input surface defines the location of the picture in the input frame.
- InputStream[i].DstXYWH defines the location of the cropped input picture in the output frame.
- mfxFrameInfo::CropXYWH in the output surface defines the actual part of the output frame. All pixels in the output frame outside this region will be filled by the specified color.

If the application uses the composition process on video streams with different frame sizes, the application should provide maximum frame size in the *mfxVideoParam* structure during the initialization, reset, or query operations.

If the application uses the composition process, the MFXVideoVPP_QueryIOSurf function returns the cumulative number of input surfaces, that is, the number required to process all input video streams. The function sets the frame size in the *mfxFrameAllocRequest* equal to the size provided by the application in the *mfxVideoParam* structure.

The composition process supports all types of surfaces.

All input surfaces should have the same type and color format, except for the per pixel alpha blending case, where it is allowable to mix NV12 and RGB surfaces.

There are three different blending use cases:

- **Luma keying.** All input surfaces should have the NV12 color format specified during VPP initialization. Part of each surface, including the first one, may be rendered transparent by using LumaKeyEnable, LumaKeyMin, and LumaKeyMax values.
- **Global alpha blending.** All input surfaces should have the same color format, NV12 or RGB, specified during VPP initialization. Each input surface, including the first one, can be blended with underlying surfaces by using GlobalAlphaEnable and GlobalAlpha values.
- **Per-pixel alpha blending.** It is allowed to mix NV12 and RGB input surfaces. Each RGB input surface, including the first one, can be blended with underlying surfaces by using PixelAlphaEnable value.

It is not allowed to mix different blending use cases in the same function call.

In the special case where the destination region of the output surface defined by output crops is fully covered with destination sub-regions of the surfaces, the fast compositing mode can be enabled. The main use case for this mode is a video-wall scenario with a fixed destination surface partition into sub-regions of potentially different size.

In order to trigger this mode, the application must cluster input surfaces into tiles, defining at least one tile by setting the NumTiles field to be greater than 0, and assigning surfaces to the corresponding tiles by setting the TileId field to the value within the 0 to NumTiles range per input surface. Tiles should also satisfy the following additional constraints:

- Each tile should not have more than 8 surfaces assigned to it.
- Tile bounding boxes, as defined by the enclosing rectangles of a union of a surfaces assigned to this tile, should not intersect.

Background color may be changed dynamically through Reset. There is no default value. YUV black is (0;128;128) or (16;128;128) depending on the sample range. The library uses a YUV or RGB triple depending on output color format.

## Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_COMPOSITE.

*mfxU16* **Y**

Y value of the background color.

*mfxU16* **R**

R value of the background color.

*mfxU16* **U**

U value of the background color.

*mfxU16* **G**

G value of the background color.

*mfxU16* **V**

V value of the background color.

*mfxU16* **B**

B value of the background color.

*mfxU16* **NumTiles**

Number of input surface clusters grouped together to enable fast compositing. May be changed dynamically at runtime through Reset.

*mfxU16* **NumInputStream**

Number of input surfaces to compose one output. May be changed dynamically at runtime through Reset. Number of surfaces can be decreased or increased, but should not exceed the number specified during initialization. Query mode 2 should be used to find the maximum supported number.

*mfxVPPCompInputStream* ***InputStream**

An array of *mfxVPPCompInputStream* structures that describe composition of input video streams. It should consist of exactly NumInputStream elements.

### mfxExtVPPDeinterlacing

struct **mfxExtVPPDeinterlacing**

Used by the application to specify different deinterlacing algorithms.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DEINTERLACING.

*mfxU16* **Mode**

Deinterlacing algorithm. See the DeinterlacingMode enumerator for details.

*mfxU16* **TelecinePattern**

Specifies telecine pattern when Mode = MFX_DEINTERLACING_FIXED_TELECINE_PATTERN. See the TelecinePattern enumerator for details.

*mfxU16* **TelecineLocation**

Specifies position inside a sequence of 5 frames where the artifacts start when TelecinePattern = MFX_TELECINE_POSITION_PROVIDED

*mfxU16* **reserved**[9]

Reserved for future use.

### mfxExtVPPDenoise

struct **mfxExtVPPDenoise**

A hint structure that configures the VPP denoise filter algorithm.

*Deprecated:*

Deprecated in API version 2.5. Use *mfxExtVPPDenoise2* instead.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DENOISE.

*mfxU16* **DenoiseFactor**

Indicates the level of noise to remove. Value range of 0 to 100 (inclusive).

### mfxExtVPPDenoise2

struct **mfxExtVPPDenoise2**

A hint structure that configures the VPP denoise filter algorithm.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DENOISE2.

*mfxDenoiseMode* **Mode**

Indicates the mode of denoise. mfxDenoiseMode enumerator.

*mfxU16* **Strength**

Denoise strength in manual mode. Value of 0-100 (inclusive) indicates the strength of denoise. The strength of denoise controls degree of possible changes of pixel values; the bigger the strength the larger the change is.

*mfxU16* **reserved**[15]

### mfxExtVPPDetail

struct **mfxExtVPPDetail**

A hint structure that configures the VPP detail/edge enhancement filter algorithm.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DETAIL.

*mfxU16* **DetailFactor**

Indicates the level of details to be enhanced. Value range of 0 to 100 (inclusive).

### mfxExtVPPDoNotUse

struct **mfxExtVPPDoNotUse**

Tells the VPP not to use certain filters in pipeline. See "Configurable VPP filters" table for complete list of configurable filters. The user can attach this structure to the *mfxVideoParam* structure when initializing video processing.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DONOTUSE.

*mfxU32* **NumAlg**

Number of filters (algorithms) not to use

*mfxU32* ***AlgList**

Pointer to a list of filters (algorithms) not to use

## mfxExtVPPDoUse

struct **mfxExtVPPDoUse**

Tells the VPP to include certain filters in the pipeline.

Each filter may be included in the pipeline in one of two different ways:

- Adding a filter ID to this structure. In this method, the default filter parameters are used.
- Attaching a filter configuration structure directly to the *mfxVideoParam* structure. In this method, adding filter ID to the *mfxExtVPPDoUse* structure is optional.

See Table "Configurable VPP filters" for complete list of configurable filters, their IDs, and configuration structures.

The user can attach this structure to the *mfxVideoParam* structure when initializing video processing.

**Note:** MFX_EXTBUFF_VPP_COMPOSITE cannot be enabled using *mfxExtVPPDoUse* because default parameters are undefined for this filter. The application must attach the appropriate filter configuration structure directly to the *mfxVideoParam* structure to enable it.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DOUSE.

*mfxU32* **NumAlg**

Number of filters (algorithms) to use

*mfxU32* ***AlgList**

Pointer to a list of filters (algorithms) to use

### mfxExtVPPFieldProcessing

struct **mfxExtVPPFieldProcessing**

Configures the VPP field processing algorithm. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and/or to the *mfxFrameData* during runtime. Runtime configuration has priority over initialization configuration. If the field processing algorithm was activated via the *mfxExtVPPDoUse* structure and the *mfxExtVPPFieldProcessing* extended buffer was not provided during initialization, this buffer must be attached to the *mfxFrameData* structure of each input surface.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_FIELD_PROCESSING.

*mfxU16* **Mode**

Specifies the mode of the field processing algorithm. See the VPPFieldProcessingMode enumerator for values of this option.

*mfxU16* **InField**

When Mode is MFX_VPP_COPY_FIELD, specifies input field. See the PicType enumerator for values of this parameter.

*mfxU16* **OutField**

When Mode is MFX_VPP_COPY_FIELD, specifies output field. See the PicType enumerator for values of this parameter.

### mfxExtVPPFrameRateConversion

struct **mfxExtVPPFrameRateConversion**

Configures the VPP frame rate conversion filter. The user can attach this structure to the *mfxVideoParam* structure when initializing, resetting, or querying capability of video processing.

On some platforms the advanced frame rate conversion algorithm (the algorithm based on frame interpolation) is not supported. To query its support, the application should add the MFX_FRCALGM_FRAME_INTERPOLATION flag to the Algorithm value in the *mfxExtVPPFrameRateConversion* structure, attach it to the structure, and call the MFXVideoVPP_Query function. If the filter is supported, the function returns a MFX_ERR_NONE status and copies the content of the input structure to the output structure. If an advanced filter is not supported, then a simple filter will be used and the function returns MFX_WRN_INCOMPATIBLE_VIDEO_PARAM, copies content of the input structure to the output structure, and corrects the Algorithm value.

If advanced FRC algorithm is not supported, both MFXVideoVPP_Init and MFXVideoVPP_Reset functions return the MFX_WRN_INCOMPATIBLE_VIDEO_PARAM status.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION.

*mfxU16* **Algorithm**

See the FrcAlgm enumerator for a list of frame rate conversion algorithms.

## mfxExtVPPImageStab

struct **mfxExtVPPImageStab**

A hint structure that configures the VPP image stabilization filter.

On some platforms this filter is not supported. To query its support, the application should use the same approach that it uses to configure VPP filters: adding the filter ID to the *mfxExtVPPDoUse* structure or by attaching the *mfxExtVPPImageStab* structure directly to the *mfxVideoParam* structure and calling the MFXVideoVPP_Query function.

If this filter is supported, the function returns a MFX_ERR_NONE status and copies the content of the input structure to the output structure. If the filter is not supported, the function returns MFX_WRN_FILTER_SKIPPED, removes the filter from the *mfxExtVPPDoUse* structure, and zeroes the *mfxExtVPPImageStab* structure.

If the image stabilization filter is not supported, both MFXVideoVPP_Init and MFXVideoVPP_Reset functions return a MFX_WRN_FILTER_SKIPPED status.

The application can retrieve the list of active filters by attaching the *mfxExtVPPDoUse* structure to the *mfxVideoParam* structure and calling the MFXVideoVPP_GetVideoParam function. The application must allocate enough memory for the filter list.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_IMAGE_STABILIZATION.

*mfxU16* **Mode**

Image stabilization mode. See ImageStabMode enumerator for values.

## mfxExtVppMctf

struct **mfxExtVppMctf**

Provides setup for the Motion-Compensated Temporal Filter (MCTF) during the VPP initialization and for control parameters at runtime. By default, MCTF is off. An application may enable it by adding MFX_EXTBUFF_VPP_MCTF to the *mfxExtVPPDoUse* buffer or by attaching *mfxExtVppMctf* to the *mfxVideoParam* structure during initialization or reset.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_MCTF.

*mfxU16* **FilterStrength**

Value in range of 0 to 20 (inclusive) to indicate the filter strength of MCTF.

The strength of the MCTF process controls the degree of possible change of pixel values eligible for MCTF - the greater the strength value, the larger the change. It is a dimensionless quantity - values in the range of 1 to 20 inclusively imply strength; value 0 stands for AUTO mode and is valid during initialization or reset only

If an invalid value is given, it is fixed to the default value of 0. If the field value is in the range of 1 to 20 inclusive, MCTF operates in fixed-strength mode with the given strength of MCTF process.

At runtime, values of 0 and greater than 20 are ignored.

### mfxExtVPPMirroring

struct **mfxExtVPPMirroring**

Configures the VPP Mirroring filter algorithm.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_MIRRORING.

*mfxU16* **Type**

Mirroring type. See MirroringType for values.

### mfxExtVPPProcAmp

struct **mfxExtVPPProcAmp**

A hint structure that configures the VPP ProcAmp filter algorithm. The structure parameters will be clipped to their corresponding range and rounded by their corresponding increment.

**Note:** There are no default values for fields in this structure, all settings must be explicitly specified every time this buffer is submitted for processing.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_PROCAMP.

*mfxF64* **Brightness**

The brightness parameter is in the range of -100.0F to 100.0F, in increments of 0.1F. Setting this field to 0.0F will disable brightness adjustment.

*mfxF64* **Contrast**

The contrast parameter in the range of 0.0F to 10.0F, in increments of 0.01F, is used for manual contrast adjustment. Setting this field to 1.0F will disable contrast adjustment. If the parameter is negative, contrast will be adjusted automatically.

*mfxF64* **Hue**

The hue parameter is in the range of -180F to 180F, in increments of 0.1F. Setting this field to 0.0F will disable hue adjustment.

*mfxF64* **Saturation**

The saturation parameter is in the range of 0.0F to 10.0F, in increments of 0.01F. Setting this field to 1.0F will disable saturation adjustment.

### mfxExtVPPRotation

struct **mfxExtVPPRotation**

Configures the VPP Rotation filter algorithm.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_ROTATION.

*mfxU16* **Angle**

Rotation angle. See Angle enumerator for supported values.

### mfxExtVPPScaling

struct **mfxExtVPPScaling**

Configures the VPP Scaling filter algorithm. Not all combinations of ScalingMode and InterpolationMethod are supported in the library. The application must use the Query API function to determine if a combination is supported.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_SCALING.

*mfxU16* **ScalingMode**

Scaling mode. See ScalingMode for values.

*mfxU16* **InterpolationMethod**

Interpolation mode for scaling algorithm. See InterpolationMode for values.

## mfxChannel

struct **mfxChannel**

A hint structure that configures the data channel.

### Public Members

*mfxDataType* **DataType**

Data type, mfxDataType enumerator.

*mfxU32* **Size**

Size of Look up table, the number of elements per dimension.

*mfxU8* ***Data**

The pointer to 3DLUT data, 8 bit unsigned integer.

*mfxU16* ***Data16**

The pointer to 3DLUT data, 16 bit unsigned integer.

*mfxU32* **reserved**[4]

Reserved for future extension.

## mfx3DLutSystemBuffer

struct **mfx3DLutSystemBuffer**

A hint structure that configures 3DLUT system buffer.

#### Public Members

*mfxChannel* **Channel**[3]

3 Channels, can be RGB or YUV, *mfxChannel* structure.

*mfxU32* **reserved**[8]

Reserved for future extension.

### mfx3DLutVideoBuffer

struct **mfx3DLutVideoBuffer**

A hint structure that configures 3DLUT video buffer.

#### Public Members

*mfxDataType* **DataType**

Data type, mfxDataType enumerator.

*mfx3DLutMemoryLayout* **MemLayout**

Indicates 3DLUT memory layout. mfx3DLutMemoryLayout enumerator.

*mfxMemId* **MemId**

Memory ID for holding the lookup table data. One MemID is dedicated for one instance of VPP.

*mfxU32* **reserved**[8]

Reserved for future extension.

### mfxExtVPP3DLut

struct **mfxExtVPP3DLut**

A hint structure that configures 3DLUT filter.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_3DLUT..

*mfx3DLutChannelMapping* **ChannelMapping**

Indicates 3DLUT channel mapping. mfx3DLutChannelMapping enumerator.

*mfxResourceType* **BufferType**

Indicates 3DLUT buffer type. mfxResourceType enumerator, can be system memory, VA surface, DX11 texture/buffer etc.

*mfx3DLutSystemBuffer* **SystemBuffer**

The 3DLUT system buffer. *mfx3DLutSystemBuffer* structure describes the details of the buffer.

*mfx3DLutVideoBuffer* **VideoBuffer**

The 3DLUT video buffer. *mfx3DLutVideoBuffer* describes the details of 3DLUT video buffer.

*mfxU32* **reserved**[4]

Reserved for future extension.

### mfxExtVPPVideoSignalInfo

struct **mfxExtVPPVideoSignalInfo**

Used to control transfer matrix and nominal range of YUV frames. The application should provide this during initialization. Supported for multiple conversions, for example YUV to YUV, YUV to RGB, and RGB to YUV.

**Note:** This structure is used by VPP only and is not compatible with *mfxExtVideoSignalInfo*.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_VIDEO_SIGNAL_INFO.

*mfxU16* **TransferMatrix**

Transfer matrix.

*mfxU16* **NominalRange**

Nominal range.

### mfxInfoVPP

struct **mfxInfoVPP**

Specifies configurations for video processing. A zero value in any of the fields indicates that the corresponding field is not explicitly specified.

### Public Members

*mfxFrameInfo* **In**

Input format for video processing.

*mfxFrameInfo* **Out**

Output format for video processing.

## mfxVPPCompInputStream

struct **mfxVPPCompInputStream**

Used to specify input stream details for composition of several input surfaces in the one output.

### Public Members

*mfxU32* **DstX**

X coordinate of location of input stream in output surface.

*mfxU32* **DstY**

Y coordinate of location of input stream in output surface.

*mfxU32* **DstW**

Width of of location of input stream in output surface.

*mfxU32* **DstH**

Height of of location of input stream in output surface.

*mfxU16* **LumaKeyEnable**

Non-zero value enables luma keying for the input stream. Luma keying is used to mark some of the areas of the frame with specified luma values as transparent. It may, for example, be used for closed captioning.

*mfxU16* **LumaKeyMin**

Minimum value of luma key, inclusive. Pixels whose luma values fit in this range are rendered transparent.

*mfxU16* **LumaKeyMax**

Maximum value of luma key, inclusive. Pixels whose luma values fit in this range are rendered transparent.

*mfxU16* **GlobalAlphaEnable**

Non-zero value enables global alpha blending for this input stream.

*mfxU16* **GlobalAlpha**

Alpha value for this stream. Should be in the range of 0 to 255, where 0 is transparent and 255 is opaque.

*mfxU16* **PixelAlphaEnable**

Non-zero value enables per pixel alpha blending for this input stream. The stream should have RGB color format.

*mfxU16* **TileId**

Specify the tile this video stream is assigned to. Should be in the range of 0 to NumTiles. Valid only if NumTiles > 0.

### mfxVPPStat

struct **mfxVPPStat**

Returns statistics collected during video processing.

#### Public Members

*mfxU32* **NumFrame**

Total number of frames processed.

*mfxU32* **NumCachedFrame**

Number of internally cached frames.

### mfxExtVPPPercEncPrefilter

struct **mfxExtVPPPercEncPrefilter**

The structure is used to configure perceptual encoding prefilter in VPP.

#### Public Members

*mfxU16* **reserved**[252]

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_PERC_ENC_PREFILTER.

## 5.2.8 Protected Structures

Protected structures.

### API

- *mfxExtCencParam*

### mfxExtCencParam

struct **_mfxExtCencParam**

Used to pass the decryption status report index for the Common Encryption usage model. The application can attach this extended buffer to the *mfxBitstream* structure at runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CENC_PARAM.

*mfxU32* **StatusReportIndex**

Decryption status report index.

## 5.2.9 DECODE_VPP Structures

Structures used by *DECODE_VPP* only.

### API

- *mfxSurfaceArray*
- *mfxVideoChannelParam*
- *mfxExtInCrops*

### mfxSurfaceArray

struct **mfxSurfaceArray**

The structure is reference counted object to return array of surfaces allocated and processed by the library.

### Public Members

*mfxHDL* **Context**

The context of the memory interface. User should not touch (change, set, null) this pointer.

*mfxStructVersion* **Version**

The version of the structure.

*mfxStatus* (***AddRef**)(struct *mfxSurfaceArray* *surface_array)

Increments the internal reference counter of the surface. The surface is not destroyed until the surface is released using the *mfxSurfaceArray::Release* function. *mfxSurfaceArray::AddRef* should be used each time a new link to the surface is created (for example, copy structure) for proper surface management.

**Param surface**
**[in]** Valid *mfxSurfaceArray*.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceArray->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***Release**)(struct *mfxSurfaceArray* *surface_array)

Decrements the internal reference counter of the surface. *mfxSurfaceArray::Release* should be called after using the *mfxSurfaceArray::AddRef* function to add a surface or when allocation logic requires it.

**Param surface_array**
**[in]** Valid *mfxSurfaceArray*.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceArray->Context is invalid (for example NULL).

MFX_ERR_UNDEFINED_BEHAVIOR If Reference Counter of surface is zero before call.

MFX_ERR_UNKNOWN Any internal error.

*mfxStatus* (***GetRefCounter**)(struct *mfxSurfaceArray* *surface_array, *mfxU32* *counter)

Returns current reference counter of *mfxSurfaceArray* structure.

**Param surface**
**[in]** Valid surface_array.

**Param counter**
**[out]** Sets counter to the current reference counter value.

**Return**

MFX_ERR_NONE If no error.

MFX_ERR_NULL_PTR If surface or counter is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceArray->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

*mfxFrameSurface1* ****Surfaces**

The array of pointers to *mfxFrameSurface1*. *mfxFrameSurface1* surfaces are allocated by the same agent who allocates *mfxSurfaceArray*.

*mfxU32* **NumSurfaces**

The size of array of pointers to *mfxFrameSurface1*.

### mfxVideoChannelParam

struct **mfxVideoChannelParam**

The structure is used for VPP channels initialization in Decode_VPP component.

#### Public Members

*mfxFrameInfo* **VPP**

The configuration parameters of VPP filters per each channel.

*mfxU16* **Protected**

Specifies the content protection mechanism.

*mfxU16* **IOPattern**

Output memory access types for SDK functions.

*mfxExtBuffer* ****ExtParam**

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

*mfxU16* **NumExtParam**

The number of extra configuration structures attached to the structure.

### mfxExtInCrops

struct **mfxExtInCrops**

The structure contains crop parameters which applied by Decode_VPP component to input surfaces before video processing operation. It is used for letterboxing operations.

### Public Members

*mfxRect* **Crops**

Extension buffer header. BufferId must be equal to MFX_EXTBUFF_CROPS. Crops parameters for letterboxing operations.

## 5.2.10 Camera Structures

Structures used by Camera Raw Acceleration Processing.

### API

- *mfxExtCamWhiteBalance*
- *mfxExtCamTotalColorControl*
- *mfxExtCamCscYuvRgb*
- *mfxExtCamHotPixelRemoval*
- *mfxExtCamBlackLevelCorrection*
- *mfxCamVignetteCorrectionElement*
- *mfxCamVignetteCorrectionParam*
- *mfxExtCamVignetteCorrection*
- *mfxExtCamBayerDenoise*
- *mfxExtCamColorCorrection3x3*
- *mfxExtCamPadding*
- *mfxExtCamPipeControl*
- *mfxCamFwdGammaSegment*
- *mfxExtCamFwdGamma*
- *mfxExtCamLensGeomDistCorrection*
- *mfxCam3DLutEntry*
- *mfxExtCam3DLut*

### mfxExtCamWhiteBalance

struct **mfxExtCamWhiteBalance**

A hint structure that configures Camera White Balance filter.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_WHITE_BALANCE.

*mfxU32* **Mode**

Specifies one of White Balance operation modes defined in enumeration mfxCamWhiteBalanceMode.

*mfxF64* **R**

White Balance Red correction..

*mfxF64* **G0**

White Balance Green Top correction..

*mfxF64* **B**

White Balance Blue correction.

*mfxF64* **G1**

White Balance Green Bottom correction..

*mfxU32* **reserved**[8]

Reserved for future extension.

## mfxExtCamTotalColorControl

struct **mfxExtCamTotalColorControl**

A hint structure that configures Camera Total Color Control filter.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_TOTAL_COLOR_CONTROL.

*mfxU16* **R**

Red element.

*mfxU16* **G**

Green element.

*mfxU16* **B**

Blue element.

*mfxU16* **C**

Cyan element.

*mfxU16* **M**

Magenta element.

*mfxU16* **Y**

Yellow element.

*mfxU16* **reserved**[6]

Reserved for future extension.

### mfxExtCamCscYuvRgb

struct **mfxExtCamCscYuvRgb**

A hint structure that configures Camera YUV to RGB format conversion.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_CSC_YUV_RGB.

*mfxF32* **PreOffset**[3]

Specifies offset for conversion from full range RGB input to limited range YUV for input color coordinate.

*mfxF32* **Matrix**[3][3]

Specifies conversion matrix with CSC coefficients.

*mfxF32* **PostOffset**[3]

Specifies offset for conversion from full range RGB input to limited range YUV for output color coordinate.

*mfxU16* **reserved**[30]

Reserved for future extension.

### mfxExtCamHotPixelRemoval

struct **mfxExtCamHotPixelRemoval**

A hint structure that configures Camera Hot Pixel Removal filter.

### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_HOT_PIXEL_REMOVAL.

*mfxU16* **PixelThresholdDifference**

Threshold for Hot Pixel difference.

*mfxU16* **PixelCountThreshold**

Count pixel detection.

*mfxU16* **reserved**[32]

Reserved for future extension.

## mfxExtCamBlackLevelCorrection

struct **mfxExtCamBlackLevelCorrection**

### Public Members

*mfxExtBuffer* **Header**

A hint structure that configures Camera black level correction. Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_BLACK_LEVEL_CORRECTION.

*mfxU16* **R**

Black Level Red correction.

*mfxU16* **G0**

Black Level Green Top correction.

*mfxU16* **B**

Black Level Blue correction.

*mfxU16* **G1**

Black Level Green Bottom correction.

*mfxU32* **reserved**[4]

Reserved for future extension.

### mfxCamVignetteCorrectionElement

struct **mfxCamVignetteCorrectionElement**

A structure that defines Camera Vignette Correction Element.

#### Public Members

*mfxU8* **integer**

Integer part of correction element.

*mfxU8* **mantissa**

Fractional part of correction element.

*mfxU8* **reserved**[6]

Reserved for future extension.

### mfxCamVignetteCorrectionParam

struct **mfxCamVignetteCorrectionParam**

A structure that defines Camera Vignette Correction Parameters.

#### Public Members

*mfxCamVignetteCorrectionElement* **R**

Red correction element.

*mfxCamVignetteCorrectionElement* **G0**

Green top correction element.

*mfxCamVignetteCorrectionElement* **B**

Blue Correction element.

*mfxCamVignetteCorrectionElement* **G1**

Green bottom correction element.

*mfxU32* **reserved**[4]

Reserved for future extension.

### mfxExtCamVignetteCorrection

struct **mfxExtCamVignetteCorrection**

A hint structure that configures Camera Vignette Correction filter.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_VIGNETTE_CORRECTION.

*mfxU32* **Width**

Width of Correction Map 2D buffer in *mfxCamVignetteCorrectionParam* elements.

*mfxU32* **Height**

Height of Correction Map 2D buffer in *mfxCamVignetteCorrectionParam* elements.

*mfxU32* **Pitch**

Pitch of Correction Map 2D buffer in *mfxCamVignetteCorrectionParam* elements.

*mfxU32* **reserved**[7]

Reserved for future extension.

*mfxCamVignetteCorrectionParam* ***CorrectionMap**

2D buffer of *mfxCamVignetteCorrectionParam* elements.

*mfxU64* **reserved1**

Reserved for alignment on 32bit and 64bit.

union *mfxExtCamVignetteCorrection*::[anonymous] **[anonymous]**

### mfxExtCamBayerDenoise

struct **mfxExtCamBayerDenoise**

A hint structure that configures Camera Bayer denoise filter.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_BAYER_DENOISE.

*mfxU16* **Threshold**

Level of denoise, legal values: [0:63].

*mfxU16* **reserved**[27]

Reserved for future extension.

### mfxExtCamColorCorrection3x3

struct **mfxExtCamColorCorrection3x3**

A hint structure that configures Camera Color correction filter.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_COLOR_CORRECTION_3X3.

*mfxF32* **CCM**[3][3]

3x3 dimension matrix providing RGB Color Correction coefficients.

*mfxU32* **reserved**[32]

Reserved for future extension.

### mfxExtCamPadding

struct **mfxExtCamPadding**

A hint structure that configures Camera Padding.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_PADDING.

*mfxU16* **Top**

Specify number of padded columns respectively. Currently only 8 pixels supported for all dimensions..

*mfxU16* **Bottom**

Specify number of padded columns respectively. Currently only 8 pixels supported for all dimensions..

*mfxU16* **Left**

Specify number of padded rows respectively. Currently only 8 pixels supported for all dimensions..

*mfxU16* **Right**

Specify number of padded rows respectively. Currently only 8 pixels supported for all dimensions..

*mfxU32* **reserved**[4]

Reserved for future extension.

### mfxExtCamPipeControl

struct **mfxExtCamPipeControl**

A hint structure that configures camera pipe control.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_PIPECONTROL.

*mfxU16* **RawFormat**

Specifies one of the four Bayer patterns defined in mfxCamBayerFormat enumeration.

*mfxU16* **reserved1**

Reserved for future extension.

*mfxU32* **reserved**[5]

Reserved for future extension.

### mfxCamFwdGammaSegment

struct **mfxCamFwdGammaSegment**

A structure that specifies forward gamma segment.

#### Public Members

*mfxU16* **Pixel**

Pixel value.

*mfxU16* **Red**

Corrected Red value.

*mfxU16* **Green**

Corrected Green value.

*mfxU16* **Blue**

Corrected Blue value.

### mfxExtCamFwdGamma

struct **mfxExtCamFwdGamma**

A hint structure that configures Camera Forward Gamma Correction filter.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_FORWARD_GAMMA_CORRECTION.

*mfxU16* **reserved**[19]

Reserved for future extension.

*mfxU16* **NumSegments**

Number of Gamma segments.

*mfxCamFwdGammaSegment* ***Segment**

Pointer to Gamma segments array.

*mfxU64* **reserved1**

Reserved for future extension.

### mfxExtCamLensGeomDistCorrection

struct **mfxExtCamLensGeomDistCorrection**

A hint structure that configures Camera Lens Geometry Distortion and Chroma Aberration Correction filter.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_LENS_GEOM_DIST_CORRECTION.

*mfxF32* **a**[3]

Polynomial coefficients a for R/G/B

*mfxF32* **b**[3]

Polynomial coefficients b for R/G/B

*mfxF32* **c**[3]

Polynomial coefficients c for R/G/B

*mfxF32* **d**[3]

Polynomial coefficients d for R/G/B

*mfxU16* **reserved**[36]

Reserved for future extension.

### mfxCam3DLutEntry

struct **mfxCam3DLutEntry**

A structure that defines 3DLUT entry.

#### Public Members

*mfxU16* **R**

R channel

*mfxU16* **G**

G channel

*mfxU16* **B**

B channel

*mfxU16* **Reserved**

Reserved for future extension.

### mfxExtCam3DLut

struct **mfxExtCam3DLut**

A hint structure that configures Camera 3DLUT filter.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUF_CAM_3DLUT.

*mfxU16* **reserved**[10]

Reserved for future extension.

*mfxU32* **Size**

LUT size, defined in MFX_CAM_3DLUT17/33/65_SIZE enumeration.

*mfxCam3DLutEntry* ***Table**

Pointer to *mfxCam3DLutEntry*, size of each dimension depends on LUT size, e.g. LUT[17][17][17] for 17x17x17 look up table.

*mfxU64* **reserved1**

Reserved for future extension.

## 5.3 Enumerator Reference

### 5.3.1 Angle

The Angle enumerator itemizes valid rotation angles.

enumerator **MFX_ANGLE_0**

0 degrees.

enumerator **MFX_ANGLE_90**

90 degrees.

enumerator **MFX_ANGLE_180**

180 degrees.

enumerator **MFX_ANGLE_270**

270 degrees.

### 5.3.2 BitstreamDataFlag

The BitstreamDataFlag enumerator uses bit-ORed values to itemize additional information about the bitstream buffer.

enumerator **MFX_BITSTREAM_NO_FLAG**

The bitstream doesn't contain any flags.

enumerator **MFX_BITSTREAM_COMPLETE_FRAME**

The bitstream buffer contains a complete frame or complementary field pair of data for the bitstream. For decoding, this means that the decoder can proceed with this buffer without waiting for the start of the next frame, which effectively reduces decoding latency. If this flag is set, but the bitstream buffer contains incomplete frame or pair of field, then decoder will produce corrupted output.

enumerator **MFX_BITSTREAM_EOS**

The bitstream buffer contains the end of the stream. For decoding, this means that the application does not have any additional bitstream data to send to decoder.

### 5.3.3 BPSEIControl

The BPSEIControl enumerator is used to control insertion of buffering period SEI in the encoded bitstream.

enumerator **MFX_BPSEI_DEFAULT**

encoder decides when to insert BP SEI.

enumerator **MFX_BPSEI_IFRAME**

BP SEI should be inserted with every I-frame

### 5.3.4 BRCStatus

The BRCStatus enumerator itemizes instructions to the encoder by `mfxExtBrc::Update`.

enumerator **MFX_BRC_OK**

CodedFrameSize is acceptable, no further recoding/padding/skip required, proceed to next frame.

enumerator **MFX_BRC_BIG_FRAME**

Coded frame is too big, recoding required.

enumerator **MFX_BRC_SMALL_FRAME**

Coded frame is too small, recoding required.

enumerator **MFX_BRC_PANIC_BIG_FRAME**

Coded frame is too big, no further recoding possible - skip frame.

enumerator **MFX_BRC_PANIC_SMALL_FRAME**

Coded frame is too small, no further recoding possible - required padding to *mfxBRCFrameStatus::MinFrameSize*.

### 5.3.5 BRefControl

The BRefControl enumerator is used to control usage of B frames as reference in AVC encoder.

enumerator **MFX_B_REF_UNKNOWN**

Default value, it is up to the encoder to use B-frames as reference.

enumerator **MFX_B_REF_OFF**

Do not use B-frames as reference.

enumerator **MFX_B_REF_PYRAMID**

Arrange B-frames in so-called “B pyramid” reference structure.

### 5.3.6 ChromaFormateIdc

The ChromaFormatIdc enumerator itemizes color-sampling formats.

enumerator **MFX_CHROMAFORMAT_MONOCHROME**

Monochrome.

enumerator **MFX_CHROMAFORMAT_YUV420**

4:2:0 color.

enumerator **MFX_CHROMAFORMAT_YUV422**

4:2:2 color.

enumerator **MFX_CHROMAFORMAT_YUV444**

4:4:4 color.

enumerator **MFX_CHROMAFORMAT_YUV400**

Equal to monochrome.

enumerator **MFX_CHROMAFORMAT_YUV411**

4:1:1 color.

enumerator **MFX_CHROMAFORMAT_YUV422H**

4:2:2 color, horizontal sub-sampling. It is equal to 4:2:2 color.

enumerator **MFX_CHROMAFORMAT_YUV422V**

4:2:2 color, vertical sub-sampling.

enumerator **MFX_CHROMAFORMAT_RESERVED1**

Reserved.

enumerator **MFX_CHROMAFORMAT_JPEG_SAMPLING**

Color sampling specified via *mfxInfoMFX::SamplingFactorH* and SamplingFactorV.

### 5.3.7 ChromaSiting

The ChromaSiting enumerator defines chroma location. Use bit-OR'ed values to specify the desired location.

enumerator **MFX_CHROMA_SITING_UNKNOWN**

Unspecified.

enumerator **MFX_CHROMA_SITING_VERTICAL_TOP**

Chroma samples are co-sited vertically on the top with the luma samples.

enumerator **MFX_CHROMA_SITING_VERTICAL_CENTER**

Chroma samples are not co-sited vertically with the luma samples.

enumerator **MFX_CHROMA_SITING_VERTICAL_BOTTOM**

Chroma samples are co-sited vertically on the bottom with the luma samples.

enumerator **MFX_CHROMA_SITING_HORIZONTAL_LEFT**

Chroma samples are co-sited horizontally on the left with the luma samples.

enumerator **MFX_CHROMA_SITING_HORIZONTAL_CENTER**

Chroma samples are not co-sited horizontally with the luma samples.

### 5.3.8 CodecFormatFourCC

The CodecFormatFourCC enumerator itemizes codecs in the FourCC format.

enumerator **MFX_CODEC_AVC**

AVC, H.264, or MPEG-4, part 10 codec.

enumerator **MFX_CODEC_HEVC**

HEVC codec.

enumerator **MFX_CODEC_MPEG2**

MPEG-2 codec.

enumerator **MFX_CODEC_VC1**

VC-1 codec.

enumerator **MFX_CODEC_VP9**

VP9 codec.

enumerator **MFX_CODEC_AV1**

AV1 codec.

enumerator **MFX_CODEC_JPEG**

JPEG codec

### 5.3.9 CodecLevel

The CodecLevel enumerator itemizes codec levels for all codecs.

enumerator **MFX_LEVEL_UNKNOWN**

Unspecified level.

#### H.264 Level 1-1.3

enumerator **MFX_LEVEL_AVC_1**

enumerator **MFX_LEVEL_AVC_1b**

enumerator **MFX_LEVEL_AVC_11**

enumerator **MFX_LEVEL_AVC_12**

enumerator **MFX_LEVEL_AVC_13**

#### H.264 Level 2-2.2

enumerator **MFX_LEVEL_AVC_2**

enumerator **MFX_LEVEL_AVC_21**

enumerator **MFX_LEVEL_AVC_22**

#### H.264 Level 3-3.2

enumerator **MFX_LEVEL_AVC_3**

enumerator **MFX_LEVEL_AVC_31**

enumerator **MFX_LEVEL_AVC_32**

#### H.264 Level 4-4.2

enumerator **MFX_LEVEL_AVC_4**

enumerator **MFX_LEVEL_AVC_41**

enumerator **MFX_LEVEL_AVC_42**

### H.264 Level 5-5.2

enumerator **MFX_LEVEL_AVC_5**

enumerator **MFX_LEVEL_AVC_51**

enumerator **MFX_LEVEL_AVC_52**

### H.264 Level 6-6.2

enumerator **MFX_LEVEL_AVC_6**

enumerator **MFX_LEVEL_AVC_61**

enumerator **MFX_LEVEL_AVC_62**

### MPEG2 Levels

enumerator **MFX_LEVEL_MPEG2_LOW**

enumerator **MFX_LEVEL_MPEG2_MAIN**

enumerator **MFX_LEVEL_MPEG2_HIGH**

enumerator **MFX_LEVEL_MPEG2_HIGH1440**

### VC-1 Level Low (Simple and Main Profiles)

enumerator **MFX_LEVEL_VC1_LOW**

enumerator **MFX_LEVEL_VC1_MEDIAN**

enumerator **MFX_LEVEL_VC1_HIGH**

### VC-1 Advanced Profile Levels

enumerator **MFX_LEVEL_VC1_0**

enumerator **MFX_LEVEL_VC1_1**

enumerator **MFX_LEVEL_VC1_2**

enumerator **MFX_LEVEL_VC1_3**

enumerator **MFX_LEVEL_VC1_4**

### HEVC Levels

enumerator **MFX_LEVEL_HEVC_1**

enumerator **MFX_LEVEL_HEVC_2**

enumerator **MFX_LEVEL_HEVC_21**

enumerator **MFX_LEVEL_HEVC_3**

enumerator **MFX_LEVEL_HEVC_31**

enumerator **MFX_LEVEL_HEVC_4**

enumerator **MFX_LEVEL_HEVC_41**

enumerator **MFX_LEVEL_HEVC_5**

enumerator **MFX_LEVEL_HEVC_51**

enumerator **MFX_LEVEL_HEVC_52**

enumerator **MFX_LEVEL_HEVC_6**

enumerator **MFX_LEVEL_HEVC_61**

enumerator **MFX_LEVEL_HEVC_62**

### AV1 Levels

enumerator **MFX_LEVEL_AV1_2**

enumerator **MFX_LEVEL_AV1_21**

enumerator **MFX_LEVEL_AV1_22**

enumerator **MFX_LEVEL_AV1_23**

enumerator **MFX_LEVEL_AV1_3**

enumerator **MFX_LEVEL_AV1_31**

enumerator **MFX_LEVEL_AV1_32**

enumerator **MFX_LEVEL_AV1_33**

enumerator **MFX_LEVEL_AV1_4**

enumerator **MFX_LEVEL_AV1_41**

enumerator **MFX_LEVEL_AV1_42**

enumerator **MFX_LEVEL_AV1_43**

enumerator **MFX_LEVEL_AV1_5**

enumerator **MFX_LEVEL_AV1_51**

enumerator **MFX_LEVEL_AV1_52**

enumerator **MFX_LEVEL_AV1_53**

enumerator **MFX_LEVEL_AV1_6**

enumerator **MFX_LEVEL_AV1_61**

enumerator **MFX_LEVEL_AV1_62**

enumerator **MFX_LEVEL_AV1_63**

enumerator **MFX_LEVEL_AV1_7**

enumerator **MFX_LEVEL_AV1_71**

enumerator **MFX_LEVEL_AV1_72**

enumerator **MFX_LEVEL_AV1_73**

### 5.3.10 CodecProfile

The CodecProfile enumerator itemizes codec profiles for all codecs.

enumerator **MFX_PROFILE_UNKNOWN**

Unspecified profile.

#### H.264 Profiles

enumerator **MFX_PROFILE_AVC_BASELINE**

enumerator **MFX_PROFILE_AVC_MAIN**

enumerator **MFX_PROFILE_AVC_EXTENDED**

enumerator **MFX_PROFILE_AVC_HIGH**

enumerator **MFX_PROFILE_AVC_HIGH10**

enumerator **MFX_PROFILE_AVC_HIGH_422**

enumerator **MFX_PROFILE_AVC_CONSTRAINED_BASELINE**

enumerator **MFX_PROFILE_AVC_CONSTRAINED_HIGH**

#### AV1 Profiles

enumerator **MFX_PROFILE_AV1_MAIN**

enumerator **MFX_PROFILE_AV1_HIGH**

enumerator **MFX_PROFILE_AV1_PRO**

### VC-1 Profiles

enumerator **MFX_PROFILE_VC1_SIMPLE**

enumerator **MFX_PROFILE_VC1_MAIN**

enumerator **MFX_PROFILE_VC1_ADVANCED**

### VP8 Profiles

enumerator **MFX_PROFILE_VP8_0**

enumerator **MFX_PROFILE_VP8_1**

enumerator **MFX_PROFILE_VP8_2**

enumerator **MFX_PROFILE_VP8_3**

### VP9 Profiles

enumerator **MFX_PROFILE_VP9_0**

enumerator **MFX_PROFILE_VP9_1**

enumerator **MFX_PROFILE_VP9_2**

enumerator **MFX_PROFILE_VP9_3**

### H.264 Constraints

Combined with H.264 profile, these flags impose additional constraints. See the H.264 specification for the list of constraints.

enumerator **MFX_PROFILE_AVC_CONSTRAINT_SET0**

enumerator **MFX_PROFILE_AVC_CONSTRAINT_SET1**

enumerator **MFX_PROFILE_AVC_CONSTRAINT_SET2**

enumerator **MFX_PROFILE_AVC_CONSTRAINT_SET3** enumerator **MFX_PROFILE_AVC_CONSTRAINT_SET4**

enumerator **MFX_PROFILE_AVC_CONSTRAINT_SET5**

#### JPEG Profiles

enumerator **MFX_PROFILE_JPEG_BASELINE**

Baseline JPEG profile.

### 5.3.11 CodingOptionValue

The CodingOptionValue enumerator defines a three-state coding option setting.

enumerator **MFX_CODINGOPTION_UNKNOWN**

Unspecified.

enumerator **MFX_CODINGOPTION_ON**

Coding option set.

enumerator **MFX_CODINGOPTION_OFF**

Coding option not set.

enumerator **MFX_CODINGOPTION_ADAPTIVE**

Reserved.

### 5.3.12 ColorFourCC

The ColorFourCC enumerator itemizes color formats.

enumerator **MFX_FOURCC_NV12**

NV12 color planes. Native format for 4:2:0/8b Gen hardware implementation.

enumerator **MFX_FOURCC_NV21**

Same as NV12 but with weaved V and U values.

enumerator **MFX_FOURCC_YV12**

YV12 color planes.

enumerator **MFX_FOURCC_IYUV**

Same as YV12 except that the U and V plane order is reversed.

enumerator **MFX_FOURCC_I420**

Alias for the IYUV color format.

enumerator **MFX_FOURCC_I422**

Same as YV16 except that the U and V plane order is reversed

enumerator **MFX_FOURCC_NV16**

4:2:2 color format with similar to NV12 layout.

enumerator **MFX_FOURCC_YUY2**

YUY2 color planes.

enumerator **MFX_FOURCC_RGB565**

2 bytes per pixel, uint16 in little-endian format, where 0-4 bits are blue, bits 5-10 are green and bits 11-15 are red.

enumerator **MFX_FOURCC_RGBP**

RGB 24 bit planar layout (3 separate channels, 8-bits per sample each). This format should be mapped to D3DFMT_R8G8B8 or VA_FOURCC_RGBP.

enumerator **MFX_FOURCC_RGB4**

RGB4 (RGB32) color planes. BGRA is the order, 'B' is 8 MSBs, then 8 bits for 'G' channel, then 'R' and 'A' channels.

enumerator **MFX_FOURCC_BGRA**

Alias for the RGB4 color format.

enumerator **MFX_FOURCC_P8**

Internal color format. The application should use the following functions to create a surface that corresponds to the Direct3D* version in use.

For Direct3D* 9: IDirectXVideoDecoderService::CreateSurface()

For Direct3D* 11: ID3D11Device::CreateBuffer()

enumerator **MFX_FOURCC_P8_TEXTURE**

Internal color format. The application should use the following functions to create a surface that corresponds to the Direct3D* version in use.

For Direct3D 9: IDirectXVideoDecoderService::CreateSurface()

For Direct3D 11: ID3D11Device::CreateTexture2D()

enumerator **MFX_FOURCC_P010**

P010 color format. This is 10 bit per sample format with similar to NV12 layout. This format should be mapped to DXGI_FORMAT_P010.

enumerator **MFX_FOURCC_I010**

10-bit YUV 4:2:0, each component has its own plane.

enumerator **MFX_FOURCC_I210**

10-bit YUV 4:2:2, each component has its own plane.

enumerator **MFX_FOURCC_P016**

P016 color format. This is 16 bit per sample format with similar to NV12 layout. This format should be mapped to DXGI_FORMAT_P016.

enumerator **MFX_FOURCC_P210**

10 bit per sample 4:2:2 color format with similar to NV12 layout.

enumerator **MFX_FOURCC_BGR4**

RGBA color format. It is similar to MFX_FOURCC_RGB4 but with different order of channels. 'R' is 8 MSBs, then 8 bits for 'G' channel, then 'B' and 'A' channels.

enumerator **MFX_FOURCC_A2RGB10**

10 bits ARGB color format packed in 32 bits. 'A' channel is two MSBs, then 'R', then 'G' and then 'B' channels. This format should be mapped to DXGI_FORMAT_R10G10B10A2_UNORM or D3DFMT_A2R10G10B10.

enumerator **MFX_FOURCC_ARGB16**

10 bits ARGB color format packed in 64 bits. 'A' channel is 16 MSBs, then 'R', then 'G' and then 'B' channels. This format should be mapped to DXGI_FORMAT_R16G16B16A16_UINT or D3DFMT_A16B16G16R16 formats.

enumerator **MFX_FOURCC_ABGR16**

10 bits ABGR color format packed in 64 bits. 'A' channel is 16 MSBs, then 'B', then 'G' and then 'R' channels. This format should be mapped to DXGI_FORMAT_R16G16B16A16_UINT or D3DFMT_A16B16G16R16 formats.

enumerator **MFX_FOURCC_R16**

16 bits single channel color format. This format should be mapped to DXGI_FORMAT_R16_TYPELESS or D3DFMT_R16F.

enumerator **MFX_FOURCC_AYUV**

YUV 4:4:4, AYUV color format. This format should be mapped to DXGI_FORMAT_AYUV.

enumerator **MFX_FOURCC_AYUV_RGB4**

RGB4 stored in AYUV surface. This format should be mapped to DXGI_FORMAT_AYUV.

enumerator **MFX_FOURCC_UYVY**

UYVY color planes. Same as YUY2 except the byte order is reversed.

enumerator **MFX_FOURCC_Y210**

10 bit per sample 4:2:2 packed color format with similar to YUY2 layout. This format should be mapped to DXGI_FORMAT_Y210.

enumerator **MFX_FOURCC_Y410**

10 bit per sample 4:4:4 packed color format. This format should be mapped to DXGI_FORMAT_Y410.

enumerator **MFX_FOURCC_Y216**

16 bit per sample 4:2:2 packed color format with similar to YUY2 layout. This format should be mapped to DXGI_FORMAT_Y216.

enumerator **MFX_FOURCC_Y416**

16 bit per sample 4:4:4 packed color format. This format should be mapped to DXGI_FORMAT_Y416.

enumerator **MFX_FOURCC_BGRP**

BGR 24 bit planar layout (3 separate channels, 8-bits per sample each). This format should be mapped to VA_FOURCC_BGRP.

enumerator **MFX_FOURCC_XYUV**

8bit per sample 4:4:4 format packed in 32 bits, X=unused/undefined, 'X' channel is 8 MSBs, then 'Y', then 'U', and then 'V' channels. This format should be mapped to VA_FOURCC_XYUV.

enumerator **MFX_FOURCC_ABGR16F**

16 bits float point ABGR color format packed in 64 bits. 'A' channel is 16 MSBs, then 'B', then 'G' and then 'R' channels. This format should be mapped to DXGI_FORMAT_R16G16B16A16_FLOAT or D3DFMT_A16B16G16R16F formats..

## 5.3.13 ContentInfo

The ContentInfo enumerator itemizes content types for the encoding session.

enumerator **MFX_CONTENT_UNKNOWN**

enumerator **MFX_CONTENT_FULL_SCREEN_VIDEO**

enumerator **MFX_CONTENT_NON_VIDEO_SCREEN**

enumerator **MFX_CONTENT_NOISY_VIDEO**

## 5.3.14 Corruption

The Corruption enumerator itemizes the decoding corruption types. It is a bit-OR'ed value of the following.

enumerator **MFX_CORRUPTION_NO**

No corruption.

enumerator **MFX_CORRUPTION_MINOR**

Minor corruption in decoding certain macro-blocks.

enumerator **MFX_CORRUPTION_MAJOR**

Major corruption in decoding the frame - incomplete data, for example.

enumerator **MFX_CORRUPTION_ABSENT_TOP_FIELD**

Top field of frame is absent in bitstream. Only bottom field has been decoded.

enumerator **MFX_CORRUPTION_ABSENT_BOTTOM_FIELD**

Bottom field of frame is absent in bitstream. Only top filed has been decoded.

enumerator **MFX_CORRUPTION_REFERENCE_FRAME**

Decoding used a corrupted reference frame. A corrupted reference frame was used for decoding this frame. For example, if the frame uses a reference frame that was decoded with minor/major corruption flag, then this frame is also marked with a reference corruption flag.

enumerator **MFX_CORRUPTION_REFERENCE_LIST**

The reference list information of this frame does not match what is specified in the Reference Picture Marking Repetition SEI message. (ITU-T H.264 D.1.8 dec_ref_pic_marking_repetition)

enumerator **MFX_CORRUPTION_HW_RESET**

The hardware reset is reported from media driver.

**Note:** Flag MFX_CORRUPTION_ABSENT_TOP_FIELD/MFX_CORRUPTION_ABSENT_BOTTOM_FIELD is set by the AVC decoder when it detects that one of fields is not present in the bitstream. Which field is absent depends on value of bottom_field_flag (ITU-T* H.264 7.4.3).

### 5.3.15 DeinterlacingMode

The DeinterlacingMode enumerator itemizes VPP deinterlacing modes.

enumerator **MFX_DEINTERLACING_BOB**

BOB deinterlacing mode.

enumerator **MFX_DEINTERLACING_ADVANCED**

Advanced deinterlacing mode.

enumerator **MFX_DEINTERLACING_AUTO_DOUBLE**

Auto mode with deinterlacing double frame rate output.

enumerator **MFX_DEINTERLACING_AUTO_SINGLE**

Auto mode with deinterlacing single frame rate output.

enumerator **MFX_DEINTERLACING_FULL_FR_OUT**

Deinterlace only mode with full frame rate output.

enumerator **MFX_DEINTERLACING_HALF_FR_OUT**

Deinterlace only Mode with half frame rate output.

enumerator **MFX_DEINTERLACING_24FPS_OUT**

24 fps fixed output mode.

enumerator **MFX_DEINTERLACING_FIXED_TELECINE_PATTERN**

Fixed telecine pattern removal mode.

enumerator **MFX_DEINTERLACING_30FPS_OUT**

30 fps fixed output mode.

enumerator **MFX_DEINTERLACING_DETECT_INTERLACE**

Only interlace detection.

enumerator **MFX_DEINTERLACING_ADVANCED_NOREF**

Advanced deinterlacing mode without using of reference frames.

enumerator **MFX_DEINTERLACING_ADVANCED_SCD**

Advanced deinterlacing mode with scene change detection.

enumerator **MFX_DEINTERLACING_FIELD_WEAVING**

Field weaving.

### 5.3.16 ErrorTypes

The ErrorTypes enumerator uses bit-ORed values to itemize bitstream error types.

enumerator **MFX_ERROR_NO**

No error in bitstream.

enumerator **MFX_ERROR_PPS**

Invalid/corrupted PPS.

enumerator **MFX_ERROR_SPS**

Invalid/corrupted SPS.

enumerator **MFX_ERROR_SLICEHEADER**

Invalid/corrupted slice header.

enumerator **MFX_ERROR_SLICEDATA**

Invalid/corrupted slice data.

enumerator **MFX_ERROR_FRAME_GAP**

Missed frames.

enumerator **MFX_ERROR_JPEG_APP0_MARKER**

Invalid/corrupted APP0 marker.

enumerator **MFX_ERROR_JPEG_APP1_MARKER**
Invalid/corrupted APP1 marker.

enumerator **MFX_ERROR_JPEG_APP2_MARKER**
Invalid/corrupted APP2 marker.

enumerator **MFX_ERROR_JPEG_APP3_MARKER**
Invalid/corrupted APP3 marker.

enumerator **MFX_ERROR_JPEG_APP4_MARKER**
Invalid/corrupted APP4 marker.

enumerator **MFX_ERROR_JPEG_APP5_MARKER**
Invalid/corrupted APP5 marker.

enumerator **MFX_ERROR_JPEG_APP6_MARKER**
Invalid/corrupted APP6 marker.

enumerator **MFX_ERROR_JPEG_APP7_MARKER**
Invalid/corrupted APP7 marker.

enumerator **MFX_ERROR_JPEG_APP8_MARKER**
Invalid/corrupted APP8 marker.

enumerator **MFX_ERROR_JPEG_APP9_MARKER**
Invalid/corrupted APP9 marker.

enumerator **MFX_ERROR_JPEG_APP10_MARKER**
Invalid/corrupted APP10 marker.

enumerator **MFX_ERROR_JPEG_APP11_MARKER**
Invalid/corrupted APP11 marker.

enumerator **MFX_ERROR_JPEG_APP12_MARKER**
Invalid/corrupted APP12 marker.

enumerator **MFX_ERROR_JPEG_APP13_MARKER**
Invalid/corrupted APP13 marker.

enumerator **MFX_ERROR_JPEG_APP14_MARKER**
Invalid/corrupted APP14 marker.

enumerator **MFX_ERROR_JPEG_DQT_MARKER**
Invalid/corrupted DQT marker.

enumerator **MFX_ERROR_JPEG_SOF0_MARKER**

Invalid/corrupted SOF0 marker.

enumerator **MFX_ERROR_JPEG_DHT_MARKER**

Invalid/corrupted DHT marker.

enumerator **MFX_ERROR_JPEG_DRI_MARKER**

Invalid/corrupted DRI marker.

enumerator **MFX_ERROR_JPEG_SOS_MARKER**

Invalid/corrupted SOS marker.

enumerator **MFX_ERROR_JPEG_UNKNOWN_MARKER**

Unknown Marker.

### 5.3.17 ExtendedBufferID

The ExtendedBufferID enumerator itemizes and defines identifiers (BufferId) for extended buffers or video processing algorithm identifiers.

enumerator **MFX_EXTBUFF_THREADS_PARAM**

*mfxExtThreadsParam* buffer ID.

enumerator **MFX_EXTBUFF_CODING_OPTION**

This extended buffer defines additional encoding controls. See the *mfxExtCodingOption* structure for details. The application can attach this buffer to the structure for encoding initialization.

enumerator **MFX_EXTBUFF_CODING_OPTION_SPSPPS**

This extended buffer defines sequence header and picture header for encoders and decoders. See the *mfxExtCodingOptionSPSPPS* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, and for obtaining raw headers from the decoders and encoders.

enumerator **MFX_EXTBUFF_VPP_DONOTUSE**

This extended buffer defines a list of VPP algorithms that applications should not use. See the *mfxExtVPPDoNotUse* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator **MFX_EXTBUFF_VPP_AUXDATA**

This extended buffer defines auxiliary information at the VPP output. See the *mfxExtVppAuxData* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding control.

enumerator **MFX_EXTBUFF_VPP_DENOISE2**

The extended buffer defines control parameters for the VPP denoise filter algorithm. See the *mfxExtVPPDenoise2* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator **MFX_EXTBUFF_VPP_3DLUT**

See the *mfxExtVPP3DLut* structure for more details.

enumerator **MFX_EXTBUFF_VPP_SCENE_ANALYSIS**

Reserved for future use.

enumerator **MFX_EXTBUFF_VPP_PROCAMP**

The extended buffer defines control parameters for the VPP ProcAmp filter algorithm. See the *mfxExtVPPProcAmp* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization or to the *mfxFrameData* structure in the *mfxFrameSurface1* structure of output surface for per-frame processing configuration.

enumerator **MFX_EXTBUFF_VPP_DETAIL**

The extended buffer defines control parameters for the VPP detail filter algorithm. See the *mfxExtVPPDetail* structure for details. The application can attach this buffer to the structure for video processing initialization.

enumerator **MFX_EXTBUFF_VIDEO_SIGNAL_INFO**

This extended buffer defines video signal type. See the *mfxExtVideoSignalInfo* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, and for retrieving such information from the decoders. If video signal info changes per frame, the application can attach this buffer to the *mfxFrameData* structure for video processing.

enumerator **MFX_EXTBUFF_VIDEO_SIGNAL_INFO_IN**

This extended buffer defines video signal type. See the *mfxExtVideoSignalInfo* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for the input of video processing if the input video signal information changes in sequence base.

enumerator **MFX_EXTBUFF_VIDEO_SIGNAL_INFO_OUT**

This extended buffer defines video signal type. See the *mfxExtVideoSignalInfo* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for the output of video processing if the output video signal information changes in sequence base.

enumerator **MFX_EXTBUFF_VPP_DOUSE**

This extended buffer defines a list of VPP algorithms that applications should use. See the *mfxExtVPPDoUse* structure for details. The application can attach this buffer to the structure for video processing initialization.

enumerator **MFX_EXTBUFF_AVC_REFLIST_CTRL**

This extended buffer defines additional encoding controls for reference list. See the *mfxExtAVCRefListCtrl* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding & decoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator **MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION**

This extended buffer defines control parameters for the VPP frame rate conversion algorithm. See the *mfxExtVPPFrameRateConversion* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator **MFX_EXTBUFF_PICTURE_TIMING_SEI**

This extended buffer configures the H.264 picture timing SEI message. See the *mfxExtPictureTimingSEI* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator **MFX_EXTBUFF_AVC_TEMPORAL_LAYERS**

This extended buffer configures the structure of temporal layers inside the encoded H.264 bitstream. See the *mfxExtAvcTemporalLayers* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator **MFX_EXTBUFF_CODING_OPTION2**

This extended buffer defines additional encoding controls. See the *mfxExtCodingOption2* structure for details. The application can attach this buffer to the structure for encoding initialization.

enumerator **MFX_EXTBUFF_VPP_IMAGE_STABILIZATION**

This extended buffer defines control parameters for the VPP image stabilization filter algorithm. See the *mfxExtVPPImageStab* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator **MFX_EXTBUFF_ENCODER_CAPABILITY**

This extended buffer is used to retrieve encoder capability. See the *mfxExtEncoderCapability* structure for details. The application can attach this buffer to the *mfxVideoParam* structure before calling MFXVideoENCODE_Query function.

enumerator **MFX_EXTBUFF_ENCODER_RESET_OPTION**

This extended buffer is used to control encoder reset behavior and also to query possible encoder reset outcome. See the *mfxExtEncoderResetOption* structure for details. The application can attach this buffer to the *mfxVideoParam* structure before calling MFXVideoENCODE_Query or MFXVideoENCODE_Reset functions.

enumerator **MFX_EXTBUFF_ENCODED_FRAME_INFO**

This extended buffer is used by the encoder to report additional information about encoded picture. See the *mfxExtAVCEncodedFrameInfo* structure for details. The application can attach this buffer to the *mfxBitstream* structure before calling MFXVideoENCODE_EncodeFrameAsync function.

enumerator **MFX_EXTBUFF_VPP_COMPOSITE**

This extended buffer is used to control composition of several input surfaces in the one output. In this mode, the VPP skips any other filters. The VPP returns error if any mandatory filter is specified and filter skipped warning for optional filter. The only supported filters are deinterlacing and interlaced scaling.

enumerator **MFX_EXTBUFF_VPP_VIDEO_SIGNAL_INFO**

This extended buffer is used to control transfer matrix and nominal range of YUV frames. The application should provide it during initialization.

enumerator **MFX_EXTBUFF_ENCODER_ROI**

This extended buffer is used by the application to specify different Region Of Interests during encoding. The application should provide it at initialization or at runtime.

enumerator **MFX_EXTBUFF_VPP_DEINTERLACING**

This extended buffer is used by the application to specify different deinterlacing algorithms.

enumerator **MFX_EXTBUFF_AVC_REFLISTS**

This extended buffer specifies reference lists for the encoder.

enumerator **MFX_EXTBUFF_DEC_VIDEO_PROCESSING**

See the *mfxExtDecVideoProcessing* structure for details.

enumerator **MFX_EXTBUFF_VPP_FIELD_PROCESSING**

The extended buffer defines control parameters for the VPP field-processing algorithm. See the *mfxExtVPPFieldProcessing* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization or to the *mfxFrameData* structure during runtime.

enumerator **MFX_EXTBUFF_CODING_OPTION3**

This extended buffer defines additional encoding controls. See the *mfxExtCodingOption3* structure for details. The application can attach this buffer to the structure for encoding initialization.

enumerator **MFX_EXTBUFF_CHROMA_LOC_INFO**

This extended buffer defines chroma samples location information. See the *mfxExtChromaLocInfo* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator **MFX_EXTBUFF_MBQP**

This extended buffer defines per-macroblock QP. See the *mfxExtMBQP* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator **MFX_EXTBUFF_MB_FORCE_INTRA**

This extended buffer defines per-macroblock force intra flag. See the *mfxExtMBForceIntra* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator **MFX_EXTBUFF_HEVC_TILES**

This extended buffer defines additional encoding controls for HEVC tiles. See the *mfxExtHEVCTiles* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator **MFX_EXTBUFF_MB_DISABLE_SKIP_MAP**

This extended buffer defines macroblock map for current frame which forces specified macroblocks to be non skip. See the *mfxExtMBDisableSkipMap* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator **MFX_EXTBUFF_HEVC_PARAM**

See the *mfxExtHEVCParam* structure for details.

enumerator **MFX_EXTBUFF_DECODED_FRAME_INFO**

This extended buffer is used by decoders to report additional information about decoded frame. See the *mfxExtDecodedFrameInfo* structure for more details.

enumerator **MFX_EXTBUFF_TIME_CODE**

See the *mfxExtTimeCode* structure for more details.

enumerator **MFX_EXTBUFF_HEVC_REGION**

This extended buffer specifies the region to encode. The application can attach this buffer to the *mfxVideoParam* structure during HEVC encoder initialization.

enumerator **MFX_EXTBUFF_PRED_WEIGHT_TABLE**

See the *mfxExtPredWeightTable* structure for details.

enumerator **MFX_EXTBUFF_DIRTY_RECTANGLES**

See the *mfxExtDirtyRect* structure for details.

enumerator **MFX_EXTBUFF_MOVING_RECTANGLES**

See the *mfxExtMoveRect* structure for details.

enumerator **MFX_EXTBUFF_CODING_OPTION_VPS**

See the *mfxExtCodingOptionVPS* structure for details.

enumerator **MFX_EXTBUFF_VPP_ROTATION**

See the *mfxExtVPPRotation* structure for details.

enumerator **MFX_EXTBUFF_ENCODED_SLICES_INFO**

See the *mfxExtEncodedSlicesInfo* structure for details.

enumerator **MFX_EXTBUFF_VPP_SCALING**

See the *mfxExtVPPScaling* structure for details.

enumerator **MFX_EXTBUFF_HEVC_REFLIST_CTRL**

This extended buffer defines additional encoding controls for reference list. See the *mfxExtAVCRefListCtrl* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding & decoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator **MFX_EXTBUFF_HEVC_REFLISTS**

This extended buffer specifies reference lists for the encoder.

enumerator **MFX_EXTBUFF_HEVC_TEMPORAL_LAYERS**

This extended buffer configures the structure of temporal layers inside the encoded H.264 bitstream. See the *mfxExtAvcTemporalLayers* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator **MFX_EXTBUFF_VPP_MIRRORING**

See the *mfxExtVPPMirroring* structure for details.

enumerator **MFX_EXTBUFF_MV_OVER_PIC_BOUNDARIES**

See the *mfxExtMVOverPicBoundaries* structure for details.

enumerator **MFX_EXTBUFF_VPP_COLORFILL**

See the *mfxExtVPPColorFill* structure for details.

enumerator **MFX_EXTBUFF_DECODE_ERROR_REPORT**

This extended buffer is used by decoders to report error information before frames get decoded. See the *mfxExtDecodeErrorReport* structure for more details.

enumerator **MFX_EXTBUFF_VPP_COLOR_CONVERSION**

See the *mfxExtColorConversion* structure for details.

enumerator **MFX_EXTBUFF_CONTENT_LIGHT_LEVEL_INFO**

This extended buffer configures HDR SEI message. See the *mfxExtContentLightLevelInfo* structure for details.

enumerator **MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME**

This extended buffer configures HDR SEI message. See the *mfxExtMasteringDisplayColourVolume* structure for details. If color volume changes per frame, the application can attach this buffer to the *mfxFrameData* structure for video processing.

enumerator **MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME_IN**

This extended buffer configures HDR SEI message. See the *mfxExtMasteringDisplayColourVolume* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for the input of video processing if the mastering display color volume changes per sequence. In this case, this buffer should be together with MFX_EXTBUFF_CONTENT_LIGHT_LEVEL_INFO to indicate the light level and mastering color volume of the input of video processing. If color Volume changes per frame instead of per sequence, the application can attach MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME to *mfxFrameData* for frame based processing.

enumerator **MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME_OUT**

This extended buffer configures HDR SEI message. See the *mfxExtMasteringDisplayColourVolume* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for the output of video processing if the mastering display color volume changes per sequence. If color volume changes per frame instead of per sequence, the application can attach the buffer with MFX_EXTBUFF_MASTERING_ DISPLAY_COLOUR_VOLUME to *mfxFrameData* for frame based processing.

enumerator **MFX_EXTBUFF_ENCODED_UNITS_INFO**

See the *mfxExtEncodedUnitsInfo* structure for details.

enumerator **MFX_EXTBUFF_VPP_MCTF**

This video processing algorithm identifier is used to enable MCTF via *mfxExtVPPDoUse* and together with *mfxExtVppMctf*

enumerator **MFX_EXTBUFF_VP9_SEGMENTATION**

Extends *mfxVideoParam* structure with VP9 segmentation parameters. See the *mfxExtVP9Segmentation* structure for details.

enumerator **MFX_EXTBUFF_VP9_TEMPORAL_LAYERS**

Extends *mfxVideoParam* structure with parameters for VP9 temporal scalability. See the *mfxExtVP9TemporalLayers* structure for details.

enumerator **MFX_EXTBUFF_VP9_PARAM**

Extends *mfxVideoParam* structure with VP9-specific parameters. See the *mfxExtVP9Param* structure for details.

enumerator **MFX_EXTBUFF_AVC_ROUNDING_OFFSET**

See the *mfxExtAVCRoundingOffset* structure for details.

enumerator **MFX_EXTBUFF_PARTIAL_BITSTREAM_PARAM**

See the *mfxExtPartialBitstreamParam* structure for details.

enumerator **MFX_EXTBUFF_BRC**

enumerator **MFX_EXTBUFF_VP8_CODING_OPTION**

This extended buffer describes VP8 encoder configuration parameters. See the *mfxExtVP8CodingOption* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator **MFX_EXTBUFF_JPEG_QT**

This extended buffer defines quantization tables for JPEG encoder.

enumerator **MFX_EXTBUFF_JPEG_HUFFMAN**

This extended buffer defines Huffman tables for JPEG encoder.

enumerator **MFX_EXTBUFF_ENCODER_IPCM_AREA**

See the *mfxExtEncoderIPCMArea* structure for details.

enumerator **MFX_EXTBUFF_INSERT_HEADERS**

See the *mfxExtInsertHeaders* structure for details.

enumerator **MFX_EXTBUFF_MVC_SEQ_DESC**

This extended buffer describes the MVC stream information of view dependencies, view identifiers, and operation points. See the ITU*-T H.264 specification chapter H.7.3.2.1.4 for details.

enumerator **MFX_EXTBUFF_MVC_TARGET_VIEWS**

This extended buffer defines target views at the decoder output.

enumerator **MFX_EXTBUFF_CENC_PARAM**

This structure is used to pass decryption status report index for Common Encryption usage model. See the mfxExtCencParam structure for more details.

enumerator **MFX_EXTBUFF_DEVICE_AFFINITY_MASK**

See the *mfxExtDeviceAffinityMask* structure for details.

enumerator **MFX_EXTBUFF_CROPS**

See the *mfxExtInCrops* structure for details.

enumerator **MFX_EXTBUFF_AV1_FILM_GRAIN_PARAM**

See the *mfxExtAV1FilmGrainParam* structure for more details.

enumerator **MFX_EXTBUFF_AV1_SEGMENTATION**

See the *mfxExtAV1Segmentation* structure for more details.

enumerator **MFX_EXTBUFF_ALLOCATION_HINTS**

See the *mfxExtAllocationHints* structure for more details.

enumerator **MFX_EXTBUFF_UNIVERSAL_TEMPORAL_LAYERS**

See the *mfxExtTemporalLayers* structure for more details.

enumerator **MFX_EXTBUFF_UNIVERSAL_REFLIST_CTRL**

This extended buffer defines additional encoding controls for reference list. See the mfxExtRefListCtrl structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding & decoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator **MFX_EXTBUFF_ENCODESTATS**

See the mfxExtEncodeStats structure for details.

enumerator **MFX_EXTBUFF_SYNCSUBMISSION**

See the *mfxExtSyncSubmission* structure for more details.

enumerator **MFX_EXTBUFF_TUNE_ENCODE_QUALITY**

See the *mfxExtTuneEncodeQuality* structure for details.

enumerator **MFX_EXTBUFF_VPP_PERC_ENC_PREFILTER**

See the *mfxExtVPPPercEncPrefilter* structure for details.

### 5.3.18 ExtMemBufferType

enumerator **MFX_MEMTYPE_PERSISTENT_MEMORY**

Memory page for persistent use.

### 5.3.19 ExtMemFrameType

The ExtMemFrameType enumerator specifies the memory type of frame. It is a bit-ORed value of one of the following. For information on working with video memory surfaces, see the *Working with Hardware Acceleration section*.

enumerator **MFX_MEMTYPE_DXVA2_DECODER_TARGET**

Frames are in video memory and belong to video decoder render targets.

enumerator **MFX_MEMTYPE_DXVA2_PROCESSOR_TARGET**

Frames are in video memory and belong to video processor render targets.

enumerator **MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET**

Frames are in video memory and belong to video decoder render targets.

enumerator **MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET**

Frames are in video memory and belong to video processor render targets.

enumerator **MFX_MEMTYPE_SYSTEM_MEMORY**

The frames are in system memory.

enumerator **MFX_MEMTYPE_RESERVED1**

enumerator **MFX_MEMTYPE_FROM_ENCODE**

Allocation request comes from an ENCODE function

enumerator **MFX_MEMTYPE_FROM_DECODE**

Allocation request comes from a DECODE function

enumerator **MFX_MEMTYPE_FROM_VPPIN**

Allocation request comes from a VPP function for input frame allocation

enumerator **MFX_MEMTYPE_FROM_VPPOUT**

Allocation request comes from a VPP function for output frame allocation

enumerator **MFX_MEMTYPE_FROM_ENC**

Allocation request comes from an ENC function

enumerator **MFX_MEMTYPE_INTERNAL_FRAME**

Allocation request for internal frames

enumerator **MFX_MEMTYPE_EXTERNAL_FRAME**

Allocation request for I/O frames

enumerator **MFX_MEMTYPE_EXPORT_FRAME**

Application requests frame handle export to some associated object. For Linux frame handle can be considered to be exported to DRM Prime FD, DRM FLink or DRM FrameBuffer Handle. Specifics of export types and export procedure depends on external frame allocator implementation

enumerator **MFX_MEMTYPE_SHARED_RESOURCE**

For DX11 allocation use shared resource bind flag.

enumerator **MFX_MEMTYPE_VIDEO_MEMORY_ENCODER_TARGET**

Frames are in video memory and belong to video encoder render targets.

### 5.3.20 Frame Data Flags

enumerator **MFX_TIMESTAMP_UNKNOWN**

Indicates that time stamp is unknown for this frame/bitstream portion.

enumerator **MFX_FRAMEORDER_UNKNOWN**

Unused entry or API functions that generate the frame output do not use this frame.

enumerator **MFX_FRAMEDATA_TIMESTAMP_UNKNOWN**

Indicates the time stamp of this frame is unknown and will be calculated by SDK.

enumerator **MFX_FRAMEDATA_ORIGINAL_TIMESTAMP**

Indicates the time stamp of this frame is not calculated and is a pass-through of the original time stamp.

### 5.3.21 FrameType

The FrameType enumerator itemizes frame types. Use bit-ORed values to specify all that apply.

enumerator **MFX_FRAMETYPE_UNKNOWN**

Frame type is unspecified.

enumerator **MFX_FRAMETYPE_I**

This frame or the first field is encoded as an I-frame/field.

enumerator **MFX_FRAMETYPE_P**

This frame or the first field is encoded as an P-frame/field.

enumerator **MFX_FRAMETYPE_B**

This frame or the first field is encoded as an B-frame/field.

enumerator **MFX_FRAMETYPE_S**

This frame or the first field is either an SI- or SP-frame/field.

enumerator **MFX_FRAMETYPE_REF**

This frame or the first field is encoded as a reference.

enumerator **MFX_FRAMETYPE_IDR**

This frame or the first field is encoded as an IDR.

enumerator **MFX_FRAMETYPE_xI**

The second field is encoded as an I-field.

enumerator **MFX_FRAMETYPE_xP**

The second field is encoded as an P-field.

enumerator **MFX_FRAMETYPE_xB**

The second field is encoded as an S-field.

enumerator **MFX_FRAMETYPE_xS**

The second field is an SI- or SP-field.

enumerator **MFX_FRAMETYPE_xREF**

The second field is encoded as a reference.

enumerator **MFX_FRAMETYPE_xIDR**

The second field is encoded as an IDR.

### 5.3.22 FrcAlgm

The FrcAlgm enumerator itemizes frame rate conversion algorithms. See description of mfxExtVPPFrameRateConversion structure for more details.

enumerator **MFX_FRCALGM_PRESERVE_TIMESTAMP**

Frame dropping/repetition based frame rate conversion algorithm with preserved original time stamps. Any inserted frames will carry MFX_TIMESTAMP_UNKNOWN.

enumerator **MFX_FRCALGM_DISTRIBUTED_TIMESTAMP**

Frame dropping/repetition based frame rate conversion algorithm with distributed time stamps. The algorithm distributes output time stamps evenly according to the output frame rate.

enumerator **MFX_FRCALGM_FRAME_INTERPOLATION**

Frame rate conversion algorithm based on frame interpolation. This flag may be combined with MFX_FRCALGM_PRESERVE_TIMESTAMP or MFX_FRCALGM_DISTRIBUTED_TIMESTAMP flags.

### 5.3.23 GeneralConstraintFlags

The GeneralConstraintFlags enumerator uses bit-ORed values to itemize HEVC bitstream indications for specific profiles. Each value indicates for format range extensions profiles. To specify HEVC Main 10 Still Picture profile applications have to set mfxInfoMFX::CodecProfile == MFX_PROFILE_HEVC_MAIN10 and mfxExtHEVCParam::GeneralConstraintFlags == MFX_HEVC_CONSTR_REXT_ONE_PICTURE_ONLY.

enumerator **MFX_HEVC_CONSTR_REXT_MAX_12BIT**

enumerator **MFX_HEVC_CONSTR_REXT_MAX_10BIT**

enumerator **MFX_HEVC_CONSTR_REXT_MAX_8BIT**

enumerator **MFX_HEVC_CONSTR_REXT_MAX_422CHROMA**

enumerator **MFX_HEVC_CONSTR_REXT_MAX_420CHROMA**

enumerator **MFX_HEVC_CONSTR_REXT_MAX_MONOCHROME**

enumerator **MFX_HEVC_CONSTR_REXT_INTRA**

enumerator **MFX_HEVC_CONSTR_REXT_ONE_PICTURE_ONLY**

enumerator **MFX_HEVC_CONSTR_REXT_LOWER_BIT_RATE**

### 5.3.24 GopOptFlag

The GopOptFlag enumerator itemizes special properties in the GOP (Group of Pictures) sequence.

enumerator **MFX_GOP_CLOSED**

The encoder generates closed GOP if this flag is set. Frames in this GOP do not use frames in previous GOP as reference.

The encoder generates open GOP if this flag is not set. In this GOP frames prior to the first frame of GOP in display order may use frames from previous GOP as reference. Frames subsequent to the first frame of GOP in display order do not use frames from previous GOP as reference.

The AVC encoder ignores this flag if IdrInterval in *mfxInfoMFX* structure is set to 0, i.e. if every GOP starts from IDR frame. In this case, GOP is encoded as closed.

This flag does not affect long-term reference frames.

enumerator **MFX_GOP_STRICT**

The encoder must strictly follow the given GOP structure as defined by parameter GopPicSize, GopRefDist etc in the *mfxVideoParam* structure. Otherwise, the encoder can adapt the GOP structure for better efficiency, whose range is constrained by parameter GopPicSize and GopRefDist etc. See also description of AdaptiveI and AdaptiveB fields in the *mfxExtCodingOption2* structure.

### 5.3.25 GPUCopy

enumerator **MFX_GPUCOPY_DEFAULT**

Use default mode for the legacy Intel(r) Media SDK implementation.

enumerator **MFX_GPUCOPY_ON**

The hint to enable GPU accelerated copying when it is supported by the library. If the library doesn't support GPU accelerated copy the operation will be made by CPU.

enumerator **MFX_GPUCOPY_OFF**

Disable GPU accelerated copying.

### 5.3.26 HEVC Profiles

enumerator **MFX_PROFILE_HEVC_MAIN**

enumerator **MFX_PROFILE_HEVC_MAIN10**

enumerator **MFX_PROFILE_HEVC_MAINSP**

enumerator **MFX_PROFILE_HEVC_REXT**

enumerator **MFX_PROFILE_HEVC_SCC**

### 5.3.27 HEVC Tiers

enumerator **MFX_TIER_HEVC_MAIN**

enumerator **MFX_TIER_HEVC_HIGH**

### 5.3.28 HEVCRegionEncoding

The HEVCRegionEncoding enumerator itemizes HEVC region's encoding.

enumerator **MFX_HEVC_REGION_ENCODING_ON**

enumerator **MFX_HEVC_REGION_ENCODING_OFF**

### 5.3.29 HEVCRegionType

The HEVCRegionType enumerator itemizes type of HEVC region.

enumerator **MFX_HEVC_REGION_SLICE**

Slice type.

### 5.3.30 ImageStabMode

The ImageStabMode enumerator itemizes image stabilization modes. See description of mfxExtVPPImageStab structure for more details.

enumerator **MFX_IMAGESTAB_MODE_UPSCALE**

Upscale mode.

enumerator **MFX_IMAGESTAB_MODE_BOXING**

Boxing mode.

### 5.3.31 InsertHDRPayload

The InsertHDRPayload enumerator itemizes HDR payloads insertion rules.

enumerator **MFX_PAYLOAD_OFF**

Do not insert payload when encoding; Clip does not have valid HDE SEI when decoding.

enumerator **MFX_PAYLOAD_IDR**

Insert payload on IDR frames when encoding; Clip has valid HDE SEI when decoding.

### 5.3.32 InterpolationMode

The InterpolationMode enumerator specifies type of interpolation method used by VPP scaling filter.

enumerator **MFX_INTERPOLATION_DEFAULT**

Default interpolation mode for scaling. Library selects the most appropriate scaling method.

enumerator **MFX_INTERPOLATION_NEAREST_NEIGHBOR**

Nearest neighbor interpolation method.

enumerator **MFX_INTERPOLATION_BILINEAR**

Bilinear interpolation method.

enumerator **MFX_INTERPOLATION_ADVANCED**

Advanced interpolation method is defined by each implementation and usually gives best quality.

### 5.3.33 DataType

enum **mfxDataType**

The mfxDataType enumerates data type for mfxDataType.

*Values:*

enumerator **MFX_DATA_TYPE_UNSET**

Undefined type.

enumerator **MFX_DATA_TYPE_U8**

8-bit unsigned integer.

enumerator **MFX_DATA_TYPE_I8**

8-bit signed integer.

enumerator **MFX_DATA_TYPE_U16**

16-bit unsigned integer.

enumerator **MFX_DATA_TYPE_I16**

16-bit signed integer.

enumerator **MFX_DATA_TYPE_U32**

32-bit unsigned integer.

enumerator **MFX_DATA_TYPE_I32**

32-bit signed integer.

enumerator **MFX_DATA_TYPE_U64**

64-bit unsigned integer.

enumerator **MFX_DATA_TYPE_I64**

64-bit signed integer.

enumerator **MFX_DATA_TYPE_F32**

32-bit single precision floating point.

enumerator **MFX_DATA_TYPE_F64**

64-bit double precision floating point.

enumerator **MFX_DATA_TYPE_PTR**

Generic type pointer.

enumerator **MFX_DATA_TYPE_FP16**

16-bit half precision floating point.

### 5.3.34 3DLutChannelMapping

enum **mfx3DLutChannelMapping**

The mfx3DLutChannelMapping enumerator specifies the channel mapping of 3DLUT.

*Values:*

enumerator **MFX_3DLUT_CHANNEL_MAPPING_DEFAULT**

Default 3DLUT channel mapping. The library selects the most appropriate 3DLUT channel mapping.

enumerator **MFX_3DLUT_CHANNEL_MAPPING_RGB_RGB**

3DLUT RGB channels map to RGB channels.

enumerator **MFX_3DLUT_CHANNEL_MAPPING_YUV_RGB**

3DLUT YUV channels map to RGB channels.

enumerator **MFX_3DLUT_CHANNEL_MAPPING_VUY_RGB**

3DLUT VUY channels map to RGB channels.

### 5.3.35 3DLutMemoryLayout

enum **mfx3DLutMemoryLayout**

The mfx3DLutMemoryLayout enumerator specifies the memory layout of 3DLUT.

*Values:*

enumerator **MFX_3DLUT_MEMORY_LAYOUT_DEFAULT**

Default 3DLUT memory layout. The library selects the most appropriate 3DLUT memory layout.

enumerator **MFX_3DLUT_MEMORY_LAYOUT_VENDOR**

The enumeration to separate default above and vendor specific.

enumerator **MFX_3DLUT_MEMORY_LAYOUT_INTEL_17LUT**

Intel specific memory layout. The enumerator indicates the attributes and memory layout of 3DLUT. 3DLUT size is 17(the number of elements per dimension), 4 channels(3 valid channels, 1 channel is reserved), every channel must be 16-bit unsigned integer. 3DLUT contains 17x17x32 entries with holes that are not filled. Take RGB as example, the nodes RxGx17 to RxGx31 are not filled, are "don't care" bits, and not accessed for the 17x17x17 nodes.

enumerator **MFX_3DLUT_MEMORY_LAYOUT_INTEL_33LUT**

Intel specific memory layout. The enumerator indicates the attributes and memory layout of 3DLUT. 3DLUT size is 33(the number of elements per dimension), 4 channels(3 valid channels, 1 channel is reserved), every channel must be 16-bit unsigned integer. 3DLUT contains 33x33x64 entries with holes that are not filled. Take RGB as example, the nodes RxGx33 to RxGx63 are not filled, are "don't care" bits, and not accessed for the 33x33x33 nodes.

enumerator **MFX_3DLUT_MEMORY_LAYOUT_INTEL_65LUT**

Intel specific memory layout. The enumerator indicates the attributes and memory layout of 3DLUT. 3DLUT size is 65(the number of elements per dimension), 4 channels(3 valid channels, 1 channel is reserved), every channel must be 16-bit unsigned integer. 3DLUT contains 65x65x128 entries with holes that are not filled. Take RGB as example, the nodes RxGx65 to RxGx127 are not filled, are "don't care" bits, and not accessed for the 65x65x65 nodes.

### 5.3.36 IntraPredBlockSize/InterPredBlockSize

IntraPredBlockSize/InterPredBlockSize specifies minimum block size of inter-prediction.

enumerator **MFX_BLOCKSIZE_UNKNOWN**

Unspecified.

enumerator **MFX_BLOCKSIZE_MIN_16X16**

16x16 minimum block size.

enumerator **MFX_BLOCKSIZE_MIN_8X8**

8x8 minimum block size. May be 16x16 or 8x8.

enumerator **MFX_BLOCKSIZE_MIN_4X4**

4x4 minimum block size. May be 16x16, 8x8, or 4x4.

### 5.3.37 IntraRefreshTypes

The IntraRefreshTypes enumerator itemizes types of intra refresh.

enumerator **MFX_REFRESH_NO**

Encode without refresh.

enumerator **MFX_REFRESH_VERTICAL**

Vertical refresh, by column of MBs.

enumerator **MFX_REFRESH_HORIZONTAL**

Horizontal refresh, by rows of MBs.

enumerator **MFX_REFRESH_SLICE**

Horizontal refresh by slices without overlapping.

### 5.3.38 IOPattern

The IOPattern enumerator itemizes memory access patterns for API functions. Use bit-ORed values to specify input and output access patterns.

enumerator **MFX_IOPATTERN_IN_VIDEO_MEMORY**

Input to functions is a video memory surface.

enumerator **MFX_IOPATTERN_IN_SYSTEM_MEMORY**

Input to functions is a linear buffer directly in system memory or in system memory through an external allocator.

enumerator **MFX_IOPATTERN_OUT_VIDEO_MEMORY**

Output to functions is a video memory surface.

enumerator **MFX_IOPATTERN_OUT_SYSTEM_MEMORY**

Output to functions is a linear buffer directly in system memory or in system memory through an external allocator.

### 5.3.39 JPEGColorFormat

The JPEGColorFormat enumerator itemizes the JPEG color format options.

enumerator **MFX_JPEG_COLORFORMAT_UNKNOWN**

enumerator **MFX_JPEG_COLORFORMAT_YCbCr**

Unknown color format. The decoder tries to determine color format from available in bitstream information. If such information is not present, then MFX_JPEG_COLORFORMAT_YCbCr color format is assumed.

enumerator **MFX_JPEG_COLORFORMAT_RGB**

Bitstream contains Y, Cb and Cr components.

### 5.3.40 JPEGScanType

The JPEGScanType enumerator itemizes the JPEG scan types.

enumerator **MFX_SCANTYPE_UNKNOWN**

Unknown scan type.

enumerator **MFX_SCANTYPE_INTERLEAVED**

Interleaved scan.

enumerator **MFX_SCANTYPE_NONINTERLEAVED**

Non-interleaved scan.

### 5.3.41 LongTermIdx

The LongTermIdx specifies long term index of picture control

enumerator **MFX_LONGTERM_IDX_NO_IDX**

Long term index of picture is undefined.

### 5.3.42 LookAheadDownSampling

The LookAheadDownSampling enumerator is used to control down sampling in look ahead bitrate control mode in AVC encoder.

enumerator **MFX_LOOKAHEAD_DS_UNKNOWN**

Default value, it is up to the encoder what down sampling value to use.

enumerator **MFX_LOOKAHEAD_DS_OFF**

Do not use down sampling, perform estimation on original size frames. This is the slowest setting that produces the best quality.

enumerator **MFX_LOOKAHEAD_DS_2x**

Down sample frames two times before estimation.

enumerator **MFX_LOOKAHEAD_DS_4x**

Down sample frames four times before estimation. This option may significantly degrade quality.

### 5.3.43 MBQPMode

The MBQPMode enumerator itemizes QP update modes.

enumerator **MFX_MBQP_MODE_QP_VALUE**

QP array contains QP values.

enumerator **MFX_MBQP_MODE_QP_DELTA**

QP array contains deltas for QP.

enumerator **MFX_MBQP_MODE_QP_ADAPTIVE**

QP array contains deltas for QP or absolute QP values.

### 5.3.44 mfxComponentType

enum **mfxComponentType**

Describes type of workload passed to MFXQueryAdapters.

*Values:*

enumerator **MFX_COMPONENT_ENCODE**

Encode workload.

enumerator **MFX_COMPONENT_DECODE**

Decode workload.

enumerator **MFX_COMPONENT_VPP**

VPP workload.

### 5.3.45 mfxHandleType

enum **mfxHandleType**

The mfxHandleType enumerator itemizes system handle types that implementations might use.

*Values:*

enumerator **MFX_HANDLE_DIRECT3D_DEVICE_MANAGER9**

Pointer to the IDirect3DDeviceManager9 interface. See Working with Microsoft* DirectX* Applications for more details on how to use this handle.

enumerator **MFX_HANDLE_D3D9_DEVICE_MANAGER**

Pointer to the IDirect3DDeviceManager9 interface. See Working with Microsoft* DirectX* Applications for more details on how to use this handle.

enumerator **MFX_HANDLE_RESERVED1**

enumerator **MFX_HANDLE_D3D11_DEVICE**

Pointer to the ID3D11Device interface. See Working with Microsoft* DirectX* Applications for more details on how to use this handle.

enumerator **MFX_HANDLE_VA_DISPLAY**

VADisplay interface. See Working with VA-API Applications for more details on how to use this handle.

enumerator **MFX_HANDLE_RESERVED3**

enumerator **MFX_HANDLE_VA_CONFIG_ID**

Pointer to VAConfigID interface. It represents external VA config for Common Encryption usage model.

enumerator **MFX_HANDLE_VA_CONTEXT_ID**

Pointer to VAContextID interface. It represents external VA context for Common Encryption usage model.

enumerator **MFX_HANDLE_CM_DEVICE**

Pointer to CmDevice interface ( Intel(r) C for Metal Runtime ).

enumerator **MFX_HANDLE_HDDLUNITE_WORKLOADCONTEXT**

Pointer to HddlUnite::WorkloadContext interface.

enumerator **MFX_HANDLE_PXP_CONTEXT**

Pointer to PXP context for protected content support.

enumerator **MFX_HANDLE_CONFIG_INTERFACE**

Pointer to interface of type *mfxConfigInterface*.

enumerator **MFX_HANDLE_MEMORY_INTERFACE**

Pointer to interface of type *mfxMemoryInterface*.

### 5.3.46 mfxIMPL

typedef *mfxI32* **mfxIMPL**

This enumerator itemizes implementation types. The implementation type is a bit OR'ed value of the base type and any decorative flags.

**Note:** This enumerator is for legacy dispatcher compatibility only. The new dispatcher does not use it.

enumerator **MFX_IMPL_AUTO**

Auto Selection/In or Not Supported/Out.

enumerator **MFX_IMPL_SOFTWARE**

Pure software implementation.

enumerator **MFX_IMPL_HARDWARE**

Hardware accelerated implementation (default device).

enumerator **MFX_IMPL_AUTO_ANY**

Auto selection of any hardware/software implementation.

enumerator **MFX_IMPL_HARDWARE_ANY**

Auto selection of any hardware implementation.

enumerator **MFX_IMPL_HARDWARE2**

Hardware accelerated implementation (2nd device).

enumerator **MFX_IMPL_HARDWARE3**

Hardware accelerated implementation (3rd device).

enumerator **MFX_IMPL_HARDWARE4**

Hardware accelerated implementation (4th device).

enumerator **MFX_IMPL_RUNTIME**

This value cannot be used for session initialization. It may be returned by the MFXQueryIMPL function to show that the session has been initialized in run-time mode.

enumerator **MFX_IMPL_VIA_ANY**

Hardware acceleration can go through any supported OS infrastructure. This is the default value. The default value is used by the legacy Intel(r) Media SDK if none of the MFX_IMPL_VIA_xxx flags are specified by the application.

enumerator **MFX_IMPL_VIA_D3D9**

Hardware acceleration goes through the Microsoft* Direct3D* 9 infrastructure.

enumerator **MFX_IMPL_VIA_D3D11**

Hardware acceleration goes through the Microsoft* Direct3D* 11 infrastructure.

enumerator **MFX_IMPL_VIA_VAAPI**

Hardware acceleration goes through the Linux* VA-API infrastructure.

enumerator **MFX_IMPL_VIA_HDDLUNITE**

Hardware acceleration goes through the HDDL* Unite*.

enumerator **MFX_IMPL_UNSUPPORTED**

One of the MFXQueryIMPL returns.

**MFX_IMPL_BASETYPE**(x)

The application can use the macro MFX_IMPL_BASETYPE(x) to obtain the base implementation type.

### 5.3.47 mfxImplCapsDeliveryFormat

enum **mfxImplCapsDeliveryFormat**

*Values:*

enumerator **MFX_IMPLCAPS_IMPLDESCSTRUCTURE**

Deliver capabilities as *mfxImplDescription* structure.

enumerator **MFX_IMPLCAPS_IMPLEMENTEDFUNCTIONS**

Deliver capabilities as *mfxImplementedFunctions* structure.

enumerator **MFX_IMPLCAPS_IMPLPATH**

Deliver pointer to the null-terminated string with the path to the implementation. String is delivered in a form of buffer of mfxChar type.

enumerator **MFX_IMPLCAPS_DEVICE_ID_EXTENDED**

Deliver extended device ID information as *mfxExtendedDeviceId* structure.

enumerator **MFX_IMPLCAPS_SURFACE_TYPES**

Deliver capabilities as *mfxSurfaceTypesSupported* structure.

### 5.3.48 mfxMediaAdapterType

enum **mfxMediaAdapterType**

The mfxMediaAdapterType enumerator itemizes types of graphics adapters.

*Values:*

enumerator **MFX_MEDIA_UNKNOWN**

Unknown type.

enumerator **MFX_MEDIA_INTEGRATED**

Integrated graphics adapter.

enumerator **MFX_MEDIA_DISCRETE**

Discrete graphics adapter.

### 5.3.49 mfxMemoryFlags

enum **mfxMemoryFlags**

The mfxMemoryFlags enumerator specifies memory access mode.

*Values:*

enumerator **MFX_MAP_READ**

The surface is mapped for reading.

enumerator **MFX_MAP_WRITE**

The surface is mapped for writing.

enumerator **MFX_MAP_READ_WRITE**

The surface is mapped for reading and writing.

enumerator **MFX_MAP_NOWAIT**

The mapping would be done immediately without any implicit synchronizations.

**Attention**

This flag is optional.

### 5.3.50 MfxNalUnitType

Specifies NAL unit types supported by the HEVC encoder.

enumerator **MFX_HEVC_NALU_TYPE_UNKNOWN**

The encoder will decide what NAL unit type to use.

enumerator **MFX_HEVC_NALU_TYPE_TRAIL_N**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator **MFX_HEVC_NALU_TYPE_TRAIL_R**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator **MFX_HEVC_NALU_TYPE_RADL_N**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator **MFX_HEVC_NALU_TYPE_RADL_R**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator **MFX_HEVC_NALU_TYPE_RASL_N**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator **MFX_HEVC_NALU_TYPE_RASL_R**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator **MFX_HEVC_NALU_TYPE_IDR_W_RADL**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator **MFX_HEVC_NALU_TYPE_IDR_N_LP**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator **MFX_HEVC_NALU_TYPE_CRA_NUT**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

### 5.3.51 mfxPriority

enum **mfxPriority**

The mfxPriority enumerator describes the session priority.

*Values:*

enumerator **MFX_PRIORITY_LOW**

Low priority: the session operation halts when high priority tasks are executing and more than 75% of the CPU is being used for normal priority tasks.

enumerator **MFX_PRIORITY_NORMAL**

Normal priority: the session operation is halted if there are high priority tasks.

enumerator **MFX_PRIORITY_HIGH**

High priority: the session operation blocks other lower priority session operations.

### 5.3.52 mfxResourceType

enum **mfxResourceType**

*Values:*

enumerator **MFX_RESOURCE_SYSTEM_SURFACE**

System memory.

enumerator **MFX_RESOURCE_VA_SURFACE_PTR**

Pointer to VA surface index.

enumerator **MFX_RESOURCE_VA_SURFACE**

Pointer to VA surface index.

enumerator **MFX_RESOURCE_VA_BUFFER_PTR**

Pointer to VA buffer index.

enumerator **MFX_RESOURCE_VA_BUFFER**

Pointer to VA buffer index.

enumerator **MFX_RESOURCE_DX9_SURFACE**

Pointer to IDirect3DSurface9.

enumerator **MFX_RESOURCE_DX11_TEXTURE**

Pointer to ID3D11Texture2D.

enumerator **MFX_RESOURCE_DX12_RESOURCE**

Pointer to ID3D12Resource.

enumerator **MFX_RESOURCE_DMA_RESOURCE**

DMA resource.

enumerator **MFX_RESOURCE_HDDLUNITE_REMOTE_MEMORY**

HDDL Unite Remote memory handle.

### 5.3.53 mfxSkipMode

enum **mfxSkipMode**

The mfxSkipMode enumerator describes the decoder skip-mode options.

*Values:*

enumerator **MFX_SKIPMODE_NOSKIP**

enumerator **MFX_SKIPMODE_MORE**

Do not skip any frames.

enumerator **MFX_SKIPMODE_LESS**

Skip more frames.

### 5.3.54 mfxStatus

enum **mfxStatus**

Itemizes status codes returned by API functions.

*Values:*

enumerator **MFX_ERR_NONE**

No error.

enumerator **MFX_ERR_UNKNOWN**

Unknown error.

enumerator **MFX_ERR_NULL_PTR**

Null pointer.

enumerator **MFX_ERR_UNSUPPORTED**

Unsupported feature.

enumerator **MFX_ERR_MEMORY_ALLOC**

Failed to allocate memory.

enumerator **MFX_ERR_NOT_ENOUGH_BUFFER**

Insufficient buffer at input/output.

enumerator **MFX_ERR_INVALID_HANDLE**

Invalid handle.

enumerator **MFX_ERR_LOCK_MEMORY**

Failed to lock the memory block.

enumerator **MFX_ERR_NOT_INITIALIZED**

Member function called before initialization.

enumerator **MFX_ERR_NOT_FOUND**

The specified object is not found.

enumerator **MFX_ERR_MORE_DATA**

Expect more data at input.

enumerator **MFX_ERR_MORE_SURFACE**

Expect more surface at output.

enumerator **MFX_ERR_ABORTED**

Operation aborted.

enumerator **MFX_ERR_DEVICE_LOST**

Lose the hardware acceleration device.

enumerator **MFX_ERR_INCOMPATIBLE_VIDEO_PARAM**

Incompatible video parameters.

enumerator **MFX_ERR_INVALID_VIDEO_PARAM**

Invalid video parameters.

enumerator **MFX_ERR_UNDEFINED_BEHAVIOR**

Undefined behavior.

enumerator **MFX_ERR_DEVICE_FAILED**

Device operation failure.

enumerator **MFX_ERR_MORE_BITSTREAM**

Expect more bitstream buffers at output.

enumerator **MFX_ERR_GPU_HANG**

Device operation failure caused by GPU hang.

enumerator **MFX_ERR_REALLOC_SURFACE**

Bigger output surface required.

enumerator **MFX_ERR_RESOURCE_MAPPED**

Write access is already acquired and user requested another write access, or read access with MFX_MEMORY_NO_WAIT flag.

enumerator **MFX_ERR_NOT_IMPLEMENTED**

Feature or function not implemented.

enumerator **MFX_ERR_MORE_EXTBUFFER**

Expect additional extended configuration buffer.

enumerator **MFX_WRN_IN_EXECUTION**

The previous asynchronous operation is in execution.

enumerator **MFX_WRN_DEVICE_BUSY**

The hardware acceleration device is busy.

enumerator **MFX_WRN_VIDEO_PARAM_CHANGED**

The video parameters are changed during decoding.

enumerator **MFX_WRN_PARTIAL_ACCELERATION**

Software acceleration is used.

enumerator **MFX_WRN_INCOMPATIBLE_VIDEO_PARAM**

Incompatible video parameters.

enumerator **MFX_WRN_VALUE_NOT_CHANGED**

The value is saturated based on its valid range.

enumerator **MFX_WRN_OUT_OF_RANGE**

The value is out of valid range.

enumerator **MFX_WRN_FILTER_SKIPPED**

One of requested filters has been skipped.

enumerator **MFX_ERR_NONE_PARTIAL_OUTPUT**

Frame is not ready, but bitstream contains partial output.

enumerator **MFX_WRN_ALLOC_TIMEOUT_EXPIRED**

Timeout expired for internal frame allocation.

enumerator **MFX_TASK_DONE**

Task has been completed.

enumerator **MFX_TASK_WORKING**

There is some more work to do.

enumerator **MFX_TASK_BUSY**

Task is waiting for resources.

enumerator **MFX_ERR_MORE_DATA_SUBMIT_TASK**

Return MFX_ERR_MORE_DATA but submit internal asynchronous task.

### 5.3.55 MirroringType

The MirroringType enumerator itemizes mirroring types.

enumerator **MFX_MIRRORING_DISABLED**

enumerator **MFX_MIRRORING_HORIZONTAL**

enumerator **MFX_MIRRORING_VERTICAL**

### 5.3.56 DenoiseMode

The mfxDenoiseMode enumerator itemizes denoise modes.

enum **mfxDenoiseMode**

The mfxDenoiseMode enumerator specifies the mode of denoise.

*Values:*

enumerator **MFX_DENOISE_MODE_DEFAULT**

Default denoise mode. The library selects the most appropriate denoise mode.

enumerator **MFX_DENOISE_MODE_VENDOR**

The enumeration to separate common denoise mode above and vendor specific.

enumerator **MFX_DENOISE_MODE_INTEL_HVS_AUTO_BDRATE**

Indicates auto BD rate improvement in pre-processing before video encoding, ignore Strength.

enumerator **MFX_DENOISE_MODE_INTEL_HVS_AUTO_SUBJECTIVE**

Indicates auto subjective quality improvement in pre-processing before video encoding, ignore Strength.

enumerator **MFX_DENOISE_MODE_INTEL_HVS_AUTO_ADJUST**

Indicates auto adjust subjective quality in post-processing (after decoding) for video playback, ignore Strength.

enumerator **MFX_DENOISE_MODE_INTEL_HVS_PRE_MANUAL**

Indicates manual mode for pre-processing before video encoding, allow to adjust the denoise strength manually.

enumerator **MFX_DENOISE_MODE_INTEL_HVS_POST_MANUAL**

Indicates manual mode for post-processing for video playback, allow to adjust the denoise strength manually.

### 5.3.57 MPEG-2 Profiles

enumerator **MFX_PROFILE_MPEG2_SIMPLE**

enumerator **MFX_PROFILE_MPEG2_MAIN**

enumerator **MFX_PROFILE_MPEG2_HIGH**

### 5.3.58 Multi-view Video Coding Extension Profiles

enumerator **MFX_PROFILE_AVC_MULTIVIEW_HIGH**

Multi-view high profile. The encoding of VDEnc or LowPower ON is not supported.

enumerator **MFX_PROFILE_AVC_STEREO_HIGH**

Stereo high profile. The encoding of VDEnc or LowPower ON is not supported.

### 5.3.59 MVPrecision

The MVPrecision enumerator specifies the motion estimation precision

enumerator **MFX_MVPRECISION_UNKNOWN**

enumerator **MFX_MVPRECISION_INTEGER**

enumerator **MFX_MVPRECISION_HALFPEL**

enumerator **MFX_MVPRECISION_QUARTERPEL**

### 5.3.60 NominalRange

The NominalRange enumerator itemizes pixel's value nominal range.

enumerator **MFX_NOMINALRANGE_UNKNOWN**

Range is not defined.

enumerator **MFX_NOMINALRANGE_0_255**

Range is from 0 to 255.

enumerator **MFX_NOMINALRANGE_16_235**

Range is from 16 to 235.

### 5.3.61 PartialBitstreamOutput

The PartialBitstreamOutput enumerator indicates flags of partial bitstream output type.

enumerator **MFX_PARTIAL_BITSTREAM_NONE**

Do not use partial output

enumerator **MFX_PARTIAL_BITSTREAM_SLICE**

Partial bitstream output will be aligned to slice granularity

enumerator **MFX_PARTIAL_BITSTREAM_BLOCK**

Partial bitstream output will be aligned to user-defined block size granularity

enumerator **MFX_PARTIAL_BITSTREAM_ANY**

Partial bitstream output will be return any coded data available at the end of SyncOperation timeout

### 5.3.62 PayloadCtrlFlags

The PayloadCtrlFlags enumerator itemizes additional payload properties.

enumerator **MFX_PAYLOAD_CTRL_SUFFIX**

Insert this payload into HEVC Suffix SEI NAL-unit.

### 5.3.63 PicStruct

The PicStruct enumerator itemizes picture structure. Use bit-OR'ed values to specify the desired picture type.

enumerator **MFX_PICSTRUCT_UNKNOWN**

Unspecified or mixed progressive/interlaced/field pictures.

enumerator **MFX_PICSTRUCT_PROGRESSIVE**

Progressive picture.

enumerator **MFX_PICSTRUCT_FIELD_TFF**

Top field in first interlaced picture.

enumerator **MFX_PICSTRUCT_FIELD_BFF**

Bottom field in first interlaced picture.

enumerator **MFX_PICSTRUCT_FIELD_REPEATED**

First field repeated: pic_struct=5 or 6 in H.264.

enumerator **MFX_PICSTRUCT_FRAME_DOUBLING**

Double the frame for display: pic_struct=7 in H.264.

enumerator **MFX_PICSTRUCT_FRAME_TRIPLING**

Triple the frame for display: pic_struct=8 in H.264.

enumerator **MFX_PICSTRUCT_FIELD_SINGLE**

Single field in a picture.

enumerator **MFX_PICSTRUCT_FIELD_TOP**

Top field in a picture: pic_struct = 1 in H.265.

enumerator **MFX_PICSTRUCT_FIELD_BOTTOM**

Bottom field in a picture: pic_struct = 2 in H.265.

enumerator **MFX_PICSTRUCT_FIELD_PAIRED_PREV**

Paired with previous field: pic_struct = 9 or 10 in H.265.

enumerator **MFX_PICSTRUCT_FIELD_PAIRED_NEXT**

Paired with next field: pic_struct = 11 or 12 in H.265

### 5.3.64 PicType

The PicType enumerator itemizes picture type.

enumerator **MFX_PICTYPE_UNKNOWN**

Picture type is unknown.

enumerator **MFX_PICTYPE_FRAME**

Picture is a frame.

enumerator **MFX_PICTYPE_TOPFIELD**

Picture is a top field.

enumerator **MFX_PICTYPE_BOTTOMFIELD**

Picture is a bottom field.

### 5.3.65 PRefType

The PRefType enumerator itemizes models of reference list construction and DPB management when GopRefDist=1.

enumerator **`MFX_P_REF_DEFAULT`**

Allow encoder to decide.

enumerator **`MFX_P_REF_SIMPLE`**

Regular sliding window used for DPB removal process.

enumerator **`MFX_P_REF_PYRAMID`**

Let N be the max reference list's size. Encoder treats each N's frame as a 'strong' reference and the others as 'weak' references. The encoder uses a 'weak' reference only for prediction of the next frame and removes it from DPB immediately after use. 'Strong' references are removed from DPB by a sliding window.

### 5.3.66 TuneQuality

The TuneQuality enumerator specifies tuning option for encode. Multiple tuning options can be combined using bit mask.

enumerator **`MFX_ENCODE_TUNE_OFF`**

Tuning quality is disabled.

enumerator **`MFX_ENCODE_TUNE_PSNR`**

The encoder optimizes quality according to Peak Signal-to-Noise Ratio (PSNR) metric.

enumerator **`MFX_ENCODE_TUNE_SSIM`**

The encoder optimizes quality according to Structural Similarity Index Measure (SSIM) metric.

enumerator **`MFX_ENCODE_TUNE_MS_SSIM`**

The encoder optimizes quality according to Multi-Scale Structural Similarity Index Measure (MS-SSIM) metric.

enumerator **`MFX_ENCODE_TUNE_VMAF`**

The encoder optimizes quality according to Video Multi-Method Assessment Fusion (VMAF) metric.

enumerator **`MFX_ENCODE_TUNE_PERCEPTUAL`**

The encoder makes perceptual quality optimization.

### 5.3.67 Protected

The Protected enumerator describes the protection schemes.

enumerator **`MFX_PROTECTION_CENC_WV_CLASSIC`**

The protection scheme is based on the Widevine* DRM from Google*.

enumerator **`MFX_PROTECTION_CENC_WV_GOOGLE_DASH`**

The protection scheme is based on the Widevine* Modular DRM* from Google*.

### 5.3.68 RateControlMethod

The RateControlMethod enumerator itemizes bitrate control methods.

enumerator **MFX_RATECONTROL_CBR**

Use the constant bitrate control algorithm.

enumerator **MFX_RATECONTROL_VBR**

Use the variable bitrate control algorithm.

enumerator **MFX_RATECONTROL_CQP**

Use the constant quantization parameter algorithm.

enumerator **MFX_RATECONTROL_AVBR**

Use the average variable bitrate control algorithm.

enumerator **MFX_RATECONTROL_LA**

Use the VBR algorithm with look ahead. It is a special bitrate control mode in the AVC encoder that has been designed to improve encoding quality. It works by performing extensive analysis of several dozen frames before the actual encoding and as a side effect significantly increases encoding delay and memory consumption.

The only available rate control parameter in this mode is *mfxInfoMFX::TargetKbps*. Two other parameters, MaxKbps and InitialDelayInKB, are ignored. To control LA depth the application can use *mfxExtCodingOption2::LookAheadDepth* parameter.

This method is not HRD compliant.

enumerator **MFX_RATECONTROL_ICQ**

Use the Intelligent Constant Quality algorithm. This algorithm improves subjective video quality of encoded stream. Depending on content, it may or may not decrease objective video quality. Only one control parameter is used - quality factor, specified by *mfxInfoMFX::ICQQuality*.

enumerator **MFX_RATECONTROL_VCM**

Use the Video Conferencing Mode algorithm. This algorithm is similar to the VBR and uses the same set of parameters *mfxInfoMFX::InitialDelayInKB*, TargetKbpsandMaxKbps. It is tuned for IPPP GOP pattern and streams with strong temporal correlation between frames. It produces better objective and subjective video quality in these conditions than other bitrate control algorithms. It does not support interlaced content, B-frames and produced stream is not HRD compliant.

enumerator **MFX_RATECONTROL_LA_ICQ**

Use Intelligent Constant Quality algorithm with look ahead. Quality factor is specified by *mfxInfoMFX::ICQQuality*. To control LA depth the application can use *mfxExtCodingOption2::LookAheadDepth* parameter.

This method is not HRD compliant.

enumerator **MFX_RATECONTROL_LA_HRD**

MFX_RATECONTROL_LA_EXT has been removed

Use HRD compliant look ahead rate control algorithm.

enumerator **MFX_RATECONTROL_QVBR**

Use the variable bitrate control algorithm with constant quality. This algorithm trying to achieve the target subjective quality with the minimum number of bits, while the bitrate constraint and HRD compliance are satisfied. It uses the same set of parameters as VBR and quality factor specified by *mfxExtCodingOption3::QVBRQuality*.

### 5.3.69 ROImode

The ROImode enumerator itemizes QP adjustment mode for ROIs.

enumerator **MFX_ROI_MODE_PRIORITY**

Priority mode.

enumerator **MFX_ROI_MODE_QP_DELTA**

QP mode

enumerator **MFX_ROI_MODE_QP_VALUE**

Absolute QP

### 5.3.70 Rotation

The Rotation enumerator itemizes the JPEG rotation options.

enumerator **MFX_ROTATION_0**

No rotation.

enumerator **MFX_ROTATION_90**

90 degree rotation.

enumerator **MFX_ROTATION_180**

180 degree rotation.

enumerator **MFX_ROTATION_270**

270 degree rotation.

### 5.3.71 SampleAdaptiveOffset

The SampleAdaptiveOffset enumerator uses bit-ORed values to itemize corresponding HEVC encoding feature.

enumerator **MFX_SAO_UNKNOWN**

Use default value for platform/TargetUsage.

enumerator **MFX_SAO_DISABLE**

Disable SAO. If set during Init leads to SPS sample_adaptive_offset_enabled_flag = 0. If set during Runtime, leads to to slice_sao_luma_flag = 0 and slice_sao_chroma_flag = 0 for current frame.

enumerator **MFX_SAO_ENABLE_LUMA**

Enable SAO for luma (slice_sao_luma_flag = 1).

enumerator **MFX_SAO_ENABLE_CHROMA**

Enable SAO for chroma (slice_sao_chroma_flag = 1).

### 5.3.72 ScalingMode

The ScalingMode enumerator itemizes variants of scaling filter implementation.

enumerator **MFX_SCALING_MODE_DEFAULT**

Default scaling mode. The library selects the most appropriate scaling method.

enumerator **MFX_SCALING_MODE_LOWPOWER**

Low power scaling mode which is applicable for library implementations. The exact scaling algorithm is defined by the library.

enumerator **MFX_SCALING_MODE_QUALITY**

The best quality scaling mode.

enumerator **MFX_SCALING_MODE_VENDOR**

The enumeration to separate common scaling controls above and vendor specific.

enumerator **MFX_SCALING_MODE_INTEL_GEN_COMPUTE**

enumerator **MFX_SCALING_MODE_INTEL_GEN_VDBOX**

The mode to run scaling operation on Execution Units (EUs).

enumerator **MFX_SCALING_MODE_INTEL_GEN_VEBOX**

The special optimization mode where scaling operation running on SFC (Scaler & Format Converter) is coupled with VDBOX (also known as Multi-Format Codec Engines). This mode is applicable for DECODE_VPP domain functions.

### 5.3.73 ScenarioInfo

The ScenarioInfo enumerator itemizes scenarios for the encoding session.

enumerator **MFX_SCENARIO_UNKNOWN**

enumerator **MFX_SCENARIO_DISPLAY_REMOTING**

enumerator **MFX_SCENARIO_VIDEO_CONFERENCE**

enumerator **MFX_SCENARIO_ARCHIVE**

enumerator **MFX_SCENARIO_LIVE_STREAMING**

enumerator **MFX_SCENARIO_CAMERA_CAPTURE**

enumerator **MFX_SCENARIO_VIDEO_SURVEILLANCE**

enumerator **MFX_SCENARIO_GAME_STREAMING**

enumerator **MFX_SCENARIO_REMOTE_GAMING**

### 5.3.74 SegmentFeature

The SegmentFeature enumerator indicates features enabled for the segment. These values are used with the mfxVP9SegmentParam::FeatureEnabled parameter.

enumerator **MFX_VP9_SEGMENT_FEATURE_QINDEX**

Quantization index delta.

enumerator **MFX_VP9_SEGMENT_FEATURE_LOOP_FILTER**

Loop filter level delta.

enumerator **MFX_VP9_SEGMENT_FEATURE_REFERENCE**

Reference frame.

enumerator **MFX_VP9_SEGMENT_FEATURE_SKIP**

Skip.

### 5.3.75 SegmentIdBlockSize

The SegmentIdBlockSize enumerator indicates the block size represented by each segment_id in segmentation map. These values are used with the mfxExtVP9Segmentation::SegmentIdBlockSize parameter.

enumerator **MFX_VP9_SEGMENT_ID_BLOCK_SIZE_UNKNOWN**

Unspecified block size.

enumerator **MFX_VP9_SEGMENT_ID_BLOCK_SIZE_8x8**

8x8 block size.

enumerator **MFX_VP9_SEGMENT_ID_BLOCK_SIZE_16x16**

16x16 block size.

enumerator **MFX_VP9_SEGMENT_ID_BLOCK_SIZE_32x32**

32x32 block size.

enumerator **MFX_VP9_SEGMENT_ID_BLOCK_SIZE_64x64**

64x64 block size.

### 5.3.76 SkipFrame

The SkipFrame enumerator is used to define usage of mfxEncodeCtrl::SkipFrame parameter.

enumerator **MFX_SKIPFRAME_NO_SKIP**

Frame skipping is disabled, *mfxEncodeCtrl::SkipFrame* is ignored.

enumerator **MFX_SKIPFRAME_INSERT_DUMMY**

Skipping is allowed, when *mfxEncodeCtrl::SkipFrame* is set encoder inserts into bitstream frame where all macroblocks are encoded as skipped. Only non-reference P- and B-frames can be skipped. If GopRefDist = 1 and *mfxEncodeCtrl::SkipFrame* is set for reference P-frame, it will be encoded as non-reference.

enumerator **MFX_SKIPFRAME_INSERT_NOTHING**

Similar to MFX_SKIPFRAME_INSERT_DUMMY, but when *mfxEncodeCtrl::SkipFrame* is set encoder inserts nothing into bitstream.

enumerator **MFX_SKIPFRAME_BRC_ONLY**

*mfxEncodeCtrl::SkipFrame* indicates number of missed frames before the current frame. Affects only BRC, current frame will be encoded as usual.

### 5.3.77 TargetUsage

The TargetUsage enumerator itemizes a range of numbers from MFX_TARGETUSAGE_1, best quality, to MFX_TARGETUSAGE_7, best speed. It indicates trade-offs between quality and speed. The application can use any number in the range. The actual number of supported target usages depends on implementation. If the specified target usage is not supported, the encoder will use the closest supported value.

enumerator **MFX_TARGETUSAGE_1**

Best quality

enumerator **MFX_TARGETUSAGE_2**

enumerator **MFX_TARGETUSAGE_3**

enumerator **MFX_TARGETUSAGE_4**

Balanced quality and speed.

enumerator **MFX_TARGETUSAGE_5**

enumerator **MFX_TARGETUSAGE_6**

enumerator **MFX_TARGETUSAGE_7**

Best speed

enumerator **MFX_TARGETUSAGE_UNKNOWN**

Unspecified target usage.

enumerator **MFX_TARGETUSAGE_BEST_QUALITY**

Best quality.

enumerator **MFX_TARGETUSAGE_BALANCED**

Balanced quality and speed.

enumerator **MFX_TARGETUSAGE_BEST_SPEED**

Best speed.

### 5.3.78 TelecinePattern

The TelecinePattern enumerator itemizes telecine patterns.

enumerator **MFX_TELECINE_PATTERN_32**

3:2 telecine.

enumerator **MFX_TELECINE_PATTERN_2332**

2:3:3:2 telecine.

enumerator **MFX_TELECINE_PATTERN_FRAME_REPEAT**

One frame repeat telecine.

enumerator **MFX_TELECINE_PATTERN_41**

4:1 telecine.

enumerator **MFX_TELECINE_POSITION_PROVIDED**

User must provide position inside a sequence of 5 frames where the artifacts start.

### 5.3.79 TimeStampCalc

The TimeStampCalc enumerator itemizes time-stamp calculation methods.

enumerator **MFX_TIMESTAMPCALC_UNKNOWN**

The time stamp calculation is based on the input frame rate if time stamp is not explicitly specified.

enumerator **MFX_TIMESTAMPCALC_TELECINE**

Adjust time stamp to 29.97fps on 24fps progressively encoded sequences if telecine attributes are available in the bitstream and time stamp is not explicitly specified. The input frame rate must be specified.

### 5.3.80 TransferMatrix

The TransferMatrix enumerator itemizes color transfer matrices.

enumerator **MFX_TRANSFERMATRIX_UNKNOWN**

Transfer matrix is not specified

enumerator **MFX_TRANSFERMATRIX_BT709**

Transfer matrix from ITU-R BT.709 standard.

enumerator **MFX_TRANSFERMATRIX_BT601**

Transfer matrix from ITU-R BT.601 standard.

### 5.3.81 TrellisControl

The TrellisControl enumerator is used to control trellis quantization in AVC encoder. The application can turn it on or off for any combination of I, P, and B frames by combining different enumerator values. For example, MFX_TRELLIS_I | MFX_TRELLIS_B turns it on for I and B frames.

enumerator **MFX_TRELLIS_UNKNOWN**

Default value, it is up to the encoder to turn trellis quantization on or off.

enumerator **MFX_TRELLIS_OFF**

Turn trellis quantization off for all frame types.

enumerator **MFX_TRELLIS_I**

Turn trellis quantization on for I-frames.

enumerator **MFX_TRELLIS_P**

Turn trellis quantization on for P-frames.

enumerator **MFX_TRELLIS_B**

Turn trellis quantization on for B-frames.

### 5.3.82 VP9ReferenceFrame

The VP9ReferenceFrame enumerator itemizes reference frame type by the mfxVP9SegmentParam::ReferenceFrame parameter.

enumerator **MFX_VP9_REF_INTRA**

Intra.

enumerator **MFX_VP9_REF_LAST**

Last.

enumerator **MFX_VP9_REF_GOLDEN**

Golden.

enumerator **MFX_VP9_REF_ALTREF**

Alternative reference.

### 5.3.83 VPPFieldProcessingMode

The VPPFieldProcessingMode enumerator is used to control VPP field processing algorithm.

enumerator **MFX_VPP_COPY_FRAME**

Copy the whole frame.

enumerator **MFX_VPP_COPY_FIELD**

Copy only one field.

enumerator **MFX_VPP_SWAP_FIELDS**

Swap top and bottom fields.

### 5.3.84 WeightedPred

The WeightedPred enumerator itemizes weighted prediction modes.

enumerator **MFX_WEIGHTED_PRED_UNKNOWN**

Allow encoder to decide.

enumerator **MFX_WEIGHTED_PRED_DEFAULT**

Use default weighted prediction.

enumerator **MFX_WEIGHTED_PRED_EXPLICIT**

Use explicit weighted prediction.

enumerator **MFX_WEIGHTED_PRED_IMPLICIT**

Use implicit weighted prediction (for B-frames only).

### 5.3.85 FilmGrainFlags

The FilmGrainFlags enumerator itemizes flags in AV1 film grain parameters.

enumerator **MFX_FILM_GRAIN_NO**

Film grain isn't added to this frame.

enumerator **MFX_FILM_GRAIN_APPLY**

Film grain is added to this frame.

enumerator **MFX_FILM_GRAIN_UPDATE**

New set of film grain parameters is sent for this frame.

enumerator **MFX_FILM_GRAIN_CHROMA_SCALING_FROM_LUMA**

Chroma scaling is inferred from luma scaling.

enumerator **MFX_FILM_GRAIN_OVERLAP**

Overlap between film grain blocks is applied.

enumerator **MFX_FILM_GRAIN_CLIP_TO_RESTRICTED_RANGE**

Clipping to the restricted (studio) range is applied after adding the film grain.

### 5.3.86 mfxHyperMode

enum **mfxHyperMode**

The mfxHyperMode enumerator describes HyperMode implementation behavior.

*Values:*

enumerator **MFX_HYPERMODE_OFF**

Don't use HyperMode implementation.

enumerator **MFX_HYPERMODE_ON**

Enable HyperMode implementation and return error if some issue on initialization.

enumerator **MFX_HYPERMODE_ADAPTIVE**

Enable HyperMode implementation and switch to single fallback if some issue on initialization.

### 5.3.87 mfxPoolAllocationPolicy

enum **mfxPoolAllocationPolicy**

Specifies the surface pool allocation policies.

*Values:*

enumerator **MFX_ALLOCATION_OPTIMAL**

Recommends to limit max pool size by sum of requested surfaces asked by components.

enumerator **MFX_ALLOCATION_UNLIMITED**

Dynamic allocation with no limit.

enumerator **MFX_ALLOCATION_LIMITED**

Max pool size is limited by NumberToPreAllocate + DeltaToAllocateOnTheFly.

### 5.3.88 mfxVPPPoolType

enum **mfxVPPPoolType**

*Values:*

enumerator **MFX_VPP_POOL_IN**

Input pool.

enumerator **MFX_VPP_POOL_OUT**

Output pool.

### 5.3.89 mfxAV1SegmentIdBlockSize

The mfxAV1SegmentIdBlockSize enumerator indicates the block size represented by each segment_id in segmentation map.

enum **mfxAV1SegmentIdBlockSize**

The AV1 SegmentIdBlockSize enumerator indicates the block size represented by each segment_id in segmentation map. These values are used with the *mfxExtAV1Segmentation::SegmentIdBlockSize* parameter.

*Values:*

enumerator **MFX_AV1_SEGMENT_ID_BLOCK_SIZE_UNSPECIFIED**

Unspecified block size.

enumerator **MFX_AV1_SEGMENT_ID_BLOCK_SIZE_4x4**

block size 4x4

enumerator **MFX_AV1_SEGMENT_ID_BLOCK_SIZE_8x8**

block size 8x8

enumerator **MFX_AV1_SEGMENT_ID_BLOCK_SIZE_16x16**

block size 16x16

enumerator **MFX_AV1_SEGMENT_ID_BLOCK_SIZE_32x32**

block size 32x32

enumerator **MFX_AV1_SEGMENT_ID_BLOCK_SIZE_64x64**

block size 64x64

enumerator **MFX_AV1_SEGMENT_ID_BLOCK_SIZE_128x128**

block size 128x128

### 5.3.90 AV1SegmentFeature

The AV1SegmentFeature enumerator indicates features enabled for the segment.

enumerator **MFX_AV1_SEGMENT_FEATURE_ALT_QINDEX**

use alternate Quantizer.

enumerator **MFX_AV1_SEGMENT_FEATURE_ALT_LF_Y_VERT**

use alternate loop filter value on y plane vertical.

enumerator **MFX_AV1_SEGMENT_FEATURE_ALT_LF_Y_HORZ**

use alternate loop filter value on y plane horizontal.

enumerator **MFX_AV1_SEGMENT_FEATURE_ALT_LF_U**

use alternate loop filter value on u plane.

enumerator **MFX_AV1_SEGMENT_FEATURE_ALT_LF_V**

use alternate loop filter value on v plane.

enumerator **MFX_AV1_SEGMENT_FEATURE_REFERENCE**

use segment reference frame.

enumerator **MFX_AV1_SEGMENT_FEATURE_SKIP**

use segment (0,0) + skip mode.

enumerator **MFX_AV1_SEGMENT_FEATURE_GLOBALMV**

use global motion vector.

### 5.3.91 mfxEncodeBlkStatsMemLayout

enum **mfxEncodeBlkStatsMemLayout**

< The enum to specify memory layout for statistics.

*Values:*

enumerator **MFX_ENCODESTATS_MEMORY_LAYOUT_DEFAULT**

The default memory layout for statistics.

### 5.3.92 mfxEncodeStatsMode

enum **mfxEncodeStatsMode**

*Values:*

enumerator **MFX_ENCODESTATS_MODE_DEFAULT**

Encode mode is selected by the implementation.

enumerator **MFX_ENCODESTATS_MODE_ENCODE**

Full encode mode.

### 5.3.93 EncodeStatsLevel

Flags to specify what statistics will be reported by the implementation.

enumerator **MFX_ENCODESTATS_LEVEL_BLK**

Block level statistics.

enumerator **MFX_ENCODESTATS_LEVEL_FRAME**

Frame level statistics.

### 5.3.94 mfxSurfaceComponent

enum **mfxSurfaceComponent**

The mfxSurfaceComponent enumerator specifies the internal surface pool to use when importing surfaces.

*Values:*

enumerator **MFX_SURFACE_COMPONENT_UNKNOWN**

Unknown surface component.

enumerator **MFX_SURFACE_COMPONENT_ENCODE**

Shared surface for encoding.

enumerator **MFX_SURFACE_COMPONENT_DECODE**

Shared surface for decoding.

enumerator **MFX_SURFACE_COMPONENT_VPP_INPUT**

Shared surface for VPP input.

enumerator **MFX_SURFACE_COMPONENT_VPP_OUTPUT**

Shared surface for VPP output.

### 5.3.95 mfxSurfaceType

enum **mfxSurfaceType**

The mfxSurfaceType enumerator specifies the surface type described by *mfxSurfaceHeader*.

*Values:*

enumerator **MFX_SURFACE_TYPE_UNKNOWN**

Unknown surface type.

enumerator **MFX_SURFACE_TYPE_D3D11_TEX2D**

D3D11 surface of type ID3D11Texture2D.

enumerator **MFX_SURFACE_TYPE_VAAPI**

VA-API surface.

enumerator **MFX_SURFACE_TYPE_OPENCL_IMG2D**

OpenCL 2D image (cl_mem).

### 5.3.96 mfxStructureType

enum **mfxStructureType**

The mfxStructureType enumerator specifies the structure type for configuration with the string interface.

*Values:*

enumerator **MFX_STRUCTURE_TYPE_UNKNOWN**

Unknown structure type.

enumerator **MFX_STRUCTURE_TYPE_VIDEO_PARAM**

Structure of type *mfxVideoParam*.

## 5.4 Define Reference

### 5.4.1 API

**MFX_DECODERDESCRIPTION_VERSION**

**MFX_DEVICEDESCRIPTION_VERSION**

The current version of *mfxDeviceDescription* structure.

**MFX_ENCODERDESCRIPTION_VERSION**

**MFX_FRAMESURFACE1_VERSION**

**MFX_FRAMESURFACEINTERFACE_VERSION**

**MFX_IMPLDESCRIPTION_VERSION**

The current version of *mfxImplDescription* structure.

**MFX_LEGACY_VERSION**

The corresponding version of the Intel(r) Media SDK legacy API that is used as a basis for the current API.

**MFX_STRUCT_VERSION**(MAJOR, MINOR)

**MFX_VARIANT_VERSION**

**MFX_VERSION**

**MFX_VERSION_MAJOR**

**MFX_VERSION_MINOR**

**MFX_VPPDESCRIPTION_VERSION**

**MFX_SURFACEARRAY_VERSION**

## 5.5 Type Reference



### 5.5.1 Basic Types

typedef char **mfxChar**

UTF-8 byte.

typedef float **mfxF32**

Single-precision floating point, 32 bit type.

typedef double **mfxF64**

Double-precision floating point, 64 bit type.

typedef void ***mfxHDL**

Handle type.

typedef char **mfxI8**
: Signed integer, 8 bit type.

typedef short **mfxI16**
: Signed integer, 16 bit type.

typedef int **mfxI32**
: Signed integer, 32 bit type.

typedef long long **mfxI64**
: Signed integer, 64 bit type.

typedef int **mfxL32**
: Signed integer, 32 bit type.

typedef *mfxHDL* **mfxMemId**
: Memory ID type.

typedef void ***mfxThreadTask**
: Thread task type.

typedef unsigned char **mfxU8**
: Unsigned integer, 8 bit type.

typedef unsigned short **mfxU16**
: Unsigned integer, 16 bit type.

typedef unsigned int **mfxU32**
: Unsigned integer, 32 bit type.

typedef unsigned long long **mfxU64**
: Unsigned integer, 64 bit type.

typedef unsigned int **mfxUL32**
: Unsigned integer, 32 bit type.

### 5.5.2 Typedefs

typedef struct _mfxConfig ***mfxConfig**
: Config handle.

typedef struct _mfxLoader ***mfxLoader**
: Loader handle.

typedef struct _mfxSession ***mfxSession**

Session handle.

typedef struct _mfxSyncPoint ***mfxSyncPoint**

Synchronization point object handle.

typedef *mfxExtAVCRefListCtrl* **mfxExtRefListCtrl**

typedef *mfxExtAVCEncodedFrameInfo* **mfxExtEncodedFrameInfo**

## 5.6 Dispatcher API

Use the Dispatcher API to load and execute the appropriate library implementation and get capabilities for the implementations available on the platform.

### 5.6.1 Dispatcher API Function Reference

#### API

- *MFXCreateConfig*
- *MFXCreateSession*
- *MFXDispReleaseImplDescription*
- *MFXEnumImplementations*
- *MFXLoad*
- *MFXSetConfigFilterProperty*
- *MFXUnload*

#### MFXCreateConfig

*mfxConfig* **MFXCreateConfig**(*mfxLoader* loader)

Creates dispatcher configuration.

Creates the dispatcher internal configuration, which is used to filter out available implementations. This configuration is used to walk through selected implementations to gather more details and select the appropriate implementation to load. The loader object remembers all created mfxConfig objects and destroys them during the mfxUnload function call.

Multiple configurations per single mfxLoader object are possible.

Usage example:

```
mfxLoader loader = MFXLoad();
mfxConfig cfg = MFXCreateConfig(loader);
MFXCreateSession(loader,0,&session);
```

**Since**

This function is available since API version 2.0.

**Parameters**

**loader** – **[in]** Loader handle.

**Returns**

Config handle or NULL pointer is failed.

### MFXCreateSession

*mfxStatus* **MFXCreateSession**(*mfxLoader* loader, *mfxU32* i, *mfxSession* *session)

Loads and initializes the implementation.

```
mfxLoader loader = MFXLoad();
int i=0;
while(1) {
   mfxImplDescription *idesc;
   MFXEnumImplementations(loader, i, MFX_IMPLCAPS_IMPLDESCSTRUCTURE, (mfxHDL*)&
↪idesc);
   if(is_good(idesc)) {
       MFXCreateSession(loader, i,&session);
       // ...
       MFXDispReleaseImplDescription(loader, idesc);
   }
   else
   {
       MFXDispReleaseImplDescription(loader, idesc);
       break;
   }
}
```

**Since**

This function is available since API version 2.0.

**Parameters**

- **loader** – **[in]** Loader handle.
- **i** – **[in]** Index of the implementation.
- **session** – **[out]** Pointer to the session handle.

**Returns**

MFX_ERR_NONE The function completed successfully. The session contains a pointer to the session handle.

MFX_ERR_NULL_PTR If loader is NULL.

MFX_ERR_NULL_PTR If session is NULL.

MFX_ERR_NOT_FOUND Provided index is out of possible range.

### MFXDispReleaseImplDescription

*mfxStatus* **MFXDispReleaseImplDescription**(*mfxLoader* loader, *mfxHDL* hdl)

Destroys handle allocated by the MFXEnumImplementations function.

**Since**

This function is available since API version 2.0.

**Parameters**

- **loader** – **[in]** Loader handle.
- **hdl** – **[in]** Handle to destroy. Can be equal to NULL.

**Returns**

MFX_ERR_NONE The function completed successfully.

MFX_ERR_NULL_PTR If loader is NULL.

MFX_ERR_INVALID_HANDLE Provided hdl handle is not associated with this loader.

### MFXEnumImplementations

*mfxStatus* **MFXEnumImplementations**(*mfxLoader* loader, *mfxU32* i, *mfxImplCapsDeliveryFormat* format, *mfxHDL* *idesc)

Iterates over filtered out implementations to gather their details. This function allocates memory to store a structure or string corresponding to the type specified by format. For example, if format is set to MFX_IMPLCAPS_IMPLDESCSTRUCTURE, then idesc will return a pointer to a structure of type *mfxImplDescription*. Use the MFXDispReleaseImplDescription function to free memory allocated to this structure or string.

**Since**

This function is available since API version 2.0.

**Parameters**

- **loader** – **[in]** Loader handle.
- **i** – **[in]** Index of the implementation.
- **format** – **[in]** Format in which capabilities need to be delivered. See the mfxImplCapsDeliveryFormat enumerator for more details.
- **idesc** – **[out]** Pointer to the structure or string corresponding to the requested format.

**Returns**

MFX_ERR_NONE The function completed successfully. The idesc contains valid information.

MFX_ERR_NULL_PTR If loader is NULL.

MFX_ERR_NULL_PTR If idesc is NULL.

MFX_ERR_NOT_FOUND Provided index is out of possible range.

MFX_ERR_UNSUPPORTED If requested format is not supported.

### MFXLoad

*mfxLoader* **MFXLoad**(void)

Creates the loader.

**Since**

This function is available since API version 2.0.

**Returns**

Loader Loader handle or NULL if failed.

### MFXSetConfigFilterProperty

*mfxStatus* **MFXSetConfigFilterProperty**(*mfxConfig* config, const *mfxU8* *name, *mfxVariant* value)

Adds additional filter properties (any fields of the *mfxImplDescription* structure) to the configuration of the loader object.

**Since**

This function is available since API version 2.0.

---

**Note:** Each new call with the same parameter name will overwrite the previously set value. This may invalidate other properties.

---

**Parameters**

- **config** – **[in]** Config handle.
- **name** – **[in]** Name of the parameter (see *mfxImplDescription* structure and example).
- **value** – **[in]** Value of the parameter.

**Returns**

MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR If config is NULL.

MFX_ERR_NULL_PTR If name is NULL.

MFX_ERR_NOT_FOUND If name contains unknown parameter name. MFX_ERR_UNSUPPORTED If value data type does not equal the parameter with provided name.

### MFXUnload

void **MFXUnload**(*mfxLoader* loader)

Destroys the dispatcher.

**Since**

This function is available since API version 2.0.

**Parameters**

**loader** – **[in]** Loader handle.

## 5.6.2 Dispatcher API Structure Reference

### API

- *mfxDecoderDescription*
- *mfxDeviceDescription*
- *mfxEncoderDescription*
- *mfxImplDescription*
- *mfxVariant*
- *mfxVPPDescription*
- *mfxAccelerationModeDescription*
- *mfxImplementedFunctions*
- *mfxExtendedDeviceId*
- *mfxPoolPolicyDescription*
- *extDeviceUUID*

### mfxDecoderDescription

struct **mfxDecoderDescription**

The *mfxDecoderDescription* structure represents the description of a decoder.

#### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumCodecs**

Number of supported decoders.

struct *mfxDecoderDescription*::*decoder* ***Codecs**

Pointer to the array of decoders.

struct **decoder**

This structure represents the decoder description.

### Public Members

*mfxU32* **CodecID**

Decoder ID in FourCC format.

*mfxU16* **reserved**[8]

Reserved for future use.

*mfxU16* **MaxcodecLevel**

Maximum supported codec level. See the CodecProfile enumerator for possible values.

*mfxU16* **NumProfiles**

Number of supported profiles.

struct *mfxDecoderDescription*::*decoder*::*decprofile* ***Profiles**

Pointer to the array of profiles supported by the codec.

struct **decprofile**

This structure represents the codec profile description.

#### Public Members

*mfxU32* **Profile**

Profile ID. See the CodecProfile enumerator for possible values.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumMemTypes**

Number of supported memory types.

struct *mfxDecoderDescription*::*decoder*::*decprofile*::*decmemdesc* ***MemDesc**

Pointer to the array of memory types.

struct **decmemdesc**

This structure represents the underlying details of the memory type.

### Public Members

*mfxResourceType* **MemHandleType**

Memory handle type.

*mfxRange32U* **Width**

Range of supported image widths.

*mfxRange32U* **Height**

Range of supported image heights.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumColorFormats**

Number of supported output color formats.

*mfxU32* ***ColorFormats**

Pointer to the array of supported output color formats (in FOURCC).

## mfxDeviceDescription

struct **mfxDeviceDescription**

This structure represents device description.

### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **reserved**[6]

reserved for future use.

*mfxU16* **MediaAdapterType**

Graphics adapter type. See the mfxMediaAdapterType enumerator for a list of possible values.

*mfxChar* **DeviceID**[MFX_STRFIELD_LEN]

Null terminated string with device ID.

*mfxU16* **NumSubDevices**

Number of available uniform sub-devices. Pure software implementation can report 0.

struct *mfxDeviceDescription*::*subdevices* ***SubDevices**

Pointer to the array of available sub-devices.

struct **subdevices**

This structure represents sub-device description.

#### Public Members

*mfxU32* **Index**

Index of the sub-device, started from 0 and increased by 1.

*mfxChar* **SubDeviceID**[MFX_STRFIELD_LEN]

Null terminated string with unique sub-device ID, mapped to the system ID.

*mfxU32* **reserved**[7]

reserved for future use.

### mfxEncoderDescription

struct **mfxEncoderDescription**

This structure represents an encoder description.

#### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumCodecs**

Number of supported encoders.

struct *mfxEncoderDescription*::*encoder* ***Codecs**

Pointer to the array of encoders.

struct **encoder**

This structure represents encoder description.

### Public Members

*mfxU32* **CodecID**

Encoder ID in FourCC format.

*mfxU16* **MaxcodecLevel**

Maximum supported codec level. See the CodecProfile enumerator for possible values.

*mfxU16* **BiDirectionalPrediction**

Indicates B-frames support.

*mfxU16* **ReportedStats**

Indicates what type of statistics can be reported: block/slice/tile/frame.

*mfxU16* **reserved**[6]

Reserved for future use.

*mfxU16* **NumProfiles**

Number of supported profiles.

struct *mfxEncoderDescription*::*encoder*::*encprofile* ***Profiles**

Pointer to the array of profiles supported by the codec.

struct **encprofile**

This structure represents the codec profile description.

#### Public Members

*mfxU32* **Profile**

Profile ID. See the CodecProfile enumerator for possible values.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumMemTypes**

Number of supported memory types.

struct *mfxEncoderDescription*::*encoder*::*encprofile*::*encmemdesc* ***MemDesc**

Pointer to the array of memory types.

struct **encmemdesc**

This structure represents the underlying details of the memory type.

#### Public Members

*mfxResourceType* **MemHandleType**

Memory handle type.

*mfxRange32U* **Width**

Range of supported image widths.

*mfxRange32U* **Height**

Range of supported image heights.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumColorFormats**

Number of supported input color formats.

*mfxU32* ***ColorFormats**

Pointer to the array of supported input color formats (in FOURCC).

### mfxImplDescription

struct **mfxImplDescription**

This structure represents the implementation description.

#### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxImplType* **Impl**

Impl type: software/hardware.

*mfxAccelerationMode* **AccelerationMode**

Default Hardware acceleration stack to use. OS dependent parameter. Use VA for Linux* and DX* for Windows*.

*mfxVersion* **ApiVersion**

Supported API version.

*mfxChar* **ImplName**[MFX_IMPL_NAME_LEN]

Null-terminated string with implementation name given by vendor.

*mfxChar* **License**[MFX_STRFIELD_LEN]

Null-terminated string with comma-separated list of license names of the implementation.

*mfxChar* **Keywords**[MFX_STRFIELD_LEN]

Null-terminated string with comma-separated list of keywords specific to this implementation that dispatcher can search for.

*mfxU32* **VendorID**

Standard vendor ID 0x8086 - Intel.

*mfxU32* **VendorImplID**

Vendor specific number with given implementation ID.

*mfxDeviceDescription* **Dev**

Supported device.

*mfxDecoderDescription* **Dec**

Decoder configuration.

*mfxEncoderDescription* **Enc**

Encoder configuration.

*mfxVPPDescription* **VPP**

VPP configuration.

*mfxAccelerationModeDescription* **AccelerationModeDescription**

Supported acceleration modes.

*mfxPoolPolicyDescription* **PoolPolicies**

Supported surface pool polices.

*mfxU32* **reserved**[8]

Reserved for future use.

*mfxU32* **NumExtParam**

Number of extension buffers. Reserved for future use. Must be 0.

*mfxExtBuffer* ****ExtParam**

Array of extension buffers.

*mfxU64* **Reserved2**

Reserved for future use.

union *mfxImplDescription*::[anonymous] **ExtParams**

Extension buffers. Reserved for future.

### mfxVariant

struct **mfxVariant**

The mfxVariantType enumerator data types for *mfxVariant* type.

#### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxVariantType* **Type**

Value type.

union *mfxVariant*::*data* **Data**

Value data member.

union **data**

Value data holder.

#### Public Members

*mfxU8* **U8**

mfxU8 data.

*mfxI8* **I8**

mfxI8 data.

*mfxU16* **U16**

mfxU16 data.

*mfxI16* **I16**

mfxI16 data.

*mfxU32* **U32**

mfxU32 data.

*mfxI32* **I32**

mfxI32 data.

*mfxU64* **U64**

mfxU64 data.

*mfxI64* **I64**

mfxI64 data.

*mfxF32* **F32**

mfxF32 data.

*mfxF64* **F64**

mfxF64 data.

mfxFP16 **FP16**

mfxFP16 data.

*mfxHDL* **Ptr**

Pointer. When this points to a string the string must be null terminated.

enum **mfxVariantType**

The mfxVariantType enumerator data types for mfxVariantType.

*Values:*

enumerator **MFX_VARIANT_TYPE_UNSET**

Undefined type.

enumerator **MFX_VARIANT_TYPE_U8**

8-bit unsigned integer.

enumerator **MFX_VARIANT_TYPE_I8**

8-bit signed integer.

enumerator **MFX_VARIANT_TYPE_U16**

16-bit unsigned integer.

enumerator **MFX_VARIANT_TYPE_I16**

16-bit signed integer.

enumerator **MFX_VARIANT_TYPE_U32**

32-bit unsigned integer.

enumerator **MFX_VARIANT_TYPE_I32**

32-bit signed integer.

enumerator **MFX_VARIANT_TYPE_U64**

64-bit unsigned integer.

enumerator **MFX_VARIANT_TYPE_I64**

64-bit signed integer.

enumerator **MFX_VARIANT_TYPE_F32**

32-bit single precision floating point.

enumerator **MFX_VARIANT_TYPE_F64**

64-bit double precision floating point.

enumerator **MFX_VARIANT_TYPE_PTR**

Generic type pointer.

enumerator **MFX_VARIANT_TYPE_FP16**

16-bit half precision floating point.

## mfxVPPDescription

struct **mfxVPPDescription**

This structure represents VPP description.

### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumFilters**

Number of supported VPP filters.

struct *mfxVPPDescription*::*filter* ***Filters**

Pointer to the array of supported filters.

struct **filter**

This structure represents the VPP filters description.

#### Public Members

*mfxU32* **FilterFourCC**

Filter ID in FourCC format.

*mfxU16* **MaxDelayInFrames**

Introduced output delay in frames.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumMemTypes**

Number of supported memory types.

struct *mfxVPPDescription*::*filter*::*memdesc* ***MemDesc**

Pointer to the array of memory types.

struct **memdesc**

This structure represents the underlying details of the memory type.

### Public Members

*mfxResourceType* **MemHandleType**

Memory handle type.

*mfxRange32U* **Width**

Range of supported image widths.

*mfxRange32U* **Height**

Range of supported image heights.

*mfxU16* **reserved**[7]

Reserved for future use.

*mfxU16* **NumInFormats**

Number of supported input color formats.

struct *mfxVPPDescription*::*filter*::*memdesc*::*format* ***Formats**

Pointer to the array of supported formats.

struct **format**

This structure represents the input color format description.

#### Public Members

*mfxU32* **InFormat**

Input color in FourCC format.

*mfxU16* **reserved**[5]

Reserved for future use.

*mfxU16* **NumOutFormat**

Number of supported output color formats.

*mfxU32* ***OutFormats**

Pointer to the array of supported output color formats (in FOURCC).

### mfxAccelerationModeDescription

struct **mfxAccelerationModeDescription**

This structure represents acceleration modes description.

#### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **reserved**[2]

reserved for future use.

*mfxU16* **NumAccelerationModes**

Number of supported acceleration modes.

*mfxAccelerationMode* ***Mode**

Pointer to the array of supported acceleration modes.

### mfxImplementedFunctions

struct **mfxImplementedFunctions**

This structure represents the list of names of implemented functions.

#### Public Members

*mfxU16* **NumFunctions**

Number of function names in the FunctionsName array.

*mfxChar* ****FunctionsName**

Array of the null-terminated strings. Each string contains name of the implemented function.

### mfxExtendedDeviceId

struct **mfxExtendedDeviceId**

Specifies various physical device properties for device matching and identification outside of oneAPI Video Processing Library (oneVPL).

### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **VendorID**

PCI vendor ID.

*mfxU16* **DeviceID**

PCI device ID.

*mfxU32* **PCIDomain**

PCI bus domain. Equals to '0' if OS doesn't support it or has sequential numbering of buses across domains.

*mfxU32* **PCIBus**

The number of the bus that the physical device is located on.

*mfxU32* **PCIDevice**

The index of the physical device on the bus.

*mfxU32* **PCIFunction**

The function number of the device on the physical device.

*mfxU8* **DeviceLUID**[8]

LUID of DXGI adapter.

*mfxU32* **LUIDDeviceNodeMask**

Bitfield identifying the node within a linked device adapter corresponding to the device.

*mfxU32* **LUIDValid**

Boolean value that will be 1 if DeviceLUID contains a valid LUID and LUIDDeviceNodeMask contains a valid node mask, and 0 if they do not.

*mfxU32* **DRMRenderNodeNum**

Number of the DRM render node from the path /dev/dri/RenderD<num>. Value equals to 0 means that this field doesn't contain valid DRM Render Node number.

*mfxU32* **DRMPrimaryNodeNum**

Number of the DRM primary node from the path /dev/dri/card<num>. Value equals to 0x7FFFFFFF means that this field doesn't contain valid DRM Primary Node number.

*mfxU16* **RevisionID**

PCI revision ID. The value contains microarchitecture version.

*mfxU8* **reserved1**[18]

Reserved for future use.

*mfxChar* **DeviceName**[MFX_STRFIELD_LEN]

Null-terminated string in utf-8 with the name of the device.

## mfxPoolPolicyDescription

struct **mfxPoolPolicyDescription**

This structure represents pool policy description.

### Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **reserved**[2]

reserved for future use.

*mfxU16* **NumPoolPolicies**

Number of supported pool policies.

*mfxPoolAllocationPolicy* ***Policy**

Pointer to the array of supported pool policies.

## extDeviceUUID

struct **extDeviceUUID**

Cross domain structure to define device UUID. It is defined here to check backward compatibility.

### Public Members

*mfxU16* **vendor_id**

PCI vendor ID. Same as *mfxExtendedDeviceId::VendorID*.

*mfxU16* **device_id**

PCI device ID. Same as *mfxExtendedDeviceId::DeviceID*.

*mfxU16* **revision_id**

PCI revision ID. Same as *mfxExtendedDeviceId::RevisionID*.

*mfxU16* **pci_domain**

PCI bus domain. Same as *mfxExtendedDeviceId::PCIDomain*.

*mfxU8* **pci_bus**

The number of the bus that the physical device is located on. Same as *mfxExtendedDeviceId::PCIBus*.

*mfxU8* **pci_dev**

The index of the physical device on the bus. Same as *mfxExtendedDeviceId::PCIDevice*.

*mfxU8* **pci_func**

The function number of the device on the physical device. Same as *mfxExtendedDeviceId::PCIFunction*.

*mfxU8* **reserved**[4]

Reserved for future use.

*mfxU8* **sub_device_id**

SubDevice ID.

## 5.6.3 Dispatcher API Enumeration Reference

### API

- *mfxAccelerationMode*
- *mfxImplType*
- *mfxAutoSelectImplType*

#### mfxAccelerationMode

enum **mfxAccelerationMode**

This enum itemizes hardware acceleration stack to use.

*Values:*

enumerator **MFX_ACCEL_MODE_NA**

Hardware acceleration is not applicable.

enumerator **MFX_ACCEL_MODE_VIA_D3D9**

Hardware acceleration goes through the Microsoft* Direct3D9* infrastructure.

enumerator **MFX_ACCEL_MODE_VIA_D3D11**

Hardware acceleration goes through the Microsoft* Direct3D11* infrastructure.

enumerator **MFX_ACCEL_MODE_VIA_VAAPI**

Hardware acceleration goes through the Linux* VA-API infrastructure.

enumerator **MFX_ACCEL_MODE_VIA_VAAPI_DRM_RENDER_NODE**

Hardware acceleration goes through the Linux* VA-API infrastructure with DRM RENDER MODE as default acceleration access point.

enumerator **MFX_ACCEL_MODE_VIA_VAAPI_DRM_MODESET**

Hardware acceleration goes through the Linux* VA-API infrastructure with DRM MODESET as default acceleration access point.

enumerator **MFX_ACCEL_MODE_VIA_VAAPI_GLX**

enumerator **MFX_ACCEL_MODE_VIA_VAAPI_X11**

Hardware acceleration goes through the Linux* VA-API infrastructure with OpenGL Extension to the X Window System as default acceleration access point. Hardware acceleration goes through the Linux* VA-API infrastructure with X11 as default acceleration access point.

enumerator **MFX_ACCEL_MODE_VIA_VAAPI_WAYLAND**

Hardware acceleration goes through the Linux* VA-API infrastructure with Wayland as default acceleration access point.

enumerator **MFX_ACCEL_MODE_VIA_HDDLUNITE**

Hardware acceleration goes through the HDDL* Unite*.

### mfxImplType

enum **mfxImplType**

This enum itemizes implementation type.

*Values:*

enumerator **MFX_IMPL_TYPE_SOFTWARE**

Pure Software Implementation.

enumerator **MFX_IMPL_TYPE_HARDWARE**

Hardware Accelerated Implementation.

### mfxAutoSelectImplType

enum **mfxAutoSelectImplType**

*Values:*

enumerator **MFX_AUTO_SELECT_IMPL_TYPE_UNKNOWN**

Unspecified automatic implementation selection.

enumerator **MFX_AUTO_SELECT_IMPL_TYPE_DEVICE_HANDLE**

Select implementation corresponding to device handle.

### 5.6.4 Dispatcher API Define Reference

#### API



#### MFX_IMPL_NAME_LEN

**`MFX_IMPL_NAME_LEN`**

Maximum allowed length of the implementation name.

#### MFX_STRFIELD_LEN

**`MFX_STRFIELD_LEN`**

Maximum allowed length of the implementation name.

Helper macro definitions to add property with single value.

#### MFX_ADD_PROPERTY_U32

**`MFX_ADD_PROPERTY_U32`**(loader, name, value)

Adds single property of mfxU32 type.

**Parameters**

- **`loader`** – **[in]** Valid mfxLoader object
- **`name`** – **[in]** Property name string
- **`value`** – **[in]** Property value

### MFX_ADD_PROPERTY_U16

**MFX_ADD_PROPERTY_U16**(loader, name, value)

Adds single property of mfxU16 type.

**Parameters**

- **loader** – **[in]** Valid mfxLoader object
- **name** – **[in]** Property name string
- **value** – **[in]** Property value

### MFX_ADD_PROPERTY_PTR

**MFX_ADD_PROPERTY_PTR**(loader, name, value)

Adds single property of pointer type.

**Parameters**

- **loader** – **[in]** Valid mfxLoader object
- **name** – **[in]** Property name string
- **value** – **[in]** Property value

Helper macro definitions to update existing property.

### MFX_UPDATE_PROPERTY_U32

**MFX_UPDATE_PROPERTY_U32**(loader, config, name, value)

Update existing property of mfxU32 type.

**Parameters**

- **loader** – **[in]** Valid mfxLoader object
- **config** – **[in]** Valid mfxConfig object
- **name** – **[in]** Property name string
- **value** – **[in]** Property value

### MFX_UPDATE_PROPERTY_U16

**MFX_UPDATE_PROPERTY_U16**(loader, config, name, value)

Update existing property of mfxU16 type.

**Parameters**

- **loader** – **[in]** Valid mfxLoader object
- **config** – **[in]** Valid mfxConfig object
- **name** – **[in]** Property name string
- **value** – **[in]** Property value

### MFX_UPDATE_PROPERTY_PTR

**MFX_UPDATE_PROPERTY_PTR**(loader, config, name, value)

Update existing property of pointer type.

**Parameters**

- **loader** – **[in]** Valid mfxLoader object
- **config** – **[in]** Valid mfxConfig object
- **name** – **[in]** Property name string
- **value** – **[in]** Property value

## 5.7 GUIDs Reference

### 5.7.1 API

static const *mfxGUID* **MFX_GUID_SURFACE_POOL** = {{0x35, 0x24, 0xf3, 0xda, 0x96, 0x4e, 0x47, 0xf1, 0xaf, 0xb4, 0xec, 0xb1, 0x15, 0x08, 0x06, 0xb1}}

GUID to obtain *mfxSurfacePoolInterface*.

CHAPTER

SIX

# ONEVPL API VERSIONING

oneVPL is the successor to Intel® Media Software Development Kit. oneVPL API versioning starts from 2.0. There is a correspondent version of Intel® Media Software Development Kit API which is used as a basis for oneVPL and defined as the `MFX_LEGACY_VERSION` macro.

CHAPTER

SEVEN

# ONEVPL EXPERIMENTAL API

All API entries defined under the ONEVPL_EXPERIMENTAL macro are considered as experimental. Backward compatibility is not guaranteed for these features. Future presence is not guaranteed as well.

By default, experimental API is turned off in the header files. To enable it, need to define ONEVPL_EXPERIMENTAL macro during the application compilation stage.

The following is a list of experimental interfaces, starting from API version 2.6.

Table 1: Experimental API

| Experimental API | Added in API Version | Removed in API V |
|---|---|---|
| *mfxExtendedDeviceId* | 2.6 | 2.10 |
| mfxExtCodingOption3::CPUEncToolsProcessing | 2.6 | 2.10 |
| *mfxExtRefListCtrl* | 2.6 | 2.8 |
| *MFX_EXTBUFF_UNIVERSAL_REFLIST_CTRL* | 2.6 | 2.8 |
| Extended enum for *mfxExtDecodeErrorReport::ErrorTypes* | 2.6 | 2.7 |
| *mfxHandleType::MFX_HANDLE_PXP_CONTEXT* | 2.6 | 2.7 |
| *mfxRefInterface* | 2.7 | 2.10 |
| All definitions in mfxencodestats.h | 2.7 | |
| *MFX_FOURCC_ABGR16F* FourCC definition | 2.8 | 2.10 |
| *MFX_CONTENT_NOISY_VIDEO* ContentInfo definition | 2.8 | 2.10 |
| Camera Processing API for RAW acceleration | 2.8 | 2.10 |
| Hint to disable external video frames caching for GPU copy | 2.8 | 2.10 |
| *mfxExtMBQP::Pitch* | 2.8 | 2.10 |
| *mfxExtSyncSubmission* | 2.9 | |
| *mfxExtVPPPercEncPrefilter* | 2.9 | |
| *mfxExtendedDeviceId::RevisionID* | 2.9 | 2.10 |
| *extDeviceUUID* | 2.9 | 2.10 |
| *mfxExtTuneEncodeQuality* | 2.9 | |
| MFX_ENCODE_TUNE_DEFAULT | 2.9 | 2.10 |
| *MFX_ENCODE_TUNE_PSNR* | 2.9 | |
| *MFX_ENCODE_TUNE_SSIM* | 2.9 | |
| *MFX_ENCODE_TUNE_MS_SSIM* | 2.9 | |
| *MFX_ENCODE_TUNE_VMAF* | 2.9 | |
| *MFX_ENCODE_TUNE_PERCEPTUAL* | 2.9 | |
| *MFX_EXTBUFF_TUNE_ENCODE_QUALITY* | 2.9 | |
| *mfxAutoSelectImplDeviceHandle* | 2.9 | |
| *mfxAutoSelectImplType* | 2.9 | |
| *mfxAutoSelectImplType::MFX_AUTO_SELECT_IMPL_TYPE_UNKNOWN* | 2.9 | |
| *mfxAutoSelectImplType::MFX_AUTO_SELECT_IMPL_TYPE_DEVICE_HANDLE* | 2.9 | |
| *MFX_CORRUPTION_HW_RESET* | 2.10 | |

Table 1 – continued from previous page

| Experimental API | Added in API Version | Removed in API V |
|---|---|---|
| MFX_ENCODE_TUNE_OFF | 2.10 | |
| mfxMemoryInterface | 2.10 | |
| mfxHandleType::MFX_HANDLE_MEMORY_INTERFACE | 2.10 | |
| mfxSurfaceComponent | 2.10 | |
| mfxSurfaceType | 2.10 | |
| mfxSurfaceHeader | 2.10 | |
| mfxSurfaceInterface | 2.10 | |
| mfxSurfaceD3D11Tex2D | 2.10 | |
| mfxSurfaceVAAPI | 2.10 | |
| mfxSurfaceOpenCLImg2D | 2.10 | |
| mfxExtSurfaceOpenCLImg2DExportDescription | 2.10 | |
| mfxImplCapsDeliveryFormat | 2.10 | |
| mfxSurfaceTypesSupported | 2.10 | |
| mfxConfigInterface | 2.10 | |
| mfxHandleType::MFX_HANDLE_CONFIG_INTERFACE | 2.10 | |
| mfxStructureType | 2.10 | |
| mfxStatus::MFX_ERR_MORE_EXTBUFFER | 2.10 | |

CHAPTER

EIGHT

# APPENDICES

## 8.1 Configuration Parameter Constraints

The *mfxFrameInfo* structure is used by both the *mfxVideoParam* structure during oneVPL class initialization and the *mfxFrameSurface1* structure during the actual oneVPL class operation. The parameter constraints described in the following tables apply.

### 8.1.1 DECODE, ENCODE, and VPP Constraints

The *DECODE, ENCODE, and VPP Constraints table* lists parameter constraints common to *DECODE*, *ENCODE*, and *VPP*.

Table 1: DECODE, ENCODE, and VPP Constraints

| Parameters | Use During Initialization | Use During Operation |
|---|---|---|
| FourCC | Any valid value. | The value must be the same as the initialization value. The only exception is *VPP* in composition mode, where in some cases it is allowed to mix RGB and NV12 surfaces. See *mfxExtVPPComposite* for more details. |
| ChromaFormat | Any valid value. | The value must be the same as the initialization value. |

### 8.1.2 DECODE Constraints

The *DECODE Constraints table* lists *DECODE* parameter constraints.

Table 2: DECODE Constraints

| Parameters | Use During Initialization | Use During Operation |
|---|---|---|
| Width, Height | Aligned frame size. | The values must be the equal to or larger than the initialization values. |
| CropX, CropY<br>CropW, CropH | Ignored. | *DECODE* output. The cropping values are per-frame based. |
| AspectRatioW, AspectRatioH | Any valid values or unspecified (zero); if unspecified, values from the input bitstream will be used. See note below the table. | DECODE output. |
| FrameRateExtN, FrameRateExtD | If unspecified, values from the input bitstream will be used. See note below the table. | DECODE output. |
| PicStruct | Ignored. | DECODE output. |

**Note:** If the application explicitly sets FrameRateExtN/FrameRateExtD or AspectRatioW/AspectRatioH during initialization, then the decoder will use these values during decoding regardless of the values from bitstream and does not update them on new SPS. If the application sets them to 0, then the decoder uses values from the stream and updates them on each SPS.

### 8.1.3 ENCODE Constraints

The *ENCODE Constraints table* lists *ENCODE* parameter constraints.

Table 3: ENCODE Constraints

| Parameters | Use During Initialization | Use During Operation |
|---|---|---|
| Width, Height | Encoded frame size. | The values must be the equal to or larger than the initialization values. |
| CropX, CropY<br>CropW, CropH | H.264: Cropped frame size<br>MPEG-2: CropW and CropH<br><br>Specify the real width and height (may be unaligned) of the coded frames. CropX and CropY must be zero. | Ignored. |
| AspectRatioW, AspectRatioH | Any valid values. | Ignored. |
| FrameRateExtN, FrameRateExtD | Any valid values. | Ignored. |
| PicStruct | *MFX_PICSTRUCT_UNKNOWN*<br>*MFX_PICSTRUCT_PROGRESSIVE*<br>*MFX_PICSTRUCT_FIELD_TFF*<br>*MFX_PICSTRUCT_FIELD_BFF* | The base value must be the same as the initialization value unless *MFX_PICSTRUCT_UNKNOWN* is specified during initialization. Add other decorative picture structure flags to indicate additional display attributes. Use *MFX_PICSTRUCT_UNKNOWN* during initialization for field attributes and *MFX_PICSTRUCT_PROGRESSIVE* for frame attributes. See the *PicStruct* enumerator for details. |

### 8.1.4 VPP Constraints

The *VPP Constraints table* lists *VPP* parameter constraints.

Table 4: VPP Constraints

| Parameters | During Initialization | During Operation |
|---|---|---|
| Width, Height | Any valid values | The values must be the equal to or larger than the initialization values. |
| CropX, CropY, CropW, CropH | Ignored | These parameters specify the region of interest from input to output. |
| AspectRatioW, AspectRatioH | Ignored | Aspect ratio values will be passed through from input to output. |
| FrameRateExtN, FrameRateExtD | Any valid values | Frame rate values will be updated with the initialization value at output. |
| PicStruct | *MFX_PICSTRUCT_UNKNOWN* *MFX_PICSTRUCT_PROGRESSIVE* *MFX_PICSTRUCT_FIELD_TFF* *MFX_PICSTRUCT_FIELD_BFF* *MFX_PICSTRUCT_FIELD_SINGLE* *MFX_PICSTRUCT_FIELD_TOP* *MFX_PICSTRUCT_FIELD_BOTTOM* | The base value must be the same as the initialization value unless *MFX_PICSTRUCT_UNKNOWN* is specified during initialization. Other decorative picture structure flags are passed through or added as needed. See the *PicStruct* enumerator for details. |

### 8.1.5 Specifying Configuration Parameters

The following *Configuration Parameters tables* summarize how to specify the configuration parameters during initialization, encoding, decoding, and video processing.

Table 5: mfxVideoParam Configuration Parameters

| Structure (param) | ENCODE Init | ENCODE Encoding | DECODE Init | DECODE Decoding | VPP Init | VPP Processing |
|---|---|---|---|---|---|---|
| Protected | R | • | R | • | R | • |
| IOPattern | M | • | M | • | M | • |
| ExtParam | O | • | O | • | O | • |
| NumExtParam | O | • | O | • | O | • |

Table 6: mfxInfoMFX Configuration Parameters

| Structure (param) | ENCODE Init | ENCODE Encoding | DECODE Init | DECODE Decoding | VPP Init | VPP Processing |
|---|---|---|---|---|---|---|
| CodecId | M | • | M | • | • | • |
| CodecProfile | O | • | O/M* | • | • | • |
| CodecLevel | O | • | O | • | • | • |
| NumThread | O | • | O | • | • | • |
| TargetUsage | O | • | • | • | • | • |
| GopPicSize | O | • | • | • | • | • |
| GopRefDist | O | • | • | • | • | • |
| GopOptFlag | O | • | • | • | • | • |
| IdrInterval | O | • | • | • | • | • |
| RateControl-Method | O | • | • | • | • | • |
| InitialDelayInKB | O | • | • | • | • | • |
| BufferSizeInKB | O | • | • | • | • | • |
| TargetKbps | M | • | • | • | • | • |
| MaxKbps | O | • | • | • | • | • |
| NumSlice | O | • | • | • | • | • |
| NumRefFrame | O | • | • | • | • | • |
| EncodedOrder | M | • | • | • | • | • |

Table 7: mfxFrameInfo Configuration Parameters

| Structure (param) | ENCODE Init | ENCODE Encoding | DECODE Init | DECODE Decoding | VPP Init | VPP Processing |
|---|---|---|---|---|---|---|
| FourCC | M | M | M | M | M | M |
| Width | M | M | M | M | M | M |
| Height | M | M | M | M | M | M |
| CropX | M | Ign | Ign | U | Ign | M |
| CropY | M | Ign | Ign | U | Ign | M |
| CropW | M | Ign | Ign | U | Ign | M |
| CropH | M | Ign | Ign | U | Ign | M |
| FrameRateExtN | M | Ign | O | U | M | U |
| FrameRateExtD | M | Ign | O | U | M | U |
| AspectRatioW | O | Ign | O | U | Ign | PT |
| AspectRatioH | O | Ign | O | U | Ign | PT |
| PicStruct | O | M | Ign | U | M | M/U |
| ChromaFormat | M | M | M | M | Ign | Ign |

Table 8: Abbreviations used in configuration parameter tables

| Abbreviation | Meaning |
|---|---|
| Ign | Ignored |
| PT | Pass Through |
| • | Does Not Apply |
| M | Mandated |
| R | Reserved |
| O | Optional |
| U | Updated at output |

**Note:** *CodecProfile* is mandated for HEVC REXT and SCC profiles and optional for other cases. If the application does not explicitly set CodecProfile during initialization, the HEVC decoder will use a profile up to Main10.

## 8.2 Multiple-segment Encoding

Multiple-segment encoding is useful in video editing applications during production, for example when the encoder encodes multiple video clips according to their time line. In general, one can define multiple-segment encoding as dividing an input sequence of frames into segments and encoding them in different encoding sessions with the same or different parameter sets. For example:

| Segment Already Encoded | Segment in Encoding | Segment to be Encoded |
|---|---|---|
| 0s | 200s | 500s |

**Note:** Different encoders can also be used.

The application must be able to:

- Extract encoding parameters from the bitstream of previously encoded segment.
- Import these encoding parameters to configure the encoder.

Encoding can then continue on the current segment using either the same or similar encoding parameters.

Extracting the header that contains the encoding parameter set from the encoded bitstream is usually the task of a format splitter (de-multiplexer). Alternatively, the *MFXVideoDECODE_DecodeHeader()* function can export the raw header if the application attaches the *mfxExtCodingOptionSPSPPS* structure as part of the parameters.

The encoder can use the *mfxExtCodingOptionSPSPPS* structure to import the encoding parameters during *MFXVideoENCODE_Init()*. The encoding parameters are in the encoded bitstream format. Upon a successful import of the header parameters, the encoder will generate bitstreams with a compatible (not necessarily bit-exact) header. The *Header Import Functions table* shows all functions that can import a header and their error codes if there are unsupported parameters in the header or the encoder is unable to achieve compatibility with the imported header.

Table 9: Header Import Functions

| Function Name | Error Code if Import Fails |
|---|---|
| *MFXVideoENCODE_Init()* | *MFX_ERR_INCOMPATIBLE_VIDEO_PARAM* |
| *MFXVideoENCODE_QueryIOSurf()* | *MFX_ERR_INCOMPATIBLE_VIDEO_PARAM* |
| *MFXVideoENCODE_Reset()* | *MFX_ERR_INCOMPATIBLE_VIDEO_PARAM* |
| *MFXVideoENCODE_Query()* | *MFX_ERR_UNSUPPORTED* |

The encoder must encode frames to a GOP sequence starting with an IDR frame for H.264 (or I frame for MPEG-2) to ensure that the current segment encoding does not refer to any frames in the previous segment. This ensures that the encoded segment is self-contained, allowing the application to insert the segment anywhere in the final bitstream. After encoding, each encoded segment is HRD compliant. Concatenated segments may not be HRD compliant.

The following example shows the encoder initialization procedure that imports H.264 sequence and picture parameter sets:

```
mfxStatus init_encoder() {
   mfxExtCodingOptionSPSPPS option, *option_array;

   /* configure mfxExtCodingOptionSPSPPS */
   memset(&option,0,sizeof(option));
   option.Header.BufferId=MFX_EXTBUFF_CODING_OPTION_SPSPPS;
   option.Header.BufferSz=sizeof(option);
   option.SPSBuffer=sps_buffer;
   option.SPSBufSize=sps_buffer_length;
   option.PPSBuffer=pps_buffer;
   option.PPSBufSize=pps_buffer_length;

   /* configure mfxVideoParam */
   mfxVideoParam param;
   //...
   param.NumExtParam=1;
   option_array=&option;
   param.ExtParam=(mfxExtBuffer**)&option_array;
```

(continues on next page)

(continued from previous page)

```

   /* encoder initialization */
   mfxStatus status;
   status=MFXVideoENCODE_Init(session, &param);
   if (status==MFX_ERR_INCOMPATIBLE_VIDEO_PARAM) {
      printf("Initialization failed.\n");
   } else {
      printf("Initialized.\n");
   }
   return status;
}
```

## 8.3 Streaming and Video Conferencing Features

The following sections address some aspects of additional requirements that streaming or video conferencing applications may use in the encoding or transcoding process. See the *Configuration Change* section for additional information.

### 8.3.1 Dynamic Bitrate Change

The oneVPL encoder supports dynamic bitrate change according to bitrate control mode and HRD conformance requirements. If HRD conformance is required, for example if the application sets the `NalHrdConformance` option in the *mfxExtCodingOption* structure to ON, the only allowed bitrate control mode is VBR. In this mode, the application can change the `TargetKbps` and `MaxKbps` values of the *mfxInfoMFX* structure by calling the *MFXVideoENCODE_Reset()* function. This sort of change in bitrate usually results in the generation of a new keyframe and sequence header. There are exceptions, such as if HRD information is absent in the stream. In this scenario, the change of `TargetKbps` does not require a change in the sequence header and as a result the encoder does not insert a keyframe.

If HRD conformance is not required, for example if the application turns off the `NalHrdConformance` option in the *mfxExtCodingOption* structure, all bitrate control modes are available. In CBR and AVBR modes the application can change `TargetKbps`. In VBR mode the application can change `TargetKbps` and `MaxKbps` values. This sort of change in bitrate will not result in the generation of a new keyframe or sequence header.

The oneVPL encoder may change some initialization parameters provided by the application during initialization. That in turn may lead to incompatibility between the parameters provided by the application during reset and the working set of parameters used by the encoder. For this reason, it is strongly recommended to retrieve the actual working parameters using the *MFXVideoENCODE_GetVideoParam()* function before making any changes to bitrate settings.

In all modes, oneVPL encoders will respond to the bitrate changes as quickly as the underlying algorithm allows, without breaking other encoding restrictions such as HRD compliance if it is enabled. How quickly the actual bitrate can catch up with the specified bitrate is implementation dependent.

Alternatively, the application may use the *CQP* encoding mode to perform customized bitrate adjustment on a per-frame base. The application may use any of the encoded or display order modes to use per-frame CQP.

### 8.3.2 Dynamic Resolution Change

The oneVPL encoder supports dynamic resolution change in all bitrate control modes. The application may change resolution by calling the *MFXVideoENCODE_Reset()* function. The application may decrease or increase resolution up to the size specified during encoder initialization.

Resolution change always results in the insertion of a key IDR frame and a new sequence parameter set in the header. The only exception is the oneVPL VP9 encoder (see section for *Dynamic reference frame scaling*). The oneVPL encoder does not guarantee HRD conformance across the resolution change point.

The oneVPL encoder may change some initialization parameters provided by the application during initialization. That in turn may lead to incompatibility of parameters provide by the application during reset and working set of parameters used by the encoder. Due to this potential incompatibility, it is strongly recommended to retrieve the actual working parameters set by *MFXVideoENCODE_GetVideoParam()* function before making any resolution change.

### 8.3.3 Dynamic Reference Frame Scaling

The VP9 standard allows changing the resolution without the insertion of a keyframe. This is possible because the VP9 encoder has the built-in capability to upscale and downscale reference frames to match the resolution of the frame being encoded. By default the oneVPL VP9 encoder inserts a keyframe when the application does *Dynamic Resolution Change*. In this case, the first frame with a new resolution is encoded using inter prediction from the scaled reference frame of the previous resolution. Dynamic scaling has the following limitations, described in the VP9 specification:

- The resolution of any active reference frame cannot exceed 2x the resolution of the current frame.
- The resolution of any active reference frame cannot be smaller than 1/16 of the current frame resolution.

In the case of dynamic scaling, the oneVPL VP9 encoder always uses a single active reference frame for the first frame after a resolution change. The VP9 encoder has the following limitations for dynamic resolution change:

- The new resolution should not exceed 16x the resolution of the current frame.
- The new resolution should be less than 1/2 of current frame resolution.

The application may force insertion of a keyframe at the point of resolution change by invoking encoder reset with *mfxExtEncoderResetOption::StartNewSequence* set to *MFX_CODINGOPTION_ON*. If a keyframe is inserted, the dynamic resolution limitations are not enforced.

Note that resolution change with dynamic reference scaling is compatible with multiref (*mfxInfoMFX::NumRefFrame* > 1). For multiref configuration, the oneVPL VP9 encoder uses multiple references within stream pieces of the same resolution and uses a single reference at the place of resolution change.

### 8.3.4 Forced Keyframe Generation

oneVPL supports forced keyframe generation during encoding. The application can set the FrameType parameter of the *mfxEncodeCtrl* structure to control how the current frame is encoded, as follows:

- If the oneVPL encoder works in the display order, the application can enforce any current frame to be a keyframe. The application cannot change the frame type of already buffered frames inside the encoder.
- If the oneVPL encoder works in the encoded order, the application must specify exact frame type for every frame. In this way, the application can enforce the current frame to have any frame type that the particular coding standard allows.

### 8.3.5 Reference List Selection

During streaming or video conferencing, if the application can obtain feedback about how well the client receives certain frames, the application may need to adjust the encoding process to use or not use certain frames as reference. This section describes how to fine-tune the encoding process based on client feedback.

The application can specify the reference window size by specifying the *mfxInfoMFX::NumRefFrame* parameter during encoding initialization. Certain platforms may have limits on the the size of the reference window. Use the *MFXVideoENCODE_GetVideoParam()* function to retrieve the current working set of parameters.

During encoding, the application can specify the actual reference list lengths by attaching the *mfxExtAVCRefListCtrl* structure to the *MFXVideoENCODE_EncodeFrameAsync()* function. `NumRefIdxL0Active` specifies the length of the reference list L0 and `NumRefIdxL1Active` specifies the length of the reference list L1. These two numbers must be less than or equal to the *mfxInfoMFX::NumRefFrame* parameter during encoding initialization.

The application can instruct the oneVPL encoder to use or not use certain reference frames. To do this, there is a prerequisite that the application uniquely identify each input frame by setting the *mfxFrameData::FrameOrder* parameter. The application then specifies the preferred reference frame list `PreferredRefList` and/or the rejected frame list `RejectedRefList`, and attaches the *mfxExtAVCRefListCtrl* structure to the *MFXVideoENCODE_EncodeFrameAsync()* function. The two lists fine-tune how the encoder chooses the reference frames for the current frame. The encoder does not keep `PreferredRefList` and the application must send it for each frame if necessary. There are limitations as follows:

- The frames in the lists are ignored if they are out of the reference window.
- If by going through the lists, the oneVPL encoder cannot find a reference frame for the current frame, the encoder will encode the current frame without using any reference frames.
- If the GOP pattern contains B-frames, the oneVPL encoder may not be able to follow the *mfxExtAVCRefListCtrl* instructions.

### 8.3.6 Low Latency Encoding and Decoding

The application can set *mfxVideoParam::AsyncDepth* = 1 to disable any decoder buffering of output frames, which is aimed to improve the transcoding throughput. With *mfxVideoParam::AsyncDepth* = 1, the application must synchronize after the decoding or transcoding operation of each frame.

The application can adjust *mfxExtCodingOption::MaxDecFrameBuffering* during encoding initialization to improve decoding latency. It is recommended to set this value equal to the number of reference frames.

### 8.3.7 Reference Picture Marking Repetition SEI Message

The application can request writing the reference picture marking repetition SEI message during encoding initialization by setting `RefPicMarkRep` of the *mfxExtCodingOption* structure. The reference picture marking repetition SEI message repeats certain reference frame information in the output bitstream for robust streaming.

The oneVPL decoder will respond to the reference picture marking repetition SEI message if the message exists in the bitstream and compare it to the reference list information specified in the sequence/picture headers. The decoder will report any mismatch of the SEI message with the reference list information in the *mfxFrameData::Corrupted* field.

### 8.3.8 Long Term Reference Frame

The application may use long term reference frames to improve coding efficiency or robustness for video conferencing applications. The application controls the long term frame marking process by attaching the *mfxExtAVCRefListCtrl* extended buffer during encoding. The oneVPL encoder itself never marks a frame as long term.

There are two control lists in the *mfxExtAVCRefListCtrl* extended buffer. The `LongTermRefList` list contains the frame orders (the `FrameOrder` value in the *mfxFrameData* structure) of the frames that should be marked as long term frames. The `RejectedRefList` list contains the frame order of the frames that should be unmarked as long term frames. The application can only mark or unmark the frames that are buffered inside the encoder. Because of this, it is recommended that the application marks a frame when it is submitted for encoding. The application can either explicitly unmark long term reference frames or wait for the IDR frame. When the IDR frame is reached, all long term reference frames will be unmarked.

The oneVPL encoder puts all long term reference frames at the end of a reference frame list. If the number of active reference frames (the `NumRefIdxL0Active` and `NumRefIdxL1Active` values in the *mfxExtAVCRefListCtrl* extended buffer) is less than than the total reference frame number (the `NumRefFrame` value in the *mfxInfoMFX* structure during the encoding initialization), the encoder may ignore some or all long term reference frames. The application may avoid this by providing a list of preferred reference frames in the `PreferredRefList` list in the *mfxExtAVCRefListCtrl* extended buffer. In this case, the encoder reorders the reference list based on the specified list.

### 8.3.9 Temporal Scalability

The application may specify the temporal hierarchy of frames by using the *mfxExtAvcTemporalLayers* extended buffer during the encoder initialization in the display order encoding mode. oneVPL inserts the prefix NAL unit before each slice with a unique temporal and priority ID. The temporal ID starts from zero and the priority ID starts from the `BaseLayerPID` value. oneVPL increases the temporal ID and priority ID value by one for each consecutive layer.

If the application needs to specify a unique sequence or picture parameter set ID, the application must use the *mfxExtCodingOptionSPSPPS* extended buffer, with all pointers and sizes set to zero and valid `SPSId` and `PPSId` fields. The same SPS and PPS ID will be used for all temporal layers.

Each temporal layer is a set of frames with the same temporal ID. Each layer is defined by the `Scale` value. The scale for layer N is equal to the ratio between the frame rate of subsequent temporal layers with a temporal ID less than or equal to N and the frame rate of the base temporal layer. The application may skip some temporal layers by specifying the `Scale` value as zero. The application should use an integer ratio of the frame rates for two consecutive temporal layers.

For example, a video sequence with 30 frames/second is typically separated by three temporal layers that can be decoded as 7.5 fps (base layer), 15 fps (base and first temporal layer) and 30 fps (all three layers). In this scenario, `Scale` should have the values {1,2,4,0,0,0,0,0}.

## 8.4 Switchable Graphics and Multiple Monitors

The following sections discuss support for switchable graphics and multiple monitor configurations.

### 8.4.1 Switchable Graphics

Switchable Graphics refers to the machine configuration that multiple graphic devices are available (integrated device for power saving and discrete devices for performance.) Usually at one time or instance, one of the graphic devices drives display and becomes the active device, and others become inactive. There are different variations of software or hardware mechanisms to switch between the graphic devices. In one of the switchable graphics variations, it is possible to register an application in an affinity list to certain graphic device so that the launch of the application automatically triggers a switch. The actual techniques to enable such a switch are outside the scope of this document. This section discusses the implication of switchable graphics to Intel® Media Software Development Kit and Intel® Media Software Development Kit applications.

As Intel® Media Software Development Kit performs hardware acceleration through graphic devices, it is critical that Intel® Media Software Development Kit can access the graphic device in the switchable graphics setting. It is recommended to add the application to the graphic device affinity list. If this is not possible, the application should handle the following cases:

- By design, during legacy Intel® Media Software Development Kit library initialization, the `MFXInit()` function searches for graphic devices. If a Intel® Media Software Development Kit implementation is successfully loaded, the `MFXInit()` function returns `mfxStatus::MFX_ERR_NONE` and the `MFXQueryIMPL()` function returns the actual implementation type. If no Intel® Media Software Development Kit implementation is loaded, the `MFXInit()` function returns `mfxStatus::MFX_ERR_UNSUPPORTED`. In the switchable graphics environment, if the application is not in the graphic device affinity list, it is possible that the graphic device will not be accessible during the library initialization. The fact that the `MFXInit()` function returns `mfxStatus::MFX_ERR_UNSUPPORTED` does not mean that hardware acceleration is permanently impossible. The user may switch the graphics later and the graphic device will become accessible. It is recommended that the application initialize the library right before the actual decoding, video processing, and encoding operations to determine the hardware acceleration capability.
- During decoding, video processing, and encoding operations, if the application is not in the graphic device affinity list, the previously accessible graphic device may become inaccessible due to a switch event. The Intel® Media Software Development Kit functions will return `mfxStatus::MFX_ERR_DEVICE_LOST` or `mfxStatus::MFX_ERR_DEVICE_FAILED`, depending on when the switch occurs and what stage the Intel® Media Software Development Kit functions operate. The application should handle these errors and exit gracefully.

### 8.4.2 Multiple Monitors

Multiple monitors refer to the machine configuration that multiple graphic devices are available. Some graphic devices connect to a display and become active and accessible under the Microsoft* DirectX* infrastructure. Graphic devices that are not connected to a display are inactive. Using the Microsoft Direct3D* 9 infrastructure, devices that are not connected to a display are not accessible.

The legacy Intel® Media Software Development Kit uses the adapter number to access a specific graphic device. Usually, the graphic device driving the main desktop becomes the primary adapter. Other graphic devices take subsequent adapter numbers after the primary adapter. Under the Microsoft Direct3D 9 infrastructure, only active adapters are accessible and have an adapter number.

Intel® Media Software Development Kit extends the `mfxIMPL` implementation type as shown in the *Intel® Media SDK mfxIMPL Implementation Type Definitions table*:

Table 10: Intel® Media SDK mfxIMPL Implementation Type Definitions

| Implementation Type | Definition |
|---|---|
| *MFX_IMPL_HARDWARE* | Intel® Media Software Development Kit should initialize on the primary adapter |
| *MFX_IMPL_HARDWARE2* | Intel® Media Software Development Kit should initialize on the 2nd graphic adapter |
| *MFX_IMPL_HARDWARE3* | Intel® Media Software Development Kit should initialize on the 3rd graphic adapter |
| *MFX_IMPL_HARDWARE4* | Intel® Media Software Development Kit should initialize on the 4th graphic adapter |
| *MFX_IMPL_HARDWARE_ANY* | Intel® Media Software Development Kit should initialize on any graphic adapter. |
| *MFX_IMPL_AUTO_ANY* | Intel® Media Software Development Kit should initialize on any graphic adapter. If not successful, load the software implementation. |

The application can use the first four definitions shown in the *Intel® Media SDK mfxIMPL Implementation Type Definitions table* to instruct the legacy Intel® Media Software Development Kit library to initialize on a specific graphic device. The application can use the definitions for *MFX_IMPL_HARDWARE_ANY* and *MFX_IMPL_AUTO_ANY* for automatic detection.

If the application uses the Microsoft DirectX surfaces for I/O, it is critical that the application and Intel® Media Software Development Kit work on the same graphic device. It is recommended that the application use the following procedure:

1. The application uses the *MFXInit()* function to initialize the legacy Intel® Media Software Development Kit, with option *MFX_IMPL_HARDWARE_ANY* or *MFX_IMPL_AUTO_ANY*. The *MFXInit()* function returns *mfxStatus::MFX_ERR_NONE* if successful.
2. The application uses the *MFXQueryIMPL()* function to check the actual implementation type. The implementation type *MFX_IMPL_HARDWARE*, *MFX_IMPL_HARDWARE2*, *MFX_IMPL_HARDWARE3*, or *MFX_IMPL_HARDWARE4* indicates the graphic adapter the Intel® Media Software Development Kit works on.
3. The application creates the Direct3D device on the respective graphic adapter and passes it to Intel® Media Software Development Kit through the *MFXVideoCORE_SetHandle()* function.

Similar to the switchable graphics cases, interruption may result if the user disconnects monitors from the graphic devices or remaps the primary adapter. If the interruption occurs during the Intel® Media Software Development Kit library initialization, the *MFXInit()* function may return *mfxStatus::MFX_ERR_UNSUPPORTED*. This means hardware acceleration is currently not available. It is recommended that the application initialize Intel® Media Software Development Kit right before the actual decoding, video processing, and encoding operations to determine the hardware acceleration capability.

If the interruption occurs during decoding, video processing, or encoding operations, oneVPL functions will return *mfxStatus::MFX_ERR_DEVICE_LOST* or *mfxStatus::MFX_ERR_DEVICE_FAILED*. The application should handle these errors and exit gracefully.

## 8.5 Working Directly with VA API for Linux*

Intel® Media Software Development Kit takes care of all memory and synchronization related operations in the VA API. The application may need to extend Intel® Media Software Development Kit functionality by working directly with the VA API for Linux*, for example to implement a customized external allocator. This section describes basic memory management and synchronization techniques.

To create the VA surface pool, the application should call the vaCreateSurfaces function:

```
const int num_surfaces = 5;
VASurfaceID surfaces[num_surfaces];
VASurfaceAttrib attrib;

attrib.type = VASurfaceAttribPixelFormat;
attrib.value.type = VAGenericValueTypeInteger;
attrib.value.value.i = VA_FOURCC_NV12;
attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;

vaCreateSurfaces(va_display, VA_RT_FORMAT_YUV420, width, height,
                 surfaces, num_surfaces, &attrib, 1);
```

To destroy the surface pool, the application should call the vaDestroySurfaces function:

```
vaDestroySurfaces(va_display, surfaces, num_surfaces);
```

If the application works with hardware acceleration through Intel® Media Software Development Kit, then it can access surface data immediately after successful completion of the *MFXVideoCORE_SyncOperation()* call. If the application works with hardware acceleration directly, then it must check surface status before accessing data in video memory. This check can be done asynchronously by calling the vaQuerySurfaceStatus function or synchronously by calling the vaSyncSurface function.

After successful synchronization, the application can access surface data. Accessing surface data is performed in two steps:

1. Create VAImage from surface.
2. Map image buffer to system memory.

After mapping, the VAImage.offsets[3] array holds offsets to each color plain in a mapped buffer and the VAImage.pitches[3] array holds color plain pitches in bytes. For packed data formats, only first entries in these arrays are valid. The following example shows how to access data in a NV12 surface:

```
VAImage image;
unsigned char *Y, *U, *V;
void* buffer;

vaDeriveImage(va_display, surfaceToMap, &image);
vaMapBuffer(va_display, image.buf, &buffer);

/* NV12 */
Y = (unsigned char*)buffer + image.offsets[0];
U = (unsigned char*)buffer + image.offsets[1];
V = U + 1;
```

After processing data in a VA surface, the application should release resources allocated for the mapped buffer and VAImage object:

```
vaUnmapBuffer(va_display, image.buf);
vaDestroyImage(va_display, image.image_id);
```

In some cases, in order to retrieve encoded bitstream data from video memory, the application must use the VABuffer to store data. The following example shows how to create, use, and destroy the VABuffer:

```
VABufferID buf_id;
size_t size;
uint32_t offset;
void *buf;

/* create buffer */
vaCreateBuffer(va_display, va_context, VAEncCodedBufferType, buf_size, 1, NULL, & buf_
↪id);

/* encode frame */
// ...

/* map buffer */
VACodedBufferSegment *coded_buffer_segment;

vaMapBuffer(va_display, buf_id, (void **)(&coded_buffer_segment));

size   = coded_buffer_segment->size;
offset = coded_buffer_segment->bit_offset;
buf    = coded_buffer_segment->buf;

/* retrieve encoded data*/
// ...

/* unmap and destroy buffer */
vaUnmapBuffer(va_display, buf_id);
vaDestroyBuffer(va_display, buf_id);
```

Note that the vaMapBuffer function returns pointers to different objects depending on the mapped buffer type. The VAImage is a plain data buffer and the encoded bitstream is a VACodedBufferSegment structure. The application cannot use VABuffer for synchronization. If encoding, it is recommended to synchronize using the VA surface as described above.

## 8.6 CQP HRD Mode Encoding

The application can configure an AVC encoder to work in CQP rate control mode with HRD model parameters. oneVPL will place HRD information to SPS/VUI and choose the appropriate profile/level. It's the responsibility of the application to provide per-frame QP, track HRD conformance, and insert required SEI messages to the bitstream.

The following example shows how to enable CQP HRD mode. The application should set *RateControlMethod* to CQP, `mfxExtCodingOption::VuiNalHrdParameters` to ON, `mfxExtCodingOption::NalHrdConformance` to OFF, and set rate control parameters similar to CBR or VBR modes (instead of QPI, QPP, and QPB). oneVPL will choose CBR or VBR HRD mode based on the `MaxKbps` parameter. If `MaxKbps` is set to zero, oneVPL will use CBR HRD model (write cbr_flag = 1 to VUI), otherwise the VBR model will be used (and cbr_flag = 0 is written to VUI).

**Note:** For CQP, if implementation does not support individual QPI, QPP and QPB parameters, then QPI parameter should be used as a QP parameter across all frames.

```
mfxExtCodingOption option, *option_array;

/* configure mfxExtCodingOption */
memset(&option,0,sizeof(option));
option.Header.BufferId         = MFX_EXTBUFF_CODING_OPTION;
option.Header.BufferSz         = sizeof(option);
option.VuiNalHrdParameters     = MFX_CODINGOPTION_ON;
option.NalHrdConformance       = MFX_CODINGOPTION_OFF;

/* configure mfxVideoParam */
mfxVideoParam param;

// ...

param.mfx.RateControlMethod         = MFX_RATECONTROL_CQP;
param.mfx.FrameInfo.FrameRateExtN   = valid_non_zero_value;
param.mfx.FrameInfo.FrameRateExtD   = valid_non_zero_value;
param.mfx.BufferSizeInKB            = valid_non_zero_value;
param.mfx.InitialDelayInKB          = valid_non_zero_value;
param.mfx.TargetKbps                = valid_non_zero_value;

if (write_cbr_flag == 1)
   param.mfx.MaxKbps = 0;
else /* write_cbr_flag = 0 */
   param.mfx.MaxKbps = valid_non_zero_value;

param.NumExtParam = 1;
option_array     = &option;
param.ExtParam     = (mfxExtBuffer **)&option_array;

/* encoder initialization */
mfxStatus sts;
sts = MFXVideoENCODE_Init(session, &param);

// ...

/* encoding */
mfxEncodeCtrl ctrl;
memset(&ctrl,0,sizeof(ctrl));
ctrl.QP = frame_qp;

sts=MFXVideoENCODE_EncodeFrameAsync(session,&ctrl,surface2,bits,&syncp);
```

CHAPTER

# NINE

# GLOSSARY

The oneVPL API and documentation uses a standard set of acronyms and terms. This section describes these conventions.



## 9.1 Acronyms and Terms

**AVC**
Advanced video codec (same as H.264 and MPEG-4, part 10).

**BRC**
Bit rate control.

**CQP**
Constant quantization parameter.

**DRM**
Digital rights management.

**DXVA2**
Microsoft DirectX* Video Acceleration standard 2.0.

**GOP**
Group of pictures. In video coding, a group of frames in a specific order. In the H.264 standard, a group of I-frames, B-frames and P-frames.

**GPB**
Generalized P/B picture. B-picture, containing only forward references in both L0 and L1.

**H.264**
Video coding standard. See ISO*/IEC* 14496-10 and ITU-T* H.264, MPEG-4 Part 10, Advanced Video Coding, May 2005.

**HDR**
High dynamic range.

**HRD**
Hypothetical reference decoder, a term used in the H.264 specification.

**IDR**
Instantaneous decoding fresh picture, a term used in the H.264 specification.

**LA**
Look ahead. Special encoding mode where encoder performs pre-analysis of several frames before actual encoding starts.

**MCTF**
Motion compensated temporal filter. Special type of noise reduction filter which utilizes motion to improve efficiency of video denoising.

**NAL**
Network abstraction layer.

**PPS**
Picture parameter set.

**QP**
Quantization parameter.

**SEI**
Supplemental enhancement information.

**SPS**
Sequence parameter set.

**VA API**
Video acceleration API.

**VBR**
Variable bit rate.

**VBV**
Video buffering verifier.

**Video memory**
Memory used by a hardware acceleration device, also known as GPU, to hold frame and other types of video data.

**VUI**
Video usability information.

## 9.2 Video Formats

**MPEG**
Moving Picture Experts Group video file.

**MPEG-2**
Moving Picture Experts Group video file. See ISO/IEC 13818-2 and ITU-T H.262, MPEG-2 Part 2, Information Technology- Generic Coding of Moving Pictures and Associate Audio Information: Video, 2000.

**NV12**
YUV 4:2:0 video format, 12 bits per pixel.

**NV16**
YUV 4:2:2 video format, 16 bits per pixel.

**P010**
YUV 4:2:0 video format, extends NV12, 10 bits per pixel.

**P210**
YUV 4:2:2 video format, 10 bits per pixel.

**UYVY**
YUV 4:2:2 video format, 16 bits per pixel.

**VC-1**
Video coding format. See SMPTE* 421M, SMPTE Standard for Television: VC-1 Compressed Video Bitstream Format and Decoding Process, August 2005.

## 9.3 Color Formats

**I010**
Color format for raw video frames, extends IYUV/I420 for 10 bit.

**IYUV**
A color format for raw video frames, also known as I420.

**RGB32**
Thirty-two-bit RGB color format.

**RGB4**
Thirty-two-bit RGB color format. Also known as RGB32.

**YUY2**
A color format for raw video frames.

**YV12**
A color format for raw video frames, similar to IYUV with U and V reversed.

CHAPTER

# TEN

# DEPRECATED API

The following is a list of deprecated interfaces, starting from API version 2.0.

Table 1: Deprecated AP

| API | Deprecated in API Version | Removed in API Version | Alternatives |
|---|---|---|---|
| *MFXQueryAdapters()* | 2.9 | | *MFXEnumImple* |
| *MFXQueryAdaptersDecode()* | 2.9 | | *MFXEnumImple* |
| *MFXQueryAdaptersNumber()* | 2.9 | | *MFXEnumImple* |
| *mfxExtCodingOption2::BitrateLimit* | 2.9 | | Flag is ignored |
| MFX_PLATFORM_UNKNOWN | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_SANDYBRIDGE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_IVYBRIDGE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_HASWELL | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_BAYTRAIL | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_BROADWELL | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_CHERRYTRAIL | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_SKYLAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_APOLLOLAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_KABYLAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_GEMINILAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_COFFEELAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_CANNONLAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_ICELAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_JASPERLAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_ELKHARTLAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_TIGERLAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_ROCKETLAKE | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_ALDERLAKE_S | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_ALDERLAKE_P | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_ARCTICSOUND_P | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_XEHP_SDV | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_DG2 | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_ATS_M | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_ALDERLAKE_N | 2.9 | | Field *mfxPlat* |
| MFX_PLATFORM_KEEMBAY | 2.9 | | Field *mfxPlat* |
| *mfxPlatform::CodeName* | 2.9 | | Field is not fille |
| *MFXInit()* | 2.3 | | *MFXLoad()* + *M* |
| *MFXInitEx()* | 2.3 | | *MFXLoad()* + *M* |
| *mfxExtVPPDenoise* | 2.5 | | Use *mfxExtVP* |
| MFX_FOURCC_RGB3 | 2.0 | | Use *MFX_FOURC* |

Table 1 – continued from prev

| API | Deprecated in API Version | Removed in API Version | Alternatives |
|---|---|---|---|
| mfxExtCodingOption::EndOfSequence | 2.0 | | Flag is ignored |
| mfxExtCodingOption::EndOfStream | 2.0 | | Flag is ignored |
| mfxExtCodingOption3::ExtBrcAdaptiveLTR | 2.4 | | Use *mfxExtCo* |
| MFX_EXTBUFF_VPP_SCENE_CHANGE | 2.0 | | Ignored |
| mfxExtVppAuxData::SpatialComplexity | 2.0 | | Field is not fille |
| mfxExtVppAuxData::TemporalComplexity | 2.0 | | Field is not fille |
| mfxExtVppAuxData::SceneChangeRate | 2.0 | | Field is not fille |

CHAPTER

# ELEVEN

# CHANGE LOG

This section describes the API evolution from version to version.



## 11.1 Version 2.10

New in this release:

- Experimental API: introduced `MFX_CORRUPTION_HW_RESET` to support media reset info report.
- Changed `MFX_ENCODE_TUNE_DEFAULT` to `MFX_ENCODE_TUNE_OFF`.
- Experimental API: Removed CPUEncToolsProcessing hint. No need to have explicit parameter. The decision to enable encoding tools will be made according to encoding parameters.
- Extended behavior of fused decode plus VPP operation to disable implicit scaling.
- Added alias `mfxExtEncodedFrameInfo` as codec-independent version of `mfxExtAVCEncodedFrameInfo`.
- Updated description of `MFXSetConfigFilterProperty()` to permit multiple properties per config object.
- Fixed 3DLUT buffer size(system memory) in programming guide.
- Clarified Region of Interest Parameters Setting for dynamic change.
- Removed current working directory from the implementation search path.
- Updated argument names and description of MFX_UUID_COMPUTE_DEVICE_ID macro.
- Added new header file mfxmemory.h, which is automatically included by mfxvideo.h. Moved the following function declarations from mfxvideo.h to mfxmemory.h
  - `MFXMemory_GetSurfaceForEncode()`

  - *MFXMemory_GetSurfaceForDecode()*
  - *MFXMemory_GetSurfaceForVPP()*
  - *MFXMemory_GetSurfaceForVPPOut()*

- Experimental API: Introduced new interface for importing and exporting surfaces. Added new function *mfxFrameSurfaceInterface::Export*. Added new structures and enumerated types:
  - *mfxMemoryInterface*
  - *mfxHandleType::MFX_HANDLE_MEMORY_INTERFACE*
  - *mfxSurfaceComponent*
  - *mfxSurfaceType*
  - *mfxSurfaceHeader*
  - *mfxSurfaceInterface*
  - *mfxSurfaceD3D11Tex2D*
  - *mfxSurfaceVAAPI*
  - *mfxSurfaceOpenCLImg2D*
  - *mfxExtSurfaceOpenCLImg2DExportDescription*
- Experimental API: Introduced capabilities query for supported surface import and export operations. Added new structures and enumerated types:
  - *mfxImplCapsDeliveryFormat*
  - *mfxSurfaceTypesSupported*
- Experimental API: Introduced new interface for configuring initialization parameters. Added new structures and enumerated types:
  - *mfxConfigInterface*
  - *mfxHandleType::MFX_HANDLE_CONFIG_INTERFACE*
  - *mfxStructureType*
  - *mfxStatus::MFX_ERR_MORE_EXTBUFFER*
- Experimental API: previously released experimental features were moved to production . See *Experimental API* for more details.
- Not supported in the encoding of VDEnc or LowPower ON:
  - `CodecProfile::MFX_PROFILE_AVC_MULTIVIEW_HIGH`
  - `CodecProfile::MFX_PROFILE_AVC_STEREO_HIGH`

## 11.2 Version 2.9

New in this release:

- Deprecated *mfxExtCodingOption2::BitrateLimit*.
- Added note that applications must call MFXVideoENCODE_Query() to check for support of *mfxExtChromaLocInfo* and *mfxExtHEVCRegion* extension buffers.
- Added AV1 HDR metadata description and further clarified *mfxExtMasteringDisplayColourVolume* and *mfxExtContentLightLevelInfo*.
- Added deprecation messages to the functions *MFXQueryAdapters()*, *MFXQueryAdaptersDecode()*, and *MFXQueryAdaptersNumber()*. Applications should use the process described in *oneVPL Dispatcher* to enumerate and select adapters.
- Fixed multiple spelling errors.
- Added extension buffer *mfxExtSyncSubmission* to return submission synchronization sync point.
- Added extension buffer *mfxExtVPPPercEncPrefilter* to control perceptual encoding prefilter.
- Deprecated *mfxPlatform::CodeName* and corresponding enum values.
- Added *mfxExtendedDeviceId::RevisionID* and *extDeviceUUID* to be aligned across multiple domains including compute and specify device UUID accordingly.
- Added extension buffer *mfxExtTuneEncodeQuality* and correspondent enumeration to specify encoding tuning option.
- Updated description of *MFXEnumImplementations()* to clarify that the input *mfxImplCapsDeliveryFormat* determines the type of structure returned.
- Updated mfxvideo++.h to use MFXLoad API.
- Added *mfxAutoSelectImplDeviceHandle* and *mfxAutoSelectImplType* for automatically selecting a suitable implementation based on application-provided device handle.

## 11.3 Version 2.8

New in this release:

- Introduced *MFX_FOURCC_ABGR16F* FourCC for 16-bit float point (per channel) 4:4:4 ABGR format.
- Clarified the *mfxExtMasteringDisplayColourVolume::DisplayPrimariesX*, *mfxExtMasteringDisplayColourVolume::DisplayPrimariesY* for the video processing usage.
- Added *MFX_CONTENT_NOISY_VIDEO* in ContentInfo definition.
- Added Camera Processing API for Camera RAW data.
- Introduced hint to disable external video frames caching for GPU copy.
- Clarified usage of *mfxExtMasteringDisplayColourVolume::InsertPayloadToggle* and *mfxExtContentLightLevelInfo::InsertPayloadToggle* during decode operations.
- Fixed multiple spelling errors.
- Experimental API: introduced *mfxExtMBQP::Pitch* value for QP map defined in *mfxExtMBQP*.
- Clarified when *MFXEnumImplementations()* may be called for implementation capabilities query.
- Added table with filenames included in the dispatcher's search process.

Bug Fixes:

- Fixed *Experimental API table* to note that `mfxExtRefListCtrl` and `MFX_EXTBUFF_UNIVERSAL_REFLIST_CTRL` were moved to production in version 2.8.

## 11.4 Version 2.7

New in this release:

- `mfxExtVppAuxData::RepeatedFrame` flag is actual again and returned back from deprecation state.
- Clarified GPUCopy control behavior.
- Introduced MFX_FOURCC_XYUV FourCC for non-alpha packed 4:4:4 format.
- Notice added to the `mfxFrameSurfaceInterface::OnComplete` to clarify when library can call this callback.
- New product names for platforms:
  - Code name Alder Lake N.
- Annotated missed aliases `mfxExtHEVCRefListCtrl`, `mfxExtHEVCRefLists`, `mfxExtHEVCTemporalLayers`.
- New dispatcher's config properties:
  - Pass through extension buffer to `mfxInitializationParam`.
  - Select host or device responsible for the memory copy between host and device.
- Refined description of struct *mfxExtMasteringDisplayColourVolume* and *mfxExtContentLightLevelInfo* for HDR SEI decoder usage.
- Experimental API: introduced interface to get statistics after encode.

Bug Fixes:

- Fixed missprint in the `mfxExtDeviceAffinityMask` description.
- MFXVideoENCODE_Query description fixed for query mode 1.

## 11.5 Version 2.6

New in this release:

- New development practice to treat some new API features as experimental was introduced. All new experimental API is wrapped with ONE_EXPERIMENTAL macro.
- Experimental API: introduced MFX_HANDLE_PXP_CONTEXT to support protected content.
- Experimental API: introduced CPUEncToolsProcessing hint to run adaptive encoding tools on CPU.
- Experimental API: extended device ID reporting to cover multi-adapter cases.
- Experimental API: introduced common alias for mfxExtAVCRefListCtrl
- Experimental API: mfxExtDecodeErrorReport ErrorTypes enum extended with new JPEG/MJPEG decode error report.
- Clarified LowPower flag meaning.
- Described that mfxExtThreadsParam can be attached to mfxInitializationParam during session initialization.
- Refined description of the MFXVideoDECODE_VPP_DecodeFrameAsync function.

- New dispatcher's config filter property: MediaAdapterType.
- Marked all deprecated fields as MFX_DEPRECATED.
- Introduced priority loading option for custom libraries.
- Clarified AV1 encoder behavior about writing of IVF headers.
- Removed outdated note about loading priority of Intel® Media Software Development Kit. For loading details see *oneVPL implementation on Intel® platforms with Xe architecture and Intel® Media Software Development Kit Coexistence*.
- Spelled out mfxVariant type usage for strings.
- New product names for platforms:
  - Code name DG2,
  - Code name ATS-M.

## 11.6 Version 2.5

New in this release:

- Added mfxMediaAdapterType to capability reporting.
- Added surface pool interface.
- Helper macro definition to simplify filter properties set up process for dispatcher.
- Added mfxExtAV1BitstreamParam, mfxExtAV1ResolutionParam and mfxExtAV1TileParam for AV1e.
- Added MFX_RESOURCE_VA_SURFACE_PTR and MFX_RESOURCE_VA_BUFFER_PTR enumerators.
- Clarified HEVC Main 10 Still Picture Profile configuration.
- External Buffer ID of mfxExtVideoSignalInfo and mfxExtMasteringDisplayColourVolume for video processing.
- New MFX_WRN_ALLOC_TIMEOUT_EXPIRED return status. Indicates that all surfaces are currently in use and timeout set by mfxExtAllocationHints for allocation of new surfaces through functions GetSurfaceForXXX expired.
- Introduced universal temporal layering structure.
- Added MFX_RESOURCE_VA_SURFACE_PTR and MFX_RESOURCE_VA_BUFFER_PTR enumerators.
- Introduced segmentation interface for AV1e, including ext-buffers and enums.
- Introduced planar I422 and I210 FourCC codes.

Bug Fixes:

- Dispatcher: Removed /etc/ld.so.cache from oneVPL search order.
- mfxSurfaceArray: CDECL attribute added to the member-functions.

Deprecated:

- mfxExtVPPDenoise extension buffer.

## 11.7 Version 2.4

- Added ability to retrieve path to the shared library with the implementation.
- Added 3DLUT (Three-Dimensional Look Up Table) filter in VPP.
- Added mfxGUID structure to specify Globally Unique Identifiers (GUIDs).
- Added QueryInterface function to mfxFrameSurfaceInterface.
- Added AdaptiveRef and alias for ExtBrcAdaptiveLTR.
- Added MFX_FOURCC_BGRP FourCC for Planar BGR format.
- Environmental variables to control dispatcher's logger.

## 11.8 Version 2.3

- Encoding in Hyper mode.
- New product names for platforms:
  - Code name Rocket Lake,
  - Code name Alder Lake S,
  - Code name Alder Lake P,
  - Code name for Arctic Sound P.
  - For spec version 2.3.1 MFX_PLATFORM_XEHP_SDV alias was added
- mfx.h header file is added which includes all header files.
- Added deprecation messages (deprecation macro) to the functions MFXInit and MFXInitEx functions definition.

# INDEX

## Symbols



## A



## B



## C



## D



## E



## G



## H