Intel® Media Software Development Kit

Reference Manual

API Version 1.10

LEGAL DISCLAIMER

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

MPEG is an international standard for video compression/decompression promoted by ISO. Implementations of MPEG CODECs, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2007-2014, Intel Corporation. All Rights reserved.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Table of Contents

Overview

The Intel® Media SDK (Software Development Kit) is a software development library that exposes the media acceleration capabilities of Intel platforms for decoding, encoding and video processing. The API library covers a wide range of Intel platforms.

The Intel® Media SDK includes simple console samples, media framework components and GUI playback and transcoding applications for hands-on experience.

This document describes the Intel® Media SDK API, abbreviated as SDK API.

Document Conventions

The Intel® Media SDK API uses the Verdana typeface for normal prose. With the exception of section headings and the table of contents, all code-related items appear in the `Courier New` typeface (`mxfStatus` and `MFXInit`). All class-related items appear in all cap boldface, such as **DECODE** and **ENCODE**. Member functions appear in initial cap boldface, such as **Init** and **Reset**, and these refer to members of all three classes, **DECODE**, **ENCODE** and **VPP**. Hyperlinks appear in underlined boldface, such as **mfxStatus**.

Acronyms and Abbreviations

API	Application Programming Interface
AVC	Advanced Video Codec (same as H.264 and MPEG-4, part 10)
Direct3D	Microsoft* Direct3D* version 9 or 11.1
Direct3D9	Microsoft* Direct3D* version 9
Direct3D11	Microsoft* Direct3D* version 11.1
DXVA2	Microsoft DirectX* Video Acceleration standard 2.0
H.264	ISO*/IEC* 14496-10 and ITU-T* H.264, MPEG-4 Part 10, Advanced Video Coding, May 2005
HRD	Hypothetical Reference Decoder
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.
MPEG	Motion Picture Expert Group
MPEG-2	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
NAL	Network Abstraction Layer

NV12	A color format for raw video frames
PPS	Picture Parameter Set
QP	Quantization Parameter
RGB3	Twenty-four-bit RGB color format. Also known as RGB24
RGB4	Thirty-two-bit RGB color format. Also known as RGB32
SDK	Intel® Media SDK
SDK decoder	Intel® Media SDK decoder
SDK execution	Intel® Media SDK execution
SDK functions	Intel® Media SDK functions
SDK library	Intel® Media SDK library
SDK session	Intel® Media SDK session
SDK video processing	Intel® Media SDK video processing
SEI	Supplemental Enhancement Information
SPS	Sequence Parameter Set
VA API	Video Acceleration API
VBR	Variable Bit Rate
VBV	Video Buffering Verifier
VC-1	SMPTE* 421M, SMPTE Standard for Television: VC-1 Compressed Video Bitstream Format and Decoding Process, August 2005
video memory	memory used by hardware acceleration device, also known as GPU, to hold frame and other types of video data
VPP	Video Processing
VUI	Video Usability Information
YUY2	A color format for raw video frames
YV12	A color format for raw video frames

Architecture

Intel® Media SDK functions fall into the following categories:

DECODE	Decode compressed video streams into raw video frames
ENCODE	Encode raw video frames into compressed bitstreams
VPP	Perform video processing on raw video frames
CORE	Auxiliary functions for synchronization
Misc	Global auxiliary functions

With the exception of the global auxiliary functions, SDK functions are named after their functioning domain and category, as illustrated in Figure 1. Here, Intel® Media SDK only exposes video domain functions.

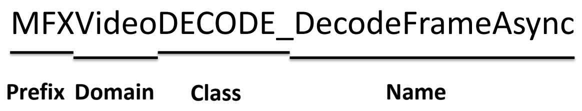

Figure 1: Intel® Media SDK Function Naming Convention

Applications use Intel® Media SDK functions by linking with the Intel® Media SDK dispatcher library, as illustrated in Figure 2. The dispatcher library identifies the hardware acceleration device on the running platform, determines the most suitable platform library, and then redirects function calls. If the dispatcher is unable to detect any suitable platform-specific hardware, the dispatcher redirects SDK function calls to the default software library.

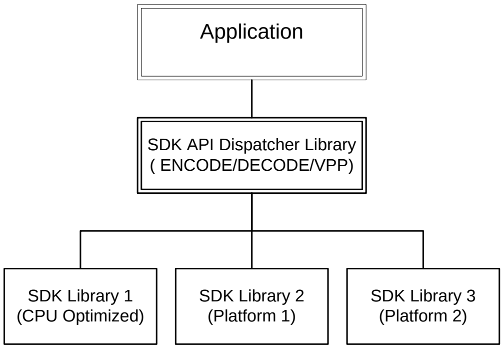

Figure 2: Intel® Media SDK Library Dispatching Mechanism

Video Decoding

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

DECODE processes only pure or elementary video streams. The library cannot process bitstreams that reside in a container format, such as MP4 or MPEG. The application must first de-multiplex the bitstreams. De-multiplexing extracts pure video streams out of the container format. The application can provide the input bitstream as one complete frame of data, less than one frame (a partial frame), or 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. That is, the first byte of a video coding layer NAL unit for H.264, or picture header for MPEG-2 and VC-1. **DECODE** passes the time stamp to the output surface for audio and video multiplexing or synchronization.

Decoding the first frame is a special case, since **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 that 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.

Video Encoding

The **ENCODE** class of functions takes 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 from 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 key frames: 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:

> <u>Display order</u>: **ENCODE** receives input frames in the display order. A few GOP structure parameters specify the GOP sequence during **ENCODE** initialization. Scene change results from the video processing stage of a pipeline can alter the GOP sequence.
>
> <u>Encoded order</u>: **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. The Intel® Media SDK library 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 (or VBR) requirements. (Stuffing is a process that appends zeros to the end of encoded frames.)

Video Processing

Video processing (**VPP**) takes raw frames as input and provides raw frames as output.

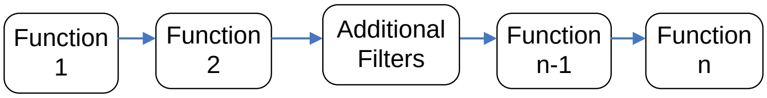

Figure 3: Video Processing Operation Pipeline

The actual conversion process is a chain operation with many single-function filters, as Figure 3 illustrates. The application specifies the input and output format, and the Intel® Media SDK configures the pipeline accordingly. The application can also attach one or more hint structures to configure individual filters or turn them on and off. Unless specifically instructed, the Intel® Media SDK builds the pipeline in a way that best utilizes hardware acceleration or generates the best video processing quality.

Table 1 shows the Intel® Media SDK 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 procedure / Configuration"* 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 **(See** Table 2 for supported conversions)	I/O parameters
De-interlace to produce progressive frames at the output (See The SDK video processing pipeline supports limited functionality for RGB4 input. Only filters that are required to convert input format to output one are included in pipeline. All optional filters are skipped. See description of `MFX_WRN_FILTER_SKIPPED` warning for more details on how to retrieve list of active filters. Table 3 for supported conversions)	I/O parameters
Crop and resize the input frames to meet the output resolution and region of display	I/O parameters
Convert input frame rate to match the output	I/O parameters
Perform inverse telecine operations	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)

Table 2: Color Conversion Support in VPP

Output Color / Input Color	NV12	RGB32
RGB4 (RGB32)	X limited	X limited
NV12	X	X
YV12	X	X
YUY2	X	X

X indicates a supported function

The SDK video processing pipeline supports limited functionality for RGB4 input. Only filters that are required to convert input format to output one are included in pipeline. All optional filters are skipped. See description of MFX_WRN_FILTER_SKIPPED warning for more details on how to retrieve list of active filters.

Table 3: Deinterlacing/Inverse Telecine Support in VPP

Input Field Rate (fps) Interlaced	Output Frame Rate (fps) Progressive						
	23.976	25	29.97	30	50	59.94	60
29.97	Inverse Telecine		X				
50		X			X		
59.94			X			X	
60				X			X

X indicates a supported function.

This table describes pure deinterlacing algorithm. The application can combine it with frame rate conversion to achieve any desirable input/output frame rate ratio. Note, that in this table input rate is field rate, i.e. number of video fields in one second of video. The SDK uses frame rate in all configuration parameters, so this input field rate should be divided by two during the SDK configuration. For example, 60i to 60p conversion in this table is represented by right bottom cell. It should be described in mfxVideoParam as input frame rate equal to 30 and output 60.

Programming Guide

This chapter describes the concepts used in programming the Intel® Media SDK.

The application must use the include file, **`mfxvideo.h`** (for C programming), or **`mfxvideo++.h`** (for C++ programming), and link the Intel® Media SDK static dispatcher library, **`libmfx.lib`** or **`libmfx.a`**. If the application is written in C then **`libstdc++.a`** library should also be linked.

```
Include these files:
      #include "mfxvideo.h"     /* The SDK include file */
      #include "mfxvideo++.h"   /* Optional for C++ development */

Link this library:
      libmfx.lib                /* The SDK static dispatcher library */
or
      libmfx.a                  /* The SDK static dispatcher library */
```

Status Codes

The Intel® Media SDK functions organize into classes for easy reference. The classes 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 de-initialize specific operations defined for the class. Call all other member functions within a given class (except **Query** and **QueryIOSurf**) within the **Init** … **Reset** (optional) … **Close** sequence.

The **Init** and **Reset** member functions both set up necessary internal structures for media processing. The difference between the two is that the **Init** functions allocate memory while the **Reset** functions only reuse allocated internal memory. Therefore, **Reset** can fail if the Intel® Media SDK needs to allocate additional memory. **Reset** functions can also fine-tune **ENCODE** and **VPP** parameters during those processes or reposition a bitstream during **DECODE**.

All Intel® Media SDK functions return status codes to indicate whether an operation succeeded or failed. See the **`mfxStatus`** enumerator for all defined status codes. The status code **`MFX_ERR_NONE`** indicates that the function successfully completed its operation. Status codes are less than **`MFX_ERR_NONE`** for all errors and greater than **`MFX_ERR_NONE`** for all warnings.

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

If an SDK function returns an error (except **`MFX_ERR_MORE_DATA`** or **`MFX_ERR_MORE_SURFACE`** or **`MFX_ERR_MORE_BITSTREAM`**), the function aborts the operation. The application must call either the **Reset** function to put 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**.

SDK Session

Before calling any SDK functions, the application must initialize the SDK library and create an SDK session. An SDK session maintains context for the use of any of **DECODE**, **ENCODE**, or **VPP** functions.

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

The application can initialize a session as a software-based session (`MFX_IMPL_SOFTWARE`) or a hardware-based session (`MFX_IMPL_HARDARE`,). In the former case, the SDK functions execute on a CPU, and in the latter case, the SDK functions use platform acceleration capabilities. For platforms that expose multiple graphic devices, the application can initialize the SDK session on any alternative graphic device (`MFX_IMPL_HARDWARE1`…`MFX_IMPL_HARDWARE4`).

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 Intel® Media SDK returns the actual implementation through the `MFXQueryIMPL` function.

Multiple Sessions

Each SDK session can run exactly one instance of **DECODE**, **ENCODE** and **VPP** functions. This is good for a simple transcoding operation. If the application needs more than one instance of **DECODE**, **ENCODE** and **VPP** in a complex transcoding setting, or needs more simultaneous transcoding operations to balance CPU/GPU workloads, the application can initialize multiple SDK sessions. Each SDK session can independently be a software-based session or hardware-based session.

The application can use multiple SDK sessions independently or run a "joined" session. Independently operated SDK sessions cannot share data unless the application explicitly synchronizes session operations (to ensure that data is valid and complete before passing from the source to the destination session.)

To join two sessions together, the application can use the function `MFXJoinSession`. Alternatively, the application can use the function `MFXCloneSession` to duplicate an existing session. Joined SDK 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, one of the sessions (the first join) serves as a parent session, scheduling execution resources, with all others child sessions relying on the parent session for resource management.

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

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

Frame and Fields

In Intel® Media SDK terminology, a frame (or frame surface, interchangeably) 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.

Frame Surface Locking

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

SDK functions define a frame-locking mechanism to avoid the need for copy operations. This mechanism is as follows:

1. The application allocates a pool of frame surfaces large enough to include SDK function I/O frame surfaces and internal cache needs. Each frame surface maintains a `Locked` counter, part of the **mfxFrameData** structure. Initially, the `Locked` counter is set to zero.
2. The application calls an SDK function with frame surfaces from the pool, whose `Locked` counter is zero. If the SDK function needs to reserve any frame surface, the SDK 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." That is, the application can read, but cannot alter, move, delete or free the frame surface.
3. In subsequent SDK executions, if the frame surface is no longer in use, the SDK 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 must not increase or decrease the `Locked` counter, since the SDK manages this field. If, for some reason, the application needs to modify the `Locked` counter, the operation must be atomic to avoid race condition. **Modifying the `Locked` counter is not recommended.**

Decoding Procedures

Example 1 shows the pseudo code of the decoding procedure. The following describes a few key points:

- The application can use the **MFXVideoDECODE_DecodeHeader** function to retrieve decoding initialization parameters from the bitstream. This step is optional if such parameters are retrievable from other sources such as an audio/video splitter.
- The application uses the **MFXVideoDECODE_QueryIOSurf** function to obtain the number of working frame surfaces required to reorder output frames.
- The application calls the **MFXVideoDECODE_DecodeFrameAsync** function for a decoding operation, with the bitstream buffer (`bits`), and an unlocked working frame surface (`work`) as input parameters. If decoding output is not available, the function returns a status code requesting additional bitstream input or working frame surfaces as follows:

  MFX_ERR_MORE_DATA: The function needs additional bitstream input. The existing buffer contains less than a frame worth of bitstream data.

  MFX_ERR_MORE_SURFACE: The function needs one more frame surface to produce any output.

- Upon successful decoding, the **MFXVideoDECODE_DecodeFrameAsync** function returns **MFX_ERR_NONE**. However, the decoded frame data (identified by the `disp` pointer) is not yet available because the **MFXVideoDECODE_DecodeFrameAsync** function is asynchronous. The application must use the **MFXVideoCORE_SyncOperation** function 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 Intel® Media SDK decoder, until the function returns **MFX_ERR_MORE_DATA**.

Bitstream Repositioning

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

1. Use the **MFXVideoDECODE_Reset** function to reset the SDK decoder.
2. Optionally, if the application maintains a sequence header that decodes correctly 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 above steps, the SDK decoder searches for a new sequence header before starting decoding.

```
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())
            append_more_bitstream(bitstream);
      find_unlocked_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_bitstream() && sts==MFX_ERR_MORE_DATA) break;
      … // other error handling
      if (sts==MFX_ERR_NONE) {
            MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
            do_something_with_decoded_frame(disp);
      }
}
MFXVideoDECODE_Close();
free_pool_of_frame_surfaces();
```

Example 1: Decoding Pseudo Code

Multiple Sequence Headers

The bitstream can contain multiple sequence headers. The SDK function returns a status code to indicate when a new sequence header is parsed.

The **MFXVideoDECODE_DecodeFrameAsync** function returns **MFX_WRN_VIDEO_PARAM_CHANGED** if the SDK decoder parsed a new sequence header in the bitstream and decoding can continue with existing frame buffers. The application can optionally retrieve new video parameters by calling **MFXVideoDECODE_GetVideoParam**.

The **MFXVideoDECODE_DecodeFrameAsync** function returns **MFX_ERR_INCOMPATIBLE_VIDEO_PARAM** if the decoder parsed a new sequence header in the bitstream and decoding cannot continue without reallocating frame buffers. The bitstream pointer moves to the first bit of the new sequence header. The application must do the following:

1. Retrieve any remaining frames by calling **MFXVideoDECODE_DecodeFrameAsync** with a NULL input bitstream pointer until the function returns **MFX_ERR_MORE_DATA**. This step is not necessary if the application plans to discard any remaining frames.

2. De-initialize the decoder by calling the MFXVideoDECODE_Close function, and restart the decoding procedure from the new bitstream position.

Encoding Procedures

Example 2 shows the pseudo code of the encoding procedure. The following describes a few key points:

- 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_EncodedFrameAsync 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 status code MFX_ERR_MORE_DATA to request additional input frames.
- Upon successful encoding, the MFXVideoENCODE_EncodeFrameAsync function returns MFX_ERR_NONE. However, 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 NULL surface pointer to drain any remaining bitstreams cached within the SDK encoder, until the function returns MFX_ERR_MORE_DATA.

Configuration Change

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

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

The application uses the following procedure to change encoding configurations:

1. The application retrieves any cached frames in the SDK encoder by calling the **MFXVideoENCODE_EncodeFrameAsync** function with a `NULL` input frame pointer until the function returns **MFX_ERR_MORE_DATA**.

   Note: The application must set the initial encoding configuration flag `EndOfStream` of the **mfxExtCodingOption** structure to `OFF` to avoid inserting an End of Stream (EOS) marker into the bitstream. An EOS marker causes the bitstream to terminate before encoding is complete.

2. The application calls the **MFXVideoENCODE_Reset** function with the new configuration:
   - If the function successfully set the configuration, the application can continue encoding as usual.
   - If the new configuration requires a new memory allocation, the function returns **MFX_ERR_INCOMPATIBLE_VIDEO_PARAM**. The application must close the SDK encoder and reinitialize the encoding procedure with the new configuration.

```
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;
      … // other error handling
      if (sts==MFX_ERR_NONE) {
            MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
            do_something_with_encoded_bits(bits);
      }
}
MFXVideoENCODE_Close();
free_pool_of_frame_surfaces();
```

Example 2: Encoding Pseudo Code

Video Processing Procedures

Example 3 shows the pseudo code of the video processing procedure. The following describes a few key points:

- 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 the other for the output.
- The video processing function MFXVideoVPP_RunFrameVPPAsync is asynchronous. The application must synchronize to make the output result ready, through the MFXVideoCORE_SyncOperation function.
- The body of the video processing procedures covers three scenarios as follows:
    - If the number of frames consumed at input is equal to the number of frames generated at output, **VPP** returns MFX_ERR_NONE when an output is ready. The application must process the output frame after synchronization, as the MFXVideoVPP_RunFrameVPPAsync function is asynchronous. At the end of a sequence, the application must provide a NULL input 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 MFX_ERR_MORE_DATA for additional input until an output is ready. When the output is ready, **VPP** returns MFX_ERR_NONE. The application must process the output frame after synchronization and provide a NULL input at the end of 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 MFX_ERR_MORE_SURFACE (when more than one output is ready), or 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 sequence to drain any remaining frames.

```
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);
in=find_unlocked_surface_and_fill_content(in_pool);
out=find_unlocked_surface_from_the_pool(out_pool);
for (;;) {
      sts=MFXVideoVPP_RunFrameVPPAsync(session,in,out,aux,&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(in_pool);
            fill_content_for_video_processing(in);
            if (end_of_input_sequence()) in=NULL;
      }
}
MFXVideoVPP_Close(session);
free_pool_of_surfaces(in_pool);
free_pool_of_surfaces(out_pool);
```

Example 3: Video Processing Pseudo Code

Configuration

The Intel® Media SDK configures the video processing pipeline operation based on the difference between the input and output formats, specified in the **mfxVideoParam** structure. A few examples follow:

- When the input color format is YUY2 and the output color format is NV12, the SDK enables color conversion from YUY2 to NV12.
- When the input is interleaved and the output is progressive, the SDK enables de-interlacing.

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 pipeline by using **mfxExtVPPDoNotUse** structure; enable them by using **mfxExtVPPDoUse** structure and configure them by using dedicated configuration structures. See Table 4 for complete list of configurable video processing filters, their IDs and configuration structures. See the **ExtendedBufferID** enumerator for more details.

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

Table 4 Configurable VPP filters

Filter ID	Configuration structure
MFX_EXTBUFF_VPP_DENOISE	**mfxExtVPPDenoise**
MFX_EXTBUFF_VPP_DETAIL	**mfxExtVPPDetail**
MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION	**mfxExtVPPFrameRateConversion**
MFX_EXTBUFF_VPP_IMAGE_STABILIZATION	**mfxExtVPPImageStab**
MFX_EXTBUFF_VPP_PICSTRUCT_DETECTION	none
MFX_EXTBUFF_VPP_PROCAMP	**mfxExtVPPProcAmp**

Example 4 shows how to configure the SDK video processing.

```
/* 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=&du;

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);
```

Example 4: Configure Video Processing

Region of Interest

During video processing operations, the application can specify a region of interest for each frame, as illustrated in Figure 4.

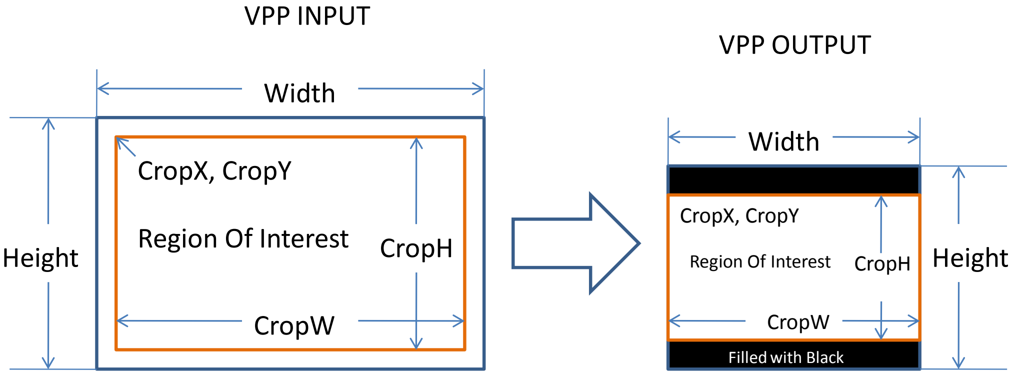

Figure 4: VPP Region of Interest Operation

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. Table 5 shows some examples.

Table 5: Examples of VPP Operations on Region of Interest

Operation	VPP Input		VPP Output	
	Width/Height	CropX, CropY, CropW, CropH	Width/Height	CropX, CropY, CropW, CropH
Cropping	720x480	16,16,688,448	720x480	16,16,688,448
Resizing	720x480	0,0,720,480	1440x960	0,0,1440,960
Horizontal stretching	720x480	0,0,720,480	640x480	0,0,640,480
16:9 → 4:3 with letter boxing at the top and bottom	1920x1088	0,0,1920,1088	720x480	0,36,720,408

4:3 → 16:9 with pillar boxing at the left and right	720x480	0,0,720,480	1920x1088	144,0,1632,1088

Transcoding Procedures

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

Asynchronous Pipeline

The application passes the output of an upstream SDK function to the input of the downstream SDK function to construct an asynchronous pipeline. Such pipeline construction is done at runtime and can be dynamically changed, as illustrated in Example 5.

```
mfxSyncPoint sp;
MFXVideoDECODE_DecodeFrameAsync(session,bs,work,vin, &sp_d);
if (going_through_vpp) {
      MFXVideoVPP_RunFrameVPPAsync(session,vin,vout, &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);
```

Example 5: Pseudo Code of Asynchronous Pipeline Construction

The Intel® Media SDK simplifies the requirement for asynchronous pipeline synchronization. The application only needs to synchronize after the last SDK function. Explicit synchronization of intermediate results is not required and in fact can slow performance.

The SDK tracks the dynamic pipeline construction and verifies dependency on input and output parameters to ensure the execution order of the pipeline function. In Example 5, the SDK 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 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 in use while the input surface is locked.

The Intel® Media SDK checks dependencies by comparing the input and output parameters of each SDK 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 that the dependency check works on the pointers to the structures only.)

There are two exceptions with respect to intermediate synchronization:

1. The application must synchronize any input before calling the SDK function **MFXVideoDecode_DecodeFrameAsync**, if the input is from any asynchronous operation.
2. When the application calls an asynchronous function to generate an output surface in video memory and passes that surface to a non-SDK component, it must explicitly synchronize the operation before passing the surface to the non-SDK component.

Surface Pool Allocation

When connecting SDK function **A** to SDK function **B**, the application must take into account the needs of both functions to calculate the number of frame surfaces in the surface pool. Typically, the application can use the formula N_a+N_b, where N_a is the frame surface needs from SDK function **A** output, and N_b is the frame surface needs from SDK function **B** input.

For performance considerations, the application must submit multiple operations and delays synchronization as much as possible, which gives the SDK flexibility to organize internal pipelining. For example, the operation sequence, ENCODE(f1) → ENCODE(f2) → SYNC(f1) → SYNC(f2) is recommended, compared with **ENCODE(f1) → SYNC(f1) → ENCODE(f2) → SYNC(f2).**

In this case, the surface pool needs additional surfaces to take into account multiple asynchronous operations before synchronization. The application can use the **AsyncDepth** parameter of the **mfxVideoParam** structure to inform an SDK function that how many asynchronous operations the application plans to perform before synchronization. The corresponding SDK **QueryIOSurf** function will reflect such consideration in the NumFrameSuggested value. Example 6 shows a way of calculating the surface needs based on NumFrameSuggested values.

```
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);
num_surfaces=     response_v[1].NumFrameSuggested
                  +response_e.NumFrameSuggested
                  -async_depth; /* double counted in ENCODE & VPP */
```

Example 6: Calculate Surface Pool Size

Pipeline Error Reporting

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

Assume the pipeline is **A**→**B**→**C**. The application synchronizes on sync point **C**. If the error occurs in SDK function **C**, then the synchronization returns the exact error code. If the error occurs before SDK function **C**, then the synchronization returns **MFX_ERR_ABORTED**. The application can then try to synchronize on sync point **B**. Similarly, if the error occurs in SDK function **B**, the synchronization returns the exact error code, or else **MFX_ERR_ABORTED**. Same logic applies if the error occurs in SDK function **A**.

Working with hardware acceleration

To fully utilize the SDK acceleration capability, the application should support OS specific infrastructures, Microsoft* DirectX* for Micorosoft* Windows* and VA API for Linux*. The exception is transcoding scenario where opaque memory type may be used. See Surface Type Neutral Transcoding for more details.

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

Depending on usage model, the application can use video memory on different stages of pipeline. Three major scenarios are illustrated on Figure 5.

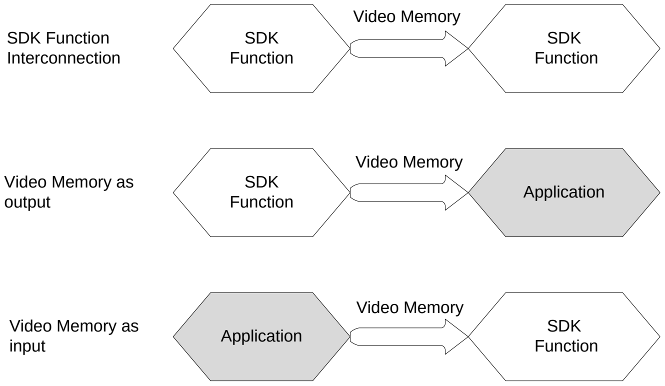

Figure 5 Usage of video memory for hardware acceleration

The application must use the **IOPattern** field of the **mfxVideoParam** structure to indicate the I/O access pattern during initialization. Subsequent SDK function calls must follow this access pattern. For example, if an SDK 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 SDK component requires the acceleration device handle. This handle is also used by SDK component to query HW capabilities. The application can share its device with the SDK by passing device handle through the **MFXVideoCORE_SetHandle** function. It is recommended to share the handle before any actual usage of the SDK.

Working with Microsoft* DirectX* Applications

The SDK supports two different infrastructures for hardware acceleration on Microsoft* Windows* OS, "Direct3D 9 DXVA2" and "Direct3D 11 Video API". In the first one the application should use the **IDirect3DDeviceManager9** interface as the acceleration device handle, in the second one - **ID3D11Device** interface. The application should share one of these interfaces with the SDK through the **MFXVideoCORE_SetHandle** function. If the application does not provide it, then the SDK creates its own internal acceleration device. This internal device could not be accessed by the application and as a result, the SDK input and output will be limited to system memory only. That in turn will reduce SDK performance. If the SDK fails to create a valid acceleration device, then SDK cannot proceed with hardware acceleration and returns an error status to the application.

The application must create the Direct3D9* device with the flag **D3DCREATE_MULTITHREADED**. Additionally the flag **D3DCREATE_FPU_PRESERVE** is recommended. This influences floating-point calculations, including PTS values.

The application must also set multithreading mode for Direct3D11* device. Example 7 Setting multithreading mode illustrates how to do it.

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

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

Example 7 Setting multithreading mode

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

When the SDK 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 SDK component Init function through an SDK external allocator callback. See the Memory Allocation and External Allocators section for details.

Only decoder **Init** function requests 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, SDK requires different surface types. It is strongly recommended to call one of the **MFXVideoENCODE_QueryIOSurf**, **MFXVideoDECODE_QueryIOSurf** or **MFXVideoVPP_QueryIOSurf** functions to determine the appropriate type.

Table 6: Supported SDK Surface Types and Color Formats for Direct3D9 shows supported Direct3D9 surface types and color formats. Table 7: Supported SDK Surface Types and Color Formats for Direct3D11 shows Direct3D11 types and formats. Note, that NV12 is the major encoding and decoding color format. Additionally, JPEG/MJPEG decoder supports RGB32 and YUY2 output and VPP supports RGB32 output.

Table 6: Supported SDK Surface Types and Color Formats for Direct3D9

SDK Class	**SDK Function Input**		**SDK Function Output**	
	Surface Type	**Color Format**	**Surface Type**	**Color Format**
DECODE	Not Applicable		Decoder Render Target	NV12
			Decoder Render Target	RGB32, YUY2 JPEG only
VPP	Decoder/Processor Render Target	Listed in ColorFourCC	Decoder Render Target	NV12
			Processor Render Target	RGB32
ENCODE	Decoder Render Target	NV12	Not Applicable	

Note: "Decoder Render Target" corresponds to DXVA2_ VideoDecoderRenderTarget type, "Processor Render Target" to DXVA2_ VideoProcessorRenderTarget.

Table 7: Supported SDK Surface Types and Color Formats for Direct3D11

SDK Class	**SDK Function Input**		**SDK Function Output**	
	Surface Type	**Color Format**	**Surface Type**	**Color Format**

DECODE	Not Applicable		Decoder Render Target	NV12
			Decoder /Processor Render Target	RGB32, YUY2 JPEG only
VPP	Decoder/Processor Render Target	Listed in **ColorFourCC**	Processor Render Target	NV12
			Processor Render Target	RGB32
ENCODE	Decoder/Processor Render Target	NV12	Not Applicable	

Note: "Decoder Render Target" corresponds to D3D11_BIND_DECODER flag, "Processor Render Target" to D3D11_BIND_RENDER_TARGET.

Working with VA API Applications

The SDK supports single infrastructure for hardware acceleration on Linux* - "VA API". The application should use the **VADisplay** interface as the acceleration device handle for this infrastructure and share it with the SDK through the **MFXVideoCORE_SetHandle** function. Because the SDK does not create internal acceleration device on Linux, the application must always share it with the SDK. This sharing should be done before any actual usage of the SDK, including capability query and component initialization. If the application fails to share the device, the SDK operation will fail.

Example 8 Obtaining VA display from X Window System and Example 9 Obtaining VA display from Direct Rendering Manager show how to obtain and share VA display with the SDK.

```
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);
```

Example 8 Obtaining VA display from X Window System

```
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);
```

Example 9 Obtaining VA display from Direct Rendering Manager

When the SDK decoder creates 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 SDK component Init function through an SDK external allocator callback. See the Memory Allocation and External Allocators section for details.

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

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.

Table 8: Supported SDK Surface Types and Color Formats for VA API shows supported by VA API color formats.

Table 8: Supported SDK Surface Types and Color Formats for VA API

SDK Class	SDK Function Input	SDK Function Output
	Color Format	Color Format
DECODE	Not Applicable	NV12
VPP	Listed in **ColorFourCC**	NV12, RGB32
ENCODE	NV12	Not Applicable

Memory Allocation and External Allocators

All SDK implementations delegate memory management to the application. The application must allocate sufficient memory for input and output parameters and buffers, and de-allocate it when SDK functions complete their operations. During execution, the SDK functions use callback functions to the application to manage memory through two external allocator interfaces:

- **mfxBufferAllocator** for general buffers
- **mfxFrameAllocator** for video frames

If an application needs to control the allocation of general buffers, it can use callback functions through the **mfxBufferAllocator** interface. Buffer allocator callbacks pass only large chunks of memory allocations, such as those of motion vectors. Unless the application specifies memory allocation through the **mfxBufferAllocator** interface, SDK functions will use heap memory by default. Example 10 shows a simple buffer allocator.

```
mfxStatus ba_alloc(mfxHDL pthis, mfxU32 nbytes,mfxU16 type, mfxMemId
*mid) {
      *mid=malloc(nbytes);
      return (*mid)?MFX_ERR_NONE:MFX_ERR_MEMORY_ALLOC;
}
mfxStatus ba_lock(mfxHDL pthis, mfxMemId mid, mfxU8 **ptr) {
      *ptr=(mfxU8 *)mid;
      return MFX_ERR_NONE;
}
mfxStatus ba_unlock(mfxHDL pthis, mfxMemId mid) {
      return MFX_ERR_NONE;
}
mfxStatus ba_free(mfxHDL pthis, mfxMemId mid) {
      if (mid) free((mfxU8 *)mid);
      return MFX_ERR_NONE;
}
```

Example 10: Buffer Allocator

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 and
- in video memory, as “decoder render targets” or “processor render targets.” See the section Working with hardware acceleration for additional details.

The external frame allocator responds only to frame allocation requests for the requested memory type and returns **MFX_ERR_UNSUPPORTED** for all others. The allocation request uses flags, part of memory type field, to indicate which SDK class initiates the request, so the external frame allocator can respond accordingly. Example 11 illustrates a simple external frame allocator.

```
typedef struct {
      mfxU16 width, height;
      mfxU8 *base;
} mid_struct;
mfxStatus fa_alloc(mfxHDL pthis, mfxFrameAllocRequest *request,
mfxFrameAllocResponse *response) {
      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) {
      mid_struct *mmid=(mid_struct *)mid;
      ptr→pitch=mmid→width;
      ptr→Y=mmid→base;
      ptr→U=ptr→Y+mmid→width*mmid→height;
      ptr→V=ptr→U+1;
      return MFX_ERR_NONE;
}
mfxStatus fa_unlock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
      if (ptr) ptr→Y=ptr→U=ptr→V=ptr→A=0;
      return MFX_ERR_NONE;
}
mfxStatus fa_gethdl(mfxHDL pthis, mfxMemId mid, mfxHDL *handle) {
      return MFX_ERR_UNSUPPORTED;
}
mfxStatus fa_free(mfxHDL pthis, mfxFrameAllocResponse *response) {
      for (int i=0;i<response→NumFrameActual;i++) {
            mid_struct *mmid=(mid_struct *)response→mids[i];
            free(mmid→base); free(mid);
      }
      return MFX_ERR_NONE;
}
```

Example 11: Example Frame Allocator

Surface Type Neutral Transcoding

Performance wise, software SDK library (running CPU instructions) prefers system memory I/O, and SDK platform implementation (accelerated by platform graphic devices) prefers video memory surface I/O. The application needs to manage both surface types (thus two data paths in a transcoding **`A`**→**`B`**) to achieve the best performance in both cases.

The SDK provides a third surface type: opaque surface. With opaque surface, the SDK will map the surface type to either system memory buffer or video memory surface at runtime. The application only needs to manage one surface type, or one transcoding data path.

It is recommended the application use opaque surfaces for any transcoding intermediate data. For example, the transcoding pipeline can be **DECODE** → Opaque Surfaces → **VPP** → Opaque Surfaces → **ENCODE**. It is possible to copy an opaque surface to a "real" surface through a **VPP** operation.

The application uses the following procedure to use opaque surface, assuming a transcoding pipeline SDK A → SDK B:

- As described in section *Surface Pool Allocation*, the application queries SDK component **`A`** and **`B`** and calculates the surface pool size. The application needs to use **`MFX_IOPATTERN_IN_OPAQUE_MEMORY`** and/or **`MFX_IOPATTERN_OUT_OPAQUE_MEMORY`** while specifying the I/O pattern. It is possible that SDK component A returns a different memory type than SDK component B, as the **`QueryIOSurf`** function returns the native allocation type and size. In this case, the surface pool type and size should follow only one SDK component: either A or B.
- The application allocates the surface pool, which is an array of the **`mfxFrameSurface1`** structures. Within the structure, specify **`Data.Y= Data.U= Data.V= Data.A= Data.MemId=0`** for all array members.
- During initialization, the application communicates the allocated surface pool to both SDK components by attaching the **`mfxExtOpaqueSurfaceAlloc`** structure as part of the initialization parameters. The application needs to use **`MFX_IOPATTERN_IN_OPAQUE_MEMORY`** and/or **`MFX_IOPATTERN_OUT_OPAQUE_MEMORY`** while specifying the I/O pattern.
- During decoding, encoding, and video processing, the application manages the surface pool and passes individual frame surface to SDK component **`A`** and **`B`** as described in section *Decoding Procedures*, section *Encoding Procedures*, and section *Video Processing Procedures*, respectively.

Example 12 shows the opaque procedure sample code.

Since the SDK manages the association of opaque surface to "real" surface types internally, the application cannot read the content of opaque surfaces. Also the application does not get any opaque-type surface allocation requests if the application specifies an external frame allocator.

If the application shares opaque surfaces among different SDK sessions, the application must join the sessions before SDK component initialization and ensure that all joined sessions have the same hardware acceleration device handle. Setting device handle is optional only if all components in pipeline belong to the same session.

```
mfxExtOpqueSurfaceAlloc osa, *posa=&osa;
memset(&osa,0,sizeof(osa));

// query frame surface allocation needs
MFXVideoDECODE_QueryIOSurf(session, &decode_param, &request_decode);
MFXVideoENCODE_QueryIOSurf(session, &encode_param, &request_encode);

// calculate the surface pool surface type and numbers
if (MFX_MEMTYPE_BASE(request_decode.Type) ==
    MFX_MEMTYPE_BASE(request_encode.Type)) {
    osa.Out.NumSurface = request_decode.NumFrameSuggested +
        request_encode.NumFrameSuggested - decode_param.AsyncDepth;
    osa.Out.Type=request_decode.Type;
} else {
    // it is also ok to use decode's NumFrameSuggested and Type.
    osa.Out.NumSurface=request_encode.NumFrameSuggested;
    osa.Out.Type=request_encode.Type;
}

// allocate surface pool and zero MemId/Y/U/V/A pointers
osa.Out.Surfaces=alloc_mfxFrameSurface1(osa.Out.NumSurface);

// attach the surface pool during decode & encode initialization
osa.Header.BufferId=MFX_EXTBUFF_OPAQUE_SURFACE_ALLOCATION;
osa.Header.BufferSz=sizeof(osa);
decode_param.NumExtParam=1;
decode_param.ExtParam=&posa;
MFXVideoDECODE_Init(session, &decode_param);

memcpy(&osa.In, &osa.Out, sizeof(osa.Out));
encode_param.NumExtParam=1;
encode_param.ExtParam=&posa;
MFXVideoENCODE_Init(session, &encode_param);
```

Example 12: Pseudo-Code of Opaque Surface Procedure

Hardware Device Error Handling

The SDK accelerates decoding, encoding and video processing through a hardware device. The SDK functions may return the following errors or warnings if the hardware device encounters errors:

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

SDK functions **Query**, **QueryIOSurf**, and **Init** return **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 SDK functions may return errors or other warnings overwriting **MFX_WRN_PARTIAL_ACCELERATION**, as it is a lower priority warning.)

SDK functions return **MFX_WRN_DEVICE_BUSY** to indicate that the hardware device is busy and unable to take commands at this time. Resume the operation by waiting for a few milliseconds and resubmitting the request. Example 13 shows the decoding pseudo-code. The same procedure applies to encoding and video processing.

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

```
mfxStatus sts=MFX_ERR_NONE;
for (;;) {
      …
      sts=MFXVideoDECODE_DecodeFrameAsync(session, bitstream,
surface_work, &surface_disp, &syncp);
      if (sts == MFX_WRN_DEVICE_BUSY) Sleep(5);
}
```

Example 13: Pseudo-Code to Handle MFX_ERR_DEVICE_BUSY

Function Reference

This section describes SDK functions and their operations.

In each function description, only commonly used status codes are documented. The function may return additional status codes, such as MFX_ERR_INVALID_HANDLE or MFX_ERR_NULL_PTR, in certain case. See the mfxStatus enumerator for a list of all status codes.

Global Functions

Global functions initialize and de-initialize the Intel® Media SDK library and perform query functions on a global scale within an application.

Member Functions	Description
MFXInit	Initializes an SDK session
MFXQueryIMPL	Queries the implementation type
MFXQueryVersion	Queries the implementation version
MFXJoinSession	Join two sessions together
MFXCloneSession	Clone the current session
MFXSetPriority	Set session priority
MFXGetPriority	Obtain session priority
MFXDisjoinSession	Remove the join state of the current session
MFXClose	De-initializes an SDK session

MFXCloneSession

Syntax

```
mfxStatus MFXCloneSession(mfxSession session, mfxSession *clone);
```

Parameters

session	SDK session handle

`clone`	Pointer to the cloned session handle

Description

This function creates a clean copy of the current session. The cloned session is an independent session. It 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**.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.1.

MFXClose

Syntax

```
mfxStatus MFXClose(mfxSession session);
```

Parameters

`session`	SDK session handle

Description

This function completes and de-initializes an SDK session. Any active tasks in execution or in queue are aborted. The application cannot call any SDK function after this function.

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

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXDisjoinSession

Syntax

```
mfxStatus MFXDisjoinSession(mfxSession session);
```

Parameters

`session`	SDK session handle

Description

This function 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 function.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_WRN_IN_EXECUTION`	Active tasks are in execution or in queue. Wait for the completion of the tasks and then call this function again.
`MFX_ERR_UNDEFINED_BEHAVIOR`	The session is independent, or this session is the parent of all joined sessions.

Change History

This function is available since SDK API 1.1.

MFXGetPriority

Syntax

```
mfxStatus MFXGetPriority(mfxSession session, mfxPriority *priority);
```

Parameters

`session`	SDK session handle
`priority`	Pointer to the priority value

Description

This function returns the current session priority.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.1.

MFXInit

Syntax

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

Parameters

`impl`	**mfxIMPL** enumerator that indicates the desired SDK implementation
`ver`	Pointer to the minimum library version or zero, if not specified
`session`	Pointer to the SDK session handle

Description

This function creates and initializes an SDK session. Call this function before calling any other SDK functions. If the desired implementation specified by `impl` is **MFX_IMPL_AUTO**, the function will search for the platform-specific SDK implementation. If the function cannot find it, it will use the software implementation.

The argument `ver` indicates the desired version of the library implementation. The loaded SDK 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 SDK release, with which an application is built.

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

Return Status

`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 SDK implementation or version.

Change History

This function is available since SDK API 1.0.

MFXJoinSession

Syntax

```
mfxStatus MFXJoinSession(mfxSession session, mfxSession child);
```

Parameters

`session`	The current session handle
`child`	The child session handle to be joined

Description

This function 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.

Return Status

`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.

Change History

This function is available since SDK API 1.1.

MFXQueryIMPL

Syntax

```
mfxStatus MFXQueryIMPL(mfxSession session, mfxIMPL *impl);
```

Parameters

`session`	SDK session handle
`impl`	Pointer to the implementation type

Description

This function returns the implementation type of a given session.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXQueryVersion

Syntax

```
mfxStatus MFXQueryVersion(mfxSession session, mfxVersion *version);
```

Parameters

`session`	SDK session handle
`version`	Pointer to the returned implementation version

Description

This function returns the SDK implementation version.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXSetPriority

Syntax

```
mfxStatus MFXSetPriority(mfxSession session, mfxPriority priority);
```

Parameters

`session`	SDK session handle
`priority`	Priority value

Description

This function sets the current session priority.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.1.

MFXVideoCORE

This class of functions consists of auxiliary functions that all functions of the SDK implementation can call.

Member Functions	
MFXVideoCORE_SetHandle	Sets system handles that the SDK implementation might need
MFXVideoCORE_GetHandle	Obtains system handles previously set
MFXVideoCORE_SetBufferAllocator	Sets the external system buffer allocator
MFXVideoCORE_SetFrameAllocator	Sets the external frame allocator
MFXVideoCORE_SyncOperation	Initializes execution of the specified sync point and returns a status code

MFXVideoCORE_SetHandle

Syntax

```
mfxStatus MFXVideoCORE_SetHandle(mfxSession session, mfxHandleType type,
mfxHDL hdl);
```

Parameters

`session`	SDK session handle
`type`	Handle type
`hdl`	Handle to be set

Description

This function sets any essential system handle that SDK 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 SDK session closes.

Return Status

`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 internal handle has been created by the SDK before this function call.

Change History

This function is available since SDK API 1.0.

MFXVideoCORE_GetHandle

Syntax

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

Parameters

`session`	SDK session handle
`type`	Handle type
`hdl`	Pointer to the handle to be set

Description

This function 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.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_ERR_NOT_FOUND`	Specified handle type not found.

Change History

This function is available since SDK API 1.0.

MFXVideoCORE_SetBufferAllocator

Syntax

```
mfxStatus MFXVideoCORE_SetBufferAllocator(mfxSession session,
mfxBufferAllocator *allocator);
```

Parameters

`session`	SDK session handle
`allocator`	Pointer to the **mfxBufferAllocator** structure

Description

This function sets the external allocator callback structure for allocation of system memory-based buffers. If the `allocator` argument is `NULL`, the SDK uses the default allocator.

The behavior of the SDK is undefined if it uses this function while a previous allocator is in use. A general guideline is to set the allocator immediately after initializing the session.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoCORE_SetFrameAllocator

Syntax

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

Parameters

`session`	SDK session handle
`allocator`	Pointer to the `mfxFrameAllocator` structure

Description

This function sets the external allocator callback structure for frame allocation. If the `allocator` argument is `NULL`, the SDK uses the default allocator, which allocates frames from system memory or hardware devices.

The behavior of the SDK 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.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoCORE_SyncOperation

Syntax

```
mfxStatus MFXVideoCORE_SyncOperation(mfxSession session, mfxSyncPoint
syncp, mfxU32 wait);
```

Parameters

`session`	SDK session handle
`syncp`	Sync point
`wait`	Wait time in milliseconds

Description

This function 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.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`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.

Change History

This function is available since SDK API 1.0.

Remarks

See status codes for specific asynchronous functions.

MFXVideoENCODE

This class of functions performs the entire encoding pipeline from the input video frames to the output bitstream.

Member Functions	
MFXVideoENCODE_Query	Queries the feature capability
MFXVideoENCODE_QueryIOSurf	Queries the number of input surface frames required for encoding
MFXVideoENCODE_Init	Initializes the encoding operation
MFXVideoENCODE_Reset	Resets the current encoding operation and prepares for the next encoding operation
MFXVideoENCODE_Close	Terminates the encoding operation and de-allocates any internal memory
MFXVideoENCODE_GetVideoParam	Obtains the current working parameter set
MFXVideoENCODE_GetEncodeStat	Obtains the statistics collected during encoding
MFXVideoENCODE_EncodeFrameAsync	Performs the encoding and returns the compressed bitstream

MFXVideoENCODE_Query

Syntax

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

Parameters

`session`	SDK session handle
`in`	Pointer to the **mfxVideoParam** structure as input
`out`	Pointer to the **mfxVideoParam** structure as output

Description

This function works in either of four modes:

1. If the `in` pointer is zero, the function returns the class configurability in the output **mfxVideoParam** structure. A non-zero value in each field of the output structure indicates that the SDK implementation can configure the field with **Init**.
2. If the `in` parameter is non-zero, the function checks the validity of the fields in the input **mfxVideoParam** structure. Then the function returns the corrected values in the output **mfxVideoParam** 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 SDK implementation supports certain profiles, levels or bitrates.
3. If the `in` parameter is non-zero and **mfxExtEncoderResetOption** structure is attached to it, then 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 such reset is possible and returns error otherwise. Unlike other modes that are independent of the SDK encoder state, this one checks if reset is possible in the present SDK encoder state. This mode also requires completely defined **mfxVideoParam** structure, unlike other modes that support partially defined configurations. See **mfxExtEncoderResetOption** description for more details.
4. If the `in` parameter is non-zero and **mfxExtEncoderCapability** structure is attached to it, then the function returns encoder capability in **mfxExtEncoderCapability** structure attached to `out`. It is recommended to fill in **mfxVideoParam** structure and set 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 **mfxVideoParam** structure is a mandated field (to be filled by the application) to identify the coding standard.

Return Status

`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 SDK HW implementations may return this status code.
`MFX_WRN_INCOMPATIBLE_VIDEO_PARAM`	The function detected some video parameters were incompatible with others; incompatibility resolved.

Change History

This function is available since SDK API 1.0.

MFXVideoENCODE_QueryIOSurf

Syntax

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

Parameters

`session`	SDK session handle
`par`	Pointer to the **mfxVideoParam** structure as input
`request`	Pointer to the **mfxFrameAllocRequest** structure as output

Description

This function 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.

The use of this function is recommended. For more information, see the section Working with hardware acceleration.

This function does not validate I/O parameters except those used in calculating the number of input surfaces.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_WRN_PARTIAL_ACCELERATION`	The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only SDK HW implementations may return this status code.
`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.

Change History

This function is available since SDK API 1.0.

MFXVideoENCODE_Init

Syntax

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

Parameters

`session`	SDK session handle
`par`	Pointer to the `mfxVideoParam` structure

Description

This function 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.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_WRN_PARTIAL_ACCELERATION`	The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only SDK HW implementations may return this status code.
`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 function is called twice without a close;

Change History

This function is available since SDK API 1.0.

MFXVideoENCODE_Reset

Syntax

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

Parameters

`session`
SDK session handle

`par`
Pointer to the **mfxVideoParam** structure

Description

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

Return Status

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

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

	resolved.

Change History

This function is available since SDK API 1.0.

MFXVideoENCODE_Close

Syntax

```
mfxStatus MFXVideoENCODE_Close(mfxSession session);
```

Parameters

`session`	SDK session handle

Description

This function terminates the current encoding operation and de-allocates any internal tables or structures.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoENCODE_GetVideoParam

Syntax

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

Parameters

`session`	SDK session handle
`par`	Pointer to the corresponding parameter structure

Description

This function 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.

Returned information

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoENCODE_GetEncodeStat

Syntax

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

Parameters

`session`	SDK session handle
`stat`	Pointer to the **mfxEncodeStat** structure

Description

This function obtains statistics collected during encoding.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoENCODE_EncodeFrameAsync

Syntax

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

Parameters

Session	SDK session handle
ctrl	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.
surface	Pointer to the frame surface structure
bs	Pointer to the output bitstream
syncp	Pointer to the returned sync point associated with this operation

Description

This function 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 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 **MFXVideoENCODE_GetVideoParam** 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.

Return Status

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 Working with Microsoft* DirectX* Applications section for

	further information.
`MFX_WRN_DEVICE_BUSY`	Hardware device is currently busy. Call this function again in a few milliseconds.
`MFX_ERR_INCOMPATIBLE_VIDEO_PARAM`	Inconsistent parameters detected not conforming to Appendix A.

Change History

This function is available since SDK API 1.0.

Remarks

If the **`EncodedOrder`** field in the **`mfxInfoMFX`** structure is true, input frames enter the encoder in the order of their encoding. However, the **`FrameOrder`** field in the **`mfxFrameData`** structure of each frame must be set to the display order. If **`EncodedOrder`** is false, the function ignores the **`FrameOrder`** field.

MFXVideoENC

This class of functions performs the first step of encoding process – motion estimation, intra prediction and mode decision. This functions are declared in **`mfxenc.h`** file.

Member Functions	
`MFXVideoENC_Query`	Queries the feature capability
`MFXVideoENC_QueryIOSurf`	Queries the number of input surface frames required for encoding
`MFXVideoENC_Init`	Initializes the encoding operation
`MFXVideoENC_Reset`	Resets the current encoding operation and prepares for the next encoding operation
`MFXVideoENC_Close`	Terminates the encoding operation and de-allocates any internal memory
`MFXVideoENC_ProcessFrameAsync`	Performs the first step of encoding process and returns intermediate data.

MFXVideoENC_Query

Syntax

```
mfxStatus MFXVideoENC_Query(mfxSession session, mfxVideoParam *in,
mfxVideoParam *out);
```

Parameters

`session`	SDK session handle
`in`	Pointer to the **mfxVideoParam** structure as input
`out`	Pointer to the **mfxVideoParam** structure as output

Description

This function works in either of two modes:

1. If the `in` pointer is zero, the function returns the class configurability in the output **mfxVideoParam** structure. A non-zero value in each field of the output structure indicates that the SDK implementation can configure the field with **Init**.
2. If the `in` parameter is non-zero, the function checks the validity of the fields in the input **mfxVideoParam** structure. Then the function returns the corrected values in the output **mfxVideoParam** 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 SDK implementation supports certain profiles, levels or bitrates.

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

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_ERR_UNSUPPORTED`	The function failed to identify a specific implementation for the required features.
`MFX_WRN_INCOMPATIBLE_VIDEO_PARAM`	The function detected some video parameters were incompatible with others; incompatibility resolved.

Change History

This function is available since SDK API 1.10.

MFXVideoENC_QueryIOSurf

Syntax

```
mfxStatus MFXVideoENC_QueryIOSurf(mfxSession session, mfxVideoParam
*par, mfxFrameAllocRequest *request);
```

Parameters

`session`	SDK session handle
`par`	Pointer to the **mfxVideoParam** structure as input
`request`	Pointer to the **mfxFrameAllocRequest** structure as output

Description

This function returns minimum and suggested numbers of the input frame surfaces required for ENC initialization and their type.

This function does not validate I/O parameters except those used in calculating the number of input surfaces.

Return Status

`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.

Change History

This function is available since SDK API 1.10.

MFXVideoENC_Init

Syntax

```
mfxStatus MFXVideoENC_Init(mfxSession session, mfxVideoParam *par);
```

Parameters

`session`	SDK session handle
`par`	Pointer to the **mfxVideoParam** structure

Description

This function performs ENC initialization.

Return Status

`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 function is called twice without a close;

Change History

This function is available since SDK API 1.10.

MFXVideoENC_Reset

Syntax

```
mfxStatus MFXVideoENC_Reset(mfxSession session, mfxVideoParam *par);
```

Parameters

`session`	SDK session handle
`par`	Pointer to the **mfxVideoParam** structure

Description

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

Return Status

`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 provided by the application video parameters are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the SDK component and then reinitialize it.
`MFX_WRN_INCOMPATIBLE_VIDEO_PARAM`	The function detected some video parameters were incompatible with others; incompatibility resolved.

Change History

This function is available since SDK API 1.10.

MFXVideoENC_Close

Syntax

```
mfxStatus MFXVideoENC_Close(mfxSession session);
```

Parameters

`session`	SDK session handle

Description

This function terminates the current encoding operation and de-allocates any internal tables or structures.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.10.

MFXVideoENC_ProcessFrameAsync

Syntax

```
mfxStatus MFXVideoENC_ProcessFrameAsync(mfxSession session, mfxENCInput
*in, mfxENCOutput *out, mfxSyncPoint *syncp);
```

Parameters

`Session`	SDK session handle
`in`	Input parameters for ENC operation.
`out`	Output parameters of encoding operation.
`syncp`	Pointer to the returned sync point associated with this operation

Description

This function performs the first step of encoding process – motion estimation, intra prediction and mode decision. Its exact operation, input and output parameters depend on usage model.

This function is stateless, i.e. each function call is independent from other calls.

This function is asynchronous.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.10.

MFXVideoDECODE

This class of functions implements a complete decoder that decompresses input bitstreams directly to output frame surfaces.

Member Functions	
MFXVideoDECODE_Query	Queries the feature capability
MFXVideoDECODE_QueryIOSurf	Queries the number of frames required for decoding
MFXVideoDECODE_DecodeHeader	Parses the bitstream to obtain the video parameters for initialization

MFXVideoDECODE_Init	Initializes the decoding operation
MFXVideoDECODE_Reset	Resets the current decoding operation and prepares for the next decoding operation
MFXVideoDECODE_Close	Terminates the decoding operation and de-allocates any internal memory
MFXVideoDECODE_GetVideoParam	Obtains the current working parameter set
MFXVideoDECODE_GetDecodeStat	Obtains statistics during decoding
MFXVideoDECODE_GetPayload	Obtains user data or SEI messages embedded in the bitstream
MFXVideoDECODE_SetSkipMode	Set decoder skip mode
MFXVideoDECODE_DecodeFrameAsync	Performs decoding from the input bitstream to the output frame surface

MFXVideoDECODE_DecodeHeader

Syntax

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

Parameters

session	SDK session handle
bs	Pointer to the bitstream
par	Pointer to the mfxVideoParam structure

Description

This function parses the input bitstream and fills the mfxVideoParam structure with appropriate values, such as resolution and frame rate, for the **Init** function. The application can then pass the resulting mfxVideoParam structure to the MFXVideoDECODE_Init function for decoder initialization.

An application can call this function at any time before or after decoder initialization. If the SDK 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.

Return Status

`MFX_ERR_NONE`	The function successfully filled **mfxVideoParam** structure. It does not mean that the stream can be decoded by SDK. 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.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_Query

Syntax

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

Parameters

`session`	SDK session handle
`in`	Pointer to the **mfxVideoParam** structure as input
`out`	Pointer to the **mfxVideoParam** structure as output

Description

This function works in one of two modes:

1. If the `in` pointer is zero, the function returns the class configurability in the output **mfxVideoParam** structure. A non-zero value in each field of the output structure indicates that the field is configurable by the SDK implementation with the **MFXVideoDECODE_Init** function).

2. If the `in` parameter is non-zero, the function checks the validity of the fields in the input **mfxVideoParam** structure. Then the function returns the corrected values to the output **mfxVideoParam** 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 SDK 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 **mfxVideoParam** structure is a mandated field (to be filled by the application) to identify the coding standard.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_ERR_UNSUPPORTED`	The function failed to identify a specific implementation.
`MFX_WRN_PARTIAL_ACCELERATION`	The underlying hardware does not fully support the specified video parameters; The decoding may be partially accelerated. Only SDK HW implementations may return this status code.
`MFX_WRN_INCOMPATIBLE_VIDEO_PARAM`	The function detected some video parameters were incompatible with others; incompatibility resolved.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_QueryIOSurf

Syntax

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

Parameters

`session`	SDK session handle
`par`	Pointer to the **mfxVideoParam** structure as input
`request`	Pointer to the **mfxFrameAllocRequest** structure as output

Description

The function 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 section Working with hardware acceleration.

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.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_WRN_PARTIAL_ACCELERATION`	The underlying hardware does not fully support the specified video parameters; The decoding may be partially accelerated. Only SDK HW implementations may return this status code.
`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.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_Init

Syntax

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

Parameters

`session`	SDK session handle
`par`	Pointer to the `mfxVideoParam` structure

Description

This function allocates memory and prepares tables and necessary structures for decoding. This function also does extensive validation to determine whether the configuration is supported as specified in the input parameters.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

MFX_WRN_PARTIAL_ACCELERATION	The underlying hardware does not fully support the specified video parameters; The decoding may be partially accelerated. Only SDK hardware implementations return this status code.
MFX_ERR_INVALID_VIDEO_PARAM	The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of parameters resulted in an incompatibility error. Incompatibility was not resolved.
MFX_WRN_INCOMPATIBLE_VIDEO_PARAM	The function detected some video parameters were incompatible; Incompatibility resolved.
MFX_ERR_UNDEFINED_BEHAVIOR	The function is called twice without a close.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_Reset

Syntax

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

Parameters

session	SDK session handle
par	Pointer to the mfxVideoParam structure

Description

This function 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.

Return Status

`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 provided by the application video parameters are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the SDK component and then reinitialize it.
`MFX_WRN_INCOMPATIBLE_VIDEO_PARAM`	The function detected some video parameters were incompatible; Incompatibility resolved.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_Close

Syntax

```
mfxStatus MFXVideoDECODE_Close(mfxSession session);
```

Parameters

`session`	SDK session handle

Description

This function terminates the current decoding operation and de-allocates any internal tables or structures.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_GetVideoParam

Syntax

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

Parameters

`session`	SDK session handle
`par`	Pointer to the corresponding parameter structure

Description

This function 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.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_GetDecodeStat

Syntax

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

Parameters

`session`	SDK session handle
`stat`	Pointer to the **mfxDecodeStat** structure

Description

This function obtains statistics collected during decoding.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_GetPayload

Syntax

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

Parameters

`session`	SDK session handle
`ts`	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`	Pointer to the **`mfxPayload`** structure; the payload contains user data in MPEG-2 or SEI messages in H.264.

Description

This function 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 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`.

Return Status

`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.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_SetSkipMode

Syntax

```
mfxStatus MFXVideoDECODE_SetSkipMode(mfxSession session, mfxSkipMode
mode);
```

Parameters

`session`	SDK session handle
`mode`	Decoder skip mode. See the **mfxSkipMode** enumerator for details.

Description

This function sets the decoder skip mode. The application may use it to increase decoding performance by sacrificing output quality. The rising of skip level firstly results in skipping of some decoding operations like deblocking and then leads to frame skipping; firstly, B then P. Particular details are platform dependent.

Return Status

`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.

Change History

This function is available since SDK API 1.0.

MFXVideoDECODE_DecodeFrameAsync

Syntax

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

Parameters

`Session`	SDK session handle
`Bs`	Pointer to the input bitstream
`surface_work`	Pointer to the working frame buffer for the

	decoder
`surface_out`	Pointer to the output frame in the display order
`Syncp`	Pointer to the sync point associated with this operation

Description

This function 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, the function 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 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 1 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.

This function is asynchronous.

Return Status

`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 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.

Change History

This function is available since SDK API 1.0.

MFXVideoVPP

This class of functions performs video processing before encoding.

Member Functions	
MFXVideoVPP_Query	Queries the feature capability
MFXVideoVPP_QueryIOSurf	Queries the number of input and output surface frames required for video processing
MFXVideoVPP_Init	Initializes the VPP operation
MFXVideoVPP_Reset	Resets the current video processing operation and prepares for the next operation
MFXVideoVPP_Close	Terminates the video processing operation and de-allocates internal memory
MFXVideoVPP_GetVideoParam	Obtains the current working parameter set
MFXVideoVPP_GetVPPStat	Obtains statistics collected during video processing

MFXVideoVPP_RunFrameVPPAsync	Performs video processing on the frame level

MFXVideoVPP_Query

Syntax

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

Parameters

session	SDK session handle
in	Pointer to the mfxVideoParam structure as input
out	Pointer to the mfxVideoParam structure as output

Description

This function works in either of two modes:

1. If in is zero, the function returns the class configurability in the output mfxVideoParam structure. A non-zero value in a field indicates that the SDK implementation can configure it with **Init**.
2. If in is non-zero, the function checks the validity of the fields in the input mfxVideoParam structure. Then the function returns the corrected values in the output mfxVideoParam 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.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_UNSUPPORTED	The SDK 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 SDK HW implementations may return this status code.
MFX_WRN_INCOMPATIBLE_VIDEO_PARAM	The function detected some video parameters were incompatible with others; incompatibility resolved.

Change History

This function is available since SDK API 1.0.

MFXVideoVPP_QueryIOSurf

Syntax

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

Parameters

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

Description

This function returns minimum and suggested numbers of input and output 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.

The function is recommended. For more information, see the Working with hardware acceleration.

This function does not validate I/O parameters except those used in calculating the number of input and output surfaces.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_WRN_PARTIAL_ACCELERATION`	The underlying hardware does not fully support the specified video parameters; The video processing may be partially accelerated. Only SDK HW implementation may return this status code.
`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.

Change History

This function is available since SDK API 1.0.

MFXVideoVPP_Init

Syntax

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

Parameters

`Session`	SDK session handle
`Par`	Pointer to the **mfxVideoParam** structure

Description

This function allocates memory and prepares tables and necessary structures for video processing. This function also does extensive validation to ensure the configuration, as specified in the input parameters, is supported.

Return Status

`MFX_ERR_NONE`	The function completed successfully.
`MFX_WRN_PARTIAL_ACCELERATION`	The underlying hardware does not fully support the specified video parameters; The video processing may be partially accelerated. Only SDK HW implementation may return this status code.
`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 function was called twice without a close.
`MFX_WRN_FILTER_SKIPPED`	The VPP skipped one or more filters requested by the application.

Change History

This function is available since SDK API 1.0. SDK API 1.6 added new return status, `MFX_WRN_FILTER_SKIPPED`.

MFXVideoVPP_Reset

Syntax

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

Parameters

`session`	SDK session handle
`par`	Pointer to the **`mfxVideoParam`** structure

Description

This function stops the current video processing operation and restores internal structures or parameters for a new operation.

Return Status

`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 provided by the application video parameters are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the SDK component and then reinitialize it.
`MFX_WRN_INCOMPATIBLE_VIDEO_PARAM`	The function detected some video parameters were incompatible with others; incompatibility resolved.

Change History

This function is available since SDK API 1.0.

MFXVideoVPP_Close

Syntax

```
mfxStatus MFXVideoVPP_Close(mfxSession session);
```

Parameters

`session`	SDK session handle

Description

This function terminates the current video processing operation and de-allocates internal tables and structures.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoVPP_GetVideoParam

Syntax

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

Parameters

`session`	SDK session handle
`par`	Pointer to the corresponding parameter structure

Description

This function obtains current working parameters to the specified output structure. To return extended buffers, the application must allocate those extended buffers and attach them as part of the output structure.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoVPP_GetVPPStat

Syntax

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

Parameters

`session`	SDK session handle
`stat`	Pointer to the **`mfxVPPStat`** structure

Description

This function obtains statistics collected during video processing.

Return Status

`MFX_ERR_NONE`	The function completed successfully.

Change History

This function is available since SDK API 1.0.

MFXVideoVPP_RunFrameVPPAsync

Syntax

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

Parameters

`session`	SDK session handle
`in`	Pointer to the input video surface structure
`out`	Pointer to the output video surface structure

`aux`	Optional pointer to the auxiliary data structure
`syncp`	Pointer to the output sync point

Description

This function 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 section Video Processing Procedures 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.

Return Status

`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 in a few milliseconds.

Change History

This function is available since SDK API 1.0.

Structure Reference

In the following structure references, all reserved fields must be zero.

mfxBitstream

Definition

```
typedef struct mfxBitStream {
    union {
        struct {
            mfxEncryptedData* EncryptedData;
            mfxExtBuffer **ExtParam;
            mfxU16  NumExtParam;
        };
        mfxU32  reserved[6];
    };    mfxI64  DecodeTimeStamp;
    mfxU64  TimeStamp;
    mfxU8*  Data;
    mfxU32  DataOffset;
    mfxU32  DataLength;
    mfxU32  MaxLength;

    mfxU16  PicStruct;
    mfxU16  FrameType;
    mfxU16  DataFlag;
    mfxU16  reserved2;
} mfxBitstream;
```

Description

The `mfxBitstream` structure defines the buffer that holds compressed video data.

Members

`EncryptedData`   Reserved and must be zero.

`ExtParam` Array of extended buffers for additional bitstream configuration. See the **ExtendedBufferID** enumerator for a complete list of extended buffers.

`NumExtParam` The number of extended buffers attached to this structure.

`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 SDK encoder from presentation time stamp provided by the application in **mfxFrameSurface1** structure and from frame rate provided by the application during the SDK encoder initialization.

`TimeStamp` Time stamp of the compressed bitstream in units of 90KHz. A value of **MFX_TIMESTAMP_UNKNOWN** indicates that there is no time stamp.

`Data` Bitstream buffer pointer—32-bytes aligned

`DataOffset` Next reading or writing position in the bitstream buffer

`DataLength` Size of the actual bitstream data in bytes

`MaxLength` Allocated bitstream buffer size in bytes

`PicStruct` Type of the picture in the bitstream; this is an output parameter.

`FrameType` Frame type of the picture in the bitstream; this is an output parameter.

`DataFlag` Indicates additional bitstream properties; see the **BitstreamDataFlag** enumerator for details.

Change History

This structure is available since SDK API 1.0.

SDK API 1.1 extended the `DataFlag` field definition.

SDK API 1.6 adds `DecodeTimeStamp` field.

SDK API 1.7 adds `ExtParam` and `NumExtParam` fields.

mfxBufferAllocator

Definition

```
typedef struct {
      mfxU32      reserved[4];
      mfxHDL      pthis;
      mfxStatus   (*Alloc)(mfxHDL pthis, mfxU32 nbytes,
                              mfxU16 type, mfxMemId *mid);
      mfxStatus   (*Lock)(mfxHDL pthis, mfxMemId mid, mfxU8 **ptr);
      mfxStatus   (*Unlock)(mfxHDL pthis, mfxMemId mid);
      mfxStatus   (*Free)(mfxHDL pthis, mfxMemId mid);
} mfxBufferAllocator;
```

Description

The `mfxBufferAllocator` structure describes callback functions `Alloc, Lock, Unlock` and `Free` that the SDK implementation can use for allocating large chunks of internal system memory. Applications that are memory-conscious can set callbacks to manage memory.

The SDK behavior is undefined when using an incompletly defined external allocator. See the section Memory Allocation and External Allocators for additional information.

Members

`pthis`	Pointer to the allocator object
Alloc	Pointer to the function that allocates a linear buffer
Lock	Pointer to the function that locks a memory block and returns the pointer to the buffer
Unlock	Pointer to the function that unlocks a linear buffer; after unlocking, any pointer to the linear buffer is invalid.
Free	Pointer to the function that de-allocates memory

Change History

This structure is available since SDK API 1.0.

Alloc

Syntax

```
mfxStatus (*Alloc)(mfxHDL pthis, mfxU32 nbytes, mfxU16 type, mfxMemId
*mid);
```

Parameters

`pthis`	Pointer to the allocator object
`nbytes`	Number of bytes in the linear buffer
`type`	Memory type; see the **ExtMemBufferType** enumerator for details.
`mid`	Pointer to the allocated memory ID

Description

This function allocates a linear buffer and returns its block ID. The allocated memory must be 32-byte aligned.

Return Status

`MFX_ERR_NONE`	The function successfully allocated the memory block.
`MFX_ERR_MEMORY_ALLOC`	The function ran out of the specified type of memory.

Change History

This function is available since SDK API 1.0.

Free

Syntax

```
mfxStatus (*Free)(mfxHDL pthis, mfxMemId mid);
```

Parameters

`pthis`	Pointer to the allocator object
`mid`	Memory block ID

Description

This function de-allocates memory specified by **`mid`**.

Return Status

`MFX_ERR_NONE`	The function successfully de-allocated the memory block.
`MFX_ERR_INVALID_HANDLE`	The memory block ID is invalid.

Change History

This function is available since SDK API 1.0.

Lock

Syntax

```
mfxStatus (*Lock)(mfxHDL pthis, mfxMemId mid, mfxU8 **ptr);
```

Parameters

pthis	Pointer to the allocator object
mid	Memory block ID
ptr	Pointer to the returned linear buffer pointer

Description

This function locks the linear buffer and returns its pointer. The returned buffer must be 32-byte aligned.

Return Status

MFX_ERR_NONE	The function successfully locked the memory block.
MFX_ERR_INVALID_HANDLE	The memory block ID is invalid.
MFX_ERR_LOCK_MEMORY	The function failed to lock the linear buffer.

Change History

This function is available since SDK API 1.0.

Unlock

Syntax

```
mfxStatus (*Unlock)(mfxHDL pthis, mfxMemId mid);
```

Parameters

pthis	Pointer to the allocator object
mid	Memory block ID

Description

This function unlocks the linear buffer and invalidates its pointer.

Return Status

MFX_ERR_NONE	The function successfully unlocked the memory block.
MFX_ERR_INVALID_HANDLE	The memory block ID is invalid.

Change History

This function is available since SDK API 1.0.

mfxDecodeStat

Definition

```
typedef struct _mfxDecodeStat {
      mfxU32      reserved[16];
      mfxU32      NumFrame;
      mfxU32      NumSkippedFrame;
      mfxU32      NumError;
      mfxU32      NumCachedFrame;
} mfxDecodeStat;
```

Description

The mfxDecodeStat structure returns statistics collected during decoding.

Members

NumFrame	Number of total decoded frames
NumSkippedFrame	Number of skipped frames
NumError	Number of errors recovered
NumCachedFrame	Number of internally cached frames

Change History

This structure is available since SDK API 1.0.

mfxEncodeCtrl

Definition

```
typedef struct {
    mfxExtBuffer    Header;
    mfxU32  reserved[5];
    mfxU16  SkipFrame;

    mfxU16  QP;

    mfxU16  FrameType;
    mfxU16  NumExtParam;
    mfxU16  NumPayload;
    mfxU16  reserved2;

    mfxExtBuffer    **ExtParam;
    mfxPayload      **Payload;
} mfxEncodeCtrl;
```

Description

The **`mfxEncodeCtrl`** structure contains parameters for per-frame based encoding control.

Members

`SkipFrame`

Skip current frame. If this flag is set, i.e. it is not equal to zero, then dummy frame is encoded by the SDK. Frame where all macroblocks are encoded as skipped. Only P and B frames can be skipped, including reference one.

`QP`

If nonzero, this value overwrites the global QP value for the current frame in the constant QP mode.

`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.

`NumExtParam`

Number of extra control buffers.

`NumPayload`

Number of payload records to insert into the bitstream.

`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 usage 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.

`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.

Change History

This structure is available since SDK API 1.0. SDK API 1.1 extended the `QP` field. Since SDK API 1.3 specification of `QP` in display order mode is allowed.

mfxEncodeStat

Definition

```
typedef struct _mfxEncodeStat {
      mfxU32      reserved[16];
      mfxU32      NumFrame;
      mfxU64      NumBit;
      mfxU32      NumCachedFrame;
} mfxEncodeStat;
```

Description

The `mfxEncodeStat` structure returns statistics collected during encoding.

Members

`NumFrame`	Number of encoded frames
`NumCachedFrame`	Number of internally cached frames
`NumBit`	Number of bits for all encoded frames

Change History

This structure is available since SDK API 1.0.

mfxExtBuffer

Definition

```
typedef struct _mfxExtBuffer {
      mfxU32      BufferId;
      mfxU32      BufferSz;
} mfxExtBuffer;
```

Description

The `mfxExtBuffer` structure is the common header definition for external buffers and video processing hints.

Members

`BufferId`	Identifier of the buffer content. See the **`ExtendedBufferID`** enumerator for a complete list of extended buffers.
`BufferSz`	Size of the buffer

Change History

This structure is available since SDK API 1.0.

mfxExtAVCRefListCtrl

Definition

```
typedef struct {
    mfxExtBuffer    Header;
    mfxU16          NumRefIdxL0Active;
    mfxU16          NumRefIdxL1Active;

    struct {
        mfxU32      FrameOrder;
        mfxU16      PicStruct;
        mfxU16      ViewId;
        mfxU16      LongTermIdx;
        mfxU16      reserved[3];
    } PreferredRefList[32], RejectedRefList[16], LongTermRefList[16];
    mfxU16      ApplyLongTermIdx;
    mfxU16      reserved[15];
} mfxExtAVCRefListCtrl;
```

Description

The mfxExtAVCRefListCtrl structure configures reference frame options for the H.264 encoder. See Reference List Selection and Long-term Reference frame chapters for more details.

Not all implementations of the SDK encoder support LongTermIdx and ApplyLongTermIdx fields in this structure. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to **mfxVideoParam** structure and call **MFXVideoENCODE_Query** function. If function returns MFX_ERR_NONE and these fields were set to one, then such functionality is supported. If function fails or sets fields to zero then this functionality is not supported.

Members

Header.BufferId	Must be **MFX_EXTBUFF_AVC_REFLIST_CTRL**
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.
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.
PreferredRefList	Specify list of frames that should be used to predict the current frame.
RejectedRefList	Specify list of frames that should not be used for prediction.
LongTermRefList	Specify list of frames that should be marked as long-term reference frame.
FrameOrder PicStruct	Together these fields are used to identify reference picture. Use FrameOrder = MFX_FRAMEORDER_UNKNOWN to mark unused entry.
ViewID	Reserved and must be zero.
LongTermIdx	Index that should be used by the SDK encoder to mark long-term reference frame.
ApplyLongTermIdx	If it is equal to zero, the SDK encoder assigns long-term index according to internal algorithm. If it is equal to one, the SDK encoder uses LongTermIdx value as long-term index.

Change History

This structure is available since SDK API 1.3.

The SDK API 1.7 adds `LongTermIdx` and `ApplyLongTermIdx` fields.

mfxExtAVCRefLists

Definition

```
typedef struct {
    mfxExtBuffer    Header;
    mfxU16          NumRefIdxL0Active;
    mfxU16          NumRefIdxL1Active;
    mfxU16          reserved[2];

    struct mfxRefPic{
        mfxU32      FrameOrder;
        mfxU16      PicStruct;
        mfxU16      reserved[5];
    } RefPicList0[32], RefPicList1[32];

}mfxExtAVCRefLists;
```

Description

The `mfxExtAVCRefLists` structure specifies reference lists for the SDK encoder. It may be used together with the `mfxExtAVCRefListCtrl` structure to create customized reference lists. If both structures are used together, then the SDK encoder takes reference lists from `mfxExtAVCRefLists` structure and modifies them according to the `mfxExtAVCRefListCtrl` instructions.

Not all implementations of the SDK encoder support this structure. The application has to use query function to determine if it is supported

Members

`Header.BufferId` Must be **MFX_EXTBUFF_AVC_REFLISTS**

`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.

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.
RefPicList0, RefPicList1	Specify L0 and L1 reference lists.
FrameOrder PicStruct	Together these fields are used to identify reference picture. Use FrameOrder = MFX_FRAMEORDER_UNKNOWN to mark unused entry. Only progressive frames are supported for now.

Change History

This structure is available since SDK API 1.9.

mfxExtCodingOption

Definition

```
typedef struct {
      mfxExtBuffer      Header;

      mfxU16            reserved1;
      mfxU16            RateDistortionOpt;
      mfxU16            MECostType;
      mfxU16            MESearchType;
      mfxI16Pair        MVSearchWindow;
      mfxU16            EndOfSequence;
      mfxU16            FramePicture;

      union {
            struct      {     /* AVC */
                  mfxU16      CAVLC;
                  mfxU16      reserved2[2];
                  mfxU16      RecoveryPointSEI;
                  mfxU16      ViewOutput;
                  mfxU16      NalHrdConformance;
                  mfxU16      SingleSeiNalUnit;
                  mfxU16      VuiVclHrdParameters;
                  mfxU16      RefPicListReordering;
                  mfxU16      ResetRefList;
                  mfxU16      RefPicMarkRep;
                  mfxU16      FieldOutput;
                  mfxU16      IntraPredBlockSize;
                  mfxU16      InterPredBlockSize;
                  mfxU16      MVPrecision;
                  mfxU16      MaxDecFrameBuffering;
                  mfxU16      AUDelimiter;
                  mfxU16      EndOfStream;
                  mfxU16      PicTimingSEI;
                  mfxU16      VuiNalHrdParameters;
            };
      };
} mfxExtCodingOption;
```

Description

The `mfxExtCodingOption` structure specifies additional options for encoding.

The application can attach this extended buffer to the **`mfxVideoParam`** structure to configure initialization.

Members

`Header.BufferId`	Must be **`MFX_EXTBUFF_CODING_OPTION`**
`RateDistortionOpt`	Set this flag if rate distortion optimization is needed. See the **`CodingOptionValue`** enumerator for values of this option.
`MECostType`	Motion estimation cost type; this value is reserved and must be zero.
`MESearchType`	Motion estimation search algorithm; this value is reserved and must be zero.
`MVSearchWindow`	Rectangular size of the search window for motion estimation; this parameter is reserved and must be (0, 0).
`EndOfSequence`	Set this flag to insert the End-of-Sequence NAL. The `IdrInterval` parameter in the **`mfxInfoMFX`** structure defines a

	video sequence. See the **CodingOptionValue** enumerator for values of this option.
CAVLC	If set, CAVLC is used; if unset, CABAC is used for encoding. See the **CodingOptionValue** enumerator for values of this option.
NalHrdConformance	If set, AVC encoder produces HRD conformant bitstream. If unset AVC encoder may violate HRD conformance. See the **CodingOptionValue** enumerator for values of this option.
SingleSeiNalUnit	If set, encoder puts all SEI messages in the singe NAL unit. It includes both kinds of messages, provided by application and created by encoder. It is three states option, see **CodingOptionValue** enumerator for values of this option: UNKNOWN - put each SEI in its own NAL unit, ON - put all SEI messages in the same NAL unit, OFF - the same as unknown
VuiVclHrdParameters	If set and VBR rate control method is used then VCL HRD parameters are written in bitstream with identical to NAL HRD parameters content. See the **CodingOptionValue** enumerator for values of this option.
RefPicListReordering	Set this flag to activate reference picture list reordering; this value is reserved and must be zero.
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.
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.
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.
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 *Intel® Media SDK Reference Manual for Multi-View Video Coding* for more details about usage of this flag.
IntraPredBlockSize	Minimum block size of intra-prediction; This value is reserved and must be zero.

InterPredBlockSize

Minimum block size of inter-prediction; This value is reserved and must be zero.

MVPrecision

Specify the motion estimation precision; this parameter is reserved and must be zero.

MaxDecFrameBuffering

Specifies the maximum number of frames buffered in a DPB. A value of zero means "unspecified."

AUDelimiter

Set this flag to insert the Access Unit Delimiter NAL. See the **CodingOptionValue** enumerator for values of this option.

EndOfStream

Set this flag to insert the End of Stream NAL. See the **CodingOptionValue** enumerator for values of this option.

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.

VuiNalHrdParameters

Set this flag to insert NAL HRD parameters in the VUI header. See the **CodingOptionValue** enumerator for values of this option.

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.

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.

Change History

This structure is available since SDK API 1.0.

SDK API 1.3 adds RefPicMarkRep, FieldOutput, NalHrdConformance, SingleSeiNalUnit and VuiVclHrdParameters fields.

SDK API 1.4 adds ViewOutput field.

SDK API 1.6 adds RecoveryPointSEI field.

mfxExtCodingOption2

Definition

```
typedef struct {
    mfxExtBuffer Header;

    mfxU16      IntRefType;
    mfxU16      IntRefCycleSize;
    mfxI16      IntRefQPDelta;

    mfxU32      MaxFrameSize;
    mfxU32      MaxSliceSize;

    mfxU16      BitrateLimit;           /* tri-state option */
    mfxU16      MBBRC;                  /* tri-state option */
    mfxU16      ExtBRC;                 /* tri-state option */
    mfxU16      LookAheadDepth;
    mfxU16      Trellis;
    mfxU16      RepeatPPS;              /* tri-state option */
    mfxU16      BRefType;
    mfxU16      AdaptiveI;              /* tri-state option */
    mfxU16      AdaptiveB;              /* tri-state option */
    mfxU16      LookAheadDS;
    mfxU16      NumMbPerSlice;
    mfxU16      SkipFrame;
    mfxU8       MinQPI;                 /* 1..51, 0 = default */
    mfxU8       MaxQPI;                 /* 1..51, 0 = default */
    mfxU8       MinQPP;                 /* 1..51, 0 = default */
    mfxU8       MaxQPP;                 /* 1..51, 0 = default */
    mfxU8       MinQPB;                 /* 1..51, 0 = default */
    mfxU8       MaxQPB;                 /* 1..51, 0 = default */
    mfxU16      FixedFrameRate;         /* tri-state option */
    mfxU16      DisableDeblockingIdc;
    mfxU16      DisableVUI;
    mfxU16      reserved2[3];
} mfxExtCodingOption2;
```

Description

The `mfxExtCodingOption2` structure together with **mfxExtCodingOption** structure specifies additional options for encoding.

The application can attach this extended buffer to the **mfxVideoParam** structure to configure initialization and to the **mfxEncodeCtrl** during runtime.

Members

`Header.BufferId`

Must be **MFX_EXTBUFF_CODING_OPTION2**.

`IntRefType`

Specifies intra refresh type. The major goal of intra refresh is improvement of error resilience without significant impact on encoded bitstream size caused by I frames. The SDK encoder achieves this by encoding part of each frame in refresh cycle using intra MBs. Zero value means no refresh. One means vertical refresh, by column of MBs. This parameter is valid only during initialization.

IntRefCycleSize

Specifies number of pictures within refresh cycle starting from 2. 0 and 1 are invalid values. This parameter is valid only during initialization.

IntRefQPDelta

Specifies QP difference for inserted intra MBs. This is signed value in [-51, 51] range. This parameter is valid during initialization and runtime.

MaxFrameSize

Specify maximum encoded frame size in byte. This parameter is used in AVBR and VBR bitrate control modes and ignored in others. The SDK 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.

MaxSliceSize

Specify maximum slice size in bytes. If this parameter is specified other controls over number of slices are ignored.

Not all codecs and SDK implementations support this value. Use **Query** function to check if this feature is supported.

BitrateLimit

Turn off this flag to remove bitrate limitations imposed by the SDK encoder. This flag is intended for special usage models and usually the application should not set it. Setting this flag may lead to violation of HRD conformance and severe visual artifacts. See the **CodingOptionValue** enumerator for values of this option. The default value is ON, i.e. bitrate is limitted. This parameter is valid only during initialization.

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.

ExtBRC

Setting this flag instructs encoder to use extended bitrate control algorithms. It generally improves objective and subjective visual quality, but it also leads to violation of HRD conformance and may significantly reduce performance. See the CodingOptionValue enumerator for values of this option. The default value is OFF.

LookAheadDepth

Specifies the depth of look ahead rate control algorithm. It is the number of frames that SDK encoder analyzes before encoding. Valid value range is from 10 to 100 inclusive. To

	instruct the SDK encoder to use the default value the application should zero this field.
`Trellis`	This option is used to control trellis quantization in AVC encoder. See **`TrellisControl`** enumerator for possible values of this option. This parameter is valid only during initialization.
`RepeatPPS`	This flag controls picture parameter set repetition in AVC encoder. Turn ON this flag 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.
`BRefType`	This option controls usage of B frames as reference. See **`BRefControl`** enumerator for possible values of this option. This parameter is valid only during initialization.
`AdaptiveI`	This flag controls insertion of I frames by the SDK encoder. Turn ON this flag 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.
`AdaptiveB`	This flag controls changing of frame type from B to P. Turn ON this flag to allow such changing. 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.
`LookAheadDS`	This option controls down sampling in look ahead bitrate control mode. See **`LookAheadDownSampling`** enumerator for possible values of this option. This parameter is valid only during initialization.
`NumMbPerSlice`	This option specifies suggested slice size in number of macroblocks. The SDK can adjust this number based on platform capability. If this option is specified, i.e. if it is not equal to zero, the SDK ignores **`mfxInfoMFX::NumSlice`** parameter.
`SkipFrame`	This options enables usage of **`mfxEncodeCtrl::SkipFrame`** flag. If it is eqaul to zero, then frame skipping is disabled and **`mfxEncodeCtrl::SkipFrame`** is ignored. If it is eqaul to 1 then skipping is allowed. Not all codecs and SDK implementations support this value. Use **`Query`** function to check if this feature is supported.

MinQPI, MaxQPI
MinQPP, MaxQPP
MinQPB, MinQPB

Minimum and maximum allowed QP values for different frame types. Valid range is 1..51 inclusive. Zero means default value, i.e.no limitations on QP.

Not all codecs and SDK implementations support this value. Use **Query** function to check if this feature is supported.

FixedFrameRate

This option sets fixed_frame_rate_flag in VUI.

Not all codecs and SDK implementations support this value. Use **Query** function to check if this feature is supported.

DisableDeblockingIdc

This option disable deblocking.

Not all codecs and SDK implementations support this value. Use **Query** function to check if this feature is supported.

DisableVUI

This option completely disables VUI in output bitstream.

Not all codecs and SDK implementations support this value. Use **Query** function to check if this feature is supported.

Change History

This structure is available since SDK API 1.6.

The SDK API 1.7 added LookAheadDepth and Trellis fields.

The SDK API 1.8 adds RepeatPPS, BRefType, AdaptiveI, AdaptiveB, LookAheadDS and NumMbPerSlice fields.

The SDK API 1.9 adds MaxSliceSize, SkipFrame, MinQPI, MaxQPI, MinQPP, MaxQPP, MinQPB, MinQPB, FixedFrameRate and DisableDeblockingIdc fields.

The SDK API 1.10 adds DisableVUIfields field.

mfxExtCodingOptionSPSPPS

Definition

```
struct {
      mfxExtBuffer      Header;
      mfxU8             *SPSBuffer;
      mfxU8             *PPSBuffer;
      mfxU16            SPSBufSize;
      mfxU16            PPSBufSize;
      mfxU16            SPSId;
```

```
        mfxU16          PPSId;
    } mfxExtCodingOptionSPSPPS;
```

Description

Attach this structure as part of the **mfxVideoParam** extended buffers to configure the SDK encoder during **MFXVideoENCODE_Init**. The sequence or picture parameters specified by this structure overwrite any such parameters specified by the **mfxVideoParam** structure or any other extended buffers attached therein.

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 SDK 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 segemnt encoding feature. If this feature is not supported, the query returns **MFX_ERR_UNSUPPORTED**.

Members

Header.BufferId	Must be **MFX_EXTBUFF_CODING_OPTION_SPSPPS**.
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.
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.
SPSBufSize	Size of the SPS in bytes
PPSBufSize	Size of the PPS in bytes
SPSId	SPS identifier; the value is reserved and must be zero.
PPSId	PPS identifier; the value is reserved and must be zero.

Change History

This structure is available since SDK API 1.0.

mfxExtOpaqueSurfaceAlloc

Definition

```
typedef struct {
      mfxExtBuffer       Header;
      mfxU32             reserved1[2];
      struct {
            mfxFrameSurface1 **Surfaces;
            mfxU32       reserved2[4];
            mfxU16       Type;
            mfxU16       NumSurface;
      } In, Out;
} mfxExtOpaqueSurfaceAlloc;
```

Description

The `mfxExtOpaqueSurfaceAlloc` structure defines the opaque surface allocation information.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_OPAQUE_SURFACE_ALLOCATION**
`Type`	Surface type chosen by the application. Any valid combination of flags may be used, for example: **MFX_MEMTYPE_SYSTEM_MEMORY |** **MFX_MEMTYPE_FROM_DECODE |** **MFX_MEMTYPE_EXTERNAL_FRAME**. The SDK ignores any irrelevant flags. See the **ExtMemFrameType** enumerator for details.
`NumSurface`	The number of allocated frame surfaces.
`Surfaces`	The array pointers of allocated frame surfaces.
`In, Out`	**In** refers to surface allocation for input and **out** refers to surface allocation for output. For decoding, **In** is ignored. For encoding, **Out** is ignored.

Change History

This structure is available since SDK API 1.3.

mfxExtVideoSignalInfo

Definition

```
typedef struct {
      mfxExtBuffer      Header;
      mfxU16            VideoFormat;
      mfxU16            VideoFullRange;
      mfxU16            ColourDescriptionPresent;
      mfxU16            ColourPrimaries;
      mfxU16            TransferCharacteristics;
      mfxU16            MatrixCoefficients;
} mfxExtVideoSignalInfo;
```

Description

The `mfxExtVideoSignalInfo` structure defines the video signal information.

Members

`Header.BufferId`	Must be **`MFX_EXTBUFF_VIDEO_SIGNAL_INFO`**

`VideoFormat`
`VideoFullRange`
`ColourPrimaries`
`TransferCharacteristics`
`MatrixCoefficients`
`ColourDescriptionPresent`
	These parameters define 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.
	If **`ColourDescriptionPresent`** is zero, the color description information (including **`ColourPrimaries`**, **`TransferCharacteristics`**, and **`MatrixCoefficients`**) will/does not present in the bitstream.

Change History

This structure is available since SDK API 1.3.

mfxExtPictureTimingSEI

Definition

```
typedef struct {
  mfxExtBuffer    Header;
  mfxU32          reserved[14];

  struct {
      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;
  } TimeStamp[3];
} mfxExtPictureTimingSEI;
```

Description

The `mfxExtPictureTimingSEI` structure configures the H.264 picture timing SEI message. The encoder ignores it if HRD information in stream is absent and PicTimingSEI option in 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.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_PICTURE_TIMING_SEI**

`ClockTimestampFlag`
`CtType`
`NuitFieldBasedFlag`
`CountingType`
`FullTimestampFlag`
`DiscontinuityFlag`
`CntDroppedFlag`
`NFrames`
`SecondsFlag`
`MinutesFlag`
`HoursFlag`
`SecondsValue`
`MinutesValue`
`HoursValue`
`TimeOffset`
	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.

Change History

This structure is available since SDK API 1.3.

mfxExtAvcTemporalLayers

Definition

```
typedef struct {
    mfxExtBuffer    Header;
    mfxU32          reserved1[4];
    mfxU16          reserved2;
    mfxU16          BaseLayerPID;

    struct {
        mfxU16 Scale;
        mfxU16 reserved[3];
    } Layer[8];
} mfxExtAvcTemporalLayers;
```

Description

The `mfxExtAvcTemporalLayers` structure configures the H.264 temporal layers hierarchy. If application attaches it to the **mfxVideoParam** structure during initialization, the SDK 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.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_AVC_TEMPORAL_LAYERS**
`BaseLayerPID`	The priority ID of the base layer; the SDK encoder increases the ID for each temporal layer and writes to the prefix NAL unit.
`Scale`	The ratio between the frame rates of the current temporal layer and the base layer.
`Layer`	The array of temporal layers; Use `Scale=0` to specify absent layers.

Change History

This structure is available since SDK API 1.3.

mfxExtVppAuxData

Definition

```
typedef struct {
    mfxExtBuffer    Header;
    union{
        struct{
            mfxU32  SpatialComplexity;
            mfxU32  TemporalComplexity;
        };
        struct{
            mfxU16  PicStruct;
            mfxU16  reserved[3];
        };
    };
    mfxU16          SceneChangeRate;
    mfxU16          RepeatedFrame;
} mfxExtVppAuxData;
```

Description

The mfxExtVppAuxData structure 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.

Members

Header.BufferId	Must be **MFX_EXTBUFF_VPP_AUXDATA**
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. By default, detection is turned off and the application should explicitly enable it by using **mfxExtVPPDoUse** buffer and MFX_EXTBUFF_VPP_PICSTRUCT_DETECTION algorithm.
SpatialComplexity	Deprecated
TemporalComplexity	Deprecated
SceneChangeRate	Deprecated
RepeatedFrame	Deprecated

Change History

This structure is available since SDK API 1.0. SDK API 1.6 adds PicStruct field and deprecates SpatialComplexity, TemporalComplexity, SceneChangeRate and RepeatedFrame fields.

mfxExtVPPDenoise

Definition

```
typedef struct {
        mfxExtBuffer      Header;
        mfxU16            DenoiseFactor;
} mfxExtVppDenoise;
```

Description

The `mfxExtVPPDenoise` structure is a hint structure that configures the **VPP** denoise filter algorithm.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_VPP_DENOISE**
`DenoiseFactor`	Value of 0-100 (inclusive) indicates the level of noise to remove.

Change History

This structure is available since SDK API 1.1.

mfxExtVPPDetail

Definition

```
typedef struct {
        mfxExtBuffer      Header;
        mfxU16            DetailFactor;
} mfxExtVppDetail;
```

Description

The `mfxExtVPPDetail` structure is a hint structure that configures the **VPP** detail/edge enhancement filter algorithm.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_VPP_DETAIL**
`DetailFactor`	0-100 value (inclusive) to indicate the level of details to be enhanced.

Change History

This structure is available since SDK API 1.1.

mfxExtVPPDoNotUse

Definition

```
typedef struct {
      mfxExtBuffer      Header;
      mfxU32            NumAlg;
      mfxU32            *AlgList;
} mfxExtVPPDoNotUse;
```

Description

The `mfxExtVPPDoNotUse` structure tells the **VPP** not to use certain filters in pipeline. See "Table 4 Configurable VPP filters" for complete list of configurable filters.

The user can attach this structure to the **`mfxVideoParam`** structure when initializing video processing.

Members

`Header.BufferId`	Must be **`MFX_EXTBUFF_VPP_DONOTUSE`**
`NumAlg`	Number of filters (algorithms) not to use
`AlgList`	Pointer to a list of filters (algorithms) not to use

Change History

This structure is available since SDK API 1.0.

mfxExtVPPDoUse

Definition

```
typedef struct {
      mfxExtBuffer      Header;
      mfxU32            NumAlg;
      mfxU32            *AlgList;
} mfxExtVPPDoUse;
```

Description

The `mfxExtVPPDoUse` structure tells the **VPP** to include certain filters in pipeline.

Each filter may be included in pipeline by two different ways. First one, by adding filter ID to this structure. In this case, default filter parameters are used. Second one, by attaching filter configuration structure directly to the **mfxVideoParam** structure. In this case, adding filter ID to mfxExtVPPDoUse structure is optional. See "Table 4 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.

Members

Header.BufferId	Must be **MFX_EXTBUFF_VPP_DOUSE**
NumAlg	Number of filters (algorithms) to use
AlgList	Pointer to a list of filters (algorithms) to use

Change History

This structure is available since SDK API 1.3.

mfxExtVPPFrameRateConversion

Definition

```
typedef struct {
      mfxExtBuffer        Header;
      mfxU16              Algorithm;
      mfxU16              reserved;
      mfxU32              reserved2[15];
} mfxExtVPPFrameRateConversion;
```

Description

The mfxExtVPPFrameRateConversion structure configures the **VPP** frame rate conversion filter. The user can attach this structure to the **mfxVideoParam** structure when initializing video processing, resetting it or query its capability.

On some platforms advanced frame rate conversion algorithm, algorithm based on frame interpolation, is not supported. To query its support the application should add MFX_FRCALGM_FRAME_INTERPOLATION flag to Algorithm value in mfxExtVPPFrameRateConversion structure, attach it to **mfxVideoParam** structure and call **MFXVideoVPP_Query** function. If filter is supported the function returns MFX_ERR_NONE status and copies content of input structure to output one. If advanced filter is not supported then simple filter will be used and function returns MFX_WRN_INCOMPATIBLE_VIDEO_PARAM, copies content of input structure to output one and corrects `Algorithm` value.

If advanced FRC algorithm is not supported both **MFXVideoVPP_Init** and **MFXVideoVPP_Reset** functions returns `MFX_WRN_INCOMPATIBLE_VIDEO_PARAM` status.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION**.
`Algorithm`	See the **FrcAlgm** enumerator for a list of frame rate conversion algorithms.

Change History

This structure is available since SDK API 1.3.

mfxExtVPPProcAmp

Definition

```
typedef struct {
      mfxExtBuffer      Header;

      mfxF64      Brightness;
      mfxF64      Contrast;
      mfxF64      Hue;
      mfxF64      Saturation;
} mfxExtVPPProcAmp;
```

Description

The `mfxExtVPPProcAmp` structure is 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.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_VPP_PROCAMP**
`Brightness`	The brightness parameter is in the range of -100.0F to 100.0F, in increments of 0.1F. The default brightness value is 0.0F.
`Contrast`	The contrast parameter is in the range of 0.0F to 10.0F, in increments of 0.01F. The default contrast value is 1.0F.
`Hue`	The hue parameter is in the range of -180F to 180F, in increments of 0.1F. The default hue value is 0.0F.

`Saturation`	The saturation parameter is in the range of 0.0F to 10.0F, in increments of 0.01F. The default saturation value is 1.0F.

Change History

This structure is available since SDK API 1.1.

mfxExtVPPImageStab

Definition

```
typedef struct {
    mfxExtBuffer    Header;
    mfxU16  Mode;
    mfxU16  reserved[11];
} mfxExtVPPImageStab;
```

Description

The `mfxExtVPPImageStab` structure is 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 - by adding filter ID to **mfxExtVPPDoUse** structure or by attaching `mfxExtVPPImageStab` structure directly to the **mfxVideoParam** structure and calling **MFXVideoVPP_Query** function. If this filter is supported function returns `MFX_ERR_NONE` status and copies content of input structure to output one. If filter is not supported function returns `MFX_WRN_FILTER_SKIPPED`, removes filter form **mfxExtVPPDoUse** structure and zeroes `mfxExtVPPImageStab` structure.

If image stabilization filter is not supported, both **MFXVideoVPP_Init** and **MFXVideoVPP_Reset** functions returns `MFX_WRN_FILTER_SKIPPED` status.

The application can retrieve list of active filters by attaching **mfxExtVPPDoUse** structure to **mfxVideoParam** structure and calling **MFXVideoVPP_GetVideoParam** function. The application must allocate enough memory for filter list.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_VPP_IMAGE_STABILIZATION**
`Mode`	Specify the image stabilization mode. It should be one of the next values: **MFX_IMAGESTAB_MODE_UPSCALE** **MFX_IMAGESTAB_MODE_BOXING**

Change History

This structure is available since SDK API 1.6.

mfxExtVPPComposite

Definition

```
typedef struct mfxVPPCompInputStream {
        mfxU32  DstX;
        mfxU32  DstY;
        mfxU32  DstW;
        mfxU32  DstH;

        mfxU16  LumaKeyEnable;
        mfxU16  LumaKeyMin;
        mfxU16  LumaKeyMax;

        mfxU16  GlobalAlphaEnable;
        mfxU16  GlobalAlpha;

        mfxU16  PixelAlphaEnable;

        mfxU16  reserved2[18];
} mfxVPPCompInputStream;

typedef struct {
    mfxExtBuffer    Header;

    /* background color*/
    union {
        mfxU16   Y;
        mfxU16   R;
    };
    union {
        mfxU16   U;
        mfxU16   G;
    };
    union {
        mfxU16   V;
        mfxU16   B;
    };

    mfxU16      reserved1[24];

    mfxU16      NumInputStream;
    mfxVPPCompInputStream *InputStream;
} mfxExtVPPComposite;
```

Description

The `mfxExtVPPComposite` structure 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 CSC (RGB, YUY2, NV12)->(RGB, NV12), deinterlacing and interlaced scaling.

The VPP returns **MFX_ERR_MORE_DATA** for additional input until an output is ready. When the output is ready, VPP returns **MFX_ERR_NONE**. The application must process the output frame after synchronization.

Composition process is controlled by:

- **mfxFrameInfo::CropXYWH** in input surface- defines location of picture in the input frame,
- **InputStream[i].DstXYWH** - defines location of the cropped input picture in the output frame,
- **mfxFrameInfo::CropXYWH** in output surface - defines actual part of output frame. All pixels in output frame outside this region will be filled by specified color.

If the application uses composition process on video streams with different frame sizes, the application should provide maximum frame size in **mfxVideoParam** during initialization, reset or query operations.

If the application uses composition process, **MFXVideoVPP_QueryIOSurf** function returns cumulative number of input surfaces, i.e. number required to process all input video streams. The function sets frame size in the **mfxFrameAllocRequest** equal to the size provided by application in the **mfxVideoParam**.

Composition process supports all types of surfaces, but opaque type has next limitations:

- all input surfaces should have the same size,
- all input surfaces should have the same color format,
- all input surfaces should be described in one **mfxExtOpaqueSurfaceAlloc** structure.

All input surfaces should have the same type and color format, except per pixel alpha blending case, where it is allowed to mix NV12 and RGB surfaces.

There are three different blending use cases:

1. Luma keying. In this case, all input surfaces should have NV12 color format specified during VPP initialization. Part of each surface, including first one, may be rendered transparent by using **LumaKeyEnable**, **LumaKeyMin** and **LumaKeyMax** values.
2. Global alpha blending. In this case, all input surfaces should have the same color format specified during VPP initialization. It should be either NV12 or RGB. Each input surface, including first one, can be blended with underling surfaces by using **GlobalAlphaEnable** and **GlobalAlpha** values.
3. Per pixel alpha blending. In this case, it is allowed to mix NV12 and RGB input surfaces. Each RGB input surface, including first one, can be blended with underling surfaces by using **PixelAlphaEnable** value.

It is not allowed to mix different blending use cases in the same function call.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_VPP_COMPOSITE**

`Y, U, V`
`R, G, B`
	background color, may be changed dynamically through Reset. No default value. YUV black is (0;128;128) or (16;128;128) depending on the sample range. The SDK uses YUV or RGB triple depending on output color format.

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 number specified during initialization. Query mode 2 should be used to find maximum supported number.

InputStream

This array of mfxVPPCompInputStream structures describes composition of input video streams. It should consist of exactly NumInputStream elements.

DstX,
DstY,
DstW,
DstH

Location of input stream in output surface.

LumaKeyEnable

None 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 be used for closed captioning, for example.

LumaKeyMin,
LumaKeyMax

Minimum and maximum values of luma key, inclusive. Pixels whose luma values fit in this range are rendered transparent.

GlobalAlphaEnable

None zero value enables global alpha blending for this input stream.

GlobalAlpha

Alpha value for this stream in [0..255] range. 0 – transparent, 255 – opaque.

PixelAlphaEnable

None zero value enables per pixel alpha blending for this input stream. The stream should have RGB color format.

Change History

This structure is available since SDK API 1.8.

The SDK API 1.9 adds LumaKeyEnable, LumaKeyMin, LumaKeyMax, GlobalAlphaEnable, GlobalAlpha and PixelAlphaEnable fields.

mfxExtVPPVideoSignalInfo

Definition

```
/* TransferMatrix */
enum {
    MFX_TRANSFERMATRIX_UNKNOWN = 0,
    MFX_TRANSFERMATRIX_BT709   = 1,
    MFX_TRANSFERMATRIX_BT601   = 2
};
```

```
/* NominalRange */
enum {
    MFX_NOMINALRANGE_UNKNOWN    = 0,
    MFX_NOMINALRANGE_0_255      = 1,
    MFX_NOMINALRANGE_16_235     = 2
};

typedef struct {
    mfxExtBuffer    Header;
    mfxU16          reserved1[4];

    struct  {
        mfxU16  TransferMatrix;
        mfxU16  NominalRange;
        mfxU16  reserved2[6];
    } In, Out;
} mfxExtVPPVideoSignalInfo;
```

Description

The `mfxExtVPPVideoSignalInfo` structure is used to control transfer matrix and nominal range of YUV frames. The application should provide it during initialization. It is supported for all kinds of conversion YUV->YUV, YUV->RGB, RGB->YUV.

This structure is used by VPP only and is not compatible with **`mfxExtVideoSignalInfo`**.

Members

`Header.BufferId`	Must be **`MFX_EXTBUFF_VPP_VIDEO_SIGNAL_INFO`**
`TransferMatrix`	Transfer matrix
`NominalRange`	Nominal range

Change History

This structure is available since SDK API 1.8.

mfxExtEncoderCapability

Definition

```
typedef struct {
    mfxExtBuffer Header;

    mfxU32      MBPerSec;
    mfxU16      reserved[58];
} mfxExtEncoderCapability;
```

Description

The `mfxExtEncoderCapability` structure is used to retrive SDK encoder capability. See description of mode 4 of the **MFXVideoENCODE_Query** function for details how to use this structure.

Not all implementations of the SDK encoder support this extended buffer. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to **mfxVideoParam** structure and call **MFXVideoENCODE_Query** function. If function returns MFX_ERR_NONE then such functionality is supported.

Members

`Header.BufferId`	Must be **MFX_EXTBUFF_ENCODER_CAPABILITY**
`MBPerSec`	Specify the maximum processing rate in macro blocks per second.

Change History

This structure is available since SDK API 1.7.

mfxExtEncoderResetOption

Definition

```
typedef struct {
    mfxExtBuffer Header;

    mfxU16      StartNewSequence;
    mfxU16      reserved[11];
} mfxExtEncoderResetOption;
```

Description

The `mfxExtEncoderResetOption` structure is used to control the SDK encoder behavior during reset. By using this structure, the application instructs the SDK encoder to start new coded sequence after reset or continue encoding of current sequence.

This structure is also used in mode 3 of **MFXVideoENCODE_Query** function to check for reset outcome before actual reset. The application should set `StartNewSequence` to required behavior and call query function. If query fails, see status codes below, then such reset is not possible in current encoder state. If the application sets `StartNewSequence` to `MFX_CODINGOPTION_UNKNOWN` then query function replaces it by actual reset type: `MFX_CODINGOPTION_ON` if the SDK encoder will begin new sequence after reset or `MFX_CODINGOPTION_OFF` if the SDK encoder will continue current sequence.

Using this structure may cause next status codes from **MFXVideoENCODE_Reset** and

MFXVideoENCODE_Query functions:

- MFX_ERR_INVALID_VIDEO_PARAM - if such 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 set StartNewSequence to on and requests resolution change to bigger than initialization value.
- MFX_ERR_NONE - if such reset is possible.

There is limited list of parameters that can be changed without starting a new coded sequence:

- bitrate parameters, TargetKbps and MaxKbps in the **mfxInfoMFX** structure.
- number of slices, NumSlice in the **mfxInfoMFX** structure. Number of slices should be equal or less than number of slices during initialization.
- number of temporal layers in **mfxExtAvcTemporalLayers** structure. Reset should be called immediately before encoding of frame from base layer and number of reference frames should be big enough for new temporal layers structure.
- Quantization parameters, QPI, QPP and QPB in the **mfxInfoMFX** structure.

As it is described in Configuration Change chapter, the application should retrieve all cached frames before calling reset. When query 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 SDK encoder, then query result may differ from reset one, because the SDK encoder may insert IDR frame to produce valid coded sequence.

Not all implementations of the SDK encoder support this extended buffer. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to **mfxVideoParam** structure and call **MFXVideoENCODE_Query** function. If function returns MFX_ERR_NONE then such functionality is supported.

See also Appendix C: Streaming and Video Conferencing Features.

Members

Header.BufferId

Must be **MFX_EXTBUFF_ENCODER_RESET_OPTION**

StartNewSequence

Instructs encoder to start new sequence after reset. It is one of the **CodingOptionValue** options:

MFX_CODINGOPTION_ON – the SDK 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 SDK 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 SDK encoder may or may not start new coded sequence. This value is also used to query reset outcome.

Change History

This structure is available since SDK API 1.7.

mfxExtAVCEncodedFrameInfo

Definition

```
typedef struct {
    mfxExtBuffer    Header;

    mfxU32          FrameOrder;
    mfxU16          PicStruct;
    mfxU16          LongTermIdx;
    mfxU32          MAD;
    mfxU16          BRCPanicMode;
    mfxU16          QP;
    mfxU32          SecondFieldOffset;
    mfxU16          reserved[2];

    struct {
            mfxU32      FrameOrder;
            mfxU16      PicStruct;
            mfxU16      LongTermIdx;
            mfxU16      reserved[4];
    } UsedRefListL0[32], UsedRefListL1[32];
} mfxExtAVCEncodedFrameInfo;
```

Description

The mfxExtAVCEncodedFrameInfo is used by the SDK encoder to report additional information about encoded picture. The application can attach this buffer to the **mfxBitstream** structure before calling **MFXVideoENCODE_EncodeFrameAsync** function. For interlaced content the SDK encoder requires two such structures. They correspond to fields in encoded order.

Not all implementations of the SDK encoder support this extended buffer. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to **mfxVideoParam** structure and call **MFXVideoENCODE_Query** function. If function returns MFX_ERR_NONE then such functionality is supported.

Members

Header.BufferId	Must be **MFX_EXTBUFF_ENCODED_FRAME_INFO**
FrameOrder	Frame order of encoded picture.
PicStruct	Picture structure of encoded picture.
LongTermIdx	Long term index of encoded picture if applicable.

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.

BRCPanicMode

Bitrate control was not able to allocate enough bits for this frame. Frame quality may be unacceptably low.

QP

Luma QP.

SecondFieldOffset

Offset to second field. Second field starts at

mfxBitstream::Data + mfxBitstream::DataOffset + mfxExtAVCEncodedFrameInfo::SecondFieldOffset

UsedRefListL0
UsedRefListL1

Reference lists that have been used to encode picture.

FrameOrder

Frame order of reference picture.

PicStruct

Picture structure of reference picture.

LongTermIdx

Long term index of reference picture if applicable.

Change History

This structure is available since SDK API 1.7.

The SDK API 1.8 adds MAD and BRCPanicMode fields.

The SDK API 1.9 adds SecondFieldOffset fields.

mfxExtEncoderROI

Definition

```
typedef struct {
    mfxExtBuffer    Header;

    mfxU16  NumROI;
    mfxU16  reserved1[11];

    struct  {
        mfxU32  Left;
        mfxU32  Top;
        mfxU32  Right;
        mfxU32  Bottom;

        mfxI16  Priority;
        mfxU16  reserved2[7];
    } ROI[256];
} mfxExtEncoderROI;
```

Description

The mfxExtEncoderROI structure is used by the application to specify different Region Of Interests during encoding. It may be used at initialization or at runtime.

Members

`Header.BufferId` Must be **MFX_EXTBUFF_ENCODER_ROI**

`NumROI` Number of ROI descriptions in array. The Query function mode 2 returns maximum supported value (set it to 256 and Query will update it to maximum supported value).

`ROI` Array of ROIs. Different ROI may overlap each over. If macroblock belongs to several ROI, **Priority** from ROI with lowest index is used.

`Left,`
`Top,`
`Right,`
`Bottom` ROI location. Should be aligned to MB boundaries (should be dividable by 16). If not, the SDK encoder truncates it to MB boundaries, for example, both 17 and 31 will be truncated to 16.

`Priority` Priority of ROI.

For VBR, CBR and AVBR modes, this is relative priority of the region in the -3…3 range. Bigger value produces better quality.

For CQP mode, this is absolute value in the -51…51 range, that will be added to the MB QP. Lesser value produces better quality.

Change History

This structure is available since SDK API 1.8.

mfxExtVPPDeinterlacing

Definition

```
enum {
    MFX_DEINTERLACING_BOB      = 0x0001,
    MFX_DEINTERLACING_ADVANCED = 0x0002
};

typedef struct {
    mfxExtBuffer    Header;
    mfxU16  Mode;
    mfxU16  reserved[11];
} mfxExtVPPDeinterlacing;
```

Description

The mfxExtVPPDeinterlacing structure is used by the application to specify different deinterlacing algorithms.

Members

Header.BufferId	Must be **MFX_EXTBUFF_VPP_DEINTERLACING**
Mode	Deinterlacing algorithm MFX_DEINTERLACING_BOB or MFX_DEINTERLACING_ADVANCED .

Change History

This structure is available since SDK API 1.8.

mfxFrameAllocator

Definition

```
typedef struct {
      mfxU32      reserved[4];
      mfxHDL      pthis;
      mfxStatus   (*Alloc)(mfxHDL pthis, mfxFrameAllocRequest *request,
                  mfxFrameAllocResponse *response);
      mfxStatus   (*Lock)(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr);
      mfxStatus   (*Unlock)(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr);
      mfxStatus   (*GetHDL)(mfxHDL pthis, mfxMemId mid, mfxHDL *handle);
      mfxStatus   (*Free)(mfxHDL pthis, mfxFrameAllocResponse *response);
} mfxFrameAllocator;
```

Description

The mfxFrameAllocator structure describes the callback functions Alloc, Lock, Unlock, GetHDL and Free that the SDK implementation might use for allocating internal frames. Applications that operate on OS-specific video surfaces must implement these callback functions.

Using the default allocator implies that frame data passes in or out of SDK functions through pointers, as opposed to using memory IDs.

The SDK behavior is undefined when using an incompletely defined external allocator. See the section Memory Allocation and External Allocators for additional information.

Members

pthis	Pointer to the allocator object
Alloc	Pointer to the function that allocates frames
Lock	Pointer to the function that locks a frame and obtain its pointers
Unlock	Pointer to the function that unlocks a frame; after unlocking, any pointers to the frame are invalid.
GetHDL	Pointer to the function that obtains the OS-specific handle
Free	Pointer to the function that de-allocates a frame

Change History

This structure is available since SDK API 1.0.

Alloc

Syntax

```
mfxStatus (*Alloc)(mfxHDL pthis, mfxFrameAllocRequest *request,
mfxFrameAllocResponse *response);
```

Parameters

pthis	Pointer to the allocator object
request	Pointer to the mfxFrameAllocRequest structure that specifies the type and number of required frames
response	Pointer to the mfxFrameAllocResponse structure that retrieves frames actually allocated

Description

This function 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 SDK components must share DirectX* surfaces, this function should pass the pre-allocated surface chain to SDK instead of allocating new DirectX surfaces. See the **Error! Reference source not found.** section for additional information.

Return Status

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.

Change History

This function is available since SDK API 1.0.

Free

Syntax

```
mfxStatus (*Free)(mfxHDL pthis, mfxFrameAllocResponse *response);
```

Parameters

pthis	Pointer to the allocator object
response	Pointer to the **mfxFrameAllocResponse** structure returned by the **Alloc** function

Description

This function de-allocates all allocated frames.

Return Status

MFX_ERR_NONE	The function successfully de-allocated the memory block.

Change History

This function is available since SDK API 1.0.

Lock

Syntax

```
mfxStatus (*Lock)(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr);
```

Parameters

`pthis`	Pointer to the allocator object
`mid`	Memory block ID
`ptr`	Pointer to the returned frame structure

Description

This function locks a frame and returns its pointer.

Return Status

`MFX_ERR_NONE`	The function successfully locked the memory block.
`MFX_ERR_LOCK_MEMORY`	This function failed to lock the frame.

Change History

This function is available since SDK API 1.0.

Unlock

Syntax

```
mfxStatus (*Unlock)(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr);
```

Parameters

`pthis`	Pointer to the allocator object
`mid`	Memory block ID
`ptr`	Pointer to the frame structure; This pointer can be `NULL`.

Description

This function unlocks a frame and invalidates the specified frame structure.

Return Status

`MFX_ERR_NONE`	The function successfully unlocked the frame.

Change History

This function is available since SDK API 1.0.

GetHDL

Syntax

```
mfxStatus (*GetHDL)(mfxHDL pthis, mfxMemId mid, mfxHDL *hdl);
```

Parameters

`pthis`	Pointer to the allocator object
`mid`	Memory block ID
`hdl`	Pointer to the returned OS-specific handle

Description

This function returns the OS-specific handle associated with a video frame. If the handle is a COM interface, the reference counter must increase. The SDK will release the interface afterward.

Return Status

`MFX_ERR_NONE`	The function successfully returned the OS-specific handle.
`MFX_ERR_UNSUPPORTED`	The function does not support obtaining OS-specific handle.

Change History

This function is available since SDK API 1.0.

mfxFrameAllocRequest

Definition

```
typedef struct _mfxFrameAllocRequest {
      mfxU32            reserved[4];
      mfxFrameInfo      Info;
      mfxU16            Type;
      mfxU16            NumFrameMin;
      mfxU16            NumFrameSuggested;
      mfxU16            reserved2;
} mfxFrameAllocRequest;
```

Description

The mfxFrameAllocRequest structure 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 any case, the minimum number of frames must be at least NumFrameMin or the called function will return an error.

Members

Info	Describes the properties of allocated frames
Type	Allocated memory type; see the **ExtMemFrameType** enumerator for details.
NumFrameMin	Minimum number of allocated frames
NumFrameSuggested	Suggested number of allocated frames

Change History

This structure is available since SDK API 1.0.

mfxFrameAllocResponse

Definition

```
typedef struct _mfxFrameAllocResponse {
      mfxU32      reserved[4];
      mfxMemId    *mids;
      mfxU16      NumFrameActual;
      mfxU16      reserved2;
} mfxFrameAllocResponse;
```

Description

The mfxFrameAllocResponse structure describes the response to multiple frame allocations. The calling function returns the number of video frames actually allocated and pointers to their memory IDs.

Members

mids	Pointer to the array of the returned memory IDs; the application allocates or frees this array.

`NumFrameActual`	Number of frames actually allocated

Change History

This structure is available since SDK API 1.0.

mfxFrameData

Definition

```
typedef struct {
    mfxU32      reserved[7];
    mfxU16      reserved1;
    mfxU16      PitchHigh;

    mfxU64      TimeStamp;
    mfxU32      FrameOrder;
    mfxU16      Locked;
    union{
        mfxU16  Pitch;
        mfxU16  PitchLow;
    };

    /* color planes */
    union {
        mfxU8   *Y;
        mfxU8   *R;
    };
    union {
        mfxU8   *UV;            /* for UV merged formats */
        mfxU8   *VU;            /* for VU merged formats */
        mfxU8   *CbCr;          /* for CbCr merged formats */
        mfxU8   *CrCb;          /* for CrCb merged formats */
        mfxU8   *Cb;
        mfxU8   *U;
        mfxU8   *G;
    };
    union {
        mfxU8   *Cr;
        mfxU8   *V;
        mfxU8   *B;
    };
    mfxU8       *A;
    mfxMemId    MemId;

    /* Additional Flags */
    mfxU16  Corrupted;
    mfxU16  DataFlag;
} mfxFrameData;
```

Description

The `mfxFrameData` structure describes frame buffer pointers.

Members

`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.

`Pitch`

Deprecated.

`PitchHigh,`
`PitchLow`

Distance in bytes between the start of two consecutive rows in a frame.

`FrameOrder`

Current frame counter for the top field of the current frame; an invalid value of **`MFX_FRAMEORDER_UNKNOWN`** indicates that SDK functions that generate the frame output do not use this frame.

`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.

`Y, U, V, A;`
`R, G, B, A;`
`Y, Cr, Cb, A;`
`Y, CbCr;`
`Y, CrCb;`
`Y, UV;`

Data pointers to corresponding color channels. 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 has to specify Y, U and V pointers. For RGB32 – R, G, B and A pointers.

```
Y, VU;
```

`MemId` Memory ID of the data buffers; if any of the preceding data pointers is non-zero then the SDK ignores `MemId`.

`DataFlag` Additional flags to indicate frame data properties. See the **FrameDataFlag** enumerator for details.

`Corrupted` Some part of the frame or field pair is corrupted. See the **Corruption** enumerator for details.

Change History

This structure is available since SDK API 1.0.

SDK API 1.3 extended the `Corrupted` and `DataFlag` fields.

SDK 1.8 replaced `Pitch` by `PitchHigh` and `PitchLow` fields.

mfxFrameInfo

Definition

```
typedef struct {
    mfxU32  reserved[4];
    mfxU16  reserved4;
    mfxU16  BitDepthLuma;
    mfxU16  BitDepthChroma;
    mfxU16  Shift;

    mfxFrameId FrameId;

    mfxU32  FourCC;
    mfxU16  Width;
    mfxU16  Height;

    mfxU16  CropX;
    mfxU16  CropY;
    mfxU16  CropW;
    mfxU16  CropH;

    mfxU32  FrameRateExtN;
    mfxU32  FrameRateExtD;
    mfxU16  reserved3;

    mfxU16  AspectRatioW;
    mfxU16  AspectRatioH;

    mfxU16  PicStruct;
    mfxU16  ChromaFormat;
    mfxU16  reserved2;
} mfxFrameInfo;
```

Description

The `mfxFrameInfo` structure specifies properties of video frames. See also Appendix A: Configuration Parameter Constraints.

Members

`BitDepthLuma`

Number of bits used to represent luma samples.

Not all codecs and SDK implementations support this value. Use **`Query`** function to check if this feature is supported.

`BitDepthChroma`

Number of bits used to represent chroma samples.

Not all codecs and SDK implementations support this value. Use **`Query`** function to check if this feature is supported.

`Shift`

Shift of luma and chroma samples. Use zero value to indicate absence of shift.

Not all codecs and SDK implementations support this value. Use **`Query`** function to check if this feature is supported.

`FourCC`

FourCC code of the color format; see the **`ColorFourCC`** enumerator for details.

`Width`

`Height`

Width and height of the video frame in pixels; `Width` must be a multiple of 16. `Height` must be a multiple of 16 for progressive frame sequence and a multiple of 32 otherwise.

`CropX, CropY, CropW, CropH`

Display the region of interest of the frame; specify the display width and height in **`mfxVideoParam`**.

`AspectRatioW`

`AspectRatioH`

These parameters 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. For H.264, there is no restriction on display aspect ratio values.

If both parameters are zero, the encoder uses default value of sample aspect ratio.

`FrameRateExtN`

`FrameRateExtD`

Specify the frame rate by the 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 default to 30 frames per second.
`PicStruct`	Picture type as specified in the **PicStruct** enumerator
`ChromaFormat`	Color sampling method; the value of `ChromaFormat` is the same as that of **ChromaFormatIdc**. `ChromaFormat` is not defined if `FourCC` is zero.

Change History

This structure is available since SDK API 1.0.

SDK API 1.9 added `BitDepthLuma`, `BitDepthChroma` and `Shift` fields.

Remarks

See Appendix A for constraints of specifying certain parameters during SDK class initialization and operation.

mfxFrameSurface1

Definition

```
typedef struct {
        mfxU32              reserved[4];
        mfxFrameInfo        Info;
        mfxFrameData        Data;
} mfxFrameSurface1;
```

Description

The `mfxFrameSurface1` structure 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**.

Members

`Info`	**mfxFrameInfo** structure specifies surface properties
`Data`	**mfxFrameData** structure describes the actual frame buffer.

Change History

This structure is available since SDK API 1.0.

mfxInfoMFX

Definition

```
typedef struct mfxInfoMFX {
      mfxU32            reserved[7];
      mfxU16            reserved4;
      mfxU16            BRCParamMultiplier;
      mfxFrameInfo      FrameInfo;
      mfxU32            CodecId;
      mfxU16            CodecProfile;
      mfxU16            CodecLevel;
      mfxU16            NumThread;

      union{
            struct {    // Encoder Options
                  mfxU16      TargetUsage;
                  mfxU16      GopPicSize;
                  mfxU16      GopRefDist;
                  mfxU16      GopOptFlag;
                  mfxU16      IdrInterval;
                  mfxU16      RateControlMethod;
                  union {
                        mfxU16      InitialDelayInKB;
                        mfxU16      QPI;
                        mfxU16      Accuracy;
                  };
                  mfxU16      BufferSizeInKB;
                  union {
                        mfxU16      TargetKbps;
                        mfxU16      QPP;
                        mfxU16      ICQQuality;
                  };
                  union {
                        mfxU16      MaxKbps;
                        mfxU16      QPB;
                        mfxU16      Convergence;
                  };
                  mfxU16      NumSlice;
                  mfxU16      NumRefFrame;
                  mfxU16      EncodedOrder;
            };
            struct {
                  mfxU16      DecodedOrder;
                  mfxU16      ExtendedPicStruct;
                  mfxU16      TimeStampCalc;
                  mfxU16      SliceGroupsPresent;
                  mfxU16      reserved2[9];
            };
      } ;
} mfxInfoMFX;
```

Description

This structure specifies configurations for decoding, encoding and transcoding processes. A zero value in any of these fields indicates that the field is not explicitly specified.

Members

`BRCParamMultiplier`

Specifies a multiplier for bitrate control parameters. Affects next four variables `InitialDelayInKB`, `BufferSizeInKB`, `TargetKbps`, `MaxKbps`. If this value is not equal to zero encoder calculates BRC parameters as value * `BRCParamMultiplier`.

`FrameInfo`

mfxFrameInfo structure that specifies frame parameters

`CodecId`

Specifies the codec format identifier in the FOURCC code; see the **CodecFormatFourCC** enumerator for details. This is a mandated input parameter for **QueryIOSurf** and **Init** functions.

`CodecProfile`

Specifies the codec profile; see the **CodecProfile** enumerator for details. Specify the codec profile explicitly or the SDK functions will determine the correct profile from other sources, such as resolution and bitrate.

`CodecLevel`

Codec level; see the **CodecLevel** enumerator for details. Specify the codec level explicitly or the SDK functions will determine the correct level from other sources, such as resolution and bitrate.

`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. See Example 14 for pseudo-code that demonstrates how SDK uses this parameter.

`GopRefDist`

Distance between I- or P- key frames; if it is zero, the GOP structure is unspecified. Note: If `GopRefDist` = 1, there are no B-frames used. See Example 14 for pseudo-code that demonstrates how SDK uses this parameter.

`GopOptFlag`

ORs of the **GopOptFlag** enumerator indicate the additional flags for the GOP specification; see Example 14 for an example of pseudo-

	code that demonstrates how to use this parameter.
IdrInterval	For H.264, **IdrInterval** specifies IDR-frame interval in terms of I-frames; if **IdrInterval=0**, then every I-frame is an IDR-frame. If **IdrInterval=1**, then every other I-frame is an IDR-frame, etc. For MPEG2, **IdrInterval** defines sequence header interval in terms of I-frames. If **IdrInterval=N**, SDK inserts the sequence header before every **Nth** I-frame. If **IdrInterval=0** (default), SDK inserts the sequence header once at the beginning of the stream. If **GopPicSize** or **GopRefDist** is zero, **IdrInterval** is undefined.
TargetUsage	Target usage model that guides the encoding process; see the **TargetUsage** enumerator for details.
RateControlMethod	Rate control method; see the **RateControlMethod** enumerator for details.
InitialDelayInKB TargetKbps MaxKbps	These parameters are for the constant bitrate (CBR) and variable bitrate control (VBR) algorithms. The SDK 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 TargetKbps. (Estimate the targeted frame size by dividing the framerate by the bitrate.) The decoder starts decoding after the buffer reaches the initial size InitialDelayInKB, which is equivalent to reaching an initial delay of InitialDelayInKB*8000/TargetKbps ms. Note: 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, level, and so on.
QPI, QPP, QPB	Quantization Parameters (QP) for I, P and B frames, respectively, for the constant QP (CQP) mode.
TargetKbps Accuracy Convergence	These parameters are for 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. The **Accuracy** value is specified in the unit of tenth of percent.

	The **Convergence** value is specified in the unit of 100 frames. The **TargetKbps** value is specified in the unit of 1000 bits per second.
ICQQuality	This parameter is for Intelligent Constant Quality (ICQ) bitrate control algorithm. It is value in the 1…51 range, where 1 corresponds the best quality.
BufferSizeInKB	BufferSizeInKB represents the maximum possible size of any compressed frames.
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**.
NumRefFrame	Number of reference frames; if NumRefFrame = 0, this parameter is not specified.
EncodedOrder	If not zero, EncodedOrder specifies that **ENCODE** takes the input surfaces in the encoded order and uses explicit frame type control. The specified GOP structures are for bitrate control only.
NumThread	Deprecated; Used to represent the number of threads the underlying implementation can use on the host processor. Always set this parameter to zero.
DecodedOrder	Deprecated; Used to instruct the decoder to decoded output in the decoded order. Always set this parameter to zero.
ExtendedPicStruct	Instructs **DECODE** to output extended picture structure values for additional display attributes. See the **PicStruct** description for details.
TimeStampCalc	Time stamp calculation method; see the **TimeStampCalc** description for details.
SliceGroupsPresent	Nonzero value indicates that slice groups are present in the bitstream. Only AVC decoder uses this field.

Change History

This structure is available since SDK API 1.0.

SDK API 1.1 extended the QPI, QPP, QPB fields.

SDK API 1.3 extended the Accuracy, Convergence, TimeStampCalc, ExtendedPicStruct and BRCParamMultiplier fields.

SDK API 1.6 added SliceGroupsPresent field.

SDK API 1.8 added `ICQQuality` field.

```
mfxU16 get_gop_sequence (…) {
      pos=display_frame_order;
      if (pos == 0)
            return MFX_FRAMETYPE_I | MFX_FRAMETYPE_IDR | MFX_FRAMETYPE_REF;

      /* Only I-frames */
      If (GopPicSize == 1)
            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;
}
```

Example 14: Pseudo-Code for GOP Structure Parameters

mfxInfoVPP

Definition

```
typedef struct _mfxInfoVPP {
      mfxU32            reserved[8];
      mfxFrameInfo      In;
      mfxFrameInfo      Out;
} mfxInfoVPP;
```

Description

The `mfxInfoVPP` structure specifies configurations for video processing. A zero value in any of the fields indicates that the corresponding field is not explicitly specified.

Members

`In`	Input format for video processing
`Out`	Output format for video processing

Change History

This structure is available since SDK API 1.0.

mfxPayload

Definition

```
typedef struct {
        mfxu32      reserved[4];
        mfxu8       *Data;
        mfxu32      NumBit;
        mfxu16      Type;
        mfxu16      BufSize;
} mfxPayload;
```

Description

The `mfxPayload` structure 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.

Members

`Type`	MPEG-2 user data start code or H.264 SEI message type
`NumBit`	Number of bits in the payload data
`Data`	Pointer to the actual payload data buffer
`BufSize`	Payload buffer size in bytes

Change History

This structure is available since SDK API 1.0.

mfxVersion

Definition

```
typedef union _mfxVersion {
      struct {
            mfxU16      Minor;
            mfxU16      Major;
            };
      mfxU32      Version;
} mfxVersion;
```

Description

The `mfxVersion` structure describes the version of the SDK implementation.

Members

`Version`	SDK implementation version number
`Major`	Major number of the SDK implementation
`Minor`	Minor number of the SDK implementation

Change History

This structure is available since SDK API 1.0.

mfxVideoParam

Definition

```
typedef struct _mfxVideoParam {
      mfxU32      reserved[3];
      mfxU16      reserved3;
      mfxU16      AsyncDepth;
      union {
            mfxInfoMFX        mfx;
            mfxInfoVPP        vpp;
      }
      mfxU16            Protected;
      mfxU16            IOPattern;
      mfxExtBuffer      **ExtParam;
      mfxU16            NumExtParam;
      mfxU16            reserved2;
} mfxVideoParam;
```

Description

The `mfxVideoParam` structure contains configuration parameters for encoding, decoding, transcoding and video processing.

Members

`AsyncDepth`
Specifies how many asynchronous operations an application performs before the application explicitly synchronizes the result. If zero, the value is not specified.

`mfx`
Configurations related to encoding, decoding and transcoding; see the definition of the **`mfxInfoMFX`** structure for details.

`vpp`
Configurations related to video processing; see the definition of the **`mfxInfoVPP`** structure for details.

`Protected`
Specifies the content protection mechanism; this is a reserved parameter. Its value must be zero.

`IOPattern`
Input and output memory access types for SDK functions; see the enumerator **`IOPattern`** for details. The **Query** functions return the natively supported **`IOPattern`** if the **Query** input argument is `NULL`. This parameter is a mandated input for **QueryIOSurf** and **Init** functions. For **DECODE**, the output pattern must be specified; for **ENCODE**, the input pattern must be specified; and for **VPP**, both input and output pattern must be specified.

`NumExtParam`
The number of extra configuration structures attached to this structure.

`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, i.e.

entries of the same type. If mfxVideoParam structure is used to query the SDK capability, then list of extended buffers attached to input and output mfxVideoParam structure should be equal, i.e. should contain the same number of extended buffers of the same type.

Change History

This structure is available since SDK API 1.0. SDK API 1.1 extended the AsyncDepth field.

mfxVPPStat

Definition

```
        typedef struct _mfxVPPStat {
        mfxU32      reserved[16];
        mfxU32      NumFrame;
        mfxU32      NumCachedFrame;
} mfxVPPStat;
```

Description

The mfxVPPStat structure returns statistics collected during video processing.

Members

NumFrame	Total number of frames processed
NumCachedFrame	Number of internally cached frames

Change History

This structure is available since SDK API 1.0.

mfxENCInput

Definition

```
typedef struct _mfxENCInput mfxENCInput;

struct _mfxENCInput{
    mfxU32  reserved[32];

    mfxFrameSurface1 *InSurface;

    mfxU16  NumFrameL0;
    mfxFrameSurface1 **L0Surface;
    mfxU16  NumFrameL1;
```

```
        mfxFrameSurface1 **L1Surface;

        mfxU16  NumExtParam;
        mfxExtBuffer    **ExtParam;
    };
```

Description

The `mfxENCInput` structure specifies input for the **ENC** class of functions.

Members

`InSurface`	Input surface.

`NumFrameL0,`
`NumFrameL1`
	Number of surfaces in L0 and L1 reference lists.

`L0Surface,`
`L1Surface`
	L0 and L1 reference lists

`NumExtParam`	Number of extended buffers.
`ExtParam`	List of extended buffers.

Change History

This structure is available since SDK API 1.10.

mfxENCOutput

Definition

```
typedef struct _mfxENCOutput mfxENCOutput;

struct _mfxENCOutput{
    mfxU32  reserved[32];

    mfxU16  NumExtParam;
    mfxExtBuffer    **ExtParam;
} ;
```

Description

The `mfxENCOutput` structure specifies output of the **ENC** class of functions.

Members

`NumExtParam`	Number of extended buffers.

`ExtParam`	List of extended buffers.

Change History

This structure is available since SDK API 1.10.

mfxExtLAControl

Definition

```
typedef struct
{
    mfxExtBuffer    Header;
    mfxU16  LookAheadDepth;
    mfxU16  DependencyDepth;
    mfxU16  DownScaleFactor;

    mfxU16  reserved1[24];

    mfxU16  NumOutStream;
    struct  mfxStream{
        mfxU16  Width;
        mfxU16  Height;
        mfxU16  reserved2[14];
    } OutStream[16];
}mfxExtLAControl;
```

Description

The `mfxExtLAControl` structure is used to control standalone look ahead behavior. This LA is performed by **ENC** class of functions and its results are used later by **ENCODE** class of functions to improve coding efficiency.

This LA is intended for one to N transcoding scenario, where one input bitstream is transcoded to several output ones with different bitrates and resolutions. Usage of integrated into the SDK encoder LA in this scenario is also possible but not efficient in term of performance and memory consumption. Standalone LA by **ENC** class of functions is executed only once for input bitstream in contrast to the integrated LA where LA is executed for each of output streams.

This structure is used at **ENC** initialization time and should be attached to the **`mfxVideoParam`** structure.

Members

`Header.BufferId`	Must be **`MFX_EXTBUFF_LOOKAHEAD_CTRL`**.
`LookAheadDepth`	Look ahead depth. This parameter has exactly the same meaning as **`LookAheadDepth`** in the **`mfxExtCodingOption2`** structure.

`DependencyDepth` Dependency depth. This parameter specifies the number of frames that SDK analyzes to calculate inter-frame dependency. It should be less than **`LookAheadDepth`** filed.

`DownScaleFactor` Down scale factor. This parameter has exactly the same meaning as **`LookAheadDS`** in the **`mfxExtCodingOption2`** structure. It is recommended to execute LA on downscaled image to improve performance without significant quality degradation.

`NumOutStream` Number of output streams in one to N transcode scenario.

`OutStream` Output stream parameters.

`Width` Output stream width.

`Height` Output stream height.

Change History

This structure is available since SDK API 1.10.

mfxExtLAFrameStatistics

Definition

```
typedef struct
{
    mfxU16  Width;
    mfxU16  Height;

    mfxU32  FrameType;
    mfxU32  FrameDisplayOrder;
    mfxU32  FrameEncodeOrder;

    mfxU32  IntraCost;
    mfxU32  InterCost;
    mfxU32  DependencyCost;

    mfxU16  reserved[24];

    mfxU64 EstimatedRate[52];
}mfxLAFrameInfo;

typedef struct  {
    mfxExtBuffer    Header;

    mfxU16  reserved[20];

    mfxU16  NumAlloc;
    mfxU16  NumStream;
    mfxU16  NumFrame;
    mfxLAFrameInfo   *FrameStat;

    mfxFrameSurface1 *OutSurface;

} mfxExtLAFrameStatistics;
```

Description

The `mfxExtLAFrameStatistics` structure is used to pass standalone look ahead statistics to the SDK encoder in one to N transcode scenario. This structure is used at runtime and should be attached to the **`mfxENCOutput`** structure and then passed, attached, to the **`mfxEncodeCtrl`** structure.

Members

`Header.BufferId`	Must be **`MFX_EXTBUFF_LOOKAHEAD_STAT`**.
`NumAlloc`	Number of allocated elements in the **`FrameStat`** array.
`NumStream`	Number of streams in the **`FrameStat`** array.
`NumFrame`	Number of frames for each stream in the **`FrameStat`** array.
`FrameStat`	LA statistics for each frame in output stream.
`Width`	Output stream width.
`Height`	Output stream height.
`FrameType`	Output frame type.
`FrameDisplayOrder`	Output frame number in display order.
`FrameEncodeOrder`	Output frame number in encoding order.
`IntraCost`	Intra cost of output frame.
`InterCost`	Inter cost of output frame.
`DependencyCost`	Aggregated dependency cost. It shows how this frame influences subsequent frames.
`EstimatedRate`	Estimated rate for each QP.
`OutSurface`	Output surface.

Change History

This structure is available since SDK API 1.10.

Enumerator Reference

BitstreamDataFlag

Description

The `BitstreamDataFlag` enumerator uses bit-ORed values to itemize additional information about the bitstream buffer.

Name/Description

`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.

`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.

Change History

This enumerator is available since SDK API 1.0.

SDK API 1.6 adds `MFX_BITSTREAM_EOS` definition.

ChromaFormatIdc

Description

The `ChromaFormatIdc` enumerator itemizes color-sampling formats.

Name/Description

`MFX_CHROMAFORMAT_MONOCHROME`	Monochrome
`MFX_CHROMAFORMAT_YUV420`	4:2:0 color
`MFX_CHROMAFORMAT_YUV422`	4:2:2 color
`MFX_CHROMAFORMAT_YUV444`	4:4:4 color
`MFX_CHROMAFORMAT_YUV400`	equal to monochrome

MFX_CHROMAFORMAT_YUV411	4:1:1 color
MFX_CHROMAFORMAT_YUV422H	4:2:2 color, horizontal subsampling. It is equal to 4:2:2 color.
MFX_CHROMAFORMAT_YUV422V	4:2:2 color, vertical subsampling

Change History

This enumerator is available since SDK API 1.0.

SDK API 1.4 adds MFX_CHROMAFORMAT_YUV400, MFX_CHROMAFORMAT_YUV411, MFX_CHROMAFORMAT_YUV422H and MFX_CHROMAFORMAT_YUV422V definitions.

CodecFormatFourCC

Description

The CodecFormatFourCC enumerator itemizes codecs in the FourCC format.

Name/Description

MFX_CODEC_AVC	AVC, H.264, or MPEG-4, part 10 codec
MFX_CODEC_MPEG2	MPEG-2 codec
MFX_CODEC_VC1	VC-1 codec
MFX_CODEC_HEVC	HEVC codec

Change History

This enumerator is available since SDK API 1.0.

SDK API 1.8 added MFX_CODEC_HEVC definition.

CodecLevel

Description

The CodecLevel enumerator itemizes codec levels for all codecs.

Name/Description

MFX_LEVEL_UNKNOWN	Unspecified codec level
MFX_LEVEL_AVC_1 MFX_LEVEL_AVC_1b MFX_LEVEL_AVC_11 MFX_LEVEL_AVC_12 MFX_LEVEL_AVC_13	H.264 level 1-1.3
MFX_LEVEL_AVC_2 MFX_LEVEL_AVC_21 MFX_LEVEL_AVC_22	H.264 level 2-2.2
MFX_LEVEL_AVC_3 MFX_LEVEL_AVC_31 MFX_LEVEL_AVC_32	H.264 level 3-3.2
MFX_LEVEL_AVC_4 MFX_LEVEL_AVC_41 MFX_LEVEL_AVC_42	H.264 level 4-4.2
MFX_LEVEL_AVC_5 MFX_LEVEL_AVC_51 MFX_LEVEL_AVC_52	H.264 level 5-5.2
MFX_LEVEL_MPEG2_LOW MFX_LEVEL_MPEG2_MAIN MFX_LEVEL_MPEG2_HIGH MFX_LEVEL_MPEG2_HIGH1440	MPEG-2 levels
MFX_LEVEL_VC1_LOW MFX_LEVEL_VC1_MEDIAN MFX_LEVEL_VC1_HIGH	VC-1 Level Low (simple & main profiles)
MFX_LEVEL_VC1_0 MFX_LEVEL_VC1_1 MFX_LEVEL_VC1_2 MFX_LEVEL_VC1_3 MFX_LEVEL_VC1_4	VC-1 advanced profile levels
MFX_LEVEL_HEVC_1 MFX_LEVEL_HEVC_2 MFX_LEVEL_HEVC_21 MFX_LEVEL_HEVC_3 MFX_LEVEL_HEVC_31 MFX_LEVEL_HEVC_4 MFX_LEVEL_HEVC_41 MFX_LEVEL_HEVC_5 MFX_LEVEL_HEVC_51 MFX_LEVEL_HEVC_52 MFX_LEVEL_HEVC_6 MFX_LEVEL_HEVC_61 MFX_LEVEL_HEVC_62	HEVC levels and tiers
MFX_TIER_HEVC_MAIN MFX_TIER_HEVC_HIGH	

Change History

This enumerator is available since SDK API 1.0.

SDK API 1.8 added HEVC level and tier definitions.

CodecProfile

Description

The `CodecProfile` enumerator itemizes codec profiles for all codecs.

Name/Description

`MFX_PROFILE_UNKNOWN`	Unspecified profile

`MFX_PROFILE_AVC_BASELINE`
`MFX_PROFILE_AVC_MAIN`
`MFX_PROFILE_AVC_EXTENDED`
`MFX_PROFILE_AVC_HIGH`
`MFX_PROFILE_AVC_CONSTRAINED_BASELINE`
`MFX_PROFILE_AVC_CONSTRAINED_HIGH`
`MFX_PROFILE_AVC_PROGRESSIVE_HIGH`
	H.264 profiles

`MFX_PROFILE_AVC_CONSTRAINT_SET0`
`MFX_PROFILE_AVC_CONSTRAINT_SET1`
`MFX_PROFILE_AVC_CONSTRAINT_SET2`
`MFX_PROFILE_AVC_CONSTRAINT_SET3`
`MFX_PROFILE_AVC_CONSTRAINT_SET4`
`MFX_PROFILE_AVC_CONSTRAINT_SET5`
	Combined with H.264 profile these flags impose additional constrains. See H.264 specification for the list of constrains.

`MFX_PROFILE_MPEG2_SIMPLE`
`MFX_PROFILE_MPEG2_MAIN`
`MFX_PROFILE_MPEG2_HIGH`
	MPEG-2 profiles

`MFX_PROFILE_VC1_SIMPLE`
`MFX_PROFILE_VC1_MAIN`
`MFX_PROFILE_VC1_ADVANCED`
	VC-1 profiles

`MFX_PROFILE_HEVC_MAIN`
`MFX_PROFILE_HEVC_MAIN10`
`MFX_PROFILE_HEVC_MAINSP`
	HEVC profiles

Change History

This enumerator is available since SDK API 1.0.
SDK API 1.3 adds `MFX_PROFILE_AVC_EXTENDED`.
SDK API 1.4 adds `MFX_PROFILE_AVC_CONSTRAINED_BASELINE`, `MFX_PROFILE_AVC_CONSTRAINED_HIGH`, `MFX_PROFILE_AVC_PROGRESSIVE_HIGH` and six

constrained flags MFX_PROFILE_AVC_CONSTRAINT_SET.
SDK API 1.8 added HEVC profile definitions.

CodingOptionValue

Description

The CodingOptionValue enumerator defines a three-state coding option setting.

Name/Description

MFX_CODINGOPTION_UNKNOWN	Unspecified
MFX_CODINGOPTION_ON	Coding option set
MFX_CODINGOPTION_OFF	Coding option not set
MFX_CODINGOPTION_ADAPTIVE	Reserved

Change History

This enumerator is available since SDK API 1.0.

SDK API 1.6 adds MFX_CODINGOPTION_ADAPTIVE option.

ColorFourCC

Description

The ColorFourCC enumerator itemizes color formats.

Name/Description

MFX_FOURCC_YV12	YV12 color planes
MFX_FOURCC_NV12	NV12 color planes
MFX_FOURCC_RGB4	RGB4 (RGB32) color planes
MFX_FOURCC_YUY2	YUY2 color planes
MFX_FOURCC_P8	Internal SDK color format. The application should use one of the functions below to create such surface, depending on Direct3D version.

	Direct3D9
	IDirectXVideoDecoderService::CreateSurface()
	Direct3D11
	ID3D11Device::CreateBuffer()

`MFX_FOURCC_P8_TEXTURE`	Internal SDK color format. The application should use one of the functions below to create such surface, depending on Direct3D version. Direct3D9 IDirectXVideoDecoderService::CreateSurface() Direct3D11 ID3D11Device::CreateTexture2D()
`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.
`MFX_FOURCC_BGR4`	ABGR color format. It is similar to MFX_FOURCC_RGB4 but with interchanged R and B channels. 'A' is 8 MSBs, then 8 bits for 'B' channel, then 'G' and 'R' channels.
`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.
`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.
`MFX_FOURCC_R16`	16 bits single channel color format. This format should be mapped to DXGI_FORMAT_R16_TYPELESS or D3DFMT_R16F.

Change History

This enumerator is available since SDK API 1.0.

The SDK API 1.1 adds `MFX_FOURCC_P8`.

The SDK API 1.6 adds `MFX_FOURCC_P8_TEXTURE`.

The SDK API 1.9 adds `MFX_FOURCC_P010`, `MFX_FOURCC_BGR4`, `MFX_FOURCC_A2RGB10`, `MFX_FOURCC_ARGB16` and `MFX_FOURCC_R16`.

Corruption

Description

The `Corruption` enumerator itemizes the decoding corruption types. It is a bit-OR'ed value of the following.

Name/Description

`MFX_CORRUPTION_MINOR`	Minor corruption in decoding certain macro-blocks
`MFX_CORRUPTION_MAJOR`	Major corruption in decoding the frame
`MFX_CORRUPTION_REFERENCE_FRAME`	Decoding used a corrupted reference frame.
`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.
`MFX_CORRUPTION_ABSENT_TOP_FIELD`	Top field of frame is absent in bitstream. Only bottom field has been decoded.
`MFX_CORRUPTION_ABSENT_BOTTOM_FIELD`	Bottom field of frame is absent in bitstream. Only top filed has been decoded.

Change History

This enumerator is available since SDK API 1.3. The SDK API 1.6 added MFX_CORRUPTION_ABSENT_TOP_FIELD and MFX_CORRUPTION_ABSENT_BOTTOM_FIELD definitions.

ExtendedBufferID

Description

The `ExtendedBufferID` enumerator itemizes and defines identifiers (`BufferId`) for extended buffers or video processing algorithm identifiers.

Name/Description

`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.
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.
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 **mfxVideoParam** structure for encoding initialization.
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.
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 **mfxVideoParam** structure for encoding initialization.
MFX_EXTBUFF_ENCODED_FRAME_INFO	This extended buffer is used by the SDK 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.
MFX_EXTBUFF_ENCODER_CAPABILITY	This extended buffer is used to retrive SDK encoder capability. See the **mfxExtEncoderCapability** structure for details. The application can attach this buffer to the **mfxVideoParam** structure before calling **MFXVideoENCODE_Query** function.
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.
MFX_EXTBUFF_OPAQUE_SURFACE_ALLOCATION	This extended buffer defines opaque surface allocation information. See the **mfxExtOpaqueSurfaceAlloc** structure for details. The application can attach this buffer to decoding, encoding, or video processing initialization.
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.

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.
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.
MFX_EXTBUFF_VPP_DENOISE	The extended buffer defines control parameters for the **VPP** denoise filter algorithm. See the mfxExtVPPDenoise structure for details. The application can attach this buffer to the mfxVideoParam structure for video processing initialization.
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 mfxVideoParam structure for video processing initialization.
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.
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 mfxVideoParam structure for video processing initialization.
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.
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.
MFX_EXTBUFF_VPP_PICSTRUCT_DETECTION	This video processor algorithm enables detection of picture structure. See PicStruct variable in mfxExtVppAuxData structure for more details.
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.

MFX_EXTBUFF_VPP_SCENE_CHANGE	Deprecated.

Change History

This enumerator is available since SDK API 1.0.

SDK API 1.6 adds MFX_EXTBUFF_VPP_IMAGE_STABILIZATION, MFX_EXTBUFF_VPP_PICSTRUCT_DETECTION, MFX_EXTBUFF_CODING_OPTION2 and deprecates MFX_EXTBUFF_VPP_SCENE_CHANGE.

SDK API 1.7 adds MFX_EXTBUFF_ENCODED_FRAME_INFO, MFX_EXTBUFF_ENCODER_CAPABILITY, MFX_EXTBUFF_ENCODER_RESET_OPTION.

See additional change history in the structure definitions.

ExtMemBufferType

Description

The ExtMemBufferType enumerator specifies the buffer type. It is a bit-ORed value of the following.

Name/Description

MFX_MEMTYPE_PERSISTENT_MEMORY	Memory page for persistent use

Change History

This enumerator is available since SDK API 1.0.

ExtMemFrameType

Description

The ExtMemFrameType enumerator specifies the memory type of frame. It is a bit-ORed value of the following. For information on working with video memory surfaces, see the section Working with hardware acceleration.

Name/Description

MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET	Frames are in video memory and belong to video decoder render targets.

`MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET`	Frames are in video memory and belong to video processor render targets.
`MFX_MEMTYPE_SYSTEM_MEMORY`	The frames are in system memory.
`MFX_MEMTYPE_FROM_ENCODE`	Allocation request comes from an **ENCODE** function
`MFX_MEMTYPE_FROM_DECODE`	Allocation request comes from a **DECODE** function
`MFX_MEMTYPE_FROM_VPPIN`	Allocation request comes from a **VPP** function for input frame allocation
`MFX_MEMTYPE_FROM_VPPOUT`	Allocation request comes from a **VPP** function for output frame allocation
`MFX_MEMTYPE_INTERNAL_FRAME`	Allocation request for internal frames
`MFX_MEMTYPE_EXTERNAL_FRAME`	Allocation request for I/O frames
`MFX_MEMTYPE_OPAQUE_FRAME`	Allocation request for opaque frames

Remarks

The application may use macro `MFX_MEMTYPE_BASE` to extract the base memory types, one of `MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET`, `MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET`, and `MFX_MEMTYPE_SYSTEM_MEMORY`.

Change History

This enumerator is available since SDK API 1.0. SDK API 1.3 extended the `MFX_MEMTYPE_OPAQUE_FRAME` definition and the `MFX_MEMTYPE_BASE` macro definition.

FrameDataFlag

Description

The `FrameDataFlag` enumerator uses bit-ORed values to itemize additional information about the frame buffer.

Name/Description

`MFX_FRAMEDATA_ORIGINAL_TIMESTAMP`	Indicates the time stamp of this frame is not calculated and is a pass-through of the original time stamp.

Change History

This enumerator is available since SDK API 1.3.

FrameType

Description

The `FrameType` enumerator itemizes frame types. Use bit-ORed values to specify all that apply.

Name/Description

`MFX_FRAMETYPE_I`	This frame or the first field is encoded as an I frame/field.
`MFX_FRAMETYPE_P`	This frame or the first field is encoded as a P frame/field.
`MFX_FRAMETYPE_B`	This frame or the first field is encoded as a B frame/field.
`MFX_FRAMETYPE_S`	This frame or the first field is either an SI- or SP-frame/field.
`MFX_FRAMETYPE_REF`	This frame or the first field is encoded as a reference.
`MFX_FRAMETYPE_IDR`	This frame or the first field is encoded as an IDR.
`MFX_FRAMETYPE_xI`	The second field is encoded as an I-field.
`MFX_FRAMETYPE_xP`	The second field is encoded as a P-field.
`MFX_FRAMETYPE_xB`	The second field is encoded as a B-field.
`MFX_FRAMETYPE_xS`	The second field is an SI- or SP-field.
`MFX_FRAMETYPE_xREF`	The second field is encoded as a reference.
`MFX_FRAMETYPE_xIDR`	The second field is encoded as an IDR.

Change History

This enumerator is available since SDK API 1.0. SDK API 1.3 extended the second field types.

FrcAlgm

Description

The FrcAlgm enumerator itemizes frame rate conversion algorithms. See description of **mfxExtVPPFrameRateConversion** structure for more details.

Name/Description

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**.

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.

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.

Change History

This enumerator is available since SDK API 1.3.

GopOptFlag

Description

The GopOptFlag enumerator itemizes special properties in the GOP (Group of Pictures) sequence.

Name/Description

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. See Appendix C: Long-term Reference frame for more details.

`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.

Change History

This enumerator is available since SDK API 1.0.

IOPattern

Description

The `IOPattern` enumerator itemizes memory access patterns for SDK functions. Use bit-ORed values to specify an input access pattern and an output access pattern.

Name/Description

`MFX_IOPATTERN_IN_VIDEO_MEMORY`

Input to SDK functions is a video memory surface

`MFX_IOPATTERN_IN_SYSTEM_MEMORY`

Input to SDK functions is a linear buffer directly in system memory or in system memory through an external allocator

`MFX_IOPATTERN_IN_OPAQUE_MEMORY`

Input to SDK functions maps at runtime to either a system memory buffer or a video memory surface.

`MFX_IOPATTERN_OUT_VIDEO_MEMORY`

Output to SDK functions is a video memory surface

`MFX_IOPATTERN_OUT_SYSTEM_MEMORY`

Output to SDK functions is a linear buffer directly in system memory or in system memory through an external allocator

`MFX_IOPATTERN_OUT_OPAQUE_MEMORY`

Output to SDK functions maps at runtime to either a system memory buffer or a video memory surface.

Change History

This enumerator is available since SDK API 1.0. SDK API 1.3 extended the `MFX_IOPATTERN_IN_OPAQUE_MEMORY` and `MFX_IOPATTERN_OUT_OPAQUE_MEMORY` definitions.

mfxHandleType

Description

The mfxHandleType enumerator itemizes system handle types that SDK implementations might use.

Name/Description

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.
MFX_HANDLE_D3D11_DEVICE	Pointer to the **ID3D11Device** interface. See Working with Microsoft* DirectX* Applications for more details on how to use this handle.
MFX_HANDLE_VA_DISPLAY	Pointer to VADisplay interface. See Working with VA API Applications for more details on how to use this handle.
MFX_HANDLE_ENCODE_CONTEXT	Pointer to VAContextID interface. It represents encoder context.

Change History

This enumerator is available since SDK API 1.0.

SDK API 1.4 added MFX_HANDLE_D3D11_DEVICE definition.

SDK API 1.8 added MFX_HANDLE_VA_DISPLAY and MFX_HANDLE_ENCODE_CONTEXT definitions.

mfxIMPL

Description

The mfxIMPL enumerator itemizes SDK implementation types. The implementation type is a bit OR'ed value of the base type and any decorative flags.

Name/Description

MFX_IMPL_AUTO	Find the best SDK implementation automatically. It includes either hardware-accelerated implementation on the default acceleration device or software implementation.

This value is obsolete and it is recommended to use `MFX_IMPL_AUTO_ANY` instead.

`MFX_IMPL_SOFTWARE` Use the software implementation

`MFX_IMPL_HARDWARE` Use the hardware-accelerated implementation on the default acceleration device

`MFX_IMPL_RUNTIME` This value cannot be used for session initialization. It may be returned by **`MFXQueryIMPL`** function to show that session has been initialized in run time mode.

`MFX_IMPL_UNSUPPORTED` Failed to locate the desired SDK implementation

If the acceleration device is not default device, use the following values to initialize the SDK libraries on an alternative acceleration device.

`MFX_IMPL_AUTO_ANY` Find the SDK implementation on any acceleration device including the default acceleration device and the SDK software library.

`MFX_IMPL_HARDWARE_ANY` Find the hardware-accelerated implementation on any acceleration device including the default acceleration device.

`MFX_IMPL_HARDWARE2` Use the hardware-accelerated implementation on the second acceleration device.

`MFX_IMPL_HARDWARE3` Use the hardware-accelerated implementation on the third acceleration device.

`MFX_IMPL_HARDWARE4` Use the hardware-accelerated implementation on the fourth acceleration device.

Use the following decorative flags to specify the OS infrastructure that hardware acceleration should base on.

`MFX_IMPL_VIA_D3D9` Hardware acceleration goes through the Microsoft* Direct3D9* infrastructure.

`MFX_IMPL_VIA_D3D11` Hardware acceleration goes through the Microsoft* Direct3D11* infrastructure.

`MFX_IMPL_VIA_VAAPI` Hardware acceleration goes through the Linux* VA API infrastructure.

`MFX_IMPL_VIA_ANY` Hardware acceleration can go through any supported OS infrastructure. This is default value, it is used by the SDK if none of `MFX_IMPL_VIA_xxx` flag is specified by application.

`MFX_IMPL_AUDIO`	Load audio library. It can be used only together with `MFX_IMPL_SOFTWARE`, any other combinations lead to error.

Change History

This enumerator is available since SDK API 1.0.

SDK API 1.1 added support of multiple devices.

SDK API 1.3 added support of OS infrastructure definitions.

SDK API 1.6 changed defauls OS infrustructure from `MFX_IMPL_VIA_D3D9` to `MFX_IMPL_VIA_ANY`.

SDK API 1.8 added support of `MFX_IMPL_AUDIO` and `MFX_IMPL_VIA_VAAPI`.

Remarks

The application can use the macro **`MFX_IMPL_BASETYPE(x)`** to obtain the base implementation type.

It is recommended that the application use **`MFX_IMPL_VIA_ANY`** if the application uses system memory or opaque surface for I/O exclusively.

mfxPriority

Description

The `mfxPriority` enumerator describes the session priority.

Name/Description

`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.
`MFX_PRIORITY_NORMAL`	Normal priority: the session operation is halted if there are high priority tasks.
`MFX_PRIORITY_HIGH`	High priority: the session operation blocks other lower priority session operations.

Change History

This enumerator is available since SDK API 1.1.

mfxSkipMode

Description

The `mfxSkipMode` enumerator describes the decoder skip-mode options.

Name/Description

`MFX_SKIPMODE_NONE`	Do not skip any frames.
`MFX_SKIPMODE_MORE`	Skip more frames.
`MFX_SKIPMODE_LESS`	Skip less frames.

Change History

This enumerator is available since SDK API 1.0.

mfxStatus

Description

The `mfxStatus` enumerator itemizes status codes returned by SDK functions.

When an SDK function returns an error status code, it generally expects a **Reset** or **Close** function to follow, (with the exception of **MFX_ERR_MORE_DATA** and **MFX_ERR_MORE_SURFACE** for asynchronous operation considerations) See section Decoding Procedures, section Encoding Procedures, and section Video Processing Procedures for more information about recovery procedures.

When an SDK function returns a warning status code, the function has performed necessary operations to continue the operation without interruption. In this case, the output might be unreliable. The application must check the validity of the output generated by the function.

Name/Description

Successful operation

`MFX_ERR_NONE`	No error

Reserved status code

`MFX_ERR_UNKNOWN`	An unknown error occurred in the library function

	operation. This is a reserved status code.
Programming related errors	
`MFX_ERR_NOT_INITIALIZED`	Member functions called without initialization.
`MFX_ERR_INVALID_HANDLE`	Invalid session or MemId handle
`MFX_ERR_NULL_PTR`	`NULL` pointer in the input or output arguments
`MFX_ERR_UNDEFINED_BEHAVIOR`	The behavior is undefined.
`MFX_ERR_NOT_ENOUGH_BUFFER`	Insufficient buffer for input or output.
`MFX_ERR_NOT_FOUND`	Specified object/item/sync point not found.
Memory related errors	
`MFX_ERR_MEMORY_ALLOC`	Failed to allocate memory.
`MFX_ERR_LOCK_MEMORY`	Failed to lock the memory block (external allocator).
Configuration related errors or warnings	
`MFX_ERR_UNSUPPORTED`	Unsupported configurations, parameters, or features
`MFX_ERR_INVALID_VIDEO_PARAM`	Invalid video parameters detected. **Init** and **Reset** functions return this status code to indicate either that mandated input parameters are unspecified, or the functions failed to correct them.
`MFX_ERR_INCOMPATIBLE_VIDEO_PARAM`	Incompatible video parameters detected. If a **Reset** function returns this status code, a component—decoder, encoder or video preprocessor—cannot process the specified configuration with existing structures and frame buffers. If the function **MFXVideoDECODE_DecodeFrameAsync** returns this status code, the bitstream contains an incompatible video parameter configuration that the decoder cannot follow.
`MFX_WRN_VIDEO_PARAM_CHANGED`	The decoder detected a new sequence header in the bitstream. Video parameters may have changed.
`MFX_WRN_VALUE_NOT_CHANGED`	The parameter has been clipped to its value range.
`MFX_WRN_OUT_OF_RANGE`	The parameter is out of valid value range.
`MFX_WRN_INCOMPATIBLE_VIDEO_PARAM`	Incompatible video parameters detected. SDK functions return this status code to indicate that there was incompatibility in the specified parameters

	and has resolved it.
`MFX_WRN_FILTER_SKIPPED`	The SDK VPP has skipped one or more optional filters requested by the application. To retrieve actual list of filters attach **`mfxExtVPPDoUse`** to **`mfxVideoParam`** and call **`MFXVideoVPP_GetVideoParam`**. The application must ensure that enough memory is allocated for filter list.

Asynchronous operation related errors or warnings

`MFX_ERR_ABORTED`	The asynchronous operation aborted.
`MFX_ERR_MORE_DATA`	Need more bitstream at decoding input, encoding input, or video processing input frames.
`MFX_ERR_MORE_SURFACE`	Need more frame surfaces at decoding or video processing output
`MFX_ERR_MORE_BITSTREAM`	Need more bitstream buffers at the encoding output
`MFX_WRN_IN_EXECUTION`	Synchronous operation still running

Hardware device related errors or warnings

`MFX_ERR_DEVICE_FAILED`	Hardware device returned unexpected errors. SDK was unable to restore operation. See section **Error! eference source not found.** and section *Hardware Device Error Handling* for more information.
`MFX_ERR_DEVICE_LOST`	Hardware device was lost; See the *Hardware Device Error Handling* section for further information.
`MFX_WRN_DEVICE_BUSY`	Hardware device is currently busy. Call this function again in a few milliseconds.
`MFX_WRN_PARTIAL_ACCELERATION`	The hardware does not support the specified configuration. Encoding, decoding, or video processing may be partially accelerated. Only SDK HW implementation may return this status code.

Change History

This enumerator is available since SDK API 1.0. SDK API 1.3 added the `MFX_ERR_MORE_BITSTREAM` return status. SDK API 1.6 added the `MFX_WRN_FILTER_SKIPPED` return status.

PicStruct

Description

The `PicStruct` enumerator itemizes picture structure. Use bit-OR'ed values to specify the desired picture type.

Name/Description

`MFX_PICSTRUCT_UNKNOWN`	Unspecified or mixed progressive/interlaced pictures
`MFX_PICSTRUCT_PROGRESSIVE`	Progressive picture
`MFX_PICSTRUCT_FIELD_TFF`	Top field in first interlaced picture
`MFX_PICSTRUCT_FIELD_BFF`	Bottom field in first interlaced picture
`MFX_PICSTRUCT_FIELD_REPEATED`	First field repeated: `pict_struct = 5 or 6` in H.264
`MFX_PICSTRUCT_FRAME_DOUBLING`	Double the frame for display: `pict_struct = 7` in H.264
`MFX_PICSTRUCT_FRAME_TRIPLING`	Triple the frame for display: `pict_struct = 8` in H.264

Change History

This enumerator is available since SDK API 1.0. SDK API 1.3 added support of combined display attributes.

Remarks

It is possible to combine the above picture structure values to indicate additional display attributes. If `ExtendedPicStruct` in the **mfxInfoMFX** structure is true, **DECODE** outputs extended picture structure values to indicate how to display an output frame as shown in the following table:

Extended `PicStruct` Values	**Description**
`MFX_PICSTRUCT_PROGRESSIVE|` `MFX_PICSTRUCT_FRAME_DOUBLING`	The output frame is progressive; Display as two identical progressive frames.
`MFX_PICSTRUCT_PROGRESSIVE|` `MFX_PICSTRUCT_FRAME_TRIPLING`	The output frame is progressive; Display as three identical progressive frames.
`MFX_PICSTRUCT_PROGRESSIVE|` `MFX_PICSTRUCT_FIELD_TFF`	The output frame is progressive; Display as two fields, top field first.

MFX_PICSTRUCT_PROGRESSIVE| MFX_PICSTRUCT_FIELD_BFF	The output frame is progressive; Display as two fields, bottom field first
MFX_PICSTRUCT_PROGRESSIVE| MFX_PICSTRUCT_FIELD_TFF| MFX_PICSTRUCT_FIELD_REPEATED	The output frame is progressive; Display as three fields: top, bottom, and top.
MFX_PICSTRUCT_PROGRESSIVE| MFX_PICSTRUCT_FIELD_BFF| MFX_PICSTRUCT_FIELD_REPEATED	The output frame is progressive; Display as three fields: bottom, top, bottom.

In the above cases, **VPP** processes the frame as a progressive frame and passes the extended picture structure values from input to output. **ENCODE** encodes the frame as a progressive frame and marks the bitstream header properly according to the extended picture structure values.

RateControlMethod

Description

The `RateControlMethod` enumerator itemizes bitrate control methods.

Name/Description

MFX_RATECONTROL_CBR	Use the constant bitrate control algorithm
MFX_RATECONTROL_VBR	Use the variable bitrate control algorithm
MFX_RATECONTROL_CQP	Use the constant quantization parameter algorithm
MFX_RATECONTROL_AVBR	Use the average variable bitrate control algorithm
MFX_RATECONTROL_LA	Use the VBR algorithm with look ahead. It is a special bitrate control mode in the SDK 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.

`MFX_RATECONTROL_ICQ` Use the intelligent constant quality algorithm. Quality factor is specified by **mfxInfoMFX::ICQQuality**.

`MFX_RATECONTROL_VCM` Use the video conferencing mode algorithm

`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.

`MFX_RATECONTROL_LA_EXT` Use extended look ahead rate control algorithm. It is intended for one to N transcode scenario and requires presence of **mfxExtLAFrameStatistics** structure at encoder input at runtime.

Change History

This enumerator is available since SDK API 1.0.

The SDK API 1.1 added the constant quantization parameter algorithm.

The SDK API 1.3 added the average variable bitrate control algorithm.

The SDK API 1.7 added the look ahead algorithm.

The SDK API 1.8 added the intelligent constant quality and video conferencing mode algorithms.

The SDK API 1.10 added the extended look ahead rate control algorithm.

TimeStampCalc

Description

The `TimeStampCalc` enumerator itemizes time-stamp calculation methods.

Name/Description

`MFX_TIMESTAMPCALC_UNKNOWN` The time stamp calculation is to base on the input frame rate, if time stamp is not explicitly specified.

`MFX_TIMESTAMPCALC_TELECINE` Adjust time stamp to 29.97fps on 24fps progressively encoded sequences if telecining attributes are available in the bitstream and time stamp is not explicitly specified. (The input frame rate must be specified.)

Change History

This enumerator is available since SDK API 1.3.

TargetUsage

Description

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 specified target usage is not supported, the SDK encoder will use the closest supported value.

Name/Description

MFX_TARGETUSAGE_1 MFX_TARGETUSAGE_2 MFX_TARGETUSAGE_3 MFX_TARGETUSAGE_4 MFX_TARGETUSAGE_5 MFX_TARGETUSAGE_6 MFX_TARGETUSAGE_7	Target usage
MFX_TARGETUSAGE_UNKNOWN	Unspecified target usage
MFX_TARGETUSAGE_BEST_QUALITY	Best quality, mapped to MFX_TARGETUSAGE_1
MFX_TARGETUSAGE_BALANCED	Balanced quality and speed, mapped to MFX_TARGETUSAGE_4
MFX_TARGETUSAGE_BEST_SPEED	Fastest speed, mapped to MFX_TARGETUSAGE_7

Change History

This enumerator is available since SDK API 1.0.

The SDK API 1.7 adds MFX_TARGETUSAGE_1 .. MFX_TARGETUSAGE_7 values.

TrellisControl

Description

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.

Due to performance reason on some target usages trellis quantization is always turned off and this control is ignored by the SDK encoder.

Name/Description

`MFX_TRELLIS_UNKNOWN`	Default value, it is up to the SDK encoder to turn trellis quantization on or off.
`MFX_TRELLIS_OFF`	Turn trellis quantization off for all frame types.
`MFX_TRELLIS_I`	Turn trellis quantization on for I frames.
`MFX_TRELLIS_P`	Turn trellis quantization on for P frames.
`MFX_TRELLIS_B`	Turn trellis quantization on for B frames.

Change History

This enumerator is available since SDK API 1.7.

BRefControl

Description

The `BRefControl` enumerator is used to control usage of B frames as reference in AVC encoder.

Name/Description

`MFX_B_REF_UNKNOWN`	Default value, it is up to the SDK encoder to use B frames as reference.
`MFX_B_REF_OFF`	Do not use B frames as reference.
`MFX_B_REF_PYRAMID`	Arrange B frames in so-called "B pyramid" reference structure.

Change History

This enumerator is available since SDK API 1.8.

LookAheadDownSampling

Description

The `LookAheadDownSampling` enumerator is used to control down sampling in look ahead bitrate control mode in AVC encoder.

Name/Description

`MFX_LOOKAHEAD_DS_UNKNOWN` Default value, it is up to the SDK encoder what down sampling value to use.

`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.

`MFX_LOOKAHEAD_DS_2x` Down sample frames two times before estimation.

`MFX_LOOKAHEAD_DS_4x` Down sample frames four times before estimation. This option may significantly degrade quality.

Change History

This enumerator is available since SDK API 1.8.

Appendices

Appendix A: Configuration Parameter Constraints

The **mfxFrameInfo** structure is used by both the **mfxVideoParam** structure during SDK class initialization and the **mfxFrameSurface1** structure during the actual SDK class function. The following constraints apply:

Constraints common for **DECODE**, **ENCODE** and **VPP:**

Parameters	During SDK initialization	During SDK 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.

Constraints for **DECODE:**

Parameters	During SDK initialization	During SDK operation
Width Height	Aligned frame size	The values must be the equal to or larger than the initialization values.
CropX, CropY 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;	**DECODE** output.
FrameRateExtN FrameRateExtD	Any valid values or unspecified (zero); if unspecified, values from the input bitstream will be used;	**DECODE** output.

`PicStruct`	Ignored	**DECODE** output.

Constraints for **VPP**:

Parameters	**During SDK initialization**	**During SDK operation**
`Width` `Height`	Any valid values	These values must be the same 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`, or `MFX_PICSTRUCT_FIELD_BFF`.	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.

Constraints for **ENCODE**:

Parameters	**During SDK initialization**	**During SDK operation**
`Width` `Height`	Encoded frame size	The values must be the same or larger than the initialization values
`CropX, CropY`	H.264: Cropped frame size	Ignored

Parameters	During SDK initialization	During SDK operation
CropW, CropH	MPEG-2: CropW and CropH specify the real width and height (maybe unaligned) of the coded frames. CropX and CropY must be zero.	
AspectRatioW AspectRatioH	Any valid values	Ignored
FrameRateExtN FrameRateExtD	Any valid values	Ignored
PicStruct	MFX_PICSTRUCT_UNKNOWN, MFX_PICSTRUCT_PROGRESSIVE, MFX_PICSTRUCT_FIELD_TFF, or 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.

The following table summarizes how to specify the configuration parameters during initialization and during encoding, decoding and video processing:

		ENCODE		DECODE		VPP	
		Init	Encoding	Init	Decoding	Init	Processing
mfxVideoParam							
	Protected	R	-	R	-	R	-
	IOPattern	M	-	M	-	M	-
	ExtParam	O	-	O	-	O	-
	NumExtParam	O	-	O	-	O	-
mfxInfoMFX							
	CodecId	M	-	M	-	-	-
	CodecProfile	O	-	O	-	-	-
	CodecLevel	O	-	O	-	-	-

		ENCODE		DECODE		VPP	
		Init	Encoding	Init	Decoding	Init	Processing
	NumThread	O	-	O	-	-	-
	TargetUsage	O	-	-	-	-	-
	GopPicSize	O	-	-	-	-	-
	GopRefDist	O	-	-	-	-	-
	GopOptFlag	O	-	-	-	-	-
	IdrInterval	O	-	-	-	-	-
	RateControlMethod	O	-	-	-	-	-
	InitialDelayInKB	O	-	-	-	-	-
	BufferSizeInKB	O	-	-	-	-	-
	TargetKbps	M	-	-	-	-	-
	MaxKbps	O	-	-	-	-	-
	NumSlice	O	-	-	-	-	-
	NumRefFrame	O	-	-	-	-	-
	EncodedOrder	M	-	-	-	-	-
mfxFrameInfo							
	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
Remarks							
	Ign	Ignored	PT	Pass Through	-	Does Not Apply	
	M	Mandated	R	Reserved			

		ENCODE		DECODE		VPP	
		Init	Encoding	Init	Decoding	Init	Processing
	O Optional /U Updated at output						

Appendix B: Multiple-Segment Encoding

Multiple-segment encoding is useful in video editing applications when during production; 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, as illustrated in Figure 6. (Note that different encoders can also be used.)

The application must be able to:

1. Extract encoding parameters from the bitstream of previously encoded segment;
2. Import these encoding parameters to configure the encoder.

Encoding can then continue on the current segment using either the same or the similar encoding parameters.

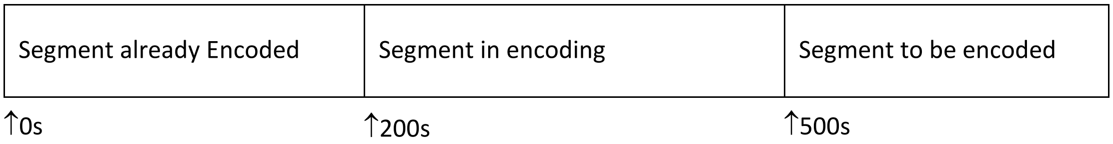

Figure 6: Multiple-Segment Encoding

Extracting the header containing the encoding parameter set from the encoded bitstream is usually the task of a format splitter (de-multiplexer). Nevertheless, the SDK **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. Table 9 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: Multiple-Segment Encoding 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 it anywhere in the final bitstream. After encoding, each encoded segment is HRD compliant. However, the concatenated segments may not be HRD compliant.

Example 15 shows an example of 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=&option_array;

      /* 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;
}
```

Example 15: Pseudo-code to Import H.264 SPS/PPS Parameters

Appendix C: Streaming and Video Conferencing Features

The following sections address a few aspects of additional requirements that streaming or video conferencing applications may use in the encoding or transcoding process. See also Configuration Change chapter.

Dynamic Bitrate Change

The SDK encoder supports dynamic bitrate change differently depending on bitrate control mode and HRD conformance requirement. If HRD conformance is required, i.e. if application sets **`NalHrdConformance`** option in **`mfxExtCodingOption`** structure to ON, the only allowed bitrate control mode is VBR. In this mode, the application can change **`TargetKbps`** and **`MaxKbps`** values. The application can change these values by calling the **`MFXVideoENCODE_Reset`** function. Such change in bitrate usually results in generation of a new key-frame and sequence header. There are some exceptions though. For example, if HRD Information is absent[1] in the stream then change of **`TargetKbps`** does not require change of sequence header and as a result the SDK encoder does not insert a key frame.

If HRD conformance is not required, i.e. if application turns off **`NalHrdConformance`** option in **`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. Such change in bitrate will not result in generation of a new key-frame or sequence header.

The SDK encoder may change some of the 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 working set of parameters used by the SDK encoder. That is why it is strongly recommended to retrieve the actual working parameters by **`MFXVideoENCODE_GetVideoParam`** function before making any changes to bitrate settings.

In all modes, the SDK 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 soon the actual bitrate can catch up with the specified bitrate is implementation dependent.

Alternatively, the application may use the CQP (constant quantization parameter) 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.

[1] HRD information is absent in the stream if both VuiVclHrdParameters and VuiNalHrdParameters options in mfxExtCodingOption structure are OFF.

Dynamic resolution change

The SDK encoder supports dynamic resolution change in all bitrate control modes. The application may change resolution by calling **MFXVideoENCODE_Reset** function. The application may decrease or increase resolution up to the size specified during encoder initialization.

Resolution change always results in insertion of key IDR frame and new sequence parameter set header. The SDK encoder does not guarantee HRD conformance across resolution change point.

The SDK encoder may change some of the 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 SDK encoder. That is why it is strongly recommended to retrieve the actual working parameters set by **MFXVideoENCODE_GetVideoParam** function before making any resolution change.

Forced Key Frame Generation

The SDK supports forced key frame 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 SDK encoder works in the display order, the application can enforce any current frame to be a key frame. The application cannot change the frame type of already buffered frames inside the SDK encoder.
- If the SDK encoder works in the encoded order, the application must exactly specify frame type for every frame thus the application can enforce the current frame to have any frame type that particular coding standard allows.

Reference List Selection

During streaming or video conferencing, if the application can obtain feedbacks about how good the client receives certain frames, the application may need to adjust the encoding process to use or not use certain frames as reference. The following paragraphs describe how to fine-tune the encoding process based on such feedbacks.

The application can specify the reference window size by specifying the parameter **mfxInfoMFX::NumRefFrame** during encoding initialization. Certain platform may have limitation on how big the size of the reference window is. Use the function **MFXVideoENCODE_GetVideoParam** 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. The

NumRefIdxL0Active parameter of the **mfxExtAVCRefListCtrl** structure specifies the length of the reference list L0 and the **NumRefIdxL1Active** parameter specifies the length of the reference list L1. These two numbers must be less or equal to the parameter **mfxInfoMFX::NumRefFrame** during encoding initialization.

The application can instruct the SDK encoder to use or not use certain reference frames. To do this, there is a prerequisite that the application must 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** in the **mfxExtAVCRefListCtrl** structure, and attach the structure to the **MFXVideoENCODE_EncodeFrameAsync** function. The two lists fine-tune how the SDK encoder chooses the reference frames of the current frame. The SDK encoder does not keep **PreferredRefList** and application has to send it for each frame if necessary. There are a few limitations:

- The frames in the lists are ignored if they are out of the reference window.
- If by going through the lists, the SDK encoder cannot find a reference frame for the current frame, the SDK encoder will encode the current frame without using any reference frames.
- If the GOP pattern contains B-frames, the SDK encoder may not be able to follow the **mfxExtAVCRefListCtrl** instructions.

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 **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 number of reference frames.

Reference Picture Marking Repetition SEI message

The application can request writing the reference picture marking repetition SEI message during encoding initialization, by setting the **RefPicMarkRep** flag in the **mfxExtCodingOption** structure. The reference picture marking repetition SEI message repeats certain reference frame information in the output bitstream for robust streaming.

The SDK decoder will respond to the reference picture marking repetition SEI message if such message exists in the bitstream, and check with 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.

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 SDK encoder itself never marks 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/unmark those frames that are buffered inside encoder. Because of this, it is recommended that the application marks a frame when it is submitted for encoding. Application can either explicitly unmark long-term reference frame or wait for IDR frame, there all long-term reference frames will be unmarked.

The SDK 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 smaller than the total reference frame number (the **NumRefFrame** value in the `mfxInfoMFX` structure during the encoding initialization), the SDK encoder may ignore some or all long term reference frames. The application may avoid this by providing list of preferred reference frames in the **PreferredRefList** list in the `mfxExtAVCRefListCtrl` extended buffer. In this case, the SDK encoder reorders the reference list based on the specified list.

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. The SDK 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. The SDK 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/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. **Scale** for layer N is equal to ratio between the frame rate of subsequence consisted of temporal layers with temporal ID lower or equal to N and frame rate of base temporal layer. The application may skip some of the 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, 30 frame per second video sequence typically is 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). **`Scale`** for this case should have next values **`{1,2,4,0,0,0,0,0}`**.

Appendix D: Switchable Graphics and Multiple Monitors

The following sections address a few aspects of supporting switchable graphics and multiple monitors configurations.

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 document discusses the implication of switchable graphics to the Intel® Media SDK and the SDK applications.

As the SDK performs hardware acceleration through Intel graphic device, it is critical that the SDK can access to the Intel graphic device in the switchable graphics setting. If possible, it is recommended to add the application to the Intel graphic device affinity list. Otherwise, the application must handle the following cases:

1> By the SDK design, during the SDK library initialization, the function **`MFXInit`** searches for Intel graphic devices. If a SDK implementation is successfully loaded, the function **`MFXInit`** returns **`MFX_ERR_NONE`** and the **`MFXQueryIMPL`** function returns the actual implementation type. If no SDK implementation is loaded, the function **`MFXInit`** returns **`MFX_ERR_UNSUPPORTED`**.

In the switchable graphics environment, if the application is not in the Intel graphic device affinity list, it is possible that the Intel graphic device is not accessible during the SDK library initialization. The fact that the **`MFXInit`** function returns **`MFX_ERR_UNSUPPORTED`** does not mean that hardware acceleration is not possible permanently. The user may switch the graphics later and by then the Intel graphic device will become accessible. It is recommended that the application initialize the SDK library right before the actual decoding, video processing, and encoding operations to determine the hardware acceleration capability.

2> During decoding, video processing, and encoding operations, if the application is not in the Intel graphic device affinity list, the previously accessible Intel graphic device may become inaccessible due to a switch event. The SDK functions will return **MFX_ERR_DEVICE_LOST** or **MFX_ERR_DEVICE_FAILED**, depending on when the switch occurs and what stage the SDK functions operate. The application needs to handle these errors and exits gracefully.

Multiple Monitors

Multiple monitors refer to the machine configuration that multiple graphic devices are available. Some of the graphic devices connect to a display, they become active and accessible under the Microsoft* DirectX* infrastructure. For those graphic devices not connected to a display, they are inactive. Specifically, under the Microsoft Direct3D9* infrastructure, those devices are not accessible.

The SDK uses the adapter number to access to a specific graphic device. Usually, the graphic device that drives the main desktop becomes the primary adapter. Other graphic devices take subsequent adapter numbers after the primary adapter. Under the Microsoft Direct3D9 infrastructure, only active adapters are accessible and thus have an adapter number.

The SDK extends the implementation type **mfxIMPL** as follows

Implementation Type	Definition
MFX_IMPL_HARDWARE	The SDK should initialize on the primary adapter
MFX_IMPL_HARDWARE2	The SDK should initialize on the 2nd graphic adapter
MFX_IMPL_HARDWARE3	The SDK should initialize on the 3rd graphic adapter
MFX_IMPL_HARDWARE4	The SDK should initialize on the 4th graphic adapter

The application can use the above definitions to instruct the SDK library to initializes on a specific graphic device. The application can also use the following definitions for automatic detection:

Implementation Type	Definition
MFX_IMPL_HARDWARE_ANY	The SDK should initialize on any graphic adapter
MFX_IMPL_AUTO_ANY	The SDK should initialize on any graphic adapter. If not successful, load the software implementation.

If the application uses the Microsoft* DirectX* surfaces for I/O, it is critical that the application and the SDK works 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 SDK library, with option **MFX_IMPL_HARDWARE_ANY** or **MFX_IMPL_AUTO_ANY**. The **MFXInit** function returns **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_HARDWARE4** indicates the graphic adapter the SDK works on.

3> The application creates the Direct3D* device on the respective graphic adapter, and passes it to the SDK through the **MFXVideoCORE_SetHandle** function.

Finally, similar to the switchable graphics cases, it is possible that the user disconnects monitors from the graphic devices or remaps the primary adapter thus causes interruption. If the interruption occurs during the SDK library initialization, the **MFXInit** function may return **MFX_ERR_UNSUPPORTED**. This means hardware acceleration is currently not available. It is recommended that the application initialize the SDK library 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, the SDK functions will return **MFX_ERR_DEVICE_LOST** or **MFX_ERR_DEVICE_FAILED**. The application needs to handle these errors and exit gracefully.

Appendix E: Working directly with VA API for Linux*

The SDK takes care of all memory and synchronization related operations in VA API. However, in some cases the application may need to extend the SDK functionality by working directly with VA API for Linux*. For example, to implement customized external allocator or **USER** functions (also known as “plug-in”). This chapter describes some basic memory management and synchronization techniques.

To create VA surface pool the application should call vaCreateSurfaces as it is shown in Example 16.

```
VASurfaceAttrib attrib;
attrib.type = VASurfaceAttribPixelFormat;
attrib.value.type = VAGenericValueTypeInteger;
attrib.value.value.i = VA_FOURCC_NV12;
attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;

#define NUM_SURFACES 5;
VASurfaceID surfaces[NUMSURFACES];

vaCreateSurfaces(va_display, VA_RT_FORMAT_YUV420,
                 width, height,
                 surfaces, NUM_SURFACES,
                 &attrib, 1);
```

Example 16: Creation of VA surfaces

To destroy surface pool the application should call vaDestroySurfaces as it is shown in Example 17.

```
vaDestroySurfaces(va_display, surfaces, NUM_SURFACES);
```

Example 17: Destroying of VA surfaces

If the application works with hardware acceleration through the SDK then it can access surface data immediately after successful completion of MFXVideoCORE_SyncOperation call. If the application works with hardware acceleration directly then it has to check surface status before accessing data in video memory. This check can be done asynchronously by calling vaQuerySurfaceStatus function or synchronously by vaSyncSurface function.

After successful synchronization the application can access surface data. It is performed in two steps. At the first step VAImage is created from surface and at the second step image buffer is mapped to system memory. After mapping VAImage.offsets[3] array holds offsets to each color plain in mapped buffer and VAImage.pitches[3] array holds color plain pitches, in bytes. For packed data formats, only first entries in these arrays are valid. Example 18 shows how to access data in NV12 surface.

```
VAImage image;
unsigned char *buffer, Y, U, V;

vaDeriveImage(va_display, surface_id, &image);
vaMapBuffer(va_display, image.buf, &buffer);

/* NV12 */
Y = buffer + image.offsets[0];
U = buffer + image.offsets[1];
V = U + 1;
```

Example 18: Accessing data in VA surface

After processing data in VA surface the application should release resources allocated for mapped buffer and VAImage object. Example 19 shows how to do it.

```
vaUnmapBuffer(va_display, image.buf);
vaDestroyImage(va_display, image.image_id);
```

Example 19: unmapping buffer and destroying VAImage

In some cases, for example, to retrieve encoded bitstream from video memory, the application has to use VABuffer to store data. Example 20 shows how to create, use and then destroy VA buffer. Note, that vaMapBuffer function returns pointers to different objects depending on mapped buffer type. It is plain data buffer for VAImage and VACodedBufferSegment structure for encoded bitstream. The application cannot use VABuffer for synchronization and in case of encoding it is recommended to synchronize by input VA surface as described above.

```
/* create buffer */
VABufferID buf_id;
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);
```

Example 20: Working with encoded bitstream buffer