Media Source Extensions™

W3C Editor's Draft 25 November 2020

This version:

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

Latest published version:

https://www.w3.org/TR/media-source/

Latest editor's draft:

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

Implementation report:

https://tidoust.github.io/media-source-testcoverage/

Editors:

Matthew Wolenetz (Google Inc.)

Mark Watson (Netflix Inc.)

Former editors:

Jerry Smith (Microsoft Corporation) (Until September 2017)

Aaron Colwell (Google Inc.) (Until April 2015)

Adrian Bateman (Microsoft Corporation) (Until April 2015)

Repository:

We are on GitHub

File a bug

Commit history

Mailing list:

public-media-wg@w3.org

Implementation:

Can I use Media Source Extensions?

Test Suite

Test Suite repository

Please check the **errata** for any errors or issues reported since publication.

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 https://www.w3.org/TR/.

The working group maintains a list of all bug reports that the editors have not yet tried to address.

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 track the GitHub repository and take part in the discussions.

This document was published by the Media Working Group as an Editor's Draft.

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.

This document was produced by a group operating under the 1 August 2017 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 15 September 2020 W3C Process Document.

Table of Contents

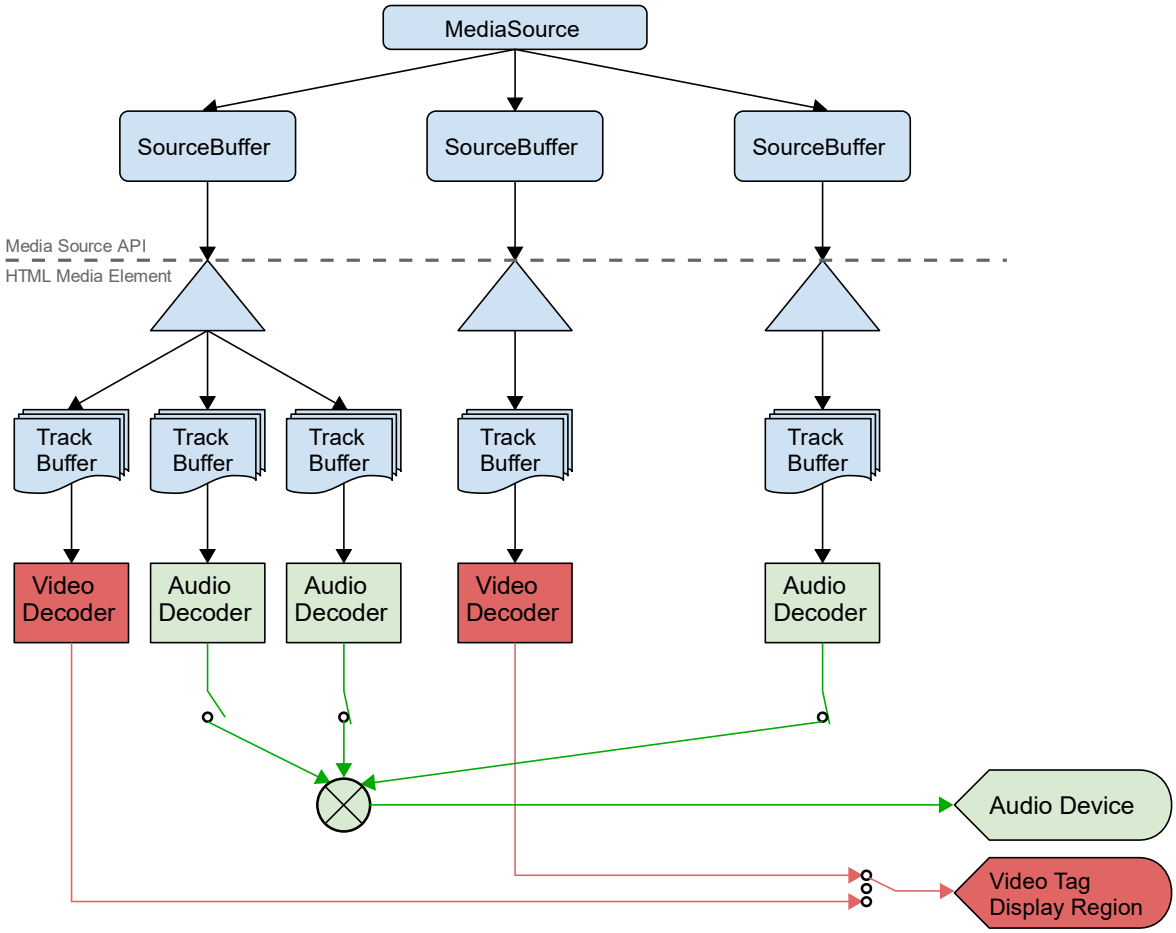
Abstract

This specification extends `HTMLMediaElement` [HTML] 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.

1. Introduction §

This section is non-normative.

This specification allows JavaScript to dynamically construct media streams for <audio> and <video>. It defines a MediaSource object that can serve as a source of media data for an HTMLMediaElement. MediaSource objects have one or more `SourceBuffer` objects. Applications append data segments to the `SourceBuffer` objects, and can adapt the quality of appended data based on system performance and other factors. Data from the `SourceBuffer` objects is managed as track buffers for audio, video and text data that is decoded and played. Byte stream specifications used with these extensions are available in the byte stream format registry [MSE-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 10 milliseconds.

Coded Frame End Timestamp
: The sum of a coded frame presentation timestamp and its coded frame duration. It represents the presentation timestamp that immediately follows the coded frame.

Coded Frame Group

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

Decode Timestamp

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

Initialization Segment

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

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

Media Segment

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

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

MediaSource object URL

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

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

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

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

Parent Media Source

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

Presentation Start Time

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

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

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 *SHOULD* be rendered.

Random Access Point

A position in a media segment where decoding and continuous playback can begin without relying on any previous data in the segment. For video this tends to be the location of I-frames. In the case of audio, most audio frames can be treated as a random access point. Since video tracks tend to have a more sparse distribution of random access points, the location of these points are usually considered the random access points for multiplexed streams.

SourceBuffer byte stream format specification

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

SourceBuffer configuration

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

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

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

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

Track Description

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

Track ID

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

2. `MediaSource` Object §

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.

Each `MediaSource` object has a ***live seekable range*** variable that stores a normalized TimeRanges object. This variable is initialized to an empty `TimeRanges` object when the `MediaSource` object is created, is maintained by `setLiveSeekableRange()` and `clearLiveSeekableRange()`, and is used in HTMLMediaElement Extensions to modify `HTMLMediaElement.seekable` behavior.

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

Terminates playback and signals that a network error has occured.

network

> 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

```
[Exposed=Window]
interface MediaSource : EventTarget {
    constructor();
    readonly        attribute SourceBufferList    sourceBuffers;
    readonly        attribute SourceBufferList    activeSourceBuffers;
    readonly        attribute ReadyState          readyState;
                    attribute unrestricted double duration;
                    attribute EventHandler        onsourceopen;
                    attribute EventHandler        onsourceended;
                    attribute EventHandler        onsourceclose;
    SourceBuffer   addSourceBuffer (DOMString type);
    undefined           removeSourceBuffer (SourceBuffer sourceBuffer);
    undefined           endOfStream (optional EndOfStreamError error);
    undefined           setLiveSeekableRange (double start, double end);
    undefined           clearLiveSeekableRange ();
    static boolean isTypeSupported (DOMString type);
};
```

2.1 Attributes §

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()`.

activeSourceBuffers **of type `SourceBufferList`, readonly**

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

`SourceBuffer` objects in this list *MUST* appear in the same order as they appear in the `sourceBuffers` attribute; e.g., if only sourceBuffers[0] and sourceBuffers[3] are in `activeSourceBuffers`, then activeSourceBuffers[0] *MUST* equal sourceBuffers[0] and activeSourceBuffers[1] *MUST* equal sourceBuffers[3].

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

readyState **of type `ReadyState`, readonly**

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

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

> NOTE
>
> The duration change algorithm will adjust *new duration* higher if there is any currently buffered coded frame with a higher end time.

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

onsourceopen **of type** `EventHandler`
: The event handler for the `sourceopen` event.

onsourceended **of type** `EventHandler`
: The event handler for the `sourceended` event.

onsourceclose **of type** `EventHandler`
: The event handler for the `sourceclose` event.

2.2 Methods §

addSourceBuffer

Adds a new `SourceBuffer` to `sourceBuffers`.

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 an empty string then throw a `TypeError` 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 `NotSupportedError` exception and abort these steps.
3. If the user agent can't handle any more SourceBuffer objects or if creating a SourceBuffer based on *type* would result in an unsupported SourceBuffer configuration, then throw a `QuotaExceededError` exception and abort these steps.

> NOTE
>
> For example, a user agent *MAY* throw a `QuotaExceededError` 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 `InvalidStateError` exception and abort these steps.

5. Create a new `SourceBuffer` object and associated resources.

6. Set the *generate timestamps flag* on the new object to the value in the "Generate Timestamps Flag" column of the byte stream format registry [MSE-REGISTRY] entry that is associated with *type*.

7. ↪ **If the *generate timestamps flag* equals true:**
   Set the `mode` attribute on the new object to `"sequence"`.

   ↪ **Otherwise:**
   Set the `mode` attribute on the new object to `"segments"`.

8. Add the new object to `sourceBuffers` and queue a task to fire a simple event named `addsourcebuffer` at `sourceBuffers`.

9. Return the new object.

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* specifies an object that is not in `sourceBuffers` then throw a `NotFoundError` exception and abort these steps.

2. If the *sourceBuffer*.`updating` attribute equals true, then run the following steps:

   1. Abort the buffer append algorithm if it is 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*.

3. Let *SourceBuffer audioTracks list* equal the `AudioTrackList` object returned by *sourceBuffer*.`audioTracks`.
4. 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. 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. Remove the `AudioTrack` object from the *HTMLMediaElement audioTracks list*.

> NOTE
>
> This should trigger `AudioTrackList` [HTML] logic to queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to the `AudioTrack` object, at the *HTMLMediaElement audioTracks list*. If the `enabled` attribute on the `AudioTrack` object was true at the beginning of this removal step, then this should also trigger `AudioTrackList` [HTML] logic to queue a task to fire a simple event named `change` at the *HTMLMediaElement audioTracks list*

        3. Remove the `AudioTrack` object from the *SourceBuffer audioTracks list*.

> NOTE
>
> This should trigger `AudioTrackList` [HTML] logic to queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to the `AudioTrack` object, at the *SourceBuffer audioTracks list*. If the `enabled` attribute on the `AudioTrack` object was true at the beginning of this removal step, then this should also trigger `AudioTrackList` [HTML] logic to queue a task to fire a simple event named `change` at the *SourceBuffer audioTracks list*

5. Let *SourceBuffer videoTracks list* equal the `VideoTrackList` object returned by *sourceBuffer*.`videoTracks`.

6. 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. 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. Remove the `VideoTrack` object from the *HTMLMediaElement videoTracks list*.

        > NOTE
        >
        > This should trigger `VideoTrackList` [HTML] logic to queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to the `VideoTrack` object, at the *HTMLMediaElement videoTracks list*. If the `selected` attribute on the `VideoTrack` object was true at the beginning of this removal step, then this should also trigger `VideoTrackList` [HTML] logic to queue a task to fire a simple event named `change` at the *HTMLMediaElement videoTracks list*

        3. Remove the `VideoTrack` object from the *SourceBuffer videoTracks list*.

        > NOTE
        >
        > This should trigger `VideoTrackList` [HTML] logic to queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to the `VideoTrack` object, at the *SourceBuffer videoTracks list*. If the `selected` attribute on the `VideoTrack` object was true at the beginning of this removal step, then this should also trigger `VideoTrackList` [HTML] logic to queue a task to fire a simple event named `change` at the *SourceBuffer videoTracks list*

7. Let *SourceBuffer textTracks list* equal the `TextTrackList` object returned by *sourceBuffer*.`textTracks`.

8. 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. 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. Remove the `TextTrack` object from the *HTMLMediaElement textTracks list*.

            > NOTE
            >
            > This should trigger `TextTrackList` [HTML] logic to queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to the `TextTrack` object, at the *HTMLMediaElement textTracks list*. If the `mode` attribute on the `TextTrack` object was `"showing"` or `"hidden"` at the beginning of this removal step, then this should also trigger `TextTrackList` [HTML] logic to queue a task to fire a simple event named `change` at the *HTMLMediaElement textTracks list*.

        3. Remove the `TextTrack` object from the *SourceBuffer textTracks list*.

            > NOTE
            >
            > This should trigger `TextTrackList` [HTML] logic to queue a task to fire a trusted event named `removetrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to the `TextTrack` object, at the *SourceBuffer textTracks list*. If the `mode` attribute on the `TextTrack` object was `"showing"` or `"hidden"` at the beginning of this removal step, then this should also trigger `TextTrackList` [HTML] logic to queue a task to fire a simple event named `change` at the *SourceBuffer textTracks list*.

9. 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`.

10. Remove *sourceBuffer* from `sourceBuffers` and queue a task to fire a simple event named `removesourcebuffer` at the `SourceBufferList` returned by `sourceBuffers`.

11. Destroy all resources for *sourceBuffer*.

`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 `InvalidStateError` exception and abort these steps.
2. If the `updating` attribute equals true on any `SourceBuffer` in `sourceBuffers`, then throw an `InvalidStateError` exception and abort these steps.
3. Run the end of stream algorithm with the *error* parameter set to *error*.

`setLiveSeekableRange`

Updates the *live seekable range* variable used in HTMLMediaElement Extensions to modify `HTMLMediaElement.seekable` behavior.

Parameter	Type	Nullable	Optional	Description
start	`double`	✗	✗	The start of the range, in seconds measured from presentation start time. While set, and if `duration` equals positive Infinity, `HTMLMediaElement.seekable` will return a non-empty TimeRanges object with a lowest range start timestamp no greater than *start*.
end	`double`	✗	✗	The end of range, in seconds measured from presentation start time. While set, and if `duration` equals positive Infinity, `HTMLMediaElement.seekable` will return a non-empty TimeRanges object with a highest range end timestamp no less than *end*.

Return type: `void`

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

1. If the `readyState` attribute is not `"open"` then throw an `InvalidStateError` exception and

abort these steps.

2. If *start* is negative or greater than *end*, then throw a `TypeError` exception and abort these steps.

3. Set *live seekable range* to be a new normalized TimeRanges object containing a single range whose start position is *start* and end position is *end*.

`clearLiveSeekableRange`

Updates the *live seekable range* variable used in HTMLMediaElement Extensions to modify `HTMLMediaElement.seekable` behavior.

No parameters.
Return type: `void`

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

1. If the `readyState` attribute is not `"open"` then throw an `InvalidStateError` exception and abort these steps.

2. If *live seekable range* contains a range, then set *live seekable range* to be a new empty `TimeRanges` object.

`isTypeSupported`, static

Check to see whether the `MediaSource` is capable of creating `SourceBuffer` objects for 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 *SHOULD* 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.

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

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

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

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

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

↪ **Otherwise**

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

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

2.4.2 Detaching from a media element §

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

1. Set the `readyState` attribute to `"closed"`.
2. Update `duration` to NaN.
3. Remove all the `SourceBuffer` objects from `activeSourceBuffers`.
4. Queue a task to fire a simple event named `removesourcebuffer` at `activeSourceBuffers`.
5. Remove all the `SourceBuffer` objects from `sourceBuffers`.
6. Queue a task to fire a simple event named `removesourcebuffer` at `sourceBuffers`.
7. Queue a task to fire a simple event named `sourceclose` at the `MediaSource`.

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

2.4.3 Seeking §

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

1. NOTE

   The media element looks for media segments containing the *new playback position* in each `SourceBuffer` object in `activeSourceBuffers`. Any position within a `TimeRange` in the current value of the `HTMLMediaElement.buffered` attribute has all necessary media segments buffered for that position.

↪ **If *new playback position* is not in any `TimeRange` of `HTMLMediaElement.buffered`**

1. If the `HTMLMediaElement.readyState` attribute is greater than `HAVE_METADATA`, then set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.

   NOTE

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

2. The media element waits until an `appendBuffer()` call causes the coded frame processing algorithm to set the `HTMLMediaElement.readyState` attribute to a value greater than `HAVE_METADATA`.

   NOTE

   The web application can use `buffered` and `HTMLMediaElement.buffered` to determine what the media element needs to resume playback.

↪ **Otherwise**

Continue

> NOTE
>
> If the `readyState` attribute is `"ended"` and the *new playback position* is within a `TimeRange` currently in `HTMLMediaElement.buffered`, then the seek operation must continue to completion here even if one or more currently selected or enabled track buffers' largest range end timestamp is less than *new playback position*. This condition should only occur due to logic in `buffered` when `readyState` is `"ended"`.

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 *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. Changes to `activeSourceBuffers` also cause these steps to run because they affect the conditions that trigger state transitions.

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

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

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

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

1. Abort these steps.

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

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

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

2. Abort these steps.

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

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

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

2. Playback may resume at this point if it was previously suspended by a transition to `HAVE_CURRENT_DATA`.

3. Abort these steps.

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

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

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

2. Playback may resume at this point if it was previously suspended by a transition to `HAVE_CURRENT_DATA`.

3. Abort these steps.

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

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

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

2. Playback is suspended at this point since the media element doesn't have enough data to advance the media timeline.

3. Abort these steps.

2.4.5 Changes to selected/enabled track state §

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

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

1. If the `SourceBuffer` associated with the previously selected video track is not associated with any other enabled tracks, run the following steps:

   1. Remove the `SourceBuffer` from `activeSourceBuffers`.

2. Queue a task to fire a simple event named `removesourcebuffer` at `activeSourceBuffers`

2. If the `SourceBuffer` associated with the newly selected video track is not already in `activeSourceBuffers`, run the following steps:

1. Add the `SourceBuffer` to `activeSourceBuffers`.

2. Queue a task to fire a simple event named `addsourcebuffer` at `activeSourceBuffers`

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

1. Remove the `SourceBuffer` associated with the audio track from `activeSourceBuffers`

2. Queue a task to fire a simple event named `removesourcebuffer` at `activeSourceBuffers`

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

1. Add the `SourceBuffer` associated with the audio track to `activeSourceBuffers`

2. Queue a task to fire a simple event named `addsourcebuffer` at `activeSourceBuffers`

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

1. Remove the `SourceBuffer` associated with the text track from `activeSourceBuffers`

2. Queue a task to fire a simple event named `removesourcebuffer` at `activeSourceBuffers`

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

1. Add the `SourceBuffer` associated with the text track to `activeSourceBuffers`

2. Queue a task to fire a simple event named `addsourcebuffer` at `activeSourceBuffers`

2.4.6 Duration change §

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

1. If the current value of `duration` is equal to *new duration*, then return.
2. If *new duration* is less than the highest presentation timestamp of any buffered coded frames for all `SourceBuffer` objects in `sourceBuffers`, then throw an `InvalidStateError` exception and abort these steps.

   > NOTE
   >
   > Duration reductions that would truncate currently buffered media are disallowed. When truncation is necessary, use `remove()` to reduce the buffered range before updating `duration`.

3. Let *highest end time* be the largest track buffer ranges end time across all the track buffers across all `SourceBuffer` objects in `sourceBuffers`.
4. If *new duration* is less than *highest end time*, then

   > NOTE
   >
   > This condition can occur because the coded frame removal algorithm preserves coded frames that start before the start of the removal range.

   1. Update *new duration* to equal *highest end time*.
5. Update `duration` to *new duration*.
6. Update the `media duration` to *new duration* and run the HTMLMediaElement duration change algorithm.

2.4.7 End of stream algorithm §

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

1. Change the `readyState` attribute value to `"ended"`.
2. Queue a task to fire a simple event named `sourceended` at the `MediaSource`.
3. ↪ **If *error* is not set**

   1. Run the duration change algorithm with *new duration* set to the largest track buffer ranges end time across all the track buffers across all `SourceBuffer` objects

in `sourceBuffers`.

> NOTE
>
> This allows the duration to properly reflect the end of the appended media segments. For example, if the duration was explicitly set to 10 seconds and only media segments for 0 to 5 seconds were appended before endOfStream() was called, then the duration will get updated to 5 seconds.

2. Notify the media element that it now has all of the media data.

↪ **If *error* is set to `"network"`**

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

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

↪ **If *error* is set to `"decode"`**

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

↪ **If the `HTMLMediaElement.readyState` attribute is greater than `HAVE_NOTHING`**
Run the media data is corrupted steps of the resource fetch algorithm's media data processing steps list.

3. *SourceBuffer* Object §

WebIDL

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

```
};
```

Enumeration description

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

WebIDL

```
[Exposed=Window]
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;
                    attribute EventHandler        onupdatestart;
                    attribute EventHandler        onupdate;
                    attribute EventHandler        onupdateend;
                    attribute EventHandler        onerror;
                    attribute EventHandler        onabort;
    undefined appendBuffer (BufferSource data);
    undefined abort ();
    undefined remove (double start, unrestricted double end);
};
```

3.1 Attributes §

mode **of type** `AppendMode`

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

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

On setting, run the following steps:

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

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

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

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

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

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

1. If this object has been removed from the `sourceBuffers` attribute of the parent media source then throw an `InvalidStateError` exception and abort these steps.
2. Let *highest end time* be the largest track buffer ranges end time across all the track buffers managed by this `SourceBuffer` object.
3. Let *intersection ranges* equal a `TimeRange` object containing a single range from 0 to *highest end time*.
4. For each audio and video track buffer managed by this `SourceBuffer`, run the following steps:

   > NOTE
   >
   > Text track-buffers are included in the calculation of *highest end time*, above, but excluded from the buffered range calculation here. They are not necessarily continuous, nor should any discontinuity within them trigger playback stall when the other media tracks are continuous over the same time range.

   1. Let *track ranges* equal the track buffer ranges for the current track buffer.
   2. If `readyState` is `"ended"`, then set the end time on the last range in *track ranges* to *highest end time*.
   3. Let *new intersection ranges* equal the intersection between the *intersection ranges* and the *track ranges*.
   4. Replace the ranges in *intersection ranges* with the *new intersection ranges*.
5. If *intersection ranges* does not contain the exact same range information as the current value of this attribute, then update the current value of this attribute to *intersection ranges*.
6. Return the current value of this attribute.

`timestampOffset` of type `double`

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

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

On setting, run the following steps:

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

audioTracks **of type `AudioTrackList`, readonly**

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

videoTracks **of type `VideoTrackList`, readonly**

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

textTracks **of type `TextTrackList`, readonly**

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

appendWindowStart **of type `double`**

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

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

On setting, run the following steps:

1. If this object has been removed from the `sourceBuffers` attribute of the parent media source, then throw an `InvalidStateError` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `InvalidStateError` exception and abort these steps.

3. If the new value is less than 0 or greater than or equal to `appendWindowEnd` then throw a `TypeError` exception and abort these steps.

4. Update the attribute to the new value.

appendWindowEnd **of type** `unrestricted double`

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

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

On setting, run the following steps:

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

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

3. If the new value equals NaN, then throw a `TypeError` and abort these steps.

4. If the new value is less than or equal to `appendWindowStart` then throw a `TypeError` exception and abort these steps.

5. Update the attribute to the new value.

onupdatestart **of type** `EventHandler`

The event handler for the `updatestart` event.

onupdate **of type** `EventHandler`

The event handler for the `update` event.

onupdateend **of type** `EventHandler`

The event handler for the `updateend` event.

onerror **of type** `EventHandler`

The event handler for the `error` event.

onabort **of type** `EventHandler`

The event handler for the `abort` event.

3.2 Methods §

appendBuffer

Appends the segment data in an `BufferSource`[WEBIDL] to the source buffer.

Parameter	Type	Nullable	Optional	Description
data	`BufferSource`	✗	✗	

Return type: `void`

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

1. Run the prepare append algorithm.
2. Add *data* to the end of the *input buffer*.
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 buffer append algorithm.

`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 `InvalidStateError` exception and abort these steps.
2. If the `readyState` attribute of the parent media source is not in the `"open"` state then throw an `InvalidStateError` exception and abort these steps.
3. If the range removal algorithm is running, then throw an `InvalidStateError` exception and abort these steps.
4. If the `updating` attribute equals true, then run the following steps:
    1. Abort the buffer append algorithm if it is 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.

5. Run the reset parser state algorithm.
6. Set `appendWindowStart` to the presentation start time.
7. Set `appendWindowEnd` to positive Infinity.

`remove`

Removes media for a specific time range.

Parameter	Type	Nullable	Optional	Description
start	`double`	✗	✗	The start of the removal range, in seconds measured from presentation start time.
end	`unrestricted double`	✗	✗	The end of the removal range, in seconds measured from presentation start time.

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 `InvalidStateError` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `InvalidStateError` exception and abort these steps.
3. If `duration` equals NaN, then throw a `TypeError` exception and abort these steps.
4. If *start* is negative or greater than `duration`, then throw a `TypeError` exception and abort these steps.
5. If *end* is less than or equal to *start* or *end* equals NaN, then throw a `TypeError` exception and abort these steps.
6. 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.
7. Run the range removal algorithm with *start* and *end* as the start and end of the removal range.

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 end timestamp*** variable that stores the highest coded frame end timestamp across all coded frames in the current coded frame group that were appended to this track buffer. 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.

Each track buffer has a ***track buffer ranges*** variable that represents the presentation time ranges occupied by the coded frames currently stored in the track buffer.

> NOTE
>
> For track buffer ranges, these presentation time ranges are based on presentation timestamps, frame durations, and potentially coded frame group start times for coded frame groups across track buffers in a muxed `SourceBuffer`.

For specification purposes, this information is treated as if it were stored in a normalized TimeRanges object. Intersected track buffer ranges are used to report `HTMLMediaElement.buffered`, and *MUST* therefore support uninterrupted playback within each range of `HTMLMediaElement.buffered`.

> NOTE
>
> These coded frame group start times differ slightly from those mentioned in the coded frame processing algorithm in that they are the earliest presentation timestamp across all track buffers following a discontinuity. Discontinuities can occur within the coded frame processing algorithm or result from the coded frame removal algorithm, regardless of `mode`. The threshold for determining disjointness of track buffer ranges is implementation-specific. For example, to reduce unexpected playback stalls, implementations *MAY* approximate the coded frame processing algorithm's discontinuity detection logic by coalescing adjacent ranges separated by a gap smaller than 2 times the maximum frame duration buffered so far in this track buffer. Implementations *MAY* also use coded frame group start times as range start times across track buffers in a muxed `SourceBuffer` to further reduce unexpected playback stalls.

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

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

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

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

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

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

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

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

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

1. *Loop Top:* If the *input buffer* is empty, then jump to the *need more data* step below.
2. If the *input buffer* contains bytes that violate the SourceBuffer byte stream format specification, then run the append error algorithm and abort this algorithm.
3. Remove any bytes that the byte stream format specifications say *MUST* be ignored from the start of the *input buffer*.

4. If the *append state* equals WAITING_FOR_SEGMENT, then run the following steps:

    1. If the beginning of the *input buffer* indicates the start of an initialization segment, set the *append state* to PARSING_INIT_SEGMENT.

    2. If the beginning of the *input buffer* indicates the start of a media segment, set *append state* to PARSING_MEDIA_SEGMENT.

    3. Jump to the *loop top* step above.

5. If the *append state* equals PARSING_INIT_SEGMENT, then run the following steps:

    1. If the *input buffer* does not contain a complete initialization segment yet, then jump to the *need more data* step below.

    2. Run the initialization segment received algorithm.

    3. Remove the initialization segment bytes from the beginning of the *input buffer*.

    4. Set *append state* to WAITING_FOR_SEGMENT.

    5. Jump to the *loop top* step above.

6. If the *append state* equals PARSING_MEDIA_SEGMENT, then run the following steps:

    1. If the *first initialization segment received flag* is false, then run the append error algorithm and abort this algorithm.

    2. If the *input buffer* contains one or more 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.

    3. If this `SourceBuffer` is full and cannot accept more media data, then set the *buffer full flag* to true.

    4. If the *input buffer* does not contain a complete media segment, then jump to the *need more data* step below.

5. Remove the media segment bytes from the beginning of the *input buffer*.
6. Set *append state* to WAITING_FOR_SEGMENT.
7. Jump to the *loop top* step above.

7. *Need more data:* Return control to the calling algorithm.

3.5.2 Reset Parser State §

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

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

3.5.3 Append Error Algorithm §

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

1. Run the reset parser state algorithm.
2. Set the `updating` attribute to false.
3. Queue a task to fire a simple event named `error` at this `SourceBuffer` object.
4. Queue a task to fire a simple event named `updateend` at this `SourceBuffer` object.

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

3.5.4 Prepare Append Algorithm §

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

1. If the `SourceBuffer` has been removed from the `sourceBuffers` attribute of the parent media source then throw an `InvalidStateError` exception and abort these steps.
2. If the `updating` attribute equals true, then throw an `InvalidStateError` exception and abort these steps.
3. If the `HTMLMediaElement.error` attribute is not null, then throw an `InvalidStateError` exception and abort these steps.
4. If the `readyState` attribute of the parent media source is in the `"ended"` state then run the following steps:
    1. Set the `readyState` attribute of the parent media source to `"open"`
    2. Queue a task to fire a simple event named `sourceopen` at the parent media source.
5. Run the coded frame eviction algorithm.
6. If the *buffer full flag* equals true, then throw a `QuotaExceededError` exception and abort these step.

> NOTE
>
> This is the signal that the implementation was unable to evict enough data to accommodate the append or the append is too big. The web application *SHOULD* use `remove()` to explicitly free up space and/or reduce the size of the append.

3.5.5 Buffer Append Algorithm §

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

1. Run the segment parser loop algorithm.
2. If the segment parser loop algorithm in the previous step was aborted, then abort this algorithm.

3. Set the `updating` attribute to false.
4. Queue a task to fire a simple event named `update` at this `SourceBuffer` object.
5. Queue a task to fire a simple event named `updateend` at this `SourceBuffer` object.

3.5.6 Range Removal §

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

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

3.5.7 Initialization Segment Received §

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

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

1. Update the `duration` attribute if it currently equals NaN:

↪ **If the initialization segment contains a duration:**
Run the duration change algorithm with *new duration* set to the duration in the initialization segment.

↪ **Otherwise:**
Run the duration change algorithm with *new duration* set to positive Infinity.

2. If the initialization segment has no audio, video, or text tracks, then run the append error algorithm and abort these steps.

3. If the *first initialization segment received flag* is true, then run the following steps:
    1. Verify the following properties. If any of the checks fail then run the append error algorithm and abort these steps.
        - 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 (e.g., 2 audio tracks), then the Track IDs match the ones in the first initialization segment.
    2. Add the appropriate track descriptions from this initialization segment to each of the track buffers.
    3. Set the *need random access point flag* on all track buffers to true.

4. Let *active track flag* equal false.

5. If the *first initialization segment received flag* is false, then run the following steps:
    1. If the initialization segment contains tracks with codecs the user agent does not support, then run the append error algorithm and abort these steps.

        > NOTE
        >
        > User agents *MAY* consider codecs, that would otherwise be supported, as "not supported" here if the codecs were not specified in the *type* parameter passed to `addSourceBuffer()`.
        > For example, MediaSource.isTypeSupported('video/webm;codecs="vp8,vorbis"') may return true, but if `addSourceBuffer()` was called with 'video/webm;codecs="vp8"' and a Vorbis track appears in the initialization segment, then the user agent *MAY* use this step to trigger a decode error.

    2. For each audio track in the initialization segment, run following steps:

1. Let *audio byte stream track ID* be the Track ID for the current track being processed.
2. Let *audio language* be a BCP 47 language tag for the language specified in the initialization segment for this track or an empty string if no language info is present.
3. If *audio language* equals the 'und' BCP 47 value, then assign an empty string to *audio language*.
4. Let *audio label* be a label specified in the initialization segment for this track or an empty string if no label info is present.
5. Let *audio kinds* be a sequence of kind strings specified in the initialization segment for this track or a sequence with a single empty string element in it if no kind information is provided.
6. For each value in *audio kinds*, run the following steps:
    1. Let *current audio kind* equal the value from *audio kinds* for this iteration of the loop.
    2. Let *new audio track* be a new `AudioTrack` object.
    3. Generate a unique ID and assign it to the `id` property on *new audio track*.
    4. Assign *audio language* to the `language` property on *new audio track*.
    5. Assign *audio label* to the `label` property on *new audio track*.
    6. Assign *current audio kind* to the `kind` property on *new audio track*.
    7. If `audioTracks.length` equals 0, then run the following steps:
        1. Set the `enabled` property on *new audio track* to true.
        2. Set *active track flag* to true.
    8. Add *new audio track* to the `audioTracks` attribute on this `SourceBuffer` object.

        > NOTE
        >
        > This should trigger `AudioTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to *new audio track*, at the `AudioTrackList` object referenced by the `audioTracks` attribute on this `SourceBuffer` object.

    9. Add *new audio track* to the `audioTracks` attribute on the HTMLMediaElement.

> NOTE
>
> This should trigger `AudioTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to *new audio track*, at the `AudioTrackList` object referenced by the `audioTracks` attribute on the HTMLMediaElement.

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

3. For each video track in the initialization segment, run following steps:
    1. Let *video byte stream track ID* be the Track ID for the current track being processed.
    2. Let *video language* be a BCP 47 language tag for the language specified in the initialization segment for this track or an empty string if no language info is present.
    3. If *video language* equals the 'und' BCP 47 value, then assign an empty string to *video language*.
    4. Let *video label* be a label specified in the initialization segment for this track or an empty string if no label info is present.
    5. Let *video kinds* be a sequence of kind strings specified in the initialization segment for this track or a sequence with a single empty string element in it if no kind information is provided.
    6. For each value in *video kinds*, run the following steps:
        1. Let *current video kind* equal the value from *video kinds* for this iteration of the loop.
        2. Let *new video track* be a new `VideoTrack` object.
        3. Generate a unique ID and assign it to the `id` property on *new video track*.
        4. Assign *video language* to the `language` property on *new video track*.
        5. Assign *video label* to the `label` property on *new video track*.
        6. Assign *current video kind* to the `kind` property on *new video track*.
        7. If `videoTracks.length` equals 0, then run the following steps:
            1. Set the `selected` property on *new video track* to true.

2. Set *active track flag* to true.

8. Add *new video track* to the `videoTracks` attribute on this `SourceBuffer` object.

> NOTE
>
> This should trigger `VideoTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to *new video track*, at the `VideoTrackList` object referenced by the `videoTracks` attribute on this `SourceBuffer` object.

9. Add *new video track* to the `videoTracks` attribute on the HTMLMediaElement.

> NOTE
>
> This should trigger `VideoTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to *new video track*, at the `VideoTrackList` object referenced by the `videoTracks` attribute on the HTMLMediaElement.

7. Create a new track buffer to store coded frames for this track.

8. Add the track description for this track to the track buffer.

4. For each text track in the initialization segment, run following steps:

1. Let *text byte stream track ID* be the Track ID for the current track being processed.

2. Let *text language* be a BCP 47 language tag for the language specified in the initialization segment for this track or an empty string if no language info is present.

3. If *text language* equals the 'und' BCP 47 value, then assign an empty string to *text language*.

4. Let *text label* be a label specified in the initialization segment for this track or an empty string if no label info is present.

5. Let *text kinds* be a sequence of kind strings specified in the initialization segment for this track or a sequence with a single empty string element in it if no kind information is provided.

6. For each value in *text kinds*, run the following steps:
    1. Let *current text kind* equal the value from *text kinds* for this iteration of the loop.
    2. Let *new text track* be a new `TextTrack` object.
    3. Generate a unique ID and assign it to the `id` property on *new text track*.
    4. Assign *text language* to the `language` property on *new text track*.
    5. Assign *text label* to the `label` property on *new text track*.
    6. Assign *current text kind* to the `kind` property on *new text track*.
    7. Populate the remaining properties on *new text track* with the appropriate information from the initialization segment.
    8. If the `mode` property on *new text track* equals `"showing"` or `"hidden"`, then set *active track flag* to true.
    9. Add *new text track* to the `textTracks` attribute on this `SourceBuffer` object.

        > NOTE
        >
        > This should trigger `TextTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to *new text track*, at the `TextTrackList` object referenced by the `textTracks` attribute on this `SourceBuffer` object.

    10. Add *new text track* to the `textTracks` attribute on the HTMLMediaElement.

        > NOTE
        >
        > This should trigger `TextTrackList` [HTML] logic to queue a task to fire a trusted event named `addtrack`, that does not bubble and is not cancelable, and that uses the `TrackEvent` interface, with the `track` attribute initialized to *new text track*, at the `TextTrackList` object referenced by the `textTracks` attribute on the HTMLMediaElement.

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

5. If *active track flag* equals true, then run the following steps:
    1. Add this `SourceBuffer` to `activeSourceBuffers`.

2. Queue a task to fire a simple event named `addsourcebuffer` at `activeSourceBuffers`

6. Set *first initialization segment received flag* to true.

6. If the `HTMLMediaElement.readyState` attribute is `HAVE_NOTHING`, then run the following steps:

1. If one or more objects in `sourceBuffers` have *first initialization segment received flag* set to false, then abort these steps.

2. Set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.

> NOTE
>
> Per `HTMLMediaElement ready states` [HTML] logic, `HTMLMediaElement.readyState` changes may trigger events on the HTMLMediaElement. This particular transition should trigger HTMLMediaElement logic to queue a task to fire a simple event named `loadedmetadata` at the media element.

7. If the *active track flag* equals true and the `HTMLMediaElement.readyState` attribute is greater than `HAVE_CURRENT_DATA`, then set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA`.

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

3.5.8 Coded Frame Processing §

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

1. For each coded frame in the media segment run the following steps:

1. *Loop Top:*

↪ **If *generate timestamps flag* equals true:**

1. Let *presentation timestamp* equal 0.

2. Let *decode timestamp* equal 0.

↳ **Otherwise:**

1. Let *presentation timestamp* be a double precision floating point representation of the coded frame's presentation timestamp in seconds.

   > NOTE
   >
   > Special processing may be needed to determine the presentation and decode timestamps for timed text frames since this information may not be explicitly present in the underlying format or may be dependent on the order of the frames. Some metadata text tracks, like MPEG2-TS PSI data, may only have implied timestamps. Format specific rules for these situations *SHOULD* be in the byte stream format specifications or in separate extension specifications.

2. Let *decode timestamp* be a double precision floating point representation of the coded frame's decode timestamp in seconds.

   > NOTE
   >
   > Implementations don't have to internally store timestamps in a double precision floating point representation. This representation is used here because it is the represention for timestamps in the HTML spec. The intention here is to make the behavior clear without adding unnecessary complexity to the algorithm to deal with the fact that adding a timestampOffset may cause a timestamp rollover in the underlying timestamp representation used by the byte stream format. Implementations can use any internal timestamp representation they wish, but the addition of timestampOffset *SHOULD* behave in a similar manner to what would happen if a double precision floating point representation was used.

2. Let *frame duration* be a double precision floating point representation of the coded frame's duration in seconds.

3. If `mode` equals `"sequence"` and *group start timestamp* is set, then run the following steps:
   1. Set `timestampOffset` equal to *group start timestamp* - *presentation timestamp*.
   2. Set *group end timestamp* equal to *group start timestamp*.
   3. Set the *need random access point flag* on all track buffers to true.

4. Unset *group start timestamp*.

4. If `timestampOffset` is not 0, then run the following steps:

    1. Add `timestampOffset` to the *presentation timestamp*.

    2. Add `timestampOffset` to the *decode timestamp*.

5. Let *track buffer* equal the track buffer that the coded frame will be added to.

6. ↪ **If *last decode timestamp* for *track buffer* is set and *decode timestamp* is less than *last decode timestamp*:**
   OR

   ↪ **If *last decode timestamp* for *track buffer* is set and the difference between *decode timestamp* and *last decode timestamp* is greater than 2 times *last frame duration*:**

    1. ↪ **If `mode` equals `"segments"`:**
       Set *group end timestamp* to *presentation timestamp*.

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

    2. Unset the *last decode timestamp* on all track buffers.

    3. Unset the *last frame duration* on all track buffers.

    4. Unset the *highest end timestamp* on all track buffers.

    5. Set the *need random access point flag* on all track buffers to true.

    6. Jump to the *Loop Top* step above to restart processing of the current coded frame.

   ↪ **Otherwise:**
   Continue.

7. Let *frame end timestamp* equal the sum of *presentation timestamp* and *frame duration*.

8. If *presentation timestamp* is less than `appendWindowStart`, then set the *need random access point flag* to true, drop the coded frame, and jump to the top of the loop to start processing the next coded frame.

> NOTE
>
> Some implementations *MAY* choose to collect some of these coded frames with *presentation timestamp* less than `appendWindowStart` and use them to generate a splice at the first coded frame that has a presentation timestamp greater than or equal to `appendWindowStart` even if that frame is not a random access point. Supporting this requires multiple decoders or faster than real-time decoding so for now this behavior will not be a normative requirement.

9. If *frame end timestamp* is greater than `appendWindowEnd`, then set the *need random access point flag* to true, drop the coded frame, and jump to the top of the loop to start processing the next coded frame.

> NOTE
>
> Some implementations *MAY* choose to collect coded frames with *presentation timestamp* less than `appendWindowEnd` and *frame end timestamp* greater than `appendWindowEnd` and use them to generate a splice across the portion of the collected coded frames within the append window at time of collection, and the beginning portion of later processed frames which only partially overlap the end of the collected coded frames. Supporting this requires multiple decoders or faster than real-time decoding so for now this behavior will not be a normative requirement. In conjunction with collecting coded frames that span `appendWindowStart`, implementations *MAY* thus support gapless audio splicing.

10. 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.
11. Let *spliced audio frame* be an unset variable for holding audio splice information
12. Let *spliced timed text frame* be an unset variable for holding timed text splice information
13. If *last decode timestamp* for *track buffer* is unset and *presentation timestamp* falls within the presentation interval of a coded frame in *track buffer*, then run the following steps:
    1. Let *overlapped frame* be the coded frame in *track buffer* that matches the condition above.
    2. ↪ **If *track buffer* contains audio coded frames:**

Run the audio splice frame algorithm and if a splice frame is returned, assign it to *spliced audio frame*.

↪ **If *track buffer* contains video coded frames:**

1. Let *remove window timestamp* equal the *overlapped frame* presentation timestamp plus 1 microsecond.
2. If the *presentation timestamp* is less than the *remove window timestamp*, then remove *overlapped frame* from *track buffer*.

> NOTE
>
> This is to compensate for minor errors in frame timestamp computations that can appear when converting back and forth between double precision floating point numbers and rationals. This tolerance allows a frame to replace an existing one as long as it is within 1 microsecond of the existing frame's start time. Frames that come slightly before an existing frame are handled by the removal step below.

↪ **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*.

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

↪ **If *highest end timestamp* for *track buffer* is not set:**

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

↪ **If *highest end timestamp* for *track buffer* is set and less than or equal to *presentation timestamp*:**

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

15. Remove all possible decoding dependencies on the coded frames removed in the previous two steps by removing all coded frames from *track buffer* between those frames removed in the previous two steps and the next random access point after those removed frames.

> NOTE
>
> Removing all coded frames until the next random access point is a conservative estimate of the decoding dependencies since it assumes all frames between the removed frames and the next random access point depended on the frames that were removed.

16. ↪ **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*.

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

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

19. If *highest end timestamp* for *track buffer* is unset or *frame end timestamp* is greater than *highest end timestamp*, then set *highest end timestamp* for *track buffer* to *frame end timestamp*.

    > NOTE
    >
    > The greater than check is needed because bidirectional prediction between coded frames can cause *presentation timestamp* to not be monotonically increasing even though the decode timestamps are monotonically increasing.

20. If *frame end timestamp* is greater than *group end timestamp*, then set *group end timestamp* equal to *frame end timestamp*.

21. If *generate timestamps flag* equals true, then set `timestampOffset` equal to *frame end timestamp*.

2. If the `HTMLMediaElement.readyState` attribute is `HAVE_METADATA` and the new coded frames cause `HTMLMediaElement.buffered` to have a `TimeRange` for the current playback position, then set the `HTMLMediaElement.readyState` attribute to `HAVE_CURRENT_DATA`.

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

3. If the `HTMLMediaElement.readyState` attribute is `HAVE_CURRENT_DATA` and the new coded frames cause `HTMLMediaElement.buffered` to have a `TimeRange` that includes the current playback position and some time beyond the current playback position, then set the `HTMLMediaElement.readyState` attribute to `HAVE_FUTURE_DATA`.

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

4. If the `HTMLMediaElement.readyState` attribute is `HAVE_FUTURE_DATA` and the new coded frames cause `HTMLMediaElement.buffered` to have a `TimeRange` that includes the current playback position and enough data to ensure uninterrupted playback, then set the `HTMLMediaElement.readyState` attribute to `HAVE_ENOUGH_DATA`.

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

5. If the media segment contains data beyond the current `duration`, then run the duration change algorithm with *new duration* set to the maximum of the current duration and the *group end timestamp*.

3.5.9 Coded Frame Removal Algorithm §

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

1. Let *start* be the starting presentation timestamp for the removal range.
2. Let *end* be the end presentation timestamp for the removal range.
3. For each track buffer in this source buffer, run the following steps:

1. Let *remove end timestamp* be the current value of `duration`

2. If this track buffer has a random access point timestamp that is greater than or equal to *end*, then update *remove end timestamp* to that random access point timestamp.

   > NOTE
   >
   > Random access point timestamps can be different across tracks because the dependencies between coded frames within a track are usually different than the dependencies in another track.

3. Remove all media data, from this track buffer, that contain starting timestamps greater than or equal to *start* and less than the *remove end timestamp*.
   1. For each removed frame, if the frame has a decode timestamp equal to the *last decode timestamp* for the frame's track, run the following steps:

      ↳ **If `mode` equals `"segments"`:**
      Set *group end timestamp* to presentation timestamp.

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

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

4. Remove all possible decoding dependencies on the coded frames removed in the previous step by removing all coded frames from this track buffer between those frames removed in the previous step and the next random access point after those removed frames.

   > NOTE
   >
   > Removing all coded frames until the next random access point is a conservative estimate of the decoding dependencies since it assumes all frames between the removed frames and the next random access point depended on the frames that were removed.

5. If this object is in `activeSourceBuffers`, the current playback position is greater than or equal to *start* and less than the *remove end timestamp*, and `HTMLMediaElement.readyState` is greater than `HAVE_METADATA`, then set the `HTMLMediaElement.readyState` attribute to `HAVE_METADATA` and stall playback.

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

> NOTE
>
> This transition occurs because media data for the current position has been removed. Playback cannot progress until media for the current playback position is appended or the selected/enabled tracks change.

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

3.5.10 Coded Frame Eviction Algorithm §

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

1. Let *new data* equal the data that is about to be appended to this SourceBuffer.
2. If the *buffer full flag* equals false, then abort these steps.
3. Let *removal ranges* equal a list of presentation time ranges that can be evicted from the presentation to make room for the *new data*.

> NOTE
>
> Implementations *MAY* use different methods for selecting *removal ranges* so web applications *SHOULD NOT* depend on a specific behavior. The web application can use the `buffered` attribute to observe whether portions of the buffered data have been evicted.

4. For each range in *removal ranges*, run the coded frame removal algorithm with *start* and *end* equal to the removal range start and end timestamp respectively.

3.5.11 Audio Splice Frame Algorithm §

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

1. Let *track buffer* be the track buffer that will contain the splice.
2. Let *new coded frame* be the new coded frame, that is being added to *track buffer*, which triggered the need for a splice.
3. Let *presentation timestamp* be the presentation timestamp for *new coded frame*
4. Let *decode timestamp* be the decode timestamp for *new coded frame*.
5. Let *frame duration* be the coded frame duration of *new coded frame*.
6. Let *overlapped frame* be the coded frame in *track buffer* with a presentation interval that contains *presentation timestamp*.
7. Update *presentation timestamp* and *decode timestamp* to the nearest audio sample timestamp based on sample rate of the audio in *overlapped frame*. If a timestamp is equidistant from both audio sample timestamps, then use the higher timestamp (e.g., `floor(x * sample_rate + 0.5) / sample_rate`).

   > NOTE
   >
   > For example, given the following values:
   >
   > - The presentation timestamp of *overlapped frame* equals 10.
   > - The sample rate of *overlapped frame* equals 8000 Hz
   > - *presentation timestamp* equals 10.01255
   > - *decode timestamp* equals 10.01255
   >
   > *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*.

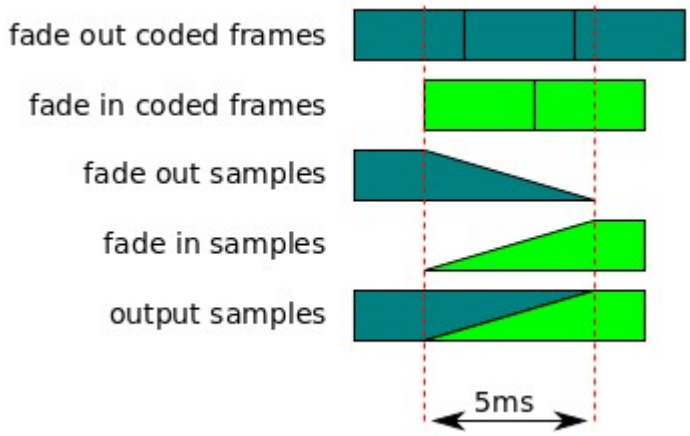
15. For each sample between *splice timestamp* and *splice end timestamp*, compute the sum of a sample from *fade out samples* and the corresponding sample in *fade in samples* and store the result in *output samples*.

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

```
[Exposed=Window]
interface SourceBufferList : EventTarget {
    readonly        attribute unsigned long length;
                    attribute EventHandler  onaddsourcebuffer;
                    attribute EventHandler  onremovesourcebuffer;
    getter SourceBuffer (unsigned long index);
```

```
};
```

4.1 Attributes §

length **of type** `unsigned long`**, readonly**
: Indicates the number of `SourceBuffer` objects in the list.

onaddsourcebuffer **of type** `EventHandler`
: The event handler for the `addsourcebuffer` event.

onremovesourcebuffer **of type** `EventHandler`
: The event handler for the `removesourcebuffer` event.

4.2 Methods §

getter
: 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: `SourceBuffer`

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. URL Object Extensions §

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

WebIDL

```
[Exposed=Window]
partial interface URL {
    static DOMString createObjectURL (MediaSource mediaSource);
};
```

5.1 Methods §

***createObjectURL*, static**

Creates URLs for `MediaSource` objects.

> NOTE
>
> This algorithm is intended to mirror the behavior of the createObjectURL()[FILE-API] method, which does not auto-revoke the created URL. Web authors are encouraged to use revokeObjectURL()[FILE-API] for any MediaSource object URL that is no longer needed for attachment to a media element.

Parameter	Type	Nullable	Optional	Description
mediaSource	`MediaSource`	✗	✗	

Return type: `DOMString`

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

1. Return a unique MediaSource object URL that can be used to dereference the *mediaSource* argument.

6. 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:**

1. If *live seekable range* is not empty:
    1. Let *union ranges* be the union of *live seekable range* and the `HTMLMediaElement.buffered` attribute.
    2. Return a single range with a start time equal to the earliest start time in *union ranges* and an end time equal to the highest end time in *union ranges* and abort these steps.
2. If the `HTMLMediaElement.buffered` attribute returns an empty `TimeRanges` object, then return an empty `TimeRanges` object and abort these steps.
3. 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 static normalized TimeRanges object based on the following steps.

1. Let *intersection ranges* equal an empty `TimeRanges` object.
2. If `activeSourceBuffers`.length does not equal 0 then run the following steps:
    1. Let *active ranges* be the ranges returned by `buffered` for each `SourceBuffer` object in `activeSourceBuffers`.
    2. Let *highest end time* be the largest range end time in the *active ranges*.
    3. Let *intersection ranges* equal a `TimeRange` object containing a single range from 0 to *highest end time*.
    4. 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 intersection between the *intersection ranges* and the *source ranges*.
4. Replace the ranges in *intersection ranges* with the *new intersection ranges*.

3. If the current value of this attribute has not been set by this algorithm or *intersection ranges* does not contain the exact same range information as the current value of this attribute, then update the current value of this attribute to *intersection ranges*.
4. Return the current value of this attribute.

7. AudioTrack Extensions §

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

WebIDL

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

Attributes §

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.

8. VideoTrack Extensions §

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

WebIDL

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

Attributes §

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.

9. TextTrack Extensions §

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

WebIDL

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

Attributes §

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. Byte Stream Formats §

The bytes provided through `appendBuffer()` 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 [MSE-REGISTRY] provides mappings between a MIME type that may be passed to `addSourceBuffer()` or `isTypeSupported()` and the byte stream format expected by a `SourceBuffer` created with that MIME type. Implementations are encouraged to register mappings for byte stream formats they support to facilitate interoperability. The byte stream format registry [MSE-REGISTRY] is the authoritative source for these mappings. If an implementation claims to support a MIME type listed in the registry, its `SourceBuffer` implementation *MUST* conform to the byte stream format specification listed in the registry entry.

> 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.
- A byte stream format *SHOULD* provide references for sourcing `AudioTrack`, `VideoTrack`, and `TextTrack` attribute values from data in initialization segments.

  > NOTE
  >
  > If the byte stream format covers a format similar to one covered in the in-band tracks spec [INBANDTRACKS], then it *SHOULD* try to use the same attribute mappings so that Media Source Extensions playback and non-Media Source Extensions playback provide the same track information.

- 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 append error algorithm 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 agents *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 append error algorithm 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.

11. Conformance §

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

12. 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 e
    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>
```

13. 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, Jer Noble, Joe Steele, John Simmons, Kevin Streeter, Mark Vickers, Matt Ward, Matthew Gregan, Michael Thornburgh, Philip Jägenstedt, Pierre Lemieux, Ralph Giles, Steven Robertson, and Tatsuya Igarashi for their contributions to this specification.

A. VideoPlaybackQuality §

This section is non-normative.

The video playback quality metrics described in previous revisions of this specification (e.g., sections 5 and 10 of the Candidate Recommendation) are now being developed as part of [MEDIA-PLAYBACK-QUALITY]. Some implementations may have implemented the earlier draft `VideoPlaybackQuality` object and the `HTMLVideoElement` extension method `getVideoPlaybackQuality()` described in those previous revisions.

B. References §

B.1 Normative references §

[dom]
DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://dom.spec.whatwg.org/

[FILE-API]
File API. Marijn Kruisselbrink; Arun Ranganathan. W3C. 11 September 2019. W3C Working Draft. URL: https://www.w3.org/TR/FileAPI/

[HTML]
HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[MSE-REGISTRY]

Media Source Extensions™ Byte Stream Format Registry. Matthew Wolenetz; Jerry Smith; Aaron Colwell. W3C. URL: https://w3c.github.io/mse-byte-stream-format-registry/

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://tools.ietf.org/html/rfc8174

[url]

URL Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://url.spec.whatwg.org/

[WEBIDL]

Web IDL. Boris Zbarsky. W3C. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

B.2 Informative references §

[INBANDTRACKS]

Sourcing In-band Media Resource Tracks from Media Containers into HTML. Silvia Pfeiffer; Bob Lund. W3C. 26 April 2015. Unofficial Draft. URL: https://dev.w3.org/html5/html-sourcing-inband-tracks/

[MEDIA-PLAYBACK-QUALITY]

Media Playback Quality. Mounir Lamouri. W3C. W3C Editor's Draft. URL: https://w3c.github.io/media-playback-quality/

↑