Media Source Extensions

W3C Editor's Draft 14 November 2013

This version:
http://dvcs.w3.org/hg/html-media/raw-file/default/media-source/media-source.html
Latest published version:
http://www.w3.org/TR/media-source/
Latest editor's draft:
http://dvcs.w3.org/hg/html-media/raw-file/default/media-source/media-source.html

Editors:
Aaron Colwell, Google Inc.
Adrian Bateman, Microsoft Corporation
Mark Watson, Netflix Inc.

Abstract

This specification extends HTMLMediaElement to allow JavaScript to generate media streams for playback. Allowing JavaScript to generate streams facilitates a variety of use cases like adaptive streaming and time shifting live streams.

If you wish to make comments or file bugs regarding this document in a manner that is tracked by the W3C, please submit them via our public bug database.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

The working groups maintains a list of all bug reports that the editors have not yet tried to address. This draft highlights some of the pending issues that are still to be discussed in the working group. No decision has been taken on the outcome of these issues including whether they are valid.

Implementors should be aware that this specification is not stable. **Implementors who are not taking part in the discussions are likely to find the specification changing out from under them in incompatible ways.** Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation stage should join the mailing list mentioned below and take part in the discussions.

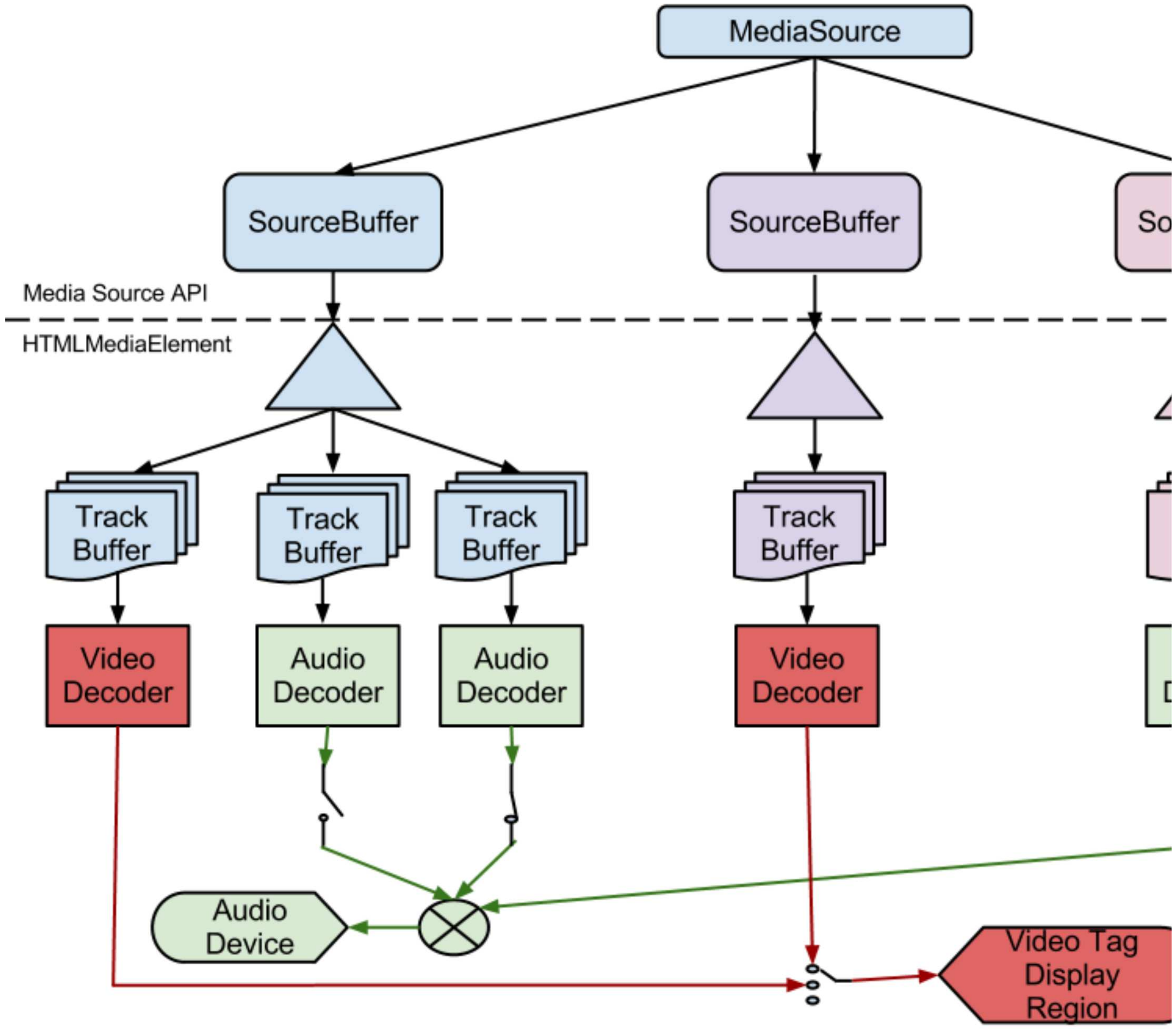
This document was published by the HTML Working Group as an Editor's Draft. If you wish to make comments regarding this document, please send them to public-html-media@w3.org (subscribe, archives). All comments are welcome.

Publication as an Editor's Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Table of Contents

1. Introduction

This specification allows JavaScript to dynamically construct media streams for <audio> and <video>. It defines objects that allow JavaScript to pass media segments to an HTMLMediaElement [HTML5]. A buffering model is also included to describe how the user agent acts when different media segments are appended at different times. Byte stream specifications used with these extensions are available in the byte stream format registry.



1.1 Goals

This specification was designed with the following goals in mind:

- Allow JavaScript to construct media streams independent of how the media is fetched.
- Define a splicing and buffering model that facilitates use cases like adaptive streaming, ad-insertion, time-shifting, and video editing.
- Minimize the need for media parsing in JavaScript.
- Leverage the browser cache as much as possible.
- Provide requirements for byte stream format specifications.
- Not require support for any particular media format or codec.

This specification defines:

- Normative behavior for user agents to enable interoperability between user agents and web applications when processing media data.
- Normative requirements to enable other specifications to define media formats to be used within this specification.

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 100 milliseconds.

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 end of stream algorithm with the *error* parameter set to `"decode"` 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.

Displayed Frame Delay

The delay, to the nearest microsecond, between a frame's presentation time and the actual time it was displayed. This delay is always greater than or equal to zero since frames must never be displayed before their presentation time. Non-zero delays are a sign of playback jitter and possible loss of A/V sync.

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 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 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 specified by the Origin of Blob URIs [FILE-API].

> 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.

Presentation 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 must 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.

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 end of stream algorithm with the *error* parameter set to `"decode"` 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

The MediaSource object represents a source of media data for an HTMLMediaElement. It keeps track of the `readyState` for this source as well as a list of `SourceBuffer` objects that can be used to add media data to the presentation. MediaSource objects are created by the web application and then attached to an HTMLMediaElement. The application uses the `SourceBuffer` objects in `sourceBuffers` to add media data to this source. The HTMLMediaElement fetches this media data from the `MediaSource` object when it is needed during playback.

WebIDL

```
enum ReadyState {
    "closed",
    "open",
    "ended"
};
```

Enumeration description

`closed`	Indicates the source is not currently attached to a media element.
`open`	The source has been opened by a media element and is ready for data to be appended to the `SourceBuffer` objects in `sourceBuffers`.
`ended`	The source is still attached to a media element, but `endOfStream()` has been called.

WebIDL

```
enum EndOfStreamError {
    "network",
    "decode"
};
```

Enumeration description

`network`	Terminates playback and signals that a network error has occured. NOTE JavaScript applications should use this status code to terminate playback with a network error. For example, if a network error occurs while fetching media data.
`decode`	Terminates playback and signals that a decoding error has occured. NOTE JavaScript applications should use this status code to terminate playback with a decode error. For example, if a parsing error occurs while processing out-of-band media data.

WebIDL

```
[Constructor]
interface MediaSource : EventTarget {
    readonly    attribute SourceBufferList    sourceBuffers;
    readonly    attribute SourceBufferList    activeSourceBuffers;
    readonly    attribute ReadyState          readyState;
                attribute unrestricted double duration;
    SourceBuffer   addSourceBuffer (DOMString type);
    void           removeSourceBuffer (SourceBuffer sourceBuffer);
    void           endOfStream (optional EndOfStreamError error);
    static boolean isTypeSupported (DOMString type);
};
```

2.1 Attributes

`activeSourceBuffers` of type *SourceBufferList*, readonly

Contains the subset of `sourceBuffers` that are providing the selected video track, the enabled audio tracks, and the "showing" or "hidden" text tracks.

> NOTE
>
> The Changes to selected/enabled track state section describes how this attribute gets updated.

`duration` of type ***unrestricted double***,

Allows the web application to set the presentation duration. The duration is initially set to NaN when the **`MediaSource`** object is created.

On getting, run the following steps:

1. If the `readyState` attribute is `"closed"` then return NaN and abort these steps.
2. Return the current value of the attribute.

On setting, run the following steps:

1. If the value being set is negative or NaN then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
2. If the `readyState` attribute is not `"open"` then throw an `INVALID_STATE_ERR` exception and abort these steps.
3. If the `updating` attribute equals true on any **`SourceBuffer`** in `sourceBuffers`, then throw an `INVALID_STATE_ERR` exception and abort these steps.
4. Run the duration change algorithm with *new duration* set to the value being assigned to this attribute.

> NOTE
>
> `appendBuffer()`, `appendStream()` and `endOfStream()` can update the duration under certain circumstances.

`readyState` of type *ReadyState*, readonly

Indicates the current state of the **`MediaSource`** object. When the **`MediaSource`** is created `readyState` must be set to `"closed"`.

`sourceBuffers` of type *SourceBufferList*, readonly

Contains the list of **`SourceBuffer`** objects associated with this **`MediaSource`**. When `readyState` equals `"closed"` this list will be empty. Once `readyState` transitions to `"open"` SourceBuffer objects can be added to this list by using `addSourceBuffer()`.

2.2 Methods

`addSourceBuffer`

Adds a new `SourceBuffer` to `sourceBuffers`.

Implementations must support at least 1 MediaSource object with the following `SourceBuffer` configurations. MediaSource objects must support each of the configurations below, 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.

- 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.

Parameter	Type	Nullable	Optional	Description
type	`DOMString`	✗	✗	

Return type: `SourceBuffer`

When this method is invoked, the user agent must run the following steps:

1. If *type* is null or an empty string then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
2. If *type* contains a MIME type that is not supported or contains a MIME type that is not supported with the types specified for the other `SourceBuffer` objects in `sourceBuffers`, then throw a `NOT_SUPPORTED_ERR` exception and abort these steps.
3. If the user agent can't handle any more SourceBuffer objects then throw a `QUOTA_EXCEEDED_ERR` exception and abort these steps.

   NOTE

   For example, a user agent may throw a `QUOTA_EXCEEDED_ERR` exception if the media element has reached the `HAVE_METADATA` readyState. This can occur if the user agent's media engine does not support adding more tracks during playback.

4. If the `readyState` attribute is not in the `"open"` state then throw an `INVALID_STATE_ERR` exception and abort these steps.
5. Create a new `SourceBuffer` object and associated resources.
6. Add the new object to `sourceBuffers` and queue a task to fire a simple event named `addsourcebuffer` at `sourceBuffers`.
7. Return the new object.

`endOfStream`

Signals the end of the stream.

Parameter	Type	Nullable	Optional	Description
error	`EndOfStreamError`	✗	✓	

Return type: `void`

When this method is invoked, the user agent must run the following steps:

1. If the `readyState` attribute is not in the `"open"` state then throw an `INVALID_STATE_ERR` exception and abort these steps.
2. If the `updating` attribute equals true on any `SourceBuffer` in `sourceBuffers`, then throw an `INVALID_STATE_ERR` exception and abort these steps.
3. Run the end of stream algorithm with the *error* parameter set to *error*.

`isTypeSupported`, static

Check to see whether the `MediaSource` is capable of creating `SourceBuffer` objects for the the specified MIME type.

NOTE

If true is returned from this method, it only indicates that the `MediaSource` implementation is capable of creating `SourceBuffer` objects for the specified MIME type. An `addSourceBuffer()` call may still fail if sufficient resources are not available to support the addition of a new `SourceBuffer`.

NOTE

This method returning true implies that HTMLMediaElement.canPlayType() will return "maybe" or "probably" since it does not make sense for a `MediaSource` to support a type the HTMLMediaElement knows it cannot play.

Parameter	Type	Nullable	Optional	Description
type	`DOMString`	✗	✗	

Return type: `boolean`

When this method is invoked, the user agent must run the following steps:

1. If *type* is an empty string, then return false.
2. If *type* does not contain a valid MIME type string, then return false.
3. If *type* contains a media type or media subtype that the MediaSource does not support, then return false.
4. If *type* contains a codec that the MediaSource does not support, then return false.
5. If the MediaSource does not support the specified combination of media type, media subtype, and codecs then return false.
6. Return true.

`removeSourceBuffer`

Removes a `SourceBuffer` from `sourceBuffers`.

Parameter	Type	Nullable	Optional	Description
sourceBuffer	`SourceBuffer`	✗	✗	

Return type: **`void`**

When this method is invoked, the user agent must run the following steps:

1. If *sourceBuffer* is null then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
2. If *sourceBuffer* specifies an object that is not in `sourceBuffers` then throw a `NOT_FOUND_ERR` exception and abort these steps.
3. If the *sourceBuffer*.`updating` attribute equals true, then run the following steps:
    1. Abort the buffer append and stream append loop algorithms if they are running.
    2. Set the *sourceBuffer*.`updating` attribute to false.
    3. Queue a task to fire a simple event named `abort` at *sourceBuffer*.
    4. Queue a task to fire a simple event named `updateend` at *sourceBuffer*.
4. Let *SourceBuffer audioTracks list* equal the `AudioTrackList` object returned by *sourceBuffer*.`audioTracks`.
5. If the *SourceBuffer audioTracks list* is not empty, then run the following steps:
    1. Let *HTMLMediaElement audioTracks list* equal the `AudioTrackList` object returned by the `audioTracks` attribute on the HTMLMediaElement.
    2. Let the *removed enabled audio track flag* equal false.
    3. For each `AudioTrack` object in the *SourceBuffer audioTracks list*, run the following steps:
        1. Set the `sourceBuffer` attribute on the `AudioTrack` object to null.
        2. If the `enabled` attribute on the `AudioTrack` object is true, then set the *removed enabled audio track flag* to true.
        3. Remove the `AudioTrack` object from the *HTMLMediaElement audioTracks list*.
        4. Queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the *HTMLMediaElement audioTracks list*.
        5. Remove the `AudioTrack` object from the *SourceBuffer audioTracks list*.
        6. Queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the *SourceBuffer audioTracks list*.
    4. If the *removed enabled audio track flag* equals true, then queue a task to fire a simple event named `change` at the *HTMLMediaElement audioTracks list*.
6. Let *SourceBuffer videoTracks list* equal the `VideoTrackList` object returned by *sourceBuffer*.`videoTracks`.
7. If the *SourceBuffer videoTracks list* is not empty, then run the following steps:

    1. Let *HTMLMediaElement videoTracks list* equal the `VideoTrackList` object returned by the `videoTracks` attribute on the HTMLMediaElement.
    2. Let the *removed selected video track flag* equal false.
    3. For each `VideoTrack` object in the *SourceBuffer videoTracks list*, run the following steps:
        1. Set the `sourceBuffer` attribute on the `VideoTrack` object to null.
        2. If the `selected` attribute on the `VideoTrack` object is true, then set the *removed selected video track flag* to true.
        3. Remove the `VideoTrack` object from the *HTMLMediaElement videoTracks list*.
        4. Queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the *HTMLMediaElement videoTracks list*.
        5. Remove the `VideoTrack` object from the *SourceBuffer videoTracks list*.
        6. Queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the *SourceBuffer videoTracks list*.
    4. If the *removed selected video track flag* equals true, then queue a task to fire a simple event named `change` at the *HTMLMediaElement videoTracks list*.
8. Let *SourceBuffer textTracks list* equal the `TextTrackList` object returned by *sourceBuffer*.`textTracks`.
9. If the *SourceBuffer textTracks list* is not empty, then run the following steps:
    1. Let *HTMLMediaElement textTracks list* equal the `TextTrackList` object returned by the `textTracks` attribute on the HTMLMediaElement.
    2. Let the *removed enabled text track flag* equal false.
    3. For each `TextTrack` object in the *SourceBuffer textTracks list*, run the following steps:
        1. Set the `sourceBuffer` attribute on the `TextTrack` object to null.
        2. If the `mode` attribute on the `TextTrack` object is set to `"showing"` or `"hidden"` , then set the *removed enabled text track flag* to true.
        3. Remove the `TextTrack` object from the *HTMLMediaElement textTracks list*.
        4. Queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the *HTMLMediaElement textTracks list*.
        5. Remove the `TextTrack` object from the *SourceBuffer textTracks list*.
        6. Queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the *SourceBuffer textTracks list*.

4. If the *removed enabled text track flag* equals true, then queue a task to fire a simple event named `change` at the *HTMLMediaElement textTracks list*.

10. If *sourceBuffer* is in `activeSourceBuffers`, then remove *sourceBuffer* from `activeSourceBuffers` and queue a task to fire a simple event named `removesourcebuffer` at the **`SourceBufferList`** returned by `activeSourceBuffers`.
11. Remove *sourceBuffer* from `sourceBuffers` and queue a task to fire a simple event named `removesourcebuffer` at the **`SourceBufferList`** returned by `sourceBuffers`.
12. Destroy all resources for *sourceBuffer*.

2.3 Event Summary

Event name	Interface	Dispatched when...
sourceopen	`Event`	`readyState` transitions from `"closed"` to `"open"` or from `"ended"` to `"open"`.
sourceended	`Event`	`readyState` transitions from `"open"` to `"ended"`.
sourceclose	`Event`	`readyState` transitions from `"open"` to `"closed"` or `"ended"` to `"closed"`.

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 absolute URL matches the MediaSource object URL, run the following steps right before the "*Perform a potentially CORS-enabled fetch*" step in the resource fetch algorithm.

↪ **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.

↪ **Otherwise**

1. Set the `readyState` attribute to `"open"`.
2. Queue a task to fire a simple event named `sourceopen` at the **`MediaSource`**.
3. Continue the resource fetch algorithm by running the "*Perform a potentially CORS-enabled fetch*" step. Text in the resource fetch algorithm that refers to "the download" or "bytes received" refer to data passed in via `appendBuffer()` and `appendStream()`. References to HTTP in the resource fetch algorithm do not apply because the HTMLMediaElement does not fetch media data via HTTP when a **`MediaSource`** is attached.

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 must be run right before the transition.

1. Set the `readyState` attribute to `"closed"`.

2. Set the `duration` attribute 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`.

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. The media element looks for media segments containing the *new playback position* in each `SourceBuffer` object in `activeSourceBuffers`.

   ↪ **If one or more of the objects in `activeSourceBuffers` is missing media segments for the *new playback position***

   1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.
   2. The media element waits until an `appendBuffer()` or an `appendStream()` 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` to determine what the media element needs to resume playback.

   ↪ **Otherwise**

   Continue

2. The media element resets all decoders and initializes each one with data from the appropriate initialization segment.
3. The media element feeds coded frames from the active track buffers into the decoders starting with the closest random access point before the the *new playback position*.
4. 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. Appending new segments and 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 `buffered` for all objects in `activeSourceBuffers` do not contain `TimeRanges` for the current playback position:**

1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.
2. If this is the first transition to `HAVE_METADATA`, then queue a task to fire a simple event named `loadedmetadata` at the media element.
3. Abort these steps.

↪ **If `buffered` for all objects in `activeSourceBuffers` contain `TimeRanges` that include the current playback position and enough data to ensure uninterrupted playback:**

1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_ENOUGH_DATA`.
2. Queue a task to fire a simple event named `canplaythrough` at the media element.
3. Playback may resume at this point if it was previously suspended by a transition to `HAVE_CURRENT_DATA`.
4. Abort these steps.

↪ **If `buffered` for at least one object in `activeSourceBuffers` contains a `TimeRange` that includes the current playback position but not enough data to ensure uninterrupted playback:**

1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_FUTURE_DATA`.
2. If the previous value of `HTMLMediaElement.readyState` was less than `HAVE_FUTURE_DATA`, then queue a task to fire a simple event named `canplay` at the media element.
3. Playback may resume at this point if it was previously suspended by a transition to `HAVE_CURRENT_DATA`.
4. Abort these steps.

↪ **If `buffered` for at least one object in `activeSourceBuffers` 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`.
2. If this is the first transition to `HAVE_CURRENT_DATA`, then queue a task to fire a simple event named `loadeddata` at the media element.
3. Playback is suspended at this point since the media element doesn't have enough data to advance the media timeline.
4. 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 tracks, 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. Set *old duration* to the current value of `duration`.
3. Update `duration` to *new duration*.
4. If the *new duration* is less than *old duration*, then call `remove(new duration, old duration)` on all objects in `sourceBuffers`.

   NOTE

   This preserves audio frames and text cues that start before and end after the `duration`.

5. If a user agent is unable to partially render audio frames or text cues that start before and end after the `duration`, then run the following steps:
   1. Update *new duration* to the highest end timestamp across all `SourceBuffer` objects in `sourceBuffers`.
   2. Update `duration` to *new duration*.
6. Update the `media controller 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, is null, or is an empty string**

   1. Run the duration change algorithm with *new duration* set to the highest end timestamp 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.

- ↪ **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.
  - ↪ **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.
- ↪ **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.
  - ↪ **If the `HTMLMediaElement.readyState` attribute is greater than `HAVE_NOTHING`**
    Run the media data is corrupted steps of the resource fetch algorithm.
- ↪ **Otherwise**
  Throw an `INVALID_ACCESS_ERR` exception.

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.

WebIDL

```
interface SourceBuffer : EventTarget {
                attribute AppendMode         mode;
    readonly    attribute boolean            updating;
    readonly    attribute TimeRanges         buffered;
                attribute double             timestampOffset;
    readonly    attribute AudioTrackList     audioTracks;
    readonly    attribute VideoTrackList     videoTracks;
    readonly    attribute TextTrackList      textTracks;
                attribute double             appendWindowStart;
                attribute unrestricted double appendWindowEnd;
    void appendBuffer (ArrayBuffer data);
    void appendBuffer (ArrayBufferView data);
    void appendStream (Stream stream, [EnforceRange] optional unsigned long long maxSize);
    void abort ();
    void remove (double start, double end);
};
```

3.1 Attributes

`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 `INVALID_STATE_ERR` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `INVALID_STATE_ERR` exception and abort these steps.
3. If the new value equals NaN, then throw an `INVALID_ACCESS_ERR` and abort these steps.
4. If the new value is less than or equal to `appendWindowStart` then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
5. Update the attribute to the new value.

`appendWindowStart` of type ***double***,

The presentation timestamp for the start of the append window. This attribute is initially set to 0.

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 `INVALID_STATE_ERR` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `INVALID_STATE_ERR` exception and abort these steps.
3. If the new value is less than 0 or greater than or equal to `appendWindowEnd` then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
4. Update the attribute to the new value.

`audioTracks` of type ***AudioTrackList***, readonly
The list of `AudioTrack` objects created by this object.

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

Indicates what `TimeRanges` are buffered in the **`SourceBuffer`**.

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 `INVALID_STATE_ERR` exception and abort these steps.
2. Return a new static normalized TimeRanges object for the media segments buffered.

`mode` of type *AppendMode*,

Controls how a sequence of media segments are handled. This attribute is initially set to `"segments"` when 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. Let *new mode* equal the new value being assigned to this attribute.
2. If *new mode* does not equal `"segments"` or `"sequence"`, then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
3. If this object has been removed from the `sourceBuffers` attribute of the parent media source, then throw an `INVALID_STATE_ERR` exception and abort these steps.
4. If the `updating` attribute equals true, then throw an `INVALID_STATE_ERR` 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 `INVALID_STATE_ERR` and abort these steps.
7. If the *new mode* equals `"sequence"`, then set the *group start timestamp* to the *highest presentation end timestamp*.
8. Update the attribute to *new mode*.

`textTracks` of type ***TextTrackList***, readonly
The list of `TextTrack` objects created by this object.

`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 `INVALID_STATE_ERR` exception and abort these steps.

3. If the `updating` attribute equals true, then throw an `INVALID_STATE_ERR` 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 `INVALID_STATE_ERR` 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*.

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

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

`videoTracks` of type ***VideoTrackList***, readonly
The list of `VideoTrack` objects created by this object.

3.2 Methods

`abort`

Aborts the current segment and resets the segment parser.

No parameters.
Return type: **`void`**

When this method is invoked, the user agent must run the following steps:

1. If this object has been removed from the `sourceBuffers` attribute of the parent media source then throw an `INVALID_STATE_ERR` exception and abort these steps.

2. If the `readyState` attribute of the parent media source is not in the `"open"` state then throw an `INVALID_STATE_ERR` exception and abort these steps.

3. If the `updating` attribute equals true, then run the following steps:

    1. Abort the buffer append and stream append loop algorithms if they are running.

    2. Set the `updating` attribute to false.

    3. Queue a task to fire a simple event named `abort` at this **`SourceBuffer`** object.

    4. Queue a task to fire a simple event named `updateend` at this **`SourceBuffer`** object.

4. Run the reset parser state algorithm.

5. Set `appendWindowStart` to 0.
6. Set `appendWindowEnd` to positive Infinity.

`appendBuffer`

Appends the segment data in an **`ArrayBuffer`**[TYPED-ARRAYS] to the source buffer.

The steps for this method are the same as the ArrayBufferView version of `appendBuffer()`.

Parameter	Type	Nullable	Optional	Description
data	`ArrayBuffer`	✗	✗	

Return type: **`void`**

`appendBuffer`

Appends the segment data in an **`ArrayBufferView`**[TYPED-ARRAYS] to the source buffer.

Parameter	Type	Nullable	Optional	Description
data	`ArrayBufferView`	✗	✗	

Return type: **`void`**

When this method is invoked, the user agent must run the following steps:

1. If *data* is null then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
2. Run the prepare append algorithm.
3. Add *data* to the end of the *input buffer*.
4. Set the `updating` attribute to true.
5. Queue a task to fire a simple event named `updatestart` at this **`SourceBuffer`** object.
6. Asynchronously run the buffer append algorithm.

`appendStream`

Appends segment data to the source buffer from a **`Stream`**[STREAMS-API].

Parameter	Type	Nullable	Optional	Description
stream	`Stream`	✗	✗	
maxSize	`unsigned long long`	✗	✓	

Return type: **`void`**

When this method is invoked, the user agent must run the following steps:

1. If *stream* is null then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
2. Run the prepare append algorithm.
3. Set the `updating` attribute to true.

4. Queue a task to fire a simple event named `updatestart` at this `SourceBuffer` object.

5. Asynchronously run the stream append loop algorithm with *stream* and *maxSize*.

`remove`

Removes media for a specific time range.

Parameter	Type	Nullable	Optional	Description
start	`double`	✗	✗	
end	`double`	✗	✗	

Return type: `void`

When this method is invoked, the user agent must run the following steps:

1. If *start* is negative or greater than `duration`, then throw an `INVALID_ACCESS_ERR` exception and abort these steps.

2. If *end* is less than or equal to *start*, then throw an `INVALID_ACCESS_ERR` exception and abort these steps.

3. If this object has been removed from the `sourceBuffers` attribute of the parent media source then throw an `INVALID_STATE_ERR` exception and abort these steps.

4. If the `updating` attribute equals true, then throw an `INVALID_STATE_ERR` 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. Set the `updating` attribute to true.

7. Queue a task to fire a simple event named `updatestart` at this `SourceBuffer` object.

8. Return control to the caller and run the rest of the steps asynchronously.

9. Run the coded frame removal algorithm with *start* and *end* as the start and end of the removal range.

10. Set the `updating` attribute to false.

11. Queue a task to fire a simple event named `update` at this `SourceBuffer` object.

12. Queue a task to fire a simple event named `updateend` at this `SourceBuffer` object.

3.3 Track Buffers

A ***track buffer*** stores the track descriptions and coded frames for an individual track. The track buffer is updated as initialization segments and media segments are appended to the `SourceBuffer`.

Each track buffer has a ***last decode timestamp*** variable that stores the decode timestamp of the last coded frame appended in the current coded frame group. The variable is initially unset to indicate that no coded frames have been appended yet.

Each track buffer has a ***last frame duration*** variable that stores the coded frame duration of the last coded frame appended in the current coded frame group. The variable is initially unset to indicate that no coded frames have been appended yet.

Each track buffer has a ***highest presentation timestamp*** variable that stores the highest presentation timestamp encountered in a coded frame appended in the current coded frame group. The variable is initially unset to indicate that no coded frames have been appended yet.

Each track buffer has a ***need random access point flag*** variable that keeps track of whether the track buffer is waiting for a random access point coded frame. The variable is initially set to true to indicate that random access point coded frame is needed before anything can be added to the track buffer.

3.4 Event Summary

Event name	Interface	Dispatched when...
updatestart	`Event`	`updating` transitions from false to true.
update	`Event`	The append or remove has successfully completed. `updating` transitions from true to false.
updateend	`Event`	The append or remove has ended.
error	`Event`	An error occurred during the append. `updating` transitions from true to false.
abort	`Event`	The append or remove was aborted by an `abort()` call. `updating` transitions from true to false.

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()` and `appendStream()` calls. The buffer is empty when the SourceBuffer object is created.

The ***buffer full flag*** keeps track of whether `appendBuffer()` or `appendStream()` 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 ***highest presentation end timestamp*** variable stores the highest presentation end timestamp encountered 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.

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 end of stream algorithm with the *error* parameter set to `"decode"` 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 an 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 flag* is false, then run the end of stream algorithm with the *error* parameter set to `"decode"` and abort this algorithm.
    2. If the *input buffer* does not contain a complete media segment header yet, then jump to the *need more data* step below.
    3. If the *input buffer* contains one or more complete coded frames, then run the coded frame processing algorithm.

        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.

    4. If this `SourceBuffer` is full and cannot accept more media data, then set the *buffer full flag* to true.
    5. If the *input buffer* does not contain a complete media segment, then jump to the *need more data* step below.
    6. Remove the media segment bytes from the beginning of the *input buffer*.
    7. Set *append state* to WAITING_FOR_SEGMENT.
    8. 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 presentation timestamp* on all track buffers.
5. Set the *need random access point flag* on all track buffers to true.
6. Remove all bytes from the *input buffer*.
7. Set *append state* to WAITING_FOR_SEGMENT.

3.5.3 Append Error Algorithm

When an error occurs during an append, run the following steps:

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.

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 `INVALID_STATE_ERR` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `INVALID_STATE_ERR` exception and abort these steps.
3. 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 .

4. Run the coded frame eviction algorithm.
5. If the *buffer full flag* equals true, then throw a `QUOTA_EXCEEDED_ERR` exception and abort these step.

   NOTE

   This is the signal that the implementation was unable to evict enough data to accomodate 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 Stream Append Loop

When a **`Stream`**[STREAMS-API] is passed to `appendStream()`, the following steps are run to transfer data from the **`Stream`** to the **`SourceBuffer`**. This algorithm is initialized with the *stream* and *maxSize* parameters from the `appendStream()` call.

1. If *maxSize* is set, then let *bytesLeft* equal *maxSize*.
2. *Loop Top:* If *maxSize* is set and *bytesLeft* equals 0, then jump to the *loop done* step below.
3. If *stream* has been closed, then jump to the *loop done* step below.
4. Read data from *stream* into *data*:

   ↪ **If *maxSize* is set:**

      1. Read up to *bytesLeft* bytes from *stream* into *data*.
      2. Subtract the number of bytes in *data* from *bytesLeft*.

   ↪ **Otherwise:**

      Read all available bytes in *stream* into *data*.

5. If an error occured while reading from *stream*, then run the append error algorithm and abort this algorithm.

6. Run the coded frame eviction algorithm.
7. If the *buffer full flag* equals true, then run the append error algorithm and abort this algorithm.

   NOTE

   The web application should use `remove()` to free up space in the **`SourceBuffer`**.

8. Add *data* to the end of the *input buffer*.
9. Run the segment parser loop algorithm.
10. If the segment parser loop algorithm in the previous step was aborted, then abort this algorithm.
11. Jump to the *loop top* step above.
12. *Loop Done:* Set the `updating` attribute to false.
13. Queue a task to fire a simple event named `update` at this **`SourceBuffer`** object.
14. 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 flag*** that tracks whether the first initialization segment has been appended. 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 end of stream algorithm with the *error* parameter set to `"decode"` and abort these steps.
3. If the *first initialization segment flag* is true, then run the following steps:
   1. Verify the following properties. If any of the checks fail then run the end of stream algorithm with the *error* parameter set to `"decode"` and abort these steps.
      - The number of audio, video, and text tracks match what was in the first initialization segment.
      - The codecs for each track, match what was specified in the first initialization segment.
      - If more than one track for a single type are present (ie 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.
4. Let *active track flag* equal false.
5. If the *first initialization segment 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 end of stream algorithm with the *error* parameter set to `"decode"` 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 *new audio track* be a new `AudioTrack` object.
   2. Generate a unique ID and assign it to the `id` property on *new audio track*.
   3. 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.

   4. Add *new audio track* to the `audioTracks` attribute on this **`SourceBuffer`** object.
   5. Queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the `AudioTrackList` object referenced by the `audioTracks` attribute on this **`SourceBuffer`** object.
   6. Add *new audio track* to the `audioTracks` attribute on the HTMLMediaElement.
   7. Queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the `AudioTrackList` object referenced by the `audioTracks` attribute on the HTMLMediaElement.
   8. Create a new track buffer to store coded frames for this track.
   9. 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 *new video track* be a new `VideoTrack` object.
   2. Generate a unique ID and assign it to the `id` property on *new video track*.
   3. 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.

   4. Add *new video track* to the `videoTracks` attribute on this **`SourceBuffer`** object.
   5. Queue a task to fire a trusted event named `addtrack`, that does not bubble and is not

cancelable, and that uses the `TrackEvent` interface, at the `VideoTrackList` object referenced by the `videoTracks` attribute on this **`SourceBuffer`** object.

  6. Add *new video track* to the `videoTracks` attribute on the HTMLMediaElement.
  7. Queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at the `VideoTrackList` object referenced by the `videoTracks` attribute on the HTMLMediaElement.
  8. Create a new track buffer to store coded frames for this track.
  9. 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 *new text track* be a new `TextTrack` object with its properties populated with the appropriate information from the initialization segment.
    2. If the `mode` property on *new text track* equals `"showing"` or `"hidden"`, then set *active track flag* to true.
    3. Add *new text track* to the `textTracks` attribute on this **`SourceBuffer`** object.
    4. Queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, at `textTracks` attribute on this **`SourceBuffer`** object.
    5. Add *new text track* to the `textTracks` attribute on the HTMLMediaElement.
    6. Queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, 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 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 flag* set to false, then abort these steps.
    2. Set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.
    3. 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`.

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:* 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 explicilty 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 *highest presentation 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*.

        3. If the *presentation timestamp* or *decode timestamp* is less than the presentation start time, then run the end of stream algorithm with the *error* parameter set to `"decode"`, and abort these steps.

    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 *highest presentation end timestamp* to *presentation timestamp*.
   ↪ **If `mode` equals `"sequence"`:**
   Set *group start timestamp* equal to the *highest presentation 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 presentation 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.

↪ **Otherwise:**
Continue.

8. Let *frame end timestamp* equal the sum of *presentation timestamp* and *frame duration*.
9. 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 that are outside the append window 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.

10. 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.
11. 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.
12. Let *spliced audio frame* be an unset variable for holding audio splice information
13. Let *spliced timed text frame* be an unset variable for holding timed text splice information
14. 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 *overlapped frame presentation timestamp* equal the presentation timestamp of *overlapped frame*.
   2. Let *remove window timestamp* equal *overlapped frame presentation timestamp* plus 1 microsecond.
   3. If the *presentation timestamp* is less than the *remove window timestamp*, then remove *overlapped frame* and any coded frames that depend on it 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.

   ↪ **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*.

15. Remove existing coded frames in *track buffer*:

    ↪ **If *highest presentation 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 presentation timestamp* for *track buffer* is set and less than *presentation timestamp***

    Remove all coded frames from *track buffer* that have a presentation timestamp greater than *highest presentation timestamp* and less than or equal to *frame end timestamp*.

16. Remove decoding dependencies of the coded frames removed in the previous step:

    ↪ **If detailed information about decoding dependencies is available:**

    Remove all coded frames from *track buffer* that have decoding dependencies on the coded frames removed in the previous step.

> NOTE
>
> For example if an I-frame is removed in the previous step, then all P-frames & B-frames that depend on that I-frame should be removed from *track buffer*. This makes sure that decode dependencies are properly maintained during overlaps.

↪ **Otherwise:**
Remove all coded frames between the coded 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.

17. ↪ **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*.

18. Set *last decode timestamp* for *track buffer* to *decode timestamp*.

19. Set *last frame duration* for *track buffer* to *frame duration*.

20. If *highest presentation timestamp* for *track buffer* is unset or *frame end timestamp* is greater than *highest presentation timestamp*, then set *highest presentation 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 eventhough the decode timestamps are monotonically increasing.

21. If *highest presentation end timestamp* is unset or *frame end timestamp* is greater than *highest presentation end timestamp*, then set *highest presentation end timestamp* equal to *frame end timestamp*.

2. If the `HTMLMediaElement.readyState` attribute is `HAVE_METADATA` and the new coded frames cause all objects in `activeSourceBuffers` to have media data for the current playback position, then run the following steps:

   1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_CURRENT_DATA`.

   2. If this is the first transition to `HAVE_CURRENT_DATA`, then queue a task to fire a simple event named `loadeddata` at the media element.

3. If the `HTMLMediaElement.readyState` attribute is `HAVE_CURRENT_DATA` and the new coded frames cause all objects in `activeSourceBuffers` to have media data beyond the current playback position, then run the following steps:

    1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_FUTURE_DATA`.

    2. Queue a task to fire a simple event named `canplay` at the media element.

4. If the `HTMLMediaElement.readyState` attribute is `HAVE_FUTURE_DATA` and the new coded frames cause all objects in `activeSourceBuffers` to have enough data to ensure uninterrupted playback, then run the following steps:

    1. Set the `HTMLMediaElement.readyState` attribute to `HAVE_ENOUGH_DATA`.

    2. Queue a task to fire a simple event named `canplaythrough` at the media element.

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 highest end timestamp reported by `HTMLMediaElement.buffered`.

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*.

    4. 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

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. (eg. 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
>
> *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)

8. If the user agent does not support crossfading then run the following steps:
    1. Remove *overlapped frame* from *track buffer*.
    2. 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.

    3. 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.

9. Let *frame end timestamp* equal the sum of *presentation timestamp* and *frame duration*.
10. Let *splice end timestamp* equal the sum of *presentation timestamp* and the splice duration of 5 milliseconds.
11. 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*.
12. Remove all the frames included in *fade out coded frames* from *track buffer*.
13. 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*.

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:

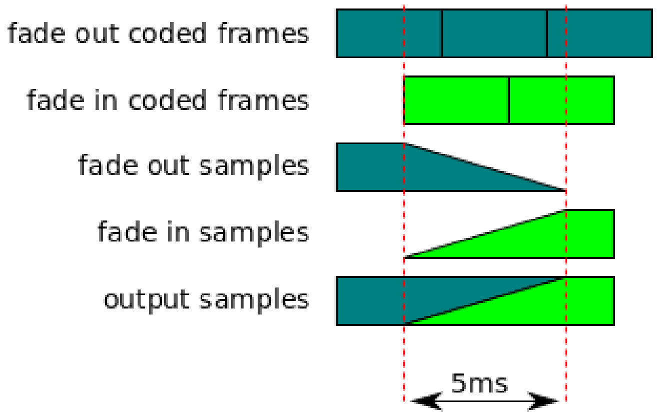
1. Let *fade out coded frames* be the coded frames that are faded out during the splice.
2. Let *fade in coded frames* be the coded frames that are faded in during the splice.
3. Let *presentation timestamp* be the presentation timestamp of the first coded frame in *fade out coded frames*.
4. Let *end 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.

4. SourceBufferList Object

SourceBufferList is a simple container object for **`SourceBuffer`** objects. It provides read-only array access and fires events when the list is modified.

WebIDL

```
interface SourceBufferList : EventTarget {
    readonly    attribute unsigned long length;
    getter SourceBuffer (unsigned long index);
};
```

4.1 Attributes

`length` of type ***unsigned long***, readonly

Indicates the number of **`SourceBuffer`** objects in the list.

4.2 Methods

`SourceBuffer`

Allows the SourceBuffer objects in the list to be accessed with an array operator (i.e. []).

Parameter	Type	Nullable	Optional	Description
index	`unsigned long`	✗	✗	

Return type: `getter`

When this method is invoked, the user agent must run the following steps:

1. If *index* is greater than or equal to the `length` attribute then return undefined and abort these steps.

2. Return the *index*'th **`SourceBuffer`** object in the list.

4.3 Event Summary

Event name	Interface	Dispatched when...
`addsourcebuffer`	`Event`	When a **`SourceBuffer`** is added to the list.
`removesourcebuffer`	`Event`	When a **`SourceBuffer`** is removed from the list.

5. VideoPlaybackQuality Object

WebIDL

```
interface VideoPlaybackQuality {
    readonly    attribute DOMHighResTimeStamp creationTime;
    readonly    attribute unsigned long       totalVideoFrames;
    readonly    attribute unsigned long       droppedVideoFrames;
    readonly    attribute unsigned long       corruptedVideoFrames;
    readonly    attribute double              totalFrameDelay;
};
```

5.1 Attributes

`corruptedVideoFrames` of type ***unsigned long***, readonly

The total number of corrupted frames that have been detected.

`creationTime` of type ***DOMHighResTimeStamp***, readonly

The timestamp returned by Performance.now() when this object was created.

`droppedVideoFrames` of type ***unsigned long***, readonly

The total number of frames dropped predecode or dropped because the frame missed its display deadline.

`totalFrameDelay` of type ***double***, readonly

The sum of all displayed frame delays for all displayed frames. (i.e., Frames included in the `totalVideoFrames` count, but not in the `droppedVideoFrames` count.

`totalVideoFrames` of type ***unsigned long***, readonly

The total number of frames that would have been displayed if no frames are dropped.

6. URL Object Extensions

This section specifies extensions to the URL[FILE-API] object definition.

WebIDL

```
partial interface URL {
    static DOMString createObjectURL (MediaSource mediaSource);
};
```

6.1 Methods

`createObjectURL`, static

Creates URLs for **`MediaSource`** objects.

NOTE

This algorithm is intended to mirror the behavior of the createObjectURL()[FILE-API] method with autoRevoke set to true.

Parameter	Type	Nullable	Optional	Description
mediaSource	`MediaSource`	✗	✗	

Return type: `DOMString`

When this method is invoked, the user agent must run the following steps:

1. If *mediaSource* is NULL the return null.
2. Return a unique MediaSource object URL that can be used to dereference the *mediaSource* argument, and run the rest of the algorithm asynchronously.
3. provide a stable state
4. Revoke the MediaSource object URL by calling revokeObjectURL() on it.

7. HTMLMediaElement Extensions

This section specifies what existing attributes on the `HTMLMediaElement` must return when a `MediaSource` is attached to the element.

The HTMLMediaElement.seekable attribute returns a new static normalized TimeRanges object created based on the following steps:

↪ **If `duration` equals NaN**
Return an empty `TimeRanges` object.

↪ **If `duration` equals positive Infinity**
Return a single range with a start time of 0 and an end time equal to the highest end time reported by the `HTMLMediaElement.buffered` attribute.

↪ **Otherwise**
Return a single range with a start time of 0 and an end time equal to `duration`.

The `HTMLMediaElement.buffered` attribute returns a new static normalized TimeRanges object created based on the following steps:

1. If `activeSourceBuffers`.length equals 0 then return an empty `TimeRanges` object and abort these steps.
2. Let *active ranges* be the ranges returned by `buffered` for each `SourceBuffer` object in `activeSourceBuffers`.
3. Let *highest end time* be the largest range end time in the *active ranges*.
4. Let *intersection ranges* equal a `TimeRange` object containing a single range from 0 to *highest end time*.
5. For each `SourceBuffer` object in `activeSourceBuffers` run the following steps:
   1. Let *source ranges* equal the ranges returned by the `buffered` attribute on the current `SourceBuffer`.
   2. If `readyState` is `"ended"`, then set the end time on the last range in *source ranges* to *highest end time*.
   3. Let *new intersection ranges* equal the the intersection between the *intersection ranges* and the *source ranges*.
   4. Replace the ranges in *intersection ranges* with the *new intersection ranges*.
6. Return the *intersection ranges*.

8. HTMLVideoElement Extensions

This section specifies new attributes and internal state that are being added to the `HTMLVideoElement`.

Each `HTMLVideoElement` will maintain a ***total video frame count*** variable that keeps track of the total number of frames that have been displayed and dropped. This variable is initialized to 0 when the element is created and whenever the media element load algorithm is invoked. It is incremented when a video frame is displayed or when the *dropped video frame count* is incremented.

Each `HTMLVideoElement` will maintain a ***dropped video frame count*** variable that keeps track of the total number of frames that have been dropped. This variable is initialized to 0 when the element is created and whenever the media element load algorithm is invoked. It is incremented when a video frame is dropped predecode or when a frame is decoded but dropped because it missed a display deadline.

Each `HTMLVideoElement` will maintain a ***corrupted video frame count*** variable that keeps track of the total number of corrupted frames detected. This variable is initialized to 0 when the element is created and whenever the media element load algorithm is invoked. It is incremented when a corrupted video frame is detected by the decoder. It is up to the implementation to determine whether to display or drop a corrupted frame. Whichever choice is made, the *total video frame count* and *dropped video frame count* must be updated appropriately.

Each `HTMLVideoElement` will maintain a ***displayed frame delay sum*** variable that keeps track of the sum of all displayed frame delays. This variable is initialized to 0 when the element is created and whenever the media element load algorithm is invoked. When a frame is displayed, its displayed frame delay is computed and added to this variable.

WebIDL

```
partial interface HTMLVideoElement {
    VideoPlaybackQuality getVideoPlaybackQuality ();
};
```

8.1 Methods

`getVideoPlaybackQuality`

Provides the current the playback quality metrics.

No parameters.
Return type: **`VideoPlaybackQuality`**

When this method is invoked, the user agent must run the following steps:

1. Let *playbackQuality* be a new instance of **`VideoPlaybackQuality`**.
2. Set *playbackQuality*.`creationTime` to the value returned by a call to Performance.now().
3. Set *playbackQuality*.`totalVideoFrames` to the current value of the *total video frame count*.
4. Set *playbackQuality*.`droppedVideoFrames` to the current value of the *dropped video frame count*.
5. Set *playbackQuality*.`corruptedVideoFrames` to the current value of the *corrupted video frame count*.

6. Set *playbackQuality*.`totalFrameDelay` to the current value of the *displayed frame delay sum*.

7. Return *playbackQuality*.

9. AudioTrack Extensions

This section specifies extensions to the HTML `AudioTrack` definition.

WebIDL

```
partial interface AudioTrack {
                attribute DOMString     kind;
                attribute DOMString     language;
    readonly    attribute SourceBuffer? sourceBuffer;
};
```

9.1 Attributes

`kind` of type ***DOMString***,

Allows the web application to get and update the track `kind`.

On getting, return the current value of the attribute. This is either the value provided when this object was created or the value provided on the last successful set operation.

On setting, run the following steps:

1. If the value being assigned to this attribute does not match one of the kind categories, then abort these steps.
2. Update this attribute to the new value.
3. If the `sourceBuffer` attribute on this track is not null, then queue a task to fire a simple event named `change` at `sourceBuffer`.`audioTracks`.
4. Queue a task to fire a simple event named `change` at the `AudioTrackList` object referenced by the `audioTracks` attribute on the HTMLMediaElement.

`language` of type ***DOMString***,

Allows the web application to get and update the track `language`.

On getting, return the current value of the attribute. This is either the value provided when this object was created or the value provided on the last successful set operation.

On setting, run the following steps:

1. If the value being assigned to this attribute is not an empty string or a BCP 47 language tag[BCP47], then abort these steps.
2. Update this attribute to the new value.
3. If the `sourceBuffer` attribute on this track is not null, then queue a task to fire a simple event named `change` at `sourceBuffer`.`audioTracks`.
4. Queue a task to fire a simple event named `change` at the `AudioTrackList` object referenced by the `audioTracks` attribute on the HTMLMediaElement.

`sourceBuffer` of type *SourceBuffer*, readonly , nullable

Returns the **`SourceBuffer`** that created this track. Returns null if this track was not created by a **`SourceBuffer`** or the **`SourceBuffer`** has been removed from the `sourceBuffers` attribute of its parent media source.

10. VideoTrack Extensions

This section specifies extensions to the HTML `VideoTrack` definition.

WebIDL

```
partial interface VideoTrack {
                attribute DOMString     kind;
                attribute DOMString     language;
    readonly    attribute SourceBuffer? sourceBuffer;
};
```

10.1 Attributes

`kind` of type ***DOMString***,

Allows the web application to get and update the track `kind`.

On getting, return the current value of the attribute. This is either the value provided when this object was created or the value provided on the last successful set operation.

On setting, run the following steps:

1. If the value being assigned to this attribute does not match one of the kind categories, then abort these steps.
2. Update this attribute to the new value.
3. If the `sourceBuffer` attribute on this track is not null, then queue a task to fire a simple event named `change` at `sourceBuffer`.`videoTracks`.
4. Queue a task to fire a simple event named `change` at the `VideoTrackList` object referenced by the `videoTracks` attribute on the HTMLMediaElement.

`language` of type ***DOMString***,

Allows the web application to get and update the track `language`.

On getting, return the current value of the attribute. This is either the value provided when this object was created or the value provided on the last successful set operation.

On setting, run the following steps:

1. If the value being assigned to this attribute is not an empty string or a BCP 47 language tag[BCP47], then abort these steps.
2. Update this attribute to the new value.
3. If the `sourceBuffer` attribute on this track is not null, then queue a task to fire a simple event named `change` at `sourceBuffer`.`videoTracks`.
4. Queue a task to fire a simple event named `change` at the `VideoTrackList` object

referenced by the `videoTracks` attribute on the HTMLMediaElement.

`sourceBuffer` of type *SourceBuffer*, readonly , nullable

Returns the **`SourceBuffer`** that created this track. Returns null if this track was not created by a **`SourceBuffer`** or the **`SourceBuffer`** has been removed from the `sourceBuffers` attribute of its parent media source.

11. TextTrack Extensions

This section specifies extensions to the HTML `TextTrack` definition.

WebIDL

```
partial interface TextTrack {
                attribute DOMString     kind;
                attribute DOMString     language;
    readonly    attribute SourceBuffer? sourceBuffer;
};
```

11.1 Attributes

`kind` of type ***DOMString***,

Allows the web application to get and update the track `kind`.

On getting, return the current value of the attribute. This is either the value provided when this object was created or the value provided on the last successful set operation.

On setting, run the following steps:

1. If the value being assigned to this attribute does not match one of the text track kinds, then abort these steps.
2. Update this attribute to the new value.
3. If the `sourceBuffer` attribute on this track is not null, then queue a task to fire a simple event named `change` at `sourceBuffer`.`textTracks`.
4. Queue a task to fire a simple event named `change` at the `TextTrackList` object referenced by the `textTracks` attribute on the HTMLMediaElement.

`language` of type ***DOMString***,

Allows the web application to get and update the track `language`.

On getting, return the current value of the attribute. This is either the value provided when this object was created or the value provided on the last successful set operation.

On setting, run the following steps:

1. If the value being assigned to this attribute is not an valid text track language, then abort these steps.
2. Update this attribute to the new value.
3. If the `sourceBuffer` attribute on this track is not null, then queue a task to fire a simple event named `change` at `sourceBuffer`.`textTracks`.

4. Queue a task to fire a simple event named `change` at the `TextTrackList` object referenced by the `textTracks` attribute on the HTMLMediaElement.

`sourceBuffer` of type *SourceBuffer*, readonly , nullable

Returns the **`SourceBuffer`** that created this track. Returns null if this track was not created by a **`SourceBuffer`** or the **`SourceBuffer`** has been removed from the `sourceBuffers` attribute of its parent media source.

12. Byte Stream Formats

The bytes provided through `appendBuffer()` and `appendStream()` for a **`SourceBuffer`** form a logical byte stream. The format and semantics of these byte streams are defined in ***byte stream format specifications***. The byte stream format registry is the authoritative source for byte stream format specifications that can be accepted by a **`SourceBuffer`**. If a **`MediaSource`** implementation claims to support any of the MIME types in the registry, then it must implement the corresponding byte stream format specification.

NOTE

The byte stream format specifications in the registry are not intended to define new storage formats. They simply outline the subset of existing storage format structures that implementations of this specification will accept.

NOTE

Byte stream format parsing and validation is implemented in the segment parser loop algorithm.

This section provides general requirements for all byte stream format specifications:

- A byte stream format specification must define initialization segments and media segments.
- It must be possible to identify segment boundaries and segment type (initialization or media) by examining the byte stream alone.
- The user agent must run the end of stream algorithm with the *error* parameter set to `"decode"` when any of the following conditions are met:

    1. The number and type of tracks are not consistent.

        NOTE

        For example, if the first initialization segment has 2 audio tracks and 1 video track, then all initialization segments that follow it in the byte stream must describe 2 audio tracks and 1 video track.

    2. Track IDs are not the same across initialization segments, for segments describing multiple tracks of a single type. (e.g. 2 audio tracks).
    3. Codecs changes across initialization segments.

        NOTE

> For example, a byte stream that starts with an initialization segment that specifies a single AAC track and later contains an initialization segment that specifies a single AMR-WB track is not allowed. Support for multiple codecs is handled with multiple `SourceBuffer` objects.

- The user agent must support the following:
  1. Track IDs changing across initialization segments if the segments describes only one track of each type.
  2. Video frame size changes. The user agent must support seamless playback.

     > NOTE
     >
     > This will cause the <video> display region to change size if the web application does not use CSS or HTML attributes (width/height) to constrain the element size.

  3. Audio channel count changes. The user agent may support this seamlessly and could trigger downmixing.

     > NOTE
     >
     > This is a quality of implementation issue because changing the channel count may require reinitializing the audio device, resamplers, and channel mixers which tends to be audible.

- The following rules apply to all media segments within a byte stream. A user agent must:
  1. Map all timestamps to the same media timeline.
  2. Support seamless playback of media segments having a timestamp gap smaller than the audio frame size. User agent must not reflect these gaps in the `buffered` attribute.

     > NOTE
     >
     > This is intended to simplify switching between audio streams where the frame boundaries don't always line up across encodings (e.g. Vorbis).

- The user agent must run the end of stream algorithm with the *error* parameter set to `"decode"` when any combination of an initialization segment and any contiguous sequence of media segments satisfies the following conditions:
  1. The number and type (audio, video, text, etc.) of all tracks in the media segments are not identified.
  2. The decoding capabilities needed to decode each track (i.e. codec and codec parameters) are not provided.
  3. Encryption parameters necessary to decrypt the content (except the encryption key itself) are not provided for all encrypted tracks.
  4. All information necessary to decode and render the earliest random access point in the sequence of media segments and all subsequence samples in the sequence (in presentation time) are not provided. This includes in particular,
     - Information that determines the intrinsic width and height of the video (specifically, this requires either the picture or pixel aspect ratio, together with the encoded resolution).
     - Information necessary to convert the video decoder output to a format suitable for

display

5. Information necessary to compute the global presentation timestamp of every sample in the sequence of media segments is not provided.

For example, if I1 is associated with M1, M2, M3 then the above must hold for all the combinations I1+M1, I1+M2, I1+M1+M2, I1+M2+M3, etc.

Byte stream specifications must at a minimum define constraints which ensure that the above requirements hold. Additional constraints may be defined, for example to simplify implementation.

13. Examples

Example use of the Media Source Extensions

```
<script>
  function onSourceOpen(videoTag, e) {
    var mediaSource = e.target;

    if (mediaSource.sourceBuffers.length > 0)
        return;

    var sourceBuffer = mediaSource.addSourceBuffer('video/webm; codecs="vorbis,vp8"');

    videoTag.addEventListener('seeking', onSeeking.bind(videoTag, mediaSource));
    videoTag.addEventListener('progress', onProgress.bind(videoTag, mediaSource));

    var initSegment = GetInitializationSegment();

    if (initSegment == null) {
      // Error fetching the initialization segment. Signal end of stream with an error.
      mediaSource.endOfStream("network");
      return;
    }

    // Append the initialization segment.
    var firstAppendHandler = function(e) {
      var sourceBuffer = e.target;
      sourceBuffer.removeEventListener('updateend', firstAppendHandler);

      // Append some initial media data.
      appendNextMediaSegment(mediaSource);
    };
    sourceBuffer.addEventListener('updateend', firstAppendHandler);
    sourceBuffer.appendBuffer(initSegment);
  }

  function appendNextMediaSegment(mediaSource) {
    if (mediaSource.readyState == "closed")
      return;

    // If we have run out of stream data, then signal end of stream.
    if (!HaveMoreMediaSegments()) {
      mediaSource.endOfStream();
      return;
    }

    // Make sure the previous append is not still pending.
    if (mediaSource.sourceBuffers[0].updating)
        return;

    var mediaSegment = GetNextMediaSegment();

    if (!mediaSegment) {
      // Error fetching the next media segment.
      mediaSource.endOfStream("network");
      return;
```

```
        }

        // NOTE: If mediaSource.readyState == “ended”, this appendBuffer() call will
        // cause mediaSource.readyState to transition to "open". The web application
        // should be prepared to handle multiple “sourceopen” events.
        mediaSource.sourceBuffers[0].appendBuffer(mediaSegment);
      }

      function onSeeking(mediaSource, e) {
        var video = e.target;

        if (mediaSource.readyState == "open") {
          // Abort current segment append.
          mediaSource.sourceBuffers[0].abort();
        }

        // Notify the media segment loading code to start fetching data at the
        // new playback position.
        SeekToMediaSegmentAt(video.currentTime);

        // Append a media segment from the new playback position.
        appendNextMediaSegment(mediaSource);
      }

      function onProgress(mediaSource, e) {
        appendNextMediaSegment(mediaSource);
      }
    </script>

    <video id="v" autoplay> </video>

    <script>
      var video = document.getElementById('v');
      var mediaSource = new MediaSource();
      mediaSource.addEventListener('sourceopen', onSourceOpen.bind(this, video));
      video.src = window.URL.createObjectURL(mediaSource);
    </script>
```

14. Acknowledgments

The editors would like to thank Alex Giladi, Bob Lund, Chris Poole, Cyril Concolato, David Dorwin, David Singer, Duncan Rowden, Frank Galligan, Glenn Adams, Jerry Smith, Joe Steele, John Simmons, Kevin Streeter, Mark Vickers, Matt Ward, Michael Thornburgh, Philip Jägenstedt, Pierre Lemieux, Ralph Giles, Steven Robertson, and Tatsuya Igarashi for their contributions to this specification.

15. Revision History

Version	Comment
14 November 2013	• Bug 23663 - Clarify seeking behavior in Section 2.4.3.
04 November 2013	• Bug 23441 - Established MSE byte stream format registry and extracted byte stream format text into separate documents.
29 October 2013	• Bug 23553 - Fixed segment parser loop so it doesn't appear to prematurely remove the media segment header. • Bug 23557 - Update SPS/PPS note to not explicitly recommend specific avc

Version	Comment
	versions.
28 October 2013	• Bug 23549 - Add definitions for decode timestamp, presentation timestamp, an presentation order. • Bug 23552 - Clarify 'this' in section 3.5.1 • Bug 23554 - Introduced presentation interval and coded frame duration terms to clarify text.
15 October 2013	• Bug 23525 - Fix mvex box error behavior.
04 October 2013	• Bug 23442 - Fix example to work when seeking in the 'ended' state.
26 July 2013	• Bug 22136 - Added text for Inband SPS/PPS support. • Bug 22776 - Clarified that implementations are only required to support one SourceBuffer configuration at a time.
18 July 2013	• Bug 22117 - Reword byte stream specs in terms of UA behavior. • Bug 22148 - Replace VideoPlaybackQuality.playbackJitter with VideoPlaybackQuality.totalFrameDelay.
02 July 2013	• Bug 22401 - Fix typo • Bug 22134 - Clarify byte stream format enforcement. • Bug 22431 - Convert videoPlaybackQuality attribute to getVideoPlaybackQuality() method. • Bug 22109 - Renamed 'coded frame sequence' to 'coded frame group' to avoid confusion around multiple 'sequence' concepts.
05 June 2013	• Bug 22139 - Added a note clarifying that byte stream specs aren't defining new storage formats. • Bug 22148 - Added playbackJitter metric. • Bug 22134 - Added minimal number of SourceBuffers requirements. • Bug 22115 - Make algorithm abort text consistent. • Bug 22113 - Address typos. • Bug 22065 - Fix infinite loop in coded frame processing algorithm.
01 June 2013	• Bug 21431 - Updated coded frame processing algorithm for text splicing. • Bug 22035 - Update addtrack and removetrack event firing text to match HTML5 language. • Bug 22111 - Remove useless playback sentence from end of stream algorithm. • Bug 22052 - Add corrupted frame metric. • Bug 22062 - Added links for filing bugs. • Bug 22125 - Add "ended" to "open" transition to remove(). • Bug 22143 - Move HTMLMediaElement.playbackQuality to HTMLVideoElement.videoPlaybackQuality.

Version	Comment
13 May 2013	• Bug 21954 - Add [EnforceRange] to appendStream's maxSize parameter. • Bug 21953 - Add NaN handling to appendWindowEnd setter algorithm. • Alphabetize definitions section. • Changed endOfStream('decode') references to make it clear that JavaScript can't intercept these calls. • Fix links for all types in the IDL that are defined in external specifications.
06 May 2013	• Bug 20901 - Remove AbortMode and add AppendMode. • Bug 21911 - Change MediaPlaybackQuality.creationTime to DOMHighResTimeStamp.
02 May 2013	• Reworked ambiguous text in a variety of places. • Added Acknowledgements section.
30 April 2013	• Bug 21822 - Fix 'fire ... event ... at the X attribute' text. • Bug 21819 & 21328 - Remove 'compressed' from coded frame definition.
24 April 2013	• Bug 21796 - Removed issue box from 'Append Error' algorithm. • Bug 21703 - Changed appendWindowEnd to 'unrestricted double'. • Bug 20760 - Adding MediaPlaybackQuality object. • Bug 21536 - Specify the origin of media data appended.
08 April 2013	• Bug 21327 - Crossfade clarifications. • Bug 21334 - Clarified seeking behavoir. • Bug 21326 - Add a note stating some implementations may choose to add fades to/from silence. • Bug 21375 - Clarified decode dependency removal. • Bug 21376 - Replace 100ms limit with 2x last frame duration limit.
26 March 2013	• Bug 21301 - Change timeline references to "media timeline" links. • Bug 19676 - Clarified "fade out coded frames" definition. • Bug 21276 - Convert a few append error scenarios to endOfStream('decode') errors. • Bug 21376 - Changed 'time' to 'decode time' to append sequence definition. • Bug 21374 - Clarify the abort() behavior. • Bug 21373 - Clarified incremental parsing text in segment parser loop. • Bug 21364 - Remove redundant condition from remove overlapped frame step. • Bug 21327 - Clarify what to do with a splice that starts with an audio frame with a duration less than 5ms. • Update to ReSpec 3.1.48
12 March 2013	• Bug 21112 - Add appendWindowStart & appendWindowEnd attributes. • Bug 19676 - Clarify overlapped frame definitions and splice logic. • Bug 21172 - Added coded frame removal and eviction algorithms.

Version	Comment
05 March 2013	• Bug 21170 - Remove 'stream aborted' step from stream append loop algorithm. • Bug 21171 - Added informative note about when addSourceBuffer() might throw an QUOTA_EXCEEDED_ERR exception. • Bug 20901 - Add support for 'continuation' and 'timestampOffset' abort modes. • Bug 21159 - Rename appendArrayBuffer to appendBuffer() and add ArrayBufferView overload. • Bug 21198 - Remove redundant 'closed' readyState checks.
25 February 2013	• Remove Source Buffer Model section since all the behavior is covered by the algorithms now. • Bug 20899 - Remove media segments must start with a random access point requirement. • Bug 21065 - Update example code to use updating attribute instead of old appending attribute.
19 February 2013	• Bug 19676, 20327 - Provide more detail for audio & video splicing. • Bug 20900 - Remove complete access unit constraint. • Bug 20948 - Setting timestampOffset in 'ended' triggers a transition to 'open' • Bug 20952 - Added update event. • Bug 20953 - Move end of append event firing out of segment parser loop. • Bug 21034 - Add steps to fire addtrack and removetrack events.
05 February 2013	• Bug 19676 - Added a note clarifying that the internal timestamp representation doesn't have to be a double. • Added steps to the coded frame processing algorithm to remove old frames when new ones overlap them. • Fix isTypeSupported() return type. • Bug 18933 - Clarify what top-level boxes to ignore for ISO-BMFF. • Bug 18400 - Add a check to avoid creating huge hidden gaps when out-of-order appends occur w/o calling abort().
31 January 2013	• Make remove() asynchronous. • Added steps to various algorithms to throw an INVALID_STATE_ERR exception when async appends or remove() are pending.
30 January 2013	• Remove early abort step on 0-byte appends so the same events fire as a normal append with bytes. • Added definition for 'enough data to ensure uninterrupted playback'. • Updated buffered ranges algorithm to properly compute the ranges for Philip's example.
15 January 2013	Replace setTrackInfo() and getSourceBuffer() with AudioTrack, VideoTrack, and TextTrack extensions.
04 January 2013	• Renamed append() to appendArrayBuffer() and made appending asynchronous. • Added SourceBuffer.appendStream().

Version	Comment
	• Added SourceBuffer.setTrackInfo() methods. • Added issue boxes to relevant sections for outstanding bugs.
14 December 2012	Pubrules, Link Checker, and Markup Validation fixes.
13 December 2012	• Added MPEG-2 Transport Stream section. • Added text to require abort() for out-of-order appends. • Renamed "track buffer" to "decoder buffer". • Redefined "track buffer" to mean the per-track buffers that hold the SourceBuffer media data. • Editorial fixes.
08 December 2012	• Added MediaSource.getSourceBuffer() methods. • Section 2 cleanup.
06 December 2012	• append() now throws a QUOTA_EXCEEDED_ERR when the SourceBuffer is full. • Added unique ID generation text to Initialization Segment Received algorithm. • Remove 2.x subsections that are already covered by algorithm text. • Rework byte stream format text so it doesn't imply that the MediaSource implementation must support all formats supported by the HTMLMediaElement.
28 November 2012	• Added transition to HAVE_METADATA when current playback position is removed. • Added remove() calls to duration change algorithm. • Added MediaSource.isTypeSupported() method. • Remove initialization segments are optional text.
09 November 2012	Converted document to ReSpec.
18 October 2012	Refactored SourceBuffer.append() & added SourceBuffer.remove().
8 October 2012	• Defined what HTMLMediaElement.seekable and HTMLMediaElement.buffered should return. • Updated seeking algorithm to run inside Step 10 of the HTMLMediaElement seeking algorithm. • Removed transition from "ended" to "open" in the seeking algorithm. • Clarified all the event targets.
1 October 2012	Fixed various addsourcebuffer & removesourcebuffer bugs and allow append() in ended state.
13 September 2012	Updated endOfStream() behavior to change based on the value of HTMLMediaElement.readyState.

Version	Comment
24 August 2012	• Added early abort on to duration change algorithm. • Added createObjectURL() IDL & algorithm. • Added Track ID & Track description definitions. • Rewrote start overlap for audio frames text. • Removed rendering silence requirement from section 2.5.
22 August 2012	• Clarified WebM byte stream requirements. • Clarified SourceBuffer.buffered return value. • Clarified addsourcebuffer & removesourcebuffer event targets. • Clarified when media source attaches to the HTMLMediaElement. • Introduced duration change algorithm and update relevant algorithms to use it.
17 August 2012	Minor editorial fixes.
09 August 2012	Change presentation start time to always be 0 instead of using format specific rules about the first media segment appended.
30 July 2012	Added SourceBuffer.timestampOffset and MediaSource.duration.
17 July 2012	Replaced SourceBufferList.remove() with MediaSource.removeSourceBuffer().
02 July 2012	Converted to the object-oriented API
26 June 2012	Converted to Editor's draft.
0.5	Minor updates before proposing to W3C HTML-WG.
0.4	Major revision. Adding source IDs, defining buffer model, and clarifying byte stream formats.
0.3	Minor text updates.
0.2	Updates to reflect initial WebKit implementation.
0.1	Initial Proposal

A. References

A.1 Informative references

[BCP47]
A. Phillips; M. Davis. *Tags for Identifying Languages*. September 2009. IETF Best Current Practice. URL: http://tools.ietf.org/html/bcp47

[FILE-API]
Arun Ranganathan; Jonas Sicking. *File API*. 25 October 2012. W3C Working Draft. URL: http://www.w3.org/TR/2012/WD-FileAPI-20121025

[HTML5]
Robin Berjon et al. *HTML5*. 17 December 2012. W3C Candidate Recommendation. URL: http://www.w3.org/TR/html5/

[STREAMS-API]
Feras Moussa. *Streams API*. 25 October 2012. W3C Editor's Draft. URL: http://dvcs.w3.org/hg/streams-api/raw-file/tip/Overview.htm

[TYPED-ARRAYS]
David Herman, Kenneth Russell. *Typed Arrays* Khronos Working Draft. (Work in progress.) URL: https://www.khronos.org/registry/typedarray/specs/latest/