ISO/IEC JTC 1/SC 29/WG 03 N1509

**ISO/IEC JTC 1/SC 29/WG 03  
MPEG Systems   
Convenorship: KATS (Korea, Republic of)**

**Document type:** Output Document

**Title:** Technologies under consideration on ISO/IEC 23090-13 VDI

**Status:** Approved

**Date of document:** 2025-05-02

**Source:** ISO/IEC JTC 1/SC 29/WG 03

**No. of pages:** 10 (with cover page)

**Email of Convenor:** young.L @ samsung . com

**Committee URL:** <https://isotc.iso.org/livelink/livelink/open/jtc1sc29wg3>

**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 03 MPEG SYSTEMS**

**ISO/IEC JTC 1/SC 29/WG 03 N1509**

**April 2025, Online**

|  |  |
| --- | --- |
| **Title** | **Technologies under consideration on ISO/IEC 23090-13 VDI** |
| **Source** | **WG 03, MPEG Systems** |
| **Status** | **Approved** |
| **Serial Number** | **25045** |

# Introduction

This document collects technologies being under study for consideration in the development of the standard ISO/IEC 23090-13 Video Decoder Interface

# Table of contents

[Introduction 1](#_Toc197107692)

[Table of contents 1](#_Toc197107693)

[1. Mapping on W3C WebCodecs 1](#_Toc197107694)

[2. AVC instantiation binding 5](#_Toc197107695)

# Mapping on W3C WebCodecs

W3C has developed the WebCodec API as a specification to offer a codec-agnostic encoding/decoding interface in the browser [2]. Since the VDI group is currently working towards the second edition of the standard [1], we believe it is relevant for the group to study potential mapping of the WebCodecs API to VDI functions.

## Background on WebCodecs

### From MDN [3]:

The **WebCodecs API** gives web developers low-level access to the individual frames of a video stream and chunks of audio. It is useful for web applications that require full control over the way media is processed. For example, video or audio editors, and video conferencing.

#### [Concepts and Usage](https://developer.mozilla.org/en-US/docs/Web/API/WebCodecs_API#concepts_and_usage)

Many Web APIs use media codecs internally. For example, the [Web Audio API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API), and the [WebRTC API](https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API). However these APIs do not allow developers to work with individual frames of a video stream and unmixed chunks of encoded audio or video.

Web developers have typically used WebAssembly in order to get around this limitation, and to work with media codecs in the browser. However, this requires additional bandwidth to download codecs that already exist in the browser, reducing performance and power efficiency, and adding additional development overhead.

The WebCodecs API provides access to codecs that are already in the browser. It gives access to raw video frames, chunks of audio data, image decoders, audio and video encoders and decoders.

[**Processing Model**](https://developer.mozilla.org/en-US/docs/Web/API/WebCodecs_API#processing_model)

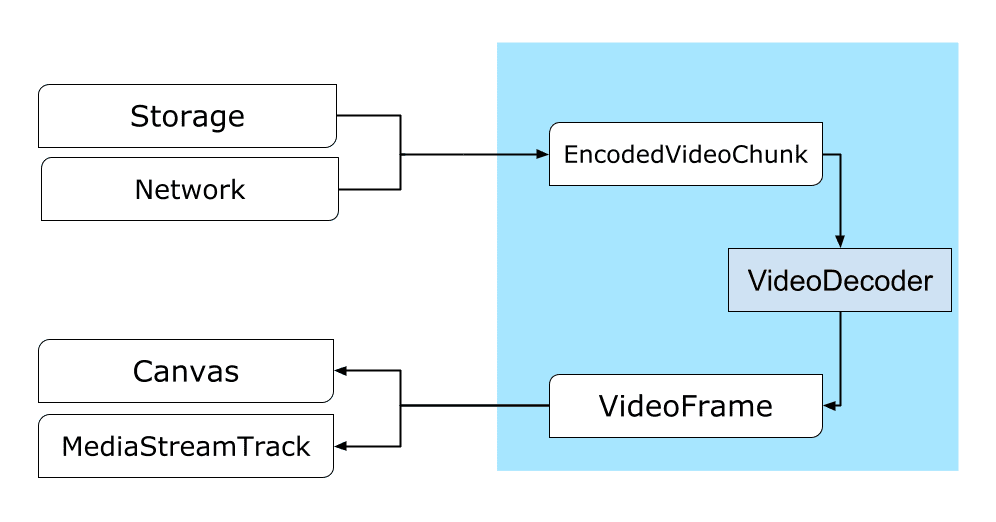
The WebCodecs API uses an asynchronous [processing model](https://w3c.github.io/webcodecs/#codec-processing-model-section). Each instance of an encoder or decoder maintains an internal, independent processing queue. When queueing a substantial amount of work, it's important to keep this model in mind.

Methods named configure(), encode(), decode(), and flush() operate asynchronously by appending control messages to the end the queue, while methods named reset() and close() synchronously abort all pending work and purge the processing queue. After reset(), more work may be queued following a call to configure(), but close() is a permanent operation.

Methods named flush() can be used to wait for the completion of all work that was pending at the time flush() was called. However, it should generally only be called once all desired work is queued. It is not intended to force progress at regular intervals. Calling it unnecessarily will affect encoder quality and cause decoders to require the next input to be a key frame.

### Example implementation

Following is an example implementation of a decoding chain using the VideoDecoder of WebCodecs [4]



*Figure 1 The path from the network or storage to a Canvas or an ImageBitmap.*

Setting up a VideoDecoder with two functions passed when the decoder is created, and codec parameters given to configure().

The set of codec parameters varies from codec to codec. For example H.264 codec might need a [binary blob](https://w3c.github.io/webcodecs/#dom-videodecoderconfig-description) of AVCC, unless it's encoded in so called Annex B format (encoderConfig.avc = { format: "annexb" }).

const init = {

output: handleFrame,

error: (e) => {

console.log(e.message);

},

};

const config = {

codec: "vp8",

codedWidth: 640,

codedHeight: 480,

};

const { supported } = await VideoDecoder.isConfigSupported(config);

if (supported) {

const decoder = new VideoDecoder(init);

decoder.configure(config);

} else {

// Try another config.

}

Once the decoder is initialized, you can start feeding it with EncodedVideoChunk objects. To create a chunk, you'll need:

* A [BufferSource](https://developer.mozilla.org/docs/Web/API/BufferSource) of encoded video data
* the chunk's start timestamp in microseconds (media time of the first encoded frame in the chunk)
* the chunk's type, one of:
  + key if the chunk can be decoded independently from previous chunks
  + delta if the chunk can only be decoded after one or more previous chunks have been decoded

Also any chunks emitted by the encoder are ready for the decoder as is. All of the things said above about error reporting and the asynchronous nature of encoder's methods are equally true for decoders as well.

const responses = await downloadVideoChunksFromServer(timestamp);

for (let i = 0; i < responses.length; i++) {

const chunk = new EncodedVideoChunk({

timestamp: responses[i].timestamp,

type: responses[i].key ? "key" : "delta",

data: new Uint8Array(responses[i].body),

});

decoder.decode(chunk);

}

await decoder.flush();

## Proposed WebCodecs Mapping

### Mapping of VDI Functions

Table X.1 shows the possible mapping of the VDI functions onto the WebCodecs API.

**Table X.1 - Possible mapping of VDI onto WebCodecs**

|  |  |
| --- | --- |
| **VDI Functionality** | **WebCodecs Mapping** |
| queryCurrentAggregate Capabilities() | VideoDecoder.queryCurrentAggregate Capabilities() a |
| getInstance() with grouping | VideoDecoder.group(group\_id) b |
| setConfig() CONFIG\_OUTPUT\_BUFFER | VideoDecoder.configure(*VideoDecoderConfig*  config) c |
| getParameter() and setParameter() | VideoDecoder, getParameter() and setParameter() d |
| a   A new method of the VideoDecoder object is used to query the current decode capabilities.  b   There is not native grouping for VideoDecoder objects in the WebCodecs. This interface assigns a group to the decoder  c  The configure method of the VideoDecoder object is extended to match the setConfig method  d  New methods of the VideoDecoder object. | |

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 chunk type (key if the chunk can be decoded independently from other chunks – e.g. containing only I frames, or delta if there are decoding dependencies to previous chunks). Typically the chunk data field is of type Uint8Array.

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

[Editor’s Note: the WebCodecs API representation in IDL needs to be done to finalize the mapping]

[Editor’s Note: The mechanisms of WebCodecs handling of metadata streams are to be further studied]

[Editor’s Note: WebCodecs provides formatting capabilities that might be relevant to VDI functions and are to be further studied]

## References

1. ISO/IEC JTC 1/SC 29/WG 03 N1311, “WD of ISO/IEC 23090-13 2nd edition Video Decoding Interface for Immersive Media”, July 2024, Online
2. WebCodecs, W3C Working Draft, 8 October 2024, <https://www.w3.org/TR/webcodecs/>
3. WebCodecs API, MDN, <https://developer.mozilla.org/en-US/docs/Web/API/WebCodecs_API>
4. WebCodecs best practices, Chrome for Developers, <https://developer.chrome.com/docs/web-platform/best-practices/webcodecs#decoding>

# AVC instantiation binding

## Background on relevant AVC features for possible mappings

### General

### Mapping the object concept on AVC

#### General

Compared with VVC, HEVC and EVC, the AVC standard does not support the concept of tiles as a way to partition a coded picture in sub regions.

However, during the development of the concept of motion constrained tile set in HEVC, JVET experts have also designed a functionally equivalent concept of the AVC standard which is the motion constrained slice group.

For convenience, we have extracted the relevant part from the AVC specification in the next clause.

#### Coded picture

**3.28 coded picture**: A *coded representation* of a *picture*. A coded picture may be either a *coded field* or a *coded frame*. Coded picture is a collective term referring to a *primary coded picture* or a *redundant coded picture*, but not to both together.

#### Slice group

**3.156 slice group**: A subset of the *macroblocks* or *macroblock pairs* of a *picture*. The division of the *picture* into slice groups is a *partitioning* of the *picture.* The *partitioning* is specified by the *macroblock to slice group map*.

**slice\_group\_id[** i **]** identifies a slice group of the i-th slice group map unit in raster scan order. The length of the slice\_group\_id[ i ] syntax element is Ceil( Log2( num\_slice\_groups\_minus1 + 1 ) ) bits. The value of slice\_group\_id[ i ] shall be in the range of 0 to num\_slice\_groups\_minus1, inclusive.

## Description of AVC mapping using picture

### Overview

The proposed AVC binding rely on mapping a video object to a coded picture. As a result, an AVC media stream with multiple video objects is made of the aggregation of multiple AVC elementary stream. This way, a video decoding engine processing such AVC media stream will merely split this input media stream into multiple AVC elementary stream and manage multiple AVC decoding instances in parallel.

The overview of the AVC binding is provided in Table 1.

*Table 1 - Overview of AVC binding*

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

### Video object identifier

To identify each video object from each other, i.e. each elementary stream, the AVC binding as the other binding requires the definition of an identifier associated with a given AVC elementary stream. To this end, the proposed AVC binding defines the video object identifier as the i-th AVC bitstream that is interleaved in the media stream. By convention, we chose to start the number at 0.

### AVC media stream structure

As described above, the proposed binding relies on the codec picture for the video object sample. Consequently, one video object corresponds to one AVC elementary stream. Creating an AVC media stream corresponds thus to combine multiple AVC elementary streams into one. From the media stream structure’s point of view, this is similar to creating a “multi-layer” AVC bitstream while no AVC scalable extension is used.

To this end, we need a way to combine those multiple AVC elementary streams into a single data stream and to allow the reverse process such that each AVC elementary stream can be decoded separately.

In a first scenario, the number of NAL units per access unit in the input AVC elementary streams is the same. It is possible to interleave all the bitstreams into a single sequence of NAL units as illustrated in Figure 2.

A group of white rectangular boxes with black text

Description automatically generated

*Figure 2 - Example of AVC media stream*

In a second scenario, 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 3.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 3 - Example of AVC media stream with different number of NAL units per access unit*

In a third scenario, the number of NAL units per access unit in both input elementary streams vary from one access unit to another as illustrated in Figure 4.

A screenshot of a computer

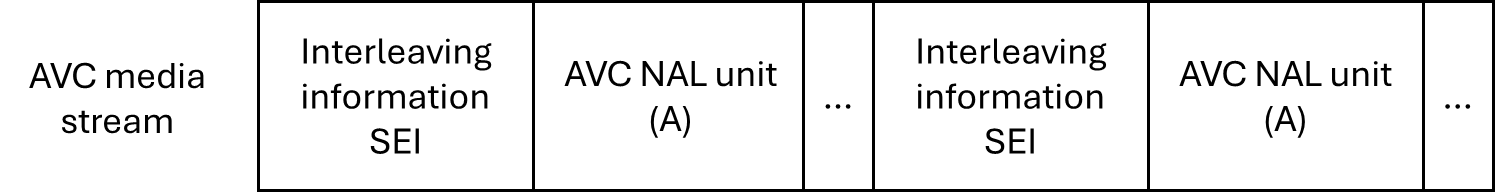
AI-generated content may be incorrect.

*Figure 4 - Example of AVC media stream with varying number of NAL units per access unit*

However, there is a minimum of information needed to reverse the process that is:

1. The number of interleaved bitstreams.
2. The interleaving pattern of interleaved NAL units.

For this reason, the proposed binding defines a new SEI message that contains those two parameters, and this SEI message would be present at the beginning of the media stream and at any start of a access unit in the media stream which does not follow the pattern signalled in the previous media stream access unit. Figure 3 illustrates the AVC media stream and the presence of this SEI message.



*Figure 3 - AVC media stream with interleaving information*

|  |  |
| --- | --- |
| 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** | ue(v) |
| } |  |
| } |  |
| } |  |

**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** specifies the number of bitstream that have been interleaved**.**

**ii\_interleaving\_pattern** indicates the interleaving pattern that was used to interleave the different bitstreams. The value 0 indicates that the NAL units of each bitstream are arranged in in the same order in a repetitive pattern, i.e. a,b,c,a,b,c etc… The 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. The value 2 indicates that the NAL units of each bitstream are arrange in a rotating pattern, i.e. a,b,c,b,c,a,c,a,b,a,b,c, etc. The 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 signalled 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 – 1 included.

### Spatial position of video objects

To enable the appending and stacking VDI operations, it is also required to support the carriage of the spatial relationship of the video objects.

Assuming the creation of a VDI SEI message code point in AVC (as proposed above), it would be possible to reuse the existing Independent layer info SEI message and to generalise it to independent decoded video sequence and not just independent layers.

#### This point is left for further study.