http://www.w3.org/TR/html5/video.html#video

HTML 5

4.8.7 The `video` element

Status: *Last call for comments.* ISSUE-7 (video-codecs), ISSUE-9 (video-synchronization) and ISSUE-10 (video-smil) block progress to Last Call

Categories

Flow content.
Phrasing content.
Embedded content.
If the element has a `controls` attribute: Interactive content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

If the element has a `src` attribute: transparent, but with no media element descendants.
If the element does not have a `src` attribute: one or more `source` elements, then, transparent, but with no media element descendants.

Content attributes:

Global attributes
`src`
`poster`
`autobuffer`
`autoplay`
`loop`
`controls`
`width`
`height`

DOM interface:

```
interface HTMLVideoElement : HTMLMediaElement {

           attribute DOMString width;

           attribute DOMString height;

  readonly attribute unsigned long videoWidth;

  readonly attribute unsigned long videoHeight;

           attribute DOMString poster;

};
```

A `video` element represents a video or movie.

Content may be provided inside the `video` element. User agents should not show this content to the user; it is intended for older Web browsers which do not support `video`, so

that legacy video plugins can be tried, or to show text to the users of these older browsers informing them of how to access the video contents.

In particular, this content is not fallback content intended to address accessibility concerns. To make video content accessible to the blind, deaf, and those with other physical or cognitive disabilities, authors are expected to provide alternative media streams and/or to embed accessibility aids (such as caption or subtitle tracks) into their media streams.

The `video` element is a media element whose media data is ostensibly video data, possibly with associated audio data.

The `src`, `autobuffer`, `autoplay`, `loop`, and `controls` attributes are the attributes common to all media elements.

The **`poster`** attribute gives the address of an image file that the user agent can show while no video data is available. The attribute, if present, must contain a valid URL. If the specified resource is to be used, then, when the element is created or when the `poster` attribute is set, its value must be resolved relative to the element, and if that is successful, the resulting absolute URL must be fetched; this must delay the load event of the element's document. The **poster frame** is then the image obtained from that resource, if any.

The image given by the `poster` attribute, the poster frame, is intended to be a representative frame of the video (typically one of the first non-blank frames) that gives the user an idea of what the video is like.

The **`poster`** DOM attribute must reflect the `poster` content attribute.

When no video data is available (the element's `readyState` attribute is either `HAVE_NOTHING`, or `HAVE_METADATA` but no video data has yet been obtained at all), the `video` element represents either the poster frame, or nothing.

When a `video` element is paused and the current playback position is the first frame of video, the element represents either the frame of video corresponding to the current playback position or the poster frame, at the discretion of the user agent.

Notwithstanding the above, the poster frame should be preferred over nothing, but the poster frame should not be shown again after a frame of video has been shown.

When a `video` element is paused at any other position, the element represents the frame of video corresponding to the current playback position, or, if that is not yet available (e.g. because the video is seeking or buffering), the last frame of the video to have been rendered.

When a `video` element is potentially playing, it represents the frame of video at the continuously increasing "current" position. When the current playback position changes such that the last frame rendered is no longer the frame corresponding to the current playback position in the video, the new frame must be rendered. Similarly, any audio

associated with the video must, if played, be played synchronized with the current playback position, at the specified volume with the specified mute state.

When a `video` element is neither potentially playing nor paused (e.g. when seeking or stalled), the element represents the last frame of the video to have been rendered.

Which frame in a video stream corresponds to a particular playback position is defined by the video stream's format.

In addition to the above, the user agent may provide messages to the user (such as "buffering", "no video loaded", "error", or more detailed information) by overlaying text or icons on the video or other areas of the element's playback area, or in another appropriate manner.

User agents that cannot render the video may instead make the element represent a link to an external video playback utility or to the video data itself.

---

video **.** **`videoWidth`**
video **.** **`videoHeight`**

These attributes return the intrinsic dimensions of the video, or zero if the dimensions are not known.

The **intrinsic width** and **intrinsic height** of the media resource are the dimensions of the resource in CSS pixels after taking into account the resource's dimensions, aspect ratio, clean aperture, resolution, and so forth, as defined for the format used by the resource.

The **`videoWidth`** DOM attribute must return the intrinsic width of the video in CSS pixels. The **`videoHeight`** DOM attribute must return the intrinsic height of the video in CSS pixels. If the element's `readyState` attribute is `HAVE_NOTHING`, then the attributes must return 0.

The `video` element supports dimension attributes.

Video content should be rendered inside the element's playback area such that the video content is shown centered in the playback area at the largest possible size that fits completely within it, with the video content's aspect ratio being preserved. Thus, if the aspect ratio of the playback area does not match the aspect ratio of the video, the video will be shown letterboxed or pillarboxed. Areas of the element's playback area that do not contain the video represent nothing.

The intrinsic width of a `video` element's playback area is the intrinsic width of the video resource, if that is available; otherwise it is the intrinsic width of the poster frame, if that is available; otherwise it is 300 CSS pixels.

The intrinsic height of a `video` element's playback area is the intrinsic height of the video resource, if that is available; otherwise it is the intrinsic height of the poster frame, if that is available; otherwise it is 150 CSS pixels.

User agents should provide controls to enable or disable the display of closed captions associated with the video stream, though such features should, again, not interfere with the page's normal rendering.

User agents may allow users to view the video content in manners more suitable to the user (e.g. full-screen or in an independent resizable window). As for the other user interface features, controls to enable this should not interfere with the page's normal rendering unless the user agent is exposing a user interface. In such an independent context, however, user agents may make full user interfaces visible, with, e.g., play, pause, seeking, and volume controls, even if the `controls` attribute is absent.

User agents may allow video playback to affect system features that could interfere with the user's experience; for example, user agents could disable screensavers while video playback is in progress.

User agents should not provide a public API to cause videos to be shown full-screen. A script, combined with a carefully crafted video file, could trick the user into thinking a system-modal dialog had been shown, and prompt the user for a password. There is also the danger of "mere" annoyance, with pages launching full-screen videos when links are clicked or pages navigated. Instead, user-agent specific interface features may be provided to easily allow the user to obtain a full-screen playback mode.

4.8.8 The `audio` element

Status: *Last call for comments*

Categories

Flow content.
Phrasing content.
Embedded content.
If the element has a `controls` attribute: Interactive content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

If the element has a `src` attribute: transparent, but with no media element descendants.
If the element does not have a `src` attribute: one or more `source` elements, then, transparent, but with no media element descendants.

Content attributes:

Global attributes
`src`
`autobuffer`
`autoplay`
`loop`
`controls`

DOM interface:

```
[NamedConstructor=Audio(),

 NamedConstructor=Audio(in DOMString src)]
```

```
interface HTMLAudioElement : HTMLMediaElement {};
```

An `audio` element represents a sound or audio stream.

Content may be provided inside the `audio` element. User agents should not show this content to the user; it is intended for older Web browsers which do not support `audio`, so that legacy audio plugins can be tried, or to show text to the users of these older browsers informing them of how to access the audio contents.

In particular, this content is not fallback content intended to address accessibility concerns. To make audio content accessible to the deaf or to those with other physical or cognitive disabilities, authors are expected to provide alternative media streams and/or to embed accessibility aids (such as transcriptions) into their media streams.

The `audio` element is a media element whose media data is ostensibly audio data.

The `src`, `autobuffer`, `autoplay`, `loop`, and `controls` attributes are the attributes common to all media elements.

When an `audio` element is potentially playing, it must have its audio data played synchronized with the current playback position, at the specified volume with the specified mute state.

When an `audio` element is not potentially playing, audio must not play for the element.

audio **= new `Audio`([*url*])**

Returns a new `audio` element, with the `src` attribute set to the value passed in the argument, if applicable.

Two constructors are provided for creating `HTMLAudioElement` objects (in addition to the factory methods from DOM Core such as `createElement()`): **`Audio()`** and **`Audio(`*`src`*`)`**. When invoked as constructors, these must return a new `HTMLAudioElement` object (a new `audio` element). The element must have its `autobuffer` attribute set to the literal value "`autobuffer`". If the *src* argument is present, the object created must have its `src` content attribute set to the provided value, and the user agent must invoke the object's resource selection algorithm before returning.

4.8.9 The `source` element

Status: *Last call for comments*

Categories

None.

Contexts in which this element may be used:

As a child of a media element, before any flow content.

Content model:

Empty.

Content attributes:

Global attributes

src
type
media

DOM interface:

```
interface HTMLSourceElement : HTMLElement {

           attribute DOMString src;

           attribute DOMString type;

           attribute DOMString media;

};
```

The source element allows authors to specify multiple media resources for media elements. It does not represent anything on its own.

The **src** attribute gives the address of the media resource. The value must be a valid URL. This attribute must be present.

The **type** attribute gives the type of the media resource, to help the user agent determine if it can play this media resource before fetching it. If specified, its value must be a valid MIME type. The codecs parameter may be specified and might be necessary to specify exactly how the resource is encoded. [RFC4281]

The following list shows some examples of how to use the codecs= MIME parameter in the type attribute.

H.264 Simple baseline profile video (main and extended video compatible) level 3 and Low-Complexity AAC audio in MP4 container

```
<source src='video.mp4' type='video/mp4; codecs="avc1.42E01E,
mp4a.40.2"'>
```

H.264 Extended profile video (baseline-compatible) level 3 and Low-Complexity AAC audio in MP4 container

```
<source src='video.mp4' type='video/mp4; codecs="avc1.58A01E,
mp4a.40.2"'>
```

H.264 Main profile video level 3 and Low-Complexity AAC audio in MP4 container

```
<source src='video.mp4' type='video/mp4; codecs="avc1.4D401E,
mp4a.40.2"'>
```

H.264 'High' profile video (incompatible with main, baseline, or extended profiles) level 3 and Low-Complexity AAC audio in MP4 container

```
<source src='video.mp4' type='video/mp4; codecs="avc1.64001E,
mp4a.40.2"'>
```

MPEG-4 Visual Simple Profile Level 0 video and Low-Complexity AAC audio in MP4 container

```
<source src='video.mp4' type='video/mp4; codecs="mp4v.20.8,
mp4a.40.2"'>
```

MPEG-4 Advanced Simple Profile Level 0 video and Low-Complexity AAC audio in MP4 container

```
<source src='video.mp4' type='video/mp4; codecs="mp4v.20.240,
mp4a.40.2"'>
```

MPEG-4 Visual Simple Profile Level 0 video and AMR audio in 3GPP container

```
<source src='video.3gp' type='video/3gpp; codecs="mp4v.20.8,
samr"'>
```

Theora video and Vorbis audio in Ogg container

```
<source src='video.ogv' type='video/ogg; codecs="theora, vorbis"'>
```
Theora video and Speex audio in Ogg container
```
<source src='video.ogv' type='video/ogg; codecs="theora, speex"'>
```
Vorbis audio alone in Ogg container
```
<source src='audio.ogg' type='audio/ogg; codecs=vorbis'>
```
Speex audio alone in Ogg container
```
<source src='audio.spx' type='audio/ogg; codecs=speex'>
```
FLAC audio alone in Ogg container
```
<source src='audio.oga' type='audio/ogg; codecs=flac'>
```
Dirac video and Vorbis audio in Ogg container
```
<source src='video.ogv' type='video/ogg; codecs="dirac, vorbis"'>
```
Theora video and Vorbis audio in Matroska container
```
<source src='video.mkv' type='video/x-matroska; codecs="theora,
vorbis"'>
```

The **`media`** attribute gives the intended media type of the media resource, to help the user agent determine if this media resource is useful to the user before fetching it. Its value must be a valid media query. [MQ]

If a `source` element is inserted as a child of a media element that is in a `Document` and whose `networkState` has the value `NETWORK_EMPTY`, the user agent must invoke the media element's resource selection algorithm.

The DOM attributes **`src`**, **`type`**, and **`media`** must reflect the respective content attributes of the same name.

4.8.10 Media elements

Status: *Last call for comments*

Media elements implement the following interface:

```
interface HTMLMediaElement : HTMLElement {

  // error state
  readonly attribute MediaError error;

  // network state
           attribute DOMString src;
  readonly attribute DOMString currentSrc;
  const unsigned short NETWORK_EMPTY = 0;
  const unsigned short NETWORK_IDLE = 1;
  const unsigned short NETWORK_LOADING = 2;
  const unsigned short NETWORK_LOADED = 3;
  const unsigned short NETWORK_NO_SOURCE = 4;
  readonly attribute unsigned short networkState;
           attribute boolean autobuffer;
  readonly attribute TimeRanges buffered;
  void load();
  DOMString canPlayType(in DOMString type);

  // ready state
  const unsigned short HAVE_NOTHING = 0;
  const unsigned short HAVE_METADATA = 1;
  const unsigned short HAVE_CURRENT_DATA = 2;
```

```
    const unsigned short HAVE_FUTURE_DATA = 3;
    const unsigned short HAVE_ENOUGH_DATA = 4;

    readonly attribute unsigned short readyState;
    readonly attribute boolean seeking;

    // playback state
             attribute float currentTime;
    readonly attribute float startTime;
    readonly attribute float duration;
    readonly attribute boolean paused;
             attribute float defaultPlaybackRate;
             attribute float playbackRate;
    readonly attribute TimeRanges played;
    readonly attribute TimeRanges seekable;
    readonly attribute boolean ended;
             attribute boolean autoplay;
             attribute boolean loop;
    void play();
    void pause();



    // controls
             attribute boolean controls;
             attribute float volume;
             attribute boolean muted;
  };
```

The **media element attributes**, `src`, `autobuffer`, `autoplay`, `loop`, and `controls`, apply to all media elements. They are defined in this section.

Media elements are used to present audio data, or video and audio data, to the user. This is referred to as **media data** in this section, since this section applies equally to media elements for audio or for video. The term **media resource** is used to refer to the complete set of media data, e.g. the complete video file, or complete audio file.

Unless otherwise specified, the task source for all the tasks queued in this section and its subsections is the **media element event task source**.

4.8.10.1 Error codes

***media* . `error`**

Returns a `MediaError` object representing the current error state of the element.

Returns null if there is no error.

All media elements have an associated error status, which records the last error the element encountered since its resource selection algorithm was last invoked. The `error` attribute, on getting, must return the `MediaError` object created for this last error, or null if there has not been an error.

```
interface MediaError {
  const unsigned short MEDIA_ERR_ABORTED = 1;
  const unsigned short MEDIA_ERR_NETWORK = 2;
```

```
    const unsigned short MEDIA_ERR_DECODE = 3;
    const unsigned short MEDIA_ERR_SRC_NOT_SUPPORTED = 4;
    readonly attribute unsigned short code;
};
```

***media* . error . code**

Returns the current error's error code, from the list below.

The `code` attribute of a `MediaError` object must return the code for the error, which must be one of the following:

***MEDIA_ERR_ABORTED* (numeric value 1)**
: The fetching process for the media resource was aborted by the user agent at the user's request.

***MEDIA_ERR_NETWORK* (numeric value 2)**
: A network error of some description caused the user agent to stop fetching the media resource, after the resource was established to be usable.

***MEDIA_ERR_DECODE* (numeric value 3)**
: An error of some description occurred while decoding the media resource, after the resource was established to be usable.

***MEDIA_ERR_SRC_NOT_SUPPORTED* (numeric value 4)**
: The media resource indicated by the `src` attribute was not suitable.

4.8.10.2 Location of the media resource

The `src` content attribute on media elements gives the address of the media resource (video, audio) to show. The attribute, if present, must contain a valid URL.

If a `src` attribute of a media element that is in a `Document` and whose `networkState` has the value `NETWORK_EMPTY` is set or changed, the user agent must invoke the media element's resource selection algorithm.

The `src` DOM attribute on media elements must reflect the respective content attribute of the same name.

***media* . currentSrc**

Returns the address of the current media resource.

Returns the empty string when there is no media resource.

The `currentSrc` DOM attribute is initially the empty string. Its value is changed by the resource selection algorithm defined below.

There are two ways to specify a media resource, the `src` attribute, or `source` elements. The attribute overrides the elements.

4.8.10.3 MIME types

A media resource can be described in terms of its *type*, specifically a MIME type, optionally with a `codecs` parameter. [RFC4281].

Types are usually somewhat incomplete descriptions; for example "`video/mpeg`" doesn't say anything except what the container type is, and even a type like "`video/mp4; codecs="avc1.42E01E, mp4a.40.2"`" doesn't include information like the actual bitrate (only the maximum bitrate). Thus, given a type, a user agent can often only know whether it *might* be able to play media of that type (with varying levels of confidence), or whether it definitely *cannot* play media of that type.

A type that the user agent knows it cannot render is one that describes a resource that the user agent definitely does not support, for example because it doesn't recognize the container type, or it doesn't support the listed codecs.

The MIME type "`application/octet-stream`" is never a type that the user agent knows it cannot render. User agents must treat that type as equivalent to the lack of any explicit Content-Type metadata when it is used to label a potential media resource.

***media* . `canPlayType`(*type*)**

Returns the empty string (a negative response), "maybe", or "probably" based on how confident the user agent is that it can play media resources of the given type.

The **`canPlayType(`*`type`*`)`** method must return the empty string if *type* is a type that the user agent knows it cannot render; it must return "`probably`" if the user agent is confident that the type represents a media resource that it can render if used in with this `audio` or `video` element; and it must return "`maybe`" otherwise. Implementors are encouraged to return "`maybe`" unless the type can be confidently established as being supported or not. Generally, a user agent should never return "`probably`" if the type doesn't have a `codecs` parameter.

This script tests to see if the user agent supports a (fictional) new format to dynamically decide whether to use a `video` element or a plugin:

```
<section id="video">
 <p><a href="playing-cats.nfv">Download video</a></p>
</section>
<script>
 var videoSection = document.getElementById('video');
 var videoElement = document.createElement('video');
 var support = videoElement.canPlayType('video/x-new-fictional-
format;codecs="kittens,bunnies"');
 if (support != "probably" && "New Fictional Video Plug-in" in
navigator.plugins) {
   // not confident of browser support
   // but we have a plugin
   // so use plugin instead
   videoElement = document.createElement("embed");
 } else if (support == "") {
   // no support from browser and no plugin
   // do nothing
   videoElement = null;
 }
 if (videoElement) {
```

```
    while (videoSection.hasChildNodes())
      videoSection.removeChild(videoSection.firstChild);
    videoElement.setAttribute("src", "playing-cats.nfv");
    videoSection.appendChild(videoElement);
  }
</script>
```

The `type` attribute of the `source` element allows the user agent to avoid downloading resources that use formats it cannot render.

4.8.10.4 Network states

***media* . `networkState`**

Returns the current state of network activity for the element, from the codes in the list below.

As media elements interact with the network, their current network activity is represented by the `networkState` attribute. On getting, it must return the current network state of the element, which must be one of the following values:

***`NETWORK_EMPTY`* (numeric value 0)**
: The element has not yet been initialized. All attributes are in their initial states.

***`NETWORK_IDLE`* (numeric value 1)**
: The element's resource selection algorithm is active and has selected a resource, but it is not actually using the network at this time.

***`NETWORK_LOADING`* (numeric value 2)**
: The user agent is actively trying to download data.

***`NETWORK_LOADED`* (numeric value 3)**
: The entire media resource has been obtained and is available to the user agent locally. Network connectivity could be lost without affecting the media playback.

***`NETWORK_NO_SOURCE`* (numeric value 4)**
: The element's resource selection algorithm is active, but it has failed to find a resource to use.

The resource selection algorithm defined below describes exactly when the `networkState` attribute changes value and what events fire to indicate changes in this state.

Some resources, e.g. streaming Web radio, can never reach the `NETWORK_LOADED` state.

4.8.10.5 Loading the media resource

***media* . `load`()**

Causes the element to reset and start selecting and loading a new media resource from scratch.

All media elements have an **autoplaying flag**, which must begin in the true state, and a **delaying-the-load-event flag**, which must begin in the false state. While the delaying-the-load-event flag is true, the element must delay the load event of its document.

When the `load()` method on a media element is invoked, the user agent must run the following steps. Note that this algorithm might get aborted, e.g. if the `load()` method itself is invoked again.

1. If the `load()` method for this element is already being invoked, then abort these steps.

2. Abort any already-running instance of the resource selection algorithm for this element.

3. If there are any tasks from the media element's media element event task source in one of the task queues, then remove those tasks.

   Basically, pending events and callbacks for the media element are discarded when the media element starts loading a new resource.

4. If the media element's `networkState` is set to `NETWORK_LOADING` or `NETWORK_IDLE`, set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_ABORTED`, fire a progress event called `abort` at the media element, in the context of the fetching process that is in progress for the element, and fire a progress event called `loadend` at the media element, in the context of the same fetching process.

5. Set the `error` attribute to null and the autoplaying flag to true.

6. Set the `playbackRate` attribute to the value of the `defaultPlaybackRate` attribute.

7. If the media element's `networkState` is not set to `NETWORK_EMPTY`, then run these substeps:

   1. If a fetching process is in progress for the media element, the user agent should stop it.

   2. Set the `networkState` attribute to NETWORK_EMPTY.
   3. If `readyState` is not set to `HAVE_NOTHING`, then set it to that state.
   4. If the `paused` attribute is false, then set to true.
   5. If `seeking` is true, set it to false.
   6. Set the current playback position to 0.
   7. Fire a simple event called `emptied` at the media element.

8. Invoke the media element's resource selection algorithm.

9. ***Playback of any previously playing media resource for this element stops.***

The **resource selection algorithm** for a media element is as follows. This algorithm is always invoked synchronously, but one of the first steps in the algorithm is to return and continue running the remaining steps asynchronously, meaning that it runs in the background with scripts and other tasks running in parallel.

1. Set the `networkState` to `NETWORK_NO_SOURCE`.

2. Asynchronously await a stable state, allowing the task that invoked this algorithm to continue. The synchronous section consists of all the remaining steps of this

algorithm until the algorithm says the synchronous section has ended. (Steps in synchronous sections are marked with □.)

3. □ If the media element has a `src` attribute, then let *mode* be *attribute*.

   □ Otherwise, if the media element does not have a `src` attribute but has a `source` element child, then let *mode* be *children* and let *candidate* be the first such `source` element child in tree order.

   □ Otherwise the media element has neither a `src` attribute nor a `source` element child: set the `networkState` to `NETWORK_EMPTY`, and abort these steps; the synchronous section ends.

4. □ Set the media element's delaying-the-load-event flag to true (this delays the load event), and set its `networkState` to `NETWORK_LOADING`.

5. □ Queue a task to fire a progress event called `loadstart` at the media element, with no relevant fetching process.

6. If *mode* is *attribute*, then run these substeps:

   1. □ Let *absolute URL* be the absolute URL that would have resulted from resolving the URL specified by the `src` attribute's value relative to the media element when the `src` attribute was last changed.

   2. End the synchronous section, continuing the remaining steps asynchronously.

   3. If *absolute URL* was obtained successfully, run the resource fetch algorithm with *absolute URL*. If that algorithm returns without aborting *this* one, then the load failed.

   4. Reaching this step indicates that the media resource failed to load or that the given URL could not be resolved. Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_SRC_NOT_SUPPORTED`.

   5. Set the element's `networkState` attribute to the NETWORK_NO_SOURCE value.

   6. Queue a task to fire a progress event called `error` at the media element, in the context of the fetching process that was used to try to obtain the media resource in the resource fetch algorithm.

   7. Queue a task to fire a progress event called `loadend` at the media element, in the context of the fetching process that was used to try to obtain the media resource in the resource fetch algorithm.

   8. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.

   9. Abort these steps. Until the `load()` method is invoked, the element won't attempt to load another resource.

Otherwise, the `source` elements will be used; run these substeps:

10. Let *pointer* be a position defined by two adjacent nodes in the media element's child list, treating the start of the list (before the first child in the list, if any) and end of the list (after the last child in the list, if any) as nodes in their own right. One node is the node before *pointer*, and the other node is the node after *pointer*. Initially, let *pointer* be the position between the *candidate* node and the next node, if there are any, or the end of the list, if it is the last node.

    As elements are inserted and removed into the media element, *pointer* must be updated as follows:

    If a new element is inserted between the two nodes that define *pointer*
    Let *pointer* be the point between the node before *pointer* and the new node. In other words, insertions at *pointer* go after *pointer*.
    If the node before *pointer* is removed
    Let *pointer* be the point between the node after *pointer* and the node before the node after *pointer*. In other words, *pointer* doesn't move relative to the remaining nodes.
    If the node after *pointer* is removed
    Let *pointer* be the point between the node before *pointer* and the node after the node before *pointer*. Just as with the previous case, *pointer* doesn't move relative to the remaining nodes.

    Other changes don't affect *pointer*.

11. *Process candidate*: If *candidate* does not have a `src` attribute, then end the synchronous section, and jump down to the *failed* step below.

12. Let *absolute URL* be the absolute URL that would have resulted from resolving the URL specified by *candidate*'s `src` attribute's value relative to the *candidate* when the `src` attribute was last changed.

13. If *absolute URL* was not obtained successfully, then end the synchronous section, and jump down to the *failed* step below.

14. If *candidate* has a `type` attribute whose value, when parsed as a MIME type (including any codecs described by the `codec` parameter), represents a type that the user agent knows it cannot render, then end the synchronous section, and jump down to the *failed* step below.

15. If *candidate* has a `media` attribute whose value, when processed according to the rules for media queries, does not match the current environment, then end the synchronous section, and jump down to the *failed* step below. [MQ]

16. End the synchronous section, continuing the remaining steps asynchronously.

17. Run the resource fetch algorithm with *absolute URL*. If that algorithm returns without aborting *this* one, then the load failed.

18. *Failed*: Queue a task to fire a simple event called `error` at the *candidate* element, in the context of the fetching process that was used to try to obtain *candidate*'s corresponding media resource in the resource fetch algorithm.

19. Asynchronously await a stable state. The synchronous section consists of all the remaining steps of this algorithm until the algorithm says the synchronous section has ended. (Steps in synchronous sections are marked with □.)

20. □ *Find next candidate*: Let *candidate* be null.

21. □ *Search loop*: If the node after *pointer* is the end of the list, then jump to the *waiting* step below.

22. □ If the node after *pointer* is a `source` element, let *candidate* be that element.

23. □ Advance *pointer* so that the node before *pointer* is now the node that was after *pointer*, and the node after *pointer* is the node after the node that used to be after *pointer*, if any.

24. □ If *candidate* is null, jump back to the *search loop* step. Otherwise, jump back to the *process candidate* step.

25. □ *Waiting*: Set the element's `networkState` attribute to the NETWORK_NO_SOURCE value.

26. □ Set the element's delaying-the-load-event flag to false. This stops delaying the load event.

27. End the synchronous section, continuing the remaining steps asynchronously.

28. Wait until the node after *pointer* is a node other than the end of the list. (This step might wait forever.)

29. Asynchronously await a stable state. The synchronous section consists of all the remaining steps of this algorithm until the algorithm says the synchronous section has ended. (Steps in synchronous sections are marked with □.)

30. □ Set the element's delaying-the-load-event flag back to true (this delays the load event again, in case it hasn't been fired yet).

31. □ Set the `networkState` back to `NETWORK_LOADING`.

32. □ Jump back to the *find next candidate* step above.

The **resource fetch algorithm** for a media element and a given absolute URL is as follows:

1. Let the *current media resource* be the resource given by the absolute URL passed to this algorithm. This is now the element's media resource.

2. Set the `currentSrc` attribute to the absolute URL of the *current media resource*.

3. Begin to fetch the *current media resource*.

   Every 350ms (±200ms) or for every byte received, whichever is *least* frequent, queue a task to fire a progress event called `progress` at the element, in the context of the fetching process started by this instance of this algorithm.

   If at any point the user agent has received no data for more than about three seconds, then queue a task to fire a progress event called `stalled` at the element, in the context of the fetching process started by this instance of this algorithm.

   User agents may allow users to selectively block or slow media data downloads. When a media element's download has been blocked altogether, the user agent must act as if it was stalled (as opposed to acting as if the connection was closed). The rate of the download may also be throttled automatically by the user agent, e.g. to balance the download with other connections sharing the same bandwidth.

   User agents may decide to not download more content at any time, e.g. after buffering five minutes of a one hour media resource, while waiting for the user to decide whether to play the resource or not, or while waiting for user input in an interactive resource. When a media element's download has been suspended, the user agent must set the `networkState` to `NETWORK_IDLE` and queue a task to fire a progress event called `suspend` at the element, in the context of the fetching process started by this instance of this algorithm. If and when downloading of the resource resumes, the user agent must set the `networkState` to `NETWORK_LOADING`.

   The `autobuffer` attribute provides a hint that the author expects that downloading the entire resource optimistically will be worth it, even in the absence of the `autoplay` attribute. In the absence of either attribute, the user agent is likely to find that waiting until the user starts playback before downloading any further content leads to a more efficient use of the network resources.

   When a user agent decides to completely stall a download, e.g. if it is waiting until the user starts playback before downloading any further content, the element's delaying-the-load-event flag must be set to false. This stops delaying the load event.

   The user agent may use whatever means necessary to fetch the resource (within the constraints put forward by this and other specifications); for example, reconnecting to the server in the face of network errors, using HTTP partial range requests, or switching to a streaming protocol. The user agent must consider a resource erroneous only if it has given up trying to fetch it.

   The networking task source tasks to process the data as it is being fetched must, when appropriate, include the relevant substeps from the following list:

   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
   If the media resource is found to have Content-Type metadata that, when parsed as a MIME type (including any codecs described by the `codec`

parameter), represents a type that the user agent knows it cannot render (even if the actual media data is in a supported format)
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
DNS errors, HTTP 4xx and 5xx errors (and equivalents in other protocols), and other fatal network errors that occur before the user agent has established whether the *current media resource* is usable, as well as the file using an unsupported container format, or using unsupported codecs for all the data, must cause the user agent to execute the following steps:

1. The user agent should cancel the fetching process.

2. Abort this subalgorithm, returning to the resource selection algorithm.

Once enough of the media data has been fetched to determine the duration of the media resource, its dimensions, and other metadata
This indicates that the resource is usable. The user agent must follow these substeps:

3. Set the current playback position to the earliest possible position.

4. Set the `readyState` attribute to `HAVE_METADATA`.

5. For `video` elements, set the `videoWidth` and `videoHeight` attributes.

6. Set the `duration` attribute to the duration of the resource.

   The user agent will queue a task to fire a simple event called `durationchange` at the element at this point.

7. Queue a task to fire a simple event called `loadedmetadata` at the element.

   Before this task is run, sa part of the event loop mechanism, the rendering will have been updated to resize the `video` element if appropriate.

8. If either the media resource or the address of the *current media resource* indicate a particular start time, then seek to that time. Ignore any resulting exceptions (if the position is out of range, it is effectively ignored).

   > For example, a fragment identifier could be used to indicate a start position.

9. Once the `readyState` attribute reaches `HAVE_CURRENT_DATA`, after the `loadeddata` event has been fired, set the element's delaying-the-load-event flag to false. This stops delaying the load event.

   A user agent that is attempting to reduce network usage while still fetching the metadata for each media resource would also stop buffering at this point, causing the `networkState` attribute to switch to the `NETWORK_IDLE` value, if the media element did not have an `autobuffer` or `autoplay` attribute.

The user agent is required to determine the duration of the media resource and go through this step before playing.

Once the entire media resource has been fetched (but potentially before any of it has been decoded)
Queue a task to fire a progress event called `progress` at the media element, in the context of the fetching process started by this instance of this algorithm.

If the connection is interrupted, causing the user agent to give up trying to fetch the resource
Fatal network errors that occur after the user agent has established whether the *current media resource* is usable must cause the user agent to execute the following steps:

10. The user agent should cancel the fetching process.

11. Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_NETWORK`.

12. Queue a task to fire a progress event called `error` at the media element, in the context of the fetching process started by this instance of this algorithm.

13. Queue a task to fire a progress event called `loadend` at the media element, in the context of the fetching process started by this instance of this algorithm.

14. Set the element's `networkState` attribute to the NETWORK_EMPTY value and queue a task to fire a simple event called `emptied` at the element.

15. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.

16. Abort the overall resource selection algorithm.

If the media data is corrupted
Fatal errors in decoding the media data that occur after the user agent has established whether the *current media resource* is usable must cause the user agent to execute the following steps:

17. The user agent should cancel the fetching process.

18. Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_DECODE`.

19. Queue a task to fire a progress event called `error` at the media element, in the context of the fetching process started by this instance of this algorithm.

20. Queue a task to fire a progress event called `loadend` at the media element, in the context of the fetching process started by this instance of this algorithm.

21. Set the element's `networkState` attribute to the NETWORK_EMPTY value and queue a task to fire a simple event called `emptied` at the element.

22. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.

23. Abort the overall resource selection algorithm.

If the media data fetching process is aborted by the user
The fetching process is aborted by the user, e.g. because the user navigated the browsing context to another page, the user agent must execute the following steps. These steps are not followed if the `load()` method itself is invoked while these steps are running, as the steps above handle that particular kind of abort.

24. The user agent should cancel the fetching process.

25. Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_ABORT`.

26. Queue a task to fire a progress event called `abort` at the media element, in the context of the fetching process started by this instance of this algorithm.

27. Queue a task to fire a progress event called `loadend` at the media element, in the context of the fetching process started by this instance of this algorithm.

28. If the media element's `readyState` attribute has a value equal to `HAVE_NOTHING`, set the element's `networkState` attribute to the NETWORK_EMPTY value and queue a task to fire a simple event called `emptied` at the element. Otherwise, set the element's `networkState` attribute to the NETWORK_IDLE value.

29. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.

30. Abort the overall resource selection algorithm.

If the media data can be fetched but has non-fatal errors or uses, in part, codecs that are unsupported, preventing the user agent from rendering the content completely correctly but not preventing playback altogether
The server returning data that is partially usable but cannot be optimally rendered must cause the user agent to render just the bits it can handle, and ignore the rest.

When the networking task source has queued the last task as part of fetching the media resource (i.e. once the download has completed), if the fetching process completes without errors, including decoding the media data, then, the user agent must move on to the next step. This might never happen, e.g. when streaming an infinite resource such as Web radio.

4. Set the `networkState` attribute to `NETWORK_LOADED`.

5. Queue a task to fire a progress event called `load` at the media element, in the context of the fetching process started by this instance of this algorithm.

6. Queue a task to fire a progress event called `loadend` at the media element, in the context of the fetching process started by this instance of this algorithm.

7. Finally, abort the overall resource selection algorithm.

If a media element whose `networkState` has the value `NETWORK_EMPTY` is inserted into a document, the user agent must invoke the media element's resource selection algorithm.

The **`autobuffer`** attribute is a boolean attribute. Its presence hints to the user agent that the author believes that the media element will likely be used, even though the element does not have an `autoplay` attribute. (The attribute has no effect if used in conjunction with the `autoplay` attribute, though including both is not an error.) This attribute may be ignored altogether. The attribute must be ignored if the `autoplay` attribute is present.

The **`autobuffer`** DOM attribute must reflect the content attribute of the same name.

media **.** **`buffered`**

Returns a `TimeRanges` object that represents the ranges of the media resource that the user agent has buffered.

The **`buffered`** attribute must return a new static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent has buffered, at the time the attribute is evaluated. Users agents must accurately determine the ranges available, even for media streams where this can only be determined by tedious inspection.

Typically this will be a single range anchored at the zero point, but if, e.g. the user agent uses HTTP range requests in response to seeking, then there could be multiple ranges.

User agents may discard previously buffered data.

Thus, a time position included within a range of the objects return by the `buffered` attribute at one time can end up being not included in the range(s) of objects returned by the same attribute at later times.

4.8.10.6 Offsets into the media resource

Status: *Being considered for removal*

media **.** **`duration`**

Returns the length of the media resource, in seconds.

Returns NaN if the duration isn't available.

Returns Infinity for unbounded streams.

media **.** **`currentTime`** **[= *value*]**

Returns the current playback position, in seconds.

Can be set, to seek to the given time.

Will throw an `INVALID_STATE_ERR` exception if there is no selected media resource.
Will throw an `INDEX_SIZE_ERR` exception if the given time is not within the ranges to which the user agent can seek.

media **.** **`startTime`**

Returns the earliest possible position, in seconds. This is the time for the start of the current clip. It might not be zero if the clip's timeline is not zero-based, or if the resource is a streaming resource (in which case it gives the earliest time that the user agent is able to seek back to).

The **`duration`** attribute must return the length of the media resource, in seconds. If no media data is available, then the attributes must return the Not-a-Number (NaN) value. If the media resource is known to be unbounded (e.g. a streaming radio), then the attribute must return the positive Infinity value.

The user agent must determine the duration of the media resource before playing any part of the media data and before setting `readyState` to a value equal to or greater than `HAVE_METADATA`, even if doing so requires seeking to multiple parts of the resource.

When the length of the media resource changes (e.g. from being unknown to known, or from a previously established length to a new length) the user agent must queue a task to fire a simple event called `durationchange` at the media element.

> If an "infinite" stream ends for some reason, then the duration would change from positive Infinity to the time of the last frame or sample in the stream, and the `durationchange` event would be fired. Similarly, if the user agent initially estimated the media resource's duration instead of determining it precisely, and later revises the estimate based on new information, then the duration would change and the `durationchange` event would be fired.

Media elements have a **current playback position**, which must initially be zero. The current position is a time.

The **`currentTime`** attribute must, on getting, return the current playback position, expressed in seconds. On setting, the user agent must seek to the new value (which might raise an exception).

If the media resource is a streaming resource, then the user agent might be unable to obtain certain parts of the resource after it has expired from its buffer. Similarly, some media resources might have a timeline that doesn't start at zero. The **earliest possible position** is the earliest position in the stream or resource that the user agent can ever obtain again.

The **`startTime`** attribute must, on getting, return the earliest possible position, expressed in seconds.

When the earliest possible position changes, then: if the current playback position is before the earliest possible position, the user agent must seek to the earliest possible position; otherwise, if the user agent has not fired a `timeupdate` event at the element in the past 15 to 250ms, then the user agent must queue a task to fire a simple event called `timeupdate` at the element.

User agents must act as if the timeline of the media resource increases linearly starting from the earliest possible position, even if the underling media data has out-of-order or even overlapping time codes.

> For example, if two clips have been concatenated into one video file, but the video format exposes the original times for the two clips, the video data might expose a timeline that goes, say, 00:15..00:29 and then 00:05..00:38. However, the user agent would not expose those times; it would instead expose the times as 00:15..00:29 and 00:29..01:02, as a single video.

The `loop` attribute is a boolean attribute that, if specified, indicates that the media element is to seek back to the start of the media resource upon reaching the end.

The `loop` DOM attribute must reflect the content attribute of the same name.

4.8.10.7 The ready states

***media* . `readyState`**

Returns a value that expresses the current state of the element with respect to rendering the current playback position, from the codes in the list below.

Media elements have a *ready state*, which describes to what degree they are ready to be rendered at the current playback position. The possible values are as follows; the ready state of a media element at any particular time is the greatest value describing the state of the element:

HAVE_NOTHING **(numeric value 0)**
: No information regarding the media resource is available. No data for the current playback position is available. Media elements whose `networkState` attribute is `NETWORK_EMPTY` are always in the `HAVE_NOTHING` state.

HAVE_METADATA **(numeric value 1)**
: Enough of the resource has been obtained that the duration of the resource is available. In the case of a `video` element, the dimensions of the video are also available. The API will no longer raise an exception when seeking. No media data is available for the immediate current playback position.

HAVE_CURRENT_DATA **(numeric value 2)**
: Data for the immediate current playback position is available, but either not enough data is available that the user agent could successfully advance the current playback position in the direction of playback at all without immediately reverting to the `HAVE_METADATA` state, or there is no more data to obtain in the direction of playback. For example, in video this corresponds to the user agent having data from the current frame, but not the next frame; and to when playback has ended.

HAVE_FUTURE_DATA **(numeric value 3)**
: Data for the immediate current playback position is available, as well as enough data for the user agent to advance the current playback position in the direction of playback at least a little without immediately reverting to the `HAVE_METADATA` state. For example, in video this corresponds to the user agent having data for at least the current frame and the next frame. The user agent cannot be in this state if playback has ended, as the current playback position can never advanced in this case.

HAVE_ENOUGH_DATA **(numeric value 4)**
: All the conditions described for the `HAVE_FUTURE_DATA` state are met, and, in addition, the user agent estimates that data is being fetched at a rate where the current playback position, if it were to advance at the rate given by the

defaultPlaybackRate attribute, would not overtake the available data before playback reaches the end of the media resource.

When the ready state of a media element whose networkState is not NETWORK_EMPTY changes, the user agent must follow the steps given below:

If the previous ready state was HAVE_NOTHING, and the new ready state is HAVE_METADATA

A loadedmetadata DOM event will be fired as part of the load() algorithm.

If the previous ready state was HAVE_METADATA and the new ready state is HAVE_CURRENT_DATA or greater

If this is the first time this occurs for this media element since the load() algorithm was last invoked, the user agent must queue a task to fire a simple event called loadeddata at the element.

If the new ready state is HAVE_FUTURE_DATA or HAVE_ENOUGH_DATA, then the relevant steps below must then be run also.

If the previous ready state was HAVE_FUTURE_DATA or more, and the new ready state is HAVE_CURRENT_DATA or less

A waiting DOM event can be fired, depending on the current state of playback.

If the previous ready state was HAVE_CURRENT_DATA or less, and the new ready state is HAVE_FUTURE_DATA

The user agent must queue a task to fire a simple event called canplay.

If the element is potentially playing, the user agent must queue a task to fire a simple event called playing.

If the new ready state is HAVE_ENOUGH_DATA

If the previous ready state was HAVE_CURRENT_DATA or less, the user agent must queue a task to fire a simple event called canplay, and, if the element is also potentially playing, queue a task to fire a simple event called playing.

If the autoplaying flag is true, and the paused attribute is true, and the media element has an autoplay attribute specified, then the user agent may also set the paused attribute to false, queue a task to fire a simple event called play, and queue a task to fire a simple event called playing.

User agents are not required to autoplay, and it is suggested that user agents honor user preferences on the matter. Authors are urged to use the autoplay attribute rather than using script to force the video to play, so as to allow the user to override the behavior if so desired.

In any case, the user agent must finally queue a task to fire a simple event called canplaythrough.

It is possible for the ready state of a media element to jump between these states discontinuously. For example, the state of a media element can jump straight from

***HAVE METADATA* to *HAVE ENOUGH DATA* without passing through the *HAVE CURRENT DATA* and *HAVE FUTURE DATA* states.**

The `readyState` DOM attribute must, on getting, return the value described above that describes the current ready state of the media element.

The `autoplay` attribute is a boolean attribute. When present, the user agent (as described in the algorithm described herein) will automatically begin playback of the media resource as soon as it can do so without stopping.

Authors are urged to use the `autoplay` attribute rather than using script to trigger automatic playback, as this allows the user to override the automatic playback when it is not desired, e.g. when using a screen reader. Authors are also encouraged to consider not using the automatic playback behavior at all, and instead to let the user agent wait for the user to start playback explicitly.

The `autoplay` DOM attribute must reflect the content attribute of the same name.

4.8.10.8 Playing the media resource

***media* . `paused`**

Returns true if playback is paused; false otherwise.

***media* . `ended`**

Returns true if playback has reached the end of the media resource.

***media* . `defaultPlaybackRate` [= *value*]**

Returns the default rate of playback, for when the user is not fast-forwarding or reversing through the media resource.

Can be set, to change the default rate of playback.

The default rate has no direct effect on playback, but if the user switches to a fast-forward mode, when they return to the normal playback mode, it is expected that the rate of playback will be returned to the default rate of playback.

***media* . `playbackRate` [= *value*]**

Returns the current rate playback, where 1.0 is normal speed.

Can be set, to change the rate of playback.

***media* . `played`**

Returns a `TimeRanges` object that represents the ranges of the media resource that the user agent has played.

***media* . `play`()**

Sets the `paused` attribute to false, loading the media resource and beginning playback if necessary. If the playback had ended, will restart it from the start.

***media* . `pause`()**

Sets the `paused` attribute to true, loading the media resource if necessary.

The `paused` attribute represents whether the media element is paused or not. The attribute must initially be true.

A media element is said to be **potentially playing** when its `paused` attribute is false, the `readyState` attribute is either `HAVE_FUTURE_DATA` or `HAVE_ENOUGH_DATA`, the element has not ended playback, playback has not stopped due to errors, and the element has not paused for user interaction.

A media element is said to have **ended playback** when the element's `readyState` attribute is `HAVE_METADATA` or greater, and either the current playback position is the end of the media resource and the direction of playback is forwards and the media element does not have a `loop` attribute specified, or the current playback position is the earliest possible position and the direction of playback is backwards.

The `ended` attribute must return true if the media element has ended playback and the direction of playback is forwards, and false otherwise.

A media element is said to have **stopped due to errors** when the element's `readyState` attribute is `HAVE_METADATA` or greater, and the user agent encounters a non-fatal error during the processing of the media data, and due to that error, is not able to play the content at the current playback position.

A media element is said to have **paused for user interaction** when its `paused` attribute is false, the `readyState` attribute is either `HAVE_FUTURE_DATA` or `HAVE_ENOUGH_DATA` and the user agent has reached a point in the media resource where the user has to make a selection for the resource to continue.

It is possible for a media element to have both ended playback and paused for user interaction at the same time.

When a media element that is potentially playing stops playing because it has paused for user interaction, the user agent must queue a task to fire a simple event called `timeupdate` at the element.

When a media element that is potentially playing stops playing because its `readyState` attribute changes to a value lower than `HAVE_FUTURE_DATA`, without the element having ended playback, or playback having stopped due to errors, or playback having paused for user interaction, or the seeking algorithm being invoked, the user agent must queue a task to fire a simple event called `timeupdate` at the element, and queue a task to fire a simple event called `waiting` at the element.

When the current playback position reaches the end of the media resource when the direction of playback is forwards, then the user agent must follow these steps:

1. If the media element has a `loop` attribute specified, then seek to the earliest possible position of the media resource and abort these steps.

2. Stop playback.

   The `ended` attribute becomes true.

3. The user agent must queue a task to fire a simple event called `timeupdate` at the element.

4. The user agent must queue a task to fire a simple event called `ended` at the element.

When the current playback position reaches the earliest possible position of the media resource when the direction of playback is backwards, then the user agent must follow these steps:

1. Stop playback.

2. The user agent must queue a task to fire a simple event called `timeupdate` at the element.

The **`defaultPlaybackRate`** attribute gives the desired speed at which the media resource is to play, as a multiple of its intrinsic speed. The attribute is mutable: on getting it must return the last value it was set to, or 1.0 if it hasn't yet been set; on setting the attribute must be set to the new value.

The **`playbackRate`** attribute gives the speed at which the media resource plays, as a multiple of its intrinsic speed. If it is not equal to the `defaultPlaybackRate`, then the implication is that the user is using a feature such as fast forward or slow motion playback. The attribute is mutable: on getting it must return the last value it was set to, or 1.0 if it hasn't yet been set; on setting the attribute must be set to the new value, and the playback must change speed (if the element is potentially playing).

If the `playbackRate` is positive or zero, then the **direction of playback** is forwards. Otherwise, it is backwards.

The "play" function in a user agent's interface must set the `playbackRate` attribute to the value of the `defaultPlaybackRate` attribute before invoking the `play()` method's steps. Features such as fast-forward or rewind must be implemented by only changing the `playbackRate` attribute.

When the `defaultPlaybackRate` or `playbackRate` attributes change value (either by being set by script or by being changed directly by the user agent, e.g. in response to user control) the user agent must queue a task to fire a simple event called `ratechange` at the media element.

The **`played`** attribute must return a new static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent has so far rendered, at the time the attribute is evaluated.

When the **`play()`** method on a media element is invoked, the user agent must run the following steps.

1. If the media element's `networkState` attribute has the value `NETWORK_EMPTY`, then the user agent must invoke the media element's resource selection algorithm.

2. If the playback has ended, then the user agent must seek to the earliest possible position of the media resource.

   This will cause the user agent to queue a task to fire a simple event called `timeupdate` at the media element.

3. If the media element's `paused` attribute is true, it must be set to false.

   If this changed the value of `paused`, the user agent must run the following substeps:

   1. Queue a task to fire a simple event called `play` at the element.

   2. If the media element's `readyState` attribute has the value `HAVE_NOTHING`, `HAVE_METADATA`, or `HAVE_CURRENT_DATA`, queue a task to fire a simple event called `waiting` at the element.

   3. Otherwise, the media element's `readyState` attribute has the value `HAVE_FUTURE_DATA` or `HAVE_ENOUGH_DATA`; queue a task to fire a simple event called `playing` at the element.

4. The media element's autoplaying flag must be set to false.

5. The method must then return.

When the **`pause()`** method is invoked, the user agent must run the following steps:

1. If the media element's `networkState` attribute has the value `NETWORK_EMPTY`, then the user agent must invoke the media element's resource selection algorithm.

2. If the media element's `paused` attribute is false, it must be set to true.

3. The media element's autoplaying flag must be set to false.

4. If the second step above changed the value of `paused`, then the user agent must queue a task to fire a simple event called `timeupdate` at the element, and queue a task to fire a simple event called `pause` at the element.

When a media element is potentially playing and its `Document` is an active document, its current playback position must increase monotonically at `playbackRate` units of media time per unit time of wall clock time.

This specification doesn't define how the user agent achieves the appropriate playback rate — depending on the protocol and media available, it is plausible that the user agent could negotiate with the server to have the server provide the media data at the appropriate rate, so that (except for the period between when the rate is changed and when the server updates the stream's playback rate) the client doesn't actually have to drop or interpolate any frames.

When the playbackRate is negative (playback is backwards), any corresponding audio must be muted. When the playbackRate is so low or so high that the user agent cannot play audio usefully, the corresponding audio must also be muted. If the playbackRate is not 1.0, the user agent may apply pitch adjustments to the audio as necessary to render it faithfully.

The playbackRate can be 0.0, in which case the current playback position doesn't move, despite playback not being paused (paused doesn't become true, and the pause event doesn't fire).

Media elements that are potentially playing while not in a Document must not play any video, but should play any audio component. Media elements must not stop playing just because all references to them have been removed; only once a media element to which no references exist has reached a point where no further audio remains to be played for that element (e.g. because the element is paused, or because the end of the clip has been reached, or because its playbackRate is 0.0) may the element be garbage collected.

When the current playback position of a media element changes (e.g. due to playback or seeking), the user agent must run the following steps. If the current playback position changes while the steps are running, then the user agent must wait for the steps to complete, and then must immediately rerun the steps. (These steps are thus run as often as possible or needed — if one iteration takes a long time, this can cause certain ranges to be skipped over as the user agent rushes ahead to "catch up".)

1. If the time was reached through the usual monotonic increase of the current playback position during normal playback, and if the user agent has not fired a timeupdate event at the element in the past 15 to 250ms, then the user agent must queue a task to fire a simple event called timeupdate at the element. (In the other cases, such as explicit seeks, relevant events get fired as part of the overall process of changing the current playback position.)

   The event thus is not to be fired faster than about 66Hz or slower than 4Hz. User agents are encouraged to vary the frequency of the event based on the system load and the average cost of processing the event each time, so that the UI updates are not any more frequent than the user agent can comfortably handle while decoding the video.

When a media element is removed from a Document, if the media element's networkState attribute has a value other than NETWORK_EMPTY then the user agent must act as if the pause() method had been invoked.

If the media element's Document stops being a fully active document, then the playback will stop until the document is active again.

4.8.10.9 Seeking

media . **seeking**

Returns true if the user agent is currently seeking.

media **.** **`seekable`**

Returns a `TimeRanges` object that represents the ranges of the media resource to which it is possible for the user agent to seek.

The **`seeking`** attribute must initially have the value false.

When the user agent is required to **seek** to a particular *new playback position* in the media resource, it means that the user agent must run the following steps:

1. If the media element's `readyState` is `HAVE_NOTHING`, then the user agent must raise an `INVALID_STATE_ERR` exception (if the seek was in response to a DOM method call or setting of a DOM attribute), and abort these steps.

2. If the *new playback position* is later than the end of the media resource, then let it be the end of the media resource instead.

3. If the *new playback position* is less than the earliest possible position, let it be that position instead.

4. If the (possibly now changed) *new playback position* is not in one of the ranges given in the `seekable` attribute, then the user agent must raise an `INDEX_SIZE_ERR` exception (if the seek was in response to a DOM method call or setting of a DOM attribute), and abort these steps.

5. The current playback position must be set to the given *new playback position*.

6. The `seeking` DOM attribute must be set to true.

7. The user agent must queue a task to fire a simple event called `timeupdate` at the element.

8. If the media element was potentially playing immediately before it started seeking, but seeking caused its `readyState` attribute to change to a value lower than `HAVE_FUTURE_DATA`, the user agent must queue a task to fire a simple event called `waiting` at the element.

9. If, when it reaches this step, the user agent has still not established whether or not the media data for the *new playback position* is available, and, if it is, decoded enough data to play back that position, the user agent must queue a task to fire a simple event called `seeking` at the element.

10. If the seek was in response to a DOM method call or setting of a DOM attribute, then continue the script. The remainder of these steps must be run asynchronously.

11. The user agent must wait until it 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.

12. The `seeking` DOM attribute must be set to false.

13. The user agent must queue a task to fire a simple event called `seeked` at the element.

The `seekable` attribute must return a new static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent is able to seek to, at the time the attribute is evaluated.

If the user agent can seek to anywhere in the media resource, e.g. because it a simple movie file and the user agent and the server support HTTP Range requests, then the attribute would return an object with one range, whose start is the time of the first frame (typically zero), and whose end is the same as the time of the first frame plus the `duration` attribute's value (which would equal the time of the last frame).

The range might be continuously changing, e.g. if the user agent is buffering a sliding window on an infinite stream. This is the behavior seen with DVRs viewing live TV, for instance.

Media resources might be internally scripted or interactive. Thus, a media element could play in a non-linear fashion. If this happens, the user agent must act as if the algorithm for seeking was used whenever the current playback position changes in a discontinuous fashion (so that the relevant events fire).

4.8.10.10 User interface

The `controls` attribute is a boolean attribute. If present, it indicates that the author has not provided a scripted controller and would like the user agent to provide its own set of controls.

If the attribute is present, or if scripting is disabled for the media element, then the user agent should **expose a user interface to the user**. This user interface should include features to begin playback, pause playback, seek to an arbitrary position in the content (if the content supports arbitrary seeking), change the volume, and show the media content in manners more suitable to the user (e.g. full-screen video or in an independent resizable window). Other controls may also be made available.

If the attribute is absent, then the user agent should avoid making a user interface available that could conflict with an author-provided user interface. User agents may make the following features available, however, even when the attribute is absent:

User agents may provide controls to affect playback of the media resource (e.g. play, pause, seeking, and volume controls), but such features should not interfere with the page's normal rendering. For example, such features could be exposed in the media element's context menu.

Where possible (specifically, for starting, stopping, pausing, and unpausing playback, for muting or changing the volume of the audio, and for seeking), user interface features exposed by the user agent must be implemented in terms of the DOM API described above, so that, e.g., all the same events fire.

The `controls` DOM attribute must reflect the content attribute of the same name.

---

media **.** `volume` **[** **=** ***value*** **]**

Returns the current playback volume, as a number in the range 0.0 to 1.0, where 0.0 is the quietest and 1.0 the loudest.

Can be set, to change the volume.

Throws an INDEX_SIZE_ERR if the new value is not in the range 0.0 .. 1.0.

***media* . muted [= *value*]**

Returns true if audio is muted, overriding the volume attribute, and false if the volume attribute is being honored.

Can be set, to change whether the audio is muted or not.

The **volume** attribute must return the playback volume of any audio portions of the media element, in the range 0.0 (silent) to 1.0 (loudest). Initially, the volume must be 1.0, but user agents may remember the last set value across sessions, on a per-site basis or otherwise, so the volume may start at other values. On setting, if the new value is in the range 0.0 to 1.0 inclusive, the attribute must be set to the new value and the playback volume must be correspondingly adjusted as soon as possible after setting the attribute, with 0.0 being silent, and 1.0 being the loudest setting, values in between increasing in loudness. The range need not be linear. The loudest setting may be lower than the system's loudest possible setting; for example the user could have set a maximum volume. If the new value is outside the range 0.0 to 1.0 inclusive, then, on setting, an INDEX_SIZE_ERR exception must be raised instead.

The **muted** attribute must return true if the audio channels are muted and false otherwise. Initially, the audio channels should not be muted (false), but user agents may remember the last set value across sessions, on a per-site basis or otherwise, so the muted state may start as muted (true). On setting, the attribute must be set to the new value; if the new value is true, audio playback for this media resource must then be muted, and if false, audio playback must then be enabled.

Whenever either the muted or volume attributes are changed, the user agent must queue a task to fire a simple event called volumechange at the media element.

4.8.10.11 Time ranges

Objects implementing the TimeRanges interface represent a list of ranges (periods) of time.

```
interface TimeRanges {
  readonly attribute unsigned long length;
  float start(in unsigned long index);
  float end(in unsigned long index);
};
```

***media* . length**

Returns the number of ranges in the object.

***time* = *media* . start(*index*)**

Returns the time for the start of the range with the given index.

Throws an INDEX_SIZE_ERR if the index is out of range.

***time* = *media* . end(*index*)**

Returns the time for the end of the range with the given index.

Throws an INDEX_SIZE_ERR if the index is out of range.

The **length** DOM attribute must return the number of ranges represented by the object.

The **start(*index*)** method must return the position of the start of the *index*th range represented by the object, in seconds measured from the start of the timeline that the object covers.

The **end(*index*)** method must return the position of the end of the *index*th range represented by the object, in seconds measured from the start of the timeline that the object covers.

These methods must raise INDEX_SIZE_ERR exceptions if called with an *index* argument greater than or equal to the number of ranges represented by the object.

When a TimeRanges object is said to be a **normalized TimeRanges object**, the ranges it represents must obey the following criteria:

- The start of a range must be greater than the end of all earlier ranges.
- The start of a range must be less than the end of that same range.

In other words, the ranges in such an object are ordered, don't overlap, aren't empty, and don't touch (adjacent ranges are folded into one bigger range).

The timelines used by the objects returned by the buffered, seekable and played DOM attributes of media elements must be the same as that element's media resource's timeline.

4.8.10.12 Event summary

The following events fire on media elements as part of the processing model described above:

Event name	Interface	Dispatched when...	Preconditions
loadstart	ProgressEvent [PROGRESS]	The user agent begins looking for media data, as part of the resource selection algorithm.	networkState equals NETWORK_LOADING
progress	ProgressEvent [PROGRESS]	The user agent is fetching media data.	networkState equals NETWORK_LOADING
suspend	ProgressEvent [PROGRESS]	The user agent is intentionally not currently fetching media data, but does not have the entire media resource downloaded.	networkState equals NETWORK_IDLE
load	ProgressEvent [PROGRESS]	The user agent finishes fetching the entire media resource.	networkState equals NETWORK_LOADED

Event name	Interface	Dispatched when...	Preconditions
abort	ProgressEvent [PROGRESS]	The user agent stops fetching the media data before it is completely downloaded.	error is an object with the code MEDIA_ERR_ABORTED. networkState equals either NETWORK_EMPTY or NETWORK_LOADED, depending on when the download was aborted.
error	ProgressEvent [PROGRESS]	An error occurs while fetching the media data.	error is an object with the code MEDIA_ERR_NETWORK or higher. networkState equals either NETWORK_EMPTY or NETWORK_LOADED, depending on when the download was aborted.
loadend	ProgressEvent [PROGRESS]	The user agent stops fetching the media data, for whatever reason.	One of load, abort, or error has just fired.
emptied	Event	A media element whose networkState was previously not in the NETWORK_EMPTY state has just switched to that state (either because of a fatal error during load that's about to be reported, or because the load() method was invoked while the resource selection algorithm was already running, in which case it is fired synchronously during the load() method call).	networkState is NETWORK_EMPTY; all the DOM attributes are in their initial states.
stalled	ProgressEvent	The user agent is trying to fetch media data, but data is unexpectedly not forthcoming.	networkState is NETWORK_LOADING.
play	Event	Playback has begun. Fired after the play() method has returned.	paused is newly false.
pause	Event	Playback has been paused. Fired after the pause method has returned.	paused is newly true.
loadedmetadata	Event	The user agent has just determined the duration and dimensions of the media resource.	readyState is newly equal to HAVE_METADATA or greater for the first time.

Event name	Interface	Dispatched when...	Preconditions
loadeddata	Event	The user agent can render the media data at the current playback position for the first time.	readyState newly increased to HAVE_CURRENT_DATA or greater for the first time.
waiting	Event	Playback has stopped because the next frame is not available, but the user agent expects that frame to become available in due course.	readyState is newly equal to or less than HAVE_CURRENT_DATA, and paused is false. Either seeking is true, or the current playback position is not contained in any of the ranges in buffered. It is possible for playback to stop for two other reasons without paused being false, but those two reasons do not fire this event: maybe playback ended, or playback stopped due to errors.
playing	Event	Playback has started.	readyState is newly equal to or greater than HAVE_FUTURE_DATA, paused is false, seeking is false, or the current playback position is contained in one of the ranges in buffered.
canplay	Event	The user agent can resume playback of the media data, but estimates that if playback were to be started now, the media resource could not be rendered at the current playback rate up to its end without having to stop for further buffering of content.	readyState newly increased to HAVE_FUTURE_DATA or greater.
canplaythrough	Event	The user agent estimates that if playback were to be started now, the media resource could be rendered at the current playback rate all the way to its end without having to stop for further buffering.	readyState is newly equal to HAVE_ENOUGH_DATA.
seeking	Event	The seeking DOM attribute changed to true and the seek operation is taking long	

Event name	Interface	Dispatched when...	Preconditions
		enough that the user agent has time to fire the event.	
seeked	`Event`	The `seeking` DOM attribute changed to false.	
timeupdate	`Event`	The current playback position changed as part of normal playback or in an especially interesting way, for example discontinuously.	
ended	`Event`	Playback has stopped because the end of the media resource was reached.	`currentTime` equals the end of the media resource; `ended` is true.
ratechange	`Event`	Either the `defaultPlaybackRate` or the `playbackRate` attribute has just been updated.	
durationchange	`Event`	The `duration` attribute has just been updated.	
volumechange	`Event`	Either the `volume` attribute or the `muted` attribute has changed. Fired after the relevant attribute's setter has returned.	

4.8.10.13 Security and privacy considerations

The main security and privacy implications of the `video` and `audio` elements come from the ability to embed media cross-origin. There are two directions that threats can flow: from hostile content to a victim page, and from a hostile page to victim content.

If a victim page embeds hostile content, the threat is that the content might contain scripted code that attempts to interact with the `Document` that embeds the content. To avoid this, user agents must ensure that there is no access from the content to the embedding page. In the case of media content that uses DOM concepts, the embedded content must be treated as if it was in its own unrelated top-level browsing context.

> For instance, if an SVG animation was embedded in a `video` element, the user agent would not give it access to the DOM of the outer page. From the perspective of scripts in the SVG resource, the SVG file would appear to be in a lone top-level browsing context with no parent.

If a hostile page embeds victim content, the threat is that the embedding page could obtain information from the content that it would not otherwise have access to. The API does expose some information: the existence of the media, its type, its duration, its size, and the performance characteristics of its host. Such information is already potentially problematic, but in practice the same information can more or less be obtained using the `img` element, and so it has been deemed acceptable.

However, significantly more sensitive information could be obtained if the user agent further exposes metadata within the content such as subtitles or chapter titles. This version of the API does not expose such information. Future extensions to this API will likely reuse a mechanism such as CORS to check that the embedded content's site has opted in to exposing such information. [CORS]

> An attacker could trick a user running within a corporate network into visiting a site that attempts to load a video from a previously leaked location on the corporation's intranet. If such a video included confidential plans for a new product, then being able to read the subtitles would present a confidentiality breach.