**ISO 23090-13:202x(2E)**

ISO TC 1 SC 29/WG 03

Secretariat: XXXX

**Information technology — Coded representation of immersive media — Part 13: Video decoding interface for immersive media**

CD stage

**Warning for WDs and CDs**

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

© ISO 2020

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO’s member body in the country of the requester.

ISO copyright office

CP 401 • Ch. de Blandonnet 8

CH-1214 Vernier, Geneva

Phone: +41 22 749 01 11

Fax: +41 22 749 09 47

Email: copyright@iso.org

Website: www.iso.org

Published in Switzerland

Contents

[Foreword vi](#_Toc210474209)

[Introduction vii](#_Toc210474210)

[1 Scope 1](#_Toc210474211)

[2 Normative references 1](#_Toc210474212)

[3 Terms and definitions 1](#_Toc210474213)

[4 Abbreviated terms 2](#_Toc210474214)

[5 Video decoding engine 2](#_Toc210474215)

[5.1 General 2](#_Toc210474216)

[5.2 Input video decoding interface 4](#_Toc210474217)

[5.3 Output video decoding interface 4](#_Toc210474218)

[5.4 Control interface to the Video Decoding Interface 5](#_Toc210474219)

[5.4.1 Functions 5](#_Toc210474220)

[5.5 Examples of video decoding engine instantiations 10](#_Toc210474221)

[5.5.1 Mapping on OpenMAX™ integration layer (OpenMAX IL) 10](#_Toc210474222)

[5.5.2 Mapping on Vulkan® Video 11](#_Toc210474223)

[5.5.3 Informative mapping 14](#_Toc210474224)

[6 VDI systems decoder model 14](#_Toc210474225)

[6.1 Introduction 14](#_Toc210474226)

[6.2 Concepts of the VDI systems decoder model 15](#_Toc210474227)

[6.2.1 General 15](#_Toc210474228)

[6.2.2 Media stream 15](#_Toc210474229)

[6.2.3 Media stream interface 15](#_Toc210474230)

[6.2.4 Input formatter 15](#_Toc210474231)

[6.2.5 Access Units (AU) 15](#_Toc210474232)

[6.2.6 Decoding Buffer (DB) 15](#_Toc210474233)

[6.2.7 Elementary Streams (ES) 16](#_Toc210474234)

[6.2.8 Elementary Stream Interface (ESI) 16](#_Toc210474235)

[6.2.9 Decoder 16](#_Toc210474236)

[6.2.10 Composition Units (CU) 16](#_Toc210474237)

[6.2.11 Composition Memory (CM) 16](#_Toc210474238)

[6.2.12 Compositor 16](#_Toc210474239)

[6.2.13 Object Time Base (OTB) 16](#_Toc210474240)

[6.2.14 Decoding Time Stamp (DTS) 16](#_Toc210474241)

[6.2.15 Composition Time Stamp (CTS) 16](#_Toc210474242)

[6.3 Timing model 16](#_Toc210474243)

[7 Video decoder interface 18](#_Toc210474244)

[7.1 General 18](#_Toc210474245)

[7.2 Operations on input media streams 18](#_Toc210474246)

[7.2.1 General 18](#_Toc210474247)

[7.2.2 Concepts 18](#_Toc210474248)

[7.2.3 Filtering by video object identifier 18](#_Toc210474249)

[7.2.4 Inserting video objects 19](#_Toc210474250)

[7.2.5 Appending two video objects 21](#_Toc210474251)

[7.2.6 Stacking two video objects 22](#_Toc210474252)

[7.3 Slice-based instantiation for ISO/IEC 23008-2 high efficiency video coding (HEVC) 23](#_Toc210474253)

[7.3.1 General 23](#_Toc210474254)

[7.3.2 Media and elementary stream constraints 23](#_Toc210474255)

[7.4 Layer-based instantiation for ISO/IEC 23090-3 versatile video coding (VVC) 24](#_Toc210474256)

[7.4.1 General 24](#_Toc210474257)

[7.4.2 Media and elementary stream constraints 24](#_Toc210474258)

[7.5 Slice-based instantiation for ISO/IEC 23094-1 essential video coding (EVC) 26](#_Toc210474259)

[7.5.1 General 26](#_Toc210474260)

[7.5.2 Media and elementary streams constraints 26](#_Toc210474261)

[7.6 Picture-based instantiation for ISO/IEC 14496-10 advanced video coding (AVC) 28](#_Toc210474262)

[7.6.1 General 28](#_Toc210474263)

[7.6.2 Media and elementary streams constraints 29](#_Toc210474264)

[Annex A (normative) Control interface IDL definition 31](#_Toc210474265)

[A.1 General 31](#_Toc210474266)

[Annex B (informative) OpenMAX IL VDI extension header 32](#_Toc210474267)

[B.1 General 32](#_Toc210474268)

[Annex C (normative) Supplemental enhancement information (SEI) syntax and semantics 33](#_Toc210474269)

[C.1 VDI SEI envelope 33](#_Toc210474270)

[C.2 Independent layer info SEI message 33](#_Toc210474271)

[C.2.4 Examples of video object positioning 36](#_Toc210474272)

[C.2.4.1 Appending 36](#_Toc210474273)

[C.2.4.2 Appending and stacking 37](#_Toc210474274)

[C.3 Interleaving information SEI message 39](#_Toc210474275)

[C.3.3 Example of AVC bitstream interleaving 39](#_Toc210474276)

[Annex D (informative) Example implementations of input formatting operations 42](#_Toc210474277)

[D.1 General 42](#_Toc210474278)

[D.2 Creating a 2-by-2 video mosaic via application control 42](#_Toc210474279)

[D.3 Creating a 2-by-2 video mosaic via SEI control 45](#_Toc210474280)

[Annex E (informative) Brief description of OpenMAX IL functions 46](#_Toc210474281)

[E.1 Decoder engine control interface 46](#_Toc210474282)

[E.1.1 OMX\_Init() and OMX\_Deinit() 46](#_Toc210474283)

[E.1.2 OMX\_GetHandle() and OMX\_FreeHandle() 46](#_Toc210474284)

[E.1.3 OMX\_SetupTunnel() and OMX\_TeardownTunnel() 46](#_Toc210474285)

[E.2 Decoder instance interface 46](#_Toc210474286)

[E.2.1 Methods 46](#_Toc210474287)

[E.2.2 Media input and output interface 47](#_Toc210474288)

[E.2.3 Format of the OpenMAX IL buffer header 48](#_Toc210474289)

[E.2.4 Buffer flags defined in OpenMAX IL 48](#_Toc210474290)

[E.2.5 Input/Output from/into GPU 48](#_Toc210474291)

[Annex F (informative) Mapping on media source extensions (MSE) 49](#_Toc210474292)

[F.1 Overview 49](#_Toc210474293)

[F.2 Mapping of VDI functions 50](#_Toc210474294)

[Annex G (informative) Mapping on WebCodecs 51](#_Toc210474295)

[G.1 Overview 51](#_Toc210474296)

[G.2 Mapping of VDI Functions 51](#_Toc210474297)

[G.3 Example implementations 52](#_Toc210474298)

[G.3.1 queryCurrentAggregateCapabilities 52](#_Toc210474299)

[G.3.2 VDIVideoDecoder class, with grouping and getInstance, constructor 53](#_Toc210474300)

[G.3.3 setConfig 53](#_Toc210474301)

[G.3.4 getParameter, setParameter 54](#_Toc210474302)

[Annex H (normative) Levels 55](#_Toc210474303)

[H.1 Introduction 55](#_Toc210474304)

[H.2 Definitions 55](#_Toc210474305)

[H.3 Level limits 56](#_Toc210474306)

[Bibliography 57](#_Toc210474307)

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see [www.iso.org/directives](https://www.iso.org/directives-and-policies.html) or [www.iec.ch/members\_experts/refdocs](https://www.iec.ch/members_experts/refdocs)).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see [www.iso.org/patents](https://www.iso.org/iso-standards-and-patents.html)) or the IEC list of patent declarations received (see <https://patents.iec.ch>).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see [www.iso.org/iso/foreword.html](https://www.iso.org/iso/foreword.html). In the IEC, see [www.iec.ch/understanding-standards](https://www.iec.ch/understanding-standards).

This document was prepared by Technical Committee ISO/IEC/JTC 1 *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

A list of all parts in the ISO 23090 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at [www.iso.org/members.html](https://www.iso.org/members.html) and [www.iec.ch/national-committees](https://www.iec.ch/national-committees).

Introduction

The interfaces and operations specified in this document come as extensions of existing video decoding engine specifications exposing hardware video decoding capabilities.

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO and IEC that he/she is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from the patent database available at [www.iso.org/patents](https://www.iso.org/patents) or [https://patents.iec.ch](https://patents.iec.ch/)*.*

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — Coded representation of immersive media — Part 13: Video decoding interface for immersive media

# Scope

This document specifies the interfaces of a video decoding engine as well as the operations related to elementary streams and metadata that can be performed by this video decoding engine. To support those operations, this document also specifies SEI messages when necessary for certain video codecs.

# Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 23008-2, *Information technology — High efficiency coding and media delivery in heterogeneous environments — Part 2: High efficiency video coding*

ISO/IEC 23090-3, *Information technology — Coded representation of immersive media — Part 3: Versatile video coding*

ISO/IEC 23094-1, *Information technology — General video coding — Part 1: Essential video coding*

# Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at [https://www.iso.org/obp](https://www.iso.org/obp/ui)

— IEC Electropedia: available at <https://www.electropedia.org/>

3.1

media stream

part of an *elementary stream* (3.2) or one or more aggregated *elementary streams* (3.2)

Note 1 to entry: Every elementary stream is a media stream, but the inverse is not true.

Note 2 to entry: A media stream may contain metadata such as non-VCL NAL units.

3.2

subframe

independently decodable unit smaller than a frame to which post-decoding processing by the decoder, if any, has been applied

3.3

video object

independently decodable substream of a video *elementary stream* (3.2)

3.4

video object identifier

integer identifying a *video object* (3.4)

# Abbreviated terms

API application programming interface

ES elementary stream

I video object identifier

IVDI input video decoding interface

MDS media stream

NAL network abstraction layer

OLS output layer set

OVDI output video decoding interface

PPS picture parameter set

SEI supplemental enhancement information

SPS sequence parameter set

VCL video coding layer

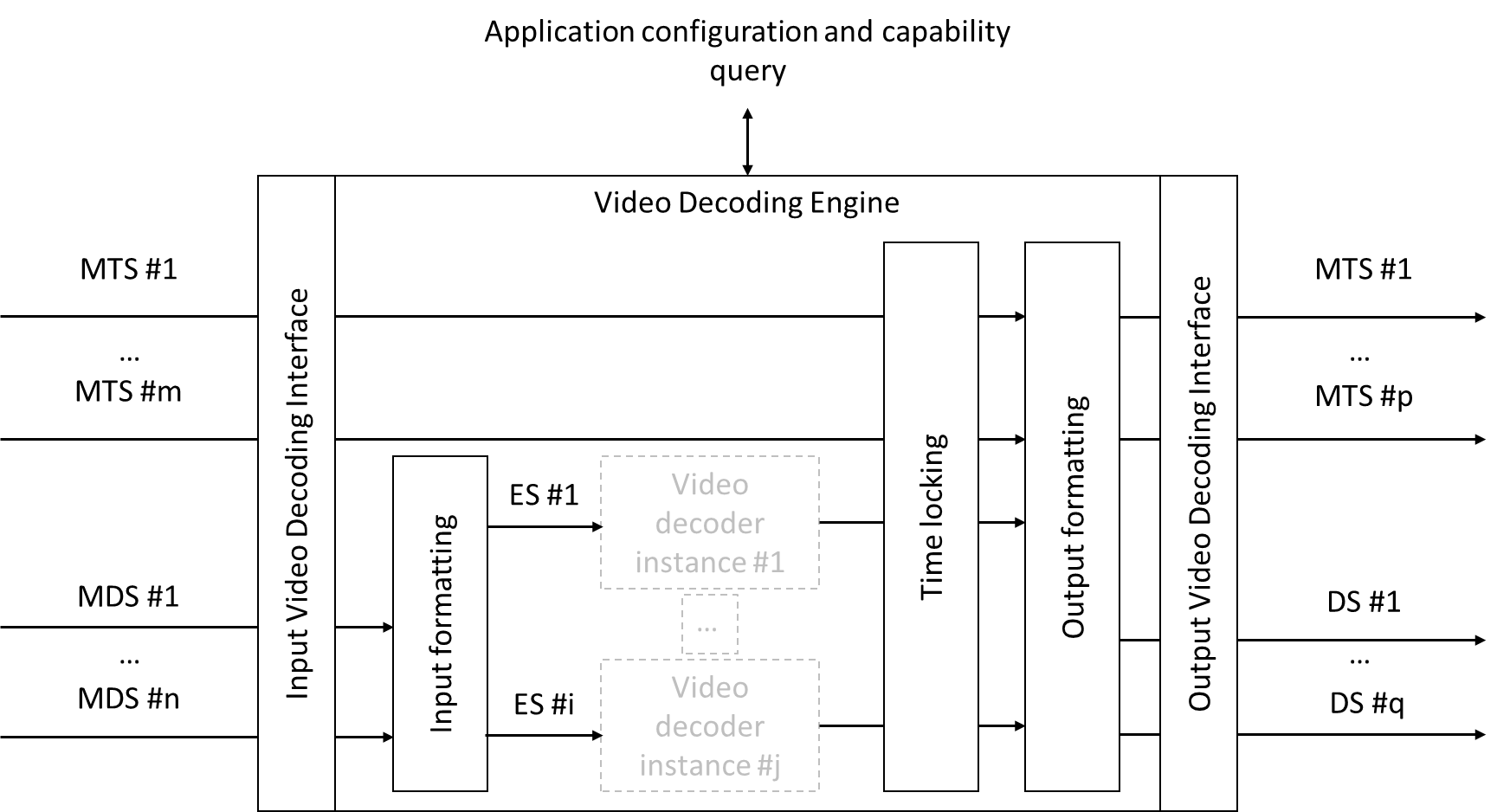
VDE video decoding engine

# Video decoding engine

## General

The video decoding engine (VDE) enables the decoding, the synchronization and the formatting of associated media streams into elementary streams. The media streams are fed through the input video decoding interface (IVDI) of the VDE and provided to the subsequent elements of the rendering pipeline via the output video decoding interface (OVDI) in their decoded form. Between the input and the output, the VDE may extract and merge independently decodable regions from a set of input media streams via the input formatting function to generate a set of elementary streams that are fed into the video decoder instances which run inside the engine. The VDE can execute a merging operation or an extraction operation on the input media streams such that the number of running video decoder instances is different from the number of input media streams required by the application. For example, a VDE can be incapable of decoding a single 4K input media stream with one decoder instance, but it can decode some of the independently decodable regions, at a lower resolution, present in that input media stream. To this end, the VDE should first verify the availability of sufficient resources to run in parallel those video decoder instances.

Figure 1 represents the architecture of the VDE and the associated IVDI and OVDI interfaces.



**Key**

|  |  |  |  |
| --- | --- | --- | --- |
| MDS | media stream |  |  |
| ES | elementary stream |  |  |
| MTS | metadata stream |  |  |
| DS | decoded sequence |  |  |
| m | number of input metadata streams |  |  |
| n | number of media streams |  |  |
| j | number of video decoder instances |  |  |
| p | number of output metadata streams |  |  |
| q | number of decoded sequences |

Figure 1 — Video decoding engine and interfaces

NOTE Multiple elementary streams that are output of the input formatting function can be fed to a single video decoder instance.

NOTE The concept of metadata stream does not yet possess a definition in this document and may be further refined in future editions of this document. Figure 2 depicts an architecture for handling multiple video decoder instances on a single hardware platform. In this scenario, one or more video decoder instances running on the same video decoder hardware engine are exposed to the application layer as several decoder instances each with their own interface.

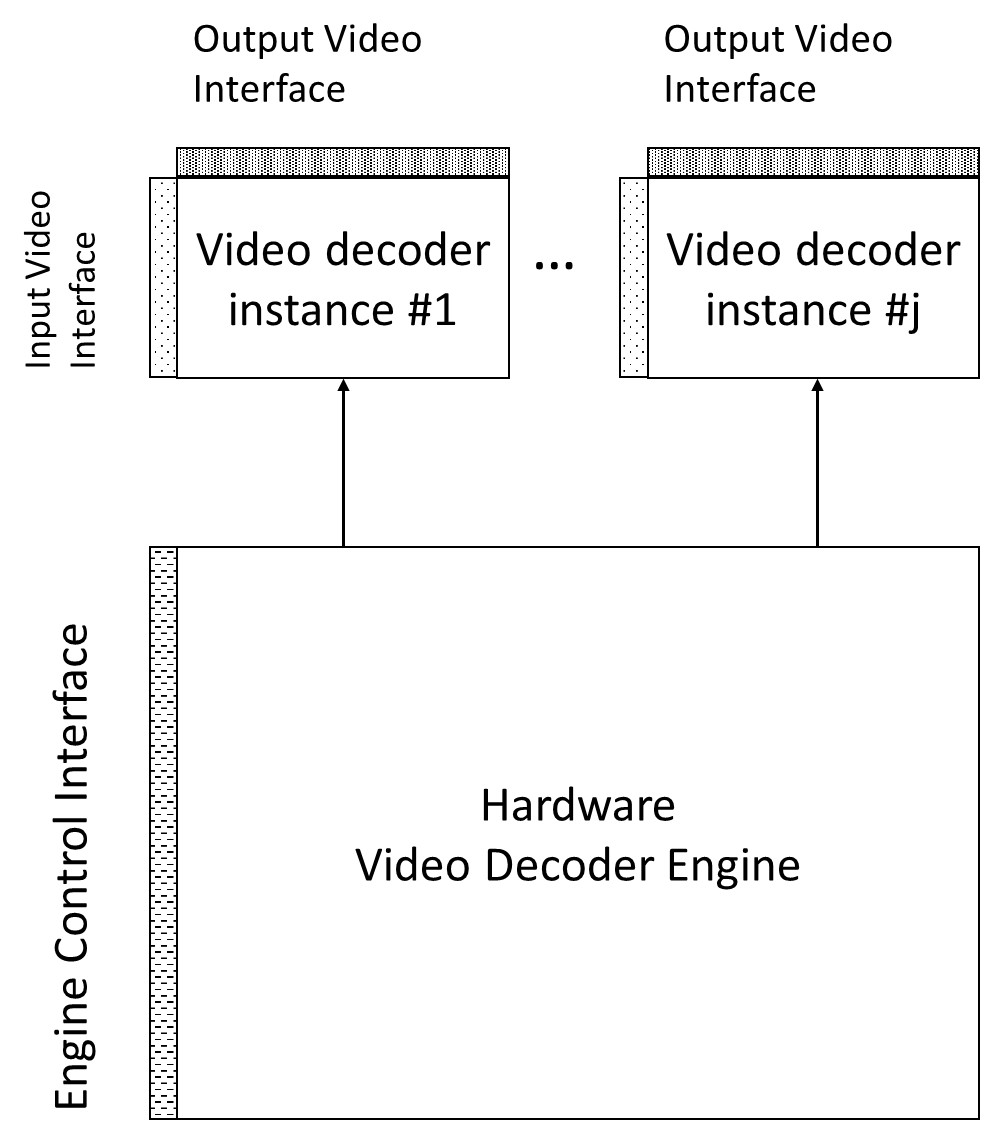


Figure 2 — Example relationship between video decoder instances and video decoder hardware engine

## Input video decoding interface

The video decoding engine accepts media streams and metadata streams. There is at least one media stream as input but there is no constraint on the number of metadata streams with respect to the number of media streams being concurrently consumed by the VDE.

The input of the VDE comprises thus:

* n media streams
* m metadata streams

## Output video decoding interface

The video decoding engine outputs decoded video sequences and metadata streams. There is at least one decoded video sequence as output but there is no constraint on the number of metadata streams with respect to the number of decoded video sequences being concurrently output by the VDE.

These two output stream types may be provided in form of multiplexed output buffers, including both decoded media data and its associated metadata.

The output of the VDE comprises thus:

* q decoded sequences
* p metadata streams

## Control interface to the Video Decoding Interface

### Functions

In order to support immersive media applications, Clause 5.4 defines an abstract video decoding interface. A video decoding platform that complies with this document shall implement this video decoding interface whose IDL can be found in Annex A.

The video decoding interface consists of the abstract functions defined in the following subclause. These functions are defined using the IDL syntax specified in ISO/IEC 19516.

Figure 3 depicts an example instantiation of decoder instances using some of the functionalities of the video decoding interface. The video decoder instances with identifiers 1 to 3 belong to the group with identifier 4. By this grouping mechanism, the three instances are instructed to write the decoded sequences into a single aggregate buffer and the decoding operations across those instances are performed in a coordinated manner such that no instance runs ahead or behind the others.

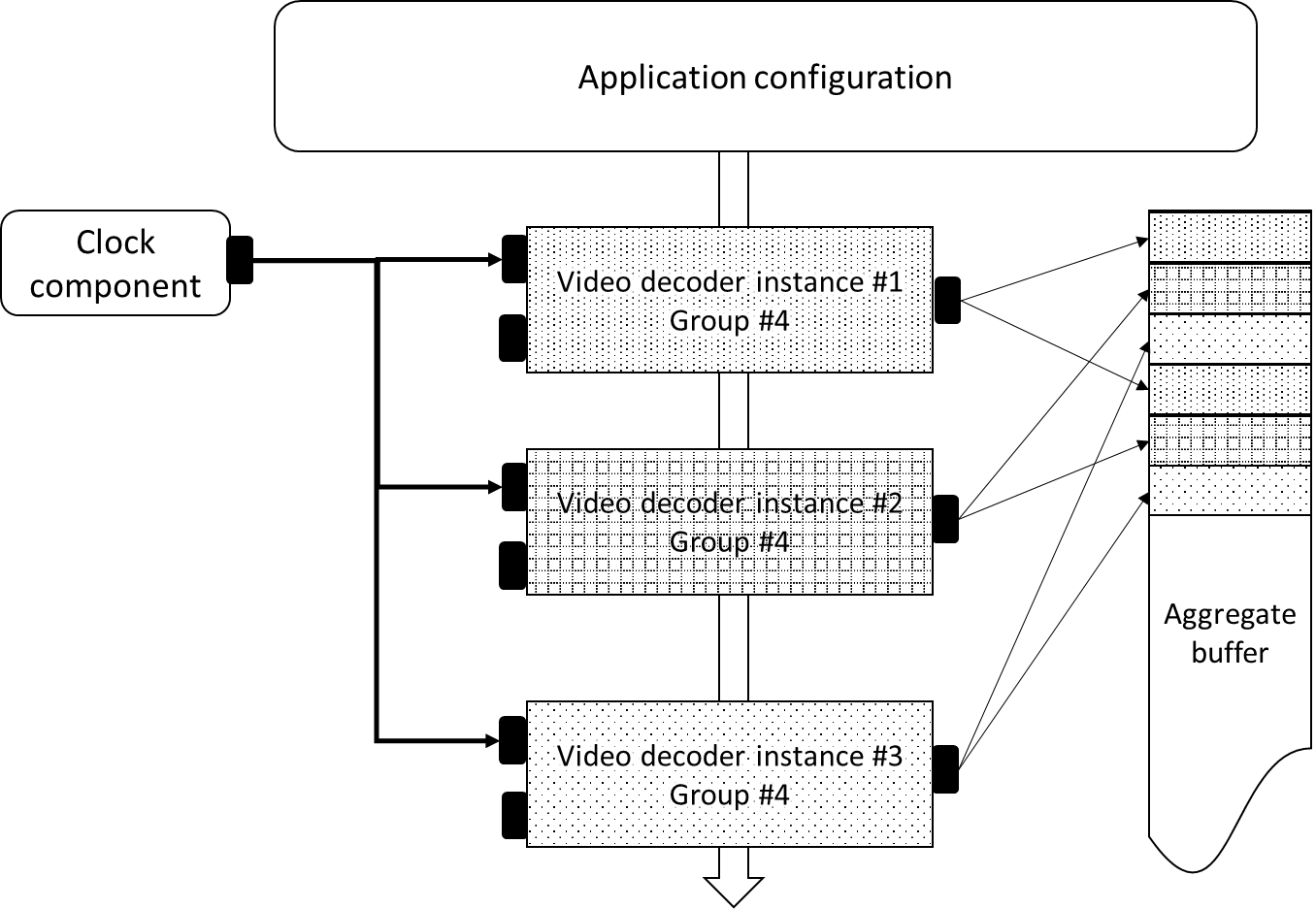


Figure 3 — Example instantiation using VDI

#### queryCurrentAggregateCapabilities()

##### Declaration

The IDL declarations of the queryCurrentAggregateCapabilities() function along with the AggregateCapabilities and PerformancePoint structures and the capabilities flags are defined as follows:

|  |
| --- |
| const unsigned long CAP\_INSTANCES\_FLAG = 0x1;  const unsigned long CAP\_BUFFER\_MEMORY\_FLAG = 0x2;  const unsigned long CAP\_BITRATE\_FLAG = 0x4;  const unsigned long CAP\_MAX\_SAMPLES\_SECOND\_FLAG = 0x8;  const unsigned long CAP\_MAX\_PERFORMANCE\_POINT\_FLAG = 0xA;  enum ChromaFormat {  CF\_monochrome,  CF\_YCbCr\_420,  CF\_YCbCr\_422,  CF\_YCbCr\_444  };  struct PerformancePoint {  float picture\_rate;  unsigned long width;  unsigned long height;  unsigned long bit\_depth;  ChromaFormat chroma\_format;  };  struct AggregateCapabilities {  unsigned long flags;  unsigned long max\_instances;  unsigned long buffer\_memory;  unsigned long bitrate;  unsigned long max\_samples\_second;  PerformancePoint max\_performance\_point;  };  AggregateCapabilities queryCurrentAggregateCapabilities (  in string component\_name,  in unsigned long flags  ); |

##### Definition

The queryCurrentAggregateCapabilities() function can be used by the application to query the instantaneous aggregate capabilities of a decoder platform for a specific codec component.

The capability flags below can set separately or in a single function call to query one or more parameters.

The component\_name provides the name of the component of the decoding platform for which the query applies. The name All may be used to indicate that the query is not for a particular component but is rather for all the components of the decoding platform. Components are hardware or software functionalities exposed by the Video Decoding Engine such as decoders.

CAP\_INSTANCES\_FLAG queries the max\_instances parameter which indicates the maximum number of decoder instances that can be instantiated at this moment for the provided decoder component.

CAP\_BUFFER\_MEMORY\_FLAG queries the buffer\_memory parameter which indicates the instantaneous global maximum available buffer size in bytes that can be allocated independently of any components at this moment on the decoder platform for buffer exchange. The allocation of the memory can be done by the application or the VDE itself depending on the VDE instantiation.

CAP\_BITRATE\_FLAG queries the bitrate parameter which indicates the instantaneous maximum coded bitrate in bits per second that the queried component can process.

CAP\_MAX\_SAMPLES\_SECOND\_FLAG queries the max\_samples\_second parameter which indicates the instantaneous maximum number of luma and chroma samples combined per second that the queried component is able to process.

CAP\_MAX\_PERFORMANCE\_POINT\_FLAG queries the max\_performance\_point parameter which indicates the maximum performance point of a bitstream that can be decoded by the indicated component in a new instance of that decoder component.

A PerformancePoint contains the following parameters:

* picture\_rate indicating the instantaneous picture rate of the maximum performance point in pictures per second.
* height indicating the height in luma samples of the maximum performance point.
* width indicating the width in luma samples of the maximum performance point.
* bit\_depth indicating the bit depth of the luma samples of the maximum performance point.
* chroma\_format indicating the assumed chroma format for this performance point.

NOTE Each parameter of the max performance point does not necessarily represent the maximum in that dimension. It is the combination of all dimensions that constitutes the maximum performance point.

#### getInstance()

##### Declaration

The IDL declarations of the getInstance() function and the associated ErrorAllocation exception are defined as follows:

|  |
| --- |
| exception ErrorAllocation {  string reason;  };  unsigned long getInstance(  in string component\_name,  inout unsigned long group\_id // optional, default value = -1  ) raises(ErrorAllocation); |

##### Definition

The result of a successful call to the getInstance()function shall provide the identifier of the instance and the group\_id that is assigned or created for this new instance, if one was requested. The default behavior is that the decoder instance does not belong to any already established group but is assigned to a newly created group.

Several decoder instances belonging to the same group means that the VDE treats those instances collectively such that the decoding states of those instances progress in synchrony and not in competition against each other. As a result, the VDE will also ensure synchronized output writing operation, possibly into an aggregate buffer. There are no conditions for two video decoder instances to be in the same group. In other words, decoder instances may belong to the same group even if their respective components are different.

#### setConfig()

##### Declaration

The IDL declarations of the setConfig() function, the associated ErrorConfig exception, the ConfigDataParameters structure and the ConfigParameters enumeration are defined as follows:

|  |
| --- |
| enum ConfigParameters {  CONFIG\_OUTPUT\_BUFFER  };   enum OutputFormat {  OF\_R,  OF\_G,  OF\_B,  OF\_RGB,  OF\_RGBA,  OF\_DEPTH,  OF\_ALPHA,  OF\_AUDIO  };  enum SampleFormat {  SF\_SCALAR,  SF\_VEC2,  SF\_VEC3,  SF\_VEC4  };  enum SampleType {  ST\_BYTE,  ST\_UNSIGNED\_BYTE,  ST\_SHORT,  ST\_UNSIGNED\_SHORT,  ST\_UNSIGNED\_INT,  ST\_FLOAT  };  struct ConfigDataParameters {  OutputFormat output\_format;  SampleFormat sample\_format;  SampleType sample\_type;  unsigned long sample\_stride;  unsigned long line\_stride;  unsigned long buffer\_offset;  unsigned long output\_buffer\_handle;  };  exception ErrorConfig {  string reason;  };  boolean setConfig (  in unsigned long instance\_id,  in ConfigParameters config\_parameters,  in ConfigDataParameters config\_data\_parameters  ) raises(ErrorConfig); |

##### Definition

The setConfig() function may be called with the parameter CONFIG\_OUTPUT\_BUFFER, in which case it provides a handle to the output buffer and a description of the write operation into that output buffer.

The parameters that are passed to this function when setting the configuration for CONFIG\_OUTPUT\_BUFFER are as follows:

* sample\_format indicating the format of each sample, which can be a scalar, a 2D vector, a 3D vector, or a 4D vector.
* sample\_type indicating the type of each component of the sample.
* sample\_stride indicating the number of bytes between 2 consecutive samples of this output.
* line\_stride indicating the number of bytes between the first byte of one line and the first byte of the following line of this output.
* buffer\_offset indicating the offset into the output buffer, starting from which the output frame should be written.
* output\_buffer\_handle provides the handle of the output buffer, to which the output of the decoder instance is to be written. The VDE is responsible for the allocation and management of the memory for the output buffer.

#### getParemeter() and setParameter()

##### Declaration

The IDL declarations of the getParameter() and setParemeter() functions as well as the associated ErrorParameter exception and the ExtParameters enumeration are defined as follows:

|  |
| --- |
| enum ExtParameters {  PARAM\_PARTIAL\_OUTPUT,  PARAM\_SUBFRAME\_OUTPUT,  PARAM\_METADATA\_CALLBACK,  PARAM\_OUTPUT\_CROP,  PARAM\_MAX\_OFFTIME\_JITTER,  PARAM\_INSTANCE\_PRIORITY  };  enum PartialOutput {  PO\_NOT\_ALLOWED,  PO\_ALLOWED,  PO\_DESIRED,  };  struct SubframeOutput {  unsigned long subframe\_id;  };  struct MetadataCallback {  unsigned long metadata\_ids[]; [Editor’s note: The notation [] is not compliant to the IDL specification.]  unsigned long metadata\_callback;  };  struct MaxOfftimeJitter {  unsigned long jitter\_millis;  };  struct CropWindow {  unsigned long x;  unsigned long y;  unsigned long width;  unsigned long height;  };   enum InstancePriority {  PRIORITY\_NOT\_AVAILABLE,  PRIORITY\_HIGH,  PRIORITY\_NORMAL,  PRIORITY\_LOW  };  exception ErrorParameter {  string reason;  };  any getParameter (  in unsigned long instance\_id,  in ExtParameters ext\_parameters,  out any parameter  );  boolean setParameter (  in unsigned long instance\_id,  in ExtParameters ext\_parameters,  in any parameter  ) raises(ErrorParameter); |

##### Definition

The getParameter() and setParameter() functions can receive the extended parameters in the clauses below.

PARAM\_PARTIAL\_OUTPUT indicates whether the output of corrupted/incomplete frames is required, desired, or not allowed. If it is not allowed, only complete decoded frames will be passed to the buffer. This may be useful to instruct the decoder on how to handle the output of corrupted or incompletely decoded frames as a result of missing or malformed data in the input.

PARAM\_SUBFRAME\_OUTPUT indicates the one or more subframes to be output by the decoder. Subframes may e.g. be auxiliary pictures in a video stream. The identifier of the subframe substream is provided as part of the SubframeOutput structure.

PARAM\_METADATA\_CALLBACK sets a callback function for a specific metadata type. The list of supported metadata types is codec dependent and shall be defined for each codec independently. The set of metadata types for which the callback is to be invoked by the VDE as well as the callback function handler are provided in the MetadataCallback structure.

PARAM\_OUTPUT\_CROP indicates that only part of the decoded frame is desired at the output. The decoder instance may use this information to intelligently reduce its decoding processing by discarding units that do not fall in the cropped output region whenever possible. The information about the area of the video to be output is provided by the CropWindow structure.

PARAM\_MAX\_OFFTIME\_JITTER indicates the maximum amount of time in microseconds between consecutive executions of the decoder instance. This parameter is relevant whenever the underlying hardware component is shared among multiple decoder instances, which requires context switching between the different decoder instances. The information about the allowed jitter is provided in the MaxOfftimeJitter structure.

PARAM\_INSTANCE\_PRIORITY indicates whether the decoding instance supports prioritization. If it does, the priority of the decoder instance can be set to low, normal or high. All the decoding instances of a group have the same priority. As a result, once a priority is set for one instance of a group, the same priority is also assigned to all the other instances of the same group.

## Examples of video decoding engine instantiations

### Mapping on OpenMAX™ integration layer (OpenMAX IL)

#### Overview

For more information on OpenMAX IL, Annex E provides a brief description of the main functions of this API.

#### Mapping of VDI functions

The function defined in 5.4 are mapped on the OpenMAX IL interface by using the extension mechanism defined by the specification. This MPEG VDI extension for OpenMAX IL is formatted as a C header file and registered with the vendor name “MPEG”.

Annex B defines the MPEG VDI extension for OpenMAX Il and provides information to access the electronic version of this extension.

### Mapping on Vulkan® Video

* + - 1. **Overview**

Vulkan® Video (VK) is an extension of the Vulkan API which defines functions exposed by Graphics Processing Units (GPU). This extension provides interfaces for an application to leverage hardware decoding and encoding capabilities present on GPUs.

A VK Video Session consists of a single decoding session on a single layer. As a result, a single VK Video Session corresponds to a single video decoder instance depicted in Figure 1.

The mapping of VDI functions on VK is summarised in Table 1.

Table 1 — Summary of VDI function mapping on Vulkan® Video

|  |  |
| --- | --- |
| VDI functions | VK mapping |
| **queryCurrent AggregateCapabilities** | New vkGetPhysicalDeviceCurrentVideoCapabilitiesMPEG() function |
| **getInstance** (grouping) | Extending VkVideoSessionCreateInfoKHR with a group identifier passed in the new structure VkVideoSessionCreateInfoGroupingMPEG. Call of existing vkCreateVideoSessionKHR(). |
| **setConfig (buffer configuration)** | Mapping on existing VkVideoSessionCreateInfoKHR and VkVideoPictureResourceKHR structures. |
| **getParameter** and **setParameter** | New VkVideoSessionOutputParameterMPEG structure |

#### The vkGetPhysicalDeviceCurrentVideoCapabilitiesMPEG() function

##### Definition

The VK Video API provides a function for querying capabilities for a single VK Video Profile which is called vkGetPhysicalDeviceVideoCapabilitiesKHR(). Similar to this function, the VDI VK mapping defines the vkGetPhysicalDeviceCurrentVideoCapabilitiesMPEG()function. In contrast to the vkGetPhysicalDeviceVideoCapabilitiesKHR() function, the vkGetPhysicalDeviceCurrentVideoCapabilitiesMPEG()function allows to query the aggregates capabilities of the physical device. When it is called with a certain profile, the aggregated capabilities pertain to this given profile.

##### Declaration

VkResult vkGetPhysicalDeviceCurrentVideoCapabilitiesMPEG(  
 VkPhysicalDevice physicalDevice,  
 VkVideoProfileKHR\* pVideoProfile,  
 VkCurrentVideoCapabilitiesMPEG\* pCapabilities);

##### Semantics

physicalDevice is the physical device whose video decode or encode capabilities are to be queried.

pVideoProfile is a pointer to a VkVideoProfileKHR structure with a chained codec-operation specific video profile structure.

pCapabilities is a pointer to a VkCurrentVideoCapabilitiesMPEG structure in which the capabilities are returned.

#### The VkCurrentVideoCapabilitiesMPEG structure

##### Definition

The VkCurrentVideoCapabilitiesMPEG structure holds the information returned by a call to the vkGetPhysicalDeviceCurrentVideoCapabilitiesMPEG() function defined in clause 5.5.2.2.

##### Declaration

typedef struct VkCurrentVideoCapabilitiesMPEG {  
 VkStructureType sType;  
 void\* pNext;  
 uint32\_t maxInstances;  
 uint32\_t bufferMemory;  
 uint32\_t bitrate;  
 uint32\_t maxSamplesSecond;  
 VkPerformancePointMPEG\* maxPerformancePoint;  
} VkCurrentVideoCapabilitiesMPEG;

##### Semantics

sType is the type of this structure.

pNext is NULL or a pointer to a structure extending this structure.

maxInstances see semantic in clause 5.4.1.1.2.

bufferMemory see semantic in clause 5.4.1.1.2.

bitrate see semantic in clause 5.4.1.1.2.

maxSamplesSecond see semantic in clause 5.4.1.1.2.

maxPerformancePoint is a pointer to a VkPerformancePointMPEG structure in which the properties of the maximum performance are returned.

#### The VkCurrentVideoCapabilitiesMPEG structure

##### Definition

The VkCurrentVideoCapabilitiesMPEG structure contains properties describing a performance point for a video processing entity.

##### Declaration

typedef struct VkPerformancePointMPEG {  
 VkStructureType sType;  
 void\* pNext;  
 uint32\_t pictureRate;  
 uint32\_t height;  
 uint32\_t width;  
 uint32\_t bitDepth;

} VkPerformancePointMPEG;

##### Semantics

sType is the type of this structure.

pNext is NULL or a pointer to a structure extending this structure.

pictureRate see semantic in clause 5.4.1.1.2.

height see semantic in clause 5.4.1.1.2.

width see semantic in clause 5.4.1.1.2.

bitDepth see semantic in clause 5.4.1.1.2.

#### The VkVideoSessionCreateInfoGroupingMPEG structure

##### Definition

The VkVideoSessionCreateInfoGroupingMPEG structure allows to attach a group identifier to a video decoding instance created via the VK Video API. This structure extends the VkVideoSessionCreateInfoKHR structure defined in the VK Video API

##### Declaration

typedef struct VkVideoSessionCreateInfoGroupingMPEG {  
 VkStructureType sType;  
 const void\* pNext;  
 uint32\_t groupId;  
} VkVideoSessionCreateInfoGroupingMPEG;

##### Semantics

sType is the type of this structure.

pNext is NULL or a pointer to a structure extending this structure.

groupId see semantic in clause **Error! Reference source not found.**.

#### The VkVideoSessionOutputParameterMPEG structure

##### Definition

The VkVideoSessionOutputParameterMPEG structure contains parameters that configure the properties of the output of the VK Video Session.

##### Declaration

typedef struct VkVideoSessionOutputParameterMPEG {  
 VkStructureType sType;  
 const void\* pNext;  
 VkFlag partialOutput;

uint32\_t\* subframeCount;  
 uint32\_t\* pSubframeOutput;  
 VkFlag outputCrop;  
 VkExtent2D\* pOutputCropWindow;  
 uint32\_t maxOfftimeJitter;  
 void\* pMetadataCallback;  
} VkVideoSessionOutputParameterMPEG;

##### Semantics

sType is the type of this structure.

pNext is NULL or a pointer to a structure extending this structure.

partialOutput see semantic in clause 5.4.1.4.2.

subframeCount and pSubframeOutput see semantic in clause 5.4.1.4.2.

outputCrop see semantic in clause 5.4.1.4.2.

pOutputCropWindow see semantic in clause 5.4.1.4.2.

maxOfftimeJitter see semantic in clause 5.4.1.4.2.

pMetadataCallback see semantic in clause 5.4.1.4.2.

### Informative mapping

This specification also provides informative mapping on other APIs such as in Annex F on the media source extension (MSE).

# VDI systems decoder model

## Introduction

The VDI systems decoder model extends on the systems decoder model (SDM) defined in ISO/IEC 14496-1. Compared to the SDM, the VDI SDM introduces a new interface called the media stream interface ahead of the elementary stream interface. This media stream interface lies at the input of the Input Formatter, also called input formatting function, which accepts as input the so-called media streams. The output of the Input Formatter is one or more elementary streams that can be further stored in decoding buffer (DB) to be passed on to the decoders via the elementary stream interface. After decoding, the output of the decoding process is passed on to a single memory called the composition memory (CM). This composition memory is the source of decoded media data for the Compositor.

These elements are depicted in Figure 4.

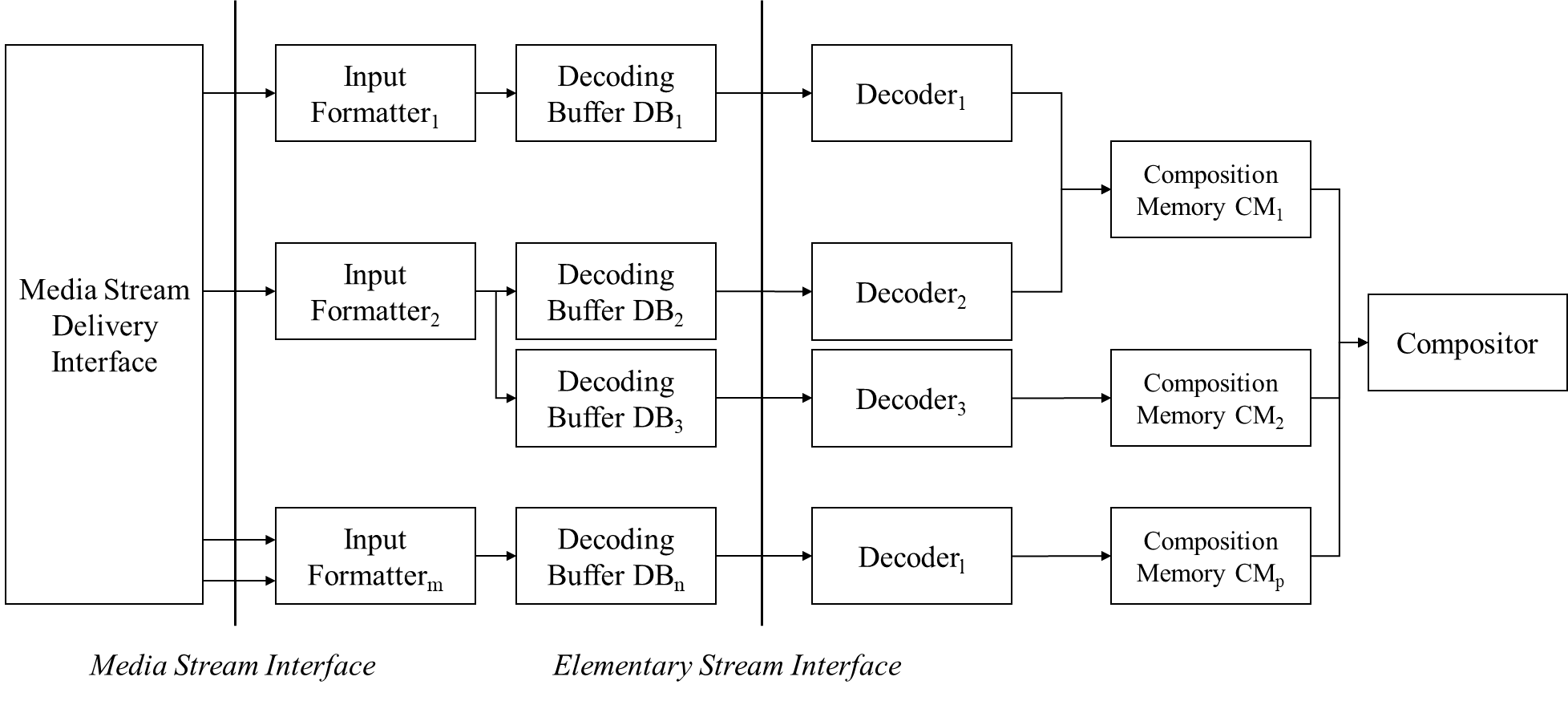


Figure 4 — VDI systems decoder model

NOTE The elements decoding buffer, decoder and compositor are identical to the homonymous concepts defined in ISO/IEC 14496-1. Compared to ISO/IEC 14496-1, the composition memory has the additional property that it can be receiving the output of multiple decoders.

## Concepts of the VDI systems decoder model

### General

The concepts necessary for the specification are the formatting, the timing, and the buffering model. The sequence of definitions corresponds to a walk from the left to the right side of the VDI SDM illustration in Figure 4.

### Media stream

### Media stream interface

The media stream interface is a concept that models the exchange of media stream data between the delivery interface and the input formatting function.

### Input formatter

The input formatter takes one or more media streams as input and generates one or more elementary streams as output. A single input formatter may be attached to several decoding buffers when it produces individual elementary streams or multi-layer elementary streams.

### Access Units (AU)

See 7.1.2.2 in ISO/IEC 14496-1.

### Decoding Buffer (DB)

See 7.1.2.4 in ISO/IEC 14496-1.

### Elementary Streams (ES)

See 7.1.2.5 in ISO/IEC 14496-1.

### Elementary Stream Interface (ESI)

See 7.1.2.6 in ISO/IEC 14496-1.

### Decoder

See 7.1.2.7 in ISO/IEC 14496-1.

### Composition Units (CU)

Decoders consume access units and produce composition units. An access unit corresponds to an integer number of composition units.

In case of multiple elementary streams attached to a single decoder (e.g. scalable coding), each composition unit is derived from access units from one or more of these streams. Composition units reside in composition memory.

In case of multiple decoders attached to a single composition memory, each composition unit is derived from access units from one or more of the elementary streams decoded by those decoders.

### Composition Memory (CM)

The composition memory is a random access memory that contains composition units. The size of this memory is not normatively specified. The composition units may come from different decoders.

### Compositor

See 7.1.2.10 in ISO/IEC 14496-1.

### Object Time Base (OTB)

See 7.1.3.2 in ISO/IEC 14496-1.

### Decoding Time Stamp (DTS)

See 7.1.3.4 in ISO/IEC 14496-1.

### Composition Time Stamp (CTS)

See 7.1.3.5 in ISO/IEC 14496-1.

## Timing model

Access units flow into the decoder of each elementary stream according to a specific arrival time.

All data associated with an access unit is removed from its decoding buffer and decoded instantaneously by the instantaneous decoding process at the time of the decoding time stamp (DTS) of this access unit.

Each composition unit (from the decoding of one or more access units) is guaranteed to be available in the composition memory (CM) at the time of composition time stamp (CTS).

A composition unit is available up to the time of the composition time stamp (CTS) of the next composition unit. After this time, the composition unit may be replaced by the new composition unit.

When a composition unit is created from multiple elementary streams each decoded by a different decoder, those elementary streams shall adhere to a single object time base. Temporally co-located access units for such elementary streams are then identified by identical DTS or CTS values.

EXAMPLE

The example in Figure 5 illustrates the arrival of two access units from two elementary streams in two decoding buffers. At the receiving terminal, both decoders instantaneously decode an AU, at that same instant in time corresponding to their DTS, and the resulting CU is located in the composition memory and remains there until the subsequent CU arrives.

A black background with white text

Description automatically generated

Figure 5 - Composition unit availability created from multiple access units

Legend:

|  |  |
| --- | --- |
| t1 | Arrival time of AU10 in Decoding Buffer1 |
| t2 | Arrival time of AU20 in Decoding Buffer 2 |
| t3 | Arrival time of AU11 in Decoding Buffer 1 |
| t4 | Arrival time of AU21 in Decoding Buffer 2 |
| t5 | Decoding time of AU10  Decoding time of AU20 |
| t6 | Composition time of CU0 |
| t7 | Decoding time of AU11  Decoding time of AU21 |
| t8 | Composition time of CU1 |

# Video decoder interface

## General

As shown in Figure 1, the hardware video decoding engine may spawn one or more video decoder instances. The number of instances running is an optimization choice for the platform when considering available resources such as computational load, energy consumption, memory, etc. However, the number of input media streams fed through the IVDI depends on the application needs to properly render the media experience. Therefore, one or more input media streams may be fed to the same video decoding instance thanks to the block called "Input formatting" in Figure 1.

This clause defines the binding for several video codecs to realize the operations on input video streams.

## Operations on input media streams

### General

The input formatting function in Figure 1 provides several operations on media streams and video objects. The input formatting function results in one or more elementary streams conforming to the profile, tier, level or any other performance constraints of the video decoder instance expected to consume them including buffer fullness of the hypothetical reference decoder model. These operations are defined in an atomic way such that more complex operations can be achieved by combining them as long as the final output consists of valid elementary streams. The actual implementation of those combined operations is out-of-scope of this specification and can be subject to optimization by the implementers. Example of possible implementations are provided in Annex D.

A media stream contains one or more video objects, and a video object is contained into one elementary stream. Each video object in an elementary stream provides information for enabling the defined operations such as a mean to determine the location and the dimension of the video object in the picture, the number of luma and chroma samples in the video object, the bit depth of the coded picture of the video object and so on.

### Concepts

MediaStream a type of media stream

ElementaryStream a type of elementary stream

AccessUnit a type of access unit

VideoObjectIdentifier a type of video object identifier

VideoObjectSample a type of video object sample

### Filtering by video object identifier

#### Definition

Function: Filtering

Definition:

Input: one media stream with at least one video object

the identifier of the selected video object to be extracted

Output: one elementary stream with one video object which corresponds to the selected one

Signature: ElementaryStream output\_stream filtering(MediaStream input\_stream, VideoObjectIdentifier id)

For each i-th access unit in the input media stream, the function makes a copy of the access unit. Then, the function lists the video object samples present in this copied access unit. If a video object sample does not correspond to the video object identifier passed as input, the video object sample is removed from the copied access unit. Lastly, the copied access unit is appended to the output elementary stream as a new access unit.

NOTE The function implements a filtering process based on the selected object identifier, which is the original access units are first copied and then removed from the unwanted objects. This way, the operation does not need to create and initialize an empty access unit, and the properties of the input access units are passed on to the access units of the output stream.

#### Description

The filtering function extracts one video object from an input media stream and returns an elementary stream as output which comprises the selected video object.

In case the video object is a slice, the filtering function extracts this slice in every coded picture from the input media stream and passes it in the output elementary stream. This case is illustrated in Figure 6 wherein Figure 6 shows the video of the input media stream on the left and the output elementary stream on the right. During this operation, the SPS, PPS and slice header may need to be updated as required by the corresponding video coding specification to correctly signal the size of the video of the output elementary stream, the information about the slices and tiles layout and the video object identifier, e.g. the slice address.

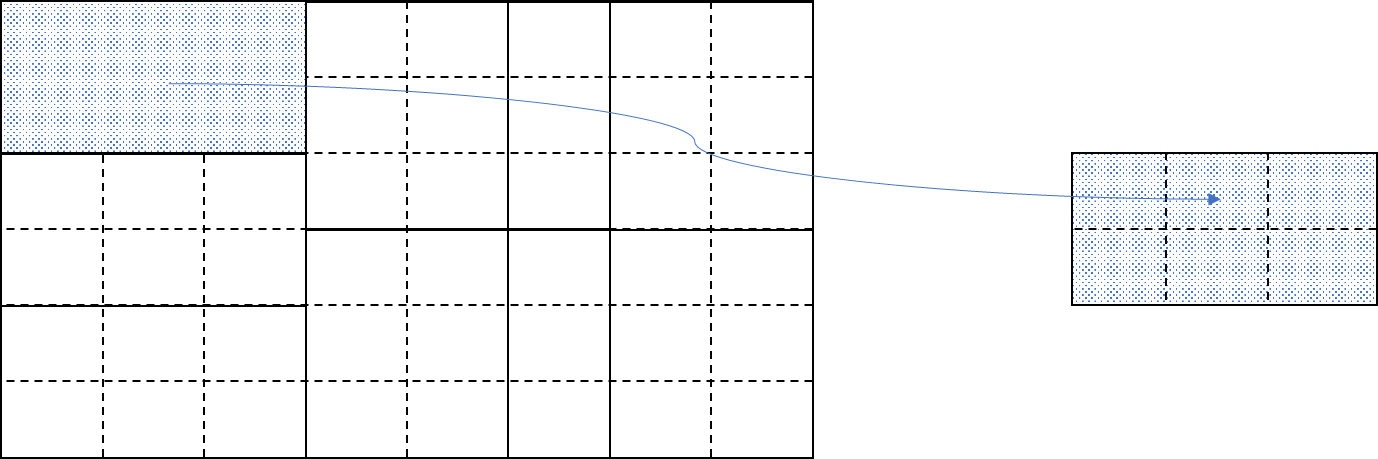


Figure 6 — Example of input and output video for the filtering function

### Inserting video objects

#### Definition

Function: Inserting

Definition:

Input: Two media streams containing at least one video object each

Output: One media stream with as many video objects as the sum of video objects in both input media streams

Signature: MediaStream output\_stream inserting(MediaStream input\_stream\_1, MediaStream input\_stream\_2)

For each i-th access unit in the first and second input media streams, the function makes a copy of the i-th access unit of the second input media stream. Then, the function lists the video object samples present in the i-th access unit from the first input media stream. Each video object sample is added to the copied access unit. Lastly, the copied access unit is appended to the output media stream as a new access unit.

NOTE 1 The inserting operation stops as soon as one of the two input media streams ends.

NOTE 2 The inserting operation is defined as the insertion of video objects of the first input media stream into the second input media stream. This way, the operation does not need to create and initialize an empty access unit, but the properties of the access units of the second input media stream are passed on to the access units of the output media stream.

#### Description

The inserting function takes the video objects from a first input media stream into a second input media stream and output the resulting output media stream which comprises the video objects from both first and second input media streams.

In case the video objects are slices, either the width or the height of the coded pictures of the input media streams are equal in order to maintain the rectangular shape of the video of the output media stream. In case the widths of the two input videos are equal, as shown in the Figure 7 a) and b), then the two videos are vertically stitched as shown in the diagram c). During this operation, the SPS, PPS and slice header may need to be updated to correctly signal the size of the video of the output media stream, the information about the slices and tiles layout and the video object identifiers, e.g. the slice addresses.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| a)   input media stream one | b)   input media stream two | c)   output elementary stream |

Figure 7 — Example of input and output video for the inserting function with identical width

If the heights of two videos are equal, as shown in Figure 8 a) and b), then the two videos are horizontally stitched as shown in the diagram c). During this operation as well, the SPS, PPS and slice header may need to be updated to correctly signal the size of the video of the output media stream, the information about the slices and tiles layout and the video object identifiers, e.g. the slice addresses.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| a)   input media stream one | b)   input media stream two | c)   output elementary stream |

Figure 8 — Example of input and output video for the inserting function with identical height

### Appending two video objects

#### Definition

Function: Appending

Definition:

Input: One media stream with at least two video objects

Output: One media stream with two video objects which are left and right spatial neighbors

Signature: MediaStream output\_stream appending(MediaStream input\_stream, VideoObjectIdentifier object\_id\_1,  
 VideoObjectIdentifier object\_id\_2)

For each i-th access unit in the input media stream, the function makes a copy of this i-th access unit. Then, the function sets the position of the video object samples that belong to the video object identified by the second video object identified right of the video object samples belonging to the object identified by the first video object identifier in this copied access unit. This positioning is done in such a way that the top boundaries of both video object samples are aligned. Lastly, the copied access unit is appended to the output media stream as a new access unit.

#### Description

The appending function positions a first video object right of a second video object in the decoded pictures of the output media stream, which contains those two video objects. The output media stream is a media containing at least the first and second video objects positioned as side-by-side neighbors.

In case the video object is a slice, the slices of both video objects in the input media stream have the same height as shown in the example Figure 9, diagram a). The slice on the right side is moved next to the slice on the left side in the output media stream. The slice in-between the two slices is moved to the right next to the two slices used as input of the operation as shown in the diagram b). During this operation, slice header may need to be updated to correctly signal the changes of the video object identifiers, e.g. the slice addresses.

|  |  |
| --- | --- |
|  |  |
| a)   input media stream | b)   output elementary stream |

Figure 9 — Example of input and output video for the appending function

### Stacking two video objects

#### Definition

Function: Stacking

Definition:

Input: One media stream with at least two video objects

Output: One media stream with two video objects which are top and bottom spatial neighbors

Signature: MediaStream output\_stream stacking(MediaStream input\_stream,   
 VideoObjectIdentifier object\_id\_1,  
 VideoObjectIdentifier object\_id\_2)

For each i-th access unit in the input media stream, the function makes a copy of this i-th access unit. Then, the function sets the position of the video object samples that belong to the video object identified by the second video object identifier below the video object samples belonging to the object identified by the first video object identifier in this copied access unit. This positioning is done in such a way that the left boundaries of both video object samples are aligned. Lastly, the copied access unit is appended to the output media stream as a new access unit.

#### Description

The stacking function positions a first video object on top of a second video object in the decoded pictures of the media stream that contains those two video objects. The output media stream contains at least the first and second video objects positioned as top-and-bottom neighbors.

In case the video object is a slice, the slices of both video objects in the input media stream have the same width as shown in the example Figure 10, diagram a). The slice on the right side is moved to below the slice on the left side in the output media stream. The slice at the below the slice on the left side and the one right next to it are moved to the right direction sequentially as shown in the diagram b). During this operation, the slice header may need to be updated to correctly signal the changes of the video object identifiers, e.g. the slice addresses.

|  |  |
| --- | --- |
|  |  |
| a)   input media stream | b)   output elementary stream |

Figure 10 — Example of input and output video for the stacking function

## Slice-based instantiation for ISO/IEC 23008-2 high efficiency video coding (HEVC)

### General

The high efficiency video coding (HEVC) is published under ISO/IEC 23008-2.

Table 2 provides the bindings of VDI concepts with the concepts defined in ISO/IEC 23008-2.

Table 2 — Correspondence between VDI concepts and HEVC concrete entities

|  |  |
| --- | --- |
| Concept | HEVC definitions |
| ElementaryStream | bitstream |
| AccessUnit | access unit |
| VideoObjectIdentifier | slice\_segment\_address |
| VideoObjectSample | slice segment |

In the remainder of the Clause **Error! Reference source not found.**, an HEVC elementary stream shall be a compliant bitstream according to ISO/IEC 23008-2.

### Media and elementary stream constraints

* + - 1. **General media stream constraints**

A HEVC media stream used as an instantiation of the media stream in Clause 7.2 shall obey the following rules:

* There shall be exactly one slice per tile and exactly one tile per slice.
* The tiling grid shall be constant for each entire coded video sequence.
* Each tile shall be motion-constrained as specified in the semantics of the temporal motion-constrained tile sets SEI message of ISO/IEC 23008-2.
* dependent\_slice\_segments\_enabled\_flag should be absent or, if present, equal to 0.

All the active SPSs of the HEVC input media stream shall be constrained as follows:

* general\_progressive\_source\_flag shall be equal to 1.
* general\_frame\_only\_constraint\_flag shall be equal to 1.
* general\_interlaced\_source\_flag shall be equal to 0.
* init\_qp\_minus26 in the PPS shall be set to the same value across all HEVC input media streams.
* The reference picture set (RPS) shall be the same across all HEVC input media streams .

## Layer-based instantiation for ISO/IEC 23090-3 versatile video coding (VVC)

### General

The versatile video coding (VVC) is published under ISO/IEC 23090-3.

Table 3 provides the bindings of VDI concepts with the concepts defined in ISO/IEC 23090-3.

Table 3 — Correspondence between VDI concepts and VVC concrete entities

|  |  |
| --- | --- |
| Concept | VVC definitions |
| ElementaryStream | bitstream |
| AccessUnit | access unit |
| VideoObjectIdentifier | nuh\_layer\_id |
| VideoObjectSample | picture unit |

In the remainder of the Clause **Error! Reference source not found.**, an VVC elementary stream shall be a compliant bitstream according to ISO/IEC 23090-3 and the independent layer info SEI message shall be defined as specified in **Error! Reference source not found.**.

### Media and elementary stream constraints

#### General media stream constraints

A VVC media stream used as an instantiation of the media stream in Clause 7.2 shall obey the following rules:

* There shall be at least one VPS in the media stream and the parameters in each VPS shall be as follows:
  + When present, the flag vps\_all\_independent\_layers\_flag shall be set to 1.
* The value of sh\_picture\_header\_in\_slice\_header\_flag shall be equal to 0 for all coded slices.
* When present, the value of vps\_num\_output\_layer\_sets\_minus2 shall be equal to 0.

#### Media and elementary stream constraints for input formatting functions

##### Constraints for the filtering function

A VVC input media stream passed as argument of the filtering function shall comply to these rules in addition to the rules in 7.4.2.1:

* There shall be VCL NAL units with at least two different nuh\_layer\_id values.
* One of the at least two different nuh\_layer\_id values shall be equal to the object identifier passed as argument of the filtering function.

A VVC elementary stream generated as output of the filtering function shall comply with these rules:

* The number of access units in the output elementary stream shall be equal to the number of access units in the input elementary stream.
* The number of VCL NAL units in the output elementary stream is equal to the number of VCL NAL units with nuh\_layer\_id equal to object identifier passed as argument of the function.
* For each VCL NAL unit in the output elementary stream, there shall exist a VCL NAL unit in the input elementary stream that is bit exact identical.
* All the NAL units in the output elementary stream shall have the same nuh\_layer\_id value and this nuh\_layer\_id value shall be equal to the object identifier passed as argument of the function.

##### Constraints for the inserting function

Two VVC input media streams passed as argument of the inserting function shall comply to these rules in addition to the rules in 7.4.2.1:

* The nuh\_layer\_id value of each NAL unit in the first input media stream shall be different from any nuh\_layer\_id value present in the second input media stream.
* If a SPS or PPS in the first input media stream has the same identifier than a SPS or PPS in the second input media stream, then those two SPSs or two PPSs shall have the same payload.

A VVC media stream generated as output of the inserting function shall comply with these rules in addition to the rules in 7.4.2.1:

* For each i-th access unit in the output media stream, the number of VCL NAL units in this access unit is equal to the total number of VCL NAL units in the i-th access units of both input media streams.
* For each VCL NAL unit of the i-th access unit in the output media stream, there shall exist a VCL NAL unit in the i-th access unit of one of the two input media streams that is bit exact identical.

##### Constraints for the appending function

A VVC input media stream passed as argument of the appending function shall comply to these rules in addition to the rules in 7.4.2.1:

* There shall be VCL NAL units with at least two different nuh\_layer\_id values.
* Two of the at least two different nuh\_layer\_id values shall be equal to the two object identifiers passed as arguments of the appending function.

A VVC media stream generated as output of the appending function shall comply with these rules in addition to the rules in 7.4.2.1:

* The number of VCL NAL units in the output media stream is equal to the number of VCL NAL units in the input media stream.
* For each VCL NAL unit in the output media stream, there shall exist a VCL NAL unit in the input media stream that is bit exact identical.
* There shall be an independent layer info SEI message whose nuh\_layer\_id is equal to the first video object identifier.
* There shall be an independent layer info SEI message whose nuh\_layer\_id is equal to the second video object identifier.
* The independent layer info SEI message whose nuh\_layer\_id is equal to the first video object identifier shall have its boundary\_identifier\_east value equal to the boundary\_identifier\_west value of the independent layer info SEI message whose nuh\_layer\_id is equal to the second video object identifier.

##### Constraints for the stacking function

A VVC input media stream passed as argument of the stacking function shall comply to these rules in addition to the rules in 7.4.2.1:

* There shall be VCL NAL units with at least two different nuh\_layer\_id values.
* Two of the at least two different nuh\_layer\_id values shall be equal to the two object identifiers passed as arguments of the appending function.

A VVC media stream generated as output of the stacking function shall comply with these rules in addition to the rules in 7.4.2.1:

* The number of VCL NAL units in the output media stream is equal to the number of VCL NAL units in the input media stream.
* For each VCL NAL unit in the output media stream, there shall exist a VCL NAL unit in the input media stream that is bit exact identical.
* There shall be an independent layer info SEI message whose nuh\_layer\_id is equal to the first video object identifier.
* There shall be an independent layer info SEI message whose nuh\_layer\_id is equal to the second video object identifier.
* The Independent layer info SEI message whose nuh\_layer\_id is equal to the first video object identifier shall have its boundary\_identifier\_south value equal to the boundary\_identifier\_north value of the independent layer info SEI message whose nuh\_layer\_id is equal to the second video object identifier.

## Slice-based instantiation for ISO/IEC 23094-1 essential video coding (EVC)

### General

The essential video coding (EVC) is published under ISO/IEC 23094-1.

Table 4 provides the bindings of VDI concepts with the concepts specified in ISO/IEC 23094-1.

Table 4 — Correspondence between VDI concepts and EVC concrete entities

|  |  |
| --- | --- |
| Concept | EVC definitions |
| ElementaryStream | bitstream |
| AccessUnit | access unit |
| VideoObjectIdentifier | the smallest value of the ID of the tiles in a slice |
| VideoObjectSample | slice |

In the remainder of the Clause **Error! Reference source not found.**, an EVC elementary stream shall be a compliant bitstream according to ISO/IEC 23094-1.

### Media and elementary streams constraints

#### General media stream constraints

An EVC media stream used as an instantiation of the media stream in Clause 7.2 shall obey the following rules:

* There shall be at least two independently decodable slices whose smallest value of the ID of the tiles in each slice that are different.

#### Media and elementary stream constraints for input formatting functions

##### Constraints for the filtering function

An EVC input media stream passed as argument of the filtering function shall comply with the following rules:

* One of the smallest values of the ID of the tiles in each slice shall be equal to the object identifier passed as argument of the filtering function.

An EVC elementary stream generated as output of the filtering function shall comply with the following rules:

* The number of access units in the output elementary stream shall be equal to the number of access units in the input media stream.
* The number of VCL NAL units in the output elementary stream is equal to the number of VCL NAL units with the smallest value of the ID of the tiles in the slice equal to object identifier passed as argument of the function.
* For each VCL NAL unit in the output elementary stream, there shall exist a VCL NAL unit in the input media stream that is bit exact identical.
* All the NAL units in the output elementary stream shall have the same smallest value of the ID of the tiles in the slice value and such value shall be equal to the object identifier passed as argument of the function.

##### Constraints for the inserting function

Two EVC input media streams passed as argument of the inserting function shall comply with the following rules:

* Al least one of the values of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples of the two media streams shall be identical.
* If the values of pic\_width\_in\_luma\_samples are identical, then the values of num\_tile\_columns\_minus1 shall be identical.
* If the values of pic\_height\_in\_luma\_samples are identical, then the values of num\_tiles\_row\_minus1 shall be identical.
* If a SPS or PPS in the first input media stream has the same identifier than a SPS or PPS in the second input media stream, then those two SPSs or two PPSs shall have the same payload.

An EVC media stream generated as output of the inserting function shall comply with the following rules:

* For each i-th access unit in the output media stream, the number of VCL NAL units in this access unit is equal to the total number of VCL NAL units in the i-th access units of both input media streams.
* For each VCL NAL unit of the i-th access unit in the output media stream, there shall exist a VCL NAL unit in the i-th access unit of one of the two input media streams that is bit exact identical.

##### Constraints for the appending function

An EVC input media stream passed as argument of the appending function shall comply with the following rules:

* At least two of the smallest values of the ID of the tiles in each slice shall be equal to the two object identifiers passed as arguments of the appending function.
* The height of the slices, number of tile rows of the tiles included in the slices when the uniform tile spacing is used, whose smallest values of the ID of the tiles in each slice are identical as arguments of the appending function are identical.

An EVC media stream generated as output of the appending function shall comply with the following rules:

* The number of VCL NAL units in the output media stream is equal to the number of VCL NAL units in the input media stream.
* For each VCL NAL unit in the output media stream, there shall exist a VCL NAL unit in the input media stream that is bit exact identical.

##### Constraints for the stacking function

An EVC input media stream passed as argument of the stacking function shall comply with the following rules:

* At least two of the smallest values of the ID of the tiles in each slice shall be equal to the two object identifiers passed as arguments of the appending function.
* The width of the slices, number of tile columns of the tiles included in the slices when the uniform tile spacing is used, whose smallest values of the ID of the tiles in each slice are identical as arguments of the appending function are identical.

An EVC media stream generated as output of the stacking function shall comply with the following rules:

* The number of VCL NAL units in the output media stream is equal to the number of VCL NAL units in the input media stream.
* For each VCL NAL unit in the output media stream, there shall exist a VCL NAL unit in the input media stream that is bit exact identical.

## Picture-based instantiation for ISO/IEC 14496-10 advanced video coding (AVC)

### General

The advanced video coding (AVC) is published under ISO/IEC 14496-10.

Table 5 provides the bindings of VDI concepts with the concepts specified in ISO/IEC 14496-10.

Table 5 — Correspondence between VDI concepts and AVC concrete entities

|  |  |
| --- | --- |
| Concept | AVC definitions |
| ElementaryStream | bitstream |
| AccessUnit | access unit |
| VideoObjectIdentifier | bitstream interleaving order |
| VideoObjectSample | coded picture |

In the remainder of the Clause 7.6, an AVC elementary stream shall be a compliant bitstream according to ISO/IEC 14496-10.

### Media and elementary streams constraints

#### General media stream constraints

An AVC media stream used as an instantiation of the media stream in Clause 7.2 shall obey the following rules:

* There shall be one interleaving information SEI message in the first access unit of the media stream.
* There may be additional interleaving information SEI messages in other access units to signal changes in the interleaving information.

#### Media and elementary stream constraints for input formatting functions

##### Constraints for the filtering function

An AVC input media stream passed as argument of the filtering function shall comply with the following rules:

* The value of the syntax element ii\_number\_of\_bitstreams in all the interleaving information SEI messages present shall be equal or greater than 2.

An AVC elementary stream generated as output of the filtering function shall comply with the following rules:

* The number of access units in the output elementary stream shall be equal to the number of access units in the input media stream.
* The number of VCL NAL units in the output elementary stream is equal to the number of VCL NAL units associated with the bitstream identified by the bitstream interleaving order passed as argument of the function.
* For each VCL NAL unit in the output elementary stream, there shall exist a VCL NAL unit in the input media stream that is bit exact identical.

##### Constraints for the inserting function

Two AVC input media streams passed as argument of the inserting function shall comply with the following rules:

* TBD [Editor’s note: Further study needed]

An AVC media stream generated as output of the inserting function shall comply with the following rules:

* For each i-th access unit in the output media stream, the number of VCL NAL units in this access unit is equal to the total number of VCL NAL units in the i-th access units of both input media streams.
* For each VCL NAL unit of the i-th access unit in the output media stream, there shall exist a VCL NAL unit in the i-th access unit of one of the two input media streams that is bit exact identical.
* The value of the syntax element ii\_number\_of\_bitstreams in the first interleaving information SEI message shall be equal to 2.

##### Constraints for the appending function

An AVC input media stream passed as argument of the appending function shall comply with the following rules:

* The value of the syntax element ii\_number\_of\_bitstreams in all the interleaving information SEI messages present shall be equal or greater than 2.
* The two object identifiers passed as arguments of the appending function shall be in the range of 0 to the value of ii\_number\_of\_bitstreams\_minus\_1 included.

An AVC media stream generated as output of the appending function shall comply with the following rules:

* The number of VCL NAL units in the output media stream is equal to the number of VCL NAL units in the input media stream.
* For each VCL NAL unit in the output media stream, there shall exist a VCL NAL unit in the input media stream that is bit exact identical.

##### Constraints for the stacking function

An AVC input media stream passed as argument of the stacking function shall comply with the following rules:

* The value of the syntax element ii\_number\_of\_bitstreams in all the interleaving information SEI messages present shall be equal or greater than 2.
* The two object identifiers passed as arguments of the appending function shall be in the range of 0 to the value of ii\_number\_of\_bitstreams\_minus\_1 included.

An AVC media stream generated as output of the stacking function shall comply with the following rules:

* The number of VCL NAL units in the output media stream is equal to the number of VCL NAL units in the input media stream.
* For each VCL NAL unit in the output media stream, there shall exist a VCL NAL unit in the input media stream that is bit exact identical.

1. (normative)  
     
   Control interface IDL definition
   1. General

The control interface to the video decoding engine is specified using the IDL syntax specified in ISO/IEC 19516.

The control interface is available as electronic attachment at <https://standards.iso.org/iso-iec/23090/-13/ed-1/en/>.

1. (informative)  
     
   OpenMAX IL VDI extension header
   1. General

The control interface to the video decoding engine is defined for the Open MAX IL interface.

The source code of the extension is available as electronic attachment at <https://standards.iso.org/iso-iec/23090/-13/ed-1/en/>.

1. (normative)  
     
   Supplemental enhancement information (SEI) syntax and semantics
   1. VDI SEI envelope
      1. **General**

This Clause defines a generic envelope for carrying SEI messages defined in this document. Some of the VDI SEI messages may only apply to certain video coding specifications.

The VDI SEI envelope is registered as a SEI payload in ISO/IEC 23090-3.

* + 1. **VDI SEI envelope syntax**

Table C.1 —defines the syntax of the VDI SEI envelope.

1. Syntax of VDI SEI envelope

|  |  |  |
| --- | --- | --- |
| **Syntax** | **Size** | **Type** |
| vdi\_sei\_envelope( payloadSize ) { |  |  |
| **vdi\_sub\_type** | 8 | unsigned integer |
| if(vdi\_sub\_type == 0) |  |  |
| independent\_layer\_info(payloadSize - 1) |  |  |
| else |  |  |
| reserved\_message(payloadSize - 1) |  |  |
| } |  |  |

* + 1. **VDI SEI envelope semantics**

vdi\_sub\_type indicates the payload type carried in the VDI SEI envelope.

* 1. Independent layer info SEI message
     1. **Independent layer info SEI message syntax**

Table C.2 —defines the syntax of the independent layer info SEI message.

1. Syntax of independent layer info SEI message

|  |  |  |
| --- | --- | --- |
| **Syntax** | **Size** | **Type** |
| independent\_layer\_info( payloadSize ) { |  |  |
| **boundary\_identifier\_north\_present\_flag** | 1 | bit |
| if(boundary\_identifier\_north\_present\_flag) |  |  |
| **boundary\_identifier\_north** | 16 | unsigned integer |
| **boundary\_identifier\_east\_present\_flag** | 1 | bit |
| if(boundary\_identifier\_east\_present\_flag) |  |  |
| **boundary\_identifier\_east** | 16 | unsigned integer |
| **boundary\_identifier\_south\_present\_flag** | 1 | bit |
| if(boundary\_identifier\_south\_present\_flag) |  |  |
| **boundary\_identifier\_south** | 16 | unsigned integer |
| **boundary\_identifier\_west\_present\_flag** | 1 | bit |
| if(boundary\_identifier\_west\_present\_flag) |  |  |
| **boundary\_identifier\_west** | 16 | unsigned integer |
| } |  |  |

* + 1. **Independent layer info SEI message semantics**

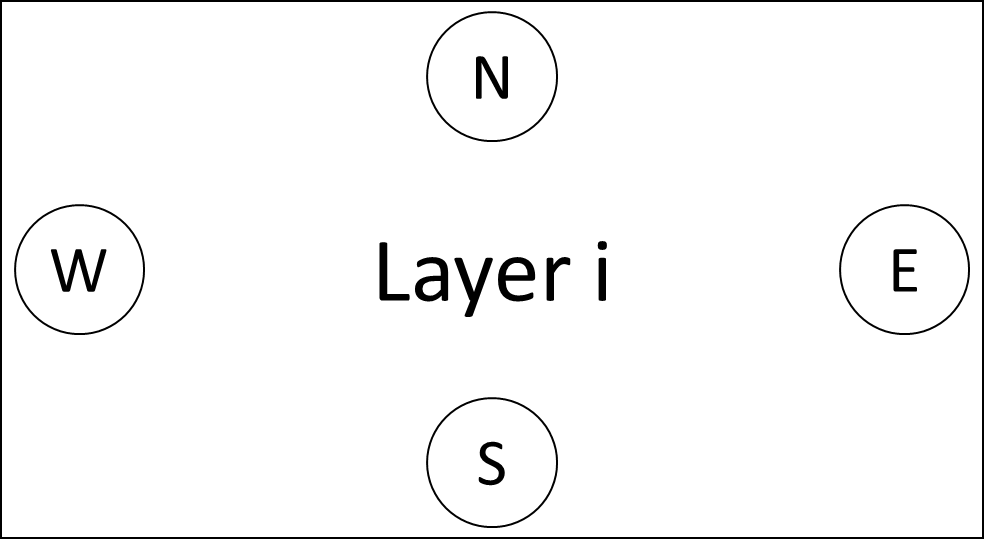
The independent layer info SEI message provides the spatial alignment of the different independent layers present in a bitstream by epxressing the relative positioning between these layers using matching boundary identifiers. In a multi-layer bitstream, there shall be at most two occurrences of a given boundary identier creating a pair of matching boudary identifier. In other words, a layer has at most one neighbor per boundary.

This SEI message may be extracted by the VDE and used for the output formatting function to correctly place the decoded pictures from each layer in the final output picture. In this case, the decoder instance may ignore this SEI message if present. Alternatively, the decoder instance may be capable of decoding a multi-layer bitstream as well as parsing the SEI message in which case no output formatting function is needed.

boundary\_identifier\_north\_present\_flag**,** boundary\_identifier\_east\_present\_flag,boundary\_identifier\_south\_present\_flag and boundary\_identifier\_west\_present\_flag equal to 1 specify that the SEI message contains a boundary identifier, respectively, for the north, east, south and west boundary.

boundary\_identifier\_north,boundary\_identifier\_east, boundary\_identifier\_south and boundary\_identifier\_west specifiy the boundary identifier, respecitvely, at the north, east, south and west boundary of the decoded picture of the associated layer; the associated layer being the layer whose nuh\_layer\_id is equal to the nuh\_layer\_id of the SEI message. If not present, the boundary identifiers respecitvely at the north, east, south and west boundary of the decoded picture of the associated layer are not defined.

Figure C.1 —illustrates the where the bounaries lie on the decoded picture associated with a layer.



1. Representation of where the boundaries are on the layers

For two layers, the i-th and j-th layers, when the pair of the boundary\_identifier\_north value of the i-th layer and the boundary\_identifier\_south value of the j-th layer are equal then the decoded picture of the i-th layer and the decoded picture of the j-th layer are adjacent in the composed output picture and they share a common boundary at the boundary north/south. For i-th and j-th layers, when the pair of the boundary\_identifier\_east value of the i-th layer and the boundary\_identifier\_west value of the j-th layer are equal then the decoded picture of the i-th layer and the decoded picture of the j-th layer are adjacent in the composed output picture and they share a common boundary at the east/west boundary.

Two decoded pictures adjacent by the north/south boundary are aligned on their west boundary in the final output picture. Two decoded pictures adjacent by the east/west boundary are aligned on their north boundary in the final output picture.

All the independent layer info SEI message present in the layers of an OLS shall collectively describe a 4-connected graph and each layer of the OLS shall be connected to the graph.

* + 1. **Process for generating the aggregated output picture**

The process for generating the final output picture is informative. The following section provides the expected operations performed for generating the final output picture based on the decoded pictures of each layer from a selected OLS:

* For each access unit:
  + If VPS present, parse VPS and store the list of layers in the bitstream.
  + For each present PPS, determine the size in luma samples of the corresponding layer.
  + For each present independent layer info SEI message, parse the payload and store the boundary identifiers for the corresponding layer.
  + If any of VPS, PPS or Independent layer info SEI message is present in the current access unit, calculate the horizontal, XPos, and vertical, YPos, positions of the top-left corner of each cropped decoded picture per layer in the final output picture. An example step sequence to calculate XPos and YPos is as follows:
    - For each layer:
      * Parse the list of cropped picture size and the boundary identifier.
      * Identify the north, east, south, west neighbouring layers.
    - Place each layer in a grid, with each grid cell corresponding to a north, east, south, west value for each layer that matches the corresponding neighboring value, respectively south, west, north, east.
    - The value XPos for each layer corresponds to the sum of the widths of each layer in the same row of the grid, left of the current layer. The value YPos for each layer is corresponds to the sum of the heights of each layer in the same column of the grid, above the current layer.
  + Initialize a picture buffer of size FinalWidth of width and FinalHeight of height for the final output picture where FinalWidth and FinalHeight are the width and height of the final output picture when all the layers are concatenated according to the defined graph.
  + For each picture unit:
    - Decode the coded picture.
  + If pictures are ready for output:
    - For each layer in selected OLS.
      * Apply conformance window cropping on the decoded picture of the current layer.
      * Retrieve XPos and YPos, the positions of the current layer in the final output picture in luma sample.
      * Copy cropped decoded picture in final output picture buffer at position (XPos, YPos) corresponding to the top-left corner of the cropped decoded picture.

If at the end of this process, the combination of all the decoded pictures does not provide decoded sample values for all the samples of the final output picture, the implementation determines the value to be used for these unused samples.

NOTE Since the further process only extracts the decoded samples from the final output picture, the sample value of the unused samples is not relevant from a normative perspective.

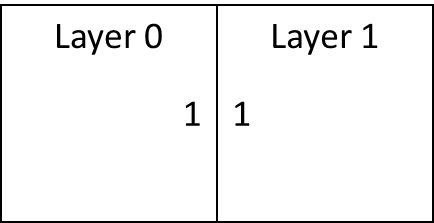
* + 1. Examples of video object positioning
       1. Appending

This example appends layer 0 to layer 1. Table C.3 —gives the properties of each layer.

1. Properties of layer 0 and layer 1

|  |  |  |  |
| --- | --- | --- | --- |
| Sequence | Layer | Resolution | Boundary identifiers (N, E, S, W) |
|  | 0 | 416 x 240 | 0 1 0 0 |
|  | 1 | 416 x 240 | 0 0 0 1 |

Figure C.2 —depicts the signaled connected map.



1. Connected map of layer 0 and layer 1

Based on this configuration, Table C.4 —presents the properties of the final output pictures.

1. Properties of the final output pictures

|  |
| --- |
| Resolution |
| 832 x 240 |
| **Output sequence** |
|  |

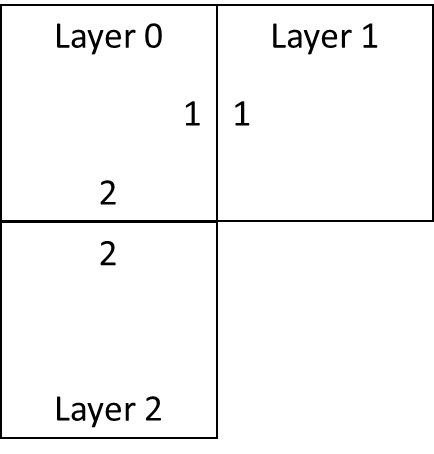
* + - 1. Appending and stacking

This example appends layer 0 next to layer 1 and stacks layer 0 on top of layer 2. Table C.5 —gives the properties of each layer.

1. Properties of layer 0, layer 1 and layer 2

|  |  |  |  |
| --- | --- | --- | --- |
| Sequence | Layer | Resolution | Boundary identifiers (N, E, S, W) |
|  | 0 | 416 x 240 | 0 1 2 0 |
|  | 1 | 416 x 240 | 0 0 0 1 |
|  | 2 | 832x480 | 2 0 0 0 |

Figure C.3 —depicts the signaled connected map.



1. Connected map of layer 0 and layer 1

Based on this configuration, Table C.6 —presents the properties of the final output pictures.

1. Properties of the final output pictures

|  |
| --- |
| Resolution |
| 832 x 720 |
| **Output sequence** |
|  |

* 1. Interleaving information SEI message
     1. **Interleaving information SEI message syntax**

|  |  |
| --- | --- |
| interleaving\_information( payloadSize ) { | **Descriptor** |
| **ii\_cancel\_flag** | u(1) |
| if( ! ii\_cancel\_flag) { |  |
| **ii\_number\_of\_bitstreams** | ue(v) |
| **ii\_interleaving\_pattern** | u(8) |
| if(ii\_interleaving\_pattern == 3) { |  |
| **ii\_number\_of\_pattern\_slots** | ue(v) |
| for(i=0; i< ii\_number\_of\_bitstreams; i++) |  |
| **ii\_pattern\_slot\_value**[ i ] | ue(v) |
| } |  |
| } |  |
| } |  |

[Editor’s note: The spatial position of the video object for AVC binding is for further study.]

[Editor’s note: The carriage of this SEI in the VDI enveloppe and its registration in the AVC specification is for further study.]

* + 1. **Independent layer info SEI message semantic**

**ii\_cancel\_flag** equal to 1 indicates that the SEI message cancels the persistence of any previous interleaving information SEI message. When equal to 0, the information that should be considered follows.

**ii\_number\_of\_bitstream\_minus1** plus 1 specifies the number of bitstreams that are interleaved.

**ii\_interleaving\_pattern** indicates the pattern with which the bitstreams are interleaved. The value 0 indicates that the NAL units of each bitstream are arranged in the same repetitive pattern, i.e. a,b,c,a,b,c etc… Value 1 indicates that the NAL units of each bitstream are arranged in symmetrical pattern, i.e. a,b,c,c,b,a,a,b,c,c,b,a, etc. Value 2 indicates that the NAL units of each bitstream are arranged in a rotating pattern, i.e. a,b,c,b,c,a,c,a,b,a,b,c, etc. Value 3 indicates an explicit pattern. The other values are reserved for future use.

**ii\_number\_of\_pattern\_slots** specifies the number of slots in the signaled pattern.

**ii\_pattern\_slot\_value**[ i ] indicates for the i-th slot of the pattern the value comprised between 0 and ii\_number\_of\_bitstreams\_minus1 included.

* + 1. Example of AVC bitstream interleaving

Creating an AVC media stream corresponds to the interleaving of multiple AVC elementary streams into one.

In a first example, the number of NAL units per access unit in the input AVC elementary streams is the identical. The AVC elementary streams can thus be interleaved into a single sequence of NAL units as illustrated in Figure C.4.

A group of white rectangular boxes with black text

Description automatically generated

1. Example of AVC media stream

In a second example, the number of NAL units per access unit in the input AVC elementary streams is different but consistent across the respective bitstream as illustrated in Figure C.5.

A screenshot of a computer

AI-generated content may be incorrect.

1. Example of AVC media stream with different number of NAL units per access unit

In a third example, the number of NAL units per access unit in both input AVC elementary streams varies as illustrated in Figure C.6.

A screenshot of a computer

AI-generated content may be incorrect.

1. Example of AVC media stream with varying number of NAL units per access unit

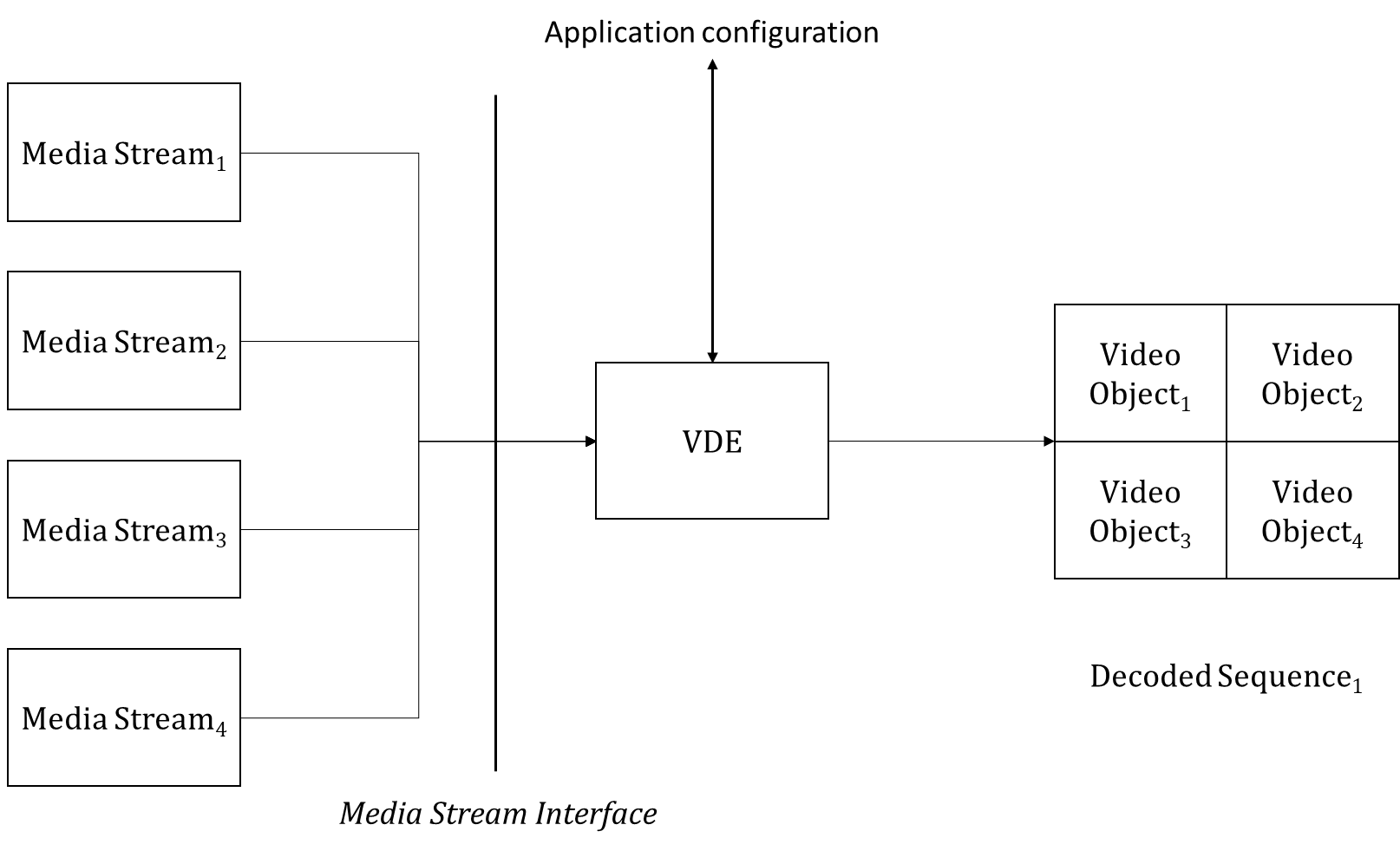
All the above examples can be signaled using the interleaving information SEI as specified in this document,

1. (informative)  
     
   Example implementations of input formatting operations
   1. General

The operations defined in Clause 7.2 as well as the associated input and output constraints provide the building blocks for the implementation of the input formatting function. The way a certain implementation converts the media streams to elementary streams based on the requested decoded sequences configuration is informative and left for optimization by the implementor as long as the output elementary streams meet the requirements of the elementary stream interface.

* 1. Creating a 2-by-2 video mosaic via application control

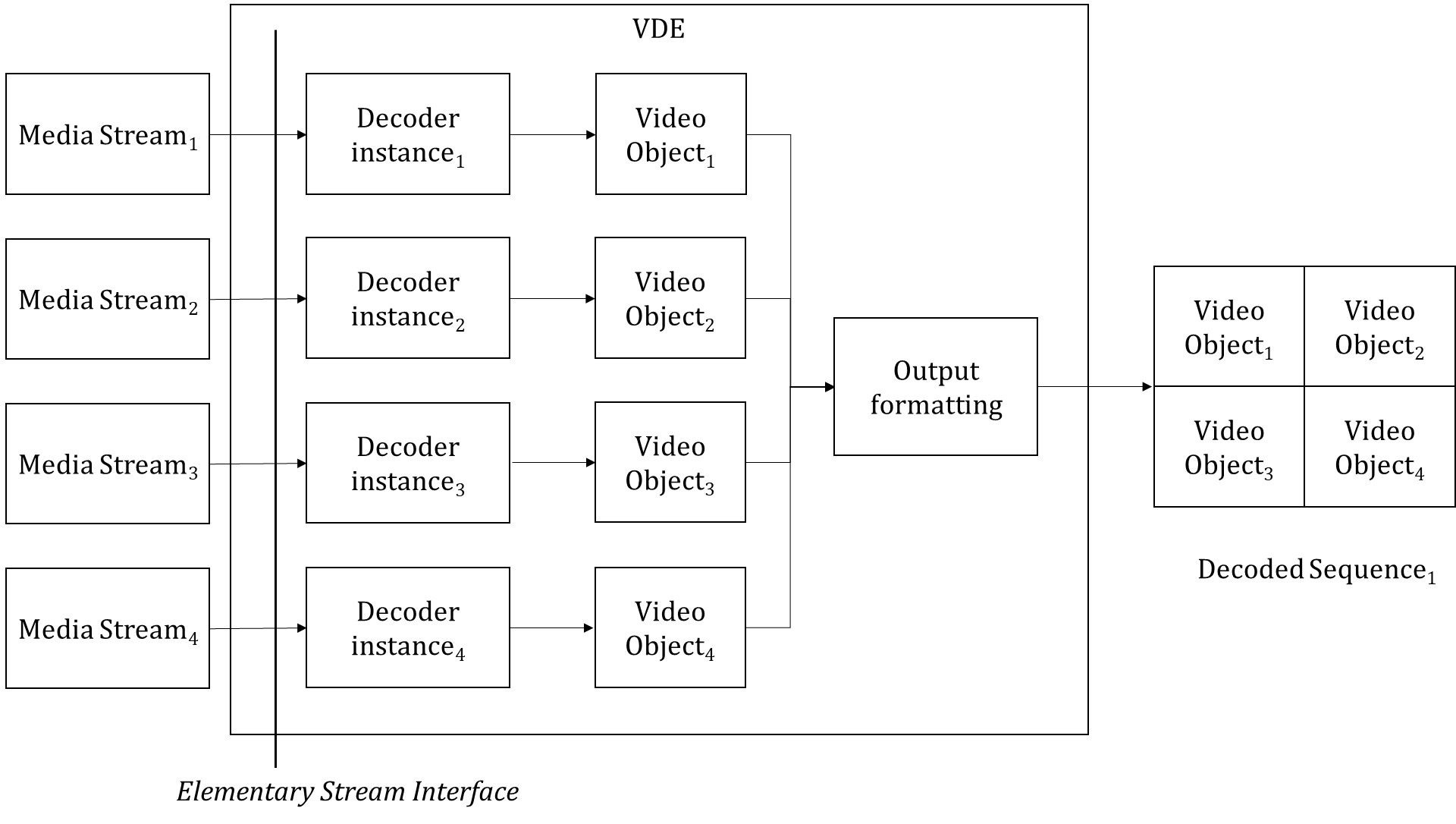
In this example, the goal is to take four video objects as input in four different media streams and obtain as output of the VDE one decoded sequence containing the four decoded video objects arranged in a 2 by 2 grid in each picture of the decoded sequence. We assume that the application is able to communicate to the VDE via an external communication channel to express how the media stream needs to be arranged in the decoded sequences. For the purpose of the illustration, consider that the four video objects have the same video resolution. Figure D.1 —depicts the input situation at the media stream interface, input of the VDE and the intended situation at the output of the VDE.



1. Mosaicking of 2-by-2 video objects with application-based VDE control

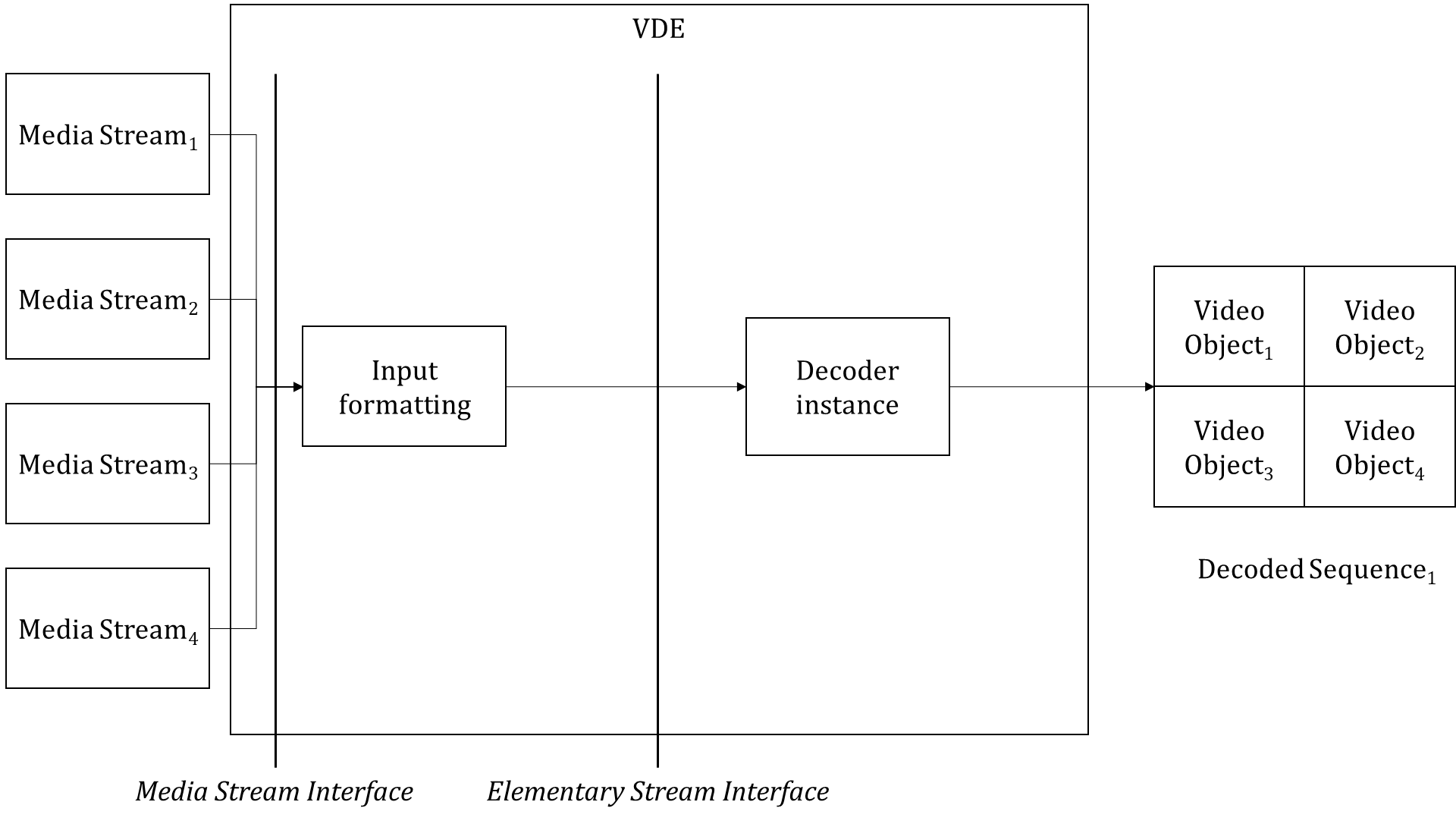
This document specifies that the output of the input formatting function is the elementary stream interface. That is, the input formatting function needs to output data streams which are elementary streams. Whether it should be one elementary stream or multiple elementary streams is implementation and platform dependent. For this example, assume two variants. The first one is to have one video decoder per media stream, the second one is to have one single decoder for the four media streams.

In the case of one video decoder per media stream, the input formatting function is effectively an identity operation, i.e. each media stream is passed on to one decoder instance. That also means that each media stream is in this case also an elementary stream which is allowed by the definition of a media stream. Then the VDE runs those four decoder instances in parallel, collects each output picture and assemble them into the 2-by-2 arrangement for each set of temporally collocated decoded pictures as shown on Figure D.2 —.



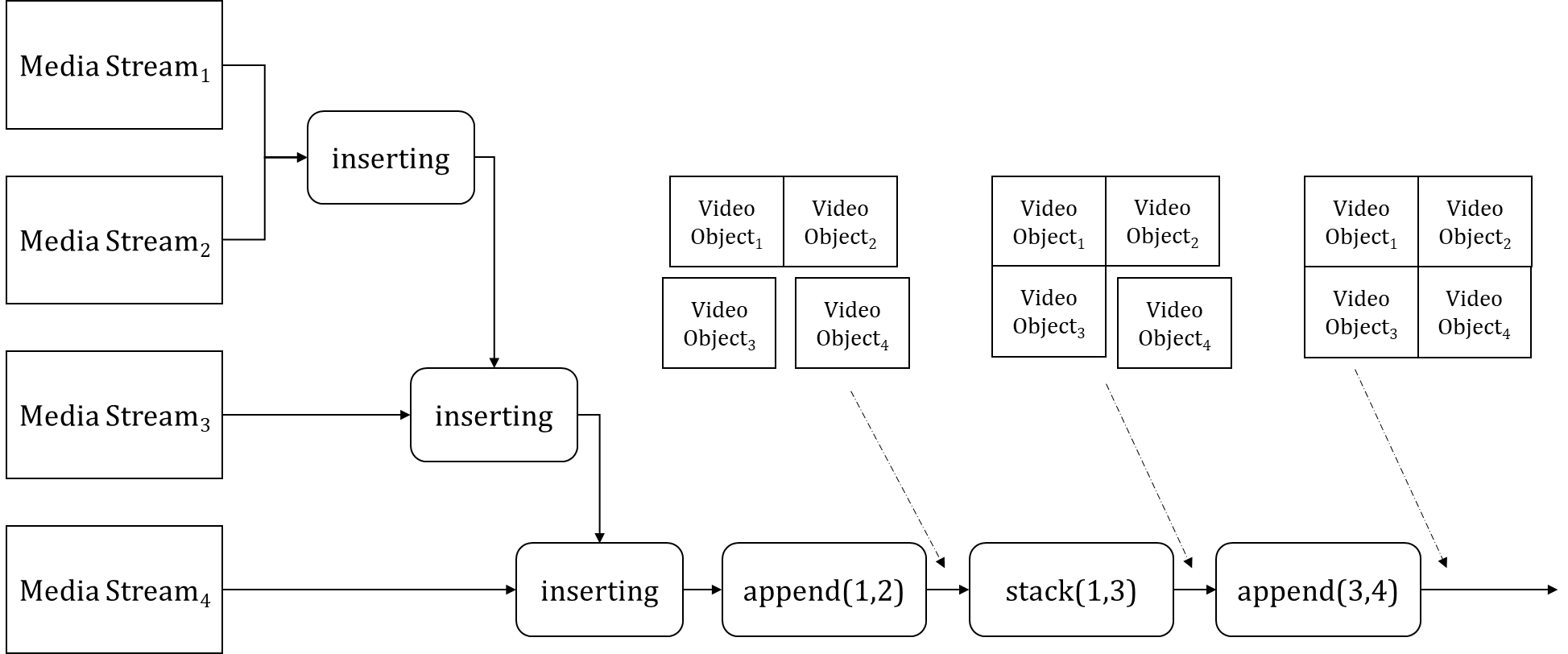
1. Mosaicking of 2-by-2 video objects with four decoder instances

In the case of one video decoder for the four media streams, the input formatting function takes care of creating a single elementary stream out of the four input media streams. From that point onwards, the VDE runs a conventional pipeline with a single decoder and outputs the decoded picture from the decoder instance without the need of further processing before the output of the VDE. This case is depicted in Figure D.3 —.



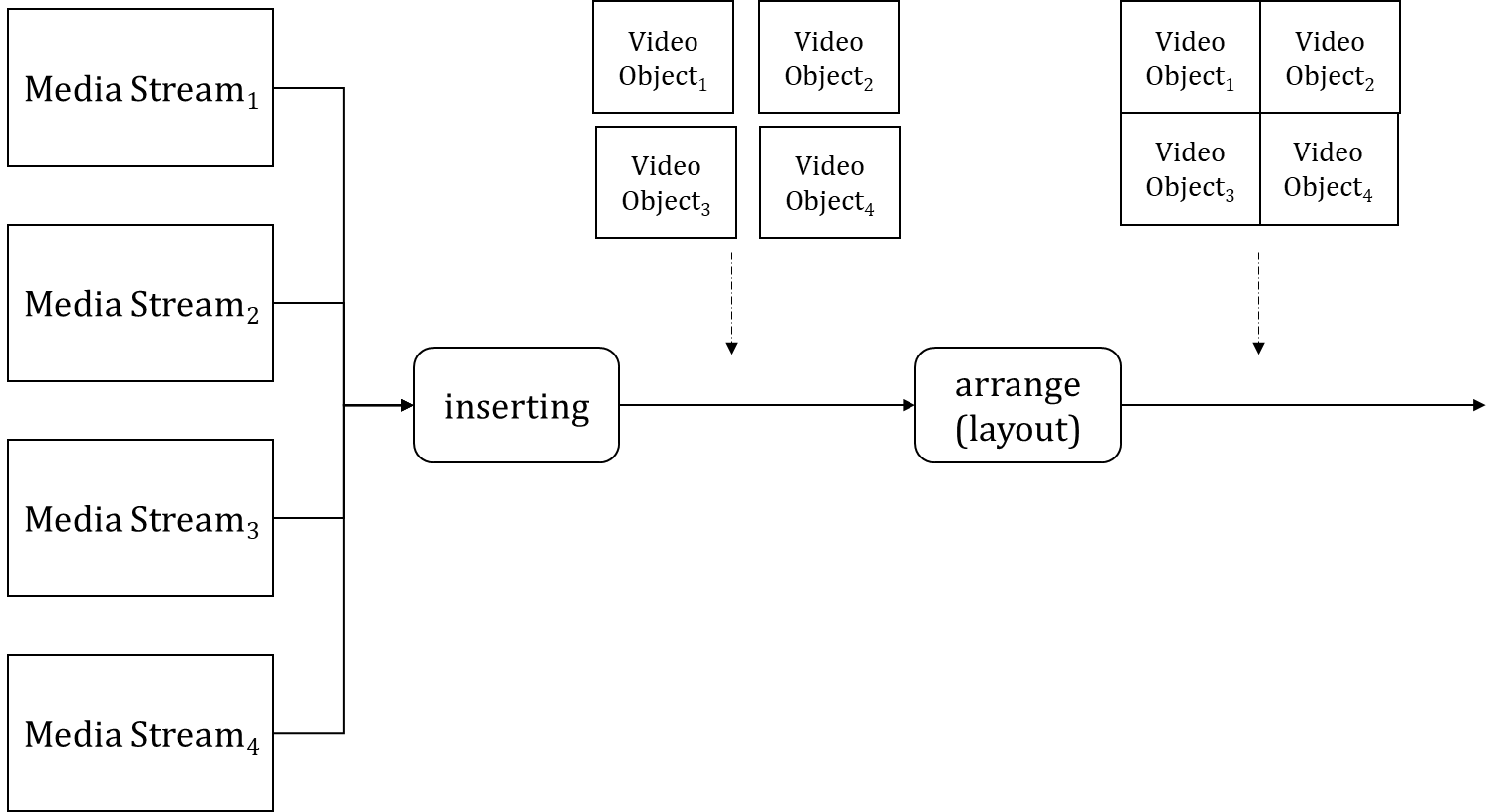
1. Mosaicking of 2-by-2 video objects with one decoder instance

Regarding the actual implementation of the input formatting function for this variant, there can be different approaches. The first one is to effectively implement the operations insert and append. In this case, the implementation would cascade those operations in a graph as shown in Figure D.4.



1. Example implementation for formatting 4 media streams with different steps

Alternatively, an implementation may support an inserting function that receives several media streams as input and generates an elementary stream in a single function call. In addition, this implementation may also have a proprietary function, say called “arrange” so that the arrangement can be established in the elementary stream in one function call. This other variant is depicted in Figure D.5.

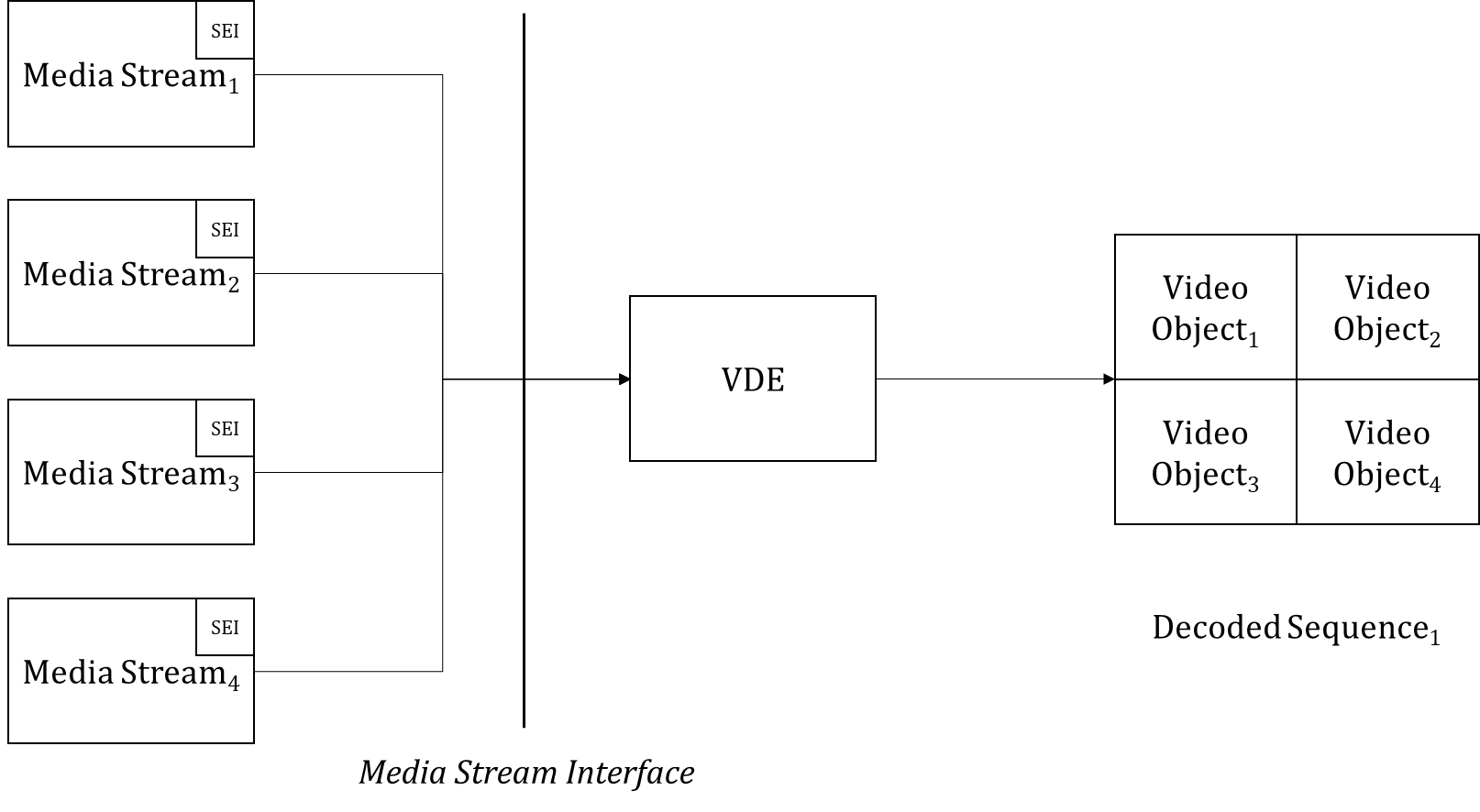


1. Example implementation for formatting 4 media streams with 2 steps

In yet another implementation, there may only be one function combining those media streams, arranging them and outputting the elementary stream.

* 1. Creating a 2-by-2 video mosaic via SEI control

This example shows the same intended output result as in Clause D.2. The difference here is that there is no communication assumed between the VDE and the application as to how to arrange the video objects together in the decoded sequence. Instead, this arrangement configuration is given by the independent layer info SEI messages contained in each media stream. This information is used for applying the append operations. Figure D.6 —illustrates this example implementation via SEI control.



1. Mosaicking of 2-by-2 video objects with SEI-based VDE control
2. (informative)  
     
   Brief description of OpenMAX IL functions
   1. Decoder engine control interface
      1. OMX\_Init() and OMX\_Deinit()

Each OpenMAX IL client needs to call this method as their first call into OpenMAX IL. This function initializes the OMX core engine prior to any usage of it. Once done, the engine needs to be released by calling the OMX\_Deinit() function.

OMX defines a naming convention for the component names with the following format: OMX.<vendor\_name>.<vendor\_specified\_convention>. Once the instance is no longer needed, the OMX\_FreeHandle() is called to free all related resources.

The function can be called multiple times with the same component name to create multiple instances of the component.

* + 1. OMX\_GetHandle() and OMX\_FreeHandle()

The OMX\_GetHandle() method is used to locate the requested component through its provided name. If the requested component is available, the OMX core engine will invoke the components methods to fill the component handle and setup the callbacks. The OpenMAX AL is the interface that will be used by the application to perform media playback and processing. However, the OpenMAX IL interface is the interface that provides direct access to video decoder components and their capabilities. That is why we focus on the OpenMAX IL interface for the purpose of understanding the missing features towards a flexible multi-video decoder platform and its interface for 6DoF applications.

* + 1. OMX\_SetupTunnel() and OMX\_TeardownTunnel()

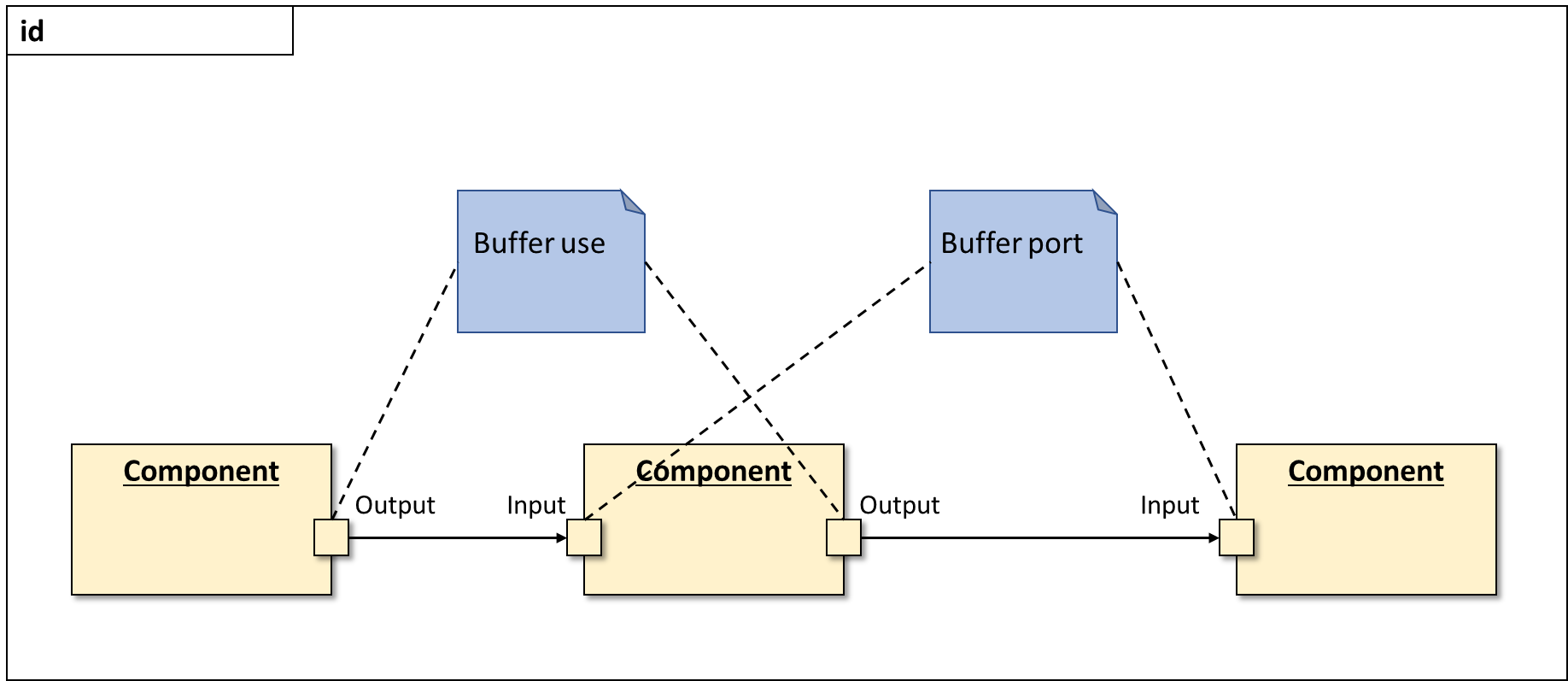
A Tunnel is used to connect the input and output ports of two connected components. The OMX\_SetupTunnel() is used to establish a tunnel connecting an output port to the input port of another component. When creating the tunnel, the components negotiate a compatible input/output format for the connected ports. When no longer needed, the application calls the OMX\_TeardownTunnel() to tear down the tunnel.

* 1. Decoder instance interface
     1. Methods

The components communicate among each other and with the application through buffer exchange. For this purpose, the OMX\_AllocateBuffer(), OMX\_UseBuffer(), OMX\_FillThisBuffer(), OMX\_EmptyThisBuffer(), and OMX\_FreeBuffer() are defined. These function calls are non-blocking.

A component asks a preceding component to fill an input buffer by calling the OMX\_FillThisBuffer() method and asks a succeeding component to retrieve the content of an output port buffer by calling the OMX\_EmptyThisBuffer() function. Only one buffer per tunnel is allowed and one of the two components acts a supplier of that buffer.

Figure E.1 —depicts an example of connected components and the buffer usage.

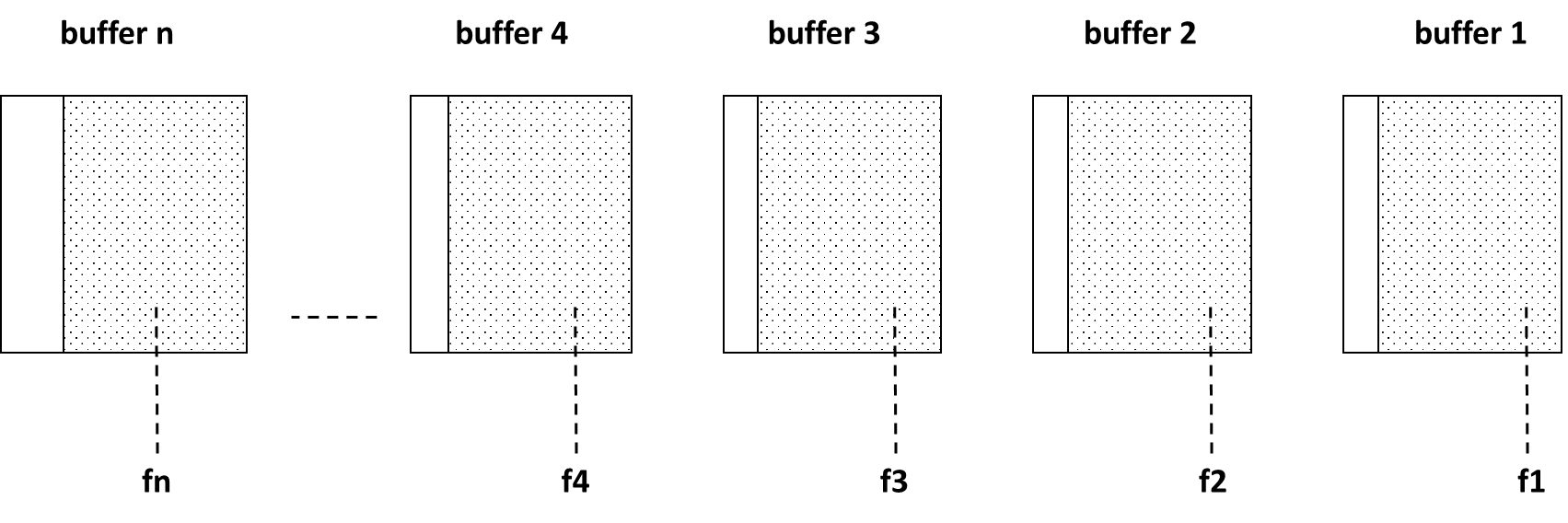


1. Example of connected components and the buffer usage

The OMX\_SetConfig() method is used to configure a component by the application. The application passes a structure that contains the configuration parameters to the component. The configuration parameters are published by each component and are component specific.

* + 1. Media input and output interface

The port configuration is used to define the format of the data to be transferred on a component port. The buffer header contains a reference to the buffer pBuffer, an offset inside that buffer nOffset, and the length of that buffer nFilledLen. Multiple buffers can be used to pass data, which allows for more flexibility in the communication between components, i.e. more than one frame can be exchanged at a time. Figure E.2 —shows an example.



1. Port configuration example

There is no requirement for frame alignment to buffer start. The application or preceding components provide frame alignment information as part of the buffer header using the OMX\_BUFFERFLAG\_ENDOFFRAME flag. It is also possible to signal sub-frame boundaries to identify NAL unit boundaries using the OMX\_BUFFERFLAG\_ENDOFSUBFRAME.

A timestamp is also provided by the buffer header for every buffer. The nTimestamp corresponds to the presentation timestamp of the first media sample that starts at the current buffer. If multiple samples are included in the current buffer, the start timestamp of the following samples is inferred from the nTimestamp and the sample duration. That information can then be propagated through the pipeline and may be passed to the application through the output buffer.

* + 1. Format of the OpenMAX IL buffer header

The buffer header structure is:

typedef struct OMX\_BUFFERHEADERTYPE

{

OMX\_U32 nSize;

OMX\_VERSIONTYPE nVersion;

OMX\_U8\* pBuffer;

OMX\_U32 nAllocLen;

OMX\_U32 nFilledLen;

OMX\_U32 nOffset;

OMX\_PTR pAppPrivate;

OMX\_PTR pPlatformPrivate;

OMX\_PTR pInputPortPrivate;

OMX\_PTR pOutputPortPrivate;

OMX\_HANDLETYPE hMarkTargetComponent;

OMX\_PTR pMarkData;

OMX\_U32 nTickCount;

OMX\_TICKS nTimeStamp;

OMX\_U32 nFlags;

OMX\_U32 nOutputPortIndex;

OMX\_U32 nInputPortIndex;

* + 1. Buffer flags defined in OpenMAX IL

The list of buffer flags is:

#define OMX\_BUFFERFLAG\_EOS 0x00000001

#define OMX\_BUFFERFLAG\_STARTTIME 0x00000002

#define OMX\_BUFFERFLAG\_DECODEONLY 0x00000004

#define OMX\_BUFFERFLAG\_DATACORRUPT 0x00000008

#define OMX\_BUFFERFLAG\_ENDOFFRAME 0x00000010

#define OMX\_BUFFERFLAG\_SYNCFRAME 0x00000020

#define OMX\_BUFFERFLAG\_EXTRADATA 0x00000040

#define OMX\_BUFFERFLAG\_CODECCONFIG 0x00000080

#define OMX\_BUFFERFLAG\_TIMESTAMPINVALID 0x00000100

#define OMX\_BUFFERFLAG\_READONLY 0x00000200

#define OMX\_BUFFERFLAG\_ENDOFSUBFRAME 0x00000400

#define OMX\_BUFFERFLAG\_SKIPFRAME 0x00000800

* + 1. Input/Output from/into GPU

OpenMAX IL introduces the possibility to use an EGL Image as an output buffer. An EGL Image is designed for sharing data between rendering-based EGL interfaces, such as OpenGL and the OpenMAX components. It is up to the component to implement the OMX\_UseEGLImage() to link the output to an EGL Image instead of a traditional buffer.

1. (informative)  
     
   Mapping on media source extensions (MSE)
   1. Overview

MSE is a set of extensions to the media source attributes of HTML5 video and audio elements. It enables flexible control of media streams through JavaScript code using the definition of MediaSource objects. A MediaSource object may have one or more SourceBuffer objects. Applications append media segments to the SourceBuffer objects. A SourceBuffer may have multiple tracks, which are decoded and played separately. 0shows an example setup of MSE which depicts the interface between the MSE API and the HTML5 media element.

A picture containing text, silhouette

Description automatically generated

**Key**

|  |  |  |  |
| --- | --- | --- | --- |
| MS | MediaSource |  |  |
| SB | SourceBuffer |  |  |
| TB | TrackBuffer |  |  |
| VDC | Video decoder |  |  |
| ADC | Audio decoder |  |  |
| ADV | Audio device |  |  |
| VT | Video tag |  |  |
| DR | Display region |  |  |

1. Overview of MSE media interfaces
   1. Mapping of VDI functions

Table F.1 shows the possible mapping of the VDI functions onto the MSE API.

1. Possible mapping of VDI onto MSE

|  |  |
| --- | --- |
| **VDI Functionality** | **MSE Mapping** |
| queryCurrentAggregate Capabilities() | MediaSource.queryCurrentAggregate Capabilities() a |
| getInstance() with grouping | MediaSource.addSourceBuffer() b |
| setConfig() CONFIG\_OUTPUT\_BUFFER | VideoTrack.setConfig() c |
| getParameter() and setParameter() | VideoTrack and AudioTrack, getParameter() and setParameter() d |
| a   A new method of the MediaSource object is used to query the current decode capabilities.  b   Tracks of the same type, e.g. VideoTracks, that belong to the same SourceBuffer are considered alternatives and only one is decoded and presented. When creating a new SourceBuffer, a group identifier for each track type may be provided. This grouping applies across all currently instantiated MediaSource objects. This allows for grouping of multiple decoder instances that belong to multiple HTML5 media elements.  c  New method of the HTML5 VideoTrack object.  d  New methods of HTML5 VideoTrack and AudioTrack objects. | |

In addition, an extension to the HTML5 video element would be needed to allow outputting data into buffers, e.g. WebGL buffers that are created through gl.createBuffer() functions.

An extension to the input byte stream format would also be needed to add support for raw media data, e.g. AVC raw media streams.

1. (informative)  
     
   Mapping on WebCodecs
   1. Overview

The WebCodecs API is a specification that offers a codec-agnostic encoding/decoding interface in the browser. The WebCodecs API uses an asynchronous processing model. Each instance of an encoder or decoder maintains an internal, independent processing queue. Methods named configure(), encode(), decode(), and flush() operate asynchronously by appending control messages to the end of the queue, while methods named reset() and close() synchronously abort all pending work and purge the processing queue.

Relevant WebCodecs API interfaces are listed in Table G.1.

1. WebCodecs interfaces

|  |  |
| --- | --- |
| **Interface** | **Description** |
| VideoDecoder | Decodes EncodedVideoChunk objects. |
| EncodedVideoChunk | Represents codec-specific encoded video bytes. |
| VideoFrame | Represents a frame of unencoded video data |
| VideoColorSpace | Represents the color space of a video frame. |
| ImageDecoder | Unpacks and decodes image data, giving access to the sequence of frames in an animated image. |
| ImageTrackList | Represents the list of tracks available in the image. |
| ImageTrack | Represents an individual image track. |

The VideoDecoder object can take as input objects of type EncodedVideoChunk, that can be read from BufferSource objects. It is required to have the timestamp of the chunk, i.e. media time of the first encoded frame in the chunk, and the chunk type. Typically, the chunk data field is of type Uint8Array.

The processing model of a video decoder in WebCodecs assumes decoding as an instantaneous process. Therefore, the group synchronization can be achieved by managing the queue of the decoder by monitoring the decodeQueueSize attribute and react accordingly to ondequeque events, by calling the decode function of the decoders in the same group.

* 1. Mapping of VDI Functions

Table G.2 shows the possible mapping of the VDI functions onto the WebCodecs API.

1. Possible mapping of VDI onto WebCodecs

|  |  |
| --- | --- |
| **VDI Functionality** | **WebCodecs Mapping** |
| queryCurrentAggregateCapabilities() | VDIVideoDecoder.queryCurrentAggregateCapabilities() a |
| getInstance() with grouping | new VDIVideoDecoder()b |
| setConfig() CONFIG\_OUTPUT\_BUFFER | VideoDecoder.configure(*VideoDecoderConfig*) c |
| getParameter() and setParameter() | VideoDecoder, getParameter() and setParameter() d |
| a   The new method queryCurrentAggregateCapabilities is built on top of isConfigSupported of the VideoDecoder object.  b   There is not native grouping for VideoDecoder objects in the WebCodecs. The VDIVideoDecoder class is an extended VideoDecoder class (of WebCodecs) to accomodate assigning a group to the decoder. The parameters of the constructor are left to the implementation.  c  The configure method of the VideoDecoder object is extended to match the setConfig method.  d  New methods of the VideoDecoder object. | |

* 1. Example implementations
     1. queryCurrentAggregateCapabilities

//Example config, used as input in the queryCurrentAggregateCapabilities function

const config = {

    codec: 'h265', // Test with common codec

    codedWidth: 1920, // Test with common resolution

    codedHeight: 1080

};

async queryCurrentAggregateCapabilities(config, flags) {

    const capabilities = {

        instances: null,

        bufferMemory: null,

        bitrate: null,

        maxSamplesPerSecond: null,

        maxPerformancePoint: null

    };

    if (flags.CAP\_INSTANCES\_FLAG) {

        // Assuming API or method to check instance limits

        capabilities.instances = 'Maximum instances not specified in WebCodecs API';

    }

if (flags.CAP\_BUFFER\_MEMORY\_FLAG) {

        // Assuming API or method to check buffer memory limits

        capabilities.bufferMemory = 'Buffer memory limits are not directly queryable via WebCodecs';

    }

    if (flags.CAP\_BITRATE\_FLAG) {

        capabilities.bitrate = 'Maximum bitrate not directly queryable, assumed from codec capability';

    }

if(flags.CAP\_MAX\_SAMPLES\_SECOND\_FLAG||flags.CAP\_MAX\_PERFORMANCE\_POINT\_FLAG) {

        const isSupported = await VideoDecoder.isConfigSupported(testConfig);

        if (flags.CAP\_MAX\_SAMPLES\_SECOND\_FLAG) {

            capabilities.maxSamplesPerSecond = isSupported.supported ? 'Supported' : 'Not supported';

        }

        if (flags.CAP\_MAX\_PERFORMANCE\_POINT\_FLAG) {

            capabilities.maxPerformancePoint = isSupported.supported ? 'Supported' : 'Not supported';

        }

    }

    return capabilities;

}

* + 1. VDIVideoDecoder class, with grouping and getInstance, constructor

class VDIVideoDecoder extends VideoDecoder {

    constructor({ output, error }) {

     super();

     this.output = output;

     this.error = error;

      this.decoder = new WebCodecs.VideoDecoder({ output: this.output, error: this.error });

}

static getInstance(component\_name, groupIdentifier) {

const decoder = new VDIVideoDecoder({ output: this.output, error: this.error });

this.groupId = groupIdentifier;

return decoder;

}

//add here the configure functionality as in G.3.3

//add here the getParameter and setParameter functionality as in G.3.4

}

* + 1. setConfig

setConfig(configParameters, configDataParameters) {

try {

decoder.configure(configDataParameters);

return true;

} catch (err) {

throw new Error(`Configuration failed: ${err}`);

}

}

* + 1. getParameter, setParameter

getParameter(configParameters) {

try {

return decoder.getParameter(configParameters);

} catch (err) {

throw new Error(`Parameter not supported: ${err}`);

}

}

setParameter(decoder, configParameters, value) {

try {

decoder.setParameter(configParameters, value);

return true;

} catch (err) {

throw new Error(`Parameter update failed: ${err}`);

}

}

1. (normative)  
     
   Levels
   1. Introduction

The levels convey the requirements of a set of media streams to meet in order to be properly decoded.

By comparing the capabilities of the underlying video decoding engine, an application can identify whether the device is capable of decoding a given set of media streams.

* 1. Definitions

[Editor’s note: This clause has been extracted in extenso from 3GPP TS 26.119. Further adaptation in the context of ISO/IEC 23090-13 is desirable.]

Concurrent video decoder instances are defined as follows.

For N video bitstreams encoded according to a video codec profile, decoding units flow into the coded picture buffer (CPB) for each stream according to a specified arrival schedule and are delivered by the common Hypothetical Stream Scheduler (HSS) that schedules the N bitstreams for decoding each of the units. For each access unit

* All data associated with an access unit is removed and decoded instantaneously by the instantaneous decoding process at CPB removal time of the access unit.
* Each decoded picture is placed in the Decoded Picture Buffer (DPB) for being referenced by the decoding process of this stream as well as for output and cropping.
* A decoded picture is removed from the DPB at the time that it becomes no longer needed for inter-prediction reference as well as the output time of the access unit is the largest of all decoded pictures remaining in the group of N decoders

Then at any point time,

* Each of the individual streams conforms to the signaled profile/level/tier and HRD parameters of the individual stream.
* The sum of the CPB size conforms to common profile/level/tier signaling
* The aggregate decoder processing speed (samples per seconds) conforms to common profile/level/tier signaling.
* The sum of the DPB size conforms to common profile/level/tier signaling
* The common DPB size conforms to common profile/level/tier signaling

A set of N concurrent decoder instances conforms to a given capabilities (defined in clause 7.1.2.2), if a set of up to N bitstreams encoded to be decodable by the HRD above, is decodable within the timing limits.

* 1. Level limits

Table 7 – Level limits

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Level** | **Max aggregated luma picture size MaxAggLumaPs (samples)** | **Max aggregated bit rate MaxAggBR (1000 bits/s)** | **Max aggregated luma sample rate MaxAggLumaSr (samples/sec)** | **Max aggregrated DB size MaxAggDB (1000 bits)** | **Max number of decoder instances** |
| 1.0 | 2 073 600 | 50 000 | 133 693 440 | 50 000 | 2 |
| 2.0 | 8 294 400 | 160 000 | 534 773 760 | 160 000 | 2 |
| 2.1 | 8 294 400 | 160 000 | 534 773 760 | 160 000 | 4 |
| 3.0 | 35 389 440 | 800 000 | 4 278 190 080 | 800 000 | 2 |
| 3.1 | 35 389 440 | 800 000 | 4 278 190 080 | 800 000 | 4 |
| 3.2 | 35 389 440 | 800 000 | 4 278 190 080 | 800 000 | 8 |

NOTE 1 The following HEVC levels were used to derive aggregated parameters:

* Level 1: Level 4.1 (high tier) @ 1,920×1,080
* Level 2: Level 5.1 (high tier) @ 3,840×2,160
* Level 3: Level 6.2 (high tier) @ 8,192×4,320

The max aggregated luma picture size in HEVC is calculated to be dividable by 16, this is a constraint that is ignored as not relevant for the levels in the present specification.

NOTE 2 The number of instances have been defined in such a way that each instance may operate at a minimum of 30fps and 1080p.

Bibliography

1. ISO/IEC 19516, Information technology — Object management group — Interface definition language (IDL) 4.2
2. ISO/IEC 14496-1, *Information technology — Coding of audio-visual objects — Part 1: Systems*
3. OpenMAX™ Integration Layer Application Programming Interface Specification Version 1.1.2, The Khronos Group Inc.
4. Vulkan® 1.3.235 - A Specification (with KHR extensions), The Khronos® Vulkan Working Group
5. *Media Source Extensions™, W3C Recommendation, 17 November 2016*