ISO/IEC JTC 1/SC 29/WG 03 N1397

**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:** 2024-12-02

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

**No. of pages:** 8 (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 N1397**

**December 2024, Kemer, TR**

|  |  |
| --- | --- |
| **Title** | **Technologies under consideration on ISO/IEC 23090-13 VDI** |
| **Source** | **WG 03, MPEG Systems** |
| **Status** | **Approved** |
| **Serial Number** | **24541** |

# 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](#_Toc184050183)

[Table of contents 1](#_Toc184050184)

[1. Mapping on W3C WebCodecs 1](#_Toc184050185)

[2. AVC instantiation binding 5](#_Toc184050186)

# 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)

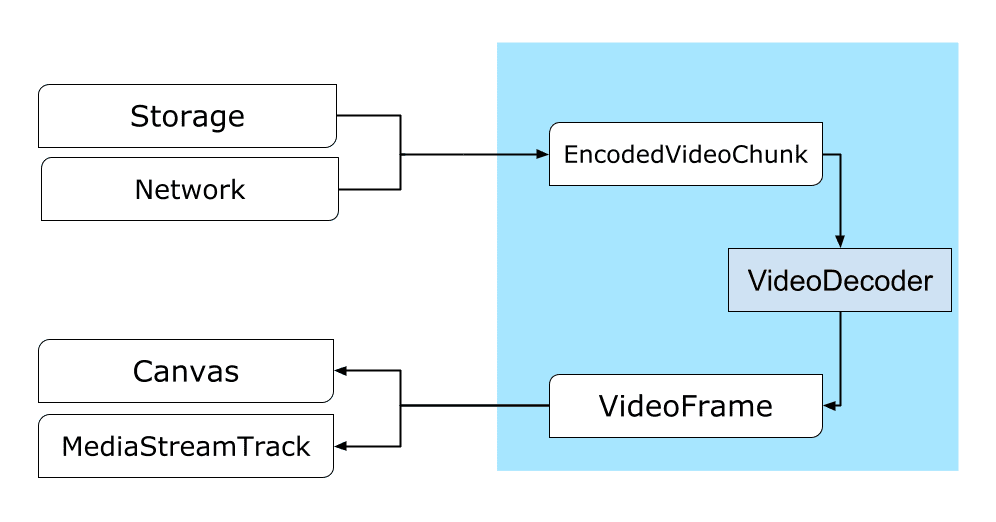
The WebCodecs API uses an asynchronous [processing model](https://w3c.github.io/webcodecs/#codec-processing-model-section). Each instance of an encoder or decoder maintains an internal, independent processing queue. When queueing a substantial amount of work, it's important to keep this model in mind.

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

0shows the possible mapping of the VDI functions onto the WebCodecs API.

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

|  |  |
| --- | --- |
| **VDI Functionality** | **MSE Mapping** |
| queryCurrentAggregate Capabilities() | VideoDecoder.queryCurrentAggregate Capabilities() a |
| getInstance() with grouping | TODO 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. An interface is to be defined in VDI  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.

[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.

### Possible approaches for AVC binding

#### General

At high-level, the future AVC binding in VDI will have to map concepts on VDI onto AVC features.

The VDI ElementaryStream and AccessUnit are straightforward to map on respectively the bitstream and access unit as defined in AVC.

The core of the mapping remains to be studied for the VideoObjectIdentifier and VideoObjectSample.

We envision three possible mappings. Those three mapping are inspired by current bindings as follows:

* Mapping #1: Using slice group (similar to HEVC and EVC binding)
* Mapping #2: Using slice (similar to HEVC and EVC binding)
* Mapping #3: Using picture (similar to VVC binding)

The following tables summarize the possible high-level binding approaches for AVC.

*Table 1 - Possible high-level AVC binding approaches*

|  |  |  |  |
| --- | --- | --- | --- |
| Concept | Mapping #1 | Mapping #2 | Mapping #3 |
| ElementaryStream | bitstream | bitstream | bitstream |
| AccessUnit | access unit | access unit | access unit |
| VideoObjectIdentifier | slice group id | Address of first macroblock in the slice | “bitstream id” |
| VideoObjectSample | slice group | slice | coded picture |

#### Discussion

On mapping #1, the binding would use the concept of slice group. Slice groups are not available in all AVC profiles and thus would limit the applicability of the binding. The advantage of the slice group concept is that it comes with identifier and this identifier can be signalled in the Picture Parameter Set (PPS).

On mapping #2, the binding would leverage directly the slice concept and therefore would be more generic to any AVC profile. However, rewriting slices in the context of AVC may be cumbersome in the absence of tools like tiles in HEVC.

On mapping #3, the binding would be on a picture level, similar to the current VVC binding in VDI. However, this would not leverage a layer concept. Therefore, there should be way to identify a bitstream which would map onto the video object identifier.

At this point, further study is needed in order to understand the best approach. For instance, it would be desirable to study the feasibility of each mapping when considering the implementation of the VDI operations of filtering, inserting, appending and stacking given the constraints of each leverage coding tools that are slice group, slice and coded picture.