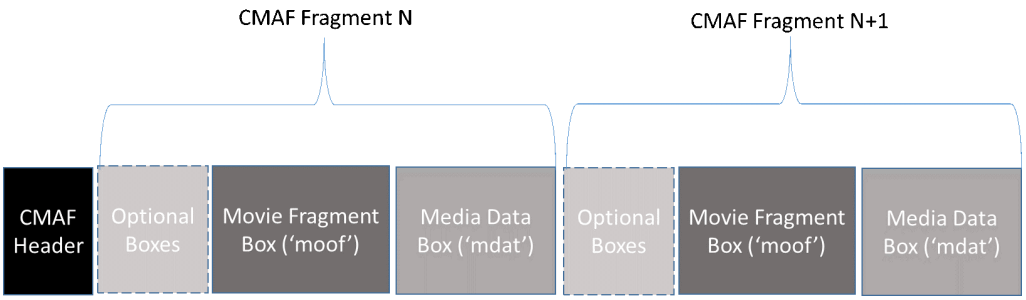
# DataCue MSE Integration for DASH Event Message Boxes

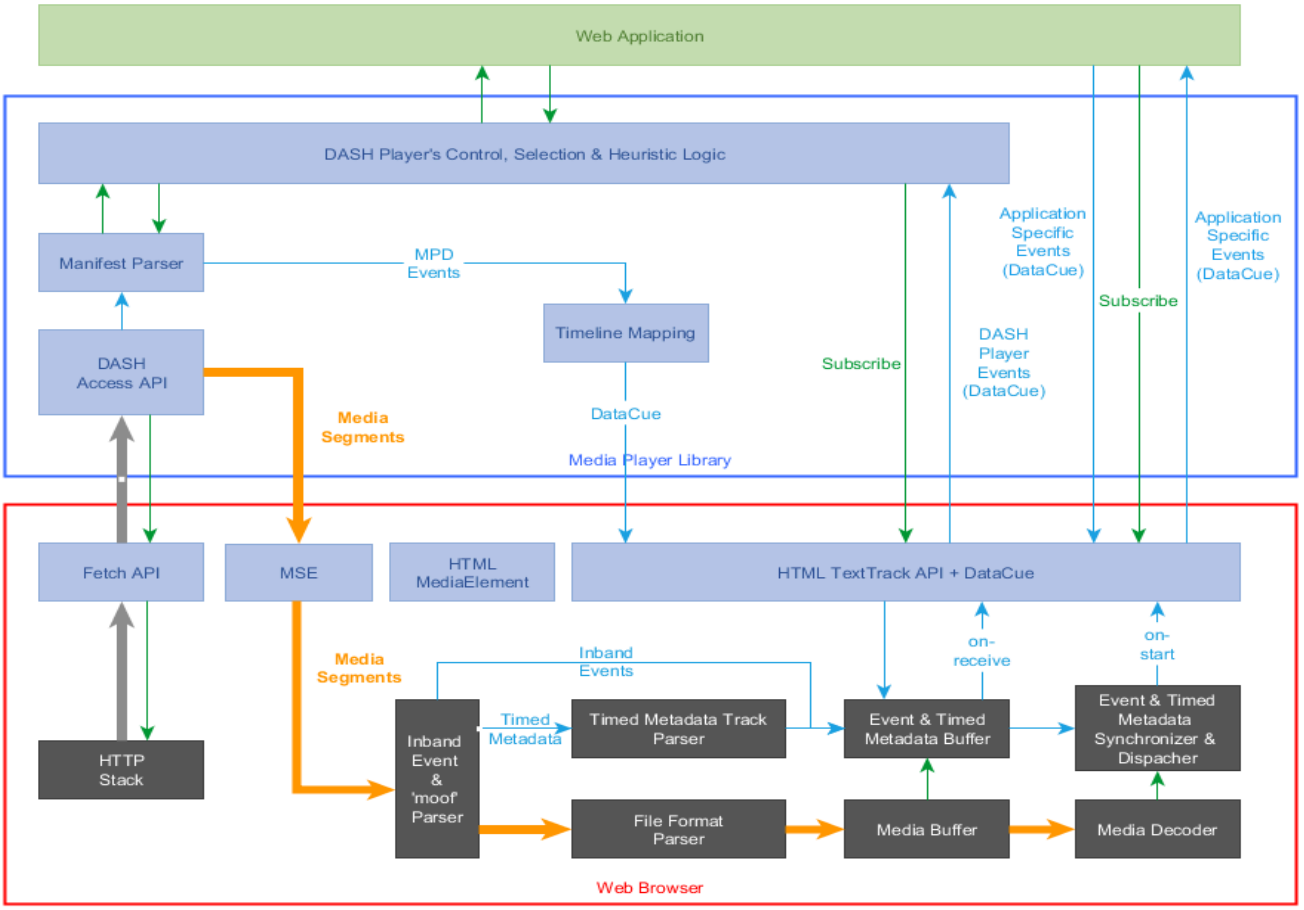
## ISO BMFF and CMAF

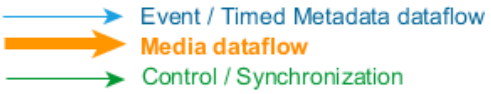



Optional Boxes includes:

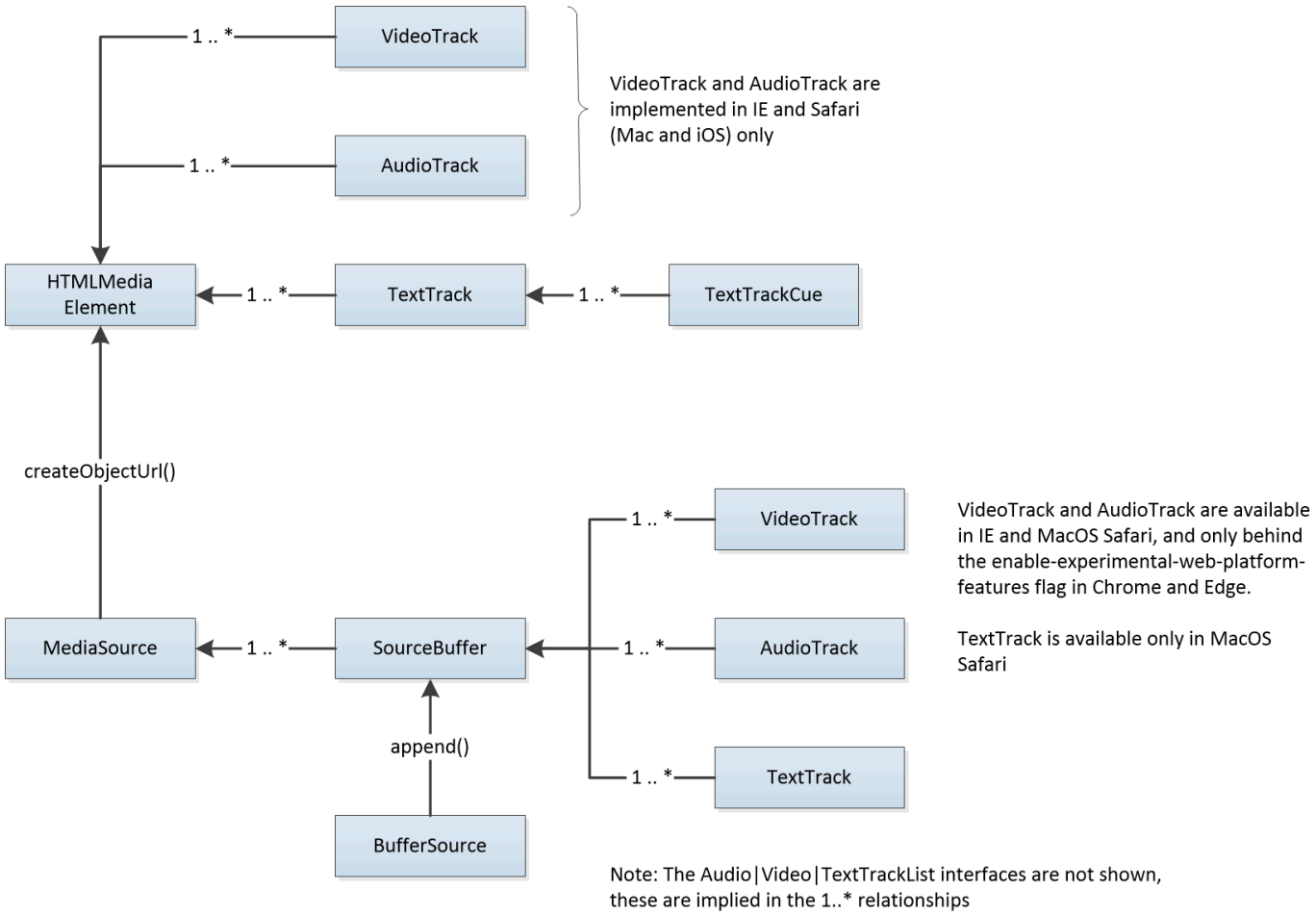
- Segment Type Box ('styp')
- Producer Reference Time Box ('prft')
- Event Message Box ('emsg')

## Architecture






## HTMLMediaElement and Media Source Extensions APIs




# Media Source Extensions

https://w3c.github.io/media-source/

## 1. Introduction

### 1.2 Definitions

**Active Track Buffers**

The track buffers that provide coded frames for the `enabled audioTracks`, the `selected videoTracks`, and the `"showing"` or `"hidden"` `textTracks`. All these tracks are associated with `SourceBuffer` objects in the `activeSourceBuffers` list.

**Append Window**

A presentation timestamp range used to filter out coded frames while appending. The append window represents a single continuous time range with a single start time and end time. Coded frames with presentation timestamp within this range are allowed to be appended to the `SourceBuffer` while coded frames outside this range are filtered out. The append window start and end times are controlled by the `appendWindowStart` and `appendWindowEnd` attributes respectively.

### Coded Frame

A unit of media data that has a presentation timestamp, a decode timestamp, and a coded frame duration.

### Coded Frame Duration

The duration of a coded frame. For video and text, the duration indicates how long the video frame or text *SHOULD* be displayed. For audio, the duration represents the sum of all the samples contained within the coded frame. For example, if an audio frame contained 441 samples @44100Hz the frame duration would be 10 milliseconds.

### Coded Frame End Timestamp

The sum of a coded frame presentation timestamp and its coded frame duration. It represents the presentation timestamp that immediately follows the coded frame.

### Coded Frame Group

A group of coded frames that are adjacent and have monotonically increasing decode timestamps without any gaps. Discontinuities detected by the coded frame processing algorithm and `abort()` calls trigger the start of a new coded frame group.

### Decode Timestamp

The decode timestamp indicates the latest time at which the frame needs to be decoded assuming instantaneous decoding and rendering of this and any dependant frames (this is equal to the presentation timestamp of the earliest frame, in presentation order, that is dependant on this frame). If frames can be decoded out of presentation order, then the decode timestamp *MUST* be present in or derivable from the byte stream. The user agent *MUST* run the append error algorithm if this is not the case. If frames cannot be decoded out of presentation order and a decode timestamp is not present in the byte stream, then the decode timestamp is equal to the presentation timestamp.

### Initialization Segment

A sequence of bytes that contain all of the initialization information required to decode a sequence of media segments. This includes codec initialization data, Track ID mappings for multiplexed segments, and timestamp offsets (e.g., edit lists).

*NOTE: The byte stream format specifications in the byte stream format registry [MSE-REGISTRY] contain format specific examples.*

### Media Segment

A sequence of bytes that contain packetized & timestamped media data for a portion of the media timeline. Media segments are always associated with the most recently appended initialization segment.

*NOTE: The byte stream format specifications in the byte stream format registry [MSE-REGISTRY] contain format specific examples.*

### MediaSource object URL

A MediaSource object URL is a unique Blob URI [FILE-API] created by `createObjectURL()`. It is used to attach a `MediaSource` object to an HTMLMediaElement.

These URLs are the same as a Blob URI, except that anything in the definition of that feature that refers to File and Blob objects is hereby extended to also apply to `MediaSource` objects.

The origin of the MediaSource object URL is the relevant settings object of `this` during the call to `createObjectURL()`.

*NOTE: For example, the origin of the MediaSource object URL affects the way that the media element is consumed by canvas.*

### Parent Media Source

The parent media source of a `SourceBuffer` object is the `MediaSource` object that created it.

### Presentation Start Time

The presentation start time is the earliest time point in the presentation and specifies the initial playback position and earliest possible position. All presentations created using this specification have a presentation start time of 0.

*NOTE: For the purposes of determining if `HTMLMediaElement.buffered` contains a `TimeRange` that includes the current playback position, implementations MAY choose to allow a current playback position at or after presentation start time and before the first `TimeRange` to play the first `TimeRange` if that `TimeRange` starts within a reasonably short time, like 1 second, after presentation start time. This allowance accommodates the reality that muxed streams commonly do not begin all tracks precisely at presentation start time. Implementations MUST report the actual buffered range, regardless of this allowance.*

### *Pre*sentation Interval

The presentation interval of a coded frame is the time interval from its presentation timestamp to the presentation timestamp plus the coded frame's duration. For example, if a coded frame has a presentation timestamp of 10 seconds and a coded frame duration of 100 milliseconds, then the presentation interval would be [10-10.1). Note that the start of the range is inclusive, but the end of the range is exclusive.

### Presentation Order

The order that coded frames are rendered in the presentation. The presentation order is achieved by ordering coded frames in monotonically increasing order by their presentation timestamps.

### Presentation Timestamp

A reference to a specific time in the presentation. The presentation timestamp in a coded frame indicates when the frame *SHOULD* be rendered.

### Random Access Point

A position in a media segment where decoding and continuous playback can begin without relying on any previous data in the segment. For video this tends to be the location of I-frames. In the case of audio, most audio frames can be treated as a random access point. Since video tracks tend to

have a more sparse distribution of random access points, the location of these points are usually considered the random access points for multiplexed streams.

**SourceBuffer byte stream format specification**

The specific byte stream format specification that describes the format of the byte stream accepted by a `SourceBuffer` instance. The byte stream format specification, for a `SourceBuffer` object, is selected based on the type passed to the `addSourceBuffer()` call that created the object.

**SourceBuffer configuration**

A specific set of tracks distributed across one or more `SourceBuffer` objects owned by a single `MediaSource` instance.

Implementations *MUST* support at least 1 `MediaSource` object with the following configurations:

- A single SourceBuffer with 1 audio track and/or 1 video track.
- Two SourceBuffers with one handling a single audio track and the other handling a single video track.

MediaSource objects *MUST* support each of the configurations above, but they are only required to support one configuration at a time. Supporting multiple configurations at once or additional configurations is a quality of implementation issue.

**Track Description**

A byte stream format specific structure that provides the Track ID, codec configuration, and other metadata for a single track. Each track description inside a single initialization segment has a unique Track ID. The user agent *MUST* run the append error algorithm if the Track ID is not unique within the initialization segment.

**Track ID**

A Track ID is a byte stream format specific identifier that marks sections of the byte stream as being part of a specific track. The Track ID in a track description identifies which sections of a media segment belong to that track.

# 2. `MediaSource` Object

## 2.4 Algorithms

### 2.4.1 Attaching to a media element

A `MediaSource` object can be attached to a media element by assigning a MediaSource object URL to the media element `src` attribute or the src attribute of a <source> inside a media element. A MediaSource object URL is created by passing a MediaSource object to `createObjectURL()`.

If the resource fetch algorithm was invoked with a media provider object that is a MediaSource object or a URL record whose object is a MediaSource object, then let mode be local, skip the first step in the resource fetch algorithm (which may otherwise set mode to remote) and add the steps and clarifications below to the "*Otherwise (mode is local)*" section of the resource fetch algorithm.

*NOTE: The resource fetch algorithm's first step is expected to eventually align with selecting local mode for URL records whose objects are media provider objects. The intent is that if the HTMLMediaElement's `src` attribute or selected child `<source>`'s `src` attribute is a `blob:` URL matching a MediaSource object URL when the respective `src` attribute was last changed, then that MediaSource object is used as the media provider object and current media resource in the local mode logic in the resource fetch algorithm. This also means that the remote mode logic that includes observance of any preload attribute is skipped when a MediaSource object is attached. Even with that eventual change to [HTML], the execution of the following steps at the beginning of the local mode logic is still required when the current media resource is a MediaSource object.*

*NOTE: Relative to the action which triggered the media element's resource selection algorithm, these steps are asynchronous. The resource fetch algorithm is run after the task that invoked the resource selection algorithm is allowed to continue and a stable state is reached. Implementations may delay the steps in the "Otherwise" clause, below, until the MediaSource object is ready for use.*

**If `readyState` is NOT set to `"closed"`**

Run the "*If the media data cannot be fetched at all, due to network errors, causing the user agent to give up trying to fetch the resource*" steps of the resource fetch algorithm's media data processing steps list.

**Otherwise**

1. Set the media element's delaying-the-load-event-flag to false.
2. Set the `readyState` attribute to `"open"`.
3. Queue a task to fire a simple event named `sourceopen` at the `MediaSource`.
4. Continue the resource fetch algorithm by running the remaining "*Otherwise (mode is local)*" steps, with these clarifications:
    1. Text in the resource fetch algorithm or the media data processing steps list that refers to "the download", "bytes received", or "whenever new data for the current media resource becomes available" refers to data passed in via `appendBuffer()`.
    2. References to HTTP in the resource fetch algorithm and the media data processing steps list do not apply because the HTMLMediaElement does not fetch media data via HTTP when a `MediaSource` is attached.

*NOTE: An attached MediaSource does not use the remote mode steps in the resource fetch algorithm, so the media element will not fire "suspend" events. Though future versions of this specification will likely remove "progress" and "stalled" events from a media element with an attached MediaSource, user agents conforming to this version of the specification may still fire these two events as these [HTML] references changed after implementations of this specification stabilized.*

### 2.4.2 Detaching from a media element

The following steps are run in any case where the media element is going to transition to NETWORK_EMPTY and queue a task to fire a simple event named emptied at the media element. These steps *SHOULD* be run right before the transition.

1. Set the `readyState` attribute to `"closed"`.
2. Update `duration` to NaN.
3. Remove all the `SourceBuffer` objects from `activeSourceBuffers`.

4. Queue a task to fire a simple event named `removesourcebuffer` at `activeSourceBuffers`.
5. Remove all the `SourceBuffer` objects from `sourceBuffers`.
6. Queue a task to fire a simple event named `removesourcebuffer` at `sourceBuffers`.
7. Queue a task to fire a simple event named `sourceclose` at the `MediaSource`.

*NOTE: Going forward, this algorithm is intended to be externally called and run in any case where the attached `MediaSource`, if any, must be detached from the media element. It MAY be called on HTMLMediaElement [HTML] operations like load() and resource fetch algorithm failures in addition to, or in place of, when the media element transitions to NETWORK_EMPTY. Resource fetch algorithm failures are those which abort either the resource fetch algorithm or the resource selection algorithm, with the exception that the "Final step" [HTML] is not considered a failure that triggers detachment.*

### 2.4.3 Seeking

Run the following steps as part of the "*Wait until the user agent has established whether or not the media data for the new playback position is available, and, if it is, until it has decoded enough data to play back that position"* step of the seek algorithm:

1. *NOTE*
   *The media element looks for media segments containing the new playback position in each `SourceBuffer` object in `activeSourceBuffers`. Any position within a `TimeRange` in the current value of the `HTMLMediaElement.buffered` attribute has all necessary media segments buffered for that position.*
   **If new playback position is not in any `TimeRange` of `HTMLMediaElement.buffered`**
   1. If the `HTMLMediaElement.readyState` attribute is greater than `HAVE_METADATA`, then set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.
      *NOTE*
      *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
   2. The media element waits until an `appendBuffer()` call causes the coded frame processing algorithm to set the `HTMLMediaElement.readyState` attribute to a value greater than `HAVE_METADATA`.
      *NOTE*
      *The web application can use `buffered` and `HTMLMediaElement.buffered` to determine what the media element needs to resume playback.*
2. **Otherwise**
   Continue
   *NOTE*
   *If the `readyState` attribute is `"ended"` and the new playback position is within a `TimeRange` currently in `HTMLMediaElement.buffered`, then the seek operation must continue to completion here even if one or more currently selected or enabled track buffers' largest range end timestamp is less than new playback position. This condition should only occur due to logic in `buffered` when `readyState` is `"ended"`.*
3. The media element resets all decoders and initializes each one with data from the appropriate initialization segment.
4. The media element feeds coded frames from the active track buffers into the decoders starting with the closest random access point before the new playback position.
5. Resume the seek algorithm at the "*Await a stable state*" step.

### 2.4.4 SourceBuffer Monitoring

The following steps are periodically run during playback to make sure that all of the `SourceBuffer` objects in `activeSourceBuffers` have enough data to ensure uninterrupted playback. Changes to `activeSourceBuffers` also cause these steps to run because they affect the conditions that trigger state transitions.

Having enough data to ensure uninterrupted playback is an implementation specific condition where the user agent determines that it currently has enough data to play the presentation without stalling for a meaningful period of time. This condition is constantly evaluated to determine when to transition the media element into and out of the `HAVE_ENOUGH_DATA` ready state. These transitions indicate when the user agent believes it has enough data buffered or it needs more data respectively.

*NOTE: An implementation MAY choose to use bytes buffered, time buffered, the append rate, or any other metric it sees fit to determine when it has enough data. The metrics used MAY change during playback so web applications SHOULD only rely on the value of `HTMLMediaElement.readyState` to determine whether more data is needed or not.*

*NOTE: When the media element needs more data, the user agent SHOULD transition it from `HAVE_ENOUGH_DATA` to `HAVE_FUTURE_DATA` early enough for a web application to be able to respond without causing an interruption in playback. For example, transitioning when the current playback position is 500ms before the end of the buffered data gives the application roughly 500ms to append more data before playback stalls.*

**If the the `HTMLMediaElement.readyState` attribute equals `HAVE_NOTHING`:**

1. Abort these steps.

**If `HTMLMediaElement.buffered` does not contain a `TimeRange` for the current playback position:**

1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.
   *NOTE*
   *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
2. Abort these steps.

**If `HTMLMediaElement.buffered` contains a `TimeRange` that includes the current playback position and enough data to ensure uninterrupted playback:**

1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_ENOUGH_DATA`.
   *NOTE*
   *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
2. Playback may resume at this point if it was previously suspended by a transition to `HAVE_CURRENT_DATA`.
3. Abort these steps.

**If `HTMLMediaElement.buffered` contains a `TimeRange` that includes the current playback position and some time beyond the current playback position, then run the following steps:**

1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_FUTURE_DATA`.
   *NOTE*
   *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
2. Playback may resume at this point if it was previously suspended by a transition to `HAVE_CURRENT_DATA`.
3. Abort these steps.

**If `HTMLMediaElement.buffered` contains a `TimeRange` that ends at the current playback position and does not have a range covering the time immediately after the current position:**

1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_CURRENT_DATA`.
   *NOTE*
   *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
2. Playback is suspended at this point since the media element doesn't have enough data to advance the media timeline.
3. Abort these steps.

### 2.4.5 Changes to selected/enabled track state

During playback `activeSourceBuffers` needs to be updated if the selected video track, the enabled audio track(s), or a text track mode changes. When one or more of these changes occur the following steps need to be followed.

**If the selected video track changes, then run the following steps:**

1. If the `SourceBuffer` associated with the previously selected video track is not associated with any other enabled tracks, run the following steps:
   1. Remove the `SourceBuffer` from `activeSourceBuffers`.
   2. Queue a task to fire a simple event named `removesourcebuffer` at `activeSourceBuffers`
2. If the `SourceBuffer` associated with the newly selected video track is not already in `activeSourceBuffers`, run the following steps:
   1. Add the `SourceBuffer` to `activeSourceBuffers`.
   2. Queue a task to fire a simple event named `addsourcebuffer` at `activeSourceBuffers`

**If an audio track becomes disabled and the `SourceBuffer` associated with this track is not associated with any other enabled or selected track, then run the following steps:**

1. Remove the `SourceBuffer` associated with the audio track from `activeSourceBuffers`
2. Queue a task to fire a simple event named `removesourcebuffer` at `activeSourceBuffers`

**If an audio track becomes enabled and the `SourceBuffer` associated with this track is not already in `activeSourceBuffers`, then run the following steps:**

1. Add the `SourceBuffer` associated with the audio track to `activeSourceBuffers`
2. Queue a task to fire a simple event named `addsourcebuffer` at `activeSourceBuffers`

**If a text track mode becomes `"disabled"` and the `SourceBuffer` associated with this track is not associated with any other enabled or selected track, then run the following steps:**

1. Remove the `SourceBuffer` associated with the text track from `activeSourceBuffers`
2. Queue a task to fire a simple event named `removesourcebuffer` at `activeSourceBuffers`

**If a text track mode becomes `"showing"` or `"hidden"` and the `SourceBuffer` associated with this track is not already in `activeSourceBuffers`, then run the following steps:**

1. Add the `SourceBuffer` associated with the text track to `activeSourceBuffers`
2. Queue a task to fire a simple event named `addsourcebuffer` at `activeSourceBuffers`

### 2.4.6 Duration change

Follow these steps when `duration` needs to change to a new duration.

1. If the current value of `duration` is equal to new duration, then return.
2. If new duration is less than the highest presentation timestamp of any buffered coded frames for all `SourceBuffer` objects in `sourceBuffers`, then throw an `InvalidStateError` exception and abort these steps.
   NOTE
   Duration reductions that would truncate currently buffered media are disallowed. When truncation is necessary, use `remove()` to reduce the buffered range before updating `duration`.
3. Let highest end time be the largest track buffer ranges end time across all the track buffers across all `SourceBuffer` objects in `sourceBuffers`.
4. If new duration is less than highest end time, then
   NOTE
   This condition can occur because the coded frame removal algorithm preserves coded frames that start before the start of the removal range.
   1. Update new duration to equal highest end time.
5. Update `duration` to new duration.
6. Update the `media duration` to new duration and run the HTMLMediaElement duration change algorithm.

### 2.4.7 End of stream algorithm

This algorithm gets called when the application signals the end of stream via an `endOfStream()` call or an algorithm needs to signal a decode error. This algorithm takes an error parameter that indicates whether an error will be signalled.

1. Change the `readyState` attribute value to `"ended"`.
2. Queue a task to fire a simple event named `sourceended` at the `MediaSource`.
3. **If error is not set**
    1. Run the duration change algorithm with new duration set to the largest track buffer ranges end time across all the track buffers across all `SourceBuffer` objects in `sourceBuffers`.

       *NOTE*
       *This allows the duration to properly reflect the end of the appended media segments. For example, if the duration was explicitly set to 10 seconds and only media segments for 0 to 5 seconds were appended before endOfStream() was called, then the duration will get updated to 5 seconds.*
    2. Notify the media element that it now has all of the media data.
4. **If error is set to `"network"`**

   **If the `HTMLMediaElement.readyState` attribute equals `HAVE_NOTHING`**

   Run the "*If the media data cannot be fetched at all, due to network errors, causing the user agent to give up trying to fetch the resource*" steps of the resource fetch algorithm's media data processing steps list.

   **If the `HTMLMediaElement.readyState` attribute is greater than `HAVE_NOTHING`**

   Run the "*If the connection is interrupted after some media data has been received, causing the user agent to give up trying to fetch the resource*" steps of the resource fetch algorithm's media data processing steps list.

   **If error is set to `"decode"`**

   **If the `HTMLMediaElement.readyState` attribute equals `HAVE_NOTHING`**

   Run the "*If the media data can be fetched but is found by inspection to be in an unsupported format, or can otherwise not be rendered at all*" steps of the resource fetch algorithm's media data processing steps list.

   **If the `HTMLMediaElement.readyState` attribute is greater than `HAVE_NOTHING`**

   Run the media data is corrupted steps of the resource fetch algorithm's media data processing steps list.

# 3. `SourceBuffer` Object

**WebIDL**

```
enum AppendMode {
    "segments",
    "sequence"

};
```

**Enumeration description**

`segments`

: The timestamps in the media segment determine where the coded frames are placed in the presentation. Media segments can be appended in any order.

`sequence`

: Media segments will be treated as adjacent in time independent of the timestamps in the media segment. Coded frames in a new media segment will be placed immediately after the coded frames in the previous media segment. The `timestampOffset` attribute will be updated if a new offset is needed to make the new media segments adjacent to the previous media segment. Setting the `timestampOffset` attribute in `"sequence"` mode allows a media segment to be placed at a specific position in the timeline without any knowledge of the timestamps in the media segment.

## 3.1 Attributes

**`mode` of type `AppendMode`**

Controls how a sequence of media segments are handled. This attribute is initially set by `addSourceBuffer()` after the object is created.

On getting, Return the initial value or the last value that was successfully set.

On setting, run the following steps:

1. If this object has been removed from the `sourceBuffers` attribute of the parent media source, then throw an `InvalidStateError` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `InvalidStateError` exception and abort these steps.
3. Let new mode equal the new value being assigned to this attribute.
4. If generate timestamps flag equals true and new mode equals `"segments"`, then throw a `TypeError` exception and abort these steps.
5. If the `readyState` attribute of the parent media source is in the `"ended"` state then run the following steps:
    1. Set the `readyState` attribute of the parent media source to `"open"`
    2. Queue a task to fire a simple event named `sourceopen` at the parent media source.
6. If the append state equals PARSING_MEDIA_SEGMENT, then throw an `InvalidStateError` and abort these steps.
7. If the new mode equals `"sequence"`, then set the group start timestamp to the group end timestamp.
8. Update the attribute to new mode.

**`updating` of type `boolean`, readonly**

Indicates whether the asynchronous continuation of an `appendBuffer()` or `remove()` operation is still being processed. This attribute is initially set to false when the object is created.

**`buffered` of type `TimeRanges`, readonly**

Indicates what `TimeRanges` are buffered in the `SourceBuffer`. This attribute is initially set to an empty `TimeRanges` object when the object is created.

When the attribute is read the following steps *MUST* occur:

1. If this object has been removed from the `sourceBuffers` attribute of the parent media source then throw an `InvalidStateError` exception and abort these steps.
2. Let highest end time be the largest track buffer ranges end time across all the track buffers managed by this `SourceBuffer` object.
3. Let intersection ranges equal a `TimeRange` object containing a single range from 0 to highest end time.
4. For each audio and video track buffer managed by this `SourceBuffer`, run the following steps:
   NOTE
   Text track-buffers are included in the calculation of highest end time, above, but excluded from the buffered range calculation here. They are not necessarily continuous, nor should any discontinuity within them trigger playback stall when the other media tracks are continuous over the same time range.
   1. Let track ranges equal the track buffer ranges for the current track buffer.
   2. If `readyState` is `"ended"`, then set the end time on the last range in track ranges to highest end time.
   3. Let new intersection ranges equal the intersection between the intersection ranges and the track ranges.
   4. Replace the ranges in intersection ranges with the new intersection ranges.
5. If intersection ranges does not contain the exact same range information as the current value of this attribute, then update the current value of this attribute to intersection ranges.
6. Return the current value of this attribute.

### `timestampOffset` of type `double`

Controls the offset applied to timestamps inside subsequent media segments that are appended to this `SourceBuffer`. The `timestampOffset` is initially set to 0 which indicates that no offset is being applied.

On getting, Return the initial value or the last value that was successfully set.

On setting, run the following steps:

1. Let new timestamp offset equal the new value being assigned to this attribute.
2. If this object has been removed from the `sourceBuffers` attribute of the parent media source, then throw an `InvalidStateError` exception and abort these steps.
3. If the `updating` attribute equals true, then throw an `InvalidStateError` exception and abort these steps.
4. If the `readyState` attribute of the parent media source is in the `"ended"` state then run the following steps:
   1. Set the `readyState` attribute of the parent media source to `"open"`
   2. Queue a task to fire a simple event named `sourceopen` at the parent media source.
5. If the append state equals PARSING_MEDIA_SEGMENT, then throw an `InvalidStateError` and abort these steps.

6. If the `mode` attribute equals `"sequence"`, then set the group start timestamp to new timestamp offset.
7. Update the attribute to new timestamp offset.

### `audioTracks` of type `AudioTrackList`, readonly

The list of `AudioTrack` objects created by this object.

### `videoTracks` of type `VideoTrackList`, readonly

The list of `VideoTrack` objects created by this object.

### `textTracks` of type `TextTrackList`, readonly

The list of `TextTrack` objects created by this object.

### `appendWindowStart` of type `double`

The presentation timestamp for the start of the append window. This attribute is initially set to the presentation start time.

On getting, Return the initial value or the last value that was successfully set.

On setting, run the following steps:

1. If this object has been removed from the `sourceBuffers` attribute of the parent media source, then throw an `InvalidStateError` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `InvalidStateError` exception and abort these steps.
3. If the new value is less than 0 or greater than or equal to `appendWindowEnd` then throw a `TypeError` exception and abort these steps.
4. Update the attribute to the new value.

### `appendWindowEnd` of type `unrestricted double`

The presentation timestamp for the end of the append window. This attribute is initially set to positive Infinity.

On getting, Return the initial value or the last value that was successfully set.

On setting, run the following steps:

1. If this object has been removed from the `sourceBuffers` attribute of the parent media source, then throw an `InvalidStateError` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `InvalidStateError` exception and abort these steps.
3. If the new value equals NaN, then throw a `TypeError` and abort these steps.
4. If the new value is less than or equal to `appendWindowStart` then throw a `TypeError` exception and abort these steps.
5. Update the attribute to the new value.

**onupdatestart of type EventHandler**

The event handler for the `updatestart` event.

**onupdate of type EventHandler**

The event handler for the `update` event.

**onupdateend of type EventHandler**

The event handler for the `updateend` event.

**onerror of type EventHandler**

The event handler for the `error` event.

**onabort of type EventHandler**

The event handler for the `abort` event.

## 3.5 Algorithms

### 3.5.1 Segment Parser Loop

All SourceBuffer objects have an internal append state variable that keeps track of the high-level segment parsing state. It is initially set to WAITING_FOR_SEGMENT and can transition to the following states as data is appended.

| Append state name | Description |
|---|---|
| WAITING_FOR_SEGMENT | Waiting for the start of an initialization segment or media segment to be appended. |
| PARSING_INIT_SEGMENT | Currently parsing an initialization segment. |
| PARSING_MEDIA_SEGMENT | Currently parsing a media segment. |

The input buffer is a byte buffer that is used to hold unparsed bytes across `appendBuffer()` calls. The buffer is empty when the SourceBuffer object is created.

The buffer full flag keeps track of whether `appendBuffer()` is allowed to accept more bytes. It is set to false when the SourceBuffer object is created and gets updated as data is appended and removed.

The group start timestamp variable keeps track of the starting timestamp for a new coded frame group in the `"sequence"` mode. It is unset when the SourceBuffer object is created and gets updated when the `mode` attribute equals `"sequence"` and the `timestampOffset` attribute is set, or the coded frame processing algorithm runs.

The group end timestamp variable stores the highest coded frame end timestamp across all coded frames in the current coded frame group. It is set to 0 when the SourceBuffer object is created and gets updated by the coded frame processing algorithm.

*NOTE: The group end timestamp stores the highest coded frame end timestamp across all track buffers in a `SourceBuffer`. Therefore, care should be taken in setting the `mode` attribute when appending multiplexed segments in which the timestamps are not aligned across tracks.*

The generate timestamps flag is a boolean variable that keeps track of whether timestamps need to be generated for the coded frames passed to the coded frame processing algorithm. This flag is set by `addSourceBuffer()` when the SourceBuffer object is created.

When the segment parser loop algorithm is invoked, run the following steps:

1. *Loop Top:* If the input buffer is empty, then jump to the *need more data* step below.
2. If the input buffer contains bytes that violate the SourceBuffer byte stream format specification, then run the append error algorithm and abort this algorithm.
3. Remove any bytes that the byte stream format specifications say *MUST* be ignored from the start of the input buffer.
4. If the append state equals WAITING_FOR_SEGMENT, then run the following steps:
    1. If the beginning of the input buffer indicates the start of an initialization segment, set the append state to PARSING_INIT_SEGMENT.
    2. If the beginning of the input buffer indicates the start of a media segment, set append state to PARSING_MEDIA_SEGMENT.
    3. Jump to the *loop top* step above.
5. If the append state equals PARSING_INIT_SEGMENT, then run the following steps:
    1. If the input buffer does not contain a complete initialization segment yet, then jump to the *need more data* step below.
    2. Run the initialization segment received algorithm.
    3. Remove the initialization segment bytes from the beginning of the input buffer.
    4. Set append state to WAITING_FOR_SEGMENT.
    5. Jump to the *loop top* step above.
6. If the append state equals PARSING_MEDIA_SEGMENT, then run the following steps:
    1. If the first initialization segment received flag is false, then run the append error algorithm and abort this algorithm.
    2. If the input buffer contains one or more event message boxes, run the event message box processing algorithm
    3. If the input buffer contains one or more complete coded frames, then run the coded frame processing algorithm.
    4. *NOTE: The frequency at which the coded frame processing algorithm is run is implementation-specific. The coded frame processing algorithm MAY be called when the input buffer contains the complete media segment or it MAY be called multiple times as complete coded frames are added to the input buffer.*
    5. If this `SourceBuffer` is full and cannot accept more media data, then set the buffer full flag to true.
    6. If the input buffer does not contain a complete media segment, then jump to the *need more data* step below.
    7. Remove the media segment bytes from the beginning of the input buffer.
    8. Set append state to WAITING_FOR_SEGMENT.
    9. Jump to the *loop top* step above.
7. *Need more data:* Return control to the calling algorithm.

### 3.5.2 Reset Parser State

When the parser state needs to be reset, run the following steps:

1. If the append state equals PARSING_MEDIA_SEGMENT and the input buffer contains some complete coded frames, then run the coded frame processing algorithm until all of these complete coded frames have been processed.
2. Unset the last decode timestamp on all track buffers.
3. Unset the last frame duration on all track buffers.
4. Unset the highest end timestamp on all track buffers.
5. Set the need random access point flag on all track buffers to true.
6. If the `mode` attribute equals `"sequence"`, then set the group start timestamp to the group end timestamp
7. Remove all bytes from the input buffer.
8. Set append state to WAITING_FOR_SEGMENT.

### 3.5.3 Append Error Algorithm

This algorithm is called when an error occurs during an append.

1. Run the reset parser state algorithm.
2. Set the `updating` attribute to false.
3. Queue a task to fire a simple event named `error` at this `SourceBuffer` object.
4. Queue a task to fire a simple event named `updateend` at this `SourceBuffer` object.
5. Run the end of stream algorithm with the error parameter set to `"decode"`.

### 3.5.4 Prepare Append Algorithm

When an append operation begins, the follow steps are run to validate and prepare the `SourceBuffer`.

1. If the `SourceBuffer` has been removed from the `sourceBuffers` attribute of the parent media source then throw an `InvalidStateError` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `InvalidStateError` exception and abort these steps.
3. If the `HTMLMediaElement.error` attribute is not null, then throw an `InvalidStateError` exception and abort these steps.
4. If the `readyState` attribute of the parent media source is in the `"ended"` state then run the following steps:
    1. Set the `readyState` attribute of the parent media source to `"open"`
    2. Queue a task to fire a simple event named `sourceopen` at the parent media source.
5. Run the coded frame eviction algorithm.
6. If the buffer full flag equals true, then throw a `QuotaExceededError` exception and abort these step.
   *NOTE: This is the signal that the implementation was unable to evict enough data to accommodate the append or the append is too big. The web application SHOULD use `remove()` to explicitly free up space and/or reduce the size of the append.*

### 3.5.5 Buffer Append Algorithm

When `appendBuffer()` is called, the following steps are run to process the appended data.

1. Run the segment parser loop algorithm.

2. If the segment parser loop algorithm in the previous step was aborted, then abort this algorithm.
3. Set the `updating` attribute to false.
4. Queue a task to fire a simple event named `update` at this `SourceBuffer` object.
5. Queue a task to fire a simple event named `updateend` at this `SourceBuffer` object.

### 3.5.6 Range Removal

Follow these steps when a caller needs to initiate a JavaScript visible range removal operation that blocks other SourceBuffer updates:

1. Let start equal the starting presentation timestamp for the removal range, in seconds measured from presentation start time.
2. Let end equal the end presentation timestamp for the removal range, in seconds measured from presentation start time.
3. Set the `updating` attribute to true.
4. Queue a task to fire a simple event named `updatestart` at this `SourceBuffer` object.
5. Return control to the caller and run the rest of the steps asynchronously.
6. Run the coded frame removal algorithm with start and end as the start and end of the removal range.
7. Set the `updating` attribute to false.
8. Queue a task to fire a simple event named `update` at this `SourceBuffer` object.
9. Queue a task to fire a simple event named `updateend` at this `SourceBuffer` object.

### 3.5.7 Initialization Segment Received

The following steps are run when the segment parser loop successfully parses a complete initialization segment:

Each SourceBuffer object has an internal first initialization segment received flag that tracks whether the first initialization segment has been appended and received by this algorithm. This flag is set to false when the SourceBuffer is created and updated by the algorithm below.

1. Update the `duration` attribute if it currently equals NaN:
   **If the initialization segment contains a duration:**
   Run the duration change algorithm with new duration set to the duration in the initialization segment.
   **Otherwise:**
   Run the duration change algorithm with new duration set to positive Infinity.
2. If the initialization segment has no audio, video, or text tracks, then run the append error algorithm and abort these steps.
3. If the first initialization segment received flag is true, then run the following steps:
    1. Verify the following properties. If any of the checks fail then run the append error algorithm and abort these steps.
        1. The number of audio, video, and text tracks match what was in the first initialization segment.
        2. The codecs for each track, match what was specified in the first initialization segment.
        3. If more than one track for a single type are present (e.g., 2 audio tracks), then the Track IDs match the ones in the first initialization segment.

2. Add the appropriate track descriptions from this initialization segment to each of the track buffers.
3. Set the need random access point flag on all track buffers to true.

4. Let active track flag equal false.
5. If the first initialization segment received flag is false, then run the following steps:
    1. If the initialization segment contains tracks with codecs the user agent does not support, then run the append error algorithm and abort these steps.
    *NOTE: User agents MAY consider codecs, that would otherwise be supported, as "not supported" here if the codecs were not specified in the type parameter passed to* `addSourceBuffer()`.
    *For example, MediaSource.isTypeSupported('video/webm;codecs="vp8,vorbis"') may return true, but if* `addSourceBuffer()` *was called with 'video/webm;codecs="vp8"' and a Vorbis track appears in the initialization segment, then the user agent MAY use this step to trigger a decode error.*
    2. For each audio track in the initialization segment, run following steps:
        1. Let audio byte stream track ID be the Track ID for the current track being processed.
        2. Let audio language be a BCP 47 language tag for the language specified in the initialization segment for this track or an empty string if no language info is present.
        3. If audio language equals the 'und' BCP 47 value, then assign an empty string to audio language.
        4. Let audio label be a label specified in the initialization segment for this track or an empty string if no label info is present.
        5. Let audio kinds be a sequence of kind strings specified in the initialization segment for this track or a sequence with a single empty string element in it if no kind information is provided.
        6. For each value in audio kinds, run the following steps:
            1. Let current audio kind equal the value from audio kinds for this iteration of the loop.
            2. Let new audio track be a new `AudioTrack` object.
            3. Generate a unique ID and assign it to the `id` property on new audio track.
            4. Assign audio language to the `language` property on new audio track.
            5. Assign audio label to the `label` property on new audio track.
            6. Assign current audio kind to the `kind` property on new audio track.
            7. If `audioTracks.length` equals 0, then run the following steps:
                1. Set the `enabled` property on new audio track to true.
                2. Set active track flag to true.
            8. Add new audio track to the `audioTracks` attribute on this `SourceBuffer` object.
            *NOTE*
            *This should trigger* `AudioTrackList` *[HTML] logic to queue a task to fire a trusted event named* `addtrack`*, that does not bubble and is not cancelable, and that uses the* `TrackEvent` *interface, with the* `track` *attribute initialized to new audio track, at the* `AudioTrackList` *object referenced by the* `audioTracks` *attribute on this* `SourceBuffer` *object.*
            9. Add new audio track to the `audioTracks` attribute on the HTMLMediaElement.
            *NOTE: This should trigger* `AudioTrackList` *[HTML] logic to queue a task to fire a trusted event named* `addtrack`*, that does not bubble and is not cancelable, and that uses the* `TrackEvent` *interface, with the* `track`

*attribute initialized to new audio track, at the `AudioTrackList` object referenced by the `audioTracks` attribute on the HTMLMediaElement.*

7. Create a new track buffer to store coded frames for this track.
8. Add the track description for this track to the track buffer.

3. For each video track in the initialization segment, run following steps:
    1. Let video byte stream track ID be the Track ID for the current track being processed.
    2. Let video language be a BCP 47 language tag for the language specified in the initialization segment for this track or an empty string if no language info is present.
    3. If video language equals the 'und' BCP 47 value, then assign an empty string to video language.
    4. Let video label be a label specified in the initialization segment for this track or an empty string if no label info is present.
    5. Let video kinds be a sequence of kind strings specified in the initialization segment for this track or a sequence with a single empty string element in it if no kind information is provided.
    6. For each value in video kinds, run the following steps:
        1. Let current video kind equal the value from video kinds for this iteration of the loop.
        2. Let new video track be a new `VideoTrack` object.
        3. Generate a unique ID and assign it to the `id` property on new video track.
        4. Assign video language to the `language` property on new video track.
        5. Assign video label to the `label` property on new video track.
        6. Assign current video kind to the `kind` property on new video track.
        7. If `videoTracks.length` equals 0, then run the following steps:
            1. Set the `selected` property on new video track to true.
            2. Set active track flag to true.
        8. Add new video track to the `videoTracks` attribute on this `SourceBuffer` object.
        *NOTE: This should trigger `VideoTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to new video track, at the `VideoTrackList` object referenced by the `videoTracks` attribute on this `SourceBuffer` object.*
        9. Add new video track to the `videoTracks` attribute on the HTMLMediaElement.
        *NOTE: This should trigger `VideoTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to new video track, at the `VideoTrackList` object referenced by the `videoTracks` attribute on the HTMLMediaElement.*
    7. Create a new track buffer to store coded frames for this track.
    8. Add the track description for this track to the track buffer.
4. For each text track in the initialization segment, run following steps:
    1. Let text byte stream track ID be the Track ID for the current track being processed.
    2. Let text language be a BCP 47 language tag for the language specified in the initialization segment for this track or an empty string if no language info is present.
    3. If text language equals the 'und' BCP 47 value, then assign an empty string to text language.
    4. Let text label be a label specified in the initialization segment for this track or an empty string if no label info is present.

5. Let text kinds be a sequence of kind strings specified in the initialization segment for this track or a sequence with a single empty string element in it if no kind information is provided.
6. For each value in text kinds, run the following steps:
    1. Let current text kind equal the value from text kinds for this iteration of the loop.
    2. Let new text track be a new `TextTrack` object.
    3. Generate a unique ID and assign it to the `id` property on new text track.
    4. Assign text language to the `language` property on new text track.
    5. Assign text label to the `label` property on new text track.
    6. Assign current text kind to the `kind` property on new text track.
    7. Populate the remaining properties on new text track with the appropriate information from the initialization segment.
    8. If the `mode` property on new text track equals `"showing"` or `"hidden"`, then set active track flag to true.
    9. Add new text track to the `textTracks` attribute on this `SourceBuffer` object.
    *NOTE*
    *This should trigger `TextTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to new text track, at the `TextTrackList` object referenced by the `textTracks` attribute on this `SourceBuffer` object.*
    10. Add new text track to the `textTracks` attribute on the HTMLMediaElement.
    *NOTE*
    *This should trigger `TextTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to new text track, at the `TextTrackList` object referenced by the `textTracks` attribute on the HTMLMediaElement.*
7. Create a new track buffer to store coded frames for this track.
8. Add the track description for this track to the track buffer.

5. If active track flag equals true, then run the following steps:
    1. Add this `SourceBuffer` to `activeSourceBuffers`.
    2. Queue a task to fire a simple event named `addsourcebuffer` at `activeSourceBuffers`
6. Set first initialization segment received flag to true.

6. If the `HTMLMediaElement.readyState` attribute is `HAVE_NOTHING`, then run the following steps:
    1. If one or more objects in `sourceBuffers` have first initialization segment received flag set to false, then abort these steps.
    2. Set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.
    *NOTE*
    *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement. This particular transition should trigger HTMLMediaElement logic to queue a task to fire a simple event named `loadedmetadata` at the media element.*
7. If the active track flag equals true and the `HTMLMediaElement.readyState` attribute is greater than `HAVE_CURRENT_DATA`, then set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.
*NOTE*

*Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*

### 3.5.8 Event Message Box Processing

Refer to the DASH-IF Event Processing document

### 3.5.8 Coded Frame Processing

When complete coded frames have been parsed by the segment parser loop then the following steps are run:

1. For each coded frame in the media segment run the following steps:
    1. *Loop Top:*
       **If generate timestamps flag equals true:**
        1. Let presentation timestamp equal 0.
        2. Let decode timestamp equal 0.
    2. **Otherwise:**
        1. Let presentation timestamp be a double precision floating point representation of the coded frame's presentation timestamp in seconds.
           *NOTE: Special processing may be needed to determine the presentation and decode timestamps for timed text frames since this information may not be explicitly present in the underlying format or may be dependent on the order of the frames. Some metadata text tracks, like MPEG2-TS PSI data, may only have implied timestamps. Format specific rules for these situations SHOULD be in the byte stream format specifications or in separate extension specifications.*
        2. Let decode timestamp be a double precision floating point representation of the coded frame's decode timestamp in seconds.
           *NOTE: Implementations don't have to internally store timestamps in a double precision floating point representation. This representation is used here because it is the represention for timestamps in the HTML spec. The intention here is to make the behavior clear without adding unnecessary complexity to the algorithm to deal with the fact that adding a timestampOffset may cause a timestamp rollover in the underlying timestamp representation used by the byte stream format. Implementations can use any internal timestamp representation they wish, but the addition of timestampOffset SHOULD behave in a similar manner to what would happen if a double precision floating point representation was used.*
    3. Let frame duration be a double precision floating point representation of the coded frame's duration in seconds.
    4. If `mode` equals `"sequence"` and group start timestamp is set, then run the following steps:
        1. Set `timestampOffset` equal to group start timestamp - presentation timestamp.
        2. Set group end timestamp equal to group start timestamp.
        3. Set the need random access point flag on all track buffers to true.
        4. Unset group start timestamp.
    5. If `timestampOffset` is not 0, then run the following steps:
        1. Add `timestampOffset` to the presentation timestamp.
        2. Add `timestampOffset` to the decode timestamp.
    6. Let track buffer equal the track buffer that the coded frame will be added to.

7. **If last decode timestamp for track buffer is set and decode timestamp is less than last decode timestamp:**
   OR
   **If last decode timestamp for track buffer is set and the difference between decode timestamp and last decode timestamp is greater than 2 times last frame duration:**
   1. **If `mode` equals `"segments"`:**
      Set group end timestamp to presentation timestamp.
      **If `mode` equals `"sequence"`:**
      Set group start timestamp equal to the group end timestamp.
   2. Unset the last decode timestamp on all track buffers.
   3. Unset the last frame duration on all track buffers.
   4. Unset the highest end timestamp on all track buffers.
   5. Set the need random access point flag on all track buffers to true.
   6. Jump to the *Loop Top* step above to restart processing of the current coded frame.
8. **Otherwise:**
   Continue.
9. Let frame end timestamp equal the sum of presentation timestamp and frame duration.
10. If presentation timestamp is less than `appendWindowStart`, then set the need random access point flag to true, drop the coded frame, and jump to the top of the loop to start processing the next coded frame.
    NOTE: Some implementations *MAY* choose to collect some of these coded frames with presentation timestamp less than `appendWindowStart` and use them to generate a splice at the first coded frame that has a presentation timestamp greater than or equal to `appendWindowStart` even if that frame is not a random access point. Supporting this requires multiple decoders or faster than real-time decoding so for now this behavior will not be a normative requirement.
11. If frame end timestamp is greater than `appendWindowEnd`, then set the need random access point flag to true, drop the coded frame, and jump to the top of the loop to start processing the next coded frame.
    *NOTE*
    *Some implementations MAY choose to collect coded frames with presentation timestamp less than `appendWindowEnd` and frame end timestamp greater than `appendWindowEnd` and use them to generate a splice across the portion of the collected coded frames within the append window at time of collection, and the beginning portion of later processed frames which only partially overlap the end of the collected coded frames. Supporting this requires multiple decoders or faster than real-time decoding so for now this behavior will not be a normative requirement. In conjunction with collecting coded frames that span `appendWindowStart`, implementations MAY thus support gapless audio splicing.*
12. If the need random access point flag on track buffer equals true, then run the following steps:
    1. If the coded frame is not a random access point, then drop the coded frame and jump to the top of the loop to start processing the next coded frame.
    2. Set the need random access point flag on track buffer to false.
13. Let spliced audio frame be an unset variable for holding audio splice information
14. Let spliced timed text frame be an unset variable for holding timed text splice information
15. If last decode timestamp for track buffer is unset and presentation timestamp falls within the presentation interval of a coded frame in track buffer, then run the following steps:
    1. Let overlapped frame be the coded frame in track buffer that matches the condition above.

2. **If track buffer contains audio coded frames:**
   Run the audio splice frame algorithm and if a splice frame is returned, assign it to spliced audio frame.
   **If track buffer contains video coded frames:**
   1. Let remove window timestamp equal the overlapped frame presentation timestamp plus 1 microsecond.
   2. If the presentation timestamp is less than the remove window timestamp, then remove overlapped frame from track buffer.
      *NOTE: This is to compensate for minor errors in frame timestamp computations that can appear when converting back and forth between double precision floating point numbers and rationals. This tolerance allows a frame to replace an existing one as long as it is within 1 microsecond of the existing frame's start time. Frames that come slightly before an existing frame are handled by the removal step below.*
3. **If track buffer contains timed text coded frames:**
   Run the text splice frame algorithm and if a splice frame is returned, assign it to spliced timed text frame.

16. Remove existing coded frames in track buffer:
    **If highest end timestamp for track buffer is not set:**
    Remove all coded frames from track buffer that have a presentation timestamp greater than or equal to presentation timestamp and less than frame end timestamp.
    **If highest end timestamp for track buffer is set and less than or equal to presentation timestamp:**
    Remove all coded frames from track buffer that have a presentation timestamp greater than or equal to highest end timestamp and less than frame end timestamp
17. Remove all possible decoding dependencies on the coded frames removed in the previous two steps by removing all coded frames from track buffer between those frames removed in the previous two steps and the next random access point after those removed frames.
    *NOTE*
    *Removing all coded frames until the next random access point is a conservative estimate of the decoding dependencies since it assumes all frames between the removed frames and the next random access point depended on the frames that were removed.*
18. **If spliced audio frame is set:**
    Add spliced audio frame to the track buffer.
    **If spliced timed text frame is set:**
    Add spliced timed text frame to the track buffer.
    **Otherwise:**
    Add the coded frame with the presentation timestamp, decode timestamp, and frame duration to the track buffer.
19. Set last decode timestamp for track buffer to decode timestamp.
20. Set last frame duration for track buffer to frame duration.
21. If highest end timestamp for track buffer is unset or frame end timestamp is greater than highest end timestamp, then set highest end timestamp for track buffer to frame end timestamp.
    *NOTE*
    *The greater than check is needed because bidirectional prediction between coded frames can cause presentation timestamp to not be monotonically increasing even though the decode timestamps are monotonically increasing.*

22. If frame end timestamp is greater than group end timestamp, then set group end timestamp equal to frame end timestamp.
23. If generate timestamps flag equals true, then set `timestampOffset` equal to frame end timestamp.

2. If the `HTMLMediaElement.readyState` attribute is `HAVE_METADATA` and the new coded frames cause `HTMLMediaElement.buffered` to have a `TimeRange` for the current playback position, then set the `HTMLMediaElement.readyState` attribute to `HAVE_CURRENT_DATA`.
   *NOTE*
   *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
3. If the `HTMLMediaElement.readyState` attribute is `HAVE_CURRENT_DATA` and the new coded frames cause `HTMLMediaElement.buffered` to have a `TimeRange` that includes the current playback position and some time beyond the current playback position, then set the `HTMLMediaElement.readyState` attribute to `HAVE_FUTURE_DATA`.
   *NOTE*
   *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
4. If the `HTMLMediaElement.readyState` attribute is `HAVE_FUTURE_DATA` and the new coded frames cause `HTMLMediaElement.buffered` to have a `TimeRange` that includes the current playback position and enough data to ensure uninterrupted playback, then set the `HTMLMediaElement.readyState` attribute to `HAVE_ENOUGH_DATA`.
   *NOTE*
   *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
5. If the media segment contains data beyond the current `duration`, then run the duration change algorithm with new duration set to the maximum of the current duration and the group end timestamp.

### 3.5.9 Coded Frame Removal Algorithm

Follow these steps when coded frames for a specific time range need to be removed from the SourceBuffer:

1. Let start be the starting presentation timestamp for the removal range.
2. Let end be the end presentation timestamp for the removal range.
3. For each track buffer in this source buffer, run the following steps:
   1. Let remove end timestamp be the current value of `duration`
   2. If this track buffer has a random access point timestamp that is greater than or equal to end, then update remove end timestamp to that random access point timestamp.
      NOTE: Random access point timestamps can be different across tracks because the dependencies between coded frames within a track are usually different than the dependencies in another track.
   3. Remove all media data, from this track buffer, that contain starting timestamps greater than or equal to start and less than the remove end timestamp.
      1. For each removed frame, if the frame has a decode timestamp equal to the last decode timestamp for the frame's track, run the following steps:
         **If `mode` equals `"segments"`:**
         Set group end timestamp to presentation timestamp.
         **If `mode` equals `"sequence"`:**
         Set group start timestamp equal to the group end timestamp.

2. Unset the last decode timestamp on all track buffers.
3. Unset the last frame duration on all track buffers.
4. Unset the highest end timestamp on all track buffers.
5. Set the need random access point flag on all track buffers to true.

4. Remove all possible decoding dependencies on the coded frames removed in the previous step by removing all coded frames from this track buffer between those frames removed in the previous step and the next random access point after those removed frames.
   *NOTE*
   *Removing all coded frames until the next random access point is a conservative estimate of the decoding dependencies since it assumes all frames between the removed frames and the next random access point depended on the frames that were removed.*
5. If this object is in `activeSourceBuffers`, the current playback position is greater than or equal to start and less than the remove end timestamp, and `HTMLMediaElement.readyState` is greater than `HAVE_METADATA`, then set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA` and stall playback.
   *NOTE*
   *Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement.*
   *NOTE*
   *This transition occurs because media data for the current position has been removed. Playback cannot progress until media for the current playback position is appended or the selected/enabled tracks change.*

4. If buffer full flag equals true and this object is ready to accept more bytes, then set the buffer full flag to false.

### 3.5.10 Coded Frame Eviction Algorithm

This algorithm is run to free up space in this source buffer when new data is appended.

1. Let new data equal the data that is about to be appended to this SourceBuffer.
2. If the buffer full flag equals false, then abort these steps.
3. Let removal ranges equal a list of presentation time ranges that can be evicted from the presentation to make room for the new data.
   NOTE: Implementations *MAY* use different methods for selecting removal ranges so web applications *SHOULD NOT* depend on a specific behavior. The web application can use the `buffered` attribute to observe whether portions of the buffered data have been evicted.
4. For each range in removal ranges, run the coded frame removal algorithm with start and end equal to the removal range start and end timestamp respectively.

### 3.5.11 Audio Splice Frame Algorithm

Follow these steps when the coded frame processing algorithm needs to generate a splice frame for two overlapping audio coded frames:

1. Let track buffer be the track buffer that will contain the splice.
2. Let new coded frame be the new coded frame, that is being added to track buffer, which triggered the need for a splice.
3. Let presentation timestamp be the presentation timestamp for new coded frame

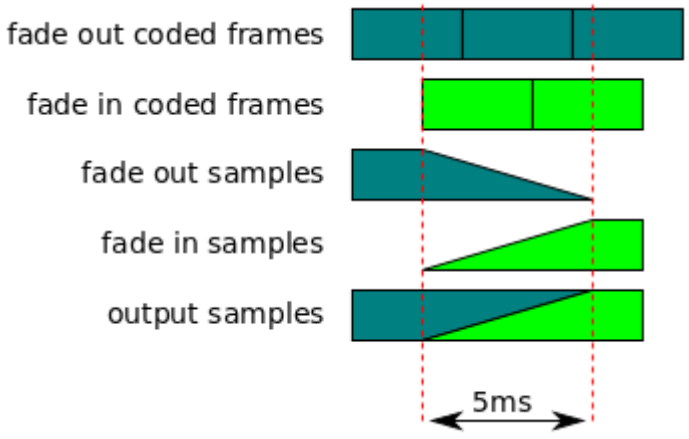
4. Let decode timestamp be the decode timestamp for new coded frame.
5. Let frame duration be the coded frame duration of new coded frame.
6. Let overlapped frame be the coded frame in track buffer with a presentation interval that contains presentation timestamp.
7. Update presentation timestamp and decode timestamp to the nearest audio sample timestamp based on sample rate of the audio in overlapped frame. If a timestamp is equidistant from both audio sample timestamps, then use the higher timestamp (e.g., `floor(x * sample_rate + 0.5) / sample_rate`).
   NOTE: For example, given the following values:
   - The presentation timestamp of overlapped frame equals 10.
   - The sample rate of overlapped frame equals 8000 Hz
   - presentation timestamp equals 10.01255
   - decode timestamp equals 10.01255
8. presentation timestamp and decode timestamp are updated to 10.0125 since 10.01255 is closer to 10 + 100/8000 (10.0125) than 10 + 101/8000 (10.012625)
9. If the user agent does not support crossfading then run the following steps:
   - Remove overlapped frame from track buffer.
   - Add a silence frame to track buffer with the following properties:
     - The presentation timestamp set to the overlapped frame presentation timestamp.
     - The decode timestamp set to the overlapped frame decode timestamp.
     - The coded frame duration set to difference between presentation timestamp and the overlapped frame presentation timestamp.
   - NOTE: Some implementations *MAY* apply fades to/from silence to coded frames on either side of the inserted silence to make the transition less jarring.
   - Return to caller without providing a splice frame.
     NOTE: This is intended to allow new coded frame to be added to the track buffer as if overlapped frame had not been in the track buffer to begin with.
10. Let frame end timestamp equal the sum of presentation timestamp and frame duration.
11. Let splice end timestamp equal the sum of presentation timestamp and the splice duration of 5 milliseconds.
12. Let fade out coded frames equal overlapped frame as well as any additional frames in track buffer that have a presentation timestamp greater than presentation timestamp and less than splice end timestamp.
13. Remove all the frames included in fade out coded frames from track buffer.
14. Return a splice frame with the following properties:
    - The presentation timestamp set to the overlapped frame presentation timestamp.
    - The decode timestamp set to the overlapped frame decode timestamp.
    - The coded frame duration set to difference between frame end timestamp and the overlapped frame presentation timestamp.
    - The fade out coded frames equals fade-out coded frames.
    - The fade in coded frame equal new coded frame.
      *NOTE: If the new coded frame is less than 5 milliseconds in duration, then coded frames that are appended after the new coded frame will be needed to properly render the splice.*
    - The splice timestamp equals presentation timestamp.
15. NOTE: See the audio splice rendering algorithm for details on how this splice frame is rendered.

### 3.5.12 Audio Splice Rendering Algorithm

The following steps are run when a spliced frame, generated by the audio splice frame algorithm, needs to be rendered by the media element:

1. Let fade out coded frames be the coded frames that are faded out during the splice.
2. Let fade in coded frames be the coded frames that are faded in during the splice.
3. Let presentation timestamp be the presentation timestamp of the first coded frame in fade out coded frames.
4. Let end timestamp be the sum of the presentation timestamp and the coded frame duration of the last frame in fade in coded frames.
5. Let splice timestamp be the presentation timestamp where the splice starts. This corresponds with the presentation timestamp of the first frame in fade in coded frames.
6. Let splice end timestamp equal splice timestamp plus five milliseconds.
7. Let fade out samples be the samples generated by decoding fade out coded frames.
8. Trim fade out samples so that it only contains samples between presentation timestamp and splice end timestamp.
9. Let fade in samples be the samples generated by decoding fade in coded frames.
10. If fade out samples and fade in samples do not have a common sample rate and channel layout, then convert fade out samples and fade in samples to a common sample rate and channel layout.
11. Let output samples be a buffer to hold the output samples.
12. Apply a linear gain fade out with a starting gain of 1 and an ending gain of 0 to the samples between splice timestamp and splice end timestamp in fade out samples.
13. Apply a linear gain fade in with a starting gain of 0 and an ending gain of 1 to the samples between splice timestamp and splice end timestamp in fade in samples.
14. Copy samples between presentation timestamp to splice timestamp from fade out samples into output samples.
15. For each sample between splice timestamp and splice end timestamp, compute the sum of a sample from fade out samples and the corresponding sample in fade in samples and store the result in output samples.
16. Copy samples between splice end timestamp to end timestamp from fade in samples into output samples.
17. Render output samples.

*NOTE: Here is a graphical representation of this algorithm.*




### 3.5.13 Text Splice Frame Algorithm

Follow these steps when the coded frame processing algorithm needs to generate a splice frame for two overlapping timed text coded frames:

1. Let track buffer be the track buffer that will contain the splice.

2. Let new coded frame be the new coded frame, that is being added to track buffer, which triggered the need for a splice.
3. Let presentation timestamp be the presentation timestamp for new coded frame
4. Let decode timestamp be the decode timestamp for new coded frame.
5. Let frame duration be the coded frame duration of new coded frame.
6. Let frame end timestamp equal the sum of presentation timestamp and frame duration.
7. Let first overlapped frame be the coded frame in track buffer with a presentation interval that contains presentation timestamp.
8. Let overlapped presentation timestamp be the presentation timestamp of the first overlapped frame.
9. Let overlapped frames equal first overlapped frame as well as any additional frames in track buffer that have a presentation timestamp greater than presentation timestamp and less than frame end timestamp.
10. Remove all the frames included in overlapped frames from track buffer.
11. Update the coded frame duration of the first overlapped frame to presentation timestamp - overlapped presentation timestamp.
12. Add first overlapped frame to the track buffer.
13. Return to caller without providing a splice frame.
    *NOTE: This is intended to allow new coded frame to be added to the track buffer as if it hadn't overlapped any frames in track buffer to begin with.*

# MSE ISO BMFF Byte Stream Format

https://w3c.github.io/mse-byte-stream-format-isobmff/

## 3. Initialization Segments

An ISO BMFF initialization segment is defined in this specification as a single File Type Box (**ftyp**) followed by a single Movie Box (**moov**).

The user agent *MUST* run the append error algorithm if any of the following conditions are met:

1. A File Type Box contains a *major_brand* or *compatible_brand* that the user agent does not support.
2. A box or field in the Movie Box is encountered that violates the requirements mandated by the *major_brand* or one of the *compatible_brands* in the File Type Box.
3. The tracks in the Movie Box contain samples (i.e., the *entry_count* in the **stts**, **stsc** or **stco** boxes are not set to zero).
4. A Movie Extends (**mvex**) box is not contained in the Movie (**moov**) box to indicate that Movie Fragments are to be expected.

The user agent *MUST* support setting the offset from media composition time to movie presentation time by handling an Edit Box (**edts**) containing a single Edit List Box (**elst**) that contains a single edit with media rate one. This edit *MAY* have a duration of 0 (indicating that it spans all subsequent media) or *MAY* have a non-zero duration (indicating the total duration of the movie including fragments).

The user agent *MUST* support codec configurations stored out-of-band in the sample entry, and for codecs which allow codec configurations stored inband in the samples themselves, the user agent *SHOULD* support codec configurations stored inband.

NOTE: For example, for codecs which include SPS and PPS parameter sets, for maximum content interoperability, user agents are strongly advised to support both inband (e.g., as defined for avc3/avc4) and out-of-band (e.g., as defined for avc1/2) storage of the SPS and PPS.

Valid top-level boxes such as **pdin**, **free**, and **sidx** are allowed to appear before the **moov** box. These boxes *MUST* be accepted and ignored by the user agent and are not considered part of the initialization segment in this specification.

The user agent *MUST* source attribute values for id, kind, label and language for `AudioTrack`, `VideoTrack` and `TextTrack` objects as described for MPEG-4 ISOBMFF in the in-band tracks spec [INBANDTRACKS].

# 4. Media Segments

An ISO BMFF media segment is defined in this specification as one optional Segment Type Box (**styp**), followed by zero or more Event Message Boxes (**emsg**), followed by a single Movie Fragment Box (**moof**), followed by one or more Media Data Boxes (**mdat**). If the Segment Type Box is not present, the segment *MUST* conform to the brands listed in the File Type Box (**ftyp**) in the initialization segment.

Valid top-level boxes other than **ftyp**, **moov**, **styp**, **emsg**, **moof**, and **mdat** are allowed to appear between the end of an initialization segment or media segment and before the beginning of a new media segment. These boxes *MUST* be accepted and ignored by the user agent and are not considered part of the media segment in this specification.

Note: **ftyp**, **moov**, **styp**, **moof**, and **mdat** are defined ISO/IEC 14496-12. **emsg** is defined in [MPEG-DASH].

The user agent *MUST* run the append error algorithm if any of the following conditions are met:

1. A box or field in the Movie Fragment Box is encountered that violates the requirements mandated by the *major_brand* or one of the *compatible_brands* in the Segment Type Box in this media segment or the File Type Box in the initialization segment if a Segment Type Box is not present.
2. This media segment contains a Segment Type Box that is not compatible with the File Type Box in the initialization segment.
3. The Movie Fragment Box does not contain at least one Track Fragment Box (**traf**).
4. The Movie Fragment Box does not use movie-fragment relative addressing.
5. External data references are being used.
6. At least one Track Fragment Box does not contain a Track Fragment Decode Time Box (**tfdt**)
7. The Media Data Boxes do not contain all the samples referenced by the Track Fragment Run Boxes (**trun**) of the Movie Fragment Box.
8. Inband parameter sets are not present in the appropriate samples and parameter sets are not present in the last initialization segment appended.

A Movie Fragment Box uses movie-fragment relative addressing when the first Track Fragment Run(**trun**) box in each Track Fragment Box has the *data-offset-present* flag set and either of the following conditions are met:

- Every Track Fragment Box in a Movie Fragment Box has the *default-base-is-moof* flag set.
  *NOTE*
  *This implies that the base-data-offset-present flag is not set.*
- The Movie Fragment Box contains a single Track Fragment Box and that box does not have the *base-data-offset-present* flag set.

## 5. Random Access Points

A random access point as defined in this specification corresponds to a Stream Access Point of type 1 or 2 as defined in Annex I of ISO/IEC 14496-12.

# Sourcing In-band Media Resource Tracks from Media Containers into HTML

https://dev.w3.org/html5/html-sourcing-inband-tracks/

## 2. MPEG-DASH

**MIME type/subtype: application/dash+xml**

[MPEGDASH] defines formats for a media manifest, called MPD (Media Presentation Description), which references media containers, called media segments. [MPEGDASH] also defines some media segments formats based on [MPEG2TS] or [ISOBMFF]. Processing of media manifests and segments to expose tracks to Web applications can be done by the user agent. Alternatively, a web application can process the manifests and segments to expose tracks. When the user agent processes MPD and media segments directly, it exposes tracks for AdaptationSet and ContentComponent elements, as defined in this document. When the Web application processes the MPD and media segments, it passes media segments to the user agent according to the MediaSource Extension [MSE] specification. In this case, the tracks are exposed by the user agent according to [MSE]. The Web application may set default track attributes from MPD data, using the trackDefaults object, that will be used by the user agent to set attributes not set from initialization segment data.

1. Track Order
   If an AdaptationSet contains ContentComponents, a track is created for each ContentComponent. Otherwise, a track is created for the AdaptationSet itself. The order of tracks specified in the MPD (Media Presentation Description) format [MPEGDASH] is maintained when sourcing multiple MPEG DASH tracks into HTML.
2. Determining the type of track
   A user agent recognises and supports data from a MPEG DASH media resource as being equivalent to a HTML track using the content type given by the MPD. The content type of the track is the first present value out of: The ContentComponents's "contentType" attribute, the AdaptationSet's "contentType" attribute, or the main type in the AdaptationSet's "mimeType" attribute (i.e. for "video/mp2t", the main type is "video").

- text track:
  - the content type is "application" or "text"
  - the content type is "video" and the AdaptationSet contains one or more ISOBMFF CEA 608 or 708 caption services.
- video track: the content type is "video"
- audio track: the content type is "audio"

3. Track Attributes for sourced Text Tracks

Data for sourcing text track attributes may exist in the media content or in the MPD. Text track attribute values are first sourced from track data in the media container, as described for text track attributes in MPEG-2 Transport Streams and text track attributes in MPEG-4 ISOBMFF. If a track attribute's value cannot be determined from the media container, then the track attribute value is sourced from data in the track's ContentComponent. If the needed attribute or element does not exist on the ContentComponent (or if the AdaptationSet doesn't contain any ContentComponents), then that attribute or element is sourced from the AdaptationSet:

| Attribute | How to source its value |
|---|---|
| **id** | The track is:<br>• An ISOBMFF CEA 608 caption service: the string "cc" concatenated with the value of the 'channel-number' field in the Accessibility descriptor in the ContentComponent or AdaptationSet.<br>• An ISOBMFF CEA 708 caption service: the string "sn" concatenated with the value of the 'service-number' field in the Accessibility descriptor in the ContentComponent or AdaptationSet.<br>• Otherwise, the content of the 'id' attribute in the ContentComponent, or AdaptationSet. |
| **kind** | The track:<br>• Represents a ContentComponent or AdaptationSet containing a Role descriptor with schemeIdURI attribute = "urn:mpeg:dash:role:2011":<br>  ○ "captions": if the Role descriptor's value is "caption"<br>  ○ "subtitles": if the Role descriptor's value is "subtitle"<br>  ○ "metadata": otherwise<br>• Is an ISOBMFF CEA 608 or 708 caption service: "captions". |
| **label** | The empty string. |

| | |
|---|---|
| **language** | The track is:<br>• An ISOBMFF CEA 608 708 caption service: the value of the 'language' field in the Accessibility descriptor, in the ContentComponent or AdaptationSet, where the corresponding 'channel-number' or 'service-number' is the same as this track's 'id' attribute. The empty string if there is no such corresponding 'channel-number' or 'service-number'.<br>• Otherwise: the content of the 'lang' attribute in the ContentComponent or AdaptationSet element. |
| **inBandMetadataTrackDispatchType** | If kind is "metadata", an XML document containing the AdaptationSet element and all child Role descriptors and ContentComponents, and their child Role descriptors. The empty string otherwise. |
| **mode** | "disabled" |

4.

Track Attributes for sourced Audio and Video Tracks

Data for sourcing audio and video track attributes may exist in the media content or in the MPD. Audio and video track attribute values are first sourced from track data in the media container, as described for audio and video track attributes in MPEG-2 Transport Streams and audio and video track attributes in MPEG-4 ISOBMFF. If a track attribute's value cannot be determined from the media container, then the track attribute value is sourced from data in the track's ContentComponent. If the needed attribute or element does not exist on the ContentComponent (or if the AdaptationSet doesn't contain any ContentComponents), then that attribute or element is sourced from the AdaptationSet:

| Attribute | How to source its value |
|---|---|
| **id** | Content of the id attribute in the ContentComponent or AdaptationSet element. Empty string if the id attribute is not present on either element. |
| **kind** | Given a Role scheme of "urn:mpeg:dash:role:2011", determine the kind attribute from the value of the Role descriptors in the ContentComponent **and** AdaptationSet elements.<br>• "alternative": if the role is "alternate" but not also "main" or "commentary", or "dub"<br>• "captions": if the role is "caption" and also "main"<br>• "descriptions": if the role is "description" and also "supplementary"<br>• "main": if the role is "main" but not also "caption", "subtitle", or "dub"<br>• "main-desc": if the role is "main" and also "description"<br>• "sign": not used |

| | |
|---|---|
| | • "subtitles": if the role is "subtitle" and also "main"<br>• "translation": if the role is "dub" and also "main"<br>• "commentary": if the role is "commentary" but not also "main"<br>• "": otherwise |
| **label** | The empty string. |
| **language** | Content of the lang attribute in the ContentComponent or AdaptationSet element. |

5.
Mapping Text Track content into text track cues
TextTrackCue objects may be sourced from DASH media content in the WebVTT, TTML, MPEG-2 TS or ISOBMFF format.
Media content with the MIME type "text/vtt" is in the WebVTT format and should be exposed as a VTTCue object as defined in [WEBVTT].
Media content with the MIME type "application/ttml+xml" is in the TTML format and should be exposed as an as yet to be defined TTMLCue object. Alternatively, browsers can also map the TTML features to VTTCue objects [WEBVTT]. Finally, browsers that cannot render TTML [ttaf1-dfxp] format data should expose them as DataCue objects [HTML51]. In this case, the TTML file must be parsed in its entirety and then converted into a sequence of TTML Intermediate Synchronic Documents (ISDs). Each ISD creates a DataCue object with attributes sourced as follows:

| Attribute | How to source its value |
|---|---|
| **id** | Decimal representation of the id attribute of the head element in the XML document. Null if there is no id attribute. |
| **startTime** | Value of the beginning media time of the active temporal interval of the ISD. |
| **endTime** | Value of the ending media time of the active temporal interval of the ISD. |
| **pauseOnExit** | "false" |
| **data** | The (UTF-16 encoded) ArrayBuffer composing the ISD resource. |

6.
Media content with the MIME type "application/mp4" or "video/mp4" is in the [ISOBMFF] format and should be exposed following the same rules as for ISOBMFF text track.
Media content with the MIME type "video/mp2t" is in the MPEG-2 TS format and should be exposed following the same rules as for MPEG-2 TS text track.

# 4. MPEG-4 ISOBMFF

**MIME type/subtype: audio/mp4, video/mp4, application/mp4**

1. Track Order
   The order of tracks specified by TrackBox (trak) boxes in the MovieBox (moov) container [ISOBMFF] is maintained when sourcing multiple MPEG-4 tracks into HTML.
2. Determining the type of track
   A user agent recognises and supports data from a TrackBox as being equivalent to a HTML track

based on the value of the handler_type field in the HandlerBox (hdlr) of the MediaBox (mdia) of the TrackBox:

- text track:
  - the handler_type value is "meta", "subt" or "text"
  - the handler_type value is "vide" and an **ISOBMFF CEA 608 or 708 caption service** is encapsulated in the video track as an SEI message as defined in [DASHIFIOP].
- video track: the handler_type value is "vide"
- audio track: the handler_type value is "soun"

3. Track Attributes for sourced Text Tracks

| Attribute | How to source its value |
|---|---|
| **id** | For ISOBMFF CEA 608 closed captions, the string "cc" concatenated with the decimal representation of the channel_number.<br><br>For ISOBMFF CEA 708 closed captions, the string "sn" concatenated with the decimal representation of the service_number field in the 'Caption Channel Service Block'.<br><br>Otherwise, the decimal representation of the track_ID of a TrackHeaderBox (tkhd) in a TrackBox (trak). |

| | |
|---|---|
| **kind** | • "captions":<br>  ○ **WebVTT caption**: handler_type is "text" and SampleEntry format is WVTTSampleEntry [ISO14496-30] and the VTT metadata header Kind is "captions"<br>  ○ **SMPTE-TT caption**: handler_type is "subt" and SampleEntry format is XMLSubtitleSampleEntry [ISO14496-30] and the namespace is set to "http://www.smpte-ra.org/schemas/2052-1/2013/smpte-tt#cea708" [SMPTE2052-11].<br>  ○ An ISOBMFF CEA 608 or 708 caption service.<br>  ○ **3GPP caption**: handler_type is "text" and the SampleEntry code (format field) is "tx3g".<br>NOTE<br>Are all sample entries of this type "captions"?<br>• "subtitles":<br>  ○ **WebVTT subtitle**: handler_type is "text" and SampleEntry format is WVTTSampleEntry [ISO14496-30] and the VTT metadata header Kind is "subtitles"<br>  ○ **SMPTE-TT subtitle**: handler_type is "subt" and SampleEntry format is XMLSubtitleSampleEntry [ISO14496-30] and the namespace is set to a TTML namespace that does not indicate a SMPTE-TT caption.<br>•<br>• "metadata": otherwise |
| **label** | Content of the name field in the HandlerBox. |

| | |
|---|---|
| **language** | If the track is an ISOBMFF CEA 608 or 708 caption service then the empty string (""). <br><br> Otherwise, the content of the language field in the MediaHeaderBox. <br><br> NOTE <br><br> No signaling is currently defined for specifying the langaugae of CEA 608 or 708 captions in ISOBMFF. MPEG DASH MPDs may specify caption track metadata, including language [DASHIFIOP]. The user agent should set the language attribute of CEA 608 or 708 caption text tracks to the empty string so that script may use the media source extensions [MSE] TrackDefault object to provide a default for the language attribute. |
| **inBandMetadataTrackDispatchType** | • kind is "metadata":<br>  ○ if a XMLMetaDataSampleEntry box is present the concatenation of the string "metx", a U+0020 SPACE character, and the value of the namespace field<br>  ○ if a TextMetaDataSampleEntry box is present the concatenation of the string "mett", a U+0020 SPACE character, and the value of the mime_format field<br>  ○ otherwise the empty string<br>• otherwise the empty string |
| **mode** | "disabled" |

4.

Track Attributes for sourced Audio and Video Tracks

| Attribute | How to source its value |
|---|---|
| **id** | Decimal representation of the track_ID of a TrackHeaderBox (tkhd) in a TrackBox (trak). |

| | |
|---|---|
| **kind** | • "alternative": not used<br>• "captions": not used<br>• "descriptions"<br>  ○ For E-AC-3 audio [ETSI102366] if the bsmod field is 2 and the asvc is 1 in the EC3SpecificBox<br>• "main": first audio (video) track<br>• "main-desc<br>  ○ For AC-3 audio [ETSI102366] if the bsmod field is 2 in the AC3SpecificBox<br>  ○ For E-AC-3 audio [ETSI102366] if the bsmod field is 2 and the asvc is 0 in the EC3SpecificBox<br>• "sign": not used<br>• "subtitles": not used<br>• "translation": not first audio (video) track<br>• "commentary": not used<br>• "": otherwise |
| **label** | Content of the name field in the HandlerBox. |
| **language** | Content of the language field in the MediaHeaderBox. |

5.

**Mapping Text Track content into text track cues for MPEG-4 ISOBMFF**

[ISOBMFF] text tracks may be in the WebVTT or TTML format [ISO14496-30], 3GPP Timed Text format [3GPP-TT], or other format.

[ISOBMFF] text tracks carry WebVTT data if the media handler type is "text" and a WVTTSampleEntry format is used, as described in [ISO14496-30]. Browsers that can render text tracks in the WebVTT format should expose a VTTCue [WEBVTT] as follows:

| Attribute | How to source its value |
|---|---|
| **id** | The cue_id field in the CueIDBox. |
| **startTime** | The sample presentation time. |
| **endTime** | The sum of the startTime and the sample duration. |
| **pauseOnExit** | "false" |
| **cue setting attributes** | The settings field in the CueSettingsBox. |
| **text** | The cue_text field in the CuePayloadBox. |

6.

ISOBMFF text tracks carry TTML data if the media handler type is "subt" and an XMLSubtileSampleEntry format is used with a TTML-based name_space field, as described in [ISO14496-30]. Browsers that can render text tracks in the TTML format should expose an as yet to be defined TTMLCue. Alternatively, browsers can also map the TTML features to VTTCue objects. Finally, browsers that cannot render TTML [ttaf1-dfxp] format data should expose them as DataCue objects [HTML51]. Each TTML subtitle sample consists of an XML document and creates a DataCue object with attributes sourced as follows:

| Attribute | How to source its value |
|---|---|
| **id** | Decimal representation of the id attribute of the head element in the XML document. Null if there is no id attribute. |
| **startTime** | Value of the beginning media time of the top-level temporal interval of the XML document. |

| | |
|---|---|
| **endTime** | Value of the ending media time of the top-level temporal interval of the XML document. |
| **pauseOnExit** | "false" |
| **data** | The (UTF-16 encoded) ArrayBuffer composing the XML document. |

7. TTML data may contain tunneled CEA708 captions [SMPTE2052-11]. Browsers that can render CEA708 data should expose it as defined for MPEG-2 TS CEA708 cues.
   3GPP timed text data is carried in [ISOBMFF] as described in [3GPP-TT]. Browsers that can render text tracks in the 3GPP Timed Text format should expose an as yet to be defined 3GPPCue. Alternatively, browsers can also map the 3GPP features to VTTCue objects.

8. **Mapping Event message content into DataCue for MPEG-4 ISOBMFF**

MPEG-DASH defines the EventMessageBox as an extension of [ISOBMFF].

***TODO: Define where the emsg box can appear in the ISOBMFF***

| Attribute | How to source its value |
|---|---|
| **id** | Decimal representation of the id attribute of the DASHEventMessageBox. |
| **startTime** | Value of the starting media time of the event message<br><br>For DASHEventMessageBox version 0:<br><br>***TODO***<br><br>For DASHEventMessageBox version 1:<br><br>***TODO*** |
| **endTime** | Value of the ending media time of the event message<br><br>For DASHEventMessageBox version 0:<br><br>***TODO***<br><br>For DASHEventMessageBox version 1:<br><br>***TODO*** |
| **pauseOnExit** | "false" |
| **data** | An Object containing data from the message_data and value DASHEventMessageBox fields: |

| | | |
|---|---|---|
| | data | An ArrayBuffer containing the message_data bytes from the DASHEventMessageBox. |
| | value | A DOMString containing the value from the DASHEventMessageBox. |
| **type** | The scheme_id_uri value from the DASHEventMessageBox. | |

# HTML

https://html.spec.whatwg.org/multipage/media.html#sourcing-in-band-text-tracks

#### *4.8.12.11.2 Sourcing in-band text tracks*

A media-resource-specific text track is a text track that corresponds to data found in the media resource.

Rules for processing and rendering such data are defined by the relevant specifications, e.g. the specification of the video format if the media resource is a video. Details for some legacy formats can be found in Sourcing In-band Media Resource Tracks from Media Containers into HTML. [INBAND]

When a media resource contains data that the user agent recognizes and supports as being equivalent to a text track, the user agent runs the steps to expose a media-resource-specific text track with the relevant data, as follows.

1. Associate the relevant data with a new text track and its corresponding new `TextTrack` object. The text track is a media-resource-specific text track.
2. Set the new text track's kind, label, and language based on the semantics of the relevant data, as defined by the relevant specification. If there is no label in that data, then the label must be set to the empty string.
3. Associate the text track list of cues with the rules for updating the text track rendering appropriate for the format in question.
4. If the new text track's kind is `chapters` or `metadata`, then set the text track in-band metadata track dispatch type as follows, based on the type of the media resource:
   If the media resource is an Ogg file
   The text track in-band metadata track dispatch type must be set to the value of the Name header field. [OGGSKELETONHEADERS]
   If the media resource is a WebM file
   The text track in-band metadata track dispatch type must be set to the value of the `CodecID` element. [WEBMCG]
   If the media resource is an MPEG-2 file
   Let stream type be the value of the "stream_type" field describing the text track's type in

the file's program map section, interpreted as an 8-bit unsigned integer. Let length be the value of the "ES_info_length" field for the track in the same part of the program map section, interpreted as an integer as defined by Generic coding of moving pictures and associated audio information. Let descriptor bytes be the length bytes following the "ES_info_length" field. The text track in-band metadata track dispatch type must be set to the concatenation of the stream type byte and the zero or more descriptor bytes bytes, expressed in hexadecimal using ASCII upper hex digits. [MPEG2]

If the media resource is an MPEG-4 file

Let the first `stsd` box of the first `stbl` box of the first `minf` box of the first `mdia` box of the text track's `trak` box in the first `moov` box of the file be the *stsd box*, if any. If the file has no *stsd box*, or if the *stsd box* has neither a `mett` box nor a `metx` box, then the text track in-band metadata track dispatch type must be set to the empty string. Otherwise, if the *stsd box* has a `mett` box then the text track in-band metadata track dispatch type must be set to the concatenation of the string "`mett`", a U+0020 SPACE character, and the value of the first `mime_format` field of the first `mett` box of the *stsd box*, or the empty string if that field is absent in that box. Otherwise, if the *stsd box* has no `mett` box but has a `metx` box then the text track in-band metadata track dispatch type must be set to the concatenation of the string "`metx`", a U+0020 SPACE character, and the value of the first `namespace` field of the first `metx` box of the *stsd box*, or the empty string if that field is absent in that box. [MPEG4]

5. Populate the new text track's list of cues with the cues parsed so far, following the guidelines for exposing cues, and begin updating it dynamically as necessary.
6. Set the new text track's readiness state to loaded.
7. Set the new text track's mode to the mode consistent with the user's preferences and the requirements of the relevant specification for the data.

   *For instance, if there are no other active subtitles, and this is a forced subtitle track (a subtitle track giving subtitles in the audio track's primary language, but only for audio that is actually in another language), then those subtitles might be activated here.*
8. Add the new text track to the media element's list of text tracks.
9. Fire an event named `addtrack` at the media element's `textTracks` attribute's `TextTrackList` object, using `TrackEvent`, with the `track` attribute initialized to the text track's `TextTrack` object.

#### *4.8.12.11.3 Sourcing out-of-band text tracks*

When a `track` element is created, it must be associated with a new text track (with its value set as defined below) and its corresponding new `TextTrack` object.