An HTML version is now published here.

This doc will no longer be updated. Please use the above link instead and file feedback via github

Abstract

This specification defines interfaces for encoding and decoding audio and video. It also includes an interface for retrieving raw video frames from MediaStreamStracks.

Definitions

Codec refers generically to the types: AudioDecoder, AudioEncoder, VideoDecoder, and VideoEncoder.

Key frame refers to an encoded frame that does not depend on any other frames for decoding.

// More to come...

Processing Model

New codec tasks may be scheduled while previous tasks are still pending. For example, web authors may call decode() without waiting for the previous decode() to generate an output. This is facilitated by the following mechanisms.

Each codec has a single **control message queue** that is a list of **control messages**.

Queuing a control message means adding the message to the end of a codec's control message queue. Invoking codec methods will often queue a control message to schedule work.

Running a control message means executing a sequence of steps defined by the method that enqueued the message.

Control messages in a control message queue are ordered by time of insertion. The oldest message is therefore the one at the front of the control message queue.

The **control message processing loop,** described below, will run control messages in the control message queue.

Running the control message processing loop means executing these steps.

1. While the control message queue is not empty
    a. Let *front message* be the next oldest control message
    b. If *front message* cannot be executed now, return.

       The User Agent must decide when further processing is blocked because of ongoing work as an implementation detail (e.g. the underlying decoder cannot accept more requests yet). The UA must restart the processing loop when the blockage is resolved.

       NOTE: a blocked processing loop is visible to authors via the decodeQueueSize and encodeQueueSize attributes.

    c. Dequeue *front message* from the control message queue.

    d. Run the *front message* control message steps.

AudioDecoder Interface

```
[Exposed=(Window,Worker)]
interface AudioDecoder {
  constructor(AudioDecoderInit init);
  readonly attribute CodecState state;
  readonly attribute long decodeQueueSize;
  void configure(AudioDecoderConfig config);
  void decode(EncodedAudioChunk chunk);
  Promise<void> flush();
  void reset();
  void close();
};

dictionary AudioDecoderInit {
  required AudioFrameOutputCallback output;
  required WebCodecsErrorCallback error;
};

callback AudioFrameOutputCallback = void(AudioFrame output);
callback WebCodecsErrorCallback = void(DOMException error);
```

Internal Slots

[[codec implementation]]
Underlying decoder implementation provided by the User Agent.
[[output callback]]
Callback given at construction for decoded outputs.
[[error callback]]
Callback given at construction for decode errors.

Constructor

1. Let d be a new AudioDecoder object.
2. Assign init.output to the [[output callback]] internal slot.
3. Assign init.error to the [[error callback]] internal slot.
4. Return d.

Attributes

state, of type CodecState, readonly
    Describes the current state of the codec.

decodeQueueSize, of type long, readonly

The number of pending decode requests. This does not include requests that have been sent to the underlying codec.

Methods

configure(AudioDecoderConfig config)

Enqueues a control message to configure the audio decoder for decoding chunks as described by config.

When invoked, run these steps:

1. If config is not a valid AudioDecoderConfig, throw a TypeError.
2. Run the Configure Decoder algorithm with *config*.

decode(EncodedAudioChunk chunk)

Enqueues a control message to decode the given chunk.

When invoked, run these steps:

1. If chunk is not a valid EncodedAudioChunk, throw a TypeError.
2. Let *output algorithm* be the AudioFrame Output algorithm.
3. Run the Decode Chunk algorithm with *chunk* and *output algorithm*.

flush()

Complete all control messages in the control queue and emit all outputs.

When invoked, run these steps:

1. Let *output algorithm* be the AudioFrame Output algorithm.
2. Run the Flush algorithm with *output algorithm.*

reset()

Immediately resets all state including configuration, control messages in the control queue, and all pending callbacks.

When invoked, run the Reset algorithm.

close()

Immediately aborts all pending work and releases system resources. Close is permanent.

When invoked, run the Close algorithm.

VideoDecoder Interface

```
[Exposed=(Window,Worker)]
interface VideoDecoder {
  constructor(VideoDecoderInit init);
  readonly attribute CodecState state;
  readonly attribute long decodeQueueSize;
  void configure(VideoDecoderConfig config);
  void decode(EncodedVideoChunk chunk);
  Promise<void> flush();
  void reset();
  void close();
};

dictionary VideoDecoderInit {
  required VideoFrameOutputCallback output;
  required WebCodecsErrorCallback error;
};

callback VideoFrameOutputCallback = void(VideoFrame output);
callback WebCodecsErrorCallback = void(DOMException error);
```

Internal Slots

[[codec implementation]]
Underlying decoder implementation provided by the User Agent.
[[output callback]]
Callback given at construction for decoded outputs.
[[error callback]]
Callback given at construction for decode errors.

Constructor

1. Let d be a new VideoDecoder object.
2. Assign init.output to the [[output callback]] internal slot.
3. Assign init.error to the [[error callback]] internal slot.
4. Return d.

Attributes

state, of type CodecState, readonly
Describes the current state of the codec.

decodeQueueSize, of type long, readonly
The number of pending decode requests. This does not include requests that

have been sent to the underlying codec.

Methods

configure(VideoDecoderConfig config)

Enqueues a control message to configure the video decoder for decoding chunks as described by config.

When invoked, run these steps:

1. If config is not a valid VideoDecoderConfig, throw a TypeError.
2. Run the Configure Decoder algorithm with *config*.

decode(EncodedVideoChunk chunk)

Enqueues a control message to decode the given chunk.

When invoked, run these steps:

1. If chunk is not a valid EncodedVideoChunk, throw a TypeError.
2. Let *output algorithm* be the VideoFrame Output algorithm.
3. Run the Decode Chunk algorithm with *chunk* and *output algorithm*.

flush()

Complete all control messages in the control queue and emit all outputs.

When invoked, run these steps:

1. Let *output algorithm* be the VideoFrame Output algorithm.
2. Run the Flush algorithm with *output algorithm.*

reset()

Immediately resets all state including configuration, control messages in the control queue, and all pending callbacks.

When invoked, run the Reset algorithm.

close()

Immediately aborts all pending work and releases system resources. Close is permanent.

When invoked, run the Close algorithm.

AudioEncoder Interface

```
[Exposed=(Window,Worker)]
interface AudioEncoder {
  constructor(AudioEncoderInit init);
  readonly attribute CodecState state;
  readonly attribute long encodeQueueSize;
  void configure(AudioEncoderConfig config);
  void encode(AudioFrame frame);
  Promise<void> flush();
  void reset();
  void close();
};

dictionary AudioEncoderInit {
  required EncodedAudioChunkOutputCallback output;
  required WebCodecsErrorCallback error;
};

callback EncodedAudioChunkOutputCallback = void(EncodedAudioChunk output);
callback WebCodecsErrorCallback = void(DOMException error);
```

Internal Slots

[[codec implementation]]
Underlying encoder implementation provided by the User Agent.
[[output callback]]
Callback given at construction for encoded outputs.
[[error callback]]
Callback given at construction for encode errors.

Constructor

1. Let e be a new AudioEncoder object.
2. Assign init.output to the [[output callback]] internal slot.
3. Assign init.error to the [[error callback]] internal slot.
4. Assign "unconfigured" to e.state.
5. Return e.

Attributes

state, of type CodecState, readonly
Describes the current state of the codec.

encodeQueueSize, of type long, readonly
The number of pending encode requests. This does not include requests that have been sent to the underlying codec.

Methods

configure(AudioEncoderConfig config)
Enqueues a control message to configure the audio encoder for encoding chunks as described by config.

When invoked, run these steps:

1. If config is not a valid AudioEncoderConfig, throw a TypeError.
2. Run the Configure Encoder algorithm with *config*.

encode(AudioFrame frame)

Enqueues a control message to encode the given frame.

When invoked, run these steps:

1. If frame is not a valid AudioFrame, throw a TypeError.
2. Let *output algorithm* be the EncodedAudioChunk Output algorithm.
3. Run the Encode Frame algorithm with *frame* and *output algorithm*.

flush()

Complete all control messages in the control queue and emit all outputs.

When invoked, run these steps:

1. Let *output algorithm* be the EncodedAudioChunk Output algorithm.
2. Run the Flush algorithm with *output algorithm.*

reset()

Immediately resets all state including configuration, control messages in the control queue, and all pending callbacks.

When invoked, run the Reset algorithm.

close()

Immediately aborts all pending work and releases system resources. Close is permanent.

When invoked, run the Close algorithm.

VideoEncoder Interface

```
[Exposed=(Window,Worker)]
interface VideoEncoder {
  constructor(VideoEncoderInit init);
  readonly attribute CodecState state;
  readonly attribute long encodeQueueSize;
  void configure(VideoEncoderConfig config);
  void encode(VideoFrame frame, optional VideoEncoderEncodeOptions options = {});
  Promise<void> flush();
  void reset();
  void close();
};
```

```
dictionary VideoEncoderInit {
  required EncodedVideoChunkOutputCallback output;
  required WebCodecsErrorCallback error;
};

callback EncodedVideoChunkOutputCallback = void(EncodedVideoChunk output);
callback WebCodecsErrorCallback = void(DOMException error);
```

Internal Slots

[[codec implementation]]
Underlying encoder implementation provided by the User Agent.
[[output callback]]
Callback given at construction for encoded outputs.
[[error callback]]
Callback given at construction for encode errors.

Constructor

1. Let e be a new VideoEncoder object.
2. Assign init.output to the [[output callback]] internal slot.
3. Assign init.error to the [[error callback]] internal slot.
4. Assign "unconfigured" to e.state.
5. Return e.

Attributes

state, of type CodecState, readonly
Describes the current state of the codec.

encodeQueueSize, of type long, readonly
The number of pending encode requests. This does not include requests that have been sent to the underlying codec.

Methods

configure(VideoEncoderConfig config)
Enqueues a control message to configure the video encoder for encoding chunks as described by config.

When invoked, run these steps:
1. If config is not a valid VideoEncoderConfig, throw a TypeError.
2. Run the Configure Encoder algorithm with *config*.

encode(VideoFrame frame, VideoEncoderEncodeOptions options)
Enqueues a control message to encode the given frame.

NOTE: This method will destroy the VideoFrame. Authors who wish to retain a copy, should call frame.clone() prior to calling encode().

When invoked, run these steps:

1. If frame is not a valid VideoFrame, throw a TypeError.
2. Let *output algorithm* be the EncodedVideoChunk Output algorithm.
3. Run the Encode Frame algorithm with *frame, options,* and *output algorithm*.

flush()

Complete all control messages in the control queue and emit all outputs.

When invoked, run these steps:

1. Let *output algorithm* be the EncodedVideoFrame Output algorithm.
2. Run the Flush algorithm with *output algorithm.*

reset()

Immediately resets all state including configuration, control messages in the control queue, and all pending callbacks.

When invoked, run the Reset algorithm.

close()

Immediately aborts all pending work and releases system resources. Close is permanent.

When invoked, run the Close algorithm.

Decoder and Encoder Algorithms

Configure Decoder (w/ config)

Run the following steps:

1. If state is “closed”, throw an InvalidStateError.
2. If the user agent cannot provide a codec implementation to support config, throw a NotSupportedError.
3. Set state to configured.
4. Queue a control message to configure the decoder using config.
5. Run the control message processing loop.

Running a control message to configure the decoder means running these steps:

1. Assign [[codec implementation]] with an implementation supporting config.

Decode Chunk (w/ chunk and output algorithm)

Run these steps:

1. If state is not configured, throw an InvalidStateError.
2. Increment decodeQueueSize.

3. Queue a control message to decode the chunk.
4. Run the control message processing loop.

Running a control message to decode the chunk means running these steps:

1. Decrement decodeQueueSize
2. Let *codec implementation queue* be the result of starting a new parallel queue.
3. Enqueue the following steps to *codec implementation queue*
    a. Attempt to use [[codec implementation]] to decode the chunk.
    b. If decoding results in an error, queue a task on the media element task source to run the codec error algorithm.
    c. Otherwise, for each output, queue a task on the media element task source to run the provided output algorithm.

Flush (w/ output algorithm)

Run these steps:

1. If state is not configured, return a Promise rejected with a newly created InvalidStateError.
2. Let *promise* be a new Promise.
3. Queue a control message to flush the codec with *promise.*
4. Return *promise*.

Running a control message takes to flush the codec means running these steps, taking the promise returned above:

1. Signal [[codec implementation]] to emit all pending outputs.
2. For each output, run the given *output algorithm*
3. Resolve *promise*.

Codec Error

Run these steps:

1. Cease processing of control queue
2. Run the Close algorithm with EncodingError

AudioFrame Output

Run these steps:

1. If state is not “configured”, abort the following steps.
2. Let buffer be an AudioBuffer containing the decoded audio data
3. Let frame be an AudioFrame containing buffer and a timestamp for the output.
4. Invoke [[output callback]] with frame.

VideoFrame Output

Run these steps:

1. If state is not “configured”, abort the following steps.
2. Let planes be a sequence of Planes containing the decoded video frame data.
3. Let pixelFormat be the PixelFormat of planes.
4. Let frameInit be an VideoFrameInit with the following keys:

    a. Let timestamp and duration be the presentation timestamp and duration from the EncodedVideoChunk associated with this output.
    b. Let codedWidth and codedHeight be the width and height of the decoded video frame in pixels, prior to any cropping or aspect ratio adjustments
    c. Let cropLeft, cropTop, cropWidth, and cropHeight be the crop region of the decoded video frame in pixels, prior to any aspect ratio adjustments
    d. Let displayWidth and displayHeight be the display size of the decoded video frame in pixels.
5. Let frame be a VideoFrame, constructed with pixelFormat, planes, and frameInit.
6. Invoke [[output callback]] with frame.

Reset
Run these steps:

1. If state is “closed”, throw an InvalidStateError.
2. Set state to “unconfigured”.
3. Signal [[codec implementation]] to cease producing output for the previous configuration.

   NOTE: Some tasks to emit outputs may already be queued in the event loop. These outputs will be dropped by the output algorithms, which abort if state is not “configured”.

4. For each control message in the control message queue:
    a. If a control message has an associated promise, reject the promise.
    b. Remove the message from the queue.

Close (with error)
Run these steps:

1. Run the reset algorithm.
2. Set state to “closed”.
3. Clear [[codec implementation]] and release associated system resources.
4. if error is set, invoke [[error callback]] with error.

Configure Encoder (w/ config)
Run the following steps:

1. If state is "closed", throw an InvalidStateError.
2. If the user agent cannot provide a codec implementation to support config, throw a NotSupportedError.
3. Set state to configured.
4. Queue a control message to configure the encoder using config.
5. Run the control message processing loop.

Running a control message to configure the encoder means running these steps:

1. Assign [[codec implementation]] with an implementation supporting config.

Encode Frame (w/ frame, options, and output algorithm)
Run these steps:

1. If state is not configured, throw an InvalidStateError.
2. If the value of frames [[detatched]] internal slot is true, throw a TypeError.
3. Let frameClone hold the result of running the Clone Frame algorithm with frame.
4. Destroy the original frame by invoking frame.destory().
5. Increment encodeQueueSize.
6. Queue a control message to encode frameClone.
7. Run the control message processing loop.

Running a control message to encode the frame means running these steps:

1. Decrement encodeQueueSize
2. Let *codec implementation queue* be the result of starting a new parallel queue.
3. Enqueue the following steps to *codec implementation queue*
    a. Attempt to use [[codec implementation]] and options to encode the frame.
    b. If encoding results in an error, queue a task on the media element task source to run the codec error algorithm.
    c. Otherwise, for each output, queue a task on the media element task source to run the provided output algorithm.

EncodedAudioChunk Output

Run these steps:

1. If state is not “configured”, abort the following steps.
2. Let chunkInit be an EncodedAudioChunkInit with the following keys:
    a. Let data contain the encoded audio data.
    b. Let type be the EnocdedAudioChunkType of the encoded audio data.
    c. Let timestamp be the timestamp from the associated input AudioFrame.
    d. Let duration be the duration from the associated input AudioFrame.
3. Let chunk be a new EncodedAudioChunk constructed chunkInit.
4. Invoke [[output callback]] with chunk.

EncodedVideoChunk Output

Run these steps:

1. If state is not “configured”, abort the following steps.
2. Let chunkInit be an EncodedVideoChunkInit with the following keys:
    a. Let data contain the encoded video data.
    b. Let type be the EncodedVideoChunkType of the encoded video data.
    c. Let timestamp be the timestamp from the associated input VideoFrame.
    d. Let duration be the duration from the associated input VideoFrame.
3. Let chunk be a new EncodedVideoChunk constructed with chunkInit.
4. Invoke [[output callback]] with chunk.

Configurations

Codec String

In other media specifications, codec strings historically accompanied mime types as the “codecs=” parameter (HTMLMediaElement, MSE, EME). In this specification, encoded media is not containerized; hence, only the value of the codecs parameter is accepted.

A **valid codec string** MUST meet the following conditions.

1. Is valid per the relevant codec specification (see examples below).

   NOTE: This needs more work. We might consider a registry of specs/strings.

2. It describes a single codec.

   NOTE: Not a comma separated list.

3. It is unambiguous about codec profile and level for codecs that define these concepts.

NOTE:
There is no unified specification for codec strings. Each codec has its own unique string format, specified by the authors of the codec. Relevant specifications include:
h264, aac - RFC6381
vp9 - https://www.webmproject.org/vp9/mp4/#codecs-parameter-string,
hevc - ISO IEC 14496-15 dated 2012 or newer in the Annex E.3
av1 - https://aomediacodec.github.io/av1-isobmff/#codecsparam,

NOTE:
Some valid examples include:
'vp8', 'vp09.00.10.08', 'avc1.4D401E', 'opus', 'mp4a.40.2', 'flac'

Invalid examples include:
'video/webm; codecs="vp8"' (invalid to supply full mimetype; valid as just 'vp8')
'codecs="opus"' (invalid to include codecs= prefix)
‘flac,vorbis’ (describes more than one codec)
‘vp9’ (ambiguous about profile and level)
'video/mp4' (describes a container, not a codec)

AudioDecoderConfig

```
dictionary AudioDecoderConfig {
  required DOMString codec;
  required unsigned long sampleRate;
  required unsigned long numberOfChannels;
  BufferSource description;
};
```

To check if an AudioDecoderConfig is a **valid audio decoder configuration**, run these steps:

1. If codec is not a valid codec string, return false.
2. Return true.

codec, of type DOMString

Contains a codec string describing the codec.

sampleRate, of type unsigned long

The number of frame samples per second.

numberOfChannels, of type unsigned long

The number of audio channels.

description, of type BufferSource

A sequence of codec specific bytes, commonly known as extradata.

NOTE: For example, the vorbis “code book”.

VideoDecoderConfig

```
dictionary VideoDecoderConfig {
  required DOMString codec;
  BufferSource description;
  required unsigned long codedWidth;
  required unsigned long codedHeight;
  unsigned long cropLeft;
  unsigned long cropTop;
  unsigned long cropWidth;
  unsigned long cropHeight;
  unsigned long displayWidth;
  unsigned long displayHeight;
};
```

To check if a VideoDecoderConfig is a **valid video decoder configuration**, run these steps:

1. If codec is not a valid codec string, return false.
2. If codedWidth = 0 or codedHeight = 0, return false.
3. If cropWidth = 0 or cropHeight = 0, return false.
4. If cropTop + cropHeight >= codedHeight, return false.
5. If cropLeft + cropWidth >= codedWidth, return false.
6. If displayWidth = 0 or displayHeight = 0, return false.
7. Return true.

codec, of type DOMString
Contains a codec string describing the codec.

description, of type BufferSource
A sequence of codec specific bytes, commonly known as extradata.

NOTE: For example, the VP9 vpcC bytes.

codedWidth, of type unsigned long
Width of the VideoFrame in pixels, prior to any cropping or aspect ratio adjustments.

codedHeight, of type unsigned long
Height of the VideoFrame in pixels, prior to any cropping or aspect ratio adjustments.

cropLeft, of type unsigned long
The number of pixels to remove from the left of the VideoFrame, prior to aspect ratio adjustments. Defaults to zero if not present.

cropTop, of type unsigned long
The number of pixels to remove from the top of the VideoFrame, prior to aspect ratio adjustments. Defaults to zero if not present.

cropWidth, of type unsigned long
The width of pixels to include in the crop, starting from cropLeft. Defaults to codedWidth if not present.

cropHeight, of type unsigned long
The height of pixels to include in the crop, starting from cropLeft. Defaults to codedHeight if not present.

displayWidth, of type double
Width of the VideoFrame when displayed. Defaults to cropWidth if not present.

displayHeight, of type double
Height of the VideoFrame when displayed. Defaults to cropHeight if not present.

AudioEncoderConfig

```
dictionary AudioEncoderConfig {
  required DOMString codec;
  unsigned long sampleRate;
  unsigned long numberOfChannels;
};
```

To check if an AudioEncoderConfig is a **valid audio encoder configuration**, run these steps:

1. If codec is not a valid codec string, return false.
2. Return true.

codec, of type DOMString

Contains a codec string describing the codec.

sampleRate, of type unsigned long

The number of frame samples per second.

numberOfChannels, of type unsigned long

The number of audio channels.

VideoEncoderConfig

```
dictionary VideoEncoderConfig {
  required DOMString codec;
  unsigned long long bitrate;
  required unsigned long cropWidth;
  required unsigned long cropHeight;
  unsigned long displayWidth;
  unsigned long displayHeight;
};
```

To check if a VideoDecoderConfig is a **valid video encoder configuration**, run these steps:

1. If codec is not a valid codec string, return false.
2. If width = 0 or height = 0, return false.
3. If displayWidth = 0 or displayHeight = 0, return false.
4. Return true.

cropWidth, of type unsigned long

The expected cropWidth of input VideoFrames to encode.

cropHeight, of type unsigned long

The expected cropHeight of input VideoFrames to encode.

displayWidth, of type double

Width of the VideoFrame when displayed. Defaults to cropWidth if not present.

displayHeight, of type double

Height of the VideoFrame when displayed. Defaults to cropHeight if not present.

VideoEncoderEncodeOptions

```
dictionary VideoEncoderEncodeOptions {
  boolean keyFrame;
};
```

keyFrame, of type boolean

Indicates whether the given frame MUST be encoded as a key frame.

CodecState

```
enum CodecState {
  "unconfigured",
  "configured",
  "closed"
};
```

- **unconfigured**: The codec is not configured for encoding or decoding.
- **configured:** A valid configuration has been provided. The codec is ready for encoding or decoding.
- **closed**: The codec is no longer usable and underlying system resources have been released.

Encoded Media Interfaces (Chunks)

These interfaces represent chunks of encoded media.

EncodedAudioChunk

```
interface EncodedAudioChunk {
  constructor(EncodedAudioChunkInit init);
  readonly attribute EncodedChunkType type;
  readonly attribute unsigned long long timestamp;  // microseconds
  readonly attribute ArrayBuffer data;
};
```

```
dictionary EncodedAudioChunkInit {
  required EncodedAudioChunkType type;
  required unsigned long long timestamp;
  required BufferSource data;
};

enum EncodedAudioChunkType {
    "key",
    "delta",
};
```

Constructor

1. Let chunk be a new EncodedAudioChunk object, initialized as follows
    a. Assign init.type to chunk.type.
    b. Assign init.timestamp to chunk.timestamp.
    c. Assign a copy of init.data to chunk.data.
2. Return chunk.

Attributes

type, of type EncodedChunkType, readonly
    Describes whether the chunk is a key frame or not.

timestamp, of type unsigned long long, readonly
    The presentation timestamp, given in microseconds.

data, of type ArrayBuffer
    A sequence of bytes containing encoded audio data.

EncodedVideoChunk

```
[Exposed=(Window,Worker)]
interface EncodedVideoChunk {
  constructor(EncodedVideoChunkInit init);
  readonly attribute EncodedVideoChunkType type;
  readonly attribute unsigned long long timestamp;  // microseconds
  readonly attribute unsigned long long? duration;  // microseconds
  readonly attribute ArrayBuffer data;
};

dictionary EncodedVideoChunkInit {
  required EncodedVideoChunkType type;
  required unsigned long long timestamp;
  unsigned long long duration;
  required BufferSource data;
};

enum EncodedVideoChunkType {
    "key",
    "delta",
};
```

Constructor

3. Let chunk be a new EncodedVideoChunk object, initialized as follows
    a. Assign init.type to chunk.type.
    b. Assign init.timestamp to chunk.timestamp.
    c. If duration is present in init, assign init.duration to chunk.duration. Otherwise, assign null to chunk.duration.
    d. Assign a copy of init.data to chunk.data.
4. Return chunk.

Attributes

type, of type EncodedChunkType, readonly
: Describes whether the chunk is a key frame or not.

timestamp, of type unsigned long long, readonly
: The presentation timestamp, given in microseconds.

duration, of type unsigned long long, readonly
: The presentation duration, given in microseconds.

data, of type ArrayBuffer
: A sequence of bytes containing encoded audio data.

Raw Media Interfaces (Frames)

These interfaces represent unencoded (raw) media.

AudioFrame

Internal Slots

[[detached]]
: Boolean indicating whether close() was invoked and underlying resources have been released.

```
[Exposed=(Window,Worker)]
interface AudioFrame {
  constructor(AudioFrameInit init);
  readonly attribute unsigned long long timestamp;
  readonly attribute AudioBuffer? buffer;
  void close();
};

dictionary AudioFrameInit {
  required unsigned long long timestamp;
  required AudioBuffer buffer;
};
```

Constructor

1. Let frame be a new AudioFrame object.
2. Assign init.timestamp to frame.timestamp.
3. Assign init.buffer to frame.buffer.
4. Assign false to the [[detached]] internal slot.
5. Return frame.

Attributes

timestamp, of type unsigned long long, readonly

The presentation timestamp, given in microseconds.

buffer, of type AudioBuffer, readonly

The buffer containing decoded audio data.

Methods

close()

Immediately frees system resources. When invoked, run these steps:

1. Release system resources for buffer and set its value to null.
2. Assign true to the [[detached]] internal slot.

NOTE: This section needs work. We should use the name and semantics of VideoFrame destory(). Similarly, we should add clone() to make a deep copy.

VideoFrame

```
[Exposed=(Window,Worker)]
interface VideoFrame {
  constructor(ImageBitmap imageBitmap, VideoFrameInit frameInit);
  constructor(PixelFormat pixelFormat, Sequence<Plane or PlaneInit> planes,
              VideoFrameInit frameInit);

  readonly attribute PixelFormat format;
  readonly attribute FrozenArray<Plane> planes;
  readonly attribute unsigned long codedWidth;
  readonly attribute unsigned long codedHeight;
  readonly attribute unsigned long cropLeft;
  readonly attribute unsigned long cropTop;
  readonly attribute unsigned long cropWidth;
  readonly attribute unsigned long cropHeight;
  readonly attribute unsigned long displayWidth;
  readonly attribute unsigned long displayHeight;
  readonly unsigned long long? duration;
  readonly unsigned long long? timestamp;
  // TODO: color space metadata.

  void destroy();
  VideoFrame clone();

  Promise<ImageBitmap> createImageBitmap(
    optional ImageBitmapOptions options = {});

};

dictionary VideoFrameInit {
  unsigned long codedWidth;
  unsigned long codedHeight;
  unsigned long cropLeft;
  unsigned long cropTop;
  unsigned long cropWidth;
  unsigned long cropHeight;
  unsigned long displayWidth;
  unsigned long displayHeight;
  unsigned long long duration;
  unsigned long long timestamp;
};
```

Internal Slots

[[detached]]

Boolean indicating whether destroy() was invoked and underlying resources have been released.

Constructors

NOTE: this section needs work. Current wording assumes a VideoFrame can always be easily represented using one of the known pixel formats. In practice, the underlying UA resources may be GPU backed or formatted in such a way that conversion to an allowed pixel format requires expensive copies and translation. When this occurs, we should allow *planes* to be null and format to be “opaque” to avoid early optimization. We should make conversion explicit and user controlled by offering a videoFrame.convertTo(format) that returns a Promise containing a new VideoFrame for which the copies/translations are performed.

VideoFrame(imageBitmap, frameInit)
Constructs a VideoFrame from an ImageBitmap.

1. If init is not a valid VideoFrameInit, throw a TypeError.
2. If the value of imageBitmap's [[Detached]] internal slot is set to true, then throw an "InvalidStateError" DOMException.
3. Let frame be a new VideoFrame.
4. Assign false to frame’s [[detached]] internal slot.
5. Use a copy of the pixel data in imageBitmap to initialize to following frame attributes:
    a. Initialize frame.pixelFormat be the underlying format of imageBitmap.
    b. Initialize frame.planes to describe the arrangement of memory of the copied pixel data.
    c. Assign regions of the copied pixel data to [[plane buffer]] internal slot of each plane as appropriate for the pixel format.
    d. Initialize frame.codedWidth and frame.codedHeight describe the width and height of the imageBitamp prior to any cropping or aspect ratio adjustments.
6. Use frameInit to initialize the remaining frame attributes:
    a. If frameInit.cropLeft is present, initialize it frame.cropLeft. Otherwise, default frame.cropLeft to zero.
    b. If frameInit.cropTop is present, initialize it to frame.cropTop. Otherwise, default frame.cropTop to zero.
    c. If frameInit.cropWidth is present, initialize it to frame.cropWidth. Otherwise, default frame.cropWidth to frame.codedWidth.
    d. If frameInit.cropHeight is present, initialize it to frame.cropHeight. Otherwise, default frame.cropHeight to frame.codedHeight.
    e. If frameInit.displayWidth is present, initialize it to frame.displayWidth. Otherwise, default frame.displayWidth to frame.codedWidth.
    f. If frameInit.displayHeight is present, initialize it to frame.displayHeight. Otherwise, default frame.displayHeight to frame.codedHeight.
    g. If frameInit.duration is present, initialize it to frame.duration. Otherwise, default frame.duration to null.
    h. If frameInit.timestamp is present, initialize it to frame.timestamp. Otherwise default frame.timestamp to null.
7. Return frame.

VideoFrame(pixelFormat, planes, frameInit)
Constructs an VideoFrame using the described format and pixel data.
1. If codedWidth, or codedHeight are not present in init, throw a TypeError.
2. If frameInit is not a valid VideoFrameInit, throw a TypeError.
3. If the number of planes is incompatible with the given pixelFormat, throw a TypeError.
4. Let frame be a new VideoFrame object.
5. Assign false to the frame’s [[detached]] internal slot.

6. Assign init.pixelFormat to frame.pixelFormat.
7. For each element p in planes:
    a. If p is a Plane, append a copy of p to frame.planes. Continue processing the next element.
    b. If p is a PlaneInit, append a new Plane q to frame.planes initialized as follows:
        i. Let sourceOffset be p.srcOffset if present, or otherwise default to zero.
        ii. Assign a copy of *length* bytes from p.src, starting at sourceOffset, to q's [[plane buffer]] internal slot.

            NOTE: the samples should be copied exactly, but the user agent may add row padding as needed to improve memory alignment.

        iii. Assign the width of each row in [[plane buffer]], including any padding, to q.stride.
        iv. Assign p.rows to q.rows.
        v. Assign the product of (q.rows * q.stride) to q.length
8. Assign frameInit.codedWidth to frame.codedWidth.
9. Assign frameInit.codedHeight to frame.codedHeight.
10. If frameInit.cropLeft is present, assign it frame.cropLeft. Otherwise, default frame.cropLeft to zero.
11. If frameInit.cropTop is present, assign it to frame.cropTop. Otherwise, default frame.cropTop to zero.
12. If frameInit.cropWidth is present, assign it to frame.cropWidth. Otherwise, default frame.cropWidth to frame.codedWidth.
13. If frameInit.cropHeight is present, assign it to frame.cropHeight. Otherwise, default frame.cropHeight to frame.codedHeight.
14. If frameInit.displayWidth is present, assign it to frame.displayWidth. Otherwise, default frame.displayWidth to frame.codedWidth.
15. If frameInit.displayHeight is present, assign it to frame.displayHeight. Otherwise, default frame.displayHeight to frame.codedHeight.
16. If frameInit.duration is present, assign it to frame.duration. Otherwise, default frame.duration to null.
17. If frameInit.timestamp is present, assign it to frame.timestamp. Otherwise default frame.timestamp to null.
18. Return frame.

Attributes

timestamp, of type unsigned long long, readonly

The presentation timestamp, given in microseconds. The timestamp is copied from the EncodedVideoChunk corresponding to this VideoFrame.

duration, of type unsigned long long, readonly

The presentation duration, given in microseconds. The duration is copied from the EncodedVideoChunk corresponding to this VideoFrame.

format, of type PixelFormat

Describes the arrangement of bytes in each plane as well as the number and order of the planes.

planes, of type FrozenArray<Plane>

Holds pixel data data, laid out as described by format and Plane attributes.

codedWidth, of type unsigned long

Width of the VideoFrame in pixels, prior to any cropping or aspect ratio adjustments

codedHeight, of type unsigned long

Height of the VideoFrame in pixels, prior to any cropping or aspect ratio adjustments

cropLeft, of type unsigned long

The number of pixels to remove from the left of the VideoFrame, prior to aspect ratio adjustments.

cropTop, of type unsigned long

The number of pixels to remove from the top of the VideoFrame, prior to aspect ratio adjustments.

cropWidth, of type unsigned long

The width of pixels to include in the crop, starting from cropLeft.

cropHeight, of type unsigned long

The height of pixels to include in the crop, starting from cropLeft.

displayWidth, of type double

Width of the VideoFrame when displayed.

displayHeight, of type double

Height of the VideoFrame when displayed.

Methods

destroy()

Immediately frees system resources. Destruction applies to all references, including references that are serialized and passed across Realms.

NOTE: Use clone() to create a deep copy. Cloned frames have their own lifetime and will not be affected by destroying the original frame.

When invoked, run these steps:

1. If [[detached]] is true, throw an InvalidStateError.
2. Remove all Planes from planes and release associated memory.
3. Assign true to the [[detached]] internal slot.

clone()

Creates a new VideoFrame with a separate lifetime containing a deep copy of this frame’s resources.

NOTE: VideoFrames may require a large amount of memory. Use clone() sparingly. Take care to manage frame lifetimes by calling destroy() when frames are no longer needed.

When invoked, run the following steps:

1. If the value of frame’s [[detatched]] slot is true, return a Promise rejected with a newly created InvalidStateError.
2. Let p be a new Promise.
3. In parallel, resolve p with the result of running the Clone Frame algorithm with frame.
4. Return p.

createImageBitmap(options)

Create an ImageBitmap from this VideoFrame.

When invoked, run these steps:

1. Let p be a new Promise.
2. If either options's resizeWidth or options's resizeHeight is present and is 0, then return p rejected with an "InvalidStateError" DOMException.
3. If the VideoFrame's [[detached]] internal slot is set to true, then return p rejected with an "InvalidStateError" DOMException.
4. Let imageBitmap be a new ImageBitmap object.
5. Set imageBitmap's bitmap data to a copy of the frame, at the frame's intrinsic width and intrinsic height (i.e., after any aspect-ratio correction has been applied), cropped to the source rectangle with formatting.
6. If the origin of image's image is not same origin with entry settings object's origin, then set the origin-clean flag of imageBitmap's bitmap to false.
7. Run this step in parallel:
    a. Resolve p with imageBitmap.

Plane

A Plane acts like a thin wrapper around an ArrayBuffer, but may actually be backed by a texture. Planes hide any padding before the first sample or after the last row.

A Plane is solely constructed by its VideoFrame. During construction, the User Agent may use knowledge of the frame’s PixelFormat to add padding to the Plane to improve memory alignment.

An Plane cannot be used after the VideoFrame is closed. A new VideoFrame can be assembled from existing Planes, and the new VideoFrame will remain valid when the original is closed. This makes it possible to efficiently add an alpha plane to an existing VideoFrame.

Internal Slots

[[parent frame]]

Refers to the {{VideoFrame}} that constructed and owns this plane.

[[plane buffer]]

Internal storage for the plane’s pixel data.

```
[Exposed=(Window,Worker)]
interface Plane {
  readonly attribute unsigned long stride;
  readonly attribute unsigned long rows;
  readonly attribute unsigned long length;

  void readInto(ArrayBufferView dst);
};

dictionary PlaneInit {
  required BufferSource src;
  required unsigned long stride;
  required unsigned long rows;
};
```

Attributes

stride, of type unsigned long, readonly
    The width of each row including any padding.

rows, of type unsigned long, readonly
    The number of rows.

length, of type unsigned long, readonly
    The total byte length of the plane (stride * rows).

Methods

readInto(dst)
    Copies the plane data into dst. When invoked, run these steps:

1. If the **[[parent frame]]** has been destroyed, throw an InvalidStateError.
2. If length is greater than dst.byteLength, throw a TypeError.
3. Copy the bytes of [[plane buffer]] into dst.

Pixel Format

Pixel formats describe the arrangement of bytes in each plane as well as the number and order of the planes.

```
enum PixelFormat {
  "I420",
  "ARGB",
  ...
};
```

- **I420:** Planar 4:2:0 YUV.
- **ARGB:** Interleaved ARGB.
- ....

Algorithms

Clone Frame (w/ frame)

1. Let cloneFrame be a new frame of the same type as *frame.*
2. Initialize each attribute and internal slot of clone with a copy of the value from the corresponding attribute of this frame.

   NOTE: User Agents are encouraged to avoid expensive copies of large objects (for instance, VideoFrame pixel data). Frame types are immutable, so the above step may be implemented using memory sharing techniques such as reference counting.

3. Return cloneFrame.

VideoTrackReader Interface

VideoTrackReader emits VideoFrames from a MediaStreamTrack. Authors may use this interface to manipulate, render, or encode streams from getUserMedia() and getDisplayMedia().

Internal Slots

[[track]]
The MediaStreamTrack provided at construction.

[[callback]]
The VideoFrameOutputCallback assigned by the last call to start().

```
[Exposed=Window]
interface VideoTrackReader {
  constructor(MediaStreamTrack track);
  readonly attribute VideoTrackReaderState state;
  attribute EventHandler onended;
  void start(VideoFrameOutputCallback callback);
  void stop();
};

enum VideoTrackReaderState {
    "started",
    "stopped",
    "ended"
};
```

Constructor

1. If track.kind is not "video", throw a TypeError.
2. If track.readyState is "ended", throw an InvalidStateError.
3. Let reader be a new VideoTrackReader object.
4. Assign track to the [[track]] internal slot.
5. Assign "stopped" to reader.readyState.
6. Return reader.

Attributes

onended, of type EventHandler

The event handler for the ended event.

Event Summary

ended

Dispatched when the [[track]]'s readyState becomes "ended", indicating no further frames will be output.

Methods

start(VideoFrameOutputCallback)

Starts calling the callback with VideoFrames from the MediaStreamTrack.

When invoked, run these steps:

1. If readyState is not "stopped", throw an InvalidStateError.
2. Assign "started" to readyState.
3. Assign callback to the [[callback]] internal slot.
4. In parallel, run the track monitor.

stop()

Stops calling the VideoFrameOutputCallback with VideoFrames from the MediaStreamTrack.

When invoked, run these steps:

1. If readyState is not "started", throw an InvalideStateError.
2. Cease running the track monitor.
3. Assign "stopped" to the readyState.
4. Assign null to the [[callback]] internal slot.

MediaStreamTrack Monitoring

The **track monitor** may be started and stopped by the user to control the calling of the VideoFrameOutputCallback.

Running the track monitor means monitoring [[track]] for the arrival of new picture data as well as changes to [[track]].readyState.

While [[track]].readyState is "live", for each new picture that arrives in [[track]], execute the following steps:

> NOTE: Video data in a MediaStreamTrack does not have a canonical binary form. The user agent should tokenize "pictures" by discrete times of capture. For example, if the source is a camera capturing 60 frames per second, the UA should construct 60 corresponding VideoFrame's each second.

1. Let planes be a sequence of Planes containing video data.
2. Let pixelFormat be the PixelFormat of planes.

   NOTE: The UA should avoid early optimizations to convert between PixelFormats. The "opaque" PixelFormat should be used when the video data does not match other defined PixelFormats. The UA must be prepared to convert "opaque" video data to a known format upon request. See videoFrame.convertTo(...)

3. Let frameInit be an VideoFrameInit with the following keys:
   a. Let timestamp and duration be the presentation timestamp and (optionally) presentation duration as determined by the [[track]] source.
   b. Let codedWidth and codedHeight be the width and height of the decoded video frame in pixels, prior to any cropping or aspect ratio adjustments
   c. Let cropLeft, cropTop, cropWidth, and cropHeight be the crop region of the decoded video frame in pixels, prior to any aspect ratio adjustments
   d. Let displayWidth and displayHeight be the display size of the decoded video frame in pixels.
4. Let frame be a VideoFrame, constructed with pixelFormat, planes, and frameInit.
5. Invoke [[callback]] with frame.

If [[track]].readyState becomes "ended", queue a task on the media element task source to run the following steps:

1. Set the VideoTrackReader state to "ended".
2. Queue a task on the media element task source to run a simple event named ended at the VideoTrackReader.