Web Audio API

Editor's Draft, 18 July 2023

▼ **More details about this document**

This version:
https://webaudio.github.io/web-audio-api/

Latest published version:
https://www.w3.org/TR/webaudio/

Previous Versions:
https://www.w3.org/TR/2021/CR-webaudio-20210114/
https://www.w3.org/TR/2020/CR-webaudio-20200611/
https://www.w3.org/TR/2018/CR-webaudio-20180918/
https://www.w3.org/TR/2018/WD-webaudio-20180619/
https://www.w3.org/TR/2015/WD-webaudio-20151208/
https://www.w3.org/TR/2013/WD-webaudio-20131010/
https://www.w3.org/TR/2012/WD-webaudio-20121213/
https://www.w3.org/TR/2012/WD-webaudio-20120802/
https://www.w3.org/TR/2012/WD-webaudio-20120315/
https://www.w3.org/TR/2011/WD-webaudio-20111215/

Feedback:
public-audio@w3.org with subject line "`[webaudio] … message topic …`" (archives)
GitHub

Implementation Report:
implementation-report.html

Test Suite:
https://github.com/web-platform-tests/wpt/tree/master/webaudio

Editors:
Paul Adenot (Mozilla (https://www.mozilla.org/))
Hongchan Choi (Google (https://www.google.com/))

Former Editors:
Raymond Toy (until Oct 2018)
Chris Wilson (Until Jan 2016)
Chris Rogers (Until Aug 2013)

Abstract

This specification describes a high-level Web API for processing and synthesizing audio in web applications. The primary paradigm is of an audio routing graph, where a number of `AudioNode` objects are connected together to define the overall audio rendering. The actual processing will primarily take place in the underlying implementation (typically optimized Assembly / C / C++ code), but direct script processing and synthesis is also supported.

The Introduction section covers the motivation behind this specification.

This API is designed to be used in conjunction with other APIs and elements on the web platform, notably: XMLHttpRequest [XHR] (using the `responseType` and `response` attributes). For games and interactive applications, it is anticipated to be used with the `canvas` 2D [2dcontext] and WebGL [WEBGL] 3D graphics APIs.

Status of this document

This is a public copy of the editors' draft. It is provided for discussion only and may change at any moment. Its publication here does not imply endorsement of its contents by W3C. Don't cite this document other than as work in progress.

Error preparing HTML-CSS output (preProcess)

If you wish to make comments regarding this document, please file an issue on the specification repository or send them to public-audio@w3.org (subscribe, archives).

This document was produced by the Web Audio Working Group.

This document was produced by groups operating under the 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 12 June 2023 W3C Process Document.

Table of Contents

Error preparing HTML-CSS output (preProcess) `flineAudioCompletionEventInit`

Error preparing HTML-CSS output (preProcess)

Error preparing HTML-CSS output (preProcess) `treamTrackAudioSourceNode` Interface

Error preparing HTML-CSS output (preProcess)

Introduction

Audio on the web has been fairly primitive up to this point and until very recently has had to be delivered through plugins such as Flash and QuickTime. The introduction of the `<audio>` element in HTML5 is very important, allowing for basic streaming audio playback. But, it is not powerful enough to handle more complex audio applications. For sophisticated web-based games or interactive applications, another solution is required. It is a goal of this specification to include the capabilities found in modern game audio engines as well as some of the mixing, processing, and filtering tasks that are found in modern desktop audio production applications.

The APIs have been designed with a wide variety of use cases [webaudio-usecases] in mind. Ideally, it should be able to support *any* use case which could reasonably be implemented with an optimized C++ engine controlled via script and run in a browser. That said, modern desktop audio software can have very advanced capabilities, some of which would be difficult or impossible to build with this system. Apple's Logic Audio is one such application which has support for external MIDI controllers, arbitrary plugin audio effects and synthesizers, highly optimized direct-to-disk audio file reading/writing, tightly integrated time-stretching, and so on. Nevertheless, the proposed system will be quite capable of supporting a large range of reasonably complex games and interactive applications, including musical ones. And it can be a very good complement to the more advanced graphics features offered by WebGL. The API has been designed so that more advanced capabilities can be added at a later time.

Features

The API supports these primary features:

- Modular routing for simple or complex mixing/effect architectures.
- High dynamic range, using 32-bit floats for internal processing.
- Sample-accurate scheduled sound playback with low latency for musical applications requiring a very high degree of rhythmic precision such as drum machines and sequencers. This also includes the possibility of dynamic creation of effects.
- Automation of audio parameters for envelopes, fade-ins / fade-outs, granular effects, filter sweeps, LFOs etc.
- Flexible handling of channels in an audio stream, allowing them to be split and merged.
- Processing of audio sources from an `<audio>` or `<video>` `media element`.
- Processing live audio input using a `MediaStream` from `getUserMedia()`.
- Integration with WebRTC
    - Processing audio received from a remote peer using a `MediaStreamTrackAudioSourceNode` and [webrtc].
    - Sending a generated or processed audio stream to a remote peer using a `MediaStreamAudioDestinationNode` and [webrtc].
- Audio stream synthesis and processing directly using scripts.

Error preparing HTML-CSS output (preProcess)

- Spatialized audio supporting a wide range of 3D games and immersive environments:
  - Panning models: equalpower, HRTF, pass-through
  - Distance Attenuation
  - Sound Cones
  - Obstruction / Occlusion
  - Source / Listener based
- A convolution engine for a wide range of linear effects, especially very high-quality room effects. Here are some examples of possible effects:
  - Small / large room
  - Cathedral
  - Concert hall
  - Cave
  - Tunnel
  - Hallway
  - Forest
  - Amphitheater
  - Sound of a distant room through a doorway
  - Extreme filters
  - Strange backwards effects
  - Extreme comb filter effects
- Dynamics compression for overall control and sweetening of the mix
- Efficient real-time time-domain and frequency-domain analysis / music visualizer support.
- Efficient biquad filters for lowpass, highpass, and other common filters.
- A Waveshaping effect for distortion and other non-linear effects
- Oscillators

Modular Routing

Modular routing allows arbitrary connections between different `AudioNode` objects. Each node can have ***inputs*** and/or ***outputs***. A ***source node*** has no inputs and a single output. A ***destination node*** has one input and no outputs. Other nodes such as filters can be placed between the source and destination nodes. The developer doesn't have to worry about low-level stream format details when two objects are connected together; the right thing just happens. For example, if a mono audio stream is connected to a stereo input it should just mix to left and right channels appropriately.

In the simplest case, a single source can be routed directly to the output. All routing occurs within an `AudioContext` containing a single `AudioDestinationNode`:

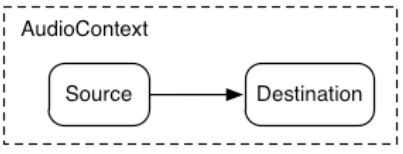


Figure 1 *A simple example of modular routing.*

Error preparing HTML-CSS output (preProcess)

Illustrating this simple routing, here's a simple example playing a single sound:

```
EXAMPLE 1
const context = new AudioContext();

function playSound() {
    const source = context.createBufferSource();
    source.buffer = dogBarkingBuffer;
    source.connect(context.destination);
    source.start(0);
}
```

Here's a more complex example with three sources and a convolution reverb send with a dynamics compressor at the final output stage:

Error preparing HTML-CSS output (preProcess)

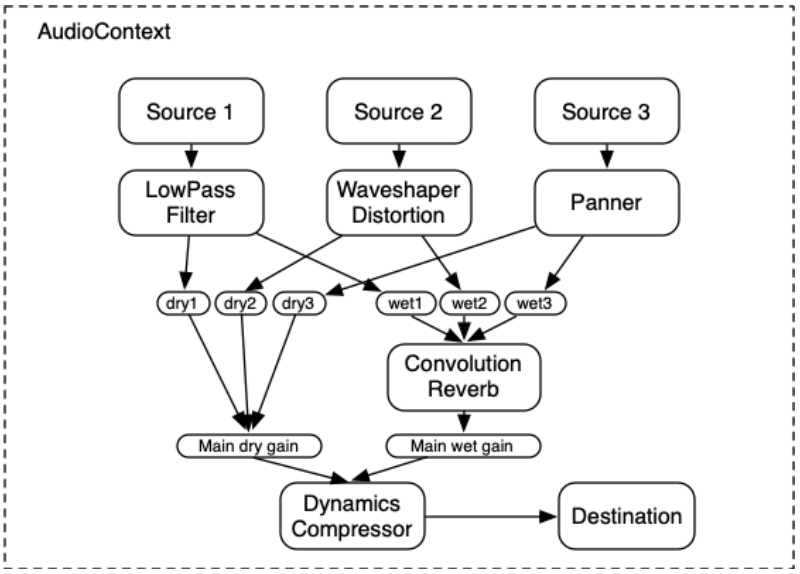


Figure 2 *A more complex example of modular routing.*

EXAMPLE 2

```
let context;
let compressor;
let reverb;

let source1, source2, source3;

let lowpassFilter;
let waveShaper;
let panner;

let dry1, dry2, dry3;
let wet1, wet2, wet3;

let mainDry;
let mainWet;

function setupRoutingGraph () {
    context = new AudioContext();

    // Create the effects nodes.
    lowpassFilter = context.createBiquadFilter();
    waveShaper = context.createWaveShaper();
    panner = context.createPanner();
    compressor = context.createDynamicsCompressor();
    reverb = context.createConvolver();

    // Create main wet and dry.
    mainDry = context.createGain();
    mainWet = context.createGain();

    // Connect final compressor to final destination.
    compressor.connect(context.destination);

    // Connect main dry and wet to compressor.
    mainDry.connect(compressor);
    mainWet.connect(compressor);

    // Connect reverb to main wet.
    reverb.connect(mainWet);

    // Create a few sources.
ontext.createBufferSource();
```

Error preparing HTML-CSS output (preProcess)

```
    source2 = context.createBufferSource();
    source3 = context.createOscillator();

    source1.buffer = manTalkingBuffer;
    source2.buffer = footstepsBuffer;
    source3.frequency.value = 440;

    // Connect source1
    dry1 = context.createGain();
    wet1 = context.createGain();
    source1.connect(lowpassFilter);
    lowpassFilter.connect(dry1);
    lowpassFilter.connect(wet1);
    dry1.connect(mainDry);
    wet1.connect(reverb);

    // Connect source2
    dry2 = context.createGain();
    wet2 = context.createGain();
    source2.connect(waveShaper);
    waveShaper.connect(dry2);
    waveShaper.connect(wet2);
    dry2.connect(mainDry);
    wet2.connect(reverb);

    // Connect source3
    dry3 = context.createGain();
    wet3 = context.createGain();
    source3.connect(panner);
    panner.connect(dry3);
    panner.connect(wet3);
    dry3.connect(mainDry);
    wet3.connect(reverb);

    // Start the sources now.
    source1.start(0);
    source2.start(0);
    source3.start(0);
}
```

Modular routing also permits the output of `AudioNode`s to be routed to an `AudioParam` parameter that controls the behavior of a different `AudioNode`. In this scenario, the output of a node can act as a modulation signal rather than an inp signal.

Error preparing HTML-CSS output (preProcess)

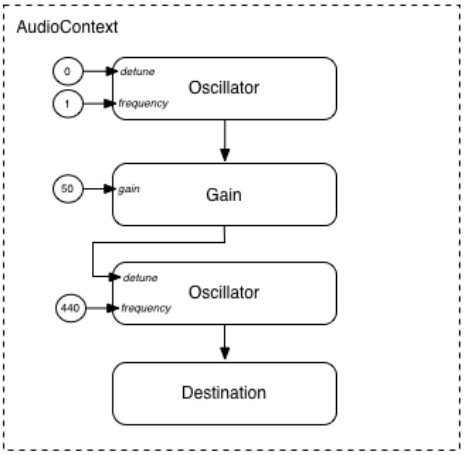


Figure 3 *Modular routing illustrating one Oscillator modulating the frequency of another.*

```
EXAMPLE 3
function setupRoutingGraph() {
    const context = new AudioContext();

    // Create the low frequency oscillator that supplies the modulation signal
    const lfo = context.createOscillator();
    lfo.frequency.value = 1.0;

    // Create the high frequency oscillator to be modulated
    const hfo = context.createOscillator();
    hfo.frequency.value = 440.0;

    // Create a gain node whose gain determines the amplitude of the modulation signal
    const modulationGain = context.createGain();
    modulationGain.gain.value = 50;

    // Configure the graph and start the oscillators
    lfo.connect(modulationGain);
    modulationGain.connect(hfo.detune);
    hfo.connect(context.destination);
    hfo.start(0);
    lfo.start(0);
}
```

API Overview

The interfaces defined are:

- An AudioContext interface, which contains an audio signal graph representing connections between `AudioNode`s.
- An `AudioNode` interface, which represents audio sources, audio outputs, and intermediate processing modules. `AudioNode`s can be dynamically connected together in a modular fashion. `AudioNode`s exist in the context of an `AudioContext`.
- An `AnalyserNode` interface, an `AudioNode` for use with music visualizers, or other visualization applications.
- An `AudioBuffer` interface, for working with memory-resident audio assets. These can represent one-shot sounds, or longer audio clips.
- An `AudioBufferSourceNode` interface, an `AudioNode` which generates audio from an AudioBuffer.
- An `AudioDestinationNode` interface, an `AudioNode` subclass representing the final destination for all rendered audio.
- An `AudioParam` interface, for controlling an individual aspect of an `AudioNode`'s functioning, such as volume.
- r interface, which works with a `PannerNode` for spatialization.

- An `AudioWorklet` interface representing a factory for creating custom nodes that can process audio directly using scripts.
- An `AudioWorkletGlobalScope` interface, the context in which AudioWorkletProcessor processing scripts run.
- An `AudioWorkletNode` interface, an `AudioNode` representing a node processed in an AudioWorkletProcessor.
- An `AudioWorkletProcessor` interface, representing a single node instance inside an audio worker.
- A `BiquadFilterNode` interface, an `AudioNode` for common low-order filters such as:
  - Low Pass
  - High Pass
  - Band Pass
  - Low Shelf
  - High Shelf
  - Peaking
  - Notch
  - Allpass
- A `ChannelMergerNode` interface, an `AudioNode` for combining channels from multiple audio streams into a single audio stream.
- A `ChannelSplitterNode` interface, an `AudioNode` for accessing the individual channels of an audio stream in the routing graph.
- A `ConstantSourceNode` interface, an `AudioNode` for generating a nominally constant output value with an `AudioParam` to allow automation of the value.
- A `ConvolverNode` interface, an `AudioNode` for applying a real-time linear effect (such as the sound of a concert hall).
- A `DelayNode` interface, an `AudioNode` which applies a dynamically adjustable variable delay.
- A `DynamicsCompressorNode` interface, an `AudioNode` for dynamics compression.
- A `GainNode` interface, an `AudioNode` for explicit gain control.
- An `IIRFilterNode` interface, an `AudioNode` for a general IIR filter.
- A `MediaElementAudioSourceNode` interface, an `AudioNode` which is the audio source from an `<audio>`, `<video>`, or other media element.
- A `MediaStreamAudioSourceNode` interface, an `AudioNode` which is the audio source from a `MediaStream` such as live audio input, or from a remote peer.
- A `MediaStreamTrackAudioSourceNode` interface, an `AudioNode` which is the audio source from a `MediaStreamTrack`.
- A `MediaStreamAudioDestinationNode` interface, an `AudioNode` which is the audio destination to a `MediaStream` sent to a remote peer.
- A `PannerNode` interface, an `AudioNode` for spatializing / positioning audio in 3D space.
- A `PeriodicWave` interface for specifying custom periodic waveforms for use by the `OscillatorNode`.
- An `OscillatorNode` interface, an `AudioNode` for generating a periodic waveform.
- A `StereoPannerNode` interface, an `AudioNode` for equal-power positioning of audio input in a stereo stream.
- A `WaveShaperNode` interface, an `AudioNode` which applies a non-linear waveshaping effect for distortion and other more subtle warming effects.

There are also several features that have been deprecated from the Web Audio API but not yet removed, pending implementation experience of their replacements:

- A `ScriptProcessorNode` interface, an `AudioNode` for generating or processing audio directly using scripts.
- An `AudioProcessingEvent` interface, which is an event type used with `ScriptProcessorNode` objects.

Error preparing HTML-CSS output (preProcess)

§ 1. The Audio API

§ 1.1. The `BaseAudioContext` Interface

This interface represents a set of `AudioNode` objects and their connections. It allows for arbitrary routing of signals to an `AudioDestinationNode`. Nodes are created from the context and are then connected together.

`BaseAudioContext` is not instantiated directly, but is instead extended by the concrete interfaces `AudioContext` (for real-time rendering) and `OfflineAudioContext` (for offline rendering).

`BaseAudioContext` are created with an internal slot ***[[pending promises]]*** that is an initially empty ordered list of promises.

Each `BaseAudioContext` has a unique media element event task source. Additionally, a `BaseAudioContext` has two private slots ***[[rendering thread state]]*** and ***[[control thread state]]*** that take values from `AudioContextState`, and that are both

✓ MDN

PROPOSED CORRECTION ISSUE 2373-1. Use new Web IDL buffer primitives

~~initialy~~ initially

Show Change | Show Current | Show Future

✓ MDN

set to "suspended".

```
enum AudioContextState {
    "suspended",
    "running",
    "closed"
};
```

`AudioContextState` enumeration description

Enum value	Description
"***suspended***"	This context is currently suspended (context time is not proceeding, audio hardware may be powered down/released).
"***running***"	Audio is being processed.
"***closed***"	This context has been released, and can no longer be used to process audio. All system audio resources have been released.

✓ MDN

```
callback DecodeErrorCallback = undefined (DOMException error);

callback DecodeSuccessCallback = undefined (AudioBuffer decodedData);
```

✓ MDN

```
[Exposed=Window]
interface BaseAudioContext : EventTarget {
```

✓ MDN

```
    readonly attribute AudioDestinationNode destination;
    readonly attribute float sampleRate;
    readonly attribute double currentTime;
    readonly attribute AudioListener listener;
    readonly attribute AudioContextState state;
    [SameObject, SecureContext]
    readonly attribute AudioWorklet audioWorklet;
    attribute EventHandler onstatechange;
```

✓ MDN

```
    AnalyserNode createAnalyser ();
    BiquadFilterNode createBiquadFilter ();
    AudioBuffer createBuffer (unsigned long numberOfChannels,
                              unsigned long length,
                              float sampleRate);
```

✓ MDN

```
    AudioBufferSourceNode createBufferSource ();
```

Error preparing HTML-CSS output (preProcess)

```
ode createChannelMerger (optional unsigned long numberOfInputs = 6);
    ChannelSplitterNode createChannelSplitter (
```

```
        optional unsigned long numberOfOutputs = 6);
    ConstantSourceNode createConstantSource ();
    ConvolverNode createConvolver ();
    DelayNode createDelay (optional double maxDelayTime = 1.0);
    DynamicsCompressorNode createDynamicsCompressor ();
    GainNode createGain ();
    IIRFilterNode createIIRFilter (sequence<double> feedforward,
                                   sequence<double> feedback);
    OscillatorNode createOscillator ();
    PannerNode createPanner ();
    PeriodicWave createPeriodicWave (sequence<float> real,
                                     sequence<float> imag,
                                     optional PeriodicWaveConstraints constraints = {});
    ScriptProcessorNode createScriptProcessor(
        optional unsigned long bufferSize = 0,
        optional unsigned long numberOfInputChannels = 2,
        optional unsigned long numberOfOutputChannels = 2);
    StereoPannerNode createStereoPanner ();
    WaveShaperNode createWaveShaper ();

    Promise<AudioBuffer> decodeAudioData (
        ArrayBuffer audioData,
        optional DecodeSuccessCallback? successCallback,
        optional DecodeErrorCallback? errorCallback);
};
```

MDN MDN MDN

§ 1.1.1. Attributes

***audioWorklet*, of type AudioWorklet, readonly**

Allows access to the `Worklet` object that can import a script containing `AudioWorkletProcessor` class definitions via the algorithms defined by [HTML] and `AudioWorklet`.

***currentTime*, of type double, readonly**

This is the time in seconds of the sample frame immediately following the last sample-frame in the block of audio most recently processed by the context's rendering graph. If the context's rendering graph has not yet processed a block of audio, then `currentTime` has a value of zero.

In the time coordinate system of `currentTime`, the value of zero corresponds to the first sample-frame in the first block processed by the graph. Elapsed time in this system corresponds to elapsed time in the audio stream generated by the `BaseAudioContext`, which may not be synchronized with other clocks in the system. (For an `OfflineAudioContext`, since the stream is not being actively played by any device, there is not even an approximation to real time.)

MDN

All scheduled times in the Web Audio API are relative to the value of `currentTime`.

MDN

When the `BaseAudioContext` is in the "`running`" state, the value of this attribute is monotonically increasing and is updated by the rendering thread in uniform increments, corresponding to one render quantum. Thus, for a running context, `currentTime` increases steadily as the system processes audio blocks, and always represents the time of the start of the next audio block to be processed. It is also the earliest possible time when any change scheduled in the current state might take effect.

`currentTime` MUST be read atomically on the control thread before being returned.

***destination*, of type AudioDestinationNode, readonly**

An `AudioDestinationNode` with a single input representing the final destination for all audio. Usually this will represent the actual audio hardware. All `AudioNode`s actively rendering audio will directly or indirectly connect to `destination`.

***listener*, of type AudioListener, readonly**

An `AudioListener` which is used for 3D spatialization.

***onstatechange*, of type EventHandler**

MDN

A property used to set an event handler for an event that is dispatched to `BaseAudioContext` when the state of the

Error preparing HTML-CSS output (preProcess)

anged (i.e. when the corresponding promise would have resolved). The event type of this event

handler is ***statechange***. An event that uses the `Event` interface will be dispatched to the event handler, which can query the AudioContext's state directly. A newly-created AudioContext will always begin in the `suspended` state, and a state change event will be fired whenever the state changes to a different state. This event is fired before the `complete` event is fired.

***sampleRate*, of type float, readonly**
The sample rate (in sample-frames per second) at which the `BaseAudioContext` handles audio. It is assumed that all `AudioNode`s in the context run at this rate. In making this assumption, sample-rate converters or "varispeed" processors are not supported in real-time processing. The ***Nyquist frequency*** is half this sample-rate value.

***state*, of type AudioContextState, readonly**
Describes the current state of the `BaseAudioContext`. Getting this attribute returns the contents of the `[[control thread state]]` slot.

✓ MDN

§ 1.1.2. Methods

createAnalyser()
Factory method for an `AnalyserNode`.

✓ MDN

No parameters.
Return type: `AnalyserNode`

createBiquadFilter()
Factory method for a `BiquadFilterNode` representing a second order filter which can be configured as one of several common filter types.

✓ MDN

No parameters.
Return type: `BiquadFilterNode`

createBuffer(numberOfChannels, length, sampleRate)
Creates an AudioBuffer of the given size. The audio data in the buffer will be zero-initialized (silent). ⌛ A `NotSupportedError` exception MUST be thrown if any of the arguments is negative, zero, or outside its nominal range.

Arguments for the BaseAudioContext.createBuffer() method.

Parameter	Type	Nullable	Optional	Description
numberOfChannels	`unsigned long`	✗	✗	Determines how many channels the buffer will have. An implementation MUST support at least 32 channels.
length	`unsigned long`	✗	✗	Determines the size of the buffer in sample-frames. This MUST be at least 1.
sampleRate	`float`	✗	✗	Describes the sample-rate of the linear PCM audio data in the buffer in sample-frames per second. An implementation MUST support sample rates in at least the range 8000 to 96000.

✓ MDN
✓ MDN
✓ MDN

Return type: `AudioBuffer`

createBufferSource()
Factory method for a `AudioBufferSourceNode`.

No parameters.
Return type: `AudioBufferSourceNode`

createChannelMerger(numberOfInputs)
Factory method for a `ChannelMergerNode` representing a channel merger. ⌛ An `IndexSizeError` exception MUST be thrown if `numberOfInputs` is less than 1 or is greater than the number of supported channels.

Arguments for the BaseAudioContext.createChannelMerger(numberOfInputs) method.

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
numberOfInputs	unsigned long	✗	✓	Determines the number of inputs. Values of up to 32 MUST be supported. If not specified, then 6 will be used.

Return type: ChannelMergerNode

createChannelSplitter(numberOfOutputs)

Factory method for a ChannelSplitterNode representing a channel splitter. ⌛ An IndexSizeError exception MUST be thrown if numberOfOutputs is less than 1 or is greater than the number of supported channels.

✓ MDN

Arguments for the BaseAudioContext.createChannelSplitter(numberOfOutputs) method.

Parameter	Type	Nullable	Optional	Description
numberOfOutputs	unsigned long	✗	✓	The number of outputs. Values of up to 32 MUST be supported. If not specified, then 6 will be used.

✓ MDN

Return type: ChannelSplitterNode

createConstantSource()

✓ MDN

Factory method for a ConstantSourceNode.

No parameters.

Return type: ConstantSourceNode

createConvolver()

Factory method for a ConvolverNode.

No parameters.

Return type: ConvolverNode

createDelay(maxDelayTime)

Factory method for a DelayNode. The initial default delay time will be 0 seconds.

Arguments for the BaseAudioContext.createDelay(maxDelayTime) method.

Parameter	Type	Nullable	Optional	Description
maxDelayTime	double	✗	✓	Specifies the maximum delay time in seconds allowed for the delay line. ⌛ If specified, this value MUST be greater than zero and less than three minutes or a NotSupportedError exception MUST be thrown. If not specified, then 1 will be used.

Return type: DelayNode

createDynamicsCompressor()

Factory method for a DynamicsCompressorNode.

No parameters.

Return type: DynamicsCompressorNode

createGain()

Factory method for GainNode.

No parameters.

Return type: GainNode

createIIRFilter(feedforward, feedback)

Arguments for the BaseAudioContext.createIIRFilter() method.

Parameter	Type	Nullable	Optional	Description

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
feedforward	sequence<double>	✗	✗	An array of the feedforward (numerator) coefficients for the transfer function of the IIR filter. The maximum length of this array is 20. If all of the values are zero, ⌛ an InvalidStateError *MUST* be thrown. ⌛ A NotSupportedError *MUST* be thrown if the array length is 0 or greater than 20.
feedback	sequence<double>	✗	✗	An array of the feedback (denominator) coefficients for the transfer function of the IIR filter. The maximum length of this array is 20. If the first element of the array is 0, ⌛ an InvalidStateError *MUST* be thrown. ⌛ A NotSupportedError *MUST* be thrown if the array length is 0 or greater than 20.

Return type: IIRFilterNode

createOscillator()

Factory method for an OscillatorNode.

No parameters.

Return type: OscillatorNode

createPanner()

Factory method for a PannerNode.

No parameters.

Return type: PannerNode

createPeriodicWave(real, imag, constraints)

Factory method to create a PeriodicWave.

When calling this method, execute these steps:

1. ⌛ If real and imag are not of the same length, an IndexSizeError MUST be thrown.
2. Let *o* be a new object of type PeriodicWaveOptions.
3. Respectively set the real and imag parameters passed to this factory method to the attributes of the same name on *o*.
4. Set the disableNormalization attribute on *o* to the value of the disableNormalization attribute of the constraints attribute passed to the factory method.
5. Construct a new PeriodicWave *p*, passing the BaseAudioContext this factory method has been called on as a first argument, and *o*.
6. Return *p*.

Arguments for the BaseAudioContext.createPeriodicWave() method.

Parameter	Type	Nullable	Optional	Description
real	sequence<float>	✗	✗	A sequence of cosine parameters. See its real constructor argument for a more detailed description.
imag	sequence<float>	✗	✗	A sequence of sine parameters. See its imag constructor

Parameter	Type	Nullable	Optional	Description
				argument for a more detailed description.
constraints	PeriodicWaveConstraints	✗	✓	If not given, the waveform is normalized. Otherwise, the waveform is normalized according the value given by `constraints`.

Return type: PeriodicWave

createScriptProcessor(bufferSize, numberOfInputChannels, numberOfOutputChannels)

Factory method for a ScriptProcessorNode. This method is DEPRECATED, as it is intended to be replaced by AudioWorkletNode. Creates a ScriptProcessorNode for direct audio processing using scripts. ⌛ An IndexSizeError exception MUST be thrown if bufferSize or numberOfInputChannels or numberOfOutputChannels are outside the valid range.

It is invalid for both numberOfInputChannels and numberOfOutputChannels to be zero. ⌛ In this case an IndexSizeError MUST be thrown.

Arguments for the BaseAudioContext.createScriptProcessor(bufferSize, numberOfInputChannels, numberOfOutputChannels) method.

Parameter	Type	Nullable	Optional	Description
bufferSize	unsigned long	✗	✓	The bufferSize parameter determines the buffer size in units of sample-frames. If it's not passed in, or if the value is 0, then the implementation will choose the best buffer size for the given environment, which will be constant power of 2 throughout the lifetime of the node. Otherwise if the author explicitly specifies the bufferSize, it *MUST* be one of the following values: 256, 512, 1024, 2048, 4096, 8192, 16384. This value controls how frequently the audioprocess event is dispatched and how many sample-frames need to be processed each call. Lower values for bufferSize will result in a lower (better) latency. Higher values will be necessary to avoid audio breakup and glitches. It is recommended for authors to not specify this buffer size and allow the implementation to pick a good buffer size to balance between latency and audio quality. If the value of this parameter is not one of the allowed power-of-2 values listed above, ⌛ an IndexSizeError *MUST* be thrown.
numberOfInputChannels	unsigned long	✗	✓	This parameter determines the number of channels for this node's input. The default value is 2. Values of up to 32 must be supported. ⌛ A NotSupportedError must be

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
				thrown if the number of channels is not supported.
numberOfOutputChannels	unsigned long	✗	✓	This parameter determines the number of channels for this node's output. The default value is 2. Values of up to 32 must be supported. ⏳ A NotSupportedError must be thrown if the number of channels is not supported.

Return type: ScriptProcessorNode

createStereoPanner()

Factory method for a StereoPannerNode.

No parameters.

Return type: StereoPannerNode

createWaveShaper()

Factory method for a WaveShaperNode representing a non-linear distortion.

No parameters.

Return type: WaveShaperNode

decodeAudioData(audioData, successCallback, errorCallback)

Asynchronously decodes the audio file data contained in the ArrayBuffer. The ArrayBuffer can, for example, be loaded from an XMLHttpRequest's response attribute after setting the responseType to "arraybuffer". Audio file data can be in any of the formats supported by the <audio> element. The buffer passed to decodeAudioData() has its content-type determined by sniffing, as described in [mimesniff].

Although the primary method of interfacing with this function is via its promise return value, the callback parameters are provided for legacy reasons.

CANDIDATE CORRECTION ISSUE 2321:
Encourage implementation to warn authors in case of a corrupted file. It isn't possible to throw because this would be a breaking change.
Show Change | Show Current | Show Future

Note: If the compressed audio data byte-stream is corrupted but the decoding can otherwise proceed, implementations are encouraged to warn authors for example via the developer tools.

⏳ When decodeAudioData is called, the following steps MUST be performed on the control thread:

1. If this's relevant global object's associated Document is not fully active then return a promise rejected with "InvalidStateError" DOMException.
2. Let *promise* be a new Promise.
3. **PROPOSED CORRECTION ISSUE 2361-1.** Use new Web IDL buffer primitives
   Next Change
   If ~~the operation IsDetachedBuffer (described in [ECMASCRIPT]) on audioData is false,~~ audioData is detached, execute the following steps:
   Show Change | Show Current | Show Future

   1. Append *promise* to [[pending promises]].
   2. **PROPOSED CORRECTION ISSUE 2361-2.** Use new Web IDL buffer primitives
      Previous Change | Next Change
      ~~Detach~~ Detach the audioData ArrayBuffer. ~~This operation is described in [ECMASCRIPT].~~ If this operations throws, jump to the step 3.
      Error preparing HTML-CSS output (preProcess) Change | Show Current | Show Future

3. Queue a decoding operation to be performed on another thread.

4. Else, execute the following error steps:

    1. Let *error* be a `DataCloneError`.
    2. Reject *promise* with *error*, and remove it from `[[pending promises]]`.
    3. Queue a media element task to invoke `errorCallback` with *error*.

5. Return *promise*.

When queuing a decoding operation to be performed on another thread, the following steps MUST happen on a thread that is not the control thread nor the rendering thread, called the ***decoding thread***.

NOTE: Multiple `decoding thread`s can run in parallel to service multiple calls to `decodeAudioData`.

1. Let *can decode* be a boolean flag, initially set to true.
2. Attempt to determine the MIME type of `audioData`, using *MIME Sniffing* § 6.2 Matching an audio or video type pattern. If the audio or video type pattern matching algorithm returns `undefined`, set *can decode* to *false*.
3. If *can decode* is *true*, attempt to decode the encoded `audioData` into linear PCM. In case of failure, set *can decode* to *false*.

PROPOSED CORRECTION ISSUE 2375. Only decode the first audio track of a multi-track media file. If the media byte-stream contains multiple audio tracks, only decode the first track to linear pcm.

NOTE: Authors who need more control over the decoding process can use [WEBCODECS].

Show Change | Show Current | Show Future

4. If *can decode* is `false`, queue a media element task to execute the following steps:

    1. Let *error* be a `DOMException` whose name is `EncodingError`.
    2. Reject *promise* with *error*, and remove it from `[[pending promises]]`.
    3. If `errorCallback` is not missing, invoke `errorCallback` with *error*.

5. Otherwise:

    1. Take the result, representing the decoded linear PCM audio data, and resample it to the sample-rate of the `BaseAudioContext` if it is different from the sample-rate of `audioData`.
    2. queue a media element task to execute the following steps:

        1. Let *buffer* be an `AudioBuffer` containing the final result (after possibly performing sample-rate conversion).
        2. Resolve *promise* with *buffer*.
        3. If `successCallback` is not missing, invoke `successCallback` with *buffer*.

✓ MDN

Arguments for the BaseAudioContext.decodeAudioData() method.

Parameter	Type	Nullable	Optional	Description
audioData	`ArrayBuffer`	✗	✗	An ArrayBuffer containing compressed audio data.
successCallback	`DecodeSuccessCallback?`	✓	✓	A callback function which will be invoked when the decoding is finished. The single argument to this callback is an AudioBuffer representing the decoded PCM audio data.

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
errorCallback	DecodeErrorCallback?	✓	✓	A callback function which will be invoked if there is an error decoding the audio file.

Return type: Promise<AudioBuffer>

§ 1.1.3. Callback DecodeSuccessCallback() Parameters

decodedData, of type AudioBuffer
The AudioBuffer containing the decoded audio data.

§ 1.1.4. Callback DecodeErrorCallback() Parameters

error, of type DOMException
The error that occurred while decoding.

§ 1.1.5. Lifetime

Once created, an AudioContext will continue to play sound until it has no more sound to play, or the page goes away.

§ 1.1.6. Lack of Introspection or Serialization Primitives

The Web Audio API takes a *fire-and-forget* approach to audio source scheduling. That is, source nodes are created for each note during the lifetime of the AudioContext, and never explicitly removed from the graph. This is incompatible with a serialization API, since there is no stable set of nodes that could be serialized.

Moreover, having an introspection API would allow content script to be able to observe garbage collections.

§ 1.1.7. System Resources Associated with BaseAudioContext Subclasses

The subclasses AudioContext and OfflineAudioContext should be considered expensive objects. Creating these objects may involve creating a high-priority thread, or using a low-latency system audio stream, both having an impact on energy consumption. It is usually not necessary to create more than one AudioContext in a document.

Constructing or resuming a BaseAudioContext subclass involves ***acquiring system resources*** for that context. For AudioContext, this also requires creation of a system audio stream. These operations return when the context begins generating output from its associated audio graph.

Additionally, a user-agent can have an implementation-defined maximum number of AudioContexts, after which any attempt to create a new AudioContext will fail, ⌛ throwing NotSupportedError.

suspend and close allow authors to ***release system resources***, including threads, processes and audio streams. Suspending a BaseAudioContext permits implementations to release some of its resources, and allows it to continue to operate later by invoking resume. Closing an AudioContext permits implementations to release all of its resources, after which it cannot be used or resumed again.

> NOTE: For example, this can involve waiting for the audio callbacks to fire regularly, or to wait for the hardware to be ready for processing.

§ 1.2. The AudioContext Interface

This interface represents an audio graph whose `AudioDestinationNode` is routed to a real-time output device that produces a signal directed at the user. In most use cases, only a single `AudioContext` is used per document.

```
enum AudioContextLatencyCategory {
        "balanced",
        "interactive",
        "playback"
};
```

`AudioContextLatencyCategory` enumeration description

Enum value	Description
"***balanced***"	Balance audio output latency and power consumption.
"***interactive***"	Provide the lowest audio output latency possible without glitching. This is the default.
"***playback***"	Prioritize sustained playback without interruption over audio output latency. Lowest power consumption.

PROPOSED ADDITION

Issue 2400 Access to a different output device

Next Change

```
enum AudioSinkType {
    "none"
};
```

`AudioSinkType` Enumeration description

Enum Value	Description
"***none***"	The audio graph will be processed without being played through an audio output device.

Show Change | Show Current | Show Future

✓ MDN

Error preparing HTML-CSS output (preProcess)

PROPOSED ADDITION

Issue 2444 Add AudioRenderCapacity Interface

Issue 2400 Access to a different output device

Next Change

```
~~[Exposed=Window]~~
~~interface AudioContext : BaseAudioContext {~~
~~    constructor (optional AudioContextOptions contextOptions = {});~~
~~    readonly attribute double baseLatency;~~
~~    readonly attribute double outputLatency;~~
~~    AudioTimestamp getOutputTimestamp ();~~
~~    Promise<undefined> resume ();~~
~~    Promise<undefined> suspend ();~~
~~    Promise<undefined> close ();~~
~~    MediaElementAudioSourceNode createMediaElementSource (HTMLMediaElement mediaElement);~~
~~    MediaStreamAudioSourceNode createMediaStreamSource (MediaStream mediaStream);~~
~~    MediaStreamTrackAudioSourceNode createMediaStreamTrackSource (~~
~~        MediaStreamTrack mediaStreamTrack);~~
~~    MediaStreamAudioDestinationNode createMediaStreamDestination ();~~
~~};~~
```

```
[Exposed=Window]
interface AudioContext : BaseAudioContext {
    constructor (optional AudioContextOptions contextOptions = {});
    readonly attribute double baseLatency;
    readonly attribute double outputLatency;
    [SecureContext] readonly attribute (DOMString or AudioSinkInfo) sinkId;
    [SecureContext] readonly attribute AudioRenderCapacity renderCapacity;
    attribute EventHandler onsinkchange;
    AudioTimestamp getOutputTimestamp ();
    Promise<undefined> resume ();
    Promise<undefined> suspend ();
    Promise<undefined> close ();
    [SecureContext] Promise<undefined> setSinkId ((DOMString or AudioSinkOptions) sinkId);
    MediaElementAudioSourceNode createMediaElementSource (HTMLMediaElement mediaElement);
    MediaStreamAudioSourceNode createMediaStreamSource (MediaStream mediaStream);
    MediaStreamTrackAudioSourceNode createMediaStreamTrackSource (
        MediaStreamTrack mediaStreamTrack);
    MediaStreamAudioDestinationNode createMediaStreamDestination ();
};
```

Show Change | Show Current | Show Future

An `AudioContext` is said to be ***allowed to start*** if the user agent allows the context state to transition from "`suspended`" to "`running`". A user agent may disallow this initial transition, and to allow it only when the `AudioContext`'s relevant global object has sticky activation.

`AudioContext` has following internal slots:

`[[suspended by user]]`

A boolean flag representing whether the context is suspended by user code. The initial value is `false`.

PROPOSED ADDITION

Issue 2400 Access to a different output device

Previous Change | Next Change

`[[sink ID]]`

A `DOMString` or an `AudioSinkInfo` representing the identifier or the information of the current audio output device respectively. The initial value is `""`, which means the default audio output device.

`[[pending resume promises]]`

An ordered list to store pending `Promises` created by `resume()`. It is initially empty.

Error preparing HTML-CSS output (preProcess)

Current | Show Future

1.2.1. Constructors

AudioContext(contextOptions)

If the current settings object's relevant global object's associated Document is NOT fully active, throw an "`InvalidStateError`" and abort these steps.

⏳ When creating an `AudioContext`, execute these steps:

✓ MDN

MDN

Error preparing HTML-CSS output (preProcess)

PROPOSED ADDITION

Issue 2456 Add a MessagePort to the AudioWorkletGlobalScope

Issue 2400 Access to a different output device

Next Change

1. ~~Set a `[[control thread state]]` to `suspended` on the `AudioContext`.~~
2. ~~Set a `[[rendering thread state]]` to `suspended` on the `AudioContext`.~~
3. ~~Let `[[pending resume promises]]` be a slot on this `AudioContext`, that is an initially empty ordered list of promises.~~
4. ~~If `contextOptions` is given, apply the options:~~
5. ~~Set the internal latency of this `AudioContext` according to `contextOptions.latencyHint`, as described in `latencyHint`.~~
6. ~~If `contextOptions.sampleRate` is specified, set the `sampleRate` of this `AudioContext` to this value. Otherwise, use the sample rate of the default output device. If the selected sample rate differs from the sample rate of the output device, this `AudioContext` MUST resample the audio output to match the sample rate of the output device.~~

   MDN

   > ~~NOTE: If resampling is required, the latency of the AudioContext may be affected, possibly by a large amount.~~

7. ~~If the context is allowed to start, send a control message to start processing.~~ MDN
8. ~~Return this `AudioContext` object.~~

1. Let *context* be a new `AudioContext` object.
2. Set a `[[control thread state]]` to `suspended` on *context*.
3. Set a `[[rendering thread state]]` to `suspended` on *context*.
4. Let *messageChannel* be a new `MessageChannel`.
5. Let *controlSidePort* be the value of *messageChannel*'s `port1` attribute.
6. Let *renderingSidePort* be the value of *messageChannel*'s `port2` attribute.
7. Let *serializedRenderingSidePort* be the result of StructuredSerializeWithTransfer(*renderingSidePort*, « *renderingSidePort* »). MDN
8. Set this `audioWorklet`'s `port` to *controlSidePort*.
9. Queue a control message to set the MessagePort on the AudioContextGlobalScope, with *serializedRenderingSidePort*.
10. If `contextOptions` is given, perform the following substeps:
    1. If `sinkId` is specified, let *sinkId* be the value of `contextOptions.sinkId` and run the following substeps:
       1. If both *sinkId* and `[[sink ID]]` are a type of `DOMString`, and they are equal to each other, abort these substeps.
       2. If *sinkId* is a type of `AudioSinkOptions` and `[[sink ID]]` is a type of `AudioSinkInfo`, and `type` in *sinkId* and `type` in `[[sink ID]]` are equal, abort these substeps.
       3. Let *validationResult* be the return value of sink identifier validation of *sinkId*.
       4. If *validationResult* is a type of `DOMException`, throw an exception with *validationResult* and abort these substeps.
       5. If *sinkId* is a type of `DOMString`, set `[[sink ID]]` to *sinkId* and abort these substeps.
       6. If *sinkId* is a type of `AudioSinkOptions`, set `[[sink ID]]` to a new instance of `AudioSinkInfo` created with the value of `type` of *sinkId*.
    2. Set the internal latency of *context* according to `contextOptions.latencyHint`, as described in `latencyHint`.
    3. If `contextOptions.sampleRate` is specified, set the `sampleRate` of *context* to this value. Otherwise, [illegible] these substeps:

Error preparing HTML-CSS output (preProcess)

1. If *sinkId* is the empty string or a type of `AudioSinkOptions`, use the sample rate of the default output device. Abort these substeps.
2. If *sinkId* is a `DOMString`, use the sample rate of the output device identified by *sinkId*. Abort these substeps.

If `contextOptions.sampleRate` differs from the sample rate of the output device, the user agent MUST resample the audio output to match the sample rate of the output device.

> NOTE: If resampling is required, the latency of *context* may be affected, possibly by a large amount.

11. If *context* is allowed to start, send a control message to start processing.
12. Return *context*.

Show Change | Show Current | Show Future

Sending a control message to start processing means executing the following steps:

PROPOSED ADDITION
Issue 2400 Access to a different output device
Previous Change | Next Change

~~1. Attempt to acquire system resources. In case of failure, abort the following steps.~~
~~2. Set the `[[rendering_thread_state]]` to `running` on the `AudioContext`.~~
~~3. queue a media element task to execute the following steps:~~
~~1. Set the `state` attribute of the `AudioContext` to "`running`".~~
~~2. queue a media element task to fire an event named `statechange` at the `AudioContext`.~~

1. Attempt to acquire system resources to use a following audio output device based on `[[sink_ID]]` for rendering:
   - The default audio output device for the empty string.
   - A audio output device identified by `[[sink_ID]]`.

   In case of failure, abort the following steps.
2. Set the `[[rendering_thread_state]]` to `running` on the `AudioContext`.
3. Queue a media element task to execute the following steps:
   1. Set the `state` attribute of the `AudioContext` to "`running`".
   2. Fire an event named `statechange` at the `AudioContext`.

Show Change | Show Current | Show Future

Note: It is unfortunately not possible to programatically notify authors that the creation of the `AudioContext` failed. User-Agents are encouraged to log an informative message if they have access to a logging mechanism, such as a developer tools console.

PROPOSED ADDITION ISSUE 2456. Add a MessagePort to the AudioWorkletGlobalScope
Previous Change | Next Change

Sending a control message to set the `MessagePort` on the `AudioWorkletGlobalScope` means executing the following steps, on the rendering thread, with *serializedRenderingSidePort*, that has been transfered to the `AudioWorkletGlobalScope`:

1. Let *deserializedPort* be the result of StructuredDeserialize(*serializedRenderingSidePort*, the current Realm).
2. Set `port` to *deserializedPort*.

Show Change | Show Current | Show Future

Error preparing HTML-CSS output (preProcess)

Arguments for the AudioContext.constructor(contextOptions) method.

Parameter	Type	Nullable	Optional	Description
contextOptions	AudioContextOptions	✗	✓	User-specified options controlling how the AudioContext should be constructed.

✓ MDN

§ 1.2.2. Attributes

***baseLatency*, of type double, readonly**

This represents the number of seconds of processing latency incurred by the AudioContext passing the audio from the AudioDestinationNode to the audio subsystem. It does not include any additional latency that might be caused by any other processing between the output of the AudioDestinationNode and the audio hardware and specifically does not include any latency incurred the audio graph itself.

For example, if the audio context is running at 44.1 kHz and the AudioDestinationNode implements double buffering internally and can process and output audio each render quantum, then the processing latency is $(2 \cdot 128)/44100 = 5.805$ms, approximately.

***outputLatency*, of type double, readonly**

The estimation in seconds of audio output latency, i.e., the interval between the time the UA requests the host system to play a buffer and the time at which the first sample in the buffer is actually processed by the audio output device. For devices such as speakers or headphones that produce an acoustic signal, this latter time refers to the time when a sample's sound is produced.

An outputLatency attribute value depends on the platform and the connected audio output device hardware. The outputLatency attribute value may change while the context is running or the associated audio output device changes. It is useful to query this value frequently when accurate synchronization is required.

PROPOSED ADDITION ISSUE 2444. Add AudioRenderCapacity Interface

Previous Change | Next Change

***renderCapacity*, of type AudioRenderCapacity, readonly**

Returns an AudioRenderCapacity instance associated with an AudioContext.

Show Change | Show Current | Show Future

PROPOSED ADDITION

Issue 2400 Access to a different output device

Previous Change | Next Change

***sinkId*, of type (DOMString or AudioSinkInfo), readonly**

Returns the value of [[sink ID]] internal slot. This attribute is cached upon update, and it returns the same object after caching.

***onsinkchange*, of type EventHandler**

An event handler for setSinkId(). The event type of this event handler is ***sinkchange***. This event will be dispatched when changing the output device is completed.

> NOTE: This is not dispatched for the initial device selection in the construction of AudioContext. The statechange event is available to check the readiness of the initial output device.

Show Change | Show Current | Show Future

✓ MDN

§ 1.2.3. Methods

close()

Closes the AudioContext, releasing the system resources being used. This will not automatically release all AudioContext-created objects, but will suspend the progression of the AudioContext's currentTime, and stop a.

Error preparing HTML-CSS output (preProcess)

⌛ When close is called, execute these steps:

1. If this's relevant global object's associated Document is not fully active then return a promise rejected with "`InvalidStateError`" `DOMException`.
2. Let *promise* be a new Promise.
3. If the `[[control thread state]]` flag on the `AudioContext` is `closed` reject the promise with `InvalidStateError`, abort these steps, returning *promise*.
4. Set the `[[control thread state]]` flag on the `AudioContext` to `closed`.
5. Queue a control message to close the `AudioContext`.
6. Return *promise*.

Running a control message to close an `AudioContext` means running these steps on the rendering thread:

1. Attempt to release system resources.
2. Set the `[[rendering thread state]]` to `suspended`.

   > This will stop rendering.

3. If this control message is being run in a reaction to the document being unloaded, abort this algorithm.

   > There is no need to notify the control thread in this case.

4. queue a media element task to execute the following steps:

   > 1. Resolve *promise*.
   > 2. If the `state` attribute of the `AudioContext` is not already "`closed`":
   >
   >   1. Set the `state` attribute of the `AudioContext` to "`closed`".
   >   2. queue a media element task to fire an event named `statechange` at the `AudioContext`.

When an `AudioContext` is closed, any `MediaStream`s and `HTMLMediaElement`s that were connected to an `AudioContext` will have their output ignored. That is, these will no longer cause any output to speakers or other output devices. For more flexibility in behavior, consider using `HTMLMediaElement.captureStream()`.

> NOTE: When an `AudioContext` has been closed, implementation can choose to aggressively release more resources than when suspending.

No parameters.
Return type: `Promise`<`undefined`>

createMediaElementSource(mediaElement)

Creates a `MediaElementAudioSourceNode` given an `HTMLMediaElement`. As a consequence of calling this method, audio playback from the `HTMLMediaElement` will be re-routed into the processing graph of the `AudioContext`.

Arguments for the AudioContext.createMediaElementSource() method.

Parameter	Type	Nullable	Optional	Description
mediaElement	`HTMLMediaElement`	✗	✗	The media element that will be re-routed.

Return type: `MediaElementAudioSourceNode`

createMediaStreamDestination()

Creates a `MediaStreamAudioDestinationNode`

No parameters.
Return type: `MediaStreamAudioDestinationNode`

createMediaStreamSource(mediaStream)

Error preparing HTML-CSS output (preProcess)

`eamAudioSourceNode`.

Arguments for the AudioContext.createMediaStreamSource() method.

Parameter	Type	Nullable	Optional	Description
mediaStream	MediaStream	✗	✗	The media stream that will act as source.

Return type: MediaStreamAudioSourceNode

createMediaStreamTrackSource(mediaStreamTrack)

Creates a MediaStreamTrackAudioSourceNode.

Arguments for the AudioContext.createMediaStreamTrackSource() method.

Parameter	Type	Nullable	Optional	Description
mediaStreamTrack	MediaStreamTrack	✗	✗	The MediaStreamTrack that will act as source. ⌛ The value of its `kind` attribute must be equal to `"audio"`, or an InvalidStateError exception MUST be thrown.

Return type: MediaStreamTrackAudioSourceNode

getOutputTimestamp()

Returns a new AudioTimestamp instance containing two related audio stream position values for the context: the contextTime member contains the time of the sample frame which is currently being rendered by the audio output device (i.e., output audio stream position), in the same units and origin as context's currentTime; the performanceTime member contains the time estimating the moment when the sample frame corresponding to the stored `contextTime` value was rendered by the audio output device, in the same units and origin as `performance.now()` (described in [hr-time-3]).

If the context's rendering graph has not yet processed a block of audio, then getOutputTimestamp call returns an AudioTimestamp instance with both members containing zero.

After the context's rendering graph has started processing of blocks of audio, its currentTime attribute value always exceeds the contextTime value obtained from getOutputTimestamp method call.

EXAMPLE 4 ¶

The value returned from getOutputTimestamp method can be used to get performance time estimation for the slightly later context's time value:

```
function outputPerformanceTime(contextTime) {
    const timestamp = context.getOutputTimestamp();
    const elapsedTime = contextTime - timestamp.contextTime;
    return timestamp.performanceTime + elapsedTime * 1000;
}
```

In the above example the accuracy of the estimation depends on how close the argument value is to the current output audio stream position: the closer the given `contextTime` is to `timestamp.contextTime`, the better the accuracy of the obtained estimation.

NOTE: The difference between the values of the context's currentTime and the contextTime obtained from getOutputTimestamp method call cannot be considered as a reliable output latency estimation because currentTime may be incremented at non-uniform time intervals, so outputLatency attribute should be used instead.

No parameters.

Return type: AudioTimestamp

Error preparing HTML-CSS output (preProcess)

Resumes the progression of the AudioContext's currentTime when it has been suspended.

⌛ When resume is called, execute these steps:

1. If this's relevant global object's associated Document is not fully active then return a promise rejected with "`InvalidStateError`" `DOMException`.
2. Let *promise* be a new Promise.
3. If the `[[control thread state]]` on the `AudioContext` is `closed` reject the promise with `InvalidStateError`, abort these steps, returning *promise*.
4. Set `[[suspended by user]]` to `false`.
5. If the context is not allowed to start, append *promise* to `[[pending promises]]` and `[[pending resume promises]]` and abort these steps, returning *promise*.
6. Set the `[[control thread state]]` on the `AudioContext` to `running`.
7. Queue a control message to resume the `AudioContext`.
8. Return *promise*.

Running a control message to resume an `AudioContext` means running these steps on the rendering thread:

1. Attempt to acquire system resources.
2. Set the `[[rendering thread state]]` on the `AudioContext` to `running`.
3. Start rendering the audio graph.
4. In case of failure, queue a media element task to execute the following steps:

   1. Reject all promises from `[[pending resume promises]]` in order, then clear `[[pending resume promises]]`.
   2. Additionally, remove those promises from `[[pending promises]]`.

5. queue a media element task to execute the following steps:

   1. Resolve all promises from `[[pending resume promises]]` in order.
   2. Clear `[[pending resume promises]]`. Additionally, remove those promises from `[[pending promises]]`.
   3. Resolve *promise*.
   4. If the `state` attribute of the `AudioContext` is not already "`running`":

      1. Set the `state` attribute of the `AudioContext` to "`running`".
      2. queue a media element task to fire an event named `statechange` at the `AudioContext`.

No parameters.
Return type: `Promise`<`undefined`>

suspend()

Suspends the progression of `AudioContext`'s `currentTime`, allows any current context processing blocks that are already processed to be played to the destination, and then allows the system to release its claim on audio hardware. This is generally useful when the application knows it will not need the `AudioContext` for some time, and wishes to temporarily release system resource associated with the `AudioContext`. The promise resolves when the frame buffer is empty (has been handed off to the hardware), or immediately (with no other effect) if the context is already `suspended`. The promise is rejected if the context has been closed.

⌛ When suspend is called, execute these steps:

1. If this's relevant global object's associated Document is not fully active then return a promise rejected with "`InvalidStateError`" `DOMException`.
2. Let *promise* be a new Promise.
3. If the `[[control thread state]]` on the `AudioContext` is `closed` reject the promise with `InvalidStateError`, abort these steps, returning *promise*.
4. Append *promise* to `[[pending promises]]`.
5. Set `[[suspended by user]]` to `true`.

Error preparing HTML-CSS output (preProcess)

`rol thread state]]` on the `AudioContext` to `suspended`.

7. Queue a control message to suspend the `AudioContext`.
8. Return *promise*.

Running a control message to suspend an `AudioContext` means running these steps on the rendering thread:

1. Attempt to release system resources.
2. Set the `[[rendering thread state]]` on the `AudioContext` to `suspended`.
3. queue a media element task to execute the following steps:

    1. Resolve *promise*.
    2. If the `state` attribute of the `AudioContext` is not already "`suspended`":

        1. Set the `state` attribute of the `AudioContext` to "`suspended`".
        2. queue a media element task to fire an event named `statechange` at the `AudioContext`.

While an `AudioContext` is suspended, `MediaStream`s will have their output ignored; that is, data will be lost by the real time nature of media streams. `HTMLMediaElement`s will similarly have their output ignored until the system is resumed. `AudioWorkletNode`s and `ScriptProcessorNode`s will cease to have their processing handlers invoked while suspended, but will resume when the context is resumed. For the purpose of `AnalyserNode` window functions, the data is considered as a continuous stream - i.e. the `resume()`/`suspend()` does not cause silence to appear in the `AnalyserNode`'s stream of data. In particular, calling `AnalyserNode` functions repeatedly when a `AudioContext` is suspended MUST return the same data.

No parameters.
Return type: `Promise`<`undefined`>

PROPOSED ADDITION

Issue 2400 Access to a different output device

Previous Change | Next Change

setSinkId((DOMString or AudioSinkOptions) sinkId)

Sets the identifier of an output device. When this method is invoked, the user agent MUST run the following steps:

1. Let *sinkId* be the method's first argument.
2. If *sinkId* is equal to `[[sink ID]]`, return a promise, resolve it immediately and abort these steps.
3. Let *validationResult* be the return value of sink identifier validation of *sinkId*.
4. If *validationResult* is not `null`, return a promise rejected with *validationResult*. Abort these steps.
5. Let *p* be a new promise.
6. Send a control message with *p* and *sinkId* to start processing.
7. Return *p*.

Sending a control message to start processing during `setSinkId()` means executing the following steps:

1. Let *p* be the promise passed into this algorithm.
2. Let *sinkId* be the sink identifier passed into this algorithm.
3. If both *sinkId* and `[[sink ID]]` are a type of `DOMString`, and they are equal to each other, queue a media element task to resolve *p* and abort these steps.
4. If *sinkId* is a type of `AudioSinkOptions` and `[[sink ID]]` is a type of `AudioSinkInfo`, and `type` in *sinkId* and `type` in `[[sink ID]]` are equal, queue a media element task to resolve *p* and abort these steps.
5. Let *wasRunning* be true.
6. Set *wasRunning* to false if the `[[rendering thread state]]` on the `AudioContext` is `"suspended"`.
7. Pause the renderer after processing the current render quantum.
8. Attempt to release system resources.
9. If *wasRunning* is true:
    1. Set the `[[rendering thread state]]` on the `AudioContext` to `"suspended"`.
    2. Queue a media element task to execute the following steps:
        1. If the `state` attribute of the `AudioContext` is not already `"suspended"`:
            1. Set the `state` attribute of the `AudioContext` to `"suspended"`.
            2. Fire an event named `statechange` at the associated `AudioContext`.
10. Attempt to acquire system resources to use a following audio output device based on `[[sink ID]]` for rendering:
    - The default audio output device for the empty string,
    - A audio output device identified by `[[sink ID]]`.

    In case of failure, reject *p* with `"InvalidAccessError"` abort the following steps.
11. Queue a media element task to execute the following steps:
    1. If *sinkId* is a type of `DOMString`, set `[[sink ID]]` to *sinkId*. Abort these steps.
    2. If *sinkId* is a type of `AudioSinkOptions` and `[[sink ID]]` is a type of `DOMString`, set `[[sink ID]]` to a new instance of `AudioSinkInfo` created with the value of `type` of *sinkId*.
    3. If *sinkId* is a type of `AudioSinkOptions` and `[[sink ID]]` is a type of `AudioSinkInfo`, set `type` of `[[sink ID]]` to the `type` value of *sinkId*.
    4. Resolve *p*.
    5. Fire an event named `sinkchange` at the associated `AudioContext`.
12. If *wasRunning* is true:
    1. Set the `[[rendering thread state]]` on the `AudioContext` to `"running"`.

MDN

Error preparing HTML-CSS output (preProcess)

2. Queue a media element task to execute the following steps:
    1. If the `state` attribute of the `AudioContext` is not already `"running"`:
        1. Set the `state` attribute of the `AudioContext` to `"running"`.
        2. Fire an event named `statechange` at the associated `AudioContext`.

Show Change | Show Current | Show Future

PROPOSED ADDITION

Issue 2400 Access to a different output device

Previous Change | Next Change

1.2.4. Validating `sinkId`

This algorithm is used to validate the information provided to modify `sinkId`:

1. Let *document* be the current settings object's associated Document.
2. Let *sinkIdArg* be the value passed in to this algorithm.
3. If *document* is not allowed to use the feature identified by `"speaker-selection"`, return a new `DOMException` whose name is `"NotAllowedError"`.
4. If *sinkIdArg* is a type of `DOMString` but it is not equal to the empty string or it does not match any audio output device identified by the result that would be provided by `enumerateDevices()`, return a new `DOMException` whose name is `"NotFoundError"`.
5. Return `null`.

Show Change | Show Current | Show Future

1.2.5. `AudioContextOptions`

The `AudioContextOptions` dictionary is used to specify user-specified options for an `AudioContext`.

PROPOSED ADDITION

Issue 2400 Access to a different output device

Previous Change | Next Change

```
~~dictionary AudioContextOptions {~~
~~    (AudioContextLatencyCategory or double) latencyHint = "interactive";~~
~~    float sampleRate;~~
~~};~~
```

```
dictionary AudioContextOptions {
    (AudioContextLatencyCategory or double) latencyHint = "interactive";
    float sampleRate;
    (DOMString or AudioSinkOptions) sinkId;
};
```

Show Change | Show Current | Show Future

1.2.5.1. Dictionary `AudioContextOptions` Members

***`latencyHint`*, of type (`AudioContextLatencyCategory or double`), defaulting to `"interactive"`**

Identify the type of playback, which affects tradeoffs between audio output latency and power consumption.

The preferred value of the `latencyHint` is a value from `AudioContextLatencyCategory`. However, a double can also be specified for the number of seconds of latency for finer control to balance latency and power consumption. It is

Error preparing HTML-CSS output (preProcess)

at the browser's discretion to interpret the number appropriately. The actual latency used is given by AudioContext's `baseLatency` attribute.

***`sampleRate`*, of type float**

Set the `sampleRate` to this value for the `AudioContext` that will be created. The supported values are the same as the sample rates for an `AudioBuffer`. ⌛ A `NotSupportedError` exception MUST be thrown if the specified sample rate is not supported.

If `sampleRate` is not specified, the preferred sample rate of the output device for this `AudioContext` is used.

✓ MDN

PROPOSED ADDITION

Issue 2400 Access to a different output device

Previous Change | Next Change

***`sinkId`*, of type (`DOMString` or `AudioSinkOptions`)**

The identifier or associated information of the audio output device. See `sinkId` for more details.

Show Change | Show Current | Show Future

PROPOSED ADDITION

Issue 2400 Access to a different output device

Previous Change

1.2.6. `AudioSinkOptions`

The `AudioSinkOptions` dictionary is used to specify options for `sinkId`.

✓ MDN

```
dictionary AudioSinkOptions {
    required AudioSinkType type;
};
```

1.2.6.1. Dictionary `AudioSinkOptions` Members

***`type`*, of type AudioSinkType**

A value of `AudioSinkType` to specify the type of the device.

1.2.7. `AudioSinkInfo`

The `AudioSinkInfo` interface is used to get information on the current audio output device via `sinkId`.

```
[Exposed=Window]
interface AudioSinkInfo {
    readonly attribute AudioSinkType type;
};
```

1.2.7.1. Attributes

***`type`*, of type AudioSinkType, readonly**

A value of `AudioSinkType` that represents the type of the device.

Show Change | Show Current | Show Future

Error preparing HTML-CSS output (preProcess)

1.2.8. `AudioTimestamp`

```
dictionary AudioTimestamp {
    double contextTime;
    DOMHighResTimeStamp performanceTime;
};
```

1.2.8.1. Dictionary `AudioTimestamp` Members

***`contextTime`*, of type double**
: Represents a point in the time coordinate system of BaseAudioContext's `currentTime`.

***`performanceTime`*, of type DOMHighResTimeStamp**
: Represents a point in the time coordinate system of a `Performance` interface implementation (described in [hr-time-3]).

✓ MDN

✓ MDN

✓ MDN

Error preparing HTML-CSS output (preProcess)

PROPOSED ADDITION ISSUE 2444. Add AudioRenderCapacity Interface

Previous Change

1.2.9. AudioRenderCapacity

```
[Exposed=Window]
interface AudioRenderCapacity : EventTarget {
    undefined start(optional AudioRenderCapacityOptions options = {});
    undefined stop();
    attribute EventHandler onupdate;
};
```

This interface provides rendering performance metrics of an `AudioContext`. In order to calculate them, the renderer collects a load value per system-level audio callback.

1.2.9.1. Attributes

***onupdate*, of type EventHandler**

The event type of this event handler is ***update***. Events dispatched to the event handler will use the `AudioRenderCapacityEvent` interface.

1.2.9.2. Methods

start(options)

Starts metric collection and analysis. This will repeatedly fire an event named `update` at `AudioRenderCapacity`, using `AudioRenderCapacityEvent`, with the given update interval in `AudioRenderCapacityOptions`.

stop()

Stops metric collection and analysis. It also stops dispatching `update` events.

1.2.10. AudioRenderCapacityOptions

The `AudioRenderCapacityOptions` dictionary can be used to provide user options for an `AudioRenderCapacity`.

✓ MDN

```
dictionary AudioRenderCapacityOptions {
    double updateInterval = 1;
};
```

1.2.10.1. Dictionary `AudioRenderCapacityOptions` Members

***updateInterval*, of type double, defaulting to `1`**

An update interval (in second) for dispaching `AudioRenderCapacityEvent`s. A load value is calculated per system-level audio callback, and multiple load values will be collected over the specified interval period. For example, if the renderer runs at a 48Khz sample rate and the system-level audio callback's buffer size is 192 frames, 250 load values will be collected over 1 second interval.

If the given value is smaller than the duration of the system-level audio callback, `NotSupportedError` is thrown.

1.2.11. AudioRenderCapacityEvent

Error preparing HTML-CSS output (preProcess)

```
[Exposed=Window]
interface AudioRenderCapacityEvent : Event {
    constructor (DOMString type, optional AudioRenderCapacityEventInit eventInitDict = {});
        readonly attribute double timestamp;
        readonly attribute double averageLoad;
        readonly attribute double peakLoad;
        readonly attribute double underrunRatio;
};

dictionary AudioRenderCapacityEventInit : EventInit {
    double timestamp = 0;
    double averageLoad = 0;
    double peakLoad = 0;
    double underrunRatio = 0;
};
```

MDN

1.2.11.1. Attributes

`timestamp`, of type double, readonly
: The start time of the data collection period in terms of the associated `AudioContext`'s `currentTime`.

`averageLoad`, of type double, readonly
: An average of collected load values over the given update interval. The precision is limited to 1/100th.

`peakLoad`, of type double, readonly
: A maximum value from collected load values over the given update interval. The precision is also limited to 1/100th.

`underrunRatio`, of type double, readonly
: A ratio between the number of buffer underruns (when a load value is greater than 1.0) and the total number of system-level audio callbacks over the given update interval.

  Where u is the number of buffer underruns and N is the number of system-level audio callbacks over the given update interval, the buffer underrun ratio is:

  - 0.0 if $u = 0$.
  - Otherwise, compute u / N and take a ceiling value of the nearest 100th.

Show Change | Show Current | Show Future

§ 1.3. The `OfflineAudioContext` Interface

`OfflineAudioContext` is a particular type of `BaseAudioContext` for rendering/mixing-down (potentially) faster than real-time. It does not render to the audio hardware, but instead renders as quickly as possible, fulfilling the returned promise with the rendered result as an `AudioBuffer`.

```
[Exposed=Window]
interface OfflineAudioContext : BaseAudioContext {
    constructor(OfflineAudioContextOptions contextOptions);
    constructor(unsigned long numberOfChannels, unsigned long length, float sampleRate);
    Promise<AudioBuffer> startRendering();
    Promise<undefined> resume();
    Promise<undefined> suspend(double suspendTime);
    readonly attribute unsigned long length;
    attribute EventHandler oncomplete;
};
```

Error preparing HTML-CSS output (preProcess)

§ 1.3.1. Constructors

OfflineAudioContext(contextOptions)

PROPOSED ADDITION ISSUE 2456. Add a MessagePort to the AudioWorkletGlobalScope

Previous Change | Next Change

~~If the current settings object's relevant global object's associated Document is NOT fully active, throw an `InvalidStateError` and abort these steps.~~

~~Let *c* be a new `OfflineAudioContext` object. Initialize *c* as follows:~~

1. ~~Set the `[[control_thread_state]]` for *c* to `"suspended"`.~~
2. ~~Set the `[[rendering_thread_state]]` for *c* to `"suspended"`.~~
3. ~~Construct an `AudioDestinationNode` with its `channelCount` set to `contextOptions.numberOfChannels`.~~

If the current settings object's relevant global object's associated Document is NOT fully active, throw an `InvalidStateError` and abort these steps.

Let *c* be a new `OfflineAudioContext` object. Initialize *c* as follows:

1. Set the `[[control_thread_state]]` for *c* to `"suspended"`.
2. Set the `[[rendering_thread_state]]` for *c* to `"suspended"`.
3. Construct an `AudioDestinationNode` with its `channelCount` set to `contextOptions.numberOfChannels`.
4. Let *messageChannel* be a new `MessageChannel`.
5. Let *controlSidePort* be the value of *messageChannel*'s `port1` attribute.
6. Let *renderingSidePort* be the value of *messageChannel*'s `port2` attribute.
7. Let *serializedRenderingSidePort* be the result of StructuredSerializeWithTransfer(*renderingSidePort*, « *renderingSidePort* »).
8. Set this `audioWorklet`'s `port` to *controlSidePort*.
9. Queue a control message to set the MessagePort on the AudioContextGlobalScope, with *serializedRenderingSidePort*.

Show Change | Show Current | Show Future

✓ MDN ✓ MDN ✓ MDN ✓ MDN

Arguments for the OfflineAudioContext.constructor(contextOptions) method.

Parameter	Type	Nullable	Optional	Description
contextOptions				The initial parameters needed to construct this context.

OfflineAudioContext(numberOfChannels, length, sampleRate)

The `OfflineAudioContext` can be constructed with the same arguments as AudioContext.createBuffer. ⌛ A `NotSupportedError` exception MUST be thrown if any of the arguments is negative, zero, or outside its nominal range.

The OfflineAudioContext is constructed as if

```
new OfflineAudioContext({
        numberOfChannels: numberOfChannels,
        length: length,
        sampleRate: sampleRate
})
```

Error preparing HTML-CSS output (preProcess)

Arguments for the OfflineAudioContext.constructor(numberOfChannels, length, sampleRate) method.

Parameter	Type	Nullable	Optional	Description
numberOfChannels	unsigned long	✗	✗	Determines how many channels the buffer will have. See `createBuffer()` for the supported number of channels.
length	unsigned long	✗	✗	Determines the size of the buffer in sample-frames.
sampleRate	float	✗	✗	Describes the sample-rate of the linear PCM audio data in the buffer in sample-frames per second. See `createBuffer()` for valid sample rates.

§ 1.3.2. Attributes

✓ MDN

***`length`*, of type unsigned long, readonly**
The size of the buffer in sample-frames. This is the same as the value of the `length` parameter for the constructor.

***`oncomplete`*, of type EventHandler**
The event type of this event handler is ***`complete`***. The event dispatched to the event handler will use the `OfflineAudioCompletionEvent` interface. It is the last event fired on an `OfflineAudioContext`.

§ 1.3.3. Methods

`startRendering()`
Given the current connections and scheduled changes, starts rendering audio.

Although the primary method of getting the rendered audio data is via its promise return value, the instance will also fire an event named `complete` for legacy reasons.

Let ***`[[rendering started]]`*** be an internal slot of this `OfflineAudioContext`. Initialize this slot to *false*.

⏳ When `startRendering` is called, the following steps MUST be performed on the control thread:

1. If this's relevant global object's associated Document is not fully active then return a promise rejected with "`InvalidStateError`" `DOMException`.
2. If the `[[rendering started]]` slot on the `OfflineAudioContext` is *true*, return a rejected promise with `InvalidStateError`, and abort these steps.
3. Set the `[[rendering started]]` slot of the `OfflineAudioContext` to *true*.
4. Let *promise* be a new promise.
5. Create a new `AudioBuffer`, with a number of channels, length and sample rate equal respectively to the `numberOfChannels`, `length` and `sampleRate` values passed to this instance's constructor in the `contextOptions` parameter. Assign this buffer to an internal slot ***`[[rendered buffer]]`*** in the `OfflineAudioContext`.
6. If an exception was thrown during the preceding `AudioBuffer` constructor call, reject *promise* with this exception.
7. Otherwise, in the case that the buffer was successfully constructed, begin offline rendering.
8. Append *promise* to `[[pending promises]]`.
9. Return *promise*.

✓ MDN

✓ MDN

✓ MDN

✓ MDN

To ***begin offline rendering***, the following steps MUST happen on a rendering thread that is created for the occasion.

1. Given the current connections and scheduled changes, start rendering `length` sample-frames of audio into `[[rendered buffer]]`
2. For every render quantum, check and `suspend` rendering if necessary.

Error preparing HTML-CSS output (preProcess)

3. If a suspended context is resumed, continue to render the buffer.
4. Once the rendering is complete, queue a media element task to execute the following steps:
    1. Resolve the *promise* created by `startRendering()` with `[[rendered buffer]]`.
    2. queue a media element task to fire an event named `complete` using an instance of `OfflineAudioCompletionEvent` whose `renderedBuffer` property is set to `[[rendered buffer]]`.

No parameters.
Return type: `Promise`<`AudioBuffer`>

`resume()`

Resumes the progression of the `OfflineAudioContext`'s `currentTime` when it has been suspended.

⏳ When resume is called, execute these steps:

1. If this's relevant global object's associated Document is not fully active then return a promise rejected with "`InvalidStateError`" `DOMException`.
2. Let *promise* be a new Promise.
3. Abort these steps and reject *promise* with `InvalidStateError` when any of following conditions is true:
    - The `[[control thread state]]` on the `OfflineAudioContext` is `closed`.
    - The `[[rendering started]]` slot on the `OfflineAudioContext` is *false*.
4. Set the `[[control thread state]]` flag on the `OfflineAudioContext` to `running`.
5. Queue a control message to resume the `OfflineAudioContext`.
6. Return *promise*.

Running a control message to resume an `OfflineAudioContext` means running these steps on the rendering thread:

1. Set the `[[rendering thread state]]` on the `OfflineAudioContext` to `running`.
2. Start rendering the audio graph.
3. In case of failure, queue a media element task to reject *promise* and abort the remaining steps.
4. queue a media element task to execute the following steps:
    1. Resolve *promise*.
    2. If the `state` attribute of the `OfflineAudioContext` is not already "`running`":
        1. Set the `state` attribute of the `OfflineAudioContext` to "`running`".
        2. queue a media element task to fire an event named `statechange` at the `OfflineAudioContext`.

No parameters.
Return type: `Promise`<`undefined`>

`suspend(suspendTime)`

Schedules a suspension of the time progression in the audio context at the specified time and returns a promise. This is generally useful when manipulating the audio graph synchronously on `OfflineAudioContext`.

Note that the maximum precision of suspension is the size of the render quantum and the specified suspension time will be rounded up to the nearest render quantum boundary. For this reason, it is not allowed to schedule multiple suspends at the same quantized frame. Also, scheduling should be done while the context is not running to ensure precise suspension.

Arguments for the OfflineAudioContext.suspend() method.

Parameter	Type	Nullable	Optional	Description
`suspendTime`	`double`	✗	✗	Schedules a suspension of the rendering at the specified time, which is quantized and rounded up to the render quantum size. If the quantized frame number 1. is negative or

Parameter	Type	Nullable	Optional	Description
				2. is less than or equal to the current time or 3. is greater than or equal to the total render duration or 4. is scheduled by another suspend for the same time, then the promise is rejected with `InvalidStateError`.

Return type: `Promise`<`undefined`>

1.3.4. `OfflineAudioContextOptions`

This specifies the options to use in constructing an `OfflineAudioContext`.

```
dictionary OfflineAudioContextOptions {
    unsigned long numberOfChannels = 1;
    required unsigned long length;
    required float sampleRate;
};
```

1.3.4.1. Dictionary `OfflineAudioContextOptions` Members

***`length`*, of type unsigned long**
: The length of the rendered `AudioBuffer` in sample-frames.

***`numberOfChannels`*, of type unsigned long, defaulting to `1`**
: The number of channels for this `OfflineAudioContext`.

***`sampleRate`*, of type float**
: The sample rate for this `OfflineAudioContext`.

1.3.5. The `OfflineAudioCompletionEvent` Interface

This is an `Event` object which is dispatched to `OfflineAudioContext` for legacy reasons.

```
[Exposed=Window]
interface OfflineAudioCompletionEvent : Event {
    constructor (DOMString type, OfflineAudioCompletionEventInit eventInitDict);
    readonly attribute AudioBuffer renderedBuffer;
};
```

1.3.5.1. Attributes

***`renderedBuffer`*, of type AudioBuffer, readonly**
: An `AudioBuffer` containing the rendered audio data.

1.3.5.2. `OfflineAudioCompletionEventInit`

```
dictionary OfflineAudioCompletionEventInit : EventInit {
    required AudioBuffer renderedBuffer;
};
```

Error preparing HTML-CSS output (preProcess)

§ 1.3.5.2.1. Dictionary `OfflineAudioCompletionEventInit` Members

***renderedBuffer*, of type AudioBuffer**
: Value to be assigned to the `renderedBuffer` attribute of the event.

§ 1.4. The `AudioBuffer` Interface

This interface represents a memory-resident audio asset. It can contain one or more channels with each channel appearing to be 32-bit floating-point linear PCM values with a nominal range of [− 1, 1] but the values are not limited to this range. Typically, it would be expected that the length of the PCM data would be fairly short (usually somewhat less than a minute). For longer sounds, such as music soundtracks, streaming should be used with the `<audio>` element and `MediaElementAudioSourceNode`.

An `AudioBuffer` may be used by one or more `AudioContext`s, and can be shared between an `OfflineAudioContext` and an `AudioContext`.

`AudioBuffer` has four internal slots:

[[number of channels]]
: The number of audio channels for this `AudioBuffer`, which is an unsigned long.

[[length]]
: The length of each channel of this `AudioBuffer`, which is an unsigned long.

[[sample rate]]
: The sample-rate, in Hz, of this `AudioBuffer`, a float.

[[internal data]]
: A data block holding the audio sample data.

```
[Exposed=Window]
interface AudioBuffer {
    constructor (AudioBufferOptions options);
    readonly attribute float sampleRate;
    readonly attribute unsigned long length;
    readonly attribute double duration;
    readonly attribute unsigned long numberOfChannels;
    Float32Array getChannelData (unsigned long channel);
    undefined copyFromChannel (Float32Array destination,
                               unsigned long channelNumber,
                               optional unsigned long bufferOffset = 0);
    undefined copyToChannel (Float32Array source,
                             unsigned long channelNumber,
                             optional unsigned long bufferOffset = 0);
};
```

MDN

§ 1.4.1. Constructors

AudioBuffer(options)

1. ⌛ If any of the values in `options` lie outside its nominal range, throw a `NotSupportedError` exception and abort the following steps.
2. Let *b* be a new `AudioBuffer` object.
3. Respectively assign the values of the attributes `numberOfChannels`, `length`, `sampleRate` of the `AudioBufferOptions` passed in the constructor to the internal slots `[[number of channels]]`, `[[length]]`, `[[sample rate]]`.
4. Set the internal slot `[[internal data]]` of this `AudioBuffer` to the result of calling `CreateByteDataBlock([[length]] * [[number of channels]])`.

   NOTE: This initializes the underlying storage to zero.

Error preparing HTML-CSS output (preProcess)

Arguments for the AudioBuffer.constructor() method.

Parameter	Type	Nullable	Optional	Description
options	AudioBufferOptions	✗	✗	An AudioBufferOptions that determine the properties for this AudioBuffer.

§ 1.4.2. Attributes

***duration*, of type double, readonly**
Duration of the PCM audio data in seconds.

This is computed from the [[sample rate]] and the [[length]] of the AudioBuffer by performing a division between the [[length]] and the [[sample rate]].

***length*, of type unsigned long, readonly**
Length of the PCM audio data in sample-frames. This MUST return the value of [[length]].

***numberOfChannels*, of type unsigned long, readonly**
The number of discrete audio channels. This MUST return the value of [[number of channels]].

***sampleRate*, of type float, readonly**
The sample-rate for the PCM audio data in samples per second. This MUST return the value of [[sample rate]].

§ 1.4.3. Methods

copyFromChannel(destination, channelNumber, bufferOffset)
The copyFromChannel() method copies the samples from the specified channel of the AudioBuffer to the `destination` array.

Let `buffer` be the AudioBuffer with N_b frames, let N_f be the number of elements in the destination array, and k be the value of bufferOffset. Then the number of frames copied from `buffer` to destination is $\max(0, \min(N_b - k, N_f))$. If this is less than N_f, then the remaining elements of destination are not modified.

Arguments for the AudioBuffer.copyFromChannel() method.

Parameter	Type	Nullable	Optional	Description
destination	Float32Array	✗	✗	The array the channel data will be copied to.
channelNumber	unsigned long	✗	✗	The index of the channel to copy the data from. If `channelNumber` is greater or equal than the number of channels of the AudioBuffer, ⌛ an IndexSizeError MUST be thrown.
bufferOffset	unsigned long	✗	✓	An optional offset, defaulting to 0. Data from the AudioBuffer starting at this offset is copied to the destination.

Return type: undefined

copyToChannel(source, channelNumber, bufferOffset)
The copyToChannel() method copies the samples to the specified channel of the AudioBuffer from the `source` array.

A UnknownError may be thrown if source cannot be copied to the buffer.

Let `buffer` be the AudioBuffer with N_b frames, let N_f be the number of elements in the source array, and k be the

set. Then the number of frames copied from source to the `buffer` is $\max(0, \min(N_b - k, N_f))$

. If this is less than N_f, then the remaining elements of `buffer` are not modified.

Arguments for the AudioBuffer.copyToChannel() method.

Parameter	Type	Nullable	Optional	Description
source	`Float32Array`	✗	✗	The array the channel data will be copied from.
channelNumber	`unsigned long`	✗	✗	The index of the channel to copy the data to. If `channelNumber` is greater or equal than the number of channels of the `AudioBuffer`, ⌛ an `IndexSizeError` MUST be thrown.
bufferOffset	`unsigned long`	✗	✓	An optional offset, defaulting to 0. Data from the `source` is copied to the `AudioBuffer` starting at this offset.

Return type: `undefined`

getChannelData(channel)

PROPOSED CORRECTION ISSUE 2361-3. Use new Web IDL buffer primitives

Previous Change | Next Change

According to the rules described in acquire the content either ~~get a reference to~~ allow writing into or getting a copy of the bytes stored in `[[internal data]]` in a new `Float32Array`

Show Change | Show Current | Show Future

A `UnknownError` may be thrown if the `[[internal data]]` or the new `Float32Array` cannot be created.

Arguments for the AudioBuffer.getChannelData() method.

Parameter	Type	Nullable	Optional	Description
channel	`unsigned long`	✗	✗	This parameter is an index representing the particular channel to get data for. An index value of 0 represents the first channel. ⌛ This index value MUST be less than `[[number of channels]]` or an `IndexSizeError` exception MUST be thrown.

Return type: `Float32Array`

NOTE: The methods `copyToChannel()` and `copyFromChannel()` can be used to fill part of an array by passing in a `Float32Array` that's a view onto the larger array. When reading data from an `AudioBuffer`'s channels, and the data can be processed in chunks, `copyFromChannel()` should be preferred to calling `getChannelData()` and accessing the resulting array, because it may avoid unnecessary memory allocation and copying.

An internal operation acquire the contents of an AudioBuffer is invoked when the contents of an `AudioBuffer` are needed by some API implementation. This operation returns immutable channel data to the invoker.

When an ***acquire the content*** operation occurs on an `AudioBuffer`, run the following steps:

1. **PROPOSED CORRECTION ISSUE 2361-4.** Use new Web IDL buffer primitives

   Previous Change | Next Change

   If ~~the operation `IsDetachedBuffer` on any of the `AudioBuffer`'s `ArrayBuffers`~~ any of the `AudioBuffer`'s `ArrayBuffers` are detached , return `true`, abort these steps, and return a zero-length channel data buffer to the invoker.

   Show Change | Show Current | Show Future

Error preparing HTML-CSS output (preProcess)

2. **PROPOSED CORRECTION ISSUE 2361-5.** Use new Web IDL buffer primitives
   Previous Change | Next Change
   ~~Detach~~ Detach all `ArrayBuffer`s for arrays previously returned by `getChannelData()` on this `AudioBuffer`.
   Show Change | Show Current | Show Future

   NOTE: Because `AudioBuffer` can only be created via `createBuffer()` or via the `AudioBuffer` constructor, this cannot throw.

3. Retain the underlying `[[internal data]]` from those `ArrayBuffer`s and return references to them to the invoker.
4. Attach `ArrayBuffer`s containing copies of the data to the `AudioBuffer`, to be returned by the next call to `getChannelData()`.

The acquire the contents of an AudioBuffer operation is invoked in the following cases:

- When `AudioBufferSourceNode.start` is called, it acquires the contents of the node's `buffer`. If the operation fails, nothing is played.
- When the `buffer` of an `AudioBufferSourceNode` is set and `AudioBufferSourceNode.start` has been previously called, the setter acquires the content of the `AudioBuffer`. If the operation fails, nothing is played.
- When a `ConvolverNode`'s `buffer` is set to an `AudioBuffer` it acquires the content of the `AudioBuffer`.
- When the dispatch of an `AudioProcessingEvent` completes, it acquires the contents of its `outputBuffer`.

NOTE: This means that `copyToChannel()` cannot be used to change the content of an `AudioBuffer` currently in use by an `AudioNode` that has acquired the content of an AudioBuffer since the `AudioNode` will continue to use the data previously acquired.

1.4.4. `AudioBufferOptions`

This specifies the options to use in constructing an `AudioBuffer`. The `length` and `sampleRate` members are required.

```
dictionary AudioBufferOptions {
    unsigned long numberOfChannels = 1;
    required unsigned long length;
    required float sampleRate;
};
```

1.4.4.1. Dictionary `AudioBufferOptions` Members

The allowed values for the members of this dictionary are constrained. See `createBuffer()`.

***`length`*, of type unsigned long**
The length in sample frames of the buffer. See `length` for constraints.

***`numberOfChannels`*, of type unsigned long, defaulting to `1`**
The number of channels for the buffer. See `numberOfChannels` for constraints.

***`sampleRate`*, of type float**
The sample rate in Hz for the buffer. See `sampleRate` for constraints.

1.5. The `AudioNode` Interface

`AudioNode`s are the building blocks of an `AudioContext`. This interface represents audio sources, the audio destination, and intermediate processing modules. These modules can be connected together to form processing graphs for rendering audio to the audio hardware. Each node can have inputs and/or outputs. A source node has no inputs and a single output. Most processing nodes such as filters will have one input and one output. Each type of `AudioNode` differs in the details of how it processes or synthesizes audio. But, in general, an `AudioNode` will process its inputs (if it has any), and generate audio for its outputs (if it has any).

Each output has one or more channels. The exact number of channels depends on the details of the specific AudioNode.

An output may connect to one or more AudioNode inputs, thus *fan-out* is supported. An input initially has no connections, but may be connected from one or more AudioNode outputs, thus *fan-in* is supported. When the connect() method is called to connect an output of an AudioNode to an input of an AudioNode, we call that a ***connection*** to the input.

Each AudioNode ***input*** has a specific number of channels at any given time. This number can change depending on the connection(s) made to the input. If the input has no connections then it has one channel which is silent.

For each input, an AudioNode performs a mixing of all connections to that input. Please see § 4 Channel Up-Mixing and Down-Mixing for normative requirements and details.

The processing of inputs and the internal operations of an AudioNode take place continuously with respect to AudioContext time, regardless of whether the node has connected outputs, and regardless of whether these outputs ultimately reach an AudioContext's AudioDestinationNode.

```
[Exposed=Window]
interface AudioNode : EventTarget {
    AudioNode connect (AudioNode destinationNode,
                       optional unsigned long output = 0,
                       optional unsigned long input = 0);
    undefined connect (AudioParam destinationParam, optional unsigned long output = 0);
    undefined disconnect ();
    undefined disconnect (unsigned long output);
    undefined disconnect (AudioNode destinationNode);
    undefined disconnect (AudioNode destinationNode, unsigned long output);
    undefined disconnect (AudioNode destinationNode,
                          unsigned long output,
                          unsigned long input);
    undefined disconnect (AudioParam destinationParam);
    undefined disconnect (AudioParam destinationParam, unsigned long output);
    readonly attribute BaseAudioContext context;
    readonly attribute unsigned long numberOfInputs;
    readonly attribute unsigned long numberOfOutputs;
    attribute unsigned long channelCount;
    attribute ChannelCountMode channelCountMode;
    attribute ChannelInterpretation channelInterpretation;
};
```

§ 1.5.1. AudioNode Creation

AudioNodes can be created in two ways: by using the constructor for this particular interface, or by using the ***factory method*** on the BaseAudioContext or AudioContext.

The BaseAudioContext passed as first argument of the constructor of an AudioNodes is called the ***associated BaseAudioContext*** of the AudioNode to be created. Similarly, when using the factory method, the associated BaseAudioContext of the AudioNode is the BaseAudioContext this factory method is called on.

To create a new AudioNode of a particular type *n* using its factory method, called on a BaseAudioContext *c*, execute these steps:

1. Let *node* be a new object of type *n*.
2. Let *option* be a dictionary of the type associated to the interface associated to this factory method.
3. For each parameter passed to the factory method, set the dictionary member of the same name on *option* to the value of this parameter.
4. Call the constructor for *n* on *node* with *c* and *option* as arguments.
5. Return *node*

Initializing an object *o* that inherits from AudioNode means executing the following steps, given the arguments *context* and *dict* passed to the constructor of this interface.

Error preparing HTML-CSS output (preProcess)

1. Set *o*'s associated `BaseAudioContext` to *context*.
2. Set its value for `numberOfInputs`, `numberOfOutputs`, `channelCount`, `channelCountMode`, `channelInterpretation` to the default value for this specific interface outlined in the section for each `AudioNode`.
3. For each member of *dict* passed in, execute these steps, with *k* the key of the member, and *v* its value. If any exceptions is thrown when executing these steps, abort the iteration and propagate the exception to the caller of the algorithm (constructor or factory method).

    1. If *k* is the name of an `AudioParam` on this interface, set the `value` attribute of this `AudioParam` to *v*.
    2. Else if *k* is the name of an attribute on this interface, set the object associated with this attribute to *v*.

✓ MDN

The ***associated interface*** for a factory method is the interface of the objects that are returned from this method. The ***associated option object*** for an interface is the option object that can be passed to the constructor for this interface.

`AudioNode`s are `EventTarget`s, as described in [DOM]. This means that it is possible to dispatch events to `AudioNode`s the same way that other `EventTarget`s accept events.

```
enum ChannelCountMode {
    "max",
    "clamped-max",
    "explicit"
};
```

The `ChannelCountMode`, in conjuction with the node's `channelCount` and `channelInterpretation` values, is used to determine the ***computedNumberOfChannels*** that controls how inputs to a node are to be mixed. The computedNumberOfChannels is determined as shown below. See § 4 Channel Up-Mixing and Down-Mixing for more information on how mixing is to be done.

`ChannelCountMode` enumeration description

Enum value	Description
"***max***"	computedNumberOfChannels is the maximum of the number of channels of all connections to an input. In this mode `channelCount` is ignored.
"***clamped-max***"	computedNumberOfChannels is determined as for "`max`" and then clamped to a maximum value of the given `channelCount`.
"***explicit***"	computedNumberOfChannels is the exact value as specified by the `channelCount`.

```
enum ChannelInterpretation {
    "speakers",
    "discrete"
};
```

`ChannelInterpretation` enumeration description

Enum value	Description
"***speakers***"	use up-mix equations or down-mix equations. In cases where the number of channels do not match any of these basic speaker layouts, revert to "`discrete`".
"***discrete***"	Up-mix by filling channels until they run out then zero out remaining channels. Down-mix by filling as many channels as possible, then dropping remaining channels.

§ 1.5.2. AudioNode Tail-Time

An `AudioNode` can have a ***tail-time***. This means that even when the `AudioNode` is fed silence, the output can be non-silent.

`AudioNode`s have a non-zero tail-time if they have internal processing state such that input in the past affects the future output. `AudioNode`s may continue to produce non-silent output for the calculated tail-time even after the input transitions

✓ MDN

Error preparing HTML-CSS output (preProcess)

§ 1.5.3. AudioNode Lifetime

AudioNode can be ***actively processing*** during a render quantum, if any of the following conditions hold.

- An AudioScheduledSourceNode is actively processing if and only if it is playing for at least part of the current rendering quantum.
- A MediaElementAudioSourceNode is actively processing if and only if its mediaElement is playing for at least part of the current rendering quantum.
- A MediaStreamAudioSourceNode or a MediaStreamTrackAudioSourceNode are actively processing when the associated MediaStreamTrack object has a `readyState` attribute equal to `"live"`, a `muted` attribute equal to `false` and an `enabled` attribute equal to `true`.
- A DelayNode in a cycle is actively processing only when the absolute value of any output sample for the current render quantum is greater than or equal to 2^{-126}.
- A ScriptProcessorNode is actively processing when its input or output is connected.
- An AudioWorkletNode is actively processing when its AudioWorkletProcessor's [[callable process]] returns `true` and either its active source flag is `true` or any AudioNode connected to one of its inputs is actively processing.
- All other AudioNodes start actively processing when any AudioNode connected to one of its inputs is actively processing, and stops actively processing when the input that was received from other actively processing AudioNode no longer affects the output.

> NOTE: This takes into account AudioNodes that have a tail-time.

AudioNodes that are not actively processing output a single channel of silence.

§ 1.5.4. Attributes

***channelCount*, of type unsigned long**

channelCount is the number of channels used when up-mixing and down-mixing connections to any inputs to the node. The default value is 2 except for specific nodes where its value is specially determined. This attribute has no effect for nodes with no inputs. If this value is set to zero or to a value greater than the implementation's maximum number of channels the implementation MUST throw a NotSupportedError exception.

In addition, some nodes have additional ***channelCount constraints*** on the possible values for the channel count:

AudioDestinationNode

The behavior depends on whether the destination node is the destination of an AudioContext or OfflineAudioContext:

AudioContext

The channel count MUST be between 1 and maxChannelCount. An IndexSizeError exception MUST be thrown for any attempt to set the count outside this range.

OfflineAudioContext

The channel count cannot be changed. An InvalidStateError exception MUST be thrown for any attempt to change the value.

AudioWorkletNode

See § 1.32.4.3.2 Configuring Channels with AudioWorkletNodeOptions Configuring Channels with AudioWorkletNodeOptions.

ChannelMergerNode

The channel count cannot be changed, and an InvalidStateError exception MUST be thrown for any attempt to change the value.

ChannelSplitterNode

The channel count cannot be changed, and an InvalidStateError exception MUST be thrown for any attempt to change the value.

ConvolverNode

The channel count cannot be greater than two, and a NotSupportedError exception MUST be thrown for any attempt to change it to a value greater than two.

MDN

Error preparing HTML-CSS output (preProcess)

DynamicsCompressorNode
The channel count cannot be greater than two, and a NotSupportedError exception MUST be thrown for any attempt to change it to a value greater than two.

PannerNode
The channel count cannot be greater than two, and a NotSupportedError exception MUST be thrown for any attempt to change it to a value greater than two.

ScriptProcessorNode
The channel count cannot be changed, and an NotSupportedError exception MUST be thrown for any attempt to change the value.

StereoPannerNode
The channel count cannot be greater than two, and a NotSupportedError exception MUST be thrown for any attempt to change it to a value greater than two.

See § 4 Channel Up-Mixing and Down-Mixing for more information on this attribute.

***channelCountMode*, of type ChannelCountMode**
channelCountMode determines how channels will be counted when up-mixing and down-mixing connections to any inputs to the node. The default value is "max". This attribute has no effect for nodes with no inputs.

In addition, some nodes have additional ***channelCountMode constraints*** on the possible values for the channel count mode:

AudioDestinationNode
If the AudioDestinationNode is the destination node of an OfflineAudioContext, then the channel count mode cannot be changed. An InvalidStateError exception MUST be thrown for any attempt to change the value.

ChannelMergerNode
The channel count mode cannot be changed from "explicit" and an InvalidStateError exception MUST be thrown for any attempt to change the value.

ChannelSplitterNode
The channel count mode cannot be changed from "explicit" and an InvalidStateError exception MUST be thrown for any attempt to change the value.

ConvolverNode
The channel count mode cannot be set to "max", and a NotSupportedError exception MUST be thrown for any attempt to set it to "max".

DynamicsCompressorNode
The channel count mode cannot be set to "max", and a NotSupportedError exception MUST be thrown for any attempt to set it to "max".

PannerNode
The channel count mode cannot be set to "max", and a NotSupportedError exception MUST be thrown for any attempt to set it to "max".

ScriptProcessorNode
The channel count mode cannot be changed from "explicit" and an NotSupportedError exception MUST be thrown for any attempt to change the value.

StereoPannerNode
The channel count mode cannot be set to "max", and a NotSupportedError exception MUST be thrown for any attempt to set it to "max".

See the § 4 Channel Up-Mixing and Down-Mixing section for more information on this attribute.

***channelInterpretation*, of type ChannelInterpretation**
channelInterpretation determines how individual channels will be treated when up-mixing and down-mixing connections to any inputs to the node. The default value is "speakers". This attribute has no effect for nodes with no inputs.

In addition, some nodes have additional ***channelInterpretation constraints*** on the possible values for the channel interpretation:

ChannelSplitterNode
The channel intepretation can not be changed from "discrete" and a InvalidStateError exception MUST be thrown for any attempt to change the value.

Error preparing HTML-CSS output (preProcess)

See § 4 Channel Up-Mixing and Down-Mixing for more information on this attribute.

***context*, of type BaseAudioContext, readonly**
The `BaseAudioContext` which owns this `AudioNode`.

***numberOfInputs*, of type unsigned long, readonly**
The number of inputs feeding into the `AudioNode`. For ***source nodes***, this will be 0. This attribute is predetermined for many `AudioNode` types, but some `AudioNode`s, like the `ChannelMergerNode` and the `AudioWorkletNode`, have variable number of inputs.

***numberOfOutputs*, of type unsigned long, readonly**
The number of outputs coming out of the `AudioNode`. This attribute is predetermined for some `AudioNode` types, but can be variable, like for the `ChannelSplitterNode` and the `AudioWorkletNode`.

§ 1.5.5. Methods

connect(destinationNode, output, input)
There can only be one connection between a given output of one specific node and a given input of another specific node. Multiple connections with the same termini are ignored.

> EXAMPLE 5
> For example:
>
> ```
> nodeA.connect(nodeB);
> nodeA.connect(nodeB);
> ```
>
> will have the same effect as
>
> ```
> nodeA.connect(nodeB);
> ```

This method returns `destination` `AudioNode` object.

Arguments for the AudioNode.connect(destinationNode, output, input) method.

Parameter	Type	Nullable	Optional	Description
destinationNode				The `destination` parameter is the `AudioNode` to connect to. ⌛ If the `destination` parameter is an `AudioNode` that has been created using another `AudioContext`, an `InvalidAccessError` MUST be thrown. That is, `AudioNode`s cannot be shared between `AudioContext`s. Multiple `AudioNode`s can be connected to the same `AudioNode`, this is described in Channel Upmixing and down mixing section.
output	`unsigned long`	✗	✓	The `output` parameter is an index describing which output of the `AudioNode` from which to connect. ⌛ If this parameter is out-of-bounds, an `IndexSizeError` exception MUST be thrown. It is possible to connect an `AudioNode` output to more than one input with multiple calls to connect(). Thus, "fan-out" is supported.
input				The `input` parameter is an index describing which input of the destination `AudioNode` to connect to. ⌛ If this parameter is out-of-bounds, an `IndexSizeError` exception MUST be thrown. It is possible to connect an `AudioNode` to another `AudioNode` which

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
				creates a ***cycle***: an AudioNode may connect to another AudioNode, which in turn connects back to the input or AudioParam of the first AudioNode.

Return type: AudioNode

connect(destinationParam, output)

Connects the AudioNode to an AudioParam, controlling the parameter value with an a-rate signal.

It is possible to connect an AudioNode output to more than one AudioParam with multiple calls to connect(). Thus, "fan-out" is supported.

It is possible to connect more than one AudioNode output to a single AudioParam with multiple calls to connect(). Thus, "fan-in" is supported.

An AudioParam will take the rendered audio data from any AudioNode output connected to it and convert it to mono by down-mixing if it is not already mono, then mix it together with other such outputs and finally will mix with the *intrinsic* parameter value (the `value` the AudioParam would normally have without any audio connections), including any timeline changes scheduled for the parameter.

The down-mixing to mono is equivalent to the down-mixing for an AudioNode with `channelCount` = 1, `channelCountMode` = "`explicit`", and `channelInterpretation` = "`speakers`".

There can only be one connection between a given output of one specific node and a specific AudioParam. Multiple connections with the same termini are ignored.

> EXAMPLE 6
> For example:
>
> ```
> nodeA.connect(param);
> nodeA.connect(param);
> ```
>
> will have the same effect as
>
> ```
> nodeA.connect(param);
> ```

Arguments for the AudioNode.connect(destinationParam, output) method.

Parameter	Type	Nullable	Optional	Description
destinationParam	AudioParam	✗	✗	The `destination` parameter is the AudioParam to connect to. This method does not return the `destination` AudioParam object. ⌛ If `destinationParam` belongs to an AudioNode that belongs to a BaseAudioContext that is different from the BaseAudioContext that has created the AudioNode on which this method was called, an `InvalidAccessError` MUST be thrown.
output	unsigned long	✗	✓	The `output` parameter is an index describing which output of the AudioNode from which to connect. ⌛ If the `parameter` is out-of-bounds, an

Parameter	Type	Nullable	Optional	Description
				IndexSizeError exception MUST be thrown.

Return type: undefined

disconnect()

Disconnects all outgoing connections from the AudioNode.

No parameters.

Return type: undefined

disconnect(output)

Disconnects a single output of the AudioNode from any other AudioNode or AudioParam objects to which it is connected.

Arguments for the AudioNode.disconnect(output) method.

Parameter	Type	Nullable	Optional	Description
output	unsigned long	✗	✗	This parameter is an index describing which output of the AudioNode to disconnect. It disconnects all outgoing connections from the given output. ⌛ If this parameter is out-of-bounds, an IndexSizeError exception MUST be thrown.

Return type: undefined

disconnect(destinationNode)

Disconnects all outputs of the AudioNode that go to a specific destination AudioNode.

Arguments for the AudioNode.disconnect(destinationNode) method.

Parameter	Type	Nullable	Optional	Description
destinationNode				The destinationNode parameter is the AudioNode to disconnect. It disconnects all outgoing connections to the given destinationNode. ⌛ If there is no connection to the destinationNode, an InvalidAccessError exception MUST be thrown.

Return type: undefined

disconnect(destinationNode, output)

Disconnects a specific output of the AudioNode from any and all inputs of some destination AudioNode.

Arguments for the AudioNode.disconnect(destinationNode, output) method.

Parameter	Type	Nullable	Optional	Description
destinationNode				The destinationNode parameter is the AudioNode to disconnect. ⌛ If there is no connection to the destinationNode from the given output, an InvalidAccessError exception MUST be thrown.
output	unsigned long	✗	✗	The output parameter is an index describing which output of the AudioNode from which to disconnect. ⌛ If this parameter is out-of-bounds, an IndexSizeError exception MUST be thrown.

Error preparing HTML-CSS output (preProcess)

Return type: `undefined`

disconnect(destinationNode, output, input)

Disconnects a specific output of the `AudioNode` from a specific input of some destination `AudioNode`.

Arguments for the AudioNode.disconnect(destinationNode, output, input) method.

Parameter	Type	Nullable	Optional	Description
destinationNode				The `destinationNode` parameter is the `AudioNode` to disconnect. ⏳ If there is no connection to the `destinationNode` from the given input to the given output, an `InvalidAccessError` exception MUST be thrown.
output	`unsigned long`	✗	✗	The `output` parameter is an index describing which output of the `AudioNode` from which to disconnect. ⏳ If this parameter is out-of-bounds, an `IndexSizeError` exception MUST be thrown.
input				The `input` parameter is an index describing which input of the destination `AudioNode` to disconnect. ⏳ If this parameter is out-of-bounds, an `IndexSizeError` exception MUST be thrown.

✓ MDN ✓ MDN ✓ MDN ✓ MDN

Return type: `undefined`

disconnect(destinationParam)

Disconnects all outputs of the `AudioNode` that go to a specific destination `AudioParam`. The contribution of this `AudioNode` to the computed parameter value goes to 0 when this operation takes effect. The intrinsic parameter value is not affected by this operation.

Arguments for the AudioNode.disconnect(destinationParam) method.

Parameter	Type	Nullable	Optional	Description
destinationParam	`AudioParam`	✗	✗	The `destinationParam` parameter is the `AudioParam` to disconnect. ⏳ If there is no connection to the `destinationParam`, an `InvalidAccessError` exception MUST be thrown.

MDN

Return type: `undefined`

disconnect(destinationParam, output)

Disconnects a specific output of the `AudioNode` from a specific destination `AudioParam`. The contribution of this `AudioNode` to the computed parameter value goes to 0 when this operation takes effect. The intrinsic parameter value is not affected by this operation.

Arguments for the AudioNode.disconnect(destinationParam, output) method.

Parameter	Type	Nullable	Optional	Description
destinationParam	`AudioParam`	✗	✗	The `destinationParam` parameter is the `AudioParam` to disconnect. ⏳ If there is no connection to the `destinationParam`, an `InvalidAccessError` exception MUST be thrown.

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
output	unsigned long	✗	✗	The output parameter is an index describing which output of the AudioNode from which to disconnect. If the parameter is out-of-bounds, an IndexSizeError exception MUST be thrown.

Return type: undefined

§ 1.5.6. AudioNodeOptions

This specifies the options that can be used in constructing all AudioNodes. All members are optional. However, the specific values used for each node depends on the actual node.

```
dictionary AudioNodeOptions {
    unsigned long channelCount;
    ChannelCountMode channelCountMode;
    ChannelInterpretation channelInterpretation;
};
```

§ *1.5.6.1. Dictionary AudioNodeOptions Members*

***channelCount*, of type unsigned long**
: Desired number of channels for the channelCount attribute.

***channelCountMode*, of type ChannelCountMode**
: Desired mode for the channelCountMode attribute.

***channelInterpretation*, of type ChannelInterpretation**
: Desired mode for the channelInterpretation attribute.

§ 1.6. The AudioParam Interface

AudioParam controls an individual aspect of an AudioNode's functionality, such as volume. The parameter can be set immediately to a particular value using the value attribute. Or, value changes can be scheduled to happen at very precise times (in the coordinate system of AudioContext's currentTime attribute), for envelopes, volume fades, LFOs, filter sweeps, grain windows, etc. In this way, arbitrary timeline-based automation curves can be set on any AudioParam. Additionally, audio signals from the outputs of AudioNodes can be connected to an AudioParam, summing with the *intrinsic* parameter value.

Some synthesis and processing AudioNodes have AudioParams as attributes whose values MUST be taken into account on a per-audio-sample basis. For other AudioParams, sample-accuracy is not important and the value changes can be sampled more coarsely. Each individual AudioParam will specify that it is either an a-rate parameter which means that its values MUST be taken into account on a per-audio-sample basis, or it is a k-rate parameter.

Implementations MUST use block processing, with each AudioNode processing one render quantum.

For each render quantum, the value of a ***k-rate*** parameter MUST be sampled at the time of the very first sample-frame, and that value MUST be used for the entire block. ***a-rate*** parameters MUST be sampled for each sample-frame of the block. Depending on the AudioParam, its rate can be controlled by setting the automationRate attribute to either "a-rate" or "k-rate". See the description of the individual AudioParams for further details.

Each AudioParam includes minValue and maxValue attributes that together form the ***simple nominal range*** for the parameter. In effect, value of the parameter is clamped to the range [minValue, maxValue]. See § 1.6.3 Computation of Value for full details.

For many AudioParams the minValue and maxValue is intended to be set to the maximum possible range. In this case, [illegible] to the ***most-positive-single-float*** value, which is 3.4028235e38. (However, in JavaScript which only supports IEEE-754 double precision float values, this must be written as 3.40282346638528860e38.) Similarly,

Error preparing HTML-CSS output (preProcess)

`minValue` should be set to the ***most-negative-single-float*** value, which is the negative of the most-positive-single-float: -3.4028235e38. (Similarly, this must be written in JavaScript as -3.4028234663852886e38.)

An `AudioParam` maintains a list of zero or more ***automation events***. Each automation event specifies changes to the parameter's value over a specific time range, in relation to its ***automation event time*** in the time coordinate system of the `AudioContext`'s `currentTime` attribute. The list of automation events is maintained in ascending order of automation event time.

The behavior of a given automation event is a function of the `AudioContext`'s current time, as well as the automation event times of this event and of adjacent events in the list. The following ***automation methods*** change the event list by adding a new event to the event list, of a type specific to the method:

- `setValueAtTime()` - `SetValue`
- `linearRampToValueAtTime()` - `LinearRampToValue`
- `exponentialRampToValueAtTime()` - `ExponentialRampToValue`
- `setTargetAtTime()` - `SetTarget`
- `setValueCurveAtTime()` - `SetValueCurve`

The following rules will apply when calling these methods:

- Automation event times are not quantized with respect to the prevailing sample rate. Formulas for determining curves and ramps are applied to the exact numerical times given when scheduling events.
- If one of these events is added at a time where there is already one or more events, then it will be placed in the list after them, but before events whose times are after the event.
- ⏳ If setValueCurveAtTime() is called for time T and duration D and there are any events having a time strictly greater than T, but strictly less than $T + D$, then a `NotSupportedError` exception MUST be thrown. In other words, it's not ok to schedule a value curve during a time period containing other events, but it's ok to schedule a value curve exactly at the time of another event.
- ⏳ Similarly a `NotSupportedError` exception MUST be thrown if any automation method is called at a time which is contained in $[T, T + D)$, T being the time of the curve and D its duration.

✓ MDN

> NOTE: `AudioParam` attributes are read only, with the exception of the `value` attribute.

The automation rate of an `AudioParam` can be selected setting the `automationRate` attribute with one of the following values. However, some `AudioParam`s have constraints on whether the automation rate can be changed.

```
enum AutomationRate {
    "a-rate",
    "k-rate"
};
```

`AutomationRate` enumeration description

Enum value	Description
"***a-rate***"	This `AudioParam` is set for a-rate processing.
"***k-rate***"	This `AudioParam` is set for k-rate processing.

Each `AudioParam` has an internal slot ***`[[current value]]`***, initially set to the `AudioParam`'s `defaultValue`.

```
[Exposed=Window]
interface AudioParam {
    attribute float value;
    attribute AutomationRate automationRate;
    readonly attribute float defaultValue;
    readonly attribute float minValue;
    readonly attribute float maxValue;
    AudioParam setValueAtTime (float value, double startTime);
    AudioParam linearRampToValueAtTime (float value, double endTime);
```

MDN

Error preparing HTML-CSS output (preProcess)

```
    AudioParam exponentialRampToValueAtTime (float value, double endTime);
    AudioParam setTargetAtTime (float target, double startTime, float timeConstant);
    AudioParam setValueCurveAtTime (sequence<float> values,
                                    double startTime,
                                    double duration);
    AudioParam cancelScheduledValues (double cancelTime);
    AudioParam cancelAndHoldAtTime (double cancelTime);
};
```

§ 1.6.1. Attributes

***automationRate*, of type AutomationRate**

The automation rate for the AudioParam. The default value depends on the actual AudioParam; see the description of each individual AudioParam for the default value.

Some nodes have additional ***automation rate constraints*** as follows:

AudioBufferSourceNode

The AudioParams playbackRate and detune MUST be "k-rate". ⌛ An InvalidStateError must be thrown if the rate is changed to "a-rate".

DynamicsCompressorNode

The AudioParams threshold, knee, ratio, attack, and release MUST be "k-rate". ⌛ An InvalidStateError must be thrown if the rate is changed to "a-rate".

PannerNode

If the panningModel is "HRTF", the setting of the automationRate for any AudioParam of the PannerNode is ignored. Likewise, the setting of the automationRate for any AudioParam of the AudioListener is ignored. In this case, the AudioParam behaves as if the automationRate were set to "k-rate".

***defaultValue*, of type float, readonly**

Initial value for the value attribute.

***maxValue*, of type float, readonly**

The nominal maximum value that the parameter can take. Together with minValue, this forms the nominal range for this parameter.

***minValue*, of type float, readonly**

The nominal minimum value that the parameter can take. Together with maxValue, this forms the nominal range for this parameter.

***value*, of type float**

The parameter's floating-point value. This attribute is initialized to the defaultValue.

Getting this attribute returns the contents of the [[current value]] slot. See § 1.6.3 Computation of Value for the algorithm for the value that is returned.

MDN

Setting this attribute has the effect of assigning the requested value to the [[current value]] slot, and calling the setValueAtTime() method with the current AudioContext's currentTime and [[current value]]. Any exceptions that would be thrown by setValueAtTime() will also be thrown by setting this attribute.

§ 1.6.2. Methods

cancelAndHoldAtTime(cancelTime)

This is similar to cancelScheduledValues() in that it cancels all scheduled parameter changes with times greater than or equal to cancelTime. However, in addition, the automation value that would have happened at cancelTime is then proprogated for all future time until other automation events are introduced.

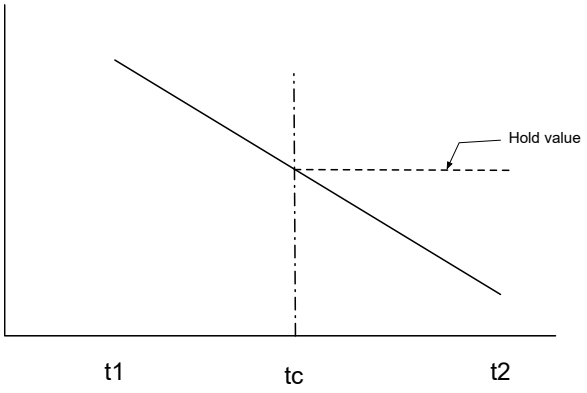
The behavior of the timeline in the face of cancelAndHoldAtTime() when automations are running and can be introduced at any time after calling cancelAndHoldAtTime() and before cancelTime is reached is quite complicated. The behavior of cancelAndHoldAtTime() is therefore specified in the following algorithm.

Let t_c be the value of cancelTime. Then

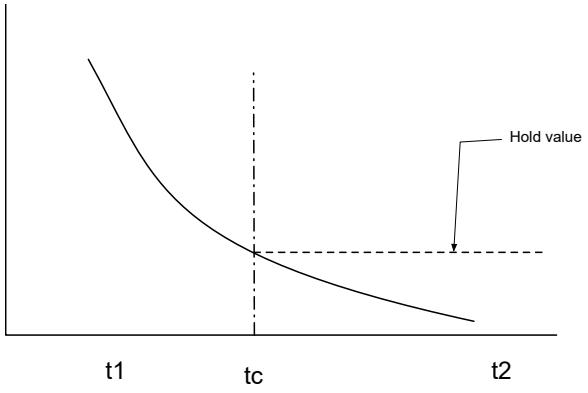
Error preparing HTML-CSS output (preProcess)

vent (if any) at time t_1 where t_1 is the largest number satisfying $t_1 \leq t_c$.

2. Let E_2 be the event (if any) at time t_2 where t_2 is the smallest number satisfying $t_c < t_2$.
3. If E_2 exists:
    1. If E_2 is a linear or exponential ramp,
        1. Effectively rewrite E_2 to be the same kind of ramp ending at time t_c with an end value that would be the value of the original ramp at time t_c.



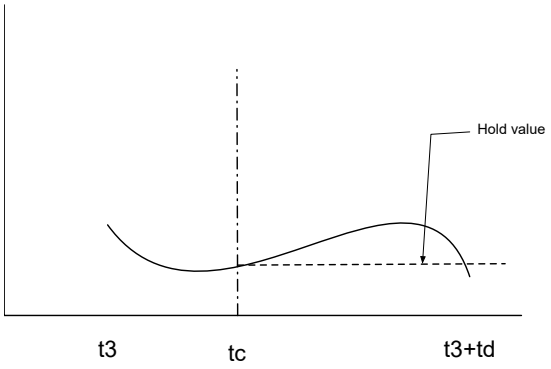
        2. Go to step 5.
    2. Otherwise, go to step 4.
4. If E_1 exists:
    1. If E_1 is a `setTarget` event,
        1. Implicitly insert a `setValueAtTime` event at time t_c with the value that the `setTarget` would have at time t_c.



        2. Go to step 5.
    2. If E_1 is a `setValueCurve` with a start time of t_3 and a duration of d
        1. If $t_c > t_3 + d$, go to step 5.
        2. Otherwise,
            1. Effectively replace this event with a `setValueCurve` event with a start time of t_3 and a new duration of $t_c - t_3$. However, this is not a true replacement; this automation MUST take care to produce the same output as the original, and not one computed using a different duration. (That

Error preparing HTML-CSS output (preProcess)

would cause sampling of the value curve in a slightly different way, producing different results.)



2. Go to step 5.

5. Remove all events with time greater than t_c.

If no events are added, then the automation value after `cancelAndHoldAtTime()` is the constant value that the original timeline would have had at time t_c.

CANDIDATE CORRECTION ISSUE 127:
A `RangeError` is only thrown for negative cancelTime values for cancelAndHoldAtTime and cancelScheduledValues. See Issue 127
Show Change | Show Current | Show Future

Arguments for the AudioParam.cancelAndHoldAtTime() method.

Parameter	Type	Nullable	Optional	Description
cancelTime	`double`	✗	✗	The time after which any previously scheduled parameter changes will be cancelled. It is a time in the same time coordinate system as the `AudioContext`'s `currentTime` attribute. ⌛ A `RangeError` exception MUST be thrown if `cancelTime` is negative ~~or is not a finite number~~. If `cancelTime` is less than `currentTime`, it is clamped to `currentTime`.

Return type: `AudioParam`

✓ MDN

cancelScheduledValues(cancelTime)

Cancels all scheduled parameter changes with times greater than or equal to `cancelTime`. Cancelling a scheduled parameter change means removing the scheduled event from the event list. Any active automations whose automation event time is less than `cancelTime` are also cancelled, and such cancellations may cause discontinuities because the original value (from before such automation) is restored immediately. Any hold values scheduled by `cancelAndHoldAtTime()` are also removed if the hold time occurs after `cancelTime`.

For a `setValueCurveAtTime()`, let T_0 and T_D be the corresponding `startTime` and `duration`, respectively of this event. Then if `cancelTime` is in the range $[T_0, T_0 + T_D]$, the event is removed from the timeline.

Arguments for the AudioParam.cancelScheduledValues() method.

Parameter	Type	Nullable	Optional	Description
cancelTime	`double`	✗	✗	The time after which any previously scheduled parameter changes will be cancelled. It is a time in the same time coordinate system as the `AudioContext`'s `currentTime` attribute. ⌛ A `RangeError` exception MUST be thrown if `cancelTime` is negative ~~or is not a~~

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
				~~finite number~~. If `cancelTime` is less than `currentTime`, it is clamped to `currentTime`.

Return type: `AudioParam`

exponentialRampToValueAtTime(value, endTime)

Schedules an exponential continuous change in parameter value from the previous scheduled parameter value to the given value. Parameters representing filter frequencies and playback rate are best changed exponentially because of the way humans perceive sound.

The value during the time interval $T_0 \leq t < T_1$ (where T_0 is the time of the previous event and T_1 is the `endTime` parameter passed into this method) will be calculated as:

$$v(t) = V_0\left(\frac{V_1}{V_0}\right)^{\frac{t-T_0}{T_1-T_0}}$$

where V_0 is the value at the time T_0 and V_1 is the `value` parameter passed into this method. If V_0 and V_1 have opposite signs or if V_0 is zero, then $v(t) = V_0$ for $T_0 \leq t < T_1$.

This also implies an exponential ramp to 0 is not possible. A good approximation can be achieved using `setTargetAtTime()` with an appropriately chosen time constant.

If there are no more events after this *ExponentialRampToValue* event then for $t \geq T_1$, $v(t) = V_1$.

If there is no event preceding this event, the exponential ramp behaves as if `setValueAtTime(value, currentTime)` were called where `value` is the current value of the attribute and `currentTime` is the context `currentTime` at the time `exponentialRampToValueAtTime()` is called.

If the preceding event is a `SetTarget` event, T_0 and V_0 are chosen from the current time and value of `SetTarget` automation. That is, if the `SetTarget` event has not started, T_0 is the start time of the event, and V_0 is the value just before the `SetTarget` event starts. In this case, the `ExponentialRampToValue` event effectively replaces the `SetTarget` event. If the `SetTarget` event has already started, T_0 is the current context time, and V_0 is the current `SetTarget` automation value at time T_0. In both cases, the automation curve is continuous.

Arguments for the AudioParam.exponentialRampToValueAtTime() method.

Parameter	Type	Nullable	Optional	Description
value	`float`	✗	✗	The value the parameter will exponentially ramp to at the given time. ⌛ A `RangeError` exception MUST be thrown if this value is equal to 0.
endTime	`double`	✗	✗	The time in the same time coordinate system as the `AudioContext`'s `currentTime` attribute where the exponential ramp ends. ⌛ A `RangeError` exception MUST be thrown if `endTime` is negative or is not a finite number. If *endTime* is less than `currentTime`, it is clamped to `currentTime`.

Return type: `AudioParam`

linearRampToValueAtTime(value, endTime)

Schedules a linear continuous change in parameter value from the previous scheduled parameter value to the given value.

Error preparing HTML-CSS output (preProcess)

The value during the time interval $T_0 \leq t < T_1$ (where T_0 is the time of the previous event and T_1 is the `endTime` parameter passed into this method) will be calculated as:

$$v(t) = V_0 + (V_1 - V_0)\frac{t - T_0}{T_1 - T_0}$$

where V_0 is the value at the time T_0 and V_1 is the `value` parameter passed into this method.

If there are no more events after this *LinearRampToValue* event then for $t \geq T_1$, $v(t) = V_1$.

If there is no event preceding this event, the linear ramp behaves as if `setValueAtTime(value, currentTime)` were called where `value` is the current value of the attribute and `currentTime` is the context `currentTime` at the time `linearRampToValueAtTime()` is called.

If the preceding event is a `SetTarget` event, T_0 and V_0 are chosen from the current time and value of `SetTarget` automation. That is, if the `SetTarget` event has not started, T_0 is the start time of the event, and V_0 is the value just before the `SetTarget` event starts. In this case, the `LinearRampToValue` event effectively replaces the `SetTarget` event. If the `SetTarget` event has already started, T_0 is the current context time, and V_0 is the current `SetTarget` automation value at time T_0. In both cases, the automation curve is continuous.

Arguments for the AudioParam.linearRampToValueAtTime() method.

Parameter	Type	Nullable	Optional	Description
value	`float`	✗	✗	The value the parameter will linearly ramp to at the given time.
endTime	`double`	✗	✗	The time in the same time coordinate system as the `AudioContext`'s `currentTime` attribute at which the automation ends. ⌛ A `RangeError` exception MUST be thrown if `endTime` is negative or is not a finite number. If *endTime* is less than `currentTime`, it is clamped to `currentTime`.

Return type: `AudioParam`

`setTargetAtTime(target, startTime, timeConstant)`

Start exponentially approaching the target value at the given time with a rate having the given time constant. Among other uses, this is useful for implementing the "decay" and "release" portions of an ADSR envelope. Please note that the parameter value does not immediately change to the target value at the given time, but instead gradually changes to the target value.

During the time interval: $T_0 \leq t$, where T_0 is the `startTime` parameter:

$$v(t) = V_1 + (V_0 - V_1)\,e^{-\left(\frac{t - T_0}{\tau}\right)}$$

where V_0 is the initial value (the `[[current value]]` attribute) at T_0 (the `startTime` parameter), V_1 is equal to the `target` parameter, and τ is the `timeConstant` parameter.

If a `LinearRampToValue` or `ExponentialRampToValue` event follows this event, the behavior is described in `linearRampToValueAtTime()` or `exponentialRampToValueAtTime()`, respectively. For all other events, the `SetTarget` event ends at the time of the next event.

Arguments for the AudioParam.setTargetAtTime() method.

Parameter	Type	Nullable	Optional	Description

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
target	float	✗	✗	The value the parameter will *start* changing to at the given time.
startTime	double	✗	✗	The time at which the exponential approach will begin, in the same time coordinate system as the AudioContext's currentTime attribute. ⌛ A RangeError exception MUST be thrown if start is negative or is not a finite number. If *startTime* is less than currentTime, it is clamped to currentTime.
timeConstant	float	✗	✗	The time-constant value of first-order filter (exponential) approach to the target value. The larger this value is, the slower the transition will be. ⌛ The value MUST be non-negative or a RangeError exception MUST be thrown. If timeConstant is zero, the output value jumps immediately to the final value. More precisely, *timeConstant* is the time it takes a first-order linear continuous time-invariant system to reach the value $1 - 1/e$ (around 63.2%) given a step input response (transition from 0 to 1 value).

Return type: AudioParam

setValueAtTime(value, startTime)

Schedules a parameter value change at the given time.

If there are no more events after this SetValue event, then for $t \geq T_0$, $v(t) = V$, where T_0 is the startTime parameter and V is the value parameter. In other words, the value will remain constant.

If the next event (having time T_1) after this SetValue event is not of type LinearRampToValue or ExponentialRampToValue, then, for $T_0 \leq t < T_1$:

$$v(t) = V$$

In other words, the value will remain constant during this time interval, allowing the creation of "step" functions.

If the next event after this SetValue event is of type LinearRampToValue or ExponentialRampToValue then please see linearRampToValueAtTime() or exponentialRampToValueAtTime(), respectively.

Arguments for the AudioParam.setValueAtTime() method.

Parameter	Type	Nullable	Optional	Description
value	float	✗	✗	The value the parameter will change to at the given time.
startTime	double	✗	✗	The time in the same time coordinate system as the BaseAudioContext's currentTime attribute at which the parameter changes to the given value. ⌛ A RangeError exception MUST be thrown if startTime is negative or is not a finite number. If *startTime* is less than currentTime, it is clamped to currentTime.

Return type: AudioParam

setValueCurveAtTime(values, startTime, duration)

Sets an array of arbitrary parameter values starting at the given time for the given duration. The number of values will be the desired duration.

Let T_0 be `startTime`, T_D be `duration`, V be the `values` array, and N be the length of the `values` array. Then, during the time interval: $T_0 \le t < T_0 + T_D$, let

$$k = \left\lfloor \frac{N-1}{T_D}(t - T_0) \right\rfloor$$

Then $v(t)$ is computed by linearly interpolating between $V[k]$ and $V[k+1]$,

After the end of the curve time interval ($t \ge T_0 + T_D$), the value will remain constant at the final curve value, until there is another automation event (if any).

An implicit call to `setValueAtTime()` is made at time $T_0 + T_D$ with value $V[N-1]$ so that following automations will start from the end of the `setValueCurveAtTime()` event.

✓ MDN

Arguments for the AudioParam.setValueCurveAtTime() method.

Parameter	Type	Nullable	Optional	Description
values	`sequence<float>`	✗	✗	A sequence of float values representing a parameter value curve. These values will apply starting at the given time and lasting for the given duration. When this method is called, an internal copy of the curve is created for automation purposes. Subsequent modifications of the contents of the passed-in array therefore have no effect on the `AudioParam`. ⌛ An `InvalidStateError` MUST be thrown if this attribute is a `sequence<float>` object that has a length less than 2.
startTime	`double`	✗	✗	The start time in the same time coordinate system as the `AudioContext`'s `currentTime` attribute at which the value curve will be applied. ⌛ A `RangeError` exception MUST be thrown if `startTime` is negative or is not a finite number. If *startTime* is less than `currentTime`, it is clamped to `currentTime`.
duration	`double`	✗	✗	The amount of time in seconds (after the `startTime` parameter) where values will be calculated according to the `values` parameter. ⌛ A `RangeError` exception MUST be thrown if `duration` is not strictly positive or is not a finite number.

✓ MDN

Return type: `AudioParam`

§ 1.6.3. Computation of Value

There are two different kind of `AudioParam`s, simple parameters and compound parameters. ***Simple parameters*** (the default) are used on their own to compute the final audio output of an `AudioNode`. ***Compound parameters*** are `AudioParam`s that are used with other `AudioParam`s to compute a value that is then used as an input to compute the output of an `AudioNode`.

Error preparing HTML-CSS output (preProcess)

e final value controlling the audio DSP and is computed by the audio rendering thread during each rendering time quantum.

The computation of the value of an `AudioParam` consists of two parts:

- the *paramIntrinsicValue* value that is computed from the `value` attribute and any automation events.
- the *paramComputedValue* that is the final value controlling the audio DSP and is computed by the audio rendering thread during each render quantum.

These values MUST be computed as follows:

1. *paramIntrinsicValue* will be calculated at each time, which is either the value set directly to the `value` attribute, or, if there are any automation events with times before or at this time, the value as calculated from these events. If automation events are removed from a given time range, then the *paramIntrinsicValue* value will remain unchanged and stay at its previous value until either the `value` attribute is directly set, or automation events are added for the time range.
2. Set `[[current value]]` to the value of *paramIntrinsicValue* at the beginning of this render quantum.
3. *paramComputedValue* is the sum of the *paramIntrinsicValue* value and the value of the input AudioParam buffer. If the sum is `NaN`, replace the sum with the `defaultValue`.
4. If this `AudioParam` is a compound parameter, compute its final value with other `AudioParam`s.
5. Set computedValue to *paramComputedValue*.

The ***nominal range*** for a computedValue are the lower and higher values this parameter can effectively have. For simple parameters, the computedValue is clamped to the simple nominal range for this parameter. Compound parameters have their final value clamped to their nominal range after having been computed from the different `AudioParam` values they are composed of.

When automation methods are used, clamping is still applied. However, the automation is run as if there were no clamping at all. Only when the automation values are to be applied to the output is the clamping done as specified above.

Error preparing HTML-CSS output (preProcess)

EXAMPLE 7

For example, consider a node *N* has an AudioParam *p* with a nominal range of [0, 1], and following automation sequence

```
N.p.setValueAtTime(0, 0);
N.p.linearRampToValueAtTime(4, 1);
N.p.linearRampToValueAtTime(0, 2);
```

The initial slope of the curve is 4, until it reaches the maximum value of 1, at which time, the output is held constant. Finally, near time 2, the slope of the curve is -4. This is illustrated in the graph below where the dashed line indicates what would have happened without clipping, and the solid line indicates the actual expected behavior of the audioparam due to clipping to the nominal range.

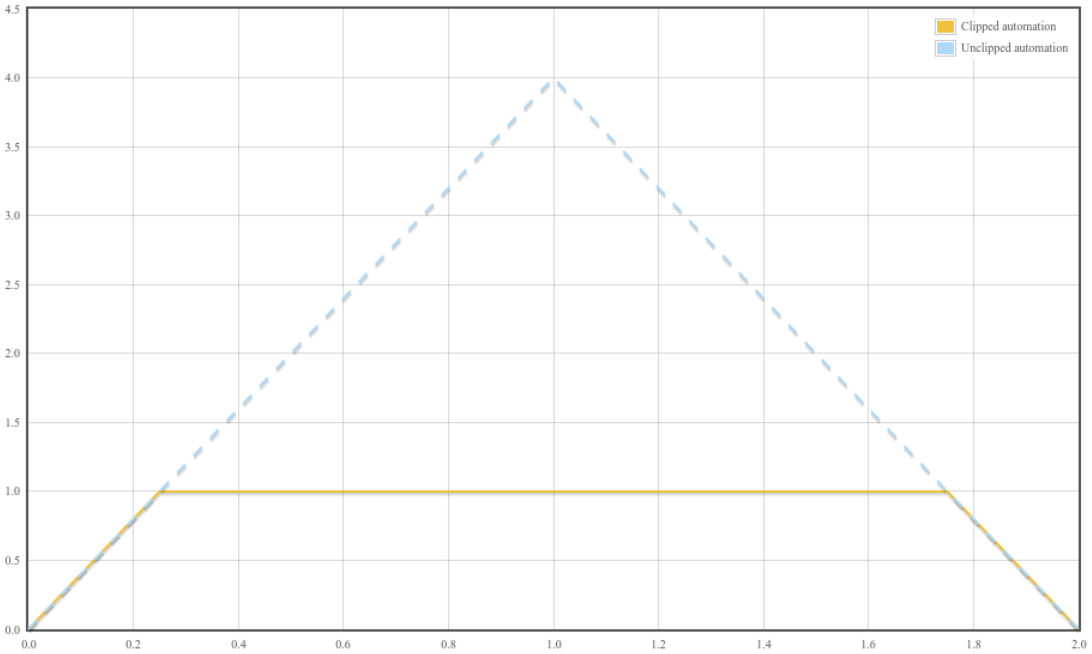


Figure 4 *An example of clipping of an AudioParam automation from the nominal range.*

Error preparing HTML-CSS output (preProcess)

§ 1.6.4. `AudioParam` Automation Example

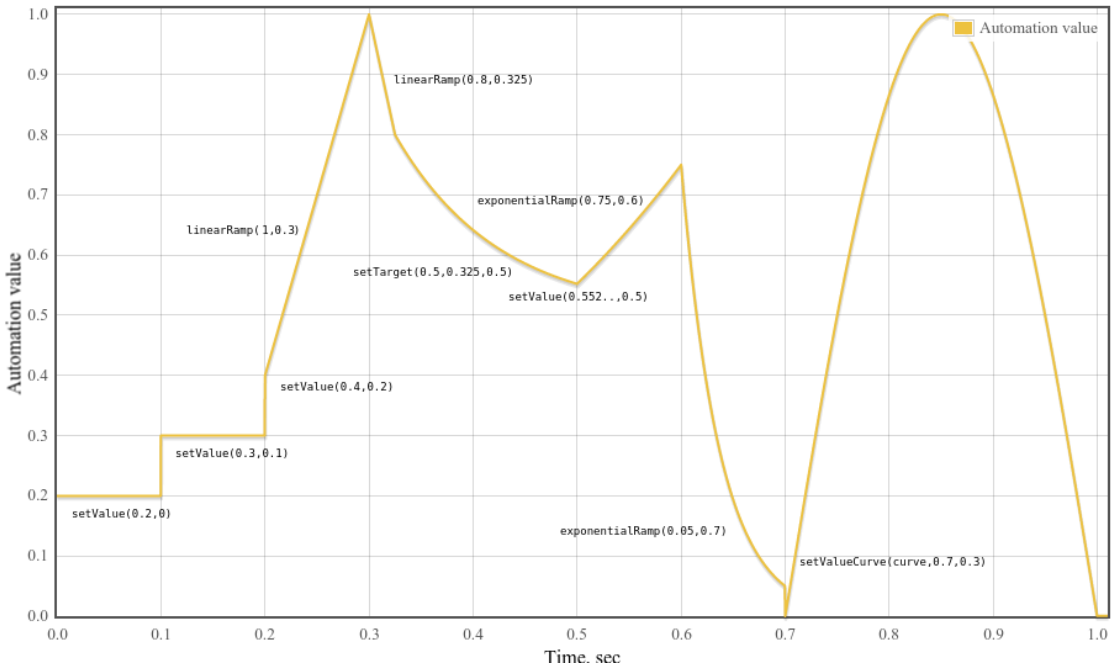


Figure 5 *An example of parameter automation.*

```
EXAMPLE 8
const curveLength = 44100;
const curve = new Float32Array(curveLength);
for (const i = 0; i < curveLength; ++i)
    curve[i] = Math.sin(Math.PI * i / curveLength);

const t0 = 0;
const t1 = 0.1;
const t2 = 0.2;
const t3 = 0.3;
const t4 = 0.325;
const t5 = 0.5;
const t6 = 0.6;
const t7 = 0.7;
const t8 = 1.0;
const timeConstant = 0.1;

param.setValueAtTime(0.2, t0);
param.setValueAtTime(0.3, t1);
param.setValueAtTime(0.4, t2);
param.linearRampToValueAtTime(1, t3);
param.linearRampToValueAtTime(0.8, t4);
param.setTargetAtTime(.5, t4, timeConstant);
// Compute where the setTargetAtTime will be at time t5 so we can make
// the following exponential start at the right point so there's no
// jump discontinuity. From the spec, we have
// v(t) = 0.5 + (0.8 - 0.5)*exp(-(t-t4)/timeConstant)
// Thus v(t5) = 0.5 + (0.8 - 0.5)*exp(-(t5-t4)/timeConstant)
param.setValueAtTime(0.5 + (0.8 - 0.5)*Math.exp(-(t5 - t4)/timeConstant), t5);
param.exponentialRampToValueAtTime(0.75, t6);
param.exponentialRampToValueAtTime(0.05, t7);
param.setValueCurveAtTime(curve, t7, t8 - t7);
```

§ 1.7. The `AudioScheduledSourceNode` Interface

Error preparing HTML-CSS output (preProcess)

The interface represents the common features of source nodes such as AudioBufferSourceNode, ConstantSourceNode, and OscillatorNode.

Before a source is started (by calling start(), the source node MUST output silence (0). After a source has been stopped (by calling stop()), the source MUST then output silence (0).

✓ MDN

AudioScheduledSourceNode cannot be instantiated directly, but is instead extended by the concrete interfaces for the source nodes.

An AudioScheduledSourceNode is said to be ***playing*** when its associated BaseAudioContext's currentTime is greater or equal to the time the AudioScheduledSourceNode is set to start, and less than the time it's set to stop.

AudioScheduledSourceNodes are created with an internal boolean slot ***[[source started]]***, initially set to false.

```
[Exposed=Window]
interface AudioScheduledSourceNode : AudioNode {
    attribute EventHandler onended;
    undefined start(optional double when = 0);
    undefined stop(optional double when = 0);
};
```

§ 1.7.1. Attributes

***onended*, of type EventHandler**

A property used to set an event handler for the ***ended*** event type that is dispatched to AudioScheduledSourceNode node types. When the source node has stopped playing (as determined by the concrete node), an event that uses the Event interface will be dispatched to the event handler.

For all AudioScheduledSourceNodes, the ended event is dispatched when the stop time determined by stop() is reached. For an AudioBufferSourceNode, the event is also dispatched because the duration has been reached or if the entire buffer has been played.

✓ MDN

§ 1.7.2. Methods

start(when)

Schedules a sound to playback at an exact time.

⏳ When this method is called, execute these steps:

1. ⏳ If this AudioScheduledSourceNode internal slot [[source started]] is true, an InvalidStateError exception MUST be thrown.
2. Check for any errors that must be thrown due to parameter constraints described below. If any exception is thrown during this step, abort those steps.
3. Set the internal slot [[source started]] on this AudioScheduledSourceNode to true.
4. Queue a control message to start the AudioScheduledSourceNode, including the parameter values in the message.
5. Send a control message to the associated AudioContext to start running its rendering thread only when all the following conditions are met:

    1. The context's [[control thread state]] is "suspended".
    2. The context is allowed to start.
    3. [[suspended by user]] flag is false.

NOTE: This allows start() to start an AudioContext that would otherwise not be allowed to start.

Arguments for the AudioScheduledSourceNode.start(when) method.

Error preparing HTML-CSS output (preProcess)

ype	Nullable	Optional	Description

Parameter	Type	Nullable	Optional	Description
when	double	✗	✓	The when parameter describes at what time (in seconds) the sound should start playing. It is in the same time coordinate system as the AudioContext's currentTime attribute. When the signal emitted by the AudioScheduledSourceNode depends on the sound's start time, the exact value of when is always used without rounding to the nearest sample frame. If 0 is passed in for this value or if the value is less than currentTime, then the sound will start playing immediately. ⌛ A RangeError exception MUST be thrown if when is negative.

Return type: undefined

stop(when)

Schedules a sound to stop playback at an exact time. If stop is called again after already having been called, the last invocation will be the only one applied; stop times set by previous calls will not be applied, unless the buffer has already stopped prior to any subsequent calls. If the buffer has already stopped, further calls to stop will have no effect. If a stop time is reached prior to the scheduled start time, the sound will not play.

⌛ When this method is called, execute these steps:

1. ⌛ If this AudioScheduledSourceNode internal slot [[source started]] is not true, an InvalidStateError exception MUST be thrown.
2. Check for any errors that must be thrown due to parameter constraints described below.
3. Queue a control message to stop the AudioScheduledSourceNode, including the parameter values in the message.

If the node is an AudioBufferSourceNode, running a control message to stop the AudioBufferSourceNode means invoking the handleStop() function in the playback algorithm.

Arguments for the AudioScheduledSourceNode.stop(when) method.

Parameter	Type	Nullable	Optional	Description
when	double	✗	✓	The when parameter describes at what time (in seconds) the source should stop playing. It is in the same time coordinate system as the AudioContext's currentTime attribute. If 0 is passed in for this value or if the value is less than currentTime, then the sound will stop playing immediately. ⌛ A RangeError exception MUST be thrown if when is negative.

Return type: undefined

§ 1.8. The AnalyserNode Interface

This interface represents a node which is able to provide real-time frequency and time-domain analysis information. The audio stream will be passed un-processed from input to output.

Property	Value	Notes
numberOfInputs	1	
numberOfOutputs	1	This output may be left unconnected.
channelCount	2	
channelCountMode	"max"	
erpretation	"speakers"	

Error preparing HTML-CSS output (preProcess)

Property	Value	Notes
tail-time	No	

```
[Exposed=Window]
interface AnalyserNode : AudioNode {
    constructor (BaseAudioContext context, optional AnalyserOptions options = {});
    undefined getFloatFrequencyData (Float32Array array);
    undefined getByteFrequencyData (Uint8Array array);
    undefined getFloatTimeDomainData (Float32Array array);
    undefined getByteTimeDomainData (Uint8Array array);
    attribute unsigned long fftSize;
    readonly attribute unsigned long frequencyBinCount;
    attribute double minDecibels;
    attribute double maxDecibels;
    attribute double smoothingTimeConstant;
};
```

§ 1.8.1. Constructors

AnalyserNode(context, options)

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the AnalyserNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `AnalyserNode` will be associated with.
options	`AnalyserOptions`	✗	✓	Optional initial parameter value for this `AnalyserNode`.

✓ MDN

§ 1.8.2. Attributes

***fftSize*, of type unsigned long**

The size of the FFT used for frequency-domain analysis (in sample-frames). ⏳ This MUST be a power of two in the range 32 to 32768, otherwise an `IndexSizeError` exception MUST be thrown. The default value is 2048. Note that large FFT sizes can be costly to compute.

If the `fftSize` is changed to a different value, then all state associated with smoothing of the frequency data (for `getByteFrequencyData()` and `getFloatFrequencyData()`) is reset. That is the previous block, $\hat{X}_{-1}[k]$, used for smoothing over time is set to 0 for all k.

Note that increasing `fftSize` does mean that the current time-domain data must be expanded to include past frames that it previously did not. This means that the `AnalyserNode` effectively MUST keep around the last 32768 sample-frames and the current time-domain data is the most recent `fftSize` sample-frames out of that.

***frequencyBinCount*, of type unsigned long, readonly**

Half the FFT size.

***maxDecibels*, of type double**

`maxDecibels` is the maximum power value in the scaling range for the FFT analysis data for conversion to unsigned byte values. The default value is -30. ⏳ If the value of this attribute is set to a value less than or equal to `minDecibels`, an `IndexSizeError` exception MUST be thrown.

***minDecibels*, of type double**

`minDecibels` is the minimum power value in the scaling range for the FFT analysis data for conversion to unsigned byte values. The default value is -100. ⏳ If the value of this attribute is set to a value more than or equal to

Error preparing HTML-CSS output (preProcess)

`ndexSizeError` exception MUST be thrown.

***smoothingTimeConstant*, of type double**

A value from 0 -> 1 where 0 represents no time averaging with the last analysis frame. The default value is 0.8. If the value of this attribute is set to a value less than 0 or more than 1, an `IndexSizeError` exception MUST be thrown.

§ 1.8.3. Methods

getByteFrequencyData(array)

PROPOSED CORRECTION ISSUE 2361-6. Use new Web IDL buffer primitives

Previous Change | Next Change

~~Get a reference to the bytes held by the `Uint8Array` passed as an argument. Copies the current frequency data to those bytes. If the array has fewer elements than the `frequencyBinCount`, the excess elements will be dropped. If the array has more elements than the `frequencyBinCount`~~ Write the current frequency data into *array*. If *array*'s byte length is less than `frequencyBinCount`, the excess elements will be dropped. If *array*'s byte length is greater than the `frequencyBinCount` , the excess elements will be ignored. The most recent `fftSize` frames are used in computing the frequency data.

Show Change | Show Current | Show Future

If another call to `getByteFrequencyData()` or `getFloatFrequencyData()` occurs within the same render quantum as a previous call, the current frequency data is not updated with the same data. Instead, the previously computed data is returned.

The values stored in the unsigned byte array are computed in the following way. Let $Y[k]$ be the current frequency data as described in FFT windowing and smoothing. Then the byte value, $b[k]$, is

$$b[k] = \left\lfloor \frac{255}{\mathrm{dB}_{max} - \mathrm{dB}_{min}} \left(Y[k] - \mathrm{dB}_{min} \right) \right\rfloor$$

where dB_{min} is `minDecibels` and dB_{max} is `maxDecibels`. If $b[k]$ lies outside the range of 0 to 255, $b[k]$ is clipped to lie in that range.

Arguments for the AnalyserNode.getByteFrequencyData() method.

Parameter	Type	Nullable	Optional	Description
array	`Uint8Array`	✗	✗	This parameter is where the frequency-domain analysis data will be copied.

✓ MDN

Return type: `undefined`

getByteTimeDomainData(array)

PROPOSED CORRECTION ISSUE 2361-7. Use new Web IDL buffer primitives

Previous Change | Next Change

~~Get a reference to the bytes held by the `Uint8Array` passed as an argument. Copies the current time-domain data (waveform data) into those bytes. If the array has fewer elements than the value of `fftSize`, the excess elements will be dropped. If the array has more elements than `fftSize`.~~ Write the current time-domain data (waveform data) into *array*. If *array*'s byte length is less than `fftSize`, the excess elements will be dropped. If *array*'s byte length is greater than the `fftSize`, the excess elements will be ignored. The most recent `fftSize` frames are used in computing the byte data.

Show Change | Show Current | Show Future

The values stored in the unsigned byte array are computed in the following way. Let $x[k]$ be the time-domain data. Then the byte value, $b[k]$, is

$$b[k] = \lfloor 128(1 + x[k]) \rfloor.$$

✓ MDN

Error preparing HTML-CSS output (preProcess)

If *b*[*k*] lies outside the range 0 to 255, *b*[*k*] is clipped to lie in that range.

Arguments for the AnalyserNode.getByteTimeDomainData() method.

Parameter	Type	Nullable	Optional	Description
array	Uint8Array	✗	✗	This parameter is where the time-domain sample data will be copied.

Return type: undefined

getFloatFrequencyData(array)

PROPOSED CORRECTION ISSUE 2361-8. Use new Web IDL buffer primitives

Previous Change | Next Change

~~Get a reference to the bytes held by the Float32Array passed as an argument. Copies the current frequency data into those bytes. If the array has fewer elements than the frequencyBinCount,~~ Write the current frequency data into *array*. If *array* has fewer elements than the `frequencyBinCount`, the excess elements will be dropped. If *array* has more elements than the `frequencyBinCount`, the excess elements will be ignored. The most recent `fftSize` frames are used in computing the frequency data.

Show Change | Show Current | Show Future

If another call to `getFloatFrequencyData()` or `getByteFrequencyData()` occurs within the same render quantum as a previous call, the current frequency data is not updated with the same data. Instead, the previously computed data is returned.

The frequency data are in dB units.

Arguments for the AnalyserNode.getFloatFrequencyData() method.

Parameter	Type	Nullable	Optional	Description
array	Float32Array	✗	✗	This parameter is where the frequency-domain analysis data will be copied.

Return type: undefined

getFloatTimeDomainData(array)

PROPOSED CORRECTION ISSUE 2361-9. Use new Web IDL buffer primitives

Previous Change

~~Get a reference to the bytes held by the Float32Array passed as an argument. Copies the current time-domain data (waveform data) into those bytes. If the array has fewer elements than the value of fftSize, the excess elements will be dropped. If the array has more elements than fftSize,~~ Write the current time-domain data (waveform data) into *array*. If *array* has fewer elements than the value of `fftSize`, the excess elements will be dropped. If *array* has more elements than the value of `fftSize`, the excess elements will be ignored. The most recent `fftSize` frames are written (after downmixing).

Show Change | Show Current | Show Future

Arguments for the AnalyserNode.getFloatTimeDomainData() method.

Parameter	Type	Nullable	Optional	Description
array	Float32Array	✗	✗	This parameter is where the time-domain sample data will be copied.

Return type: undefined

§ 1.8.4. AnalyserOptions

Error preparing HTML-CSS output (preProcess)

This specifies the options to be used when constructing an `AnalyserNode`. All members are optional; if not specified, the normal default values are used to construct the node.

```
dictionary AnalyserOptions : AudioNodeOptions {
    unsigned long fftSize = 2048;
    double maxDecibels = -30;
    double minDecibels = -100;
    double smoothingTimeConstant = 0.8;
};
```

1.8.4.1. Dictionary `AnalyserOptions` Members

***fftSize*, of type unsigned long, defaulting to `2048`**
The desired initial size of the FFT for frequency-domain analysis.

***maxDecibels*, of type double, defaulting to `-30`**
The desired initial maximum power in dB for FFT analysis.

***minDecibels*, of type double, defaulting to `-100`**
The desired initial minimum power in dB for FFT analysis.

***smoothingTimeConstant*, of type double, defaulting to `0.8`**
The desired initial smoothing constant for the FFT analysis.

1.8.5. Time-Domain Down-Mixing

When the ***current time-domain data*** are computed, the input signal must be down-mixed to mono as if `channelCount` is 1, `channelCountMode` is "`max`" and `channelInterpretation` is "`speakers`". This is independent of the settings for the `AnalyserNode` itself. The most recent `fftSize` frames are used for the down-mixing operation.

1.8.6. FFT Windowing and Smoothing over Time

When the ***current frequency data*** are computed, the following operations are to be performed:

1. Compute the current time-domain data.
2. Apply a Blackman window to the time domain input data.
3. Apply a Fourier transform to the windowed time domain input data to get real and imaginary frequency data.
4. Smooth over time the frequency domain data.
5. Convert to dB.

In the following, let N be the value of the `fftSize` attribute of this `AnalyserNode`.

Applying a Blackman window consists in the following operation on the input time domain data. Let $x[n]$ for $n = 0, \ldots, N-1$ be the time domain data. The Blackman window is defined by

$$\alpha = 0.16$$
$$a_0 = \frac{1-\alpha}{2}$$
$$a_1 = \frac{1}{2}$$
$$a_2 = \frac{\alpha}{2}$$
$$w[n] = a_0 - a_1 \cos\frac{2\pi n}{N} + a_2 \cos\frac{4\pi n}{N}, \text{ for } n = 0, \ldots, N-1$$

Error preparing HTML-CSS output (preProcess)

The windowed signal $\hat{x}[n]$ is

$$\hat{x}[n] = x[n]w[n], \text{ for } n = 0, \ldots, N-1$$

Applying a Fourier transform consists of computing the Fourier transform in the following way. Let $X[k]$ be the complex frequency domain data and $\hat{x}[n]$ be the windowed time domain data computed above. Then

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} \hat{x}[n]\, W_N^{-kn}$$

for $k = 0, \ldots, N/2 - 1$ where $W_N = e^{2\pi i / N}$.

Smoothing over time frequency data consists in the following operation:

- Let $\hat{X}_{-1}[k]$ be the result of this operation on the previous block. The ***previous block*** is defined as being the buffer computed by the previous smoothing over time operation, or an array of N zeros if this is the first time we are smoothing over time.
- Let τ be the value of the `smoothingTimeConstant` attribute for this `AnalyserNode`.
- Let $X[k]$ be the result of applying a Fourier transform of the current block.

Then the smoothed value, $\hat{X}[k]$, is computed by

$$\hat{X}[k] = \tau \hat{X}_{-1}[k] + (1 - \tau)\,|X[k]|$$

for $k = 0, \ldots, N-1$.

Conversion to dB consists of the following operation, where $\hat{X}[k]$ is computed in smoothing over time:

$$Y[k] = 20\log_{10}\hat{X}[k]$$

for $k = 0, \ldots, N-1$.

This array, $Y[k]$, is copied to the output array for `getFloatFrequencyData()`. For `getByteFrequencyData()`, the $Y[k]$ is clipped to lie between `minDecibels` and `maxDecibels` and then scaled to fit in an unsigned byte such that `minDecibels` is represented by the value 0 and `maxDecibels` is represented by the value 255.

§ 1.9. The `AudioBufferSourceNode` Interface

This interface represents an audio source from an in-memory audio asset in an `AudioBuffer`. It is useful for playing audio assets which require a high degree of scheduling flexibility and accuracy. If sample-accurate playback of network- or disk-backed assets is required, an implementer should use `AudioWorkletNode` to implement playback.

The `start()` method is used to schedule when sound playback will happen. The `start()` method may not be issued multiple times. The playback will stop automatically when the buffer's audio data has been completely played (if the `loop` attribute is `false`), or when the `stop()` method has been called and the specified time has been reached. Please see more details in the `start()` and `stop()` descriptions.

Property	Value	Notes
`numberOfInputs`	0	
`numberOfOutputs`	1	
`channelCount`	2	
`ntMode`	"`max`"	

Property	Value	Notes
channelInterpretation	"speakers"	
tail-time	No	

The number of channels of the output equals the number of channels of the AudioBuffer assigned to the buffer attribute, or is one channel of silence if buffer is null.

In addition, if the buffer has more than one channel, then the AudioBufferSourceNode output must change to a single channel of silence at the beginning of a render quantum after the time at which any one of the following conditions holds:

- the end of the buffer has been reached;
- the duration has been reached;
- the stop time has been reached.

A ***playhead position*** for an AudioBufferSourceNode is defined as any quantity representing a time offset in seconds, relative to the time coordinate of the first sample frame in the buffer. Such values are to be considered independently from the node's playbackRate and detune parameters. In general, playhead positions may be subsample-accurate and need not refer to exact sample frame positions. They may assume valid values between 0 and the duration of the buffer.

The playbackRate and detune attributes form a compound parameter. They are used together to determine a ***computedPlaybackRate*** value:

```
computedPlaybackRate(t) = playbackRate(t) * pow(2, detune(t) / 1200)
```

The nominal range for this compound parameter is $(-\infty, \infty)$.

AudioBufferSourceNodes are created with an internal boolean slot ***[[buffer set]]***, initially set to false.

```
[Exposed=Window]
interface AudioBufferSourceNode : AudioScheduledSourceNode {
    constructor (BaseAudioContext context,
                 optional AudioBufferSourceOptions options = {});
    attribute AudioBuffer? buffer;
    readonly attribute AudioParam playbackRate;
    readonly attribute AudioParam detune;
    attribute boolean loop;
    attribute double loopStart;
    attribute double loopEnd;
    undefined start (optional double when = 0,
                     optional double offset,
                     optional double duration);
};
```

§ 1.9.1. Constructors

AudioBufferSourceNode(context, options)

When the constructor is called with a BaseAudioContext *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the AudioBufferSourceNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	BaseAudioContext	✗	✗	The BaseAudioContext this new AudioBufferSourceNode will be associated with.
options	AudioBufferSourceOptions	✗	✓	Optional initial parameter value for this

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
				AudioBufferSourceNode.

§ 1.9.2. Attributes

***buffer*, of type AudioBuffer, nullable**
Represents the audio asset to be played.

To set the `buffer` attribute, execute these steps:

1. Let *new buffer* be the `AudioBuffer` or `null` value to be assigned to `buffer`.
2. If *new buffer* is not `null` and `[[buffer set]]` is true, throw an `InvalidStateError` and abort these steps.
3. If *new buffer* is not `null`, set `[[buffer set]]` to true.
4. Assign *new buffer* to the `buffer` attribute.
5. If `start()` has previously been called on this node, perform the operation acquire the content on `buffer`.

***detune*, of type AudioParam, readonly**
An additional parameter, in cents, to modulate the speed at which is rendered the audio stream. This parameter is a compound parameter with `playbackRate` to form a computedPlaybackRate.

Parameter	Value	Notes
`defaultValue`	0	
`minValue`	most-negative-single-float	Approximately -3.4028235e38
`maxValue`	most-positive-single-float	Approximately 3.4028235e38
`automationRate`	"`k-rate`"	Has automation rate constraints

***loop*, of type boolean**
Indicates if the region of audio data designated by `loopStart` and `loopEnd` should be played continuously in a loop. The default value is `false`.

***loopEnd*, of type double**
An optional playhead position where looping should end if the `loop` attribute is true. Its value is exclusive of the content of the loop. Its default `value` is 0, and it may usefully be set to any value between 0 and the duration of the buffer. If `loopEnd` is less than or equal to 0, or if `loopEnd` is greater than the duration of the buffer, looping will end at the end of the buffer.

***loopStart*, of type double**
An optional playhead position where looping should begin if the `loop` attribute is true. Its default `value` is 0, and it may usefully be set to any value between 0 and the duration of the buffer. If `loopStart` is less than 0, looping will begin at 0. If `loopStart` is greater than the duration of the buffer, looping will begin at the end of the buffer.

***playbackRate*, of type AudioParam, readonly**
The speed at which to render the audio stream. This is a compound parameter with `detune` to form a computedPlaybackRate.

Parameter	Value	Notes
`defaultValue`	1	
`minValue`	most-negative-single-float	Approximately -3.4028235e38
`maxValue`	most-positive-single-float	Approximately 3.4028235e38
`automationRate`	"`k-rate`"	Has automation rate constraints

§ 1.9.3. Methods

start(when, offset, duration)
Error preparing HTML-CSS output (preProcess) playback at an exact time.

⌛ When this method is called, execute these steps:

1. ⌛ If this `AudioBufferSourceNode` internal slot `[[source started]]` is `true`, an `InvalidStateError` exception MUST be thrown.
2. Check for any errors that must be thrown due to parameter constraints described below. If any exception is thrown during this step, abort those steps.
3. Set the internal slot `[[source started]]` on this `AudioBufferSourceNode` to `true`.
4. Queue a control message to start the `AudioBufferSourceNode`, including the parameter values in the message.
5. Acquire the contents of the `buffer` if the `buffer` has been set.
6. Send a control message to the associated `AudioContext` to start running its rendering thread only when all the following conditions are met:

   1. The context's `[[control thread state]]` is `suspended`.
   2. The context is allowed to start.
   3. `[[suspended by user]]` flag is `false`.

> NOTE: This allows `start()` to start an `AudioContext` that would otherwise not be allowed to start.

Running a control message to start the `AudioBufferSourceNode` means invoking the `handleStart()` function in the playback algorithm which follows.

Arguments for the AudioBufferSourceNode.start(when, offset, duration) method.

Parameter	Type	Nullable	Optional	Description
when	`double`	✗	✓	The `when` parameter describes at what time (in seconds) the sound should start playing. It is in the same time coordinate system as the `AudioContext`'s `currentTime` attribute. If 0 is passed in for this value or if the value is less than **currentTime**, then the sound will start playing immediately. ⌛ A `RangeError` exception MUST be thrown if `when` is negative.
offset	`double`	✗	✓	The `offset` parameter supplies a playhead position where playback will begin. If 0 is passed in for this value, then playback will start from the beginning of the buffer. ⌛ A `RangeError` exception MUST be thrown if `offset` is negative. If `offset` is greater than `loopEnd`, `playbackRate` is positive or zero, and `loop` is `true`, playback will begin at `loopEnd`. If `offset` is greater than `loopStart`, `playbackRate` is negative, and `loop` is `true`, playback will begin at `loopStart`. `offset` is silently clamped to [0, `duration`], when `startTime` is reached, where `duration` is the value of the `duration` attribute of the `AudioBuffer` set to the `buffer` attribute of this `AudioBufferSourceNode`.
duration	`double`	✗	✓	The `duration` parameter describes the duration of sound to be played, expressed as seconds of total buffer content to be output, including any whole or partial loop iterations. The units of `duration` are independent of the effects of `playbackRate`. For example, a `duration` of 5 seconds with a playback rate of 0.5 will output 5 seconds of buffer content at half speed, producing 10 seconds of audible output. ⌛ A `RangeError` exception MUST be thrown if `duration` is negative.

Error preparing HTML-CSS output (preProcess)

Return type: `undefined`

1.9.4. `AudioBufferSourceOptions`

This specifies options for constructing a `AudioBufferSourceNode`. All members are optional; if not specified, the normal default is used in constructing the node.

```
dictionary AudioBufferSourceOptions {
    AudioBuffer? buffer;
    float detune = 0;
    boolean loop = false;
    double loopEnd = 0;
    double loopStart = 0;
    float playbackRate = 1;
};
```

1.9.4.1. Dictionary `AudioBufferSourceOptions` Members

***`buffer`*, of type AudioBuffer, nullable**
: The audio asset to be played. This is equivalent to assigning `buffer` to the `buffer` attribute of the `AudioBufferSourceNode`.

***`detune`*, of type float, defaulting to `0`**
: The initial value for the `detune` AudioParam.

***`loop`*, of type boolean, defaulting to `false`**
: The initial value for the `loop` attribute.

***`loopEnd`*, of type double, defaulting to `0`**
: The initial value for the `loopEnd` attribute.

***`loopStart`*, of type double, defaulting to `0`**
: The initial value for the `loopStart` attribute.

***`playbackRate`*, of type float, defaulting to `1`**
: The initial value for the `playbackRate` AudioParam.

1.9.5. Looping

This section is non-normative. Please see the playback algorithm for normative requirements.

Setting the `loop` attribute to true causes playback of the region of the buffer defined by the endpoints `loopStart` and `loopEnd` to continue indefinitely, once any part of the looped region has been played. While `loop` remains true, looped playback will continue until one of the following occurs:

- `stop()` is called,
- the scheduled stop time has been reached,
- the `duration` has been exceeded, if `start()` was called with a `duration` value.

The body of the loop is considered to occupy a region from `loopStart` up to, but not including, `loopEnd`. The direction of playback of the looped region respects the sign of the node's playback rate. For positive playback rates, looping occurs from `loopStart` to `loopEnd`; for negative rates, looping occurs from `loopEnd` to `loopStart`.

Looping does not affect the interpretation of the `offset` argument of `start()`. Playback always starts at the requested offset, and looping only begins once the body of the loop is encountered during playback.

The effective loop start and end points are required to lie within the range of zero and the buffer duration, as specified in the algorithm below. `loopEnd` is further constrained to be at or after `loopStart`. If any of these constraints are violated, the loop is considered to include the entire buffer contents.

Loop endpoints have subsample accuracy. When endpoints do not fall on exact sample frame offsets, or when the playback rate is not equal to 1, playback of the loop is interpolated to splice the beginning and end of the loop together just as if the looped audio occurred in sequential, non-looped regions of the buffer.

Loop-related properties may be varied during playback of the buffer, and in general take effect on the next rendering
Error preparing HTML-CSS output (preProcess)
ts are defined by the normative playback algorithm which follows.

The default values of the `loopStart` and `loopEnd` attributes are both 0. Since a `loopEnd` value of zero is equivalent to the length of the buffer, the default endpoints cause the entire buffer to be included in the loop.

Note that the values of the loop endpoints are expressed as time offsets in terms of the sample rate of the buffer, meaning that these values are independent of the node's `playbackRate` parameter which can vary dynamically during the course of playback.

§ 1.9.6. Playback of AudioBuffer Contents

This normative section specifies the playback of the contents of the buffer, accounting for the fact that playback is influenced by the following factors working in combination:

- A starting offset, which can be expressed with sub-sample precision.
- Loop points, which can be expressed with sub-sample precision and can vary dynamically during playback.
- Playback rate and detuning parameters, which combine to yield a single computedPlaybackRate that can assume finite values which may be positive or negative.

The algorithm to be followed internally to generate output from an `AudioBufferSourceNode` conforms to the following principles:

- Resampling of the buffer may be performed arbitrarily by the UA at any desired point to increase the efficiency or quality of the output.
- Sub-sample start offsets or loop points may require additional interpolation between sample frames.
- The playback of a looped buffer should behave identically to an unlooped buffer containing consecutive occurrences of the looped audio content, excluding any effects from interpolation.

The description of the algorithm is as follows:

```
let buffer; // AudioBuffer employed by this node
let context; // AudioContext employed by this node

// The following variables capture attribute and AudioParam values for the node.
// They are updated on a k-rate basis, prior to each invocation of process().
let loop;
let detune;
let loopStart;
let loopEnd;
let playbackRate;

// Variables for the node's playback parameters
let start = 0, offset = 0, duration = Infinity; // Set by start()
let stop = Infinity; // Set by stop()


// Variables for tracking node's playback state
let bufferTime = 0, started = false, enteredLoop = false;
let bufferTimeElapsed = 0;
let dt = 1 / context.sampleRate;

// Handle invocation of start method call
function handleStart(when, pos, dur) {
    if (arguments.length >= 1) {
        start = when;
    }
    offset = pos;
    if (arguments.length >= 3) {
        duration = dur;
    }
}

// Handle invocation of stop method call
function handleStop(when) {
```

Error preparing HTML-CSS output (preProcess)

```
    if (arguments.length >= 1) {
        stop = when;
    } else {
        stop = context.currentTime;
    }
}

// Interpolate a multi-channel signal value for some sample frame.
// Returns an array of signal values.
function playbackSignal(position) {
    /*
        This function provides the playback signal function for buffer, which is a
        function that maps from a playhead position to a set of output signal
        values, one for each output channel. If |position| corresponds to the
        location of an exact sample frame in the buffer, this function returns
        that frame. Otherwise, its return value is determined by a UA-supplied
        algorithm that interpolates sample frames in the neighborhood of
        |position|.

        If |position| is greater than or equal to |loopEnd| and there is no subsequent
        sample frame in buffer, then interpolation should be based on the sequence
        of subsequent frames beginning at |loopStart|.
     */
     ...
}

// Generate a single render quantum of audio to be placed
// in the channel arrays defined by output. Returns an array
// of |numberOfFrames| sample frames to be output.
function process(numberOfFrames) {
    let currentTime = context.currentTime; // context time of next rendered frame
    const output = []; // accumulates rendered sample frames

    // Combine the two k-rate parameters affecting playback rate
    const computedPlaybackRate = playbackRate * Math.pow(2, detune / 1200);

    // Determine loop endpoints as applicable
    let actualLoopStart, actualLoopEnd;
    if (loop && buffer != null) {
        if (loopStart >= 0 && loopEnd > 0 && loopStart < loopEnd) {
            actualLoopStart = loopStart;
            actualLoopEnd = Math.min(loopEnd, buffer.duration);
        } else {
            actualLoopStart = 0;
            actualLoopEnd = buffer.duration;
        }
    } else {
        // If the loop flag is false, remove any record of the loop having been entered
        enteredLoop = false;
    }

    // Handle null buffer case
    if (buffer == null) {
        stop = currentTime; // force zero output for all time
    }

    // Render each sample frame in the quantum
    for (let index = 0; index < numberOfFrames; index++) {
        // Check that currentTime and bufferTimeElapsed are
        // within allowable range for playback
        if (currentTime < start || currentTime >= stop || bufferTimeElapsed >= duration) {
            output.push(0); // this sample frame is silent
            currentTime += dt;
            continue;
```

Error preparing HTML-CSS output (preProcess)

```
        if (!started) {
            // Take note that buffer has started playing and get initial
            // playhead position.
            if (loop && computedPlaybackRate >= 0 && offset >= actualLoopEnd) {
                offset = actualLoopEnd;
            }
            if (computedPlaybackRate < 0 && loop && offset < actualLoopStart) {
                offset = actualLoopStart;
            }
            bufferTime = offset;
            started = true;
        }

        // Handle loop-related calculations
        if (loop) {
            // Determine if looped portion has been entered for the first time
            if (!enteredLoop) {
                if (offset < actualLoopEnd && bufferTime >= actualLoopStart) {
                    // playback began before or within loop, and playhead is
                    // now past loop start
                    enteredLoop = true;
                }
                if (offset >= actualLoopEnd && bufferTime < actualLoopEnd) {
                    // playback began after loop, and playhead is now prior
                    // to the loop end
                    enteredLoop = true;
                }
            }

            // Wrap loop iterations as needed. Note that enteredLoop
            // may become true inside the preceding conditional.
            if (enteredLoop) {
                while (bufferTime >= actualLoopEnd) {
                    bufferTime -= actualLoopEnd - actualLoopStart;
                }
                while (bufferTime < actualLoopStart) {
                    bufferTime += actualLoopEnd - actualLoopStart;
                }
            }
        }

        if (bufferTime >= 0 && bufferTime < buffer.duration) {
            output.push(playbackSignal(bufferTime));
        } else {
            output.push(0); // past end of buffer, so output silent frame
        }

        bufferTime += dt * computedPlaybackRate;
        bufferTimeElapsed += dt * computedPlaybackRate;
        currentTime += dt;
    } // End of render quantum loop

    if (currentTime >= stop) {
        // End playback state of this node.  No further invocations of process()
        // will occur.  Schedule a change to set the number of output channels to 1.
    }

    return output;
}
```

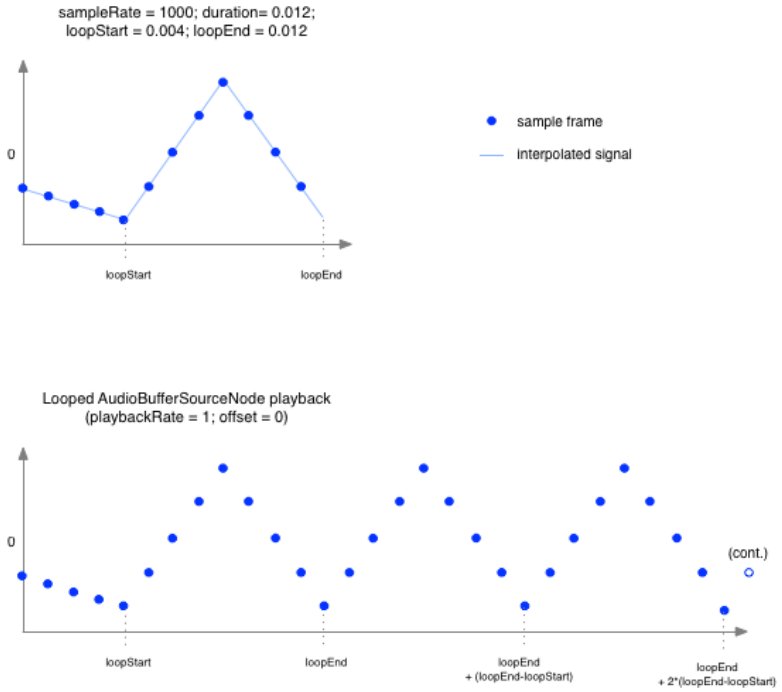
The following non-normative figures illustrate the behavior of the algorithm in assorted key scenarios. Dynamic resampling of the buffer is not considered, but as long as the times of loop positions are not changed this does not materially affect the resulting playback. In all figures, the following conventions apply:

Error preparing HTML-CSS output (preProcess)

- context sample rate is 1000 Hz
- `AudioBuffer` content is shown with the first sample frame at the *x* origin.
- output signals are shown with the sample frame located at time `start` at the *x* origin.
- linear interpolation is depicted throughout, although a UA could employ other interpolation techniques.
- the `duration` values noted in the figures refer to the `buffer`, not arguments to `start()`

This figure illustrates basic playback of a buffer, with a simple loop that ends after the last sample frame in the buffer:

MDN

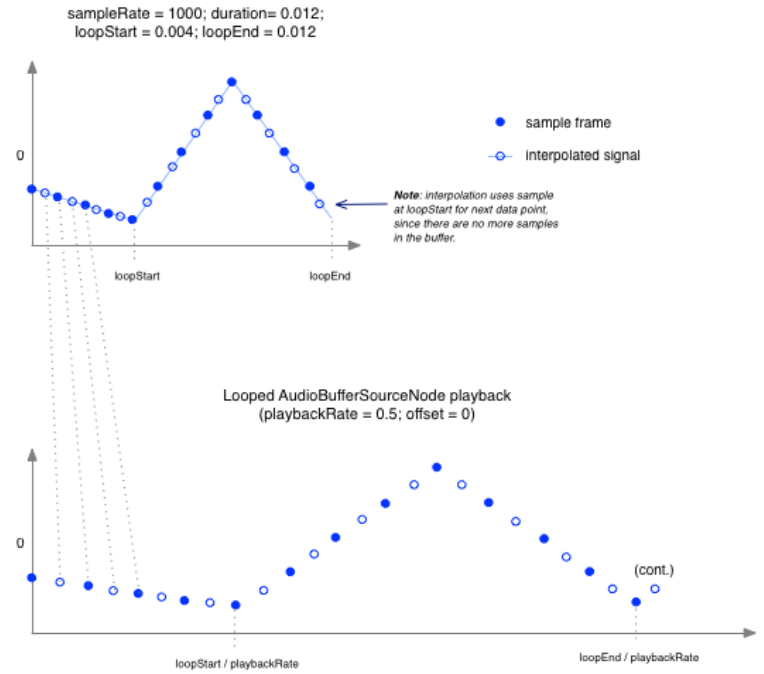


MDN

Figure 6 *`AudioBufferSourceNode` basic playback*

MDN

This figure illustrates `playbackRate` interpolation, showing half-speed playback of buffer contents in which every other output sample frame is interpolated. Of particular note is the last sample frame in the looped output, which is interpolated using the loop start point:



MDN

MDN

MDN

Figure 7 *`AudioBufferSourceNode` playbackRate interpolation*

Error preparing HTML-CSS output (preProcess)

This figure illustrates sample rate interpolation, showing playback of a buffer whose sample rate is 50% of the context sample rate, resulting in a computed playback rate of 0.5 that corrects for the difference in sample rate between the buffer and the context. The resulting output is the same as the preceding example, but for different reasons.

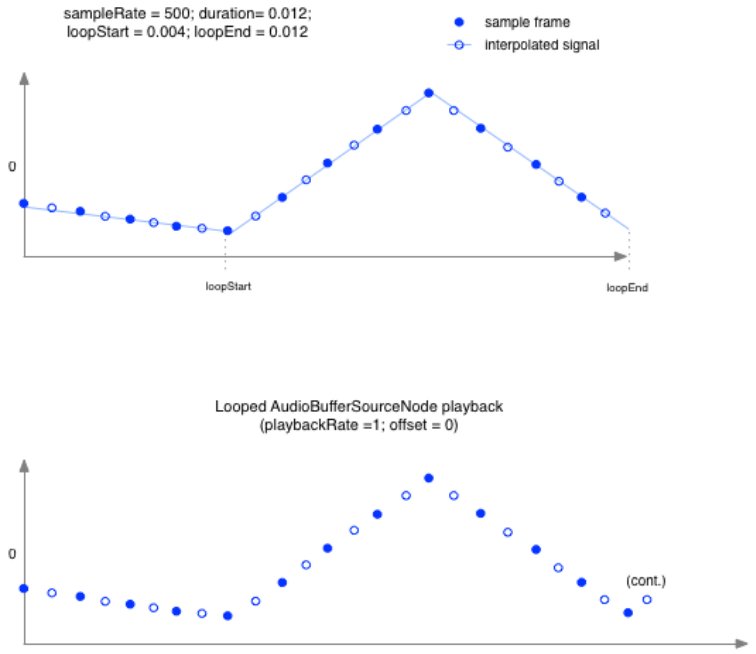


Figure 8 *`AudioBufferSourceNode` sample rate interpolation.*

This figure illustrates subsample offset playback, in which the offset within the buffer begins at exactly half a sample frame. Consequently, every output frame is interpolated:

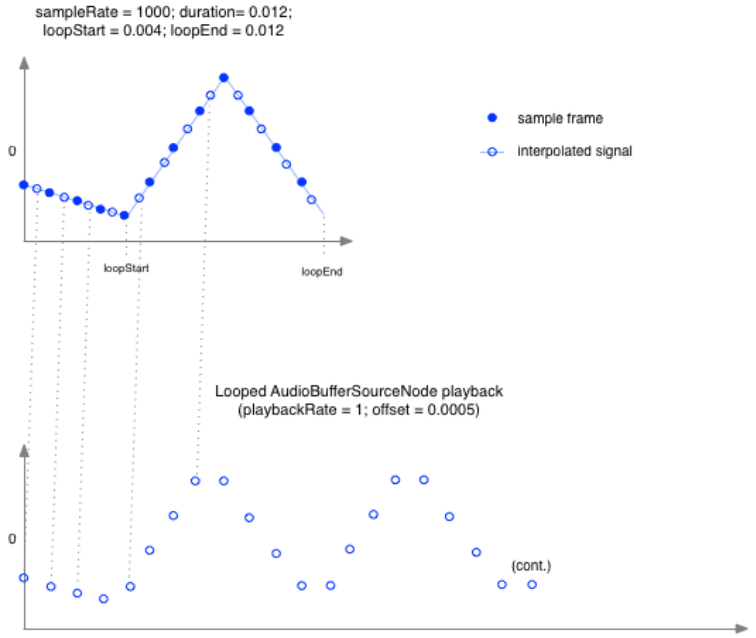


Figure 9 *`AudioBufferSourceNode` subsample offset playback*

This figure illustrates subsample loop playback, showing how fractional frame offsets in the loop endpoints map to interpolated data points in the buffer that respect these offsets as if they were references to exact sample frames:

Error preparing HTML-CSS output (preProcess)

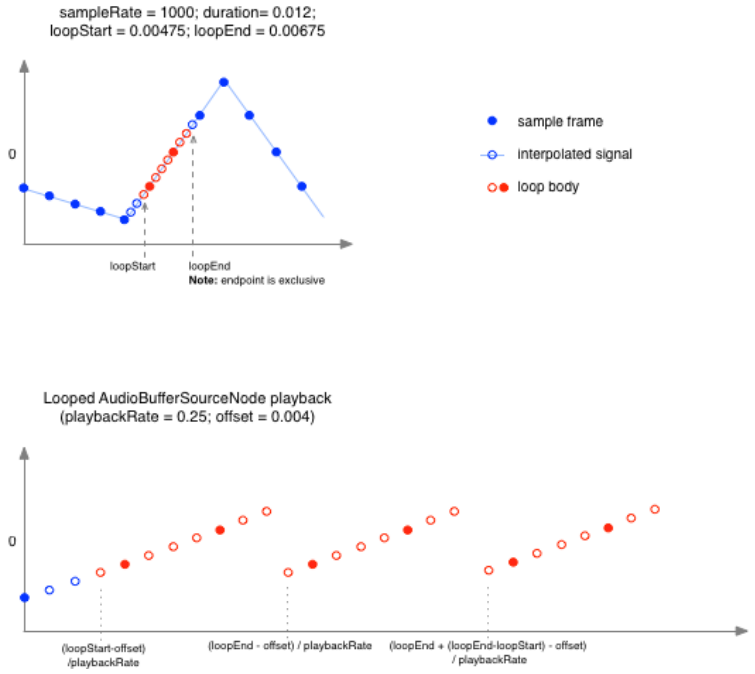


Figure 10 *`AudioBufferSourceNode` subsample loop playback*

§ 1.10. The `AudioDestinationNode` Interface

This is an `AudioNode` representing the final audio destination and is what the user will ultimately hear. It can often be considered as an audio output device which is connected to speakers. All rendered audio to be heard will be routed to this node, a "terminal" node in the `AudioContext`'s routing graph. There is only a single AudioDestinationNode per `AudioContext`, provided through the `destination` attribute of `AudioContext`.

The output of a `AudioDestinationNode` is produced by summing its input, allowing to capture the output of an `AudioContext` into, for example, a `MediaStreamAudioDestinationNode`, or a `MediaRecorder` (described in [mediastream-recording]).

The `AudioDestinationNode` can be either the destination of an `AudioContext` or `OfflineAudioContext`, and the channel properties depend on what the context is.

For an `AudioContext`, the defaults are

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	1	
`channelCount`	2	
`channelCountMode`	"`explicit`"	
`channelInterpretation`	"`speakers`"	
tail-time	No	

The `channelCount` can be set to any value less than or equal to `maxChannelCount`. ⌛ An `IndexSizeError` exception MUST be thrown if this value is not within the valid range. Giving a concrete example, if the audio hardware supports 8-channel output, then we may set `channelCount` to 8, and render 8 channels of output.

For an `OfflineAudioContext`, the defaults are

Property	Value	Notes
`numberOfInputs`	1	
`tputs`	1	

Error preparing HTML-CSS output (preProcess)

Property	Value	Notes
channelCount	numberOfChannels	
channelCountMode	"explicit"	
channelInterpretation	"speakers"	
tail-time	No	

where `numberOfChannels` is the number of channels specified when constructing the `OfflineAudioContext`. This value may not be changed; ⌛ a `NotSupportedError` exception MUST be thrown if `channelCount` is changed to a different value.

```
[Exposed=Window]
interface AudioDestinationNode : AudioNode {
    readonly attribute unsigned long maxChannelCount;
};
```

§ 1.10.1. Attributes

***maxChannelCount*, of type unsigned long, readonly**

The maximum number of channels that the `channelCount` attribute can be set to. An `AudioDestinationNode` representing the audio hardware end-point (the normal case) can potentially output more than 2 channels of audio if the audio hardware is multi-channel. `maxChannelCount` is the maximum number of channels that this hardware is capable of supporting.

§ 1.11. The `AudioListener` Interface

This interface represents the position and orientation of the person listening to the audio scene. All `PannerNode` objects spatialize in relation to the `BaseAudioContext`'s `listener`. See § 6 Spatialization/Panning for more details about spatialization.

The `positionX`, `positionY`, and `positionZ` parameters represent the location of the listener in 3D Cartesian coordinate space. `PannerNode` objects use this position relative to individual audio sources for spatialization.

The `forwardX`, `forwardY`, and `forwardZ` parameters represent a direction vector in 3D space. Both a `forward` vector and an `up` vector are used to determine the orientation of the listener. In simple human terms, the `forward` vector represents which direction the person's nose is pointing. The `up` vector represents the direction the top of a person's head is pointing. These two vectors are expected to be linearly independent. For normative requirements of how these values are to be interpreted, see the § 6 Spatialization/Panning section.

```
[Exposed=Window]
interface AudioListener {
    readonly attribute AudioParam positionX;
    readonly attribute AudioParam positionY;
    readonly attribute AudioParam positionZ;
    readonly attribute AudioParam forwardX;
    readonly attribute AudioParam forwardY;
    readonly attribute AudioParam forwardZ;
    readonly attribute AudioParam upX;
    readonly attribute AudioParam upY;
    readonly attribute AudioParam upZ;
    undefined setPosition (float x, float y, float z);
    undefined setOrientation (float x, float y, float z, float xUp, float yUp, float zUp);
};
```

✓ MDN

§ 1.11.1. Attributes

Error preparing HTML-CSS output (preProcess)

oParam, readonly

Sets the x coordinate component of the forward direction the listener is pointing in 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

***forwardY*, of type AudioParam, readonly**

Sets the y coordinate component of the forward direction the listener is pointing in 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

***forwardZ*, of type AudioParam, readonly**

Sets the z coordinate component of the forward direction the listener is pointing in 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	-1	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

***positionX*, of type AudioParam, readonly**

Sets the x coordinate position of the audio listener in a 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

***positionY*, of type AudioParam, readonly**

Sets the y coordinate position of the audio listener in a 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

***positionZ*, of type AudioParam, readonly**

Sets the z coordinate position of the audio listener in a 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38

Parameter	Value	Notes
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

***upX*, of type AudioParam, readonly**
Sets the x coordinate component of the up direction the listener is pointing in 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

***upY*, of type AudioParam, readonly**
Sets the y coordinate component of the up direction the listener is pointing in 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	1	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

***upZ*, of type AudioParam, readonly**
Sets the z coordinate component of the up direction the listener is pointing in 3D Cartesian coordinate space.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	

§ 1.11.2. Methods

setOrientation(x, y, z, xUp, yUp, zUp)
This method is DEPRECATED. It is equivalent to setting forwardX.value, forwardY.value, forwardZ.value, upX.value, upY.value, and upZ.value directly with the given x, y, z, xUp, yUp, and zUp values, respectively.

⌛ Consequently, if any of the forwardX, forwardY, forwardZ, upX, upY and upZ AudioParams have an automation curve set using setValueCurveAtTime() at the time this method is called, a NotSupportedError MUST be thrown.

setOrientation() describes which direction the listener is pointing in the 3D cartesian coordinate space. Both a forward vector and an up vector are provided. In simple human terms, the ***forward*** vector represents which direction the person's nose is pointing. The ***up*** vector represents the direction the top of a person's head is pointing. These two vectors are expected to be linearly independent. For normative requirements of how these values are to be interpreted, see the § 6 Spatialization/Panning.

The x, y, and z parameters represent a forward direction vector in 3D space, with the default value being (0,0,-1).

The xUp, yUp, and zUp parameters represent an up direction vector in 3D space, with the default value being (0,1,0).

Arguments for the AudioListener.setOrientation() method.

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
x	float	✗	✗	forward *x* direction fo the AudioListener
y	float	✗	✗	forward *y* direction fo the AudioListener
z	float	✗	✗	forward *z* direction fo the AudioListener
xUp	float	✗	✗	up *x* direction fo the AudioListener
yUp	float	✗	✗	up *y* direction fo the AudioListener
zUp	float	✗	✗	up *z* direction fo the AudioListener

Return type: undefined

setPosition(x, y, z)

This method is DEPRECATED. It is equivalent to setting positionX.value, positionY.value, and positionZ.value directly with the given x, y, and z values, respectively.

⏳ Consequently, any of the positionX, positionY, and positionZ AudioParams for this AudioListener have an automation curve set using setValueCurveAtTime() at the time this method is called, a NotSupportedError MUST be thrown.

setPosition() sets the position of the listener in a 3D cartesian coordinate space. PannerNode objects use this position relative to individual audio sources for spatialization.

The x, y, and z parameters represent the coordinates in 3D space.

The default value is (0,0,0).

Arguments for the AudioListener.setPosition() method.

Parameter	Type	Nullable	Optional	Description
x	float	✗	✗	*x*-coordinate of the position of the AudioListener
y	float	✗	✗	*y*-coordinate of the position of the AudioListener
z	float	✗	✗	*z*-coordinate of the position of the AudioListener

✓ MDN

§ 1.11.3. Processing

Because AudioListener's parameters can be connected with AudioNodes and they can also affect the output of PannerNodes in the same graph, the node ordering algorithm should take the AudioListener into consideration when computing the order of processing. For this reason, all the PannerNodes in the graph have the AudioListener as input.

§ 1.12. The AudioProcessingEvent Interface - DEPRECATED

This is an Event object which is dispatched to ScriptProcessorNode nodes. It will be removed when the ScriptProcessorNode is removed, as the replacement AudioWorkletNode uses a different approach.

The event handler processes audio from the input (if any) by accessing the audio data from the inputBuffer attribute. The audio data which is the result of the processing (or the synthesized data if there are no inputs) is then placed into the outputBuffer.

✓ MDN

```
[Exposed=Window]
interface AudioProcessingEvent : Event {
    constructor (DOMString type, AudioProcessingEventInit eventInitDict);
    readonly attribute double playbackTime;
```

Error preparing HTML-CSS output (preProcess)

```
    readonly attribute AudioBuffer inputBuffer;
    readonly attribute AudioBuffer outputBuffer;
};
```

1.12.1. Attributes

***inputBuffer*, of type AudioBuffer, readonly**
: An AudioBuffer containing the input audio data. It will have a number of channels equal to the `numberOfInputChannels` parameter of the createScriptProcessor() method. This AudioBuffer is only valid while in the scope of the `audioprocess` event handler functions. Its values will be meaningless outside of this scope.

***outputBuffer*, of type AudioBuffer, readonly**
: An AudioBuffer where the output audio data MUST be written. It will have a number of channels equal to the `numberOfOutputChannels` parameter of the createScriptProcessor() method. Script code within the scope of the `audioprocess` event handler functions are expected to modify the `Float32Array` arrays representing channel data in this AudioBuffer. Any script modifications to this AudioBuffer outside of this scope will not produce any audible effects.

***playbackTime*, of type double, readonly**
: The time when the audio will be played in the same time coordinate system as the `AudioContext`'s `currentTime`.

1.12.2. `AudioProcessingEventInit`

```
dictionary AudioProcessingEventInit : EventInit {
    required double playbackTime;
    required AudioBuffer inputBuffer;
    required AudioBuffer outputBuffer;
};
```

1.12.2.1. Dictionary `AudioProcessingEventInit` Members

***inputBuffer*, of type AudioBuffer**
: Value to be assigned to the `inputBuffer` attribute of the event.

***outputBuffer*, of type AudioBuffer**
: Value to be assigned to the `outputBuffer` attribute of the event.

***playbackTime*, of type double**
: Value to be assigned to the `playbackTime` attribute of the event.

1.13. The `BiquadFilterNode` Interface

`BiquadFilterNode` is an `AudioNode` processor implementing very common low-order filters.

Low-order filters are the building blocks of basic tone controls (bass, mid, treble), graphic equalizers, and more advanced filters. Multiple `BiquadFilterNode` filters can be combined to form more complex filters. The filter parameters such as `frequency` can be changed over time for filter sweeps, etc. Each `BiquadFilterNode` can be configured as one of a number of common filter types as shown in the IDL below. The default filter type is `"lowpass"`.

Both `frequency` and `detune` form a compound parameter and are both a-rate. They are used together to determine a ***computedFrequency*** value:

```
computedFrequency(t) = frequency(t) * pow(2, detune(t) / 1200)
```

The nominal range for this compound parameter is [0, Nyquist frequency].

Property	Value	Notes
`numberOfInputs`	1	
`tputs`	1	

Error preparing HTML-CSS output (preProcess)

Property	Value	Notes
channelCount	2	
channelCountMode	"max"	
channelInterpretation	"speakers"	
tail-time	Yes	Continues to output non-silent audio with zero input. Since this is an IIR filter, the filter produces non-zero input forever, but in practice, this can be limited after some finite time where the output is sufficiently close to zero. The actual time depends on the filter coefficients.

The number of channels of the output always equals the number of channels of the input.

```
enum BiquadFilterType {
    "lowpass",
    "highpass",
    "bandpass",
    "lowshelf",
    "highshelf",
    "peaking",
    "notch",
    "allpass"
};
```

BiquadFilterType enumeration description

Enum value	Description
"*lowpass*"	A lowpass filter allows frequencies below the cutoff frequency to pass through and attenuates frequencies above the cutoff. It implements a standard second-order resonant lowpass filter with 12dB/octave rolloff. **frequency** The cutoff frequency **Q** Controls how peaked the response will be at the cutoff frequency. A large value makes the response more peaked. **gain** Not used in this filter type
"*highpass*"	A highpass filter is the opposite of a lowpass filter. Frequencies above the cutoff frequency are passed through, but frequencies below the cutoff are attenuated. It implements a standard second-order resonant highpass filter with 12dB/octave rolloff. **frequency** The cutoff frequency below which the frequencies are attenuated **Q** Controls how peaked the response will be at the cutoff frequency. A large value makes the response more peaked. **gain** Not used in this filter type

Error preparing HTML-CSS output (preProcess)

Enum value	Description
"*bandpass*"	A bandpass filter allows a range of frequencies to pass through and attenuates the frequencies below and above this frequency range. It implements a second-order bandpass filter. **frequency** The center of the frequency band **Q** Controls the width of the band. The width becomes narrower as the Q value increases. **gain** Not used in this filter type
"*lowshelf*"	The lowshelf filter allows all frequencies through, but adds a boost (or attenuation) to the lower frequencies. It implements a second-order lowshelf filter. **frequency** The upper limit of the frequences where the boost (or attenuation) is applied. **Q** Not used in this filter type. **gain** The boost, in dB, to be applied. If the value is negative, the frequencies are attenuated.
"*highshelf*"	The highshelf filter is the opposite of the lowshelf filter and allows all frequencies through, but adds a boost to the higher frequencies. It implements a second-order highshelf filter **frequency** The lower limit of the frequences where the boost (or attenuation) is applied. **Q** Not used in this filter type. **gain** The boost, in dB, to be applied. If the value is negative, the frequencies are attenuated.
"*peaking*"	The peaking filter allows all frequencies through, but adds a boost (or attenuation) to a range of frequencies. **frequency** The center frequency of where the boost is applied. **Q** Controls the width of the band of frequencies that are boosted. A large value implies a narrow width. **gain** The boost, in dB, to be applied. If the value is negative, the frequencies are attenuated.
"*notch*"	The notch filter (also known as a band-stop or band-rejection filter) is the opposite of a bandpass filter. It allows all frequencies through, except for a set of frequencies. **frequency** The center frequency of where the notch is applied. **Q** Controls the width of the band of frequencies that are attenuated. A large value implies a narrow width. **gain** Not used in this filter type.
"*allpass*"	An allpass filter allows all frequencies through, but changes the phase relationship between the various frequencies. It implements a second-order allpass filter **frequency** The frequency where the center of the phase transition occurs. Viewed another way, this is the frequency with maximal group delay.

Error preparing HTML-CSS output (preProcess)

Enum value	Description
	Q Controls how sharp the phase transition is at the center frequency. A larger value implies a sharper transition and a larger group delay. **gain** Not used in this filter type.

All attributes of the `BiquadFilterNode` are a-rate `AudioParam`s.

```
[Exposed=Window]
interface BiquadFilterNode : AudioNode {
    constructor (BaseAudioContext context, optional BiquadFilterOptions options = {});
    attribute BiquadFilterType type;
    readonly attribute AudioParam frequency;
    readonly attribute AudioParam detune;
    readonly attribute AudioParam Q;
    readonly attribute AudioParam gain;
    undefined getFrequencyResponse (Float32Array frequencyHz,
                                    Float32Array magResponse,
                                    Float32Array phaseResponse);
};
```

§ 1.13.1. Constructors

BiquadFilterNode(context, options)

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the BiquadFilterNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `BiquadFilterNode` will be associated with.
options	`BiquadFilterOptions`	✗	✓	Optional initial parameter value for this `BiquadFilterNode`.

§ 1.13.2. Attributes

***Q*, of type AudioParam, readonly**

The Q factor of the filter.

For `lowpass` and `highpass` filters the `Q` value is interpreted to be in dB. For these filters the nominal range is $[-Q_{lim}, Q_{lim}]$ where Q_{lim} is the largest value for which $10^{Q/20}$ does not overflow. This is approximately 770.63678.

For the `bandpass`, `notch`, `allpass`, and `peaking` filters, this value is a linear value. The value is related to the bandwidth of the filter and hence should be a positive value. The nominal range is [0, 3.4028235*e*38], the upper limit being the most-positive-single-float.

This is not used for the `lowshelf` and `highshelf` filters.

Parameter	Value	Notes
`defaultValue`	1	

Error preparing HTML-CSS output (preProcess)

Parameter	Value	Notes
`minValue`	most-negative-single-float	Approximately -3.4028235e38, but see above for the actual limits for different filters
`maxValue`	most-positive-single-float	Approximately 3.4028235e38, but see above for the actual limits for different filters
`automationRate`	"`a-rate`"	

***`detune`*, of type AudioParam, readonly**

A detune value, in cents, for the frequency. It forms a compound parameter with `frequency` to form the computedFrequency.

Parameter	Value	Notes
`defaultValue`	0	
`minValue`	≈ -153600	
`maxValue`	≈ 153600	This value is approximately $1200 \log_2$FLT_MAX where FLT_MAX is the largest `float` value.
`automationRate`	"`a-rate`"	

***`frequency`*, of type AudioParam, readonly**

The frequency at which the `BiquadFilterNode` will operate, in Hz. It forms a compound parameter with `detune` to form the computedFrequency.

Parameter	Value	Notes
`defaultValue`	350	
`minValue`	0	
`maxValue`	Nyquist frequency	
`automationRate`	"`a-rate`"	

✓ MDN

***`gain`*, of type AudioParam, readonly**

The gain of the filter. Its value is in dB units. The gain is only used for `lowshelf`, `highshelf`, and `peaking` filters.

Parameter	Value	Notes
`defaultValue`	0	
`minValue`	most-negative-single-float	Approximately -3.4028235e38
`maxValue`	≈ 1541	This value is approximately $40 \log_{10}$FLT_MAX where FLT_MAX is the largest `float` value.
`automationRate`	"`a-rate`"	

***`type`*, of type BiquadFilterType**

The type of this `BiquadFilterNode`. Its default value is "`lowpass`". The exact meaning of the other parameters depend on the value of the `type` attribute.

Error preparing HTML-CSS output (preProcess)

1.13.3. Methods

getFrequencyResponse(frequencyHz, magResponse, phaseResponse)

⏳ Given the [[current value]] from each of the filter parameters, synchronously calculates the frequency response for the specified frequencies. The three parameters MUST be Float32Arrays of the same length, or an InvalidAccessError MUST be thrown.

The frequency response returned MUST be computed with the AudioParam sampled for the current processing block.

Arguments for the BiquadFilterNode.getFrequencyResponse() method.

Parameter	Type	Nullable	Optional	Description
frequencyHz	Float32Array	✗	✗	This parameter specifies an array of frequencies, in Hz, at which the response values will be calculated.
magResponse	Float32Array	✗	✗	This parameter specifies an output array receiving the linear magnitude response values. If a value in the `frequencyHz` parameter is not within [0, sampleRate/2], where `sampleRate` is the value of the sampleRate property of the AudioContext, the corresponding value at the same index of the `magResponse` array MUST be `NaN`.
phaseResponse	Float32Array	✗	✗	This parameter specifies an output array receiving the phase response values in radians. If a value in the `frequencyHz` parameter is not within [0; sampleRate/2], where `sampleRate` is the value of the sampleRate property of the AudioContext, the corresponding value at the same index of the `phaseResponse` array MUST be `NaN`.

Return type: undefined

1.13.4. BiquadFilterOptions

This specifies the options to be used when constructing a BiquadFilterNode. All members are optional; if not specified, the normal default values are used to construct the node.

✓ MDN

```
dictionary BiquadFilterOptions : AudioNodeOptions {
    BiquadFilterType type = "lowpass";
    float Q = 1;
    float detune = 0;
    float frequency = 350;
    float gain = 0;
};
```

1.13.4.1. Dictionary BiquadFilterOptions Members

Q, of type float, defaulting to 1
: The desired initial value for Q.

***detune*, of type float, defaulting to 0**
: The desired initial value for detune.

Error preparing HTML-CSS output (preProcess)

***frequency*, of type float, defaulting to `350`**
The desired initial value for `frequency`.

***gain*, of type float, defaulting to `0`**
The desired initial value for `gain`.

***type*, of type BiquadFilterType, defaulting to `"lowpass"`**
The desired initial type of the filter.

§ 1.13.5. Filters Characteristics

There are multiple ways of implementing the type of filters available through the `BiquadFilterNode` each having very different characteristics. The formulas in this section describe the filters that a conforming implementation MUST implement, as they determine the characteristics of the different filter types. They are inspired by formulas found in the Audio EQ Cookbook.

✓ MDN

The `BiquadFilterNode` processes audio with a transfer function of

$$H(z) = \frac{\frac{b_0}{a_0} + \frac{b_1}{a_0}z^{-1} + \frac{b_2}{a_0}z^{-2}}{1 + \frac{a_1}{a_0}z^{-1} + \frac{a_2}{a_0}z^{-2}}$$

which is equivalent to a time-domain equation of:

$$a_0 y(n) + a_1 y(n-1) + a_2 y(n-2) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2)$$

The initial filter state is 0.

NOTE: While fixed filters are stable, it is possible to create unstable biquad filters using automations of `AudioParam`s. It is the developers responsibility to manage this.

NOTE: The UA may produce a warning to notify the user that NaN values have occurred in the filter state. This is usually indicative of an unstable filter.

The coefficients in the transfer function above are different for each node type. The following intermediate variables are necessary for their computation, based on the computedValue of the `AudioParam`s of the `BiquadFilterNode`.

- Let F_s be the value of the `sampleRate` attribute for this `AudioContext`.
- Let f_0 be the value of the computedFrequency.
- Let G be the value of the `gain` `AudioParam`.
- Let Q be the value of the `Q` `AudioParam`.
- Finally let

$$A = 10^{\frac{G}{40}}$$
$$\omega_0 = 2\pi\frac{f_0}{F_s}$$
$$\alpha_Q = \frac{\sin\omega_0}{2Q}$$
$$\alpha_{Q_{dB}} = \frac{\sin\omega_0}{2 \cdot 10^{Q/20}}$$
$$S = 1$$
$$\alpha_S = \frac{\sin\omega_0}{2}\sqrt{\left(A + \frac{1}{A}\right)\left(\frac{1}{S} - 1\right) + 2}$$

Error preparing HTML-CSS output (preProcess)

The six coefficients ($b_0, b_1, b_2, a_0, a_1, a_2$) for each filter type, are:

"lowpass"

$$b_0 = \frac{1 - \cos\omega_0}{2}$$
$$b_1 = 1 - \cos\omega_0$$
$$b_2 = \frac{1 - \cos\omega_0}{2}$$
$$a_0 = 1 + \alpha_{Q_{dB}}$$
$$a_1 = -2\cos\omega_0$$
$$a_2 = 1 - \alpha_{Q_{dB}}$$

"highpass"

$$b_0 = \frac{1 + \cos\omega_0}{2}$$
$$b_1 = -(1 + \cos\omega_0)$$
$$b_2 = \frac{1 + \cos\omega_0}{2}$$
$$a_0 = 1 + \alpha_{Q_{dB}}$$
$$a_1 = -2\cos\omega_0$$
$$a_2 = 1 - \alpha_{Q_{dB}}$$

✓ MDN

"bandpass"

$$b_0 = \alpha_Q$$
$$b_1 = 0$$
$$b_2 = -\alpha_Q$$
$$a_0 = 1 + \alpha_Q$$
$$a_1 = -2\cos\omega_0$$
$$a_2 = 1 - \alpha_Q$$

"notch"

$$b_0 = 1$$
$$b_1 = -2\cos\omega_0$$
$$b_2 = 1$$
$$a_0 = 1 + \alpha_Q$$
$$a_1 = -2\cos\omega_0$$
$$a_2 = 1 - \alpha_Q$$

"allpass"

$$b_0 = 1 - \alpha_Q$$
$$b_1 = -2\cos\omega_0$$
$$b_2 = 1 + \alpha_Q$$
$$a_0 = 1 + \alpha_Q$$
$$a_1 = -2\cos\omega_0$$
$$a_2 = 1 - \alpha_Q$$

✓ MDN

Error preparing HTML-CSS output (preProcess)

"peaking"

$$b_0 = 1 + \alpha_Q A$$
$$b_1 = -2\cos\omega_0$$
$$b_2 = 1 - \alpha_Q A$$
$$a_0 = 1 + \frac{\alpha_Q}{A}$$
$$a_1 = -2\cos\omega_0$$
$$a_2 = 1 - \frac{\alpha_Q}{A}$$

"lowshelf"

$$b_0 = A\left[(A+1) - (A-1)\cos\omega_0 + 2\alpha_S\sqrt{A}\right]$$
$$b_1 = 2A\left[(A-1) - (A+1)\cos\omega_0\right]$$
$$b_2 = A\left[(A+1) - (A-1)\cos\omega_0 - 2\alpha_S\sqrt{A}\right]$$
$$a_0 = (A+1) + (A-1)\cos\omega_0 + 2\alpha_S\sqrt{A}$$
$$a_1 = -2\left[(A-1) + (A+1)\cos\omega_0\right]$$
$$a_2 = (A+1) + (A-1)\cos\omega_0 - 2\alpha_S\sqrt{A})$$

✓ MDN

"highshelf"

$$b_0 = A\left[(A+1) + (A-1)\cos\omega_0 + 2\alpha_S\sqrt{A}\right]$$
$$b_1 = -2A\left[(A-1) + (A+1)\cos\omega_0\right]$$
$$b_2 = A\left[(A+1) + (A-1)\cos\omega_0 - 2\alpha_S\sqrt{A}\right]$$
$$a_0 = (A+1) - (A-1)\cos\omega_0 + 2\alpha_S\sqrt{A}$$
$$a_1 = 2\left[(A-1) - (A+1)\cos\omega_0\right]$$
$$a_2 = (A+1) - (A-1)\cos\omega_0 - 2\alpha_S\sqrt{A}$$

§ 1.14. The ChannelMergerNode Interface

✓ MDN

The ChannelMergerNode is for use in more advanced applications and would often be used in conjunction with ChannelSplitterNode.

Property	Value	Notes
numberOfInputs	see notes	Defaults to 6, but is determined by ChannelMergerOptions.numberOfInputs or the value specified by createChannelMerger.
numberOfOutputs	1	
channelCount	1	Has channelCount constraints
channelCountMode	"explicit"	Has channelCountMode constraints
channelInterpretation	"speakers"	
tail-time	No	

This interface represents an AudioNode for combining channels from multiple audio streams into a single audio stream. It has a variable number of inputs (defaulting to 6), but not all of them need be connected. There is a single output whose audio

Error preparing HTML-CSS output (preProcess)

stream has a number of channels equal to the number of inputs when any of the inputs is actively processing. If none of the inputs are actively processing, then output is a single channel of silence.

To merge multiple inputs into one stream, each input gets downmixed into one channel (mono) based on the specified mixing rule. An unconnected input still counts as **one silent channel** in the output. Changing input streams does **not** affect the order of output channels.

EXAMPLE 9

For example, if a default `ChannelMergerNode` has two connected stereo inputs, the first and second input will be downmixed to mono respectively before merging. The output will be a 6-channel stream whose first two channels are be filled with the first two (downmixed) inputs and the rest of channels will be silent.

Also the `ChannelMergerNode` can be used to arrange multiple audio streams in a certain order for the multi-channel speaker array such as 5.1 surround set up. The merger does not interpret the channel identities (such as left, right, etc.), but simply combines channels in the order that they are input.

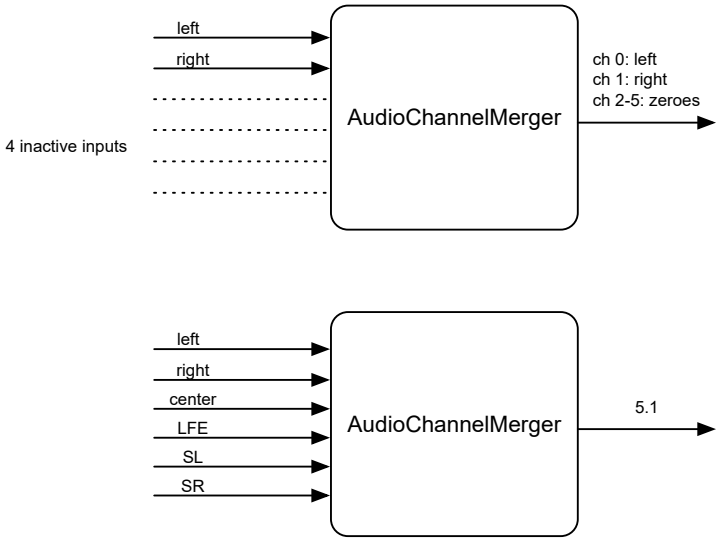


Figure 11 *A diagram of ChannelMerger*

```
[Exposed=Window]
interface ChannelMergerNode : AudioNode {
    constructor (BaseAudioContext context, optional ChannelMergerOptions options = {});
};
```

§ 1.14.1. Constructors

`ChannelMergerNode(context, options)`

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the ChannelMergerNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `ChannelMergerNode` will be associated with.
options	`ChannelMergerOptions`	✗	✓	Optional initial parameter value for this `ChannelMergerNode`.

Error preparing HTML-CSS output (preProcess)

1.14.2. `ChannelMergerOptions`

```
dictionary ChannelMergerOptions : AudioNodeOptions {
    unsigned long numberOfInputs = 6;
};
```

1.14.2.1. Dictionary `ChannelMergerOptions` Members

***`numberOfInputs`*, of type unsigned long, defaulting to `6`**
: The number inputs for the `ChannelMergerNode`. See `createChannelMerger()` for constraints on this value.

✓ MDN

1.15. The `ChannelSplitterNode` Interface

The `ChannelSplitterNode` is for use in more advanced applications and would often be used in conjunction with `ChannelMergerNode`.

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	see notes	This defaults to 6, but is otherwise determined from `ChannelSplitterOptions.numberOfOutputs` or the value specified by `createChannelSplitter` or the `numberOfOutputs` member of the `ChannelSplitterOptions` dictionary for the `constructor`.
`channelCount`	`numberOfOutputs`	Has channelCount constraints
`channelCountMode`	"`explicit`"	Has channelCountMode constraints
`channelInterpretation`	"`discrete`"	Has channelInterpretation constraints
tail-time	No	

✓ MDN

This interface represents an `AudioNode` for accessing the individual channels of an audio stream in the routing graph. It has a single input, and a number of "active" outputs which equals the number of channels in the input audio stream. For example, if a stereo input is connected to an `ChannelSplitterNode` then the number of active outputs will be two (one from the left channel and one from the right). There are always a total number of N outputs (determined by the `numberOfOutputs` parameter to the `AudioContext` method `createChannelSplitter()`), The default number is 6 if this value is not provided. Any outputs which are not "active" will output silence and would typically not be connected to anything.

Error preparing HTML-CSS output (preProcess)

EXAMPLE 10 ¶

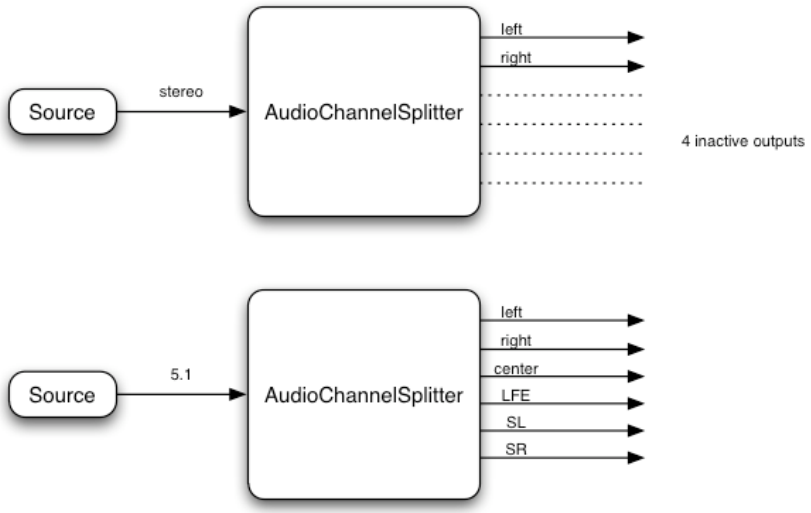


Figure 12 *A diagram of a ChannelSplitter*

Please note that in this example, the splitter does **not** interpret the channel identities (such as left, right, etc.), but simply splits out channels in the order that they are input.

One application for `ChannelSplitterNode` is for doing "matrix mixing" where individual gain control of each channel is desired.

```
[Exposed=Window]
interface ChannelSplitterNode : AudioNode {
    constructor (BaseAudioContext context, optional ChannelSplitterOptions options = {});
};
```

§ **1.15.1. Constructors**

`ChannelSplitterNode(context, options)`

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the ChannelSplitterNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `ChannelSplitterNode` will be associated with.
options	`ChannelSplitterOptions`	✗	✓	Optional initial parameter value for this `ChannelSplitterNode`.

§ **1.15.2. `ChannelSplitterOptions`**

```
dictionary ChannelSplitterOptions : AudioNodeOptions {
    unsigned long numberOfOutputs = 6;
```

Error preparing HTML-CSS output (preProcess)

§ 1.15.2.1. Dictionary `ChannelSplitterOptions` Members

***numberOfOutputs*, of type unsigned long, defaulting to 6**
: The number outputs for the `ChannelSplitterNode`. See `createChannelSplitter()` for constraints on this value.

§ 1.16. The `ConstantSourceNode` Interface

This interface represents a constant audio source whose output is nominally a constant value. It is useful as a constant source node in general and can be used as if it were a constructible `AudioParam` by automating its `offset` or connecting another node to it.

The single output of this node consists of one channel (mono).

Property	Value	Notes
`numberOfInputs`	0	
`numberOfOutputs`	1	
`channelCount`	2	
`channelCountMode`	"`max`"	
`channelInterpretation`	"`speakers`"	
tail-time	No	

```
[Exposed=Window]
interface ConstantSourceNode : AudioScheduledSourceNode {
    constructor (BaseAudioContext context, optional ConstantSourceOptions options = {});
    readonly attribute AudioParam offset;
};
```

§ 1.16.1. Constructors

ConstantSourceNode(context, options)
: When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the ConstantSourceNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `ConstantSourceNode` will be associated with.
options	`ConstantSourceOptions`	✗	✓	Optional initial parameter value for this `ConstantSourceNode`.

§ 1.16.2. Attributes

***offset*, of type AudioParam, readonly**
: The constant value of the source.

Parameter	Value	Notes
`:Value`	1	

Error preparing HTML-CSS output (preProcess)

Parameter	Value	Notes
`minValue`	most-negative-single-float	Approximately -3.4028235e38
`maxValue`	most-positive-single-float	Approximately 3.4028235e38
`automationRate`	"`a-rate`"	

1.16.3. `ConstantSourceOptions`

This specifies options for constructing a `ConstantSourceNode`. All members are optional; if not specified, the normal defaults are used for constructing the node.

```
dictionary ConstantSourceOptions {
    float offset = 1;
};
```

1.16.3.1. Dictionary `ConstantSourceOptions` Members

***`offset`*, of type float, defaulting to `1`**
: The initial value for the `offset` AudioParam of this node.

MDN

1.17. The `ConvolverNode` Interface

This interface represents a processing node which applies a linear convolution effect given an impulse response.

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	1	
`channelCount`	2	Has channelCount constraints
`channelCountMode`	"`clamped-max`"	Has channelCountMode constraints
`channelInterpretation`	"`speakers`"	
tail-time	Yes	Continues to output non-silent audio with zero input for the length of the `buffer`.

The input of this node is either mono (1 channel) or stereo (2 channels) and cannot be increased. Connections from nodes with more channels will be down-mixed appropriately.

There are channelCount constraints and channelCountMode constraints for this node. These constraints ensure that the input to the node is either mono or stereo.

```
[Exposed=Window]
interface ConvolverNode : AudioNode {
    constructor (BaseAudioContext context, optional ConvolverOptions options = {});
    attribute AudioBuffer? buffer;
    attribute boolean normalize;
};
```

1.17.1. Constructors

`ConvolverNode(context, options)`
: When the constructor is called with a `BaseAudioContext` *context* and an option object *options*, execute these steps:

Error preparing HTML-CSS output (preProcess)

s `normalize` to the inverse of the value of `disableNormalization`.

2. If `buffer` exists, set the `buffer` attribute to its value.

> NOTE: This means that the buffer will be normalized according to the value of the `normalize` attribute.

3. Let *o* be new `AudioNodeOptions` dictionary.
4. If `channelCount` exists in *options*, set `channelCount` on *o* with the same value.
5. If `channelCountMode` exists in *options*, set `channelCountMode` on *o* with the same value.
6. If `channelInterpretation` exists in *options*, set `channelInterpretation` on *o* with the same value.
7. Initialize the AudioNode *this*, with *c* and *o* as argument.

Arguments for the ConvolverNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `ConvolverNode` will be associated with.
options	`ConvolverOptions`	✗	✓	Optional initial parameter value for this `ConvolverNode`.

§ 1.17.2. Attributes

***`buffer`*, of type AudioBuffer, nullable**

At the time when this attribute is set, the `buffer` and the state of the `normalize` attribute will be used to configure the `ConvolverNode` with this impulse response having the given normalization. The initial value of this attribute is null.

When setting the ***`buffer` attribute***, execute the following steps synchronously:

1. If the buffer `number of channels` is not 1, 2, 4, or if the `sample-rate` of the buffer is not the same as the `sample-rate` of its associated `BaseAudioContext`, a `NotSupportedError` MUST be thrown.
2. Acquire the content of the `AudioBuffer`.

> NOTE: If the `buffer` is set to an new buffer, audio may glitch. If this is undesirable, it is recommended to create a new `ConvolverNode` to replace the old, possibly cross-fading between the two.

> NOTE: The `ConvolverNode` produces a mono output only in the single case where there is a single input channel and a single-channel `buffer`. In all other cases, the output is stereo. In particular, when the `buffer` has four channels and there are two input channels, the `ConvolverNode` performs matrix "true" stereo convolution. For normative information please see the channel configuration diagrams

***`normalize`*, of type boolean**

Controls whether the impulse response from the buffer will be scaled by an equal-power normalization when the `buffer` atttribute is set. Its default value is `true` in order to achieve a more uniform output level from the convolver when loaded with diverse impulse responses. If `normalize` is set to `false`, then the convolution will be rendered with no pre-processing/scaling of the impulse response. Changes to this value do not take effect until the next time the `buffer` attribute is set.

If the `normalize` attribute is false when the `buffer` attribute is set then the `ConvolverNode` will perform a linear convolution given the exact impulse response contained within the `buffer`.

Otherwise, if the `normalize` attribute is true when the `buffer` attribute is set then the `ConvolverNode` will first perform a scaled RMS-power analysis of the audio data contained within `buffer` to calculate a *normalizationScale* given this algorithm:

```
function calculateNormalizationScale(buffer) {
    const GainCalibration = 0.00125;
    const GainCalibrationSampleRate = 44100;
    .nPower = 0.000125;
```

Error preparing HTML-CSS output (preProcess)

```
5
6       // Normalize by RMS power.
7       const numberOfChannels = buffer.numberOfChannels;
8       const length = buffer.length;
9
10      let power = 0;
11
12      for (let i = 0; i < numberOfChannels; i++) {
13          let channelPower = 0;
14          const channelData = buffer.getChannelData(i);
15
16          for (let j = 0; j < length; j++) {
17              const sample = channelData[j];
18              channelPower += sample * sample;
19          }
20
21          power += channelPower;
22      }
23
24      power = Math.sqrt(power / (numberOfChannels * length));
25
26      // Protect against accidental overload.
27      if (!isFinite(power) || isNaN(power) || power < MinPower)
28          power = MinPower;
29
30      let scale = 1 / power;
31
32      // Calibrate to make perceived volume same as unprocessed.
33      scale *= GainCalibration;
34
35      // Scale depends on sample-rate.
36      if (buffer.sampleRate)
37          scale *= GainCalibrationSampleRate / buffer.sampleRate;
38
39      // True-stereo compensation.
40      if (numberOfChannels == 4)
41          scale *= 0.5;
42
43      return scale;
44  }
```


During processing, the ConvolverNode will then take this calculated *normalizationScale* value and multiply it by the result of the linear convolution resulting from processing the input with the impulse response (represented by the `buffer`) to produce the final output. Or any mathematically equivalent operation may be used, such as pre-multiplying the input by *normalizationScale*, or pre-multiplying a version of the impulse-response by *normalizationScale*.

§ 1.17.3. `ConvolverOptions`

The specifies options for constructing a `ConvolverNode`. All members are optional; if not specified, the node is contructing using the normal defaults.

```
dictionary ConvolverOptions : AudioNodeOptions {
    AudioBuffer? buffer;
    boolean disableNormalization = false;
};
```

§ *1.17.3.1. Dictionary `ConvolverOptions` Members*

***buffer*, of type AudioBuffer, nullable**

The desired buffer for the `ConvolverNode`. This buffer will be normalized according to the value of

ation.

***disableNormalization*, of type boolean, defaulting to `false`**
The opposite of the desired initial value for the `normalize` attribute of the `ConvolverNode`.

✓ MDN

§ 1.17.4. Channel Configurations for Input, Impulse Response and Output

Implementations MUST support the following allowable configurations of impulse response channels in a `ConvolverNode` to achieve various reverb effects with 1 or 2 channels of input.

As shown in the diagram below, single channel convolution operates on a mono audio input, using a mono impulse response, and generating a mono output. The remaining images in the diagram illustrate the supported cases for mono and stereo playback where the number of channels of the input is 1 or 2, and the number of channels in the `buffer` is 1, 2, or 4. Developers desiring more complex and arbitrary matrixing can use a `ChannelSplitterNode`, multiple single-channel `ConvolverNode`s and a `ChannelMergerNode`.

If this node is not actively processing, the output is a single channel of silence.

NOTE: The diagrams below show the outputs when actively processing.

✓ MDN

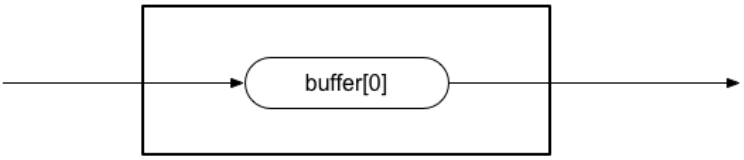
✓ MDN

✓ MDN

✓ MDN

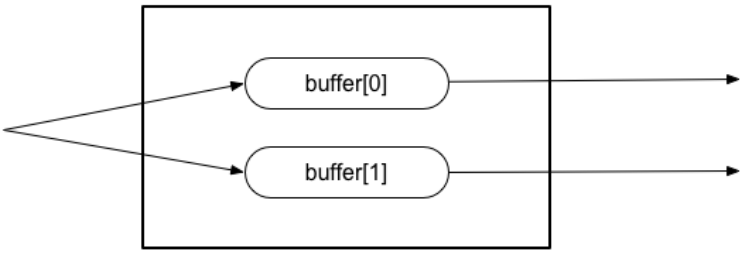
Error preparing HTML-CSS output (preProcess)

Mono with mono response



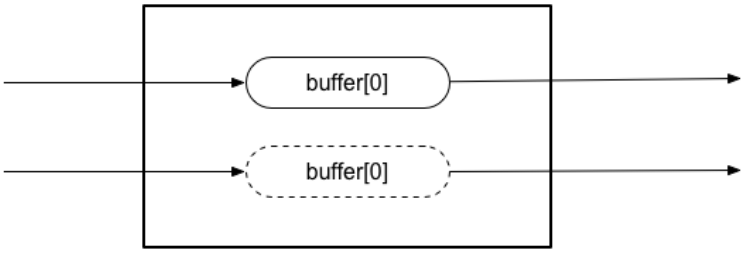
✓ MDN

Mono with stereo response

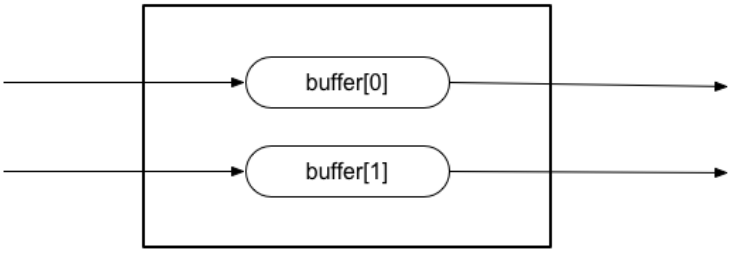


✓ MDN

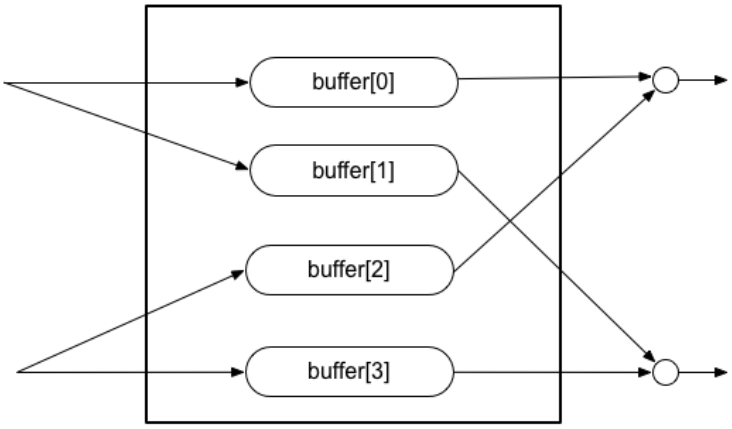
Stereo with mono response



Stereo with stereo response



Stereo with "true"
stereo matrix response



Error preparing HTML-CSS output (preProcess)

Mono with stereo matrix response

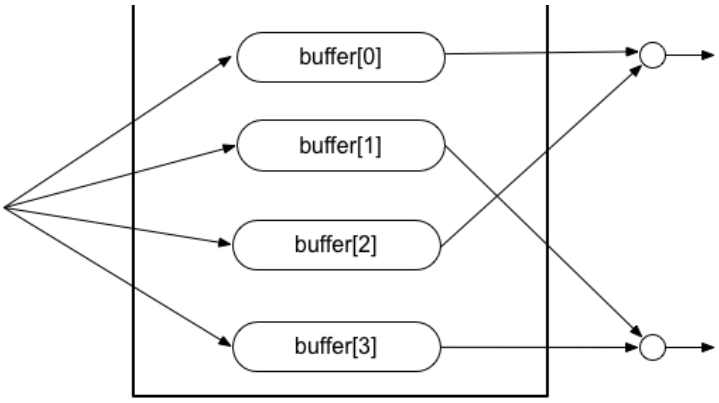


Figure 13 *A graphical representation of supported input and output channel count possibilities when using a* `ConvolverNode`.

§ 1.18. The `DelayNode` Interface

A delay-line is a fundamental building block in audio applications. This interface is an `AudioNode` with a single input and single output.

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	1	
`channelCount`	2	
`channelCountMode`	"`max`"	
`channelInterpretation`	"`speakers`"	
tail-time	Yes	Continues to output non-silent audio with zero input up to the `maxDelayTime` of the node.

The number of channels of the output always equals the number of channels of the input.

It delays the incoming audio signal by a certain amount. Specifically, at each time *t*, input signal *input(t)*, delay time *delayTime(t)* and output signal *output(t)*, the output will be *output(t) = input(t - delayTime(t))*. The default `delayTime` is 0 seconds (no delay).

When the number of channels in a `DelayNode`'s input changes (thus changing the output channel count also), there may be delayed audio samples which have not yet been output by the node and are part of its internal state. If these samples were received earlier with a different channel count, they MUST be upmixed or downmixed before being combined with newly received input so that all internal delay-line mixing takes place using the single prevailing channel layout.

NOTE: By definition, a `DelayNode` introduces an audio processing latency equal to the amount of the delay.

```
[Exposed=Window]
interface DelayNode : AudioNode {
    constructor (BaseAudioContext context, optional DelayOptions options = {});
    readonly attribute AudioParam delayTime;
};
```

§ 1.18.1. Constructors

`DelayNode(context, options)`

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Error preparing HTML-CSS output (preProcess)

Arguments for the DelayNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	BaseAudioContext	✗	✗	The BaseAudioContext this new DelayNode will be associated with.
options	DelayOptions	✗	✓	Optional initial parameter value for this DelayNode.

1.18.2. Attributes

***delayTime*, of type AudioParam, readonly**
: An AudioParam object representing the amount of delay (in seconds) to apply. Its default `value` is 0 (no delay). The minimum value is 0 and the maximum value is determined by the `maxDelayTime` argument to the AudioContext method `createDelay()` or the `maxDelayTime` member of the DelayOptions dictionary for the `constructor`.

If DelayNode is part of a cycle, then the value of the `delayTime` attribute is clamped to a minimum of one render quantum.

Parameter	Value	Notes
`defaultValue`	0	
`minValue`	0	
`maxValue`	`maxDelayTime`	
`automationRate`	"`a-rate`"	

1.18.3. `DelayOptions`

This specifies options for constructing a DelayNode. All members are optional; if not given, the node is constructed using the normal defaults.

```
dictionary DelayOptions : AudioNodeOptions {
    double maxDelayTime = 1;
    double delayTime = 0;
};
```

1.18.3.1. Dictionary `DelayOptions` Members

***delayTime*, of type double, defaulting to `0`**
: The initial delay time for the node.

***maxDelayTime*, of type double, defaulting to `1`**
: The maximum delay time for the node. See `createDelay(maxDelayTime)` for constraints.

1.18.4. Processing

A DelayNode has an internal buffer that holds `delayTime` seconds of audio.

The processing of a DelayNode is broken down in two parts: writing to the delay line, and reading from the delay line. This is done via two internal AudioNodes (that are not available to authors and exist only to ease the description of the inner workings of the node). Both are created from a DelayNode.

Creating a ***DelayWriter*** for a DelayNode means creating an object that has the same interface as an AudioNode, and that writes the input audio into the internal buffer of the DelayNode. It has the same input connections as the DelayNode it was created from.

Error preparing HTML-CSS output (preProcess)

Creating a ***DelayReader*** for a `DelayNode` means creating an object that has the same interface as an `AudioNode`, and that can read the audio data from the internal buffer of the `DelayNode`. It is connected to the same `AudioNode`s as the `DelayNode` it was created from. A DelayReader is a source node.

When processing an input buffer, a DelayWriter MUST write the audio to the internal buffer of the `DelayNode`.

When producing an output buffer, a DelayReader MUST yield exactly the audio that was written to the corresponding DelayWriter `delayTime` seconds ago.

> NOTE: This means that channel count changes are reflected after the delay time has passed.

§ 1.19. The `DynamicsCompressorNode` Interface

`DynamicsCompressorNode` is an `AudioNode` processor implementing a dynamics compression effect.

Dynamics compression is very commonly used in musical production and game audio. It lowers the volume of the loudest parts of the signal and raises the volume of the softest parts. Overall, a louder, richer, and fuller sound can be achieved. It is especially important in games and musical applications where large numbers of individual sounds are played simultaneous to control the overall signal level and help avoid clipping (distorting) the audio output to the speakers.

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	1	
`channelCount`	2	Has channelCount constraints
`channelCountMode`	"`clamped-max`"	Has channelCountMode constraints
`channelInterpretation`	"`speakers`"	
tail-time	Yes	This node has a tail-time such that this node continues to output non-silent audio with zero input due to the look-ahead delay.

```
[Exposed=Window]
interface DynamicsCompressorNode : AudioNode {
    constructor (BaseAudioContext context,
                 optional DynamicsCompressorOptions options = {});
    readonly attribute AudioParam threshold;
    readonly attribute AudioParam knee;
    readonly attribute AudioParam ratio;
    readonly attribute float reduction;
    readonly attribute AudioParam attack;
    readonly attribute AudioParam release;
};
```

§ 1.19.1. Constructors

✓ MDN

`DynamicsCompressorNode(context, options)`

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Let ***`[[internal reduction]]`*** be a private slot on *this*, that holds a floating point number, in decibels. Set `[[internal reduction]]` to 0.0.

Arguments for the DynamicsCompressorNode.constructor() method.

Parameter	Type	Nullable	Optional	Description

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
context	BaseAudioContext	✗	✗	The BaseAudioContext this new DynamicsCompressorNode will be associated with.
options	DynamicsCompressorOptions	✗	✓	Optional initial parameter value for this DynamicsCompressorNode.

§ 1.19.2. Attributes

***attack*, of type AudioParam, readonly**
The amount of time (in seconds) to reduce the gain by 10dB.

✓ MDN

Parameter	Value	Notes
defaultValue	.003	
minValue	0	
maxValue	1	
automationRate	"k-rate"	Has automation rate constraints

***knee*, of type AudioParam, readonly**
A decibel value representing the range above the threshold where the curve smoothly transitions to the "ratio" portion.

Parameter	Value	Notes
defaultValue	30	
minValue	0	
maxValue	40	
automationRate	"k-rate"	Has automation rate constraints

✓ MDN

***ratio*, of type AudioParam, readonly**
The amount of dB change in input for a 1 dB change in output.

Parameter	Value	Notes
defaultValue	12	
minValue	1	
maxValue	20	
automationRate	"k-rate"	Has automation rate constraints

***reduction*, of type float, readonly**
A read-only decibel value for metering purposes, representing the current amount of gain reduction that the compressor is applying to the signal. If fed no signal the value will be 0 (no gain reduction). When this attribute is read, return the value of the private slot [[internal reduction]].

***release*, of type AudioParam, readonly**
The amount of time (in seconds) to increase the gain by 10dB.

Parameter	Value	Notes
defaultValue	.25	
minValue	0	

Error preparing HTML-CSS output (preProcess)

Parameter	Value	Notes
maxValue	1	
automationRate	"k-rate"	Has automation rate constraints

MDN

***threshold*, of type AudioParam, readonly**
The decibel value above which the compression will start taking effect.

Parameter	Value	Notes
defaultValue	-24	
minValue	-100	
maxValue	0	
automationRate	"k-rate"	Has automation rate constraints

§ 1.19.3. DynamicsCompressorOptions

This specifies the options to use in constructing a DynamicsCompressorNode. All members are optional; if not specified the normal defaults are used in constructing the node.

```
dictionary DynamicsCompressorOptions : AudioNodeOptions {
    float attack = 0.003;
    float knee = 30;
    float ratio = 12;
    float release = 0.25;
    float threshold = -24;
};
```

§ *1.19.3.1. Dictionary DynamicsCompressorOptions Members*

***attack*, of type float, defaulting to `0.003`**
The initial value for the attack AudioParam.

***knee*, of type float, defaulting to `30`**
The initial value for the knee AudioParam.

***ratio*, of type float, defaulting to `12`**
The initial value for the ratio AudioParam.

***release*, of type float, defaulting to `0.25`**
The initial value for the release AudioParam.

***threshold*, of type float, defaulting to `-24`**
The initial value for the threshold AudioParam.

§ 1.19.4. Processing

Dynamics compression can be implemented in a variety of ways. The DynamicsCompressorNode implements a dynamics processor that has the following characteristics:

MDN

- Fixed look-ahead (this means that an DynamicsCompressorNode adds a fixed latency to the signal chain).
- Configurable attack speed, release speed, threshold, knee hardness and ratio.
- Side-chaining is not supported.
- The gain reduction is reported *via* the `reduction` property on the DynamicsCompressorNode.
- The compression curve has three parts:
  - The first part is the identity: $f(x) = x$.
  - The second part is the soft-knee portion, which MUST be a monotonically increasing function.

Error preparing HTML-CSS output (preProcess)

- The third part is a linear function: $f(x) = \frac{1}{ratio} \cdot x$.

This curve MUST be continuous and piece-wise differentiable, and corresponds to a target output level, based on the input level.

Graphically, such a curve would look something like this:

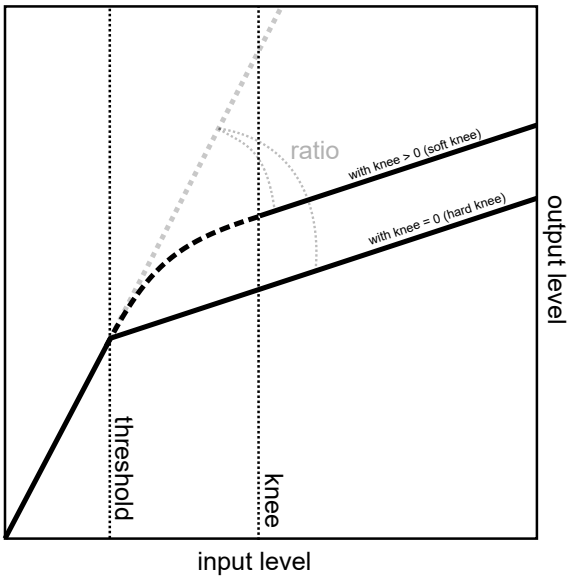


Figure 14 *A typical compression curve, showing the knee portion (soft or hard) as well as the threshold.*

Internally, the `DynamicsCompressorNode` is described with a combination of other `AudioNode`s, as well as a special algorithm, to compute the gain reduction value.

The following `AudioNode` graph is used internally, `input` and `output` respectively being the input and output `AudioNode`, `context` the `BaseAudioContext` for this `DynamicsCompressorNode`, and a new class, ***EnvelopeFollower***, that instantiates a special object that behaves like an `AudioNode`, described below:

```
const delay = new DelayNode(context, {delayTime: 0.006});
const gain = new GainNode(context);
const compression = new EnvelopeFollower();

input.connect(delay).connect(gain).connect(output);
input.connect(compression).connect(gain.gain);
```

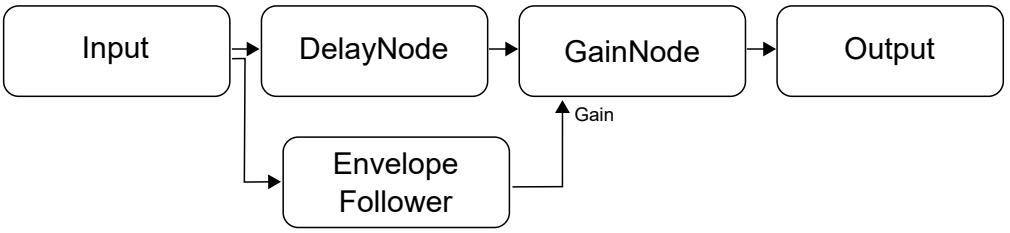


Figure 15 *The graph of internal `AudioNode`s used as part of the `DynamicsCompressorNode` processing algorithm.*

NOTE: This implements the pre-delay and the application of the reduction gain.

The following algorithm describes the processing performed by an EnvelopeFollower object, to be applied to the input signal to produce the gain reduction value. An EnvelopeFollower has two slots holding floating point values. Those values persist accros invocation of this algorithm.

- Let ***[[detector average]]*** be a floating point number, initialized to 0.0.
- Let ***[[compressor gain]]*** be a floating point number, initialized to 1.0.

Error preparing HTML-CSS output (preProcess)

The following algorithm allow determining a value for *reduction gain*, for each sample of input, for a render quantum of audio.

1. Let *attack* and *release* have the values of `attack` and `release`, respectively, sampled at the time of processing (those are k-rate parameters), mutiplied by the sample-rate of the `BaseAudioContext` this `DynamicsCompressorNode` is associated with.
2. Let *detector average* be the value of the slot `[[detector average]]`.
3. Let *compressor gain* be the value of the slot `[[compressor gain]]`.
4. For each sample *input* of the render quantum to be processed, execute the following steps:
    1. If the absolute value of *input* is less than 0.0001, let *attenuation* be 1.0. Else, let *shaped input* be the value of applying the compression curve to the absolute value of *input*. Let *attenuation* be *shaped input* divided by the absolute value of *input*.
    2. Let *releasing* be `true` if *attenuation* is greater than *compressor gain*, `false` otherwise.
    3. Let *detector rate* be the result of applying the detector curve to *attenuation*.
    4. Subtract *detector average* from *attenuation*, and multiply the result by *detector rate*. Add this new result to *detector average*.
    5. Clamp *detector average* to a maximum of 1.0.
    6. Let *envelope rate* be the result of computing the envelope rate based on values of *attack* and *release*.
    7. If *releasing* is `true`, set *compressor gain* to be the product of *compressor gain* and *envelope rate*, clamped to a maximum of 1.0.
    8. Else, if *releasing* is `false`, let *gain increment* to be *detector average* minus *compressor gain*. Multiply *gain increment* by *envelope rate*, and add the result to *compressor gain*.
    9. Compute *reduction gain* to be *compressor gain* multiplied by the return value of computing the makeup gain.
    10. Compute *metering gain* to be *reduction gain*, converted to decibel.
5. Set `[[compressor gain]]` to *compressor gain*.
6. Set `[[detector average]]` to *detector average*.
7. Atomically set the internal slot `[[internal reduction]]` to the value of *metering gain*.

> NOTE: This step makes the metering gain update once per block, at the end of the block processing.

The makeup gain is a fixed gain stage that only depends on ratio, knee and threshold parameter of the compressor, and not on the input signal. The intent here is to increase the output level of the compressor so it is comparable to the input level.

Computing the makeup gain means executing the following steps:

1. Let *full range gain* be the value returned by applying the compression curve to the value 1.0.
2. Let *full range makeup gain* be the inverse of *full range gain*.
3. Return the result of taking the 0.6 power of *full range makeup gain*.

Computing the envelope rate is done by applying a function to the ratio of the *compressor gain* and the *detector average*. User-agents are allowed to choose the shape of the envelope function. However, this function MUST respect the following constraints:

- The envelope rate MUST be the calculated from the ratio of the *compressor gain* and the *detector average*.

  > NOTE: When attacking, this number less than or equal to 1, when releasing, this number is strictly greater than 1.

- The attack curve MUST be a continuous, monotonically increasing function in the range [0, 1]. The shape of this curve MAY be controlled by `attack`.
- The release curve MUST be a continuous, monotonically decreasing function that is always greater than 1. The shape of this curve MAY be controlled by `release`.

This operation returns the value computed by applying this function to the ratio of *compressor gain* and *detector average*.

Applying the ***detector curve*** to the change rate when attacking or releasing allow implementing *adaptive release*. It is a function that MUST respect the following constraints:

- The output of the function MUST be in [0, 1].
- The function MUST be monotonically increasing, continuous.

> NOTE: It is allowed, for example, to have a compressor that performs an *adaptive release*, that is, releasing faster the harder the compression, or to have curves for attack and release that are not of the same shape.

Applying a ***compression curve*** to a value means computing the value of this sample when passed to a function, and returning the computed value. This function MUST respect the following characteristics:

1. Let *threshold* and *knee* have the values of `threshold` and `knee`, respectively, converted to linear units and sampled at the time of processing of this block (as k-rate parameters).
2. Calculate the sum of `threshold` plus `knee` also sampled at the time of processing of this block (as k-rate parameters).
3. Let *knee end threshold* have the value of this sum converted to linear units.
4. Let *ratio* have the value of the `ratio`, sampled at the time of processing of this block (as a k-rate parameter).
5. This function is the identity up to the value of the linear *threshold* (i.e., $f(x) = x$).
6. From the *threshold* up to the *knee end threshold*, User-Agents can choose the curve shape. The whole function MUST be monotonically increasing and continuous.

   > NOTE: If the *knee* is 0, the `DynamicsCompressorNode` is called a hard-knee compressor.

7. This function is linear, based on the *ratio*, after the *threshold* and the soft knee (i.e., $f(x) = \frac{1}{ratio} \cdot x$).

Converting a value v in ***linear gain unit to decibel*** means executing the following steps:

1. If v is equal to zero, return -1000.
2. Else, return $20 \log_{10} v$.

Converting a value v in ***decibels to linear gain unit*** means returning $10^{v/20}$.

§ 1.20. The `GainNode` Interface

Changing the gain of an audio signal is a fundamental operation in audio applications. This interface is an `AudioNode` with a single input and single output:

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	1	
`channelCount`	2	
`channelCountMode`	"`max`"	
`channelInterpretation`	"`speakers`"	
tail-time	No	

Each sample of each channel of the input data of the `GainNode` MUST be multiplied by the computedValue of the `gain` `AudioParam`.

```
[Exposed=Window]
interface GainNode : AudioNode {
    constructor (BaseAudioContext context, optional GainOptions options = {});
    readonly attribute AudioParam gain;
};
```

Error preparing HTML-CSS output (preProcess)

§ 1.20.1. Constructors

✓ MDN

GainNode(context, options)

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the GainNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `GainNode` will be associated with.
options	`GainOptions`	✗	✓	Optional initial parameter values for this `GainNode`.

§ 1.20.2. Attributes

***gain*, of type AudioParam, readonly**

Represents the amount of gain to apply.

Parameter	Value	Notes
`defaultValue`	1	
`minValue`	most-negative-single-float	Approximately -3.4028235e38
`maxValue`	most-positive-single-float	Approximately 3.4028235e38
`automationRate`	"`a-rate`"	

§ 1.20.3. `GainOptions`

✓ MDN

This specifies options to use in constructing a `GainNode`. All members are optional; if not specified, the normal defaults are used in constructing the node.

```
dictionary GainOptions : AudioNodeOptions {
    float gain = 1.0;
};
```

§ *1.20.3.1. Dictionary `GainOptions` Members*

***gain*, of type float, defaulting to `1.0`**

The initial gain value for the `gain` AudioParam.

§ 1.21. The `IIRFilterNode` Interface

✓ MDN

`IIRFilterNode` is an `AudioNode` processor implementing a general IIR Filter. In general, it is best to use `BiquadFilterNode`'s to implement higher-order filters for the following reasons:

- Generally less sensitive to numeric issues
- Filter parameters can be automated

✓ MDN

- Can be used to create all even-ordered IIR filters

However, odd-ordered filters cannot be created, so if such filters are needed or automation is not needed, then IIR filters may be appropriate.

Error preparing HTML-CSS output (preProcess)ents of the IIR filter cannot be changed.

Property	Value	Notes
numberOfInputs	1	
numberOfOutputs	1	
channelCount	2	
channelCountMode	"max"	
channelInterpretation	"speakers"	
tail-time	Yes	Continues to output non-silent audio with zero input. Since this is an IIR filter, the filter produces non-zero input forever, but in practice, this can be limited after some finite time where the output is sufficiently close to zero. The actual time depends on the filter coefficients.

The number of channels of the output always equals the number of channels of the input.

```
[Exposed=Window]
interface IIRFilterNode : AudioNode {
    constructor (BaseAudioContext context, IIRFilterOptions options);
    undefined getFrequencyResponse (Float32Array frequencyHz,
                                    Float32Array magResponse,
                                    Float32Array phaseResponse);
};
```

✓ MDN

§ 1.21.1. Constructors

IIRFilterNode(context, options)

When the constructor is called with a BaseAudioContext *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the IIRFilterNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	BaseAudioContext	✗	✗	The BaseAudioContext this new IIRFilterNode will be associated with.
options	IIRFilterOptions	✗	✗	Initial parameter value for this IIRFilterNode.

§ 1.21.2. Methods

getFrequencyResponse(frequencyHz, magResponse, phaseResponse)

⏳ Given the current filter parameter settings, synchronously calculates the frequency response for the specified frequencies. The three parameters MUST be Float32Arrays of the same length, or an InvalidAccessError MUST be thrown.

Arguments for the IIRFilterNode.getFrequencyResponse() method.

Parameter	Type	Nullable	Optional	Description
frequencyHz	Float32Array	✗	✗	This parameter specifies an array of frequencies, in Hz, at which the response values will be calculated.

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
magResponse	Float32Array	✗	✗	This parameter specifies an output array receiving the linear magnitude response values. If a value in the `frequencyHz` parameter is not within [0, sampleRate/2], where `sampleRate` is the value of the `sampleRate` property of the `AudioContext`, the corresponding value at the same index of the `magResponse` array MUST be `NaN`.
phaseResponse	Float32Array	✗	✗	This parameter specifies an output array receiving the phase response values in radians. If a value in the `frequencyHz` parameter is not within [0; sampleRate/2], where `sampleRate` is the value of the `sampleRate` property of the `AudioContext`, the corresponding value at the same index of the `phaseResponse` array MUST be `NaN`.

Return type: `undefined`

1.21.3. `IIRFilterOptions`

The `IIRFilterOptions` dictionary is used to specify the filter coefficients of the `IIRFilterNode`.

```
dictionary IIRFilterOptions : AudioNodeOptions {
    required sequence<double> feedforward;
    required sequence<double> feedback;
};
```

1.21.3.1. Dictionary `IIRFilterOptions` Members

***feedforward*, of type sequence<double>**
: The feedforward coefficients for the `IIRFilterNode`. This member is required. See `feedforward` argument of `createIIRFilter()` for other constraints.

***feedback*, of type sequence<double>**
: The feedback coefficients for the `IIRFilterNode`. This member is required. See `feedback` argument of `createIIRFilter()` for other constraints.

1.21.4. Filter Definition

Let b_m be the `feedforward` coefficients and a_n be the `feedback` coefficients specified by `createIIRFilter()` or the `IIRFilterOptions` dictionary for the `constructor`. Then the transfer function of the general IIR filter is given by

$$H(z) = \frac{\sum_{m=0}^{M} b_m z^{-m}}{\sum_{n=0}^{N} a_n z^{-n}}$$

where $M + 1$ is the length of the b array and $N + 1$ is the length of the a array. The coefficient a_0 MUST not be 0 (see `feedback parameter` for `createIIRFilter()`). At least one of b_m MUST be non-zero (see `feedforward parameter` for `createIIRFilter()`).

Error preparing HTML-CSS output (preProcess)

Equivalently, the time-domain equation is:

$$\sum_{k=0}^{N} a_k y(n-k) = \sum_{k=0}^{M} b_k x(n-k)$$

The initial filter state is the all-zeroes state.

NOTE: The UA may produce a warning to notify the user that NaN values have occurred in the filter state. This is usually indicative of an unstable filter.

§ 1.22. The `MediaElementAudioSourceNode` Interface

This interface represents an audio source from an `<audio>` or `<video>` element.

MDN

Property	Value	Notes
`numberOfInputs`	0	
`numberOfOutputs`	1	
tail-time reference	No	

The number of channels of the output corresponds to the number of channels of the media referenced by the `HTMLMediaElement`. Thus, changes to the media element's `src` attribute can change the number of channels output by this node.

If the sample rate of the `HTMLMediaElement` differs from the sample rate of the associated `AudioContext`, then the output from the `HTMLMediaElement` must be resampled to match the context's `sample rate`.

A `MediaElementAudioSourceNode` is created given an `HTMLMediaElement` using the `AudioContext` `createMediaElementSource()` method or the `mediaElement` member of the `MediaElementAudioSourceOptions` dictionary for the `constructor`.

The number of channels of the single output equals the number of channels of the audio referenced by the `HTMLMediaElement` passed in as the argument to `createMediaElementSource()`, or is 1 if the `HTMLMediaElement` has no audio.

The `HTMLMediaElement` MUST behave in an identical fashion after the `MediaElementAudioSourceNode` has been created, *except* that the rendered audio will no longer be heard directly, but instead will be heard as a consequence of the `MediaElementAudioSourceNode` being connected through the routing graph. Thus pausing, seeking, volume, `src` attribute changes, and other aspects of the `HTMLMediaElement` MUST behave as they normally would if *not* used with a `MediaElementAudioSourceNode`.

¶

```
EXAMPLE 11
const mediaElement = document.getElementById('mediaElementID');
const sourceNode = context.createMediaElementSource(mediaElement);
sourceNode.connect(filterNode);
```

```
[Exposed=Window]
interface MediaElementAudioSourceNode : AudioNode {
    constructor (AudioContext context, MediaElementAudioSourceOptions options);
    [SameObject] readonly attribute HTMLMediaElement mediaElement;
};
```

Error preparing HTML-CSS output (preProcess)

§ 1.22.1. Constructors

MediaElementAudioSourceNode(context, options)

1. initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the MediaElementAudioSourceNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	AudioContext	✗	✗	The AudioContext this new MediaElementAudioSourceNode will be associated with.
options	MediaElementAudioSourceOptions	✗	✗	Initial parameter value for this MediaElementAudioSourceNode.

§ 1.22.2. Attributes

***mediaElement*, of type HTMLMediaElement, readonly**
The HTMLMediaElement used when constructing this MediaElementAudioSourceNode.

§ 1.22.3. MediaElementAudioSourceOptions

This specifies the options to use in constructing a MediaElementAudioSourceNode.

```
dictionary MediaElementAudioSourceOptions {
    required HTMLMediaElement mediaElement;
};
```

§ *1.22.3.1. Dictionary MediaElementAudioSourceOptions Members*

✓ MDN

***mediaElement*, of type HTMLMediaElement**
The media element that will be re-routed. This MUST be specified.

§ 1.22.4. Security with MediaElementAudioSourceNode and Cross-Origin Resources

HTMLMediaElement allows the playback of cross-origin resources. Because Web Audio allows inspection of the content of the resource (e.g. using a MediaElementAudioSourceNode, and an AudioWorkletNode or ScriptProcessorNode to read the samples), information leakage can occur if scripts from one origin inspect the content of a resource from another origin.

To prevent this, a MediaElementAudioSourceNode MUST output *silence* instead of the normal output of the HTMLMediaElement if it has been created using an HTMLMediaElement for which the execution of the fetch algorithm [FETCH] labeled the resource as CORS-cross-origin.

§ 1.23. The MediaStreamAudioDestinationNode Interface

✓ MDN

This interface is an audio destination representing a MediaStream with a single MediaStreamTrack whose `kind` is `"audio"`. This MediaStream is created when the node is created and is accessible via the `stream` attribute. This stream can be used in a similar way as a MediaStream obtained via `getUserMedia()`, and can, for example, be sent to a remote peer using the `RTCPeerConnection` (described in [webrtc]) `addStream()` method.

Property	Value	Notes
puts	1	

Error preparing HTML-CSS output (preProcess)

Property	Value	Notes
numberOfOutputs	0	
channelCount	2	
channelCountMode	"explicit"	
channelInterpretation	"speakers"	
tail-time	No	

MDN

The number of channels of the input is by default 2 (stereo).

```
[Exposed=Window]
interface MediaStreamAudioDestinationNode : AudioNode {
    constructor (AudioContext context, optional AudioNodeOptions options = {});
    readonly attribute MediaStream stream;
};
```

§ 1.23.1. Constructors

MediaStreamAudioDestinationNode(context, options)

1. Initialize the AudioNode *this*, with *context* and *options* as arguments.

MDN

Arguments for the MediaStreamAudioDestinationNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	AudioContext	✗	✗	The BaseAudioContext this new MediaStreamAudioDestinationNode will be associated with.
options	AudioNodeOptions	✗	✓	Optional initial parameter value for this MediaStreamAudioDestinationNode.

MDN

§ 1.23.2. Attributes

***stream*, of type MediaStream, readonly**

A MediaStream containing a single MediaStreamTrack with the same number of channels as the node itself, and whose kind attribute has the value "audio".

§ 1.24. The MediaStreamAudioSourceNode Interface

This interface represents an audio source from a MediaStream.

Property	Value	Notes
numberOfInputs	0	
numberOfOutputs	1	
tail-time reference	No	

The number of channels of the output corresponds to the number of channels of the MediaStreamTrack. When the MediaStreamTrack ends, this AudioNode outputs one channel of silence.

Error preparing HTML-CSS output (preProcess)

If the sample rate of the MediaStreamTrack differs from the sample rate of the associated AudioContext, then the output of the MediaStreamTrack is resampled to match the context's sample rate.

```
[Exposed=Window]
interface MediaStreamAudioSourceNode : AudioNode {
    constructor (AudioContext context, MediaStreamAudioSourceOptions options);
    [SameObject] readonly attribute MediaStream mediaStream;
};
```

§ 1.24.1. Constructors

MediaStreamAudioSourceNode(context, options)

1. If the mediaStream member of options does not reference a MediaStream that has at least one MediaStreamTrack whose kind attribute has the value "audio", throw an InvalidStateError and abort these steps. Else, let this stream be *inputStream*.
2. Let *tracks* be the list of all MediaStreamTracks of *inputStream* that have a kind of "audio".
3. Sort the elements in *tracks* based on their id attribute using an ordering on sequences of code unit values.
4. Initialize the AudioNode *this*, with *context* and *options* as arguments.
5. Set an internal slot ***[[input track]]*** on this MediaStreamAudioSourceNode to be the first element of *tracks*. This is the track used as the input audio for this MediaStreamAudioSourceNode.

After construction, any change to the MediaStream that was passed to the constructor do not affect the underlying output of this AudioNode.

The slot [[input track]] is only used to keep a reference to the MediaStreamTrack.

> NOTE: This means that when removing the track chosen by the constructor of the MediaStreamAudioSourceNode from the MediaStream passed into this constructor, the MediaStreamAudioSourceNode will still take its input from the same track.

> NOTE: The behaviour for picking the track to output is arbitrary for legacy reasons. MediaStreamTrackAudioSourceNode can be used instead to be explicit about which track to use as input.

Arguments for the MediaStreamAudioSourceNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	AudioContext	✗	✗	The AudioContext this new MediaStreamAudioSourceNode will be associated with.
options	MediaStreamAudioSourceOptions	✗	✗	Initial parameter value for this MediaStreamAudioSourceNode.

§ 1.24.2. Attributes

***mediaStream*, of type MediaStream, readonly**
The MediaStream used when constructing this MediaStreamAudioSourceNode.

§ 1.24.3. MediaStreamAudioSourceOptions

✓ MDN

Error preparing HTML-CSS output (preProcess)

This specifies the options for constructing a MediaStreamAudioSourceNode.

```
dictionary MediaStreamAudioSourceOptions {
    required MediaStream mediaStream;
};
```

§ *1.24.3.1. Dictionary `MediaStreamAudioSourceOptions` Members*

***`mediaStream`*, of type MediaStream**
The media stream that will act as a source. This MUST be specified.

§ 1.25. The `MediaStreamTrackAudioSourceNode` Interface

This interface represents an audio source from a MediaStreamTrack.

Property	Value	Notes
numberOfInputs	0	
numberOfOutputs	1	
tail-time reference	No	

The number of channels of the output corresponds to the number of channels of the mediaStreamTrack.

If the sample rate of the MediaStreamTrack differs from the sample rate of the associated AudioContext, then the output of the mediaStreamTrack is resampled to match the context's sample rate.

```
[Exposed=Window]
interface MediaStreamTrackAudioSourceNode : AudioNode {
    constructor (AudioContext context, MediaStreamTrackAudioSourceOptions options);
};
```

§ 1.25.1. Constructors

`MediaStreamTrackAudioSourceNode(context, options)`

1. If the mediaStreamTrack's `kind` attribute is not `"audio"`, throw an InvalidStateError and abort these steps.
2. Initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the MediaStreamTrackAudioSourceNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	AudioContext	✗	✗	The AudioContext this new MediaStreamTrackAudioSourc will be associated with.
options	MediaStreamTrackAudioSourceOptions	✗	✗	Initial parameter value for this MediaStreamTrackAudioSourc

§ 1.25.2. `MediaStreamTrackAudioSourceOptions`

Error preparing HTML-CSS output (preProcess)

This specifies the options for constructing a MediaStreamTrackAudioSourceNode. This is required.

```
dictionary MediaStreamTrackAudioSourceOptions {
    required MediaStreamTrack mediaStreamTrack;
};
```

1.25.2.1. Dictionary *MediaStreamTrackAudioSourceOptions* Members

***mediaStreamTrack*, of type MediaStreamTrack**
The media stream track that will act as a source. If this MediaStreamTrack `kind` attribute is not `"audio"`, an InvalidStateError MUST be thrown.

1.26. The OscillatorNode Interface

OscillatorNode represents an audio source generating a periodic waveform. It can be set to a few commonly used waveforms. Additionally, it can be set to an arbitrary periodic waveform through the use of a PeriodicWave object.

Oscillators are common foundational building blocks in audio synthesis. An OscillatorNode will start emitting sound at the time specified by the `start()` method.

Mathematically speaking, a *continuous-time* periodic waveform can have very high (or infinitely high) frequency information when considered in the frequency domain. When this waveform is sampled as a discrete-time digital audio signal at a particular sample-rate, then care MUST be taken to discard (filter out) the high-frequency information higher than the Nyquist frequency before converting the waveform to a digital form. If this is not done, then *aliasing* of higher frequencies (than the Nyquist frequency) will fold back as mirror images into frequencies lower than the Nyquist frequency. In many cases this will cause audibly objectionable artifacts. This is a basic and well-understood principle of audio DSP.

There are several practical approaches that an implementation may take to avoid this aliasing. Regardless of approach, the *idealized* discrete-time digital audio signal is well defined mathematically. The trade-off for the implementation is a matter of implementation cost (in terms of CPU usage) versus fidelity to achieving this ideal.

It is expected that an implementation will take some care in achieving this ideal, but it is reasonable to consider lower-quality, less-costly approaches on lower-end hardware.

Both `frequency` and `detune` are a-rate parameters, and form a compound parameter. They are used together to determine a ***computedOscFrequency*** value:

```
computedOscFrequency(t) = frequency(t) * pow(2, detune(t) / 1200)
```

The OscillatorNode's instantaneous phase at each time is the definite time integral of computedOscFrequency, assuming a phase angle of zero at the node's exact start time. Its nominal range is [-Nyquist frequency, Nyquist frequency].

The single output of this node consists of one channel (mono).

Property	Value	Notes
`numberOfInputs`	0	
`numberOfOutputs`	1	
`channelCount`	2	
`channelCountMode`	`"max"`	
`channelInterpretation`	`"speakers"`	
tail-time	No	

```
enum OscillatorType {
    "sine",
    "square",
```

Error preparing HTML-CSS output (preProcess)

```
    "triangle",
    "custom"
};
```

OscillatorType enumeration description

Enum value	Description
"***sine***"	A sine wave
"***square***"	A square wave of duty period 0.5
"***sawtooth***"	A sawtooth wave
"***triangle***"	A triangle wave
"***custom***"	A custom periodic wave

```
[Exposed=Window]
interface OscillatorNode : AudioScheduledSourceNode {
    constructor (BaseAudioContext context, optional OscillatorOptions options = {});
    attribute OscillatorType type;
    readonly attribute AudioParam frequency;
    readonly attribute AudioParam detune;
    undefined setPeriodicWave (PeriodicWave periodicWave);
};
```

§ 1.26.1. Constructors

OscillatorNode(context, options)

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the OscillatorNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `OscillatorNode` will be associated with.
options	`OscillatorOptions`	✗	✓	Optional initial parameter value for this `OscillatorNode`.

§ 1.26.2. Attributes

***detune*, of type AudioParam, readonly**

A detuning value (in cents) which will offset the `frequency` by the given amount. Its default `value` is 0. This parameter is a-rate. It forms a compound parameter with `frequency` to form the computedOscFrequency. The nominal range listed below allows this parameter to detune the `frequency` over the entire possible range of frequencies.

Parameter	Value	Notes
`defaultValue`	0	
`minValue`	≈ − 153600	
`maxValue`	≈ 153600	This value is approximately 1200 $\log_2$FLT_MAX where FLT_MAX is the largest `float` value.
`ionRate`	"`a-rate`"	

Error preparing HTML-CSS output (preProcess)

***frequency*, of type AudioParam, readonly**

The frequency (in Hertz) of the periodic waveform. Its default `value` is 440. This parameter is a-rate. It forms a compound parameter with `detune` to form the computedOscFrequency. Its nominal range is [-Nyquist frequency, Nyquist frequency].

Parameter	Value	Notes
`defaultValue`	440	
`minValue`	-Nyquist frequency	
`maxValue`	Nyquist frequency	
`automationRate`	"`a-rate`"	

✓ MDN

***type*, of type OscillatorType**

The shape of the periodic waveform. It may directly be set to any of the type constant values except for "`custom`". ⌛ Doing so MUST throw an `InvalidStateError` exception. The `setPeriodicWave()` method can be used to set a custom waveform, which results in this attribute being set to "`custom`". The default value is "`sine`". When this attribute is set, the phase of the oscillator MUST be conserved.

§ 1.26.3. Methods

✓ MDN

setPeriodicWave(periodicWave)

✓ MDN

Sets an arbitrary custom periodic waveform given a `PeriodicWave`.

Arguments for the OscillatorNode.setPeriodicWave() method.

Parameter	Type	Nullable	Optional	Description
periodicWave	`PeriodicWave`	✗	✗	custom waveform to be used by the oscillator

Return type: `undefined`

✓ MDN

§ 1.26.4. `OscillatorOptions`

This specifies the options to be used when constructing an `OscillatorNode`. All of the members are optional; if not specified, the normal default values are used for constructing the oscillator.

```
dictionary OscillatorOptions : AudioNodeOptions {
    OscillatorType type = "sine";
    float frequency = 440;
    float detune = 0;
    PeriodicWave periodicWave;
};
```

✓ MDN

§ *1.26.4.1. Dictionary `OscillatorOptions` Members*

***detune*, of type float, defaulting to `0`**

The initial detune value for the `OscillatorNode`.

***frequency*, of type float, defaulting to `440`**

The initial frequency for the `OscillatorNode`.

***periodicWave*, of type PeriodicWave**

The `PeriodicWave` for the `OscillatorNode`. If this is specified, then any valid value for `type` is ignored; it is treated as if "`custom`" were specified.

***type*, of type OscillatorType, defaulting to `"sine"`**

✓ MDN

Error preparing HTML-CSS output (preProcess)

r to be constructed. If this is set to "custom" without also specifying a `periodicWave`, then an ⌛ `InvalidStateError` exception MUST be thrown. If `periodicWave` is specified, then any valid value for `type` is

ignored; it is treated as if it were set to "custom".

§ 1.26.5. Basic Waveform Phase

The idealized mathematical waveforms for the various oscillator types are defined below. In summary, all waveforms are defined mathematically to be an odd function with a positive slope at time 0. The actual waveforms produced by the oscillator may differ to prevent aliasing affects.

The oscillator MUST produce the same result as if a `PeriodicWave`, with the appropriate Fourier series and with `disableNormalization` set to false, were used to create these basic waveforms.

"`sine`"
The waveform for sine oscillator is:

$$x(t) = \sin t$$

"`square`"
The waveform for the square wave oscillator is:

$$x(t) = \begin{cases} 1 & \text{for } 0 \le t < \pi \\ -1 & \text{for } -\pi < t < 0. \end{cases}$$

This is extended to all t by using the fact that the waveform is an odd function with period 2π.

"`sawtooth`"
The waveform for the sawtooth oscillator is the ramp:

$$x(t) = \frac{t}{\pi} \text{ for } -\pi < t \le \pi;$$

This is extended to all t by using the fact that the waveform is an odd function with period 2π.

"`triangle`"
The waveform for the triangle oscillator is:

$$x(t) = \begin{cases} \frac{2}{\pi} t & \text{for } 0 \le t \le \frac{\pi}{2} \\ 1 - \frac{2}{\pi}\left(t - \frac{\pi}{2}\right) & \text{for } \frac{\pi}{2} < t \le \pi. \end{cases}$$

This is extended to all t by using the fact that the waveform is an odd function with period 2π.

§ 1.27. The `PannerNode` Interface

This interface represents a processing node which positions / spatializes an incoming audio stream in three-dimensional space. The spatialization is in relation to the `BaseAudioContext`'s `AudioListener` (`listener` attribute).

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	1	
`channelCount`	2	Has channelCount constraints
`channelCountMode`	"`clamped-max`"	Has channelCountMode constraints
`channelInterpretation`	"`speakers`"	

Property	Value	Notes
tail-time	Maybe	If the panningModel is set to "HRTF", the node will produce non-silent output for silent input due to the inherent processing for head responses. Otherwise the tail-time is zero.

The input of this node is either mono (1 channel) or stereo (2 channels) and cannot be increased. Connections from nodes with fewer or more channels will be up-mixed or down-mixed appropriately.

If the node is actively processing, the output of this node is hard-coded to stereo (2 channels) and cannot be configured. If the node is not actively processing, then the output is a single channel of silence.

The PanningModelType enum determines which spatialization algorithm will be used to position the audio in 3D space. The default is "equalpower".

```
enum PanningModelType {
        "equalpower",
        "HRTF"
};
```

PanningModelType enumeration description

Enum value	Description
"equalpower"	A simple and efficient spatialization algorithm using equal-power panning. NOTE: When this panning model is used, all the AudioParams used to compute the output of this node are a-rate.
"HRTF"	A higher quality spatialization algorithm using a convolution with measured impulse responses from human subjects. This panning method renders stereo output. NOTE: When this panning model is used, all the AudioParams used to compute the output of this node are k-rate.

The ***effective automation rate*** for an AudioParam of a PannerNode is determined by the panningModel and automationRate of the AudioParam. If the panningModel is "HRTF", the effective automation rate is "k-rate", independent of the setting of the automationRate. Otherwise the effective automation rate is the value of automationRate.

The DistanceModelType enum determines which algorithm will be used to reduce the volume of an audio source as it moves away from the listener. The default is "inverse".

In the description of each distance model below, let d be the distance between the listener and the panner; d_{ref} be the value of the refDistance attribute; d_{max} be the value of the maxDistance attribute; and f be the value of the rolloffFactor attribute.

```
enum DistanceModelType {
    "linear",
    "inverse",
    "exponential"
};
```

DistanceModelType enumeration description

Enum value	Description

Error preparing HTML-CSS output (preProcess)

Enum value	Description
"*linear*"	A linear distance model which calculates *distanceGain* according to: $$1 - f\frac{\max\left[\min\left(d, d'_{max}\right), d'_{ref}\right] - d'_{ref}}{d'_{max} - d'_{ref}}$$ where $d'_{ref} = \min\left(d_{ref}, d_{max}\right)$ and $d'_{max} = \max\left(d_{ref}, d_{max}\right)$. In the case where $d'_{ref} = d'_{max}$, the value of the linear model is taken to be $1 - f$. Note that d is clamped to the interval $\left[d'_{ref}, d'_{max}\right]$.
"*inverse*"	An inverse distance model which calculates *distanceGain* according to: $$\frac{d_{ref}}{d_{ref} + f\left[\max\left(d, d_{ref}\right) - d_{ref}\right]}$$ That is, d is clamped to the interval $\left[d_{ref}, \infty\right)$. If $d_{ref} = 0$, the value of the inverse model is taken to be 0, independent of the value of d and f.
"*exponential*"	An exponential distance model which calculates *distanceGain* according to: $$\left[\frac{\max\left(d, d_{ref}\right)}{d_{ref}}\right]^{-f}$$ That is, d is clamped to the interval $\left[d_{ref}, \infty\right)$. If $d_{ref} = 0$, the value of the exponential model is taken to be 0, independent of d and f.

MDN

MDN

```
[Exposed=Window]
interface PannerNode : AudioNode {
    constructor (BaseAudioContext context, optional PannerOptions options = {});
    attribute PanningModelType panningModel;
    readonly attribute AudioParam positionX;
    readonly attribute AudioParam positionY;
    readonly attribute AudioParam positionZ;
    readonly attribute AudioParam orientationX;
    readonly attribute AudioParam orientationY;
    readonly attribute AudioParam orientationZ;
    attribute DistanceModelType distanceModel;
    attribute double refDistance;
    attribute double maxDistance;
    attribute double rolloffFactor;
    attribute double coneInnerAngle;
    attribute double coneOuterAngle;
    attribute double coneOuterGain;
    undefined setPosition (float x, float y, float z);
    undefined setOrientation (float x, float y, float z);
};
```

Error preparing HTML-CSS output (preProcess)

§ 1.27.1. Constructors

PannerNode(context, options)

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the PannerNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `PannerNode` will be associated with.
options	`PannerOptions`	✗	✓	Optional initial parameter value for this `PannerNode`.

§ 1.27.2. Attributes

***coneInnerAngle*, of type double**

A parameter for directional audio sources that is an angle, in degrees, inside of which there will be no volume reduction. The default value is 360. The behavior is undefined if the angle is outside the interval [0, 360].

***coneOuterAngle*, of type double**

A parameter for directional audio sources that is an angle, in degrees, outside of which the volume will be reduced to a constant value of `coneOuterGain`. The default value is 360. The behavior is undefined if the angle is outside the interval [0, 360].

***coneOuterGain*, of type double**

A parameter for directional audio sources that is the gain outside of the `coneOuterAngle`. The default value is 0. It is a linear value (not dB) in the range [0, 1]. An `InvalidStateError` MUST be thrown if the parameter is outside this range.

***distanceModel*, of type DistanceModelType**

Specifies the distance model used by this `PannerNode`. Defaults to "`inverse`".

***maxDistance*, of type double**

The maximum distance between source and listener, after which the volume will not be reduced any further. The default value is 10000. A `RangeError` exception MUST be thrown if this is set to a non-positive value.

***orientationX*, of type AudioParam, readonly**

Describes the *x*-component of the vector of the direction the audio source is pointing in 3D Cartesian coordinate space.

Parameter	Value	Notes
`defaultValue`	1	
`minValue`	most-negative-single-float	Approximately -3.4028235e38
`maxValue`	most-positive-single-float	Approximately 3.4028235e38
`automationRate`	"`a-rate`"	Has automation rate constraints

***orientationY*, of type AudioParam, readonly**

Describes the *y*-component of the vector of the direction the audio source is pointing in 3D cartesian coordinate space.

Parameter	Value	Notes
`defaultValue`	0	
`minValue`	most-negative-single-float	Approximately -3.4028235e38
`maxValue`	most-positive-single-float	Approximately 3.4028235e38
`automationRate`	"`a-rate`"	Has automation rate constraints

Error preparing HTML-CSS output (preProcess)

***orientationZ*, of type AudioParam, readonly**
Describes the *z*-component of the vector of the direction the audio source is pointing in 3D cartesian coordinate space.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	Has automation rate constraints

***panningModel*, of type PanningModelType**
Specifies the panning model used by this PannerNode. Defaults to "equalpower".

***positionX*, of type AudioParam, readonly**
Sets the *x*-coordinate position of the audio source in a 3D Cartesian system.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	Has automation rate constraints

***positionY*, of type AudioParam, readonly**
Sets the *y*-coordinate position of the audio source in a 3D Cartesian system.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	Has automation rate constraints

***positionZ*, of type AudioParam, readonly**
Sets the *z*-coordinate position of the audio source in a 3D Cartesian system.

Parameter	Value	Notes
defaultValue	0	
minValue	most-negative-single-float	Approximately -3.4028235e38
maxValue	most-positive-single-float	Approximately 3.4028235e38
automationRate	"a-rate"	Has automation rate constraints

***refDistance*, of type double**
A reference distance for reducing volume as source moves further from the listener. For distances less than this, the volume is not reduced. The default value is 1. ⏳ A RangeError exception MUST be thrown if this is set to a negative value.

***rolloffFactor*, of type double**
Describes how quickly the volume is reduced as source moves away from listener. The default value is 1. ⏳ A RangeError exception MUST be thrown if this is set to a negative value.

The nominal range for the rolloffFactor specifies the minimum and maximum values the rolloffFactor can have. Values outside the range are clamped to lie within this range. The nominal range depends on the distanceModel as follows:

Error preparing HTML-CSS output (preProcess)

"linear"
The nominal range is [0, 1].

"inverse"
The nominal range is [0, ∞).

"exponential"
The nominal range is [0, ∞).

Note that the clamping happens as part of the processing of the distance computation. The attribute reflects the value that was set and is not modified.

§ 1.27.3. Methods

setOrientation(x, y, z)

This method is DEPRECATED. It is equivalent to setting orientationX.value, orientationY.value, and orientationZ.value attribute directly, with the x, y and z parameters, respectively.

⌛ Consequently, if any of the orientationX, orientationY, and orientationZ AudioParams have an automation curve set using setValueCurveAtTime() at the time this method is called, a NotSupportedError MUST be thrown.

Describes which direction the audio source is pointing in the 3D cartesian coordinate space. Depending on how directional the sound is (controlled by the **cone** attributes), a sound pointing away from the listener can be very quiet or completely silent.

The x, y, z parameters represent a direction vector in 3D space.

The default value is (1,0,0).

Arguments for the PannerNode.setOrientation() method.

Parameter	Type	Nullable	Optional	Description
x	float	✗	✗	
y	float	✗	✗	
z	float	✗	✗	

Return type: undefined

setPosition(x, y, z)

This method is DEPRECATED. It is equivalent to setting positionX.value, positionY.value, and positionZ.value attribute directly with the x, y and z parameters, respectively.

⌛ Consequently, if any of the positionX, positionY, and positionZ AudioParams have an automation curve set using setValueCurveAtTime() at the time this method is called, a NotSupportedError MUST be thrown.

Sets the position of the audio source relative to the listener attribute. A 3D cartesian coordinate system is used.

The x, y, z parameters represent the coordinates in 3D space.

The default value is (0,0,0).

Arguments for the PannerNode.setPosition() method.

Parameter	Type	Nullable	Optional	Description
x	float	✗	✗	
y	float	✗	✗	
z	float	✗	✗	

Error preparing HTML-CSS output (preProcess)

Return type: undefined

§ 1.27.4. PannerOptions

This specifies options for constructing a PannerNode. All members are optional; if not specified, the normal default is used in constructing the node.

```
dictionary PannerOptions : AudioNodeOptions {
    PanningModelType panningModel = "equalpower";
    DistanceModelType distanceModel = "inverse";
    float positionX = 0;
    float positionY = 0;
    float positionZ = 0;
    float orientationX = 1;
    float orientationY = 0;
    float orientationZ = 0;
    double refDistance = 1;
    double maxDistance = 10000;
    double rolloffFactor = 1;
    double coneInnerAngle = 360;
    double coneOuterAngle = 360;
    double coneOuterGain = 0;
};
```

✓ MDN

§ *1.27.4.1. Dictionary PannerOptions Members*

***coneInnerAngle*, of type double, defaulting to `360`**
The initial value for the coneInnerAngle attribute of the node.

***coneOuterAngle*, of type double, defaulting to `360`**
The initial value for the coneOuterAngle attribute of the node.

***coneOuterGain*, of type double, defaulting to `0`**
The initial value for the coneOuterGain attribute of the node.

***distanceModel*, of type DistanceModelType, defaulting to `"inverse"`**
The distance model to use for the node.

***maxDistance*, of type double, defaulting to `10000`**
The initial value for the maxDistance attribute of the node.

***orientationX*, of type float, defaulting to `1`**
The initial *x*-component value for the orientationX AudioParam.

***orientationY*, of type float, defaulting to `0`**
The initial *y*-component value for the orientationY AudioParam.

***orientationZ*, of type float, defaulting to `0`**
The initial *z*-component value for the orientationZ AudioParam.

***panningModel*, of type PanningModelType, defaulting to `"equalpower"`**
The panning model to use for the node.

✓ MDN

***positionX*, of type float, defaulting to `0`**
The initial *x*-coordinate value for the positionX AudioParam.

***positionY*, of type float, defaulting to `0`**
The initial *y*-coordinate value for the positionY AudioParam.

***positionZ*, of type float, defaulting to `0`**
The initial *z*-coordinate value for the positionZ AudioParam.

***refDistance*, of type double, defaulting to `1`**
The initial value for the refDistance attribute of the node.

***rolloffFactor*, of type double, defaulting to `1`**
The initial value for the rolloffFactor attribute of the node.

§ 1.27.5. Channel Limitations

The set of channel limitations for StereoPannerNode also apply to PannerNode.

Error preparing HTML-CSS output (preProcess)

§ 1.28. The PeriodicWave Interface

✓ MDN

PeriodicWave represents an arbitrary periodic waveform to be used with an OscillatorNode.

A conforming implementation MUST support PeriodicWave up to at least 8192 elements.

```
[Exposed=Window]
interface PeriodicWave {
    constructor (BaseAudioContext context, optional PeriodicWaveOptions options = {});
};
```

§ 1.28.1. Constructors

PeriodicWave(context, options)

1. Let *p* be a new PeriodicWave object. Let ***[[real]]*** and ***[[imag]]*** be two internal slots of type Float32Array, and let ***[[normalize]]*** be an internal slot.
2. Process options according to one of the following cases:
    1. If both options.real and options.imag are present
        1. ⌛ If the lengths of options.real and options.imag are different or if either length is less than 2, throw an IndexSizeError and abort this algorithm.
        2. Set [[real]] and [[imag]] to new arrays with the same length as options.real.
        3. Copy all elements from options.real to [[real]] and options.imag to [[imag]].
    2. If only options.real is present
        1. ⌛ If length of options.real is less than 2, throw an IndexSizeError and abort this algorithm.
        2. Set [[real]] and [[imag]] to arrays with the same length as options.real.
        3. Copy options.real to [[real]] and set [[imag]] to all zeros.
    3. If only options.imag is present
        1. ⌛ If length of options.imag is less than 2, throw an IndexSizeError and abort this algorithm.
        2. Set [[real]] and [[imag]] to arrays with the same length as options.imag.
        3. Copy options.imag to [[imag]] and set [[real]] to all zeros.

    ✓ MDN

    4. Otherwise
        1. Set [[real]] and [[imag]] to zero-filled arrays of length 2.
        2. Set element at index 1 of [[imag]] to 1.

    NOTE: When setting this PeriodicWave on an OscillatorNode, this is equivalent to using the built-in type "sine".
3. Set element at index 0 of both [[real]] and [[imag]] to 0. (This sets the DC component to 0.)
4. Initialize [[normalize]] to the inverse of the disableNormalization attribute of the PeriodicWaveConstraints on the PeriodicWaveOptions.
5. Return *p*.

Arguments for the PeriodicWave.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	BaseAudioContext	✗	✗	The BaseAudioContext this new PeriodicWave will be associated with. Unlike AudioBuffer, PeriodicWaves can't be shared accross AudioContexts or

Error preparing HTML-CSS output (preProcess)

Parameter	Type	Nullable	Optional	Description
				`OfflineAudioContext`s. It is associated with a particular `BaseAudioContext`.
options	`PeriodicWaveOptions`	✗	✓	Optional initial parameter value for this `PeriodicWave`.

§ 1.28.2. `PeriodicWaveConstraints`

The `PeriodicWaveConstraints` dictionary is used to specify how the waveform is normalized.

```
dictionary PeriodicWaveConstraints {
    boolean disableNormalization = false;
};
```

§ *1.28.2.1. Dictionary `PeriodicWaveConstraints` Members*

***`disableNormalization`*, of type boolean, defaulting to `false`**
: Controls whether the periodic wave is normalized or not. If `true`, the waveform is not normalized; otherwise, the waveform is normalized.

✓ MDN

§ 1.28.3. `PeriodicWaveOptions`

The `PeriodicWaveOptions` dictionary is used to specify how the waveform is constructed. If only one of `real` or `imag` is specified. The other is treated as if it were an array of all zeroes of the same length, as specified below in description of the dictionary members. If neither is given, a `PeriodicWave` is created that MUST be equivalent to an `OscillatorNode` with `type` "`sine`". If both are given, the sequences must have the same length; otherwise an ⌛ error of type `NotSupportedError` MUST be thrown.

```
dictionary PeriodicWaveOptions : PeriodicWaveConstraints {
    sequence<float> real;
    sequence<float> imag;
};
```

§ *1.28.3.1. Dictionary `PeriodicWaveOptions` Members*

***`imag`*, of type sequence<float>**
: The `imag` parameter represents an array of `sine` terms. The first element (index 0) does not exist in the Fourier series. The second element (index 1) represents the fundamental frequency. The third represents the first overtone and so on.

***`real`*, of type sequence<float>**
: The `real` parameter represents an array of `cosine` terms. The first element (index 0) is the DC-offset of the periodic waveform. The second element (index 1) represents the fundmental frequency. The third represents the first overtone and so on.

✓ MDN

Error preparing HTML-CSS output (preProcess)

§ 1.28.4. Waveform Generation

The `createPeriodicWave()` method takes two arrays to specify the Fourier coefficients of the `PeriodicWave`. Let a and b represent the `[[real]]` and `[[imag]]` arrays of length L, respectively. Then the basic time-domain waveform, $x(t)$, can be computed using:

$$x(t) = \sum_{k=1}^{L-1} [a[k]\cos 2\pi kt + b[k]\sin 2\pi kt]$$

This is the basic (unnormalized) waveform.

§ 1.28.5. Waveform Normalization

If the internal slot `[[normalize]]` of this `PeriodicWave` is `true` (the default), the waveform defined in the previous section is normalized so that the maximum value is 1. The normalization is done as follows.

Let

$$\tilde{x}(n) = \sum_{k=1}^{L-1} \left(a[k]\cos\frac{2\pi kn}{N} + b[k]\sin\frac{2\pi kn}{N} \right)$$

where N is a power of two. (Note: $\tilde{x}(n)$ can conveniently be computed using an inverse FFT.) The fixed normalization factor f is computed as follows.

$$f = \max_{n=0,\ \ldots\ ,N-1} |\tilde{x}(n)|$$

Thus, the actual normalized waveform $\hat{x}(n)$ is:

$$\hat{x}(n) = \frac{\tilde{x}(n)}{f}$$

This fixed normalization factor MUST be applied to all generated waveforms.

✓ MDN

§ 1.28.6. Oscillator Coefficients

The builtin oscillator types are created using `PeriodicWave` objects. For completeness the coefficients for the `PeriodicWave` for each of the builtin oscillator types is given here. This is useful if a builtin type is desired but without the default normalization.

In the following descriptions, let a be the array of real coefficients and b be the array of imaginary coefficients for `createPeriodicWave()`. In all cases $a[n] = 0$ for all n because the waveforms are odd functions. Also, $b[0] = 0$ in all cases. Hence, only $b[n]$ for $n \geq 1$ is specified below.

"`sine`"

$$b[n] = \begin{cases} 1 & \text{for } n = 1 \\ 0 & \text{otherwise} \end{cases}$$

"`square`"

$$b[n] = \frac{2}{n\pi}\left[1 - (-1)^n\right]$$

Error preparing HTML-CSS output (preProcess)

"sawtooth"

$$b[n] = (-1)^{n+1}\frac{2}{n\pi}$$

"triangle"

$$b[n] = \frac{8\sin\frac{n\pi}{2}}{(\pi n)^2}$$

§ 1.29. The ScriptProcessorNode Interface - DEPRECATED

This interface is an AudioNode which can generate, process, or analyse audio directly using a script. This node type is deprecated, to be replaced by the AudioWorkletNode; this text is only here for informative purposes until implementations remove this node type.

Property	Value	Notes
numberOfInputs	1	
numberOfOutputs	1	
channelCount	numberOfInputChannels	This is the number of channels specified when constructing this node. There are channelCount constraints.
channelCountMode	"explicit"	Has channelCountMode constraints
channelInterpretation	"speakers"	
tail-time	No	

✓ MDN

The ScriptProcessorNode is constructed with a bufferSize which MUST be one of the following values: 256, 512, 1024, 2048, 4096, 8192, 16384. This value controls how frequently the audioprocess event is dispatched and how many sample-frames need to be processed each call. audioprocess events are only dispatched if the ScriptProcessorNode has at least one input or one output connected. Lower numbers for bufferSize will result in a lower (better) latency. Higher numbers will be necessary to avoid audio breakup and glitches. This value will be picked by the implementation if the bufferSize argument to createScriptProcessor() is not passed in, or is set to 0.

numberOfInputChannels and numberOfOutputChannels determine the number of input and output channels. It is invalid for both numberOfInputChannels and numberOfOutputChannels to be zero.

```
[Exposed=Window]
interface ScriptProcessorNode : AudioNode {
    attribute EventHandler onaudioprocess;
    readonly attribute long bufferSize;
};
```

§ 1.29.1. Attributes

***bufferSize*, of type long, readonly**

The size of the buffer (in sample-frames) which needs to be processed each time audioprocess is fired. Legal values are (256, 512, 1024, 2048, 4096, 8192, 16384).

***onaudioprocess*, of type EventHandler**

A property used to set an event handler for the ***audioprocess*** event type that is dispatched to ScriptProcessorNode node types. The event dispatched to the event handler uses the AudioProcessingEvent interface.

Error preparing HTML-CSS output (preProcess)

§ 1.30. The `StereoPannerNode` Interface

This interface represents a processing node which positions an incoming audio stream in a stereo image using a low-cost panning algorithm. This panning effect is common in positioning audio components in a stereo stream.

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	1	
`channelCount`	2	Has channelCount constraints
`channelCountMode`	"`clamped-max`"	Has channelCountMode constraints
`channelInterpretation`	"`speakers`"	
tail-time	No	

The input of this node is stereo (2 channels) and cannot be increased. Connections from nodes with fewer or more channels will be up-mixed or down-mixed appropriately.

The output of this node is hard-coded to stereo (2 channels) and cannot be configured.

```
[Exposed=Window]
interface StereoPannerNode : AudioNode {
    constructor (BaseAudioContext context, optional StereoPannerOptions options = {});
    readonly attribute AudioParam pan;
};
```

§ 1.30.1. Constructors

`StereoPannerNode(context, options)`

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Arguments for the StereoPannerNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
`context`	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `StereoPannerNode` will be associated with.
`options`	`StereoPannerOptions`	✗	✓	Optional initial parameter value for this `StereoPannerNode`.

§ 1.30.2. Attributes

***pan*, of type AudioParam, readonly**

The position of the input in the output's stereo image. -1 represents full left, +1 represents full right.

Parameter	Value	Notes
`defaultValue`	0	
`minValue`	-1	
`maxValue`	1	
`automationRate`	"`a-rate`"	

✓ MDN

Error preparing HTML-CSS output (preProcess)

1.30.3. `StereoPannerOptions`

This specifies the options to use in constructing a `StereoPannerNode`. All members are optional; if not specified, the normal default is used in constructing the node.

```
dictionary StereoPannerOptions : AudioNodeOptions {
    float pan = 0;
};
```

1.30.3.1. Dictionary `StereoPannerOptions` Members

***pan*, of type float, defaulting to `0`**
: The initial value for the `pan` AudioParam.

1.30.4. Channel Limitations

Because its processing is constrained by the above definitions, `StereoPannerNode` is limited to mixing no more than 2 channels of audio, and producing exactly 2 channels. It is possible to use a `ChannelSplitterNode`, intermediate processing by a subgraph of `GainNode`s and/or other nodes, and recombination via a `ChannelMergerNode` to realize arbitrary approaches to panning and mixing.

1.31. The `WaveShaperNode` Interface

`WaveShaperNode` is an `AudioNode` processor implementing non-linear distortion effects.

Non-linear waveshaping distortion is commonly used for both subtle non-linear warming, or more obvious distortion effects. Arbitrary non-linear shaping curves may be specified.

Property	Value	Notes
`numberOfInputs`	1	
`numberOfOutputs`	1	
`channelCount`	2	
`channelCountMode`	"`max`"	
`channelInterpretation`	"`speakers`"	
tail-time	Maybe	There is a tail-time only if the `oversample` attribute is set to "`2x`" or "`4x`". The actual duration of this tail-time depends on the implementation.

The number of channels of the output always equals the number of channels of the input.

```
enum OverSampleType {
    "none",
    "2x",
    "4x"
};
```

`OverSampleType` enumeration description

Enum value	Description
"***none***"	Don't oversample
"***2x***"	Oversample two times

Enum value	Description
"*4x*"	Oversample four times

```
[Exposed=Window]
interface WaveShaperNode : AudioNode {
    constructor (BaseAudioContext context, optional WaveShaperOptions options = {});
    attribute Float32Array? curve;
    attribute OverSampleType oversample;
};
```

§ 1.31.1. Constructors

WaveShaperNode(context, options)

When the constructor is called with a `BaseAudioContext` *c* and an option object *option*, the user agent MUST initialize the AudioNode *this*, with *context* and *options* as arguments.

Also, let ***[[curve set]]*** be an internal slot of this `WaveShaperNode`. Initialize this slot to `false`. If `options` is given and specifies a `curve`, set `[[curve set]]` to `true`.

Arguments for the WaveShaperNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	`BaseAudioContext`	✗	✗	The `BaseAudioContext` this new `WaveShaperNode` will be associated with.
options	`WaveShaperOptions`	✗	✓	Optional initial parameter value for this `WaveShaperNode`.

§ 1.31.2. Attributes

***curve*, of type Float32Array, nullable**

The shaping curve used for the waveshaping effect. The input signal is nominally within the range [-1, 1]. Each input sample within this range will index into the shaping curve, with a signal level of zero corresponding to the center value of the curve array if there are an odd number of entries, or interpolated between the two centermost values if there are an even number of entries in the array. Any sample value less than -1 will correspond to the first value in the curve array. Any sample value greater than +1 will correspond to the last value in the curve array.

The implementation MUST perform linear interpolation between adjacent points in the curve. Initially the curve attribute is null, which means that the WaveShaperNode will pass its input to its output without modification.

Values of the curve are spread with equal spacing in the [-1; 1] range. This means that a `curve` with a even number of value will not have a value for a signal at zero, and a `curve` with an odd number of value will have a value for a signal at zero. The output is determined by the following algorithm.

1. Let x be the input sample, y be the corresponding output of the node, c_k be the k'th element of the `curve`, and N be the length of the `curve`.
2. Let

$$v = \frac{N-1}{2}(x+1)$$
$$k = \lfloor v \rfloor$$
$$f = v - k$$

Error preparing HTML-CSS output (preProcess)

3. Then

$$y = \begin{cases} c_0 & v < 0 \\ c_{N-1} & v \geq N-1 \\ (1-f)\,c_k + f c_{k+1} & \text{otherwise} \end{cases}$$

⌛ A `InvalidStateError` MUST be thrown if this attribute is set with a `Float32Array` that has a `length` less than 2.

When this attribute is set, an internal copy of the curve is created by the `WaveShaperNode`. Subsequent modifications of the contents of the array used to set the attribute therefore have no effect.

To set the `curve` attribute, execute these steps:

1. Let *new curve* be a `Float32Array` to be assigned to `curve` or `null`. .
2. If *new curve* is not `null` and `[[curve set]]` is true, throw an `InvalidStateError` and abort these steps.
3. If *new curve* is not `null`, set `[[curve set]]` to true.
4. Assign *new curve* to the `curve` attribute.

> NOTE: The use of a curve that produces a non-zero output value for zero input value will cause this node to produce a DC signal even if there are no inputs connected to this node. This will persist until the node is disconnected from downstream nodes.

***`oversample`*, of type OverSampleType**

Specifies what type of oversampling (if any) should be used when applying the shaping curve. The default value is "`none`", meaning the curve will be applied directly to the input samples. A value of "`2x`" or "`4x`" can improve the quality of the processing by avoiding some aliasing, with the "`4x`" value yielding the highest quality. For some applications, it's better to use no oversampling in order to get a very precise shaping curve.

A value of "`2x`" or "`4x`" means that the following steps MUST be performed:

1. Up-sample the input samples to 2x or 4x the sample-rate of the `AudioContext`. Thus for each render quantum, generate 256 (for 2x) or 512 (for 4x) samples.
2. Apply the shaping curve.
3. Down-sample the result back to the sample-rate of the `AudioContext`. Thus taking the 256 (or 512) processed samples, generating 128 as the final result.

The exact up-sampling and down-sampling filters are not specified, and can be tuned for sound quality (low aliasing, etc.), low latency, or performance.

> NOTE: Use of oversampling introduces some degree of audio processing latency due to the up-sampling and down-sampling filters. The amount of this latency can vary from one implementation to another.

§ 1.31.3. `WaveShaperOptions`

This specifies the options for constructing a `WaveShaperNode`. All members are optional; if not specified, the normal default is used in constructing the node.

```
dictionary WaveShaperOptions : AudioNodeOptions {
    sequence<float> curve;
    OverSampleType oversample = "none";
};
```

✓ MDN

Error preparing HTML-CSS output (preProcess)

1.31.3.1. Dictionary `WaveShaperOptions` Members

`curve`, of type sequence<float>
The shaping curve for the waveshaping effect.

`oversample`, of type OverSampleType, defaulting to `"none"`
The type of oversampling to use for the shaping curve.

1.32. The `AudioWorklet` Interface

PROPOSED ADDITION ISSUE 2456. Add a MessagePort to the AudioWorkletGlobalScope

Previous Change | Next Change

~~[Exposed=Window, SecureContext]~~
~~interface AudioWorklet : Worklet {~~
~~};~~

```
[Exposed=Window, SecureContext]
interface AudioWorklet : Worklet {
  readonly attribute MessagePort port;
};
```

✓ MDN

1.32.1. Attributes

***`port`*, of type MessagePort, readonly**
A `MessagePort` connected to the port on the `AudioWorkletGlobalScope`.

> NOTE: Authors that register an event listener on the `"message"` event of this `port` should call `close` on either end of the `MessageChannel` (either in the `AudioWorklet` or the `AudioWorkletGlobalScope` side) to allow for resources to be collected.

Show Change | Show Current | Show Future

1.32.2. Concepts

✓ MDN

The `AudioWorklet` object allows developers to supply scripts (such as JavaScript or WebAssembly code) to process audio on the rendering thread, supporting custom `AudioNode`s. This processing mechanism ensures synchronous execution of the script code with other built-in `AudioNode`s in the audio graph.

An associated pair of objects MUST be defined in order to realize this mechanism: `AudioWorkletNode` and `AudioWorkletProcessor`. The former represents the interface for the main global scope similar to other `AudioNode` objects, and the latter implements the internal audio processing within a special scope named `AudioWorkletGlobalScope`.

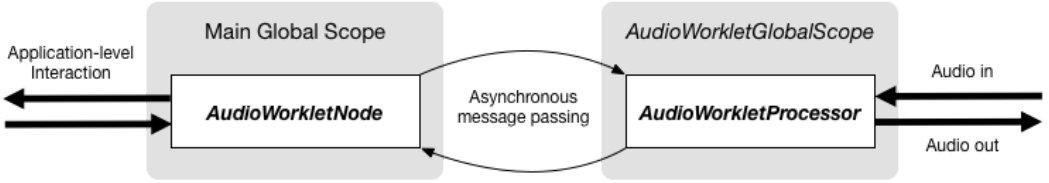


Figure 16 *`AudioWorkletNode` and `AudioWorkletProcessor`*

Each `BaseAudioContext` possesses exactly one `AudioWorklet`.

The `AudioWorklet`'s worklet global scope type is `AudioWorkletGlobalScope`.

Error preparing HTML-CSS output (preProcess)

The `AudioWorklet`'s worklet destination type is `"audioworklet"`.

Importing a script via the `addModule(moduleUrl)` method registers class definitions of `AudioWorkletProcessor` under the `AudioWorkletGlobalScope`. There are two internal storage areas for the imported class constructor and the active instances created from the constructor.

`AudioWorklet` has one internal slot:

- ***node name to parameter descriptor map*** which is a map containing an identical set of string keys from node name to processor constructor map that are associated with the matching parameterDescriptors values. This internal storage is populated as a consequence of calling the `registerProcessor()` method in the rendering thread. The population is guaranteed to complete prior to the resolution of the promise returned by `addModule()` on a context's `audioWorklet`.

EXAMPLE 12

```
// bypass-processor.js script file, runs on AudioWorkletGlobalScope
class BypassProcessor extends AudioWorkletProcessor {
    process (inputs, outputs) {
        // Single input, single channel.
        const input = inputs[0];
        const output = outputs[0];
        output[0].set(input[0]);

        // Process only while there are active inputs.
        return false;
    }
};

registerProcessor('bypass-processor', BypassProcessor);
```

EXAMPLE 13

```
// The main global scope
const context = new AudioContext();
context.audioWorklet.addModule('bypass-processor.js').then(() => {
    const bypassNode = new AudioWorkletNode(context, 'bypass-processor');
});
```

At the instantiation of `AudioWorkletNode` in the main global scope, the counterpart `AudioWorkletProcessor` will also be created in `AudioWorkletGlobalScope`. These two objects communicate via the asynchronous message passing described in § 2 Processing model.

§ 1.32.3. The `AudioWorkletGlobalScope` Interface

This special execution context is designed to enable the generation, processing, and analysis of audio data directly using a script in the audio rendering thread. The user-supplied script code is evaluated in this scope to define one or more `AudioWorkletProcessor` subclasses, which in turn are used to instantiate `AudioWorkletProcessor`s, in a 1:1 association with `AudioWorkletNode`s in the main scope.

Exactly one `AudioWorkletGlobalScope` exists for each `AudioContext` that contains one or more `AudioWorkletNode`s. The running of imported scripts is performed by the UA as defined in [HTML]. Overriding the default specified in [HTML], `AudioWorkletGlobalScope`s must not be terminated arbitrarily by the user agent. ✓ MDN

An `AudioWorkletGlobalScope` has the following internal slots:

- ***node name to processor constructor map*** which is a map that stores key-value pairs of *processor name* → *`AudioWorkletProcessorConstructor`* instance. Initially this map is empty and populated when the `registerProcessor()` method is called.
- ***pending processor construction data*** stores temporary data generated by the `AudioWorkletNode` constructor for the instantiation of the corresponding `AudioWorkletProcessor`. The pending processor construction data contains the following items: ✓ MDN
    - ***node reference*** which is initially empty. This storage is for an `AudioWorkletNode` reference that is transferred to the `AudioWorkletNode` constructor. ✓ MDN

Error preparing HTML-CSS output (preProcess)

- ***transferred port*** which is initially empty. This storage is for a deserialized `MessagePort` that is transferred from the `AudioWorkletNode` constructor.

NOTE: The `AudioWorkletGlobalScope` may also contain any other data and code to be shared by these instances. As an example, multiple processors might share an ArrayBuffer defining a wavetable or an impulse response.

NOTE: An `AudioWorkletGlobalScope` is associated with a single `BaseAudioContext`, and with a single audio rendering thread for that context. This prevents data races from occurring in global scope code running within concurrent threads.

PROPOSED ADDITION ISSUE 2456. Add a MessagePort to the AudioWorkletGlobalScope

Previous Change | Next Change

```
~~callback AudioWorkletProcessorConstructor = AudioWorkletProcessor (object options);~~

~~[Global (Worklet, AudioWorklet), Exposed AudioWorklet]~~
~~interface AudioWorkletGlobalScope : WorkletGlobalScope {~~
~~    undefined registerProcessor (DOMString name,~~
~~                                 AudioWorkletProcessorConstructor processorCtor);~~
~~    readonly attribute unsigned long long currentFrame;~~
~~    readonly attribute double currentTime;~~
~~    readonly attribute float sampleRate;~~
~~};~~
```

```
callback AudioWorkletProcessorConstructor = AudioWorkletProcessor (object options);

[Global=(Worklet,AudioWorklet),Exposed=AudioWorklet]
interface AudioWorkletGlobalScope : WorkletGlobalScope {
    undefined registerProcessor (DOMString name,
                                 AudioWorkletProcessorConstructor processorCtor);
    readonly attribute unsigned long long currentFrame;
    readonly attribute double currentTime;
    readonly attribute float sampleRate;
    readonly attribute MessagePort port;
};
```

Show Change | Show Current | Show Future

1.32.3.1. Attributes

***currentFrame*, of type unsigned long long, readonly**
: The current frame of the block of audio being processed. This must be equal to the value of the `[[current frame]]` internal slot of the `BaseAudioContext`.

***currentTime*, of type double, readonly**
: The context time of the block of audio being processed. By definition this will be equal to the value of `BaseAudioContext`'s `currentTime` attribute that was most recently observable in the control thread.

***sampleRate*, of type float, readonly**
: The sample rate of the associated `BaseAudioContext`.

Error preparing HTML-CSS output (preProcess)

PROPOSED ADDITION ISSUE 2456. Add a MessagePort to the AudioWorkletGlobalScope

Previous Change

***port*, of type MessagePort, readonly**

A `MessagePort` connected to the port on the `AudioWorklet`.

> NOTE: Authors that register an event listener on the `"message"` event of this `port` should call `close` on either end of the `MessageChannel` (either in the `AudioWorklet` or the `AudioWorkletGlobalScope` side) to allow for resources to be collected.

Show Change | Show Current | Show Future

✓ MDN

§ *1.32.3.2. Methods*

registerProcessor(name, processorCtor)

Registers a class constructor derived from `AudioWorkletProcessor`.

When the `registerProcessor(name, processorCtor)` method is called, perform the following steps. If an exception is thrown in any step, abort the remaining steps.

1. If *name* is an empty string, ⌛ throw a `NotSupportedError`.
2. If *name* already exists as a key in the node name to processor constructor map, ⌛ throw a `NotSupportedError`.
3. If the result of `IsConstructor`(`argument=`*`processorCtor`*) is `false`, ⌛ throw a `TypeError` .
4. Let *prototype* be the result of `Get`(`O=`*`processorCtor`*, `P="prototype"`).
5. If the result of `Type`(`argument=`*`prototype`*) is not `Object`, ⌛ throw a `TypeError` .
6. Let *parameterDescriptorsValue* be the result of `Get`(`O=`*`processorCtor`*, `P="parameterDescriptors"`).
7. If *parameterDescriptorsValue* is not `undefined`, execute the following steps:
   1. Let *parameterDescriptorSequence* be the result of the conversion from *parameterDescriptorsValue* to an IDL value of type `sequence<AudioParamDescriptor>`.
   2. Let *paramNames* be an empty Array.
   3. For each *descriptor* of *parameterDescriptorSequence*:
      1. Let *paramName* be the value of the member `name` in *descriptor*. ⌛ Throw a `NotSupportedError` if *paramNames* already contains *paramName* value.
      2. Append *paramName* to the *paramNames* array.
      3. Let *defaultValue* be the value of the member `defaultValue` in *descriptor*.
      4. Let *minValue* be the value of the member `minValue` in *descriptor*.
      5. Let *maxValue* be the value of the member `maxValue` in *descriptor*.
      6. If the expresstion *minValue* <= *defaultValue* <= *maxValue* is false, ⌛ throw an `InvalidStateError`.
8. Append the key-value pair *name* → *processorCtor* to node name to processor constructor map of the associated `AudioWorkletGlobalScope`.
9. queue a media element task to append the key-value pair *name* → *parameterDescriptorSequence* to the node name to parameter descriptor map of the associated `BaseAudioContext`.

> NOTE: The class constructor should only be looked up once, thus it does not have the opportunity to dynamically change after registration.

Arguments for the AudioWorkletGlobalScope.registerProcessor(name, processorCtor) method.

✓ MDN

Parameter	Type	Nullable	Optional	Description
Error preparing HTML-CSS output (preProcess)	`DOMString`	✗	✗	A string key that represents a class constructor to be

Parameter	Type	Nullable	Optional	Description
				registered. This key is used to look up the constructor of AudioWorkletProcessor during construction of an AudioWorkletNode.
processorCtor	AudioWorkletProcessorConstructor	✗	✗	A class constructor extended from AudioWorkletProcessor.

Return type: undefined

§ *1.32.3.3. The instantiation of AudioWorkletProcessor*

At the end of the AudioWorkletNode construction, A struct named ***processor construction data*** will be prepared for cross-thread transfer. This struct contains the following items:

- ***name*** which is a DOMString that is to be looked up in the node name to processor constructor map.
- ***node*** which is a reference to the AudioWorkletNode created.
- ***options*** which is a serialized AudioWorkletNodeOptions given to the AudioWorkletNode's constructor.
- ***port*** which is a serialized MessagePort paired with the AudioWorkletNode's port.

Upon the arrival of the transferred data on the AudioWorkletGlobalScope, the rendering thread will invoke the algorithm below:

1. Let *constructionData* be the processor construction data transferred from the control thread.
2. Let *processorName*, *nodeReference* and *serializedPort* be *constructionData*'s name, node, and port respectively.
3. Let *serializedOptions* be *constructionData*'s options.
4. Let *deserializedPort* be the result of StructuredDeserialize(*serializedPort*, the current Realm).
5. Let *deserializedOptions* be the result of StructuredDeserialize(*serializedOptions*, the current Realm).
6. Let *processorCtor* be the result of looking up *processorName* on the AudioWorkletGlobalScope's node name to processor constructor map.
7. Store *nodeReference* and *deserializedPort* to node reference and transferred port of this AudioWorkletGlobalScope's pending processor construction data respectively.
8. Construct a callback function from *processorCtor* with the argument of *deserializedOptions*. If any exceptions are thrown in the callback, queue a task to the control thread to fire an event named processorerror using ErrorEvent at *nodeReference*.
9. Empty the pending processor construction data slot.

§ 1.32.4. The AudioWorkletNode Interface

This interface represents a user-defined AudioNode which lives on the control thread. The user can create an AudioWorkletNode from a BaseAudioContext, and such a node can be connected with other built-in AudioNodes to form an audio graph.

Property	Value	Notes
numberOfInputs	1	
numberOfOutputs	1	
channelCount	2	
channelCountMode	"max"	
channelInterpretation	"speakers"	

Error preparing HTML-CSS output (preProcess)

Property	Value	Notes
tail-time	See notes	Any tail-time is handled by the node itself

Every AudioWorkletProcessor has an associated ***active source*** flag, initially `true`. This flag causes the node to be retained in memory and perform audio processing in the absence of any connected inputs.

All tasks posted from an AudioWorkletNode are posted to the task queue of its associated BaseAudioContext.

```
[Exposed=Window]
interface AudioParamMap {
    readonly maplike<DOMString, AudioParam>;
};
```

This interface has "entries", "forEach", "get", "has", "keys", "values", @@iterator methods and a "size" getter brought by `readonly maplike`.

```
[Exposed=Window, SecureContext]
interface AudioWorkletNode : AudioNode {
    constructor (BaseAudioContext context, DOMString name,
                 optional AudioWorkletNodeOptions options = {});
    readonly attribute AudioParamMap parameters;
    readonly attribute MessagePort port;
    attribute EventHandler onprocessorerror;
};
```

§ *1.32.4.1. Constructors*

AudioWorkletNode(context, name, options)

Arguments for the AudioWorkletNode.constructor() method.

Parameter	Type	Nullable	Optional	Description
context	BaseAudioContext	✗	✗	The BaseAudioContext this new AudioWorkletNode will be associated with.
name	DOMString	✗	✗	A string that is a key for the BaseAudioContext's node name to parameter descriptor map.
options	AudioWorkletNodeOptions	✗	✓	Optional initial parameters value for this AudioWorkletNode.

When the constructor is called, the user agent MUST perform the following steps on the control thread:

When the AudioWorkletNode constructor is invoked with *context*, *nodeName*, *options*:

1. If *nodeName* does not exist as a key in the BaseAudioContext's node name to parameter descriptor map, throw a InvalidStateError exception and abort these steps.
2. Let *node* be this value.
3. Initialize the AudioNode *node* with *context* and *options* as arguments.
4. Configure input, output and output channels of *node* with *options*. Abort the remaining steps if any exception is thrown.
5. Let *messageChannel* be a new MessageChannel.
6. Set the value of *messageChannel*'s port1 attribute.

Error preparing HTML-CSS output (preProcess)

7. Let *processorPortOnThisSide* be the value of *messageChannel*'s `port2` attribute.
8. Let *serializedProcessorPort* be the result of StructuredSerializeWithTransfer(*processorPortOnThisSide*, « *processorPortOnThisSide* »).
9. Convert *options* dictionary to *optionsObject*.
10. Let *serializedOptions* be the result of StructuredSerialize(*optionsObject*).
11. Set *node*'s `port` to *nodePort*.
12. Let *parameterDescriptors* be the result of retrieval of *nodeName* from node name to parameter descriptor map:
    1. Let *audioParamMap* be a new `AudioParamMap` object.
    2. For each *descriptor* of *parameterDescriptors*:
        1. Let *paramName* be the value of `name` member in *descriptor*.
        2. Let *audioParam* be a new `AudioParam` instance with `automationRate`, `defaultValue`, `minValue`, and `maxValue` having values equal to the values of corresponding members on *descriptor*.
        3. Append a key-value pair *paramName* → *audioParam* to *audioParamMap*'s entries.
    3. If `parameterData` is present on *options*, perform the following steps:
        1. Let *parameterData* be the value of `parameterData`.
        2. For each *paramName* → *paramValue* of *parameterData*:
            1. If there exists a map entry on *audioParamMap* with key *paramName*, let *audioParamInMap* be such entry.
            2. Set `value` property of *audioParamInMap* to *paramValue*.
    4. Set *node*'s `parameters` to *audioParamMap*.
13. Queue a control message to invoke the `constructor` of the corresponding `AudioWorkletProcessor` with the processor construction data that consists of: *nodeName*, *node*, *serializedOptions*, and *serializedProcessorPort*.

§ *1.32.4.2. Attributes*

***onprocessorerror*, of type EventHandler**

When an unhandled exception is thrown from the processor's `constructor`, `process` method, or any user-defined class method, the processor will queue a media element task to fire an event named ***processorerror*** using ErrorEvent at the associated `AudioWorkletNode`.

The `ErrorEvent` is created and initialized appropriately with its `message`, `filename`, `lineno`, `colno` attributes on the control thread.

Note that once a unhandled exception is thrown, the processor will output silence throughout its lifetime.

***parameters*, of type AudioParamMap, readonly**

The `parameters` attribute is a collection of `AudioParam` objects with associated names. This maplike object is populated from a list of `AudioParamDescriptor`s in the `AudioWorkletProcessor` class constructor at the instantiation.

***port*, of type MessagePort, readonly**

Every `AudioWorkletNode` has an associated `port` which is the `MessagePort`. It is connected to the port on the corresponding `AudioWorkletProcessor` object allowing bidirectional communication between the `AudioWorkletNode` and its `AudioWorkletProcessor`.

> NOTE: Authors that register an event listener on the `"message"` event of this `port` should call `close` on either end of the `MessageChannel` (either in the `AudioWorkletProcessor` or the `AudioWorkletNode` side) to allow for resources to be collected.

§ *1.32.4.3. `AudioWorkletNodeOptions`*

Error preparing HTML-CSS output (preProcess)

The AudioWorkletNodeOptions dictionary can be used to initialize attibutes in the instance of an AudioWorkletNode.

```
dictionary AudioWorkletNodeOptions : AudioNodeOptions {
    unsigned long numberOfInputs = 1;
    unsigned long numberOfOutputs = 1;
    sequence<unsigned long> outputChannelCount;
    record<DOMString, double> parameterData;
    object processorOptions;
};
```

1.32.4.3.1. Dictionary AudioWorkletNodeOptions Members

***numberOfInputs*, of type unsigned long, defaulting to 1**
: This is used to initialize the value of the AudioNode numberOfInputs attribute.

***numberOfOutputs*, of type unsigned long, defaulting to 1**
: This is used to initialize the value of the AudioNode numberOfOutputs attribute.

***outputChannelCount*, of type sequence<unsigned long>**
: This array is used to configure the number of channels in each output.

***parameterData*, of type record<DOMString, double>**
: This is a list of user-defined key-value pairs that are used to set the initial value of an AudioParam with the matched name in the AudioWorkletNode.

***processorOptions*, of type object**
: This holds any user-defined data that may be used to initialize custom properties in an AudioWorkletProcessor instance that is associated with the AudioWorkletNode.

1.32.4.3.2. Configuring Channels with AudioWorkletNodeOptions

The following algorithm describes how an AudioWorkletNodeOptions can be used to configure various channel configurations.

1. Let *node* be an AudioWorkletNode instance that is given to this algorithm.
2. If both numberOfInputs and numberOfOutputs are zero, throw a NotSupportedError and abort the remaining steps.
3. If outputChannelCount exists,
    1. If any value in outputChannelCount is zero or greater than the implementation's maximum number of channels, throw a NotSupportedError and abort the remaining steps.
    2. If the length of outputChannelCount does not equal numberOfOutputs, throw an IndexSizeError and abort the remaining steps.
    3. If both numberOfInputs and numberOfOutputs are 1, set the channel count of the *node* output to the one value in outputChannelCount.
    4. Otherwise set the channel count of the *k*th output of the *node* to the *k*th element of outputChannelCount sequence and return.
4. If outputChannelCount exists,
    1. If both numberOfInputs and numberOfOutputs are 1, set the initial channel count of the *node* output to 1 and return.

        NOTE: For this case, the output chanel count will change to computedNumberOfChannels dynamically based on the input and the channelCountMode at runtime.
    2. Otherwise set the channel count of each output of the *node* to 1 and return.

Error preparing HTML-CSS output (preProcess) etProcessor Interface

This interface represents an audio processing code that runs on the audio rendering thread. It lives in the AudioWorkletGlobalScope, and the definition of the class manifests the actual audio processing. Note that the an AudioWorkletProcessor construction can only happen as a result of an AudioWorkletNode contruction.

```
[Exposed=AudioWorklet]
interface AudioWorkletProcessor {
    constructor ();
    readonly attribute MessagePort port;
};

callback AudioWorkletProcessCallback =
  boolean (FrozenArray<FrozenArray<Float32Array>> inputs,
           FrozenArray<FrozenArray<Float32Array>> outputs,
           object parameters);
```

AudioWorkletProcessor has two internal slots:

[[node reference]]
A reference to the associated AudioWorkletNode.

[[callable process]]
A boolean flag representing whether process() is a valid function that can be invoked.

§ *1.32.5.1. Constructors*

AudioWorkletProcessor()
When the constructor for AudioWorkletProcessor is invoked, the following steps are performed on the rendering thread.

1. Let *nodeReference* be the result of looking up node reference on the pending processor construction data of the current AudioWorkletGlobalScope. Throw a TypeError exception if the slot is empty.
2. Let *processor* be the this value.
3. Set *processor*'s [[node reference]] to *nodeReference*.
4. Set *processor*'s [[callable process]] to `true`.
5. Let *deserializedPort* be the result of looking up transferred port from the pending processor construction data.
6. Set *processor*'s port to *deserializedPort*.
7. Empty the pending processor construction data slot.

§ *1.32.5.2. Attributes*

***port*, of type MessagePort, readonly**
Every AudioWorkletProcessor has an associated `port` which is a MessagePort. It is connected to the port on the corresponding AudioWorkletNode object allowing bidirectional communication between an AudioWorkletNode and its AudioWorkletProcessor.

> NOTE: Authors that register an event listener on the `"message"` event of this port should call close on either end of the MessageChannel (either in the AudioWorkletProcessor or the AudioWorkletNode side) to allow for resources to be collected.

§ *1.32.5.3. Callback AudioWorkletProcessCallback*

Users can define a custom audio processor by extending AudioWorkletProcessor. The subclass MUST define an AudioWorkletProcessCallback named ***process()*** that implements the audio processing algorithm and may have a static property named ***parameterDescriptors*** which is an iterable of AudioParamDescriptors.

The process() callback function is handled as specified when rendering a graph.

Error preparing HTML-CSS output (preProcess)

The return value of this callback controls the lifetime of the `AudioWorkletProcessor`'s associated `AudioWorkletNode`.

This lifetime policy can support a variety of approaches found in built-in nodes, including the following:

- Nodes that transform their inputs, and are active only while connected inputs and/or script references exist. Such nodes SHOULD return `false` from process() which allows the presence or absence of connected inputs to determine whether the `AudioWorkletNode` is actively processing.
- Nodes that transform their inputs, but which remain active for a tail-time after their inputs are disconnected. In this case, process() SHOULD return `true` for some period of time after `inputs` is found to contain zero channels. The current time may be obtained from the global scope's `currentTime` to measure the start and end of this tail-time interval, or the interval could be calculated dynamically depending on the processor's internal state.
- Nodes that act as sources of output, typically with a lifetime. Such nodes SHOULD return `true` from process() until the point at which they are no longer producing an output.

Note that the preceding definition implies that when no return value is provided from an implementation of process(), the effect is identical to returning `false` (since the effective return value is the falsy value `undefined`). This is a reasonable behavior for any `AudioWorkletProcessor` that is active only when it has active inputs.

The example below shows how `AudioParam` can be defined and used in an `AudioWorkletProcessor`.

```
EXAMPLE 14
class MyProcessor extends AudioWorkletProcessor {
  static get parameterDescriptors() {
    return [{
      name: 'myParam',
      defaultValue: 0.5,
      minValue: 0,
      maxValue: 1,
      automationRate: "k-rate"
    }];
  }

  process(inputs, outputs, parameters) {
    // Get the first input and output.
    const input = inputs[0];
    const output = outputs[0];
    const myParam = parameters.myParam;

    // A simple amplifier for single input and output. Note that the
    // automationRate is "k-rate", so it will have a single value at index [0]
    // for each render quantum.
    for (let channel = 0; channel < output.length; ++channel) {
      for (let i = 0; i < output[channel].length; ++i) {
        output[channel][i] = input[channel][i] * myParam[0];
      }
    }
  }
}
```

§ 1.32.5.3.1. Callback `AudioWorkletProcessCallback` Parameters

The following describes the parameters to the `AudioWorkletProcessCallback` function.

In general, the `inputs` and `outputs` arrays will be reused between calls so that no memory allocation is done. However, if the topology changes, because, say, the number of channels in the input or the output changes, new arrays are reallocated. New arrays are also reallocated if any part of the `inputs` or `outputs` arrays are transferred.

`inputs`, of type `FrozenArray<FrozenArray<Float32Array>>`

Error preparing HTML-CSS output (preProcess)

er from the incoming connections provided by the user agent. `inputs[n][m]` is a `Float32Array` of audio samples for the *m*th channel of the *n*th input. While the number of inputs is fixed at construction, the number

of channels can be changed dynamically based on computedNumberOfChannels.

If there are no actively processing `AudioNode`s connected to the *n*th input of the `AudioWorkletNode` for the current render quantum, then the content of `inputs[n]` is an empty array, indicating that zero channels of input are available. This is the only circumstance under which the number of elements of `inputs[n]` can be zero.

`outputs`, of type `FrozenArray<FrozenArray<Float32Array>>`
The output audio buffer that is to be consumed by the user agent. `outputs[n][m]` is a `Float32Array` object containing the audio samples for *m*th channel of *n*th output. Each of the `Float32Array`s are zero-filled. The number of channels in the output will match computedNumberOfChannels only when the node has a single output.

`parameters`, of type `object`
An ordered map of *name* → *parameterValues*. `parameters["name"]` returns *parameterValues*, which is a `FrozenArray`<`Float32Array`> with the automation values of the *name* `AudioParam`.

For each array, the array contains the computedValue of the parameter for all frames in the render quantum. However, if no automation is scheduled during this render quantum, the array MAY have length 1 with the array element being the constant value of the `AudioParam` for the render quantum.

This object is frozen according the the following steps

1. Let *parameter* be the ordered map of the name and parameter values.
2. SetIntegrityLevel(*parameter*, frozen)

This frozen ordered map computed in the algorithm is passed to the `parameters` argument.

> NOTE: This means the object cannot be modified and hence the same object can be used for successive calls unless length of an array changes.

1.32.5.4. *`AudioParamDescriptor`*

The `AudioParamDescriptor` dictionary is used to specify properties for an `AudioParam` object that is used in an `AudioWorkletNode`.

```
dictionary AudioParamDescriptor {
    required DOMString name;
    float defaultValue = 0;
    float minValue = -3.4028235e38;
    float maxValue = 3.4028235e38;
    AutomationRate automationRate = "a-rate";
};
```

1.32.5.4.1. Dictionary `AudioParamDescriptor` Members

There are constraints on the values for these members. See the algorithm for handling an AudioParamDescriptor for the constraints.

***`automationRate`*, of type AutomationRate, defaulting to `"a-rate"`**
Represents the default automation rate.

***`defaultValue`*, of type float, defaulting to `0`**
Represents the default value of the parameter.

***`maxValue`*, of type float, defaulting to `3.4028235e38`**
Represents the maximum value.

***`minValue`*, of type float, defaulting to `-3.4028235e38`**
Represents the minimum value.

***`name`*, of type DOMString**
Represents the name of the parameter.

Error preparing HTML-CSS output (preProcess)

§ 1.32.6. AudioWorklet Sequence of Events

The following figure illustrates an idealized sequence of events occurring relative to an `AudioWorklet`:

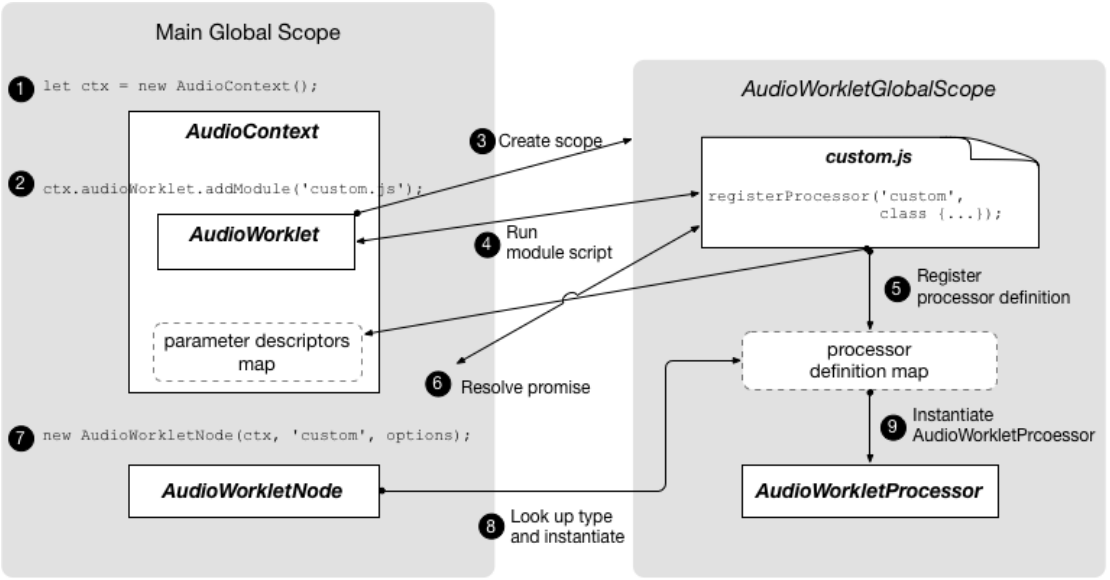


Figure 17 *`AudioWorklet` sequence*

The steps depicted in the diagram are one possible sequence of events involving the creation of an `AudioContext` and an associated `AudioWorkletGlobalScope`, followed by the creation of an `AudioWorkletNode` and its associated `AudioWorkletProcessor`.

1. An `AudioContext` is created.
2. In the main scope, `context.audioWorklet` is requested to add a script module.
3. Since none exists yet, a new `AudioWorkletGlobalScope` is created in association with the context. This is the global scope in which `AudioWorkletProcessor` class definitions will be evaluated. (On subsequent calls, this previously created scope will be used.)
4. The imported script is run in the newly created global scope.
5. As part of running the imported script, an `AudioWorkletProcessor` is registered under a key (`"custom"` in the above diagram) within the `AudioWorkletGlobalScope`. This populates maps both in the global scope and in the `AudioContext`.
6. The promise for the `addModule()` call is resolved.
7. In the main scope, an `AudioWorkletNode` is created using the user-specified key along with a dictionary of options.
8. As part of the node's creation, this key is used to look up the correct `AudioWorkletProcessor` subclass for instantiation.
9. An instance of the `AudioWorkletProcessor` subclass is instantiated with a structured clone of the same options dictionary. This instance is paired with the previously created `AudioWorkletNode`.

Error preparing HTML-CSS output (preProcess)

1.32.7. AudioWorklet Examples

1.32.7.1. The BitCrusher Node

Bitcrushing is a mechanism by which the quality of an audio stream is reduced both by quantizing the sample value (simulating a lower bit-depth), and by quantizing in time resolution (simulating a lower sample rate). This example shows

Error preparing HTML-CSS output (preProcess)

how to use AudioParams (in this case, treated as a-rate) inside an AudioWorkletProcessor.

EXAMPLE 15

```
const context = new AudioContext();
context.audioWorklet.addModule('bitcrusher.js').then(() => {
    const osc = new OscillatorNode(context);
    const amp = new GainNode(context);

    // Create a worklet node. 'BitCrusher' identifies the
    // AudioWorkletProcessor previously registered when
    // bitcrusher.js was imported. The options automatically
    // initialize the correspondingly named AudioParams.
    const bitcrusher = new AudioWorkletNode(context, 'bitcrusher', {
        parameterData: {bitDepth: 8}
    });

    osc.connect(bitcrusher).connect(amp).connect(context.destination);
    osc.start();
});
```

EXAMPLE 16

```
class Bitcrusher extends AudioWorkletProcessor {
    static get parameterDescriptors () {
        return [{
            name: 'bitDepth',
            defaultValue: 12,
            minValue: 1,
            maxValue: 16
        }, {
            name: 'frequencyReduction',
            defaultValue: 0.5,
            minValue: 0,
            maxValue: 1
        }];
    }

    constructor (options) {
        // The initial parameter value can be set by passing |options|
        // to the processor's constructor.
        super(options);
        this._phase = 0;
        this._lastSampleValue = 0;
    }

    process (inputs, outputs, parameters) {
        const input = inputs[0];
        const output = outputs[0];
        const bitDepth = parameters.bitDepth;
        const frequencyReduction = parameters.frequencyReduction;

        if (bitDepth.length > 1) {
            // The bitDepth parameter array has 128 sample values.
            for (let channel = 0; channel < output.length; ++channel) {
                for (let i = 0; i < output[channel].length; ++i) {
                    let step = Math.pow(0.5, bitDepth[i]);

                    // Use modulo for indexing to handle the case where
                    // the length of the frequencyReduction array is 1.
                    this._phase += frequencyReduction[i % frequencyReduction.length];
                    if (this._phase >= 1.0) {
                        this._phase -= 1.0;
                        this._lastSampleValue =
                            step * Math.floor(input[channel][i] / step + 0.5);
                    }
                    output[channel][i] = this._lastSampleValue;
```

Error preparing HTML-CSS output (preProcess)

```
                }
            }
        } else {
            // Because we know bitDepth is constant for this call,
            // we can lift the computation of step outside the loop,
            // saving many operations.
            const step = Math.pow(0.5, bitDepth[0]);
            for (let channel = 0; channel < output.length; ++channel) {
                for (let i = 0; i < output[channel].length; ++i) {
                    this._phase += frequencyReduction[i % frequencyReduction.length];
                    if (this._phase >= 1.0) {
                        this._phase -= 1.0;
                        this._lastSampleValue =
                            step * Math.floor(input[channel][i] / step + 0.5);
                    }
                    output[channel][i] = this._lastSampleValue;
                }
            }
        }
        // No need to return a value; this node's lifetime is dependent only on its
        // input connections.
    }
});

registerProcessor('bitcrusher', Bitcrusher);
```

NOTE: In the definition of AudioWorkletProcessor class, an InvalidStateError will be thrown if the author-supplied constructor has an explicit return value that is not `this` or does not properly call `super()`.

1.32.7.2. VU Meter Node

This example of a simple sound level meter further illustrates how to create an AudioWorkletNode subclass that acts lik a native AudioNode, accepting constructor options and encapsulating the inter-thread communication (asynchronous) between AudioWorkletNode and AudioWorkletProcessor. This node does not use any output.

Error preparing HTML-CSS output (preProcess)

CANDIDATE CORRECTION ISSUE 2359. Fix typo in code; semi-colon is incorrect.

Show Change | Show Current | Show Future

EXAMPLE 17

```
/* vumeter-node.js: Main global scope */

export default class VUMeterNode extends AudioWorkletNode {
    constructor (context, updateIntervalInMS) {
        super(context, 'vumeter', {
            numberOfInputs: 1,
            numberOfOutputs: 0,
            channelCount: 1,
            processorOptions: {
                updateIntervalInMS: updateIntervalInMS || 16.67;
            }
        });

        // States in AudioWorkletNode
        this._updateIntervalInMS = updateIntervalInMS;
        this._volume = 0;

        // Handles updated values from AudioWorkletProcessor
        this.port.onmessage = event => {
            if (event.data.volume)
                this._volume = event.data.volume;
        }
        this.port.start();
    }

    get updateInterval() {
        return this._updateIntervalInMS;
    }

    set updateInterval(updateIntervalInMS) {
        this._updateIntervalInMS = updateIntervalInMS;
        this.port.postMessage({updateIntervalInMS: updateIntervalInMS});
    }

    draw () {
        // Draws the VU meter based on the volume value
        // every |this._updateIntervalInMS| milliseconds.
    }
};
```

EXAMPLE 18

```
/* vumeter-processor.js: AudioWorkletGlobalScope */

const SMOOTHING_FACTOR = 0.9;
const MINIMUM_VALUE = 0.00001;

registerProcessor('vumeter', class extends AudioWorkletProcessor {
    constructor (options) {
        super();
        this._volume = 0;
        this._updateIntervalInMS = options.processorOptions.updateIntervalInMS;
        this._nextUpdateFrame = this._updateIntervalInMS;

        this.port.onmessage = event => {
            if (event.data.updateIntervalInMS)
                this._updateIntervalInMS = event.data.updateIntervalInMS;
        }
    }

    get intervalInFrames () {
```

Error preparing HTML-CSS output (preProcess)

```
        return this._updateIntervalInMS / 1000 * sampleRate;
    }

    process (inputs, outputs, parameters) {
        const input = inputs[0];
        // Note that the input will be down-mixed to mono; however, if no inputs are
        // connected then zero channels will be passed in.
        if (input.length > 0) {
            const samples = input[0];
            let sum = 0;
            let rms = 0;

            // Calculated the squared-sum.
            for (let i = 0; i < samples.length; ++i)
                sum += samples[i] * samples[i];

            // Calculate the RMS level and update the volume.
            rms = Math.sqrt(sum / samples.length);
            this._volume = Math.max(rms, this._volume * SMOOTHING_FACTOR);

            // Update and sync the volume property with the main thread.
            this._nextUpdateFrame -= samples.length;
            if (this._nextUpdateFrame < 0) {
                this._nextUpdateFrame += this.intervalInFrames;
                this.port.postMessage({volume: this._volume});
            }
        }

        // Keep on processing if the volume is above a threshold, so that
        // disconnecting inputs does not immediately cause the meter to stop
        // computing its smoothed value.
        return this._volume >= MINIMUM_VALUE;
    }

});
```

EXAMPLE 19

```
/* index.js: Main global scope, entry point */
import VUMeterNode from './vumeter-node.js';

const context = new AudioContext();
context.audioWorklet.addModule('vumeter-processor.js').then(() => {
    const oscillator = new OscillatorNode(context);
    const vuMeterNode = new VUMeterNode(context, 25);
    oscillator.connect(vuMeterNode);
    oscillator.start();

    function drawMeter () {
        vuMeterNode.draw();
        requestAnimationFrame(drawMeter);
    }

    drawMeter();
});
```

§ 2. Processing model

§ 2.1. Background

This section is non-normative.

Error preparing HTML-CSS output (preProcess)

Real-time audio systems that require low latency are often implemented using *callback functions*, where the operating system calls the program back when more audio has to be computed in order for the playback to stay uninterrupted. Such a callback is ideally called on a high priority thread (often the highest priority on the system). This means that a program that deals with audio only executes code from this callback. Crossing thread boundaries or adding some buffering between a rendering thread and the callback would naturally add latency or make the system less resilient to glitches.

For this reason, the traditional way of executing asynchronous operations on the Web Platform, the event loop, does not work here, as the thread is not *continuously executing*. Additionally, a lot of unnecessary and potentially blocking operations are available from traditional execution contexts (Windows and Workers), which is not something that is desirable to reach an acceptable level of performance.

Additionally, the Worker model makes creating a dedicated thread necessary for a script execution context, while all `AudioNode`s usually share the same execution context.

> NOTE: This section specifies how the end result should look like, not how it should be implemented. In particular, instead of using message queue, implementors can use memory that is shared between threads, as long as the memory operations are not reordered.

§ 2.2. Control Thread and Rendering Thread

The Web Audio API MUST be implemented using a control thread, and a rendering thread.

The ***control thread*** is the thread from which the `AudioContext` is instantiated, and from which authors manipulate the audio graph, that is, from where the operation on a `BaseAudioContext` are invoked. The ***rendering thread*** is the thread on which the actual audio output is computed, in reaction to the calls from the control thread. It can be a real-time, callback-based audio thread, if computing audio for an `AudioContext`, or a normal thread if computing audio for an `OfflineAudioContext`.

The control thread uses a traditional event loop, as described in [HTML].

The rendering thread uses a specialized rendering loop, described in the section Rendering an audio graph

Communication from the control thread to the rendering thread is done using control message passing. Communication in the other direction is done using regular event loop tasks.

Each `AudioContext` has a single ***control message queue*** that is a list of ***control messages*** that are operations running on the rendering thread.

Queuing a control message means adding the message to the end of the control message queue of an `BaseAudioContext`.

> NOTE: For example, successfuly calling `start()` on an `AudioBufferSourceNode` `source` adds a control message to the control message queue of the associated `BaseAudioContext`.

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

Swapping a control message queue Q_A with another control message queue Q_B means executing the following steps:

1. Let Q_C be a new, empty control message queue.
2. Move all the control messages Q_A to Q_C.
3. Move all the control messages Q_B to Q_A.
4. Move all the control messages Q_C to Q_B.

§ 2.3. Asynchronous Operations

Calling methods on `AudioNode`s is effectively asynchronous, and MUST to be done in two phases: a synchronous part and an asynchronous part. For each method, some part of the execution happens on the control thread (for example, throwing an exception in case of invalid parameters), and some part happens on the rendering thread (for example, changing the value of

Error preparing HTML-CSS output (preProcess)

In the description of each operation on AudioNodes and BaseAudioContexts, the synchronous section is marked with a ⌛. All the other operations are executed in parallel, as described in [HTML].

The synchronous section is executed on the control thread, and happens immediately. If it fails, the method execution is aborted, possibly throwing an exception. If it succeeds, a control message, encoding the operation to be executed on the rendering thread is enqueued on the control message queue of this rendering thread.

The synchronous and asynchronous sections order with respect to other events MUST be the same: given two operation A and B with respective synchronous and asynchronous section A_{Sync} and A_{Async}, and B_{Sync} and B_{Async}, if A happens before B, then A_{Sync} happens before B_{Sync}, and A_{Async} happens before B_{Async}. In other words, synchronous and asynchronous sections can't be reordered.

§ 2.4. Rendering an Audio Graph

Rendering an audio graph is done in blocks of 128 samples-frames. A block of 128 samples-frames is called a ***render quantum***, and the ***render quantum size*** is 128.

Operations that happen ***atomically*** on a given thread can only be executed when no other atomic operation is running on another thread.

The algorithm for rendering a block of audio from a `BaseAudioContext` G with a control message queue Q is comprised of multiple steps and explained in further detail in the algorithm of rendering a graph.

PROPOSED ADDITION ISSUE 2375. Add definition for various concepts related to system-level audio callbacks

The `AudioContext` rendering thread is driven by a ***system-level audio callback***, that is periodically at regular intevals. Each call has a ***system-level audio callback buffer size***, which is a varying number of sample-frames that needs to be computed on time before the next system-level audio callback arrives.

A ***load value*** is computed for each system-level audio callback, by dividing its execution duration by the system-level audio callback buffer size divided by `sampleRate`.

Ideally the load value is below 1.0, meaning that it took less time to render the audio that it took to play it out. An ***audio buffer underrun*** happens when this load value is greater than 1.0: the system could not render audio fast enough for real-time.

> The render quantum size for an audio graph is not necessarily a divisor of the system-level audio callback buffer size. This causes increased audio latencies and reduced possible maximum load without audio buffer underrun.

Note that the concepts of system-level audio callback and load value do not apply to `OfflineAudioContexts`.

> ~~In practice, the `AudioContext` rendering thread is often running off a system-level audio callback, that executes in an isochronous fashion.~~
>
> ~~An `OfflineAudioContext` is not required to have a system-level audio callback, but behaves as if it did with the callback happening as soon as the previous callback is finished.~~

Show Change | Show Current | Show Future

The audio callback is also queued as a task in the control message queue. The UA MUST perform the following algorithms to process render quanta to fulfill such task by filling up the requested buffer size. Along with the control message queue, each `AudioContext` has a regular task queue, called its ***associated task queue*** for tasks that are posted to the rendering thread from the control thread. An additional microtask checkpoint is performed after processing a render quantum to run any microtasks that might have been queued during the execution of the `process` methods of `AudioWorkletProcessor`.

All tasks posted from an `AudioWorkletNode` are posted to the associated task queue of its associated `BaseAudioContext`.

The following step MUST be performed once before the rendering loop starts.

1. Set the internal slot ***[[current frame]]*** of the `BaseAudioContext` to 0. Also set `currentTime` to 0.

Error preparing HTML-CSS output (preProcess)

The following steps MUST be performed when rendering a render quantum.

1. Let *render result* be `false`.
2. Process the control message queue.
    1. Let $Q_{rendering}$ be an empty control message queue. Atomically swap $Q_{rendering}$ with the current control message queue.
    2. While there are messages in $Q_{rendering}$, execute the following steps:
        1. Execute the asynchronous section of the oldest message of $Q_{rendering}$.
        2. Remove the oldest message of $Q_{rendering}$.
3. Process the `BaseAudioContext`'s associated task queue.
    1. Let *task queue* be the `BaseAudioContext`'s associated task queue.
    2. Let *task count* be the number of tasks in the in *task queue*
    3. While *task count* is not equal to 0, execute the following steps:
        1. Let *oldest task* be the first runnable task in *task queue*, and remove it from *task queue*.
        2. Set the rendering loop's currently running task to *oldest task*.
        3. Perform *oldest task*'s steps.
        4. Set the rendering loop currently running task back to `null`.
        5. Decrement *task count*
        6. Perform a microtask checkpoint.
4. Process a render quantum.
    1. If the `[[rendering thread state]]` of the `BaseAudioContext` is not `running`, return false.
    2. Order the `AudioNode`s of the `BaseAudioContext` to be processed.
        1. Let *ordered node list* be an empty list of `AudioNode`s and `AudioListener`. It will contain an ordered list of `AudioNode`s and the `AudioListener` when this ordering algorithm terminates.
        2. Let *nodes* be the set of all nodes created by this `BaseAudioContext`, and still alive.
        3. Add the `AudioListener` to *nodes*.
        4. Let *cycle breakers* be an empty set of `DelayNode`s. It will contain all the `DelayNode`s that are part of a cycle.
        5. For each `AudioNode` *node* in *nodes*:
            1. If *node* is a `DelayNode` that is part of a cycle, add it to *cycle breakers* and remove it from *nodes*.
        6. For each `DelayNode` *delay* in *cycle breakers*:
            1. Let *delayWriter* and *delayReader* respectively be a DelayWriter and a DelayReader, for *delay*. Add *delayWriter* and *delayReader* to *nodes*. Disconnect *delay* from all its input and outputs.

                NOTE: This breaks the cycle: if a `DelayNode` is in a cycle, its two ends can be considered separately, because delay lines cannot be smaller than one render quantum when in a cycle.
        7. If *nodes* contains cycles, mute all the `AudioNode`s that are part of this cycle, and remove them from *nodes*.
        8. Consider all elements in *nodes* to be unmarked. While there are unmarked elements in *nodes*:
            1. Choose an element *node* in *nodes*.
            2. Visit *node*.

        Visiting a node means performing the following steps:
            1. If *node* is marked, abort these steps.
            2. Mark *node*.

Error preparing HTML-CSS output (preProcess)

3. If *node* is an `AudioNode`, Visit each `AudioNode` connected to the input of *node*.
4. For each `AudioParam` *param* of *node*:
    1. For each `AudioNode` *param input node* connected to *param*:
        1. Visit *param input node*
5. Add *node* to the beginning of *ordered node list*.

9. Reverse the order of *ordered node list*.

3. Compute the value(s) of the `AudioListener`'s `AudioParam`s for this block.
4. For each `AudioNode`, in *ordered node list*:
    1. For each `AudioParam` of this `AudioNode`, execute these steps:
        1. If this `AudioParam` has any `AudioNode` connected to it, sum the buffers made available for reading by all `AudioNode` connected to this `AudioParam`, down mix the resulting buffer down to a mono channel, and call this buffer the ***input AudioParam buffer***.
        2. Compute the value(s) of this `AudioParam` for this block.
        3. Queue a control message to set the `[[current value]]` slot of this `AudioParam` according to § 1.6.3 Computation of Value.
    2. If this `AudioNode` has any `AudioNode`s connected to its input, sum the buffers made available for reading by all `AudioNode`s connected to this `AudioNode`. The resulting buffer is called the ***input buffer***. Up or down-mix it to match if number of input channels of this `AudioNode`.
    3. If this `AudioNode` is a source node, compute a block of audio, and make it available for reading.
    4. If this `AudioNode` is an `AudioWorkletNode`, execute these substeps:
        1. Let *processor* be the associated `AudioWorkletProcessor` instance of `AudioWorkletNode`.
        2. Let *O* be the ECMAScript object corresponding to *processor*.
        3. Let *processCallback* be an uninitialized variable.
        4. Let *completion* be an uninitialized variable.
        5. Prepare to run script with the current settings object.
        6. Prepare to run a callback with the current settings object.
        7. Let *getResult* be Get(*O*, "process").
        8. If *getResult* is an abrupt completion, set *completion* to *getResult* and jump to the step labeled return.
        9. Set *processCallback* to *getResult*.[[Value]].
        10. If ! IsCallable(*processCallback*) is `false`, then:
            1. Set *completion* to new Completion {[[Type]]: throw, [[Value]]: a newly created TypeError object, [[Target]]: empty}.
            2. Jump to the step labeled return.
        11. Set `[[callable process]]` to `true`.
        12. Perform the following substeps:
            1. Let *args* be a Web IDL arguments list consisting of `inputs`, `outputs`, and `parameters`.
            2. Let *esArgs* be the result of converting *args* to an ECMAScript arguments list.
            3. Let *callResult* be the Call(*processCallback*, *O*, *esArgs*). This operation computes a block of audio with *esArgs*. Upon a successful function call, a buffer containing copies of the elements of the `Float32Array`s passed via the `outputs` is made available for reading. Any `Promise` resolved within this call will be queued into the microtask queue in the `AudioWorkletGlobalScope`.
            4. If *callResult* is an abrupt completion, set *completion* to *callResult* and jump to the step labeled return.
            5. Set *processor*'s active source flag to ToBoolean(*callResult*.[[Value]]).
        13. **Return:** at this point *completion* will be set to an ECMAScript completion value.

Error preparing HTML-CSS output (preProcess)

1. Clean up after running a callback with the current settings object.
2. Clean up after running script with the current settings object.
3. If *completion* is an abrupt completion:
    1. Set `[[callable process]]` to `false`.
    2. Set *processor*'s active source flag to `false`.
    3. Make a silent output buffer available for reading.
    4. Queue a task to the control thread fire an `ErrorEvent` named `processorerror` at the associated `AudioWorkletNode`.
5. If this `AudioNode` is a destination node, record the input of this `AudioNode`.
6. Else, process the input buffer, and make available for reading the resulting buffer.

5. Atomically perform the following steps:
    1. Increment `[[current frame]]` by the render quantum size.
    2. Set `currentTime` to `[[current frame]]` divided by `sampleRate`.
6. Set *render result* to `true`.

5. Perform a microtask checkpoint.
6. Return *render result*.

Muting an `AudioNode` means that its output MUST be silence for the rendering of this audio block.

Making a buffer available for reading from an `AudioNode` means putting it in a state where other `AudioNode`s connected to this `AudioNode` can safely read from it.

> NOTE: For example, implementations can choose to allocate a new buffer, or have a more elaborate mechanism, reusing an existing buffer that is now unused.

Recording the input of an `AudioNode` means copying the input data of this `AudioNode` for future usage.

Computing a block of audio means running the algorithm for this `AudioNode` to produce 128 sample-frames.

Processing an input buffer means running the algorithm for an `AudioNode`, using an input buffer and the value(s) of the `AudioParam`(s) of this `AudioNode` as the input for this algorithm.

§ 2.5. Unloading a document

Additional unloading document cleanup steps are defined for documents that use `BaseAudioContext`:

1. Reject all the promises of `[[pending promises]]` with `InvalidStateError`, for each `AudioContext` and `OfflineAudioContext` whose relevant global object is the same as the document's associated Window.
2. Stop all `decoding thread`s.
3. Queue a control message to `close()` the `AudioContext` or `OfflineAudioContext`.

§ 3. Dynamic Lifetime

§ 3.1. Background

> NOTE: The normative description of `AudioContext` and `AudioNode` lifetime characteristics is described by the AudioContext lifetime and AudioNode lifetime.

This section is non-normative.

In addition to allowing the creation of static routing configurations, it should also be possible to do custom effect routing on dynamically allocated voices which have a limited lifetime. For the purposes of this discussion, let's call these short-lived

Error preparing HTML-CSS output (preProcess)

voices "notes". Many audio applications incorporate the ideas of notes, examples being drum machines, sequencers, and 3D games with many one-shot sounds being triggered according to game play.

In a traditional software synthesizer, notes are dynamically allocated and released from a pool of available resources. The note is allocated when a MIDI note-on message is received. It is released when the note has finished playing either due to it having reached the end of its sample-data (if non-looping), it having reached a sustain phase of its envelope which is zero, or due to a MIDI note-off message putting it into the release phase of its envelope. In the MIDI note-off case, the note is not released immediately, but only when the release envelope phase has finished. At any given time, there can be a large number of notes playing but the set of notes is constantly changing as new notes are added into the routing graph, and old ones are released.

The audio system automatically deals with tearing-down the part of the routing graph for individual "note" events. A "note" is represented by an `AudioBufferSourceNode`, which can be directly connected to other processing nodes. When the note has finished playing, the context will automatically release the reference to the `AudioBufferSourceNode`, which in turn will release references to any nodes it is connected to, and so on. The nodes will automatically get disconnected from the graph and will be deleted when they have no more references. Nodes in the graph which are long-lived and shared between dynamic voices can be managed explicitly. Although it sounds complicated, this all happens automatically with no extra handling required.

§ 3.2. Example

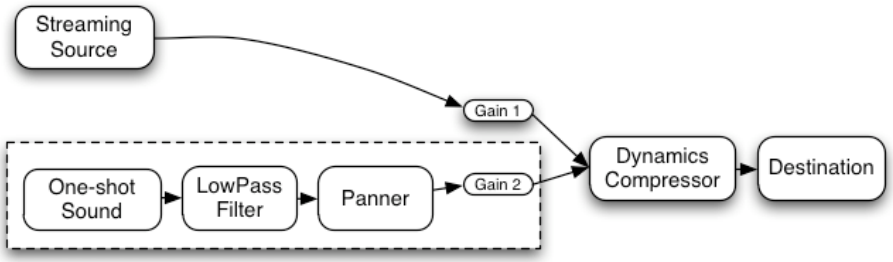


Figure 18 *A graph featuring a subgraph that will be releases early.*

The low-pass filter, panner, and second gain nodes are directly connected from the one-shot sound. So when it has finished playing the context will automatically release them (everything within the dotted line). If there are no longer any references to the one-shot sound and connected nodes, then they will be immediately removed from the graph and deleted. The

Error preparing HTML-CSS output (preProcess)

streaming source has a global reference and will remain connected until it is explicitly disconnected. Here's how it might look in JavaScript:

```
EXAMPLE 20
let context = 0;
let compressor = 0;
let gainNode1 = 0;
let streamingAudioSource = 0;

// Initial setup of the "long-lived" part of the routing graph
function setupAudioContext() {
        context = new AudioContext();

        compressor = context.createDynamicsCompressor();
        gainNode1 = context.createGain();

        // Create a streaming audio source.
        const audioElement = document.getElementById('audioTagID');
        streamingAudioSource = context.createMediaElementSource(audioElement);
        streamingAudioSource.connect(gainNode1);

        gainNode1.connect(compressor);
        compressor.connect(context.destination);
}

// Later in response to some user action (typically mouse or key event)
// a one-shot sound can be played.
function playSound() {
        const oneShotSound = context.createBufferSource();
        oneShotSound.buffer = dogBarkingBuffer;

        // Create a filter, panner, and gain node.
        const lowpass = context.createBiquadFilter();
        const panner = context.createPanner();
        const gainNode2 = context.createGain();

        // Make connections
        oneShotSound.connect(lowpass);
        lowpass.connect(panner);
        panner.connect(gainNode2);
        gainNode2.connect(compressor);

        // Play 0.75 seconds from now (to play immediately pass in 0)
        oneShotSound.start(context.currentTime + 0.75);
}
```

§ 4. Channel Up-Mixing and Down-Mixing

This section is normative.

An AudioNode input has ***mixing rules*** for combining the channels from all of the connections to it. As a simple example, if an input is connected from a mono output and a stereo output, then the mono connection will usually be up-mixed to stereo and summed with the stereo connection. But, of course, it's important to define the exact mixing rules for every input to every `AudioNode`. The default mixing rules for all of the inputs have been chosen so that things "just work" without worrying too much about the details, especially in the very common case of mono and stereo streams. Of course, the rules can be changed for advanced use cases, especially multi-channel.

To define some terms, ***up-mixing*** refers to the process of taking a stream with a smaller number of channels and converting it to a stream with a larger number of channels. ***down-mixing*** refers to the process of taking a stream with a larger number of channels and converting it to a stream with a smaller number of channels.

Error preparing HTML-CSS output (preProcess)

ds to mix all the outputs connected to this input. As part of this process it computes an internal value computedNumberOfChannels representing the actual number of channels of the input at any given time.

For each input of an AudioNode, an implementation MUST:

1. Compute computedNumberOfChannels.
2. For each connection to the input:

    1. up-mix or down-mix the connection to computedNumberOfChannels according to the ChannelInterpretation value given by the node's channelInterpretation attribute.
    2. Mix it together with all of the other mixed streams (from other connections). This is a straight-forward summing together of each of the corresponding channels that have been up-mixed or down-mixed in step 1 for each connection.

§ 4.1. Speaker Channel Layouts

When channelInterpretation is "speakers" then the up-mixing and down-mixing is defined for specific channel layouts.

Mono (one channel), stereo (two channels), quad (four channels), and 5.1 (six channels) MUST be supported. Other channel layouts may be supported in future version of this specification.

§ 4.2. Channel Ordering

Channel ordering is defined by the following table. Individual multichannel formats MAY not support all intermediate channels. Implementations MUST present the channels provided in the order defined below, skipping over those channels not present.

Order	Label	Mono	Stereo	Quad	5.1
0	SPEAKER_FRONT_LEFT	0	0	0	0
1	SPEAKER_FRONT_RIGHT		1	1	1
2	SPEAKER_FRONT_CENTER				2
3	SPEAKER_LOW_FREQUENCY				3
4	SPEAKER_BACK_LEFT			2	4
5	SPEAKER_BACK_RIGHT			3	5
6	SPEAKER_FRONT_LEFT_OF_CENTER				
7	SPEAKER_FRONT_RIGHT_OF_CENTER				
8	SPEAKER_BACK_CENTER				
9	SPEAKER_SIDE_LEFT				
10	SPEAKER_SIDE_RIGHT				
11	SPEAKER_TOP_CENTER				
12	SPEAKER_TOP_FRONT_LEFT				
13	SPEAKER_TOP_FRONT_CENTER				
14	SPEAKER_TOP_FRONT_RIGHT				
15	SPEAKER_TOP_BACK_LEFT				
16	SPEAKER_TOP_BACK_CENTER				
17	SPEAKER_TOP_BACK_RIGHT				

§ 4.3. Implication of tail-time on input and output channel count

When an AudioNode has a non-zero tail-time, and an output channel count that depends on the input channels count, the AudioNode's tail-time must be taken into account when the input channel count changes.

When there is a decrease in input channel count, the change in output channel count MUST happen when the input that was received with greater channel count no longer affects the output.

When there is an increase in input channel count, the behavior depends on the AudioNode type:

- For a DelayNode or a DynamicsCompressorNode, the number of output channels MUST increase when the input [illegible]th greater channel count begins to affect the output.

Error preparing HTML-CSS output (preProcess)

- For other `AudioNode`s that have a tail-time, the number of output channels MUST increase immediately.

  > NOTE: For a `ConvolverNode`, this only applies to the case where the impulse response is mono. Otherwise, the `ConvolverNode` always outputs a stereo signal regardless of its input channel count.

> NOTE: Intuitively, this allows not losing stereo information as part of processing: when multiple input render quanta of different channel count contribute to an output render quantum then the output render quantum's channel count is a superset of the input channel count of the input render quantums.

§ 4.4. Up Mixing Speaker Layouts

```
Mono up-mix:

    1 -> 2 : up-mix from mono to stereo
        output.L = input;
        output.R = input;

    1 -> 4 : up-mix from mono to quad
        output.L = input;
        output.R = input;
        output.SL = 0;
        output.SR = 0;

    1 -> 5.1 : up-mix from mono to 5.1
        output.L = 0;
        output.R = 0;
        output.C = input; // put in center channel
        output.LFE = 0;
        output.SL = 0;
        output.SR = 0;

Stereo up-mix:

    2 -> 4 : up-mix from stereo to quad
        output.L = input.L;
        output.R = input.R;
        output.SL = 0;
        output.SR = 0;

    2 -> 5.1 : up-mix from stereo to 5.1
        output.L = input.L;
        output.R = input.R;
        output.C = 0;
        output.LFE = 0;
        output.SL = 0;
        output.SR = 0;

Quad up-mix:

    4 -> 5.1 : up-mix from quad to 5.1
        output.L = input.L;
        output.R = input.R;
        output.C = 0;
        output.LFE = 0;
        output.SL = input.SL;
        output.SR = input.SR;
```

Error preparing HTML-CSS output (preProcess)

§ 4.5. Down Mixing Speaker Layouts

A down-mix will be necessary, for example, if processing 5.1 source material, but playing back stereo.

```
Mono down-mix:

    2 -> 1 : stereo to mono
        output = 0.5 * (input.L + input.R);

    4 -> 1 : quad to mono
        output = 0.25 * (input.L + input.R + input.SL + input.SR);

    5.1 -> 1 : 5.1 to mono
        output = sqrt(0.5) * (input.L + input.R) + input.C + 0.5 * (input.SL + input.SR)

Stereo down-mix:

    4 -> 2 : quad to stereo
        output.L = 0.5 * (input.L + input.SL);
        output.R = 0.5 * (input.R + input.SR);

    5.1 -> 2 : 5.1 to stereo
        output.L = L + sqrt(0.5) * (input.C + input.SL)
        output.R = R + sqrt(0.5) * (input.C + input.SR)

Quad down-mix:

    5.1 -> 4 : 5.1 to quad
        output.L = L + sqrt(0.5) * input.C
        output.R = R + sqrt(0.5) * input.C
        output.SL = input.SL
        output.SR = input.SR
```

§ 4.6. Channel Rules Examples

EXAMPLE 21

```
// Set gain node to explicit 2-channels (stereo).
gain.channelCount = 2;
gain.channelCountMode = "explicit";
gain.channelInterpretation = "speakers";

// Set "hardware output" to 4-channels for DJ-app with two stereo output busses.
context.destination.channelCount = 4;
context.destination.channelCountMode = "explicit";
context.destination.channelInterpretation = "discrete";

// Set "hardware output" to 8-channels for custom multi-channel speaker array
// with custom matrix mixing.
context.destination.channelCount = 8;
context.destination.channelCountMode = "explicit";
context.destination.channelInterpretation = "discrete";

// Set "hardware output" to 5.1 to play an HTMLAudioElement.
context.destination.channelCount = 6;
context.destination.channelCountMode = "explicit";
context.destination.channelInterpretation = "speakers";

// Explicitly down-mix to mono.
gain.channelCount = 1;
gain.channelCountMode = "explicit";
gain.channelInterpretation = "speakers";
```

Error preparing HTML-CSS output (preProcess)

§ 5. Audio Signal Values

§ 5.1. Audio sample format

Linear pulse code modulation (linear PCM) describes a format where the audio values are sampled at a regular interval, and where the quantization levels between two successive values are linearly uniform.

Whenever signal values are exposed to script in this specification, they are in linear 32-bit floating point pulse code modulation format (linear 32-bit float PCM), often in the form of `Float32Array` objects.

§ 5.2. Rendering

The range of all audio signals at a destination node of any audio graph is nominally [-1, 1]. The audio rendition of signal values outside this range, or of the values `NaN`, positive infinity or negative infinity, is undefined by this specification.

§ 6. Spatialization/Panning

§ 6.1. Background

A common feature requirement for modern 3D games is the ability to dynamically spatialize and move multiple audio sources in 3D space. For example OpenAL has this ability.

Using a `PannerNode`, an audio stream can be spatialized or positioned in space relative to an `AudioListener`. A `BaseAudioContext` will contain a single `AudioListener`. Both panners and listeners have a position in 3D space using a right-handed cartesian coordinate system. The units used in the coordinate system are not defined, and do not need to be because the effects calculated with these coordinates are independent/invariant of any particular units such as meters or feet. `PannerNode` objects (representing the source stream) have an *orientation* vector representing in which direction the sound is projecting. Additionally, they have a *sound cone* representing how directional the sound is. For example, the sound could be omnidirectional, in which case it would be heard anywhere regardless of its orientation, or it can be more directional and heard only if it is facing the listener. `AudioListener` objects (representing a person's ears) have forward and up vectors representing in which direction the person is facing.

The coordinate system for spatialization is shown in the diagram below, with the default values shown. The locations for the `AudioListener` and `PannerNode` are moved from the default positions so we can see things better.

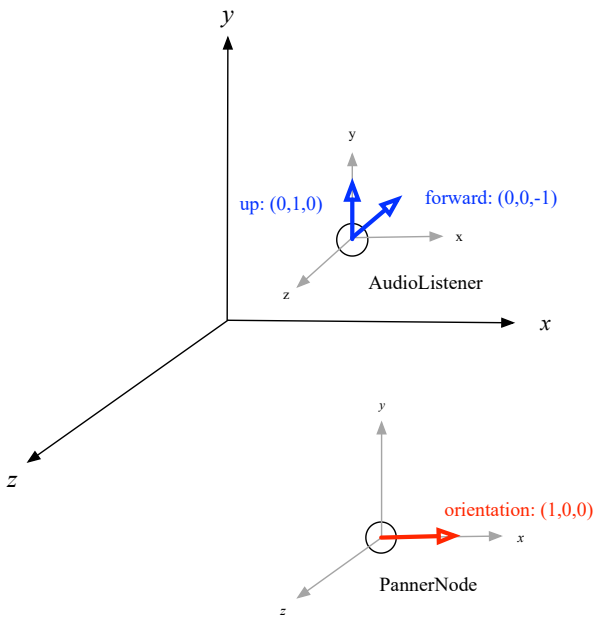


Figure 19 *Diagram of the coordinate system with AudioListener and PannerNode attributes shown.*

Error preparing HTML-CSS output (preProcess)

During rendering, the `PannerNode` calculates an *azimuth* and *elevation*. These values are used internally by the implementation in order to render the spatialization effect. See the Panning Algorithm section for details of how these values are used.

§ 6.2. Azimuth and Elevation

The following algorithm MUST be used to calculate the *azimuth* and *elevation* for the `PannerNode`. The implementation must appropriately account for whether the various `AudioParam`s below are "`a-rate`" or "`k-rate`".

```
// Let |context| be a BaseAudioContext and let |panner| be a
// PannerNode created in |context|.

// Calculate the source-listener vector.
const listener = context.listener;
const sourcePosition = new Vec3(panner.positionX.value, panner.positionY.value,
                                panner.positionZ.value);
const listenerPosition =
    new Vec3(listener.positionX.value, listener.positionY.value,
             listener.positionZ.value);
const sourceListener = sourcePosition.diff(listenerPosition).normalize();

if (sourceListener.magnitude == 0) {
  // Handle degenerate case if source and listener are at the same point.
  azimuth = 0;
  elevation = 0;
  return;
}

// Align axes.
const listenerForward = new Vec3(listener.forwardX.value, listener.forwardY.value,
                                 listener.forwardZ.value);
const listenerUp =
    new Vec3(listener.upX.value, listener.upY.value, listener.upZ.value);
const listenerRight = listenerForward.cross(listenerUp);

if (listenerRight.magnitude == 0) {
  // Handle the case where listener's 'up' and 'forward' vectors are linearly
  // dependent, in which case 'right' cannot be determined
  azimuth = 0;
  elevation = 0;
  return;
}

// Determine a unit vector orthogonal to listener's right, forward
const listenerRightNorm = listenerRight.normalize();
const listenerForwardNorm = listenerForward.normalize();
const up = listenerRightNorm.cross(listenerForwardNorm);

const upProjection = sourceListener.dot(up);
const projectedSource = sourceListener.diff(up.scale(upProjection)).normalize();

azimuth = 180 * Math.acos(projectedSource.dot(listenerRightNorm)) / Math.PI;

// Source in front or behind the listener.
const frontBack = projectedSource.dot(listenerForwardNorm);
if (frontBack < 0)
  azimuth = 360 - azimuth;

// Make azimuth relative to "forward" and not "right" listener vector.
if ((azimuth >= 0) && (azimuth <= 270))
  azimuth = 90 - azimuth;
else
  azimuth = 450 - azimuth;

```

Error preparing HTML-CSS output (preProcess)

```
56 elevation = 90 - 180 * Math.acos(sourceListener.dot(up)) / Math.PI;
57
58 if (elevation > 90)
59   elevation = 180 - elevation;
60 else if (elevation < -90)
61   elevation = -180 - elevation;
```

§ 6.3. Panning Algorithm

Mono-to-stereo and *stereo-to-stereo* panning MUST be supported. *Mono-to-stereo* processing is used when all connections to the input are mono. Otherwise *stereo-to-stereo* processing is used.

§ 6.3.1. PannerNode "equalpower" Panning

This is a simple and relatively inexpensive algorithm which provides basic, but reasonable results. It is used for the for the `PannerNode` when the `panningModel` attribute is set to "`equalpower`", in which case the *elevation* value is ignored. This algorithm MUST be implemented using the appropriate rate as specified by the `automationRate`. If any of the `PannerNode`'s `AudioParam`s or the `AudioListener`'s `AudioParam`s are "`a-rate`", a-rate processing must be used.

1. For each sample to be computed by this `AudioNode`:

    1. Let *azimuth* be the value computed in the azimuth and elevation section.
    2. The *azimuth* value is first contained to be within the range [-90, 90] according to:

    ```
    // First, clamp azimuth to allowed range of [-180, 180].
    azimuth = max(-180, azimuth);
    azimuth = min(180, azimuth);

    // Then wrap to range [-90, 90].
    if (azimuth < -90)
        azimuth = -180 - azimuth;
    else if (azimuth > 90)
        azimuth = 180 - azimuth;
    ```

    3. A normalized value *x* is calculated from *azimuth* for a mono input as:

    ```
    x = (azimuth + 90) / 180;
    ```

    Or for a stereo input as:

    ```
    if (azimuth <= 0) { // -90 -> 0
        // Transform the azimuth value from [-90, 0] degrees into the range [-90, 90].
        x = (azimuth + 90) / 90;
    } else { // 0 -> 90
        // Transform the azimuth value from [0, 90] degrees into the range [-90, 90].
        x = azimuth / 90;
    }
    ```

    4. Left and right gain values are calculated as:

    ```
    gainL = cos(x * Math.PI / 2);
    gainR = sin(x * Math.PI / 2);
    ```

    5. For mono input, the stereo output is calculated as:

    ```
    outputL = input * gainL;
    outputR = input * gainR;
    ```

Error preparing HTML-CSS output (preProcess)

Else for stereo input, the output is calculated as:

```
if (azimuth <= 0) {
    outputL = inputL + inputR * gainL;
    outputR = inputR * gainR;
} else {
    outputL = inputL * gainL;
    outputR = inputR + inputL * gainR;
}
```

6. Apply the distance gain and cone gain where the computation of the distance is described in Distance Effects and the cone gain is described in Sound Cones:

```
let distance = distance();
let distanceGain = distanceModel(distance);
let totalGain = coneGain() * distanceGain();
outputL = totalGain * outputL;
outputR = totalGain * outputR;
```

§ 6.3.2. PannerNode "HRTF" Panning (Stereo Only)

This requires a set of HRTF (Head-related Transfer Function) impulse responses recorded at a variety of azimuths and elevations. The implementation requires a highly optimized convolution function. It is somewhat more costly than "equalpower", but provides more perceptually spatialized sound.

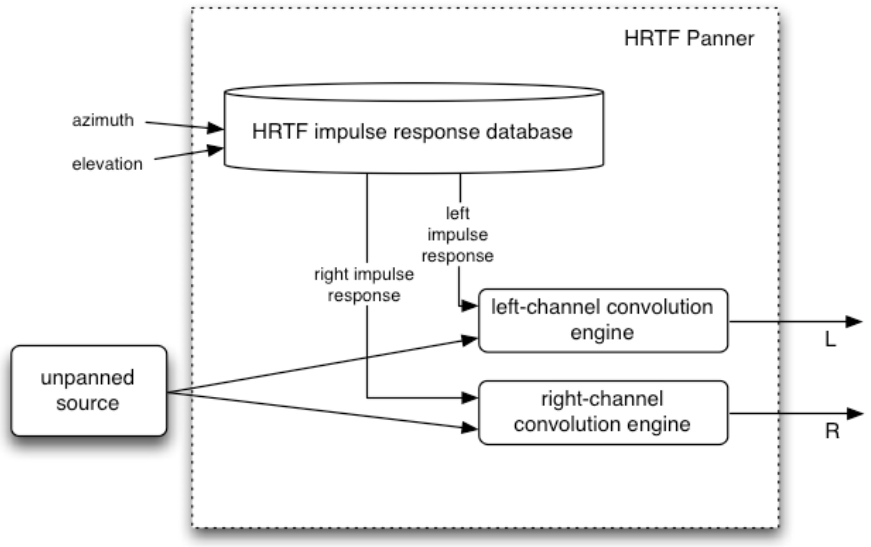


Figure 20 *A diagram showing the process of panning a source using HRTF.*

§ 6.3.3. StereoPannerNode Panning

For a `StereoPannerNode`, the following algorithm MUST be implemented.

1. For each sample to be computed by this `AudioNode`

    1. Let *pan* be the computedValue of the `pan` `AudioParam` of this `StereoPannerNode`.
    2. Clamp *pan* to [-1, 1].

```
pan = max(-1, pan);
pan = min(1, pan);
```

Error preparing HTML-CSS output (preProcess)

3. Calculate *x* by normalizing *pan* value to [0, 1]. For mono input:

```
x = (pan + 1) / 2;
```

For stereo input:

```
if (pan <= 0)
    x = pan + 1;
else
    x = pan;
```

4. Left and right gain values are calculated as:

```
gainL = cos(x * Math.PI / 2);
gainR = sin(x * Math.PI / 2);
```

5. For mono input, the stereo output is calculated as:

```
outputL = input * gainL;
outputR = input * gainR;
```

Else for stereo input, the output is calculated as:

```
if (pan <= 0) {
    outputL = inputL + inputR * gainL;
    outputR = inputR * gainR;
} else {
    outputL = inputL * gainL;
    outputR = inputR + inputL * gainR;
}
```

§ 6.4. Distance Effects

Sounds which are closer are louder, while sounds further away are quieter. Exactly *how* a sound's volume changes according to distance from the listener depends on the `distanceModel` attribute.

During audio rendering, a *distance* value will be calculated based on the panner and listener positions according to:

```
function distance(panner) {
  const pannerPosition = new Vec3(panner.positionX.value, panner.positionY.value,
                                  panner.positionZ.value);
  const listener = context.listener;
  const listenerPosition =
      new Vec3(listener.positionX.value, listener.positionY.value,
               listener.positionZ.value);
  return pannerPosition.diff(listenerPosition).magnitude;
}
```

distance will then be used to calculate *distanceGain* which depends on the `distanceModel` attribute. See the `DistanceModelType` section for details of how this is calculated for each distance model.

As part of its processing, the `PannerNode` scales/multiplies the input audio signal by *distanceGain* to make distant sounds quieter and nearer ones louder.

§ 6.5. Sound Cones

The listener and each sound source have an orientation vector describing which way they are facing. Each sound source's sound projection characteristics are described by an inner and outer "cone" describing the sound intensity as a function of

the source/listener angle from the source's orientation vector. Thus, a sound source pointing directly at the listener will be louder than if it is pointed off-axis. Sound sources can also be omni-directional.

The following diagram ilustrates the relationship between the source's cone with respect to the listener. In the diagram, `coneInnerAngle = 50` and `coneOuterAngle = 120`. That is, the inner cone extends 25 deg on each side of the direction vector. Similarly, the outer cone is 60 deg on each side.

Figure 21 *Cone angles for a source in relationship to the source orientation and the listeners position and orientation.*

The following algorithm MUST be used to calculate the gain contribution due to the cone effect, given the source (the `PannerNode`) and the listener:

```
function coneGain() {
  const sourceOrientation =
      new Vec3(source.orientationX, source.orientationY, source.orientationZ);
  if (sourceOrientation.magnitude == 0 ||
      ((source.coneInnerAngle == 360) && (source.coneOuterAngle == 360)))
    return 1; // no cone specified - unity gain

  // Normalized source-listener vector
  const sourcePosition = new Vec3(panner.positionX.value, panner.positionY.value,
                                  panner.positionZ.value);
  const listenerPosition =
      new Vec3(listener.positionX.value, listener.positionY.value,
               listener.positionZ.value);
  const sourceToListener = sourcePosition.diff(listenerPosition).normalize();

  const normalizedSourceOrientation = sourceOrientation.normalize();

  // Angle between the source orientation vector and the source-listener vector
  const angle = 180 *
                Math.acos(sourceToListener.dot(normalizedSourceOrientation)) /
                Math.PI;
  const absAngle = Math.abs(angle);
```

Error preparing HTML-CSS output (preProcess)

```
23
24   // Divide by 2 here since API is entire angle (not half-angle)
25   const absInnerAngle = Math.abs(source.coneInnerAngle) / 2;
26   const absOuterAngle = Math.abs(source.coneOuterAngle) / 2;
27   let gain = 1;
28
29   if (absAngle <= absInnerAngle) {
30     // No attenuation
31     gain = 1;
32   } else if (absAngle >= absOuterAngle) {
33     // Max attenuation
34     gain = source.coneOuterGain;
35   } else {
36     // Between inner and outer cones
37     // inner -> outer, x goes from 0 -> 1
38     const x = (absAngle - absInnerAngle) / (absOuterAngle - absInnerAngle);
39     gain = (1 - x) + source.coneOuterGain * x;
40   }
41
42   return gain;
43 }
```

§ 7. Performance Considerations

§ 7.1. Latency

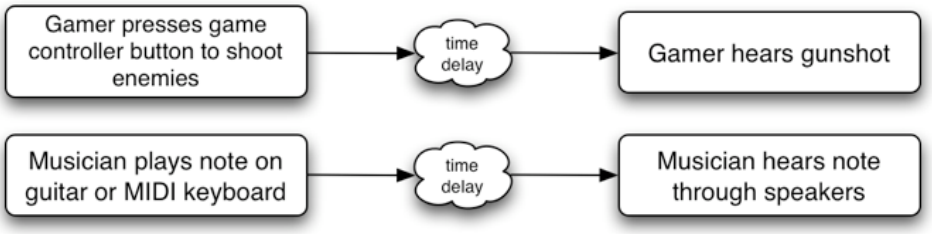


Figure 22 *Use cases in which the latency can be important*

For web applications, the time delay between mouse and keyboard events (keydown, mousedown, etc.) and a sound being heard is important.

This time delay is called latency and is caused by several factors (input device latency, internal buffering latency, DSP processing latency, output device latency, distance of user's ears from speakers, etc.), and is cumulative. The larger this latency is, the less satisfying the user's experience is going to be. In the extreme, it can make musical production or game-play impossible. At moderate levels it can affect timing and give the impression of sounds lagging behind or the game being non-responsive. For musical applications the timing problems affect rhythm. For gaming, the timing problems affect precision of gameplay. For interactive applications, it generally cheapens the users experience much in the same way that very low animation frame-rates do. Depending on the application, a reasonable latency can be from as low as 3-6 milliseconds to 25-50 milliseconds.

Implementations will generally seek to minimize overall latency.

Along with minimizing overall latency, implementations will generally seek to minimize the difference between an `AudioContext`'s `currentTime` and an `AudioProcessingEvent`'s `playbackTime`. Deprecation of `ScriptProcessorNode` will make this consideration less important over time.

Additionally, some `AudioNode`s can add latency to some paths of the audio graph, notably:

- The `AudioWorkletNode` can run a script that buffers internally, adding delay to the signal path.

Error preparing HTML-CSS output (preProcess)

hose role is to add controlled latency time.

- The `BiquadFilterNode` and `IIRFilterNode` filter design can delay incoming samples, as a natural consequence of the causal filtering process.
- The `ConvolverNode` depending on the impulse, can delay incoming samples, as a natural result of the convolution operation.
- The `DynamicsCompressorNode` has a look ahead algorithm that causes delay in the signal path.
- The `MediaStreamAudioSourceNode`, `MediaStreamTrackAudioSourceNode` and `MediaStreamAudioDestinationNode`, depending on the implementation, can add buffers internally that add delays.
- The `ScriptProcessorNode` can have buffers between the control thread and the rendering thread.
- The `WaveShaperNode`, when oversampling, and depending on the oversampling technique, add delays to the signal path.

§ 7.2. Audio Buffer Copying

When an acquire the content operation is performed on an `AudioBuffer`, the entire operation can usually be implemented without copying channel data. In particular, the last step SHOULD be performed lazily at the next `getChannelData()` call. That means a sequence of consecutive acquire the contents operations with no intervening `getChannelData()` (e.g. multiple `AudioBufferSourceNode`s playing the same `AudioBuffer`) can be implemented with no allocations or copying.

Implementations can perform an additional optimization: if `getChannelData()` is called on an `AudioBuffer`, fresh `ArrayBuffer`s have not yet been allocated, but all invokers of previous acquire the content operations on an `AudioBuffer` have stopped using the `AudioBuffer`'s data, the raw data buffers can be recycled for use with new `AudioBuffer`s, avoiding any reallocation or copying of the channel data.

§ 7.3. AudioParam Transitions

While no automatic smoothing is done when directly setting the `value` attribute of an `AudioParam`, for certain parameters, smooth transition are preferable to directly setting the value.

Using the `setTargetAtTime()` method with a low `timeConstant` allows authors to perform a smooth transition.

§ 7.4. Audio Glitching

Audio glitches are caused by an interruption of the normal continuous audio stream, resulting in loud clicks and pops. It is considered to be a catastrophic failure of a multi-media system and MUST be avoided. It can be caused by problems with the threads responsible for delivering the audio stream to the hardware, such as scheduling latencies caused by threads not having the proper priority and time-constraints. It can also be caused by the audio DSP trying to do more work than is possible in real-time given the CPU's speed.

§ 8. Security and Privacy Considerations

Per the Self-Review Questionnaire: Security and Privacy § questions:

1. Does this specification deal with personally-identifiable information?

   It would be possible to perform a hearing test using Web Audio API, thus revealing the range of frequencies audible to a person (this decreases with age). It is difficult to see how this could be done without the realization and consent of the user, as it requires active particpation.

2. Does this specification deal with high-value data?

   No. Credit card information and the like is not used in Web Audio. It is possible to use Web Audio to process or analyze voice data, which might be a privacy concern, but access to the user's microphone is permission-based via `getUserMedia()`.

Error preparing HTML-CSS output (preProcess) on introduce new state for an origin that persists across browsing sessions?

No. AudioWorklet does not persist across browsing sessions.

4. Does this specification expose persistent, cross-origin state to the web?

   Yes, the supported audio sample rate(s) and the output device channel count are exposed. See `AudioContext`.

5. Does this specification expose any other data to an origin that it doesn't currently have access to?

   Yes. When giving various information on available `AudioNode`s, the Web Audio API potentially exposes information on characteristic features of the client (such as audio hardware sample-rate) to any page that makes use of the `AudioNode` interface. Additionally, timing information can be collected through the `AnalyserNode` or `ScriptProcessorNode` interface. The information could subsequently be used to create a fingerprint of the client.

   Research by Princeton CITP's Web Transparency and Accountability Project has shown that `DynamicsCompressorNode` and `OscillatorNode` can be used to gather entropy from a client to fingerprint a device. This is due to small, and normally inaudible, differences in DSP architecture, resampling strategies and rounding trade-offs between differing implementations. The precise compiler flags used and also the CPU architecture (ARM vs. x86) contribute to this entropy.

   In practice however, this merely allows deduction of information already readily available by easier means (User Agent string), such as "this is browser X running on platform Y". However, to reduce the possibility of additional fingerprinting, we mandate browsers take action to mitigate fingerprinting issues that might be possible from the output of any node.

   Fingerprinting via clock skew has been described by Steven J Murdoch and Sebastian Zander. It might be possible to determine this from `getOutputTimestamp`. Skew-based fingerprinting has also been demonstrated by Nakibly et. al. for HTML. The *High Resolution Time § 10. Privacy Considerations* section should be consulted for further information on clock resolution and drift.

   Fingerprinting via latency is also possible; it might be possible to deduce this from `baseLatency` and `outputLatency`. Mitigation strategies include adding jitter (dithering) and quantization so that the exact skew is incorrectly reported. Note however that most audio systems aim for low latency, to synchronise the audio generated by WebAudio to other audio or video sources or to visual cues (for example in a game, or an audio recording or music making environment). Excessive latency decreases usability and may be an accessibility issue.

   Fingerprining via the sample rate of the `AudioContext` is also possible. We recommend the following steps to be taken to minimize this:

   1. 44.1 kHz and 48 kHz are allowed as default rates; the system will choose between them for best applicability. (Obviously, if the audio device is natively 44.1, 44.1 will be chosen, etc., but also the system may choose the most "compatible" rate—e.g. if the system is natively 96kHz, 48kHz would likely be chosen, not 44.1kHz.
   2. The system should resample to one of those two rates for devices that are natively at different rates, despite the fact that this may cause extra battery drain due to resampled audio. (Again, the system will choose the most compatible rate—e.g. if the native system is 16kHz, it's expected that 48kHz would be chosen.)
   3. It is expected (though not mandated) that browsers would offer a user affordance to force use of the native rate—e.g. by setting a flag in the browser on the device. This setting would not be exposed in the API.
   4. It is also expected behavior that a different rate could be explicitly requested in the constructor for `AudioContext` (this is already in the specification; it normally causes the audio rendering to be done at the requested sampleRate, and then up- or down-sampled to the device output), and if that rate is natively supported, the rendering could be passed straight through. This would enable apps to render to higher rates without user intervention (although it's not observable from Web Audio that the audio output is not downsampled on output)—for example, if `MediaDevices` capabilities were read (with user intervention) and indicated a higher rate was supported.

   Fingerprinting via the number of output channels for the `AudioContext` is possible as well. We recommend that `maxChannelCount` be set to two (stereo). Stereo is by far the most common number of channels.

6. Does this specification enable new script execution/loading mechanisms?

   No. It does use the [HTML] script execution method, defined in that specification.

7. Does this specification allow an origin access to a user's location?

Error preparing HTML-CSS output (preProcess)

8. Does this specification allow an origin access to sensors on a user's device?

   Not directly. Currently, audio input is not specified in this document, but it will involve gaining access to the client machine's audio input or microphone. This will require asking the user for permission in an appropriate way, probably via the `getUserMedia()` API.

   Additionally, the security and privacy considerations from the Media Capture and Streams specification should be noted. In particular, analysis of ambient audio or playing unique audio may enable identification of user location down to the level of a room or even simultaneous occupation of a room by disparate users or devices. Access to both audio output and audio input might also enable communication between otherwise partitioned contexts in one browser.

9. Does this specification allow an origin access to aspects of a user's local computing environment?

   Not directly; all requested sample rates are supported, with upsampling if needed. It is possible to use Media Capture and Streams to probe for supported audio sample rates with MediaTrackSupportedConstraints. This requires explicit user consent. This does provide a small measure of fingerprinting. However, in practice most consumer and prosumer devices use one of two standardized sample rates: 44.1kHz (originally used by CD) and 48kHz (originally used by DAT). Highly resource constrained devices may support the speech-quality 11kHz sample rate, and higher-end devices often support 88.2, 96, or even the audiophile 192kHz rate.

   Requiring all implementations to upsample to a single, commonly-supported rate such as 48kHz would increase CPU cost for no particular benefit, and requiring higher-end devices to use a lower rate would merely result in Web Audio being labelled as unsuitable for professional use.

10. Does this specification allow an origin access to other devices?

    It typically does not allow access to other networked devices (an exception in a high-end recording studio might be Dante networked devices, although these typically use a separate, dedicated network). It does of necessity allow access to the user's audio output device or devices, which are sometimes separate units to the computer.

    For voice or sound-actuated devices, Web Audio API *might* be used to control other devices. In addition, if the sound-operated device is sensitive to near ultrasonic frequencies, such control might not be audible. This possibility also exists with HTML, through either the <audio> or <video> element. At common audio sampling rates, there is (by design) insufficient headroom for much ultrasonic information:

    The limit of human hearing is usually stated as 20kHz. For a 44.1kHz sampling rate, the Nyquist limit is 22.05kHz. Given that a true brickwall filter cannot be physically realized, the space between 20kHz and 22.05kHz is used for a rapid rolloff filter to strongly attenuate all frequencies above Nyquist.

    At 48kHz sampling rate, there is still rapid attenuation in the 20kHz to 24kHz band (but it is easier to avoid phase ripple errors in the passband).

11. Does this specification allow an origin some measure of control over a user agent's native UI?

    If the UI has audio components, such as a voice assistant or screenreader, Web Audio API might be used to emulate aspects of the native UI to make an attack seem more like a local system event. This possibility also exists with HTML, through the <audio> element.

12. Does this specification expose temporary identifiers to the web?

    No.

13. Does this specification distinguish between behavior in first-party and third-party contexts?

    No.

14. How should this specification work in the context of a user agent's "incognito" mode?

    Not differently.

15. Does this specification persist data to a user's local device?

    No.

16. Does this specification have a "Security Considerations" and "Privacy Considerations" section?

    Yes (you are reading it).

17. Does this specification allow downgrading default security characteristics?

Error preparing HTML-CSS output (preProcess)

No.

§ 9. Requirements and Use Cases

Please see [webaudio-usecases].

§ 10. Common Definitions for Specification Code

This section describes common functions and classes employed by JavaScript code used within this specification.

```
// Three dimensional vector class.
class Vec3 {
    // Construct from 3 coordinates.
    constructor(x, y, z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    // Dot product with another vector.
    dot(v) {
        return (this.x * v.x) + (this.y * v.y) + (this.z * v.z);
    }

    // Cross product with another vector.
    cross(v) {
        return new Vec3((this.y * v.z) - (this.z * v.y),
            (this.z * v.x) - (this.x * v.z),
            (this.x * v.y) - (this.y * v.x));
    }

    // Difference with another vector.
    diff(v) {
        return new Vec3(this.x - v.x, this.y - v.y, this.z - v.z);
    }

    // Get the magnitude of this vector.
    get magnitude() {
        return Math.sqrt(dot(this));
    }

    // Get a copy of this vector multiplied by a scalar.
    scale(s) {
        return new Vec3(this.x * s, this.y * s, this.z * s);
    }

    // Get a normalized copy of this vector.
    normalize() {
        const m = magnitude;
        if (m == 0) {
            return new Vec3(0, 0, 0);
        }
        return scale(1 / m);
    }
}
```

Error preparing HTML-CSS output (preProcess)

§ 11. Change Log

§ 11.1. Changes since Recommendation of 17 Jun 2021

- Issue 2456: Add a MessagePort to the AudioWorkletGlobalScope
  - Proposed Addition 1
  - Proposed Addition 2
  - Proposed Addition 3
  - Proposed Addition 4
  - Proposed Addition 5
  - Proposed Addition 6
- Issue 2444: Add AudioRenderCapacity interface and related classes
  - Proposed Addition 1
  - Proposed Addition 2
  - Proposed Addition 3
- Issue 2361: Use new Web IDL buffer primitives
  - Proposed Correction 1
  - Proposed Correction 2
  - Proposed Correction 3
  - Proposed Correction 4
  - Proposed Correction 5
  - Proposed Correction 6
  - Proposed Correction 7
  - Proposed Correction 8
  - Proposed Correction 9
- Issue 127: RangeError is thrown only for negative cancelTime
  - Proposed Correction 1
- Issue 2359: Fix a typo in the VUMeterNode example
  - Proposed Correction 1
  - Proposed Correction 2
- Issue 2373: Fix a typo: initially instead of initialy
  - Proposed Correction 1
- Issue 2321: Warn about corrupted fils in decodeAudioData.
  - Proposed Correction 1
- Issue 2375: decodeAudioData only decodes the first track of a multi-track audio file.
  - Proposed Correction 1
- Issue 2475: Add definition for various concepts related to system-level audio callbacks
  - Proposed Addition 1
  - Issue 2400 : Access to a different output device: AudioContext.setSinkId()
    - Proposed Addition 1
    - Proposed Addition 2
    - Proposed Addition 3

Error preparing HTML-CSS output (preProcess) Addition 4

- Proposed Addition 5
- Proposed Addition 6
- Proposed Addition 7
- Proposed Addition 8
- Proposed Addition 9
- Proposed Addition 10
- Proposed Addition 11

§ 11.2. Since Proposed Recommendation of 6 May 2021

- Styling, status and boilerplate updates for Recommendation
- Updated links to RFC2119, High-Resolution Time, and Audio EQ Cookbook
- A spelling error was corrected

§ 11.3. Since Candidate Recommendation of 14 January 2021

- PR 2333: Update links to point to W3C versions
- PR 2334: Use bikeshed to link to ErrorEvent
- PR 2331: Add MIMESniff to normative references
- PR 2328: MediaStream must be resampled to match the context sample rate
- PR 2318: Restore empty of pending processor construction data after successful initialization of AudioWorkProcessor#port.
- PR 2317: Standardize h3/h4 interface and dictionary markup
- PR 2312: Rework description of control thread state and rendering thread state
- PR 2311: Adjust the steps to process a context's regular task queue
- PR 2310: OscillatorNode output is mono
- PR 2308: Refine phrasing for "allowed to start"
- PR 2307: Replace "queue a task" with "queue a media element task"
- PR 2306: Move some steps from AudioWorkletProcessor constructor to the instantiation algorithm
- PR 2304: Add required components for ES operations in the rendering loop
- PR 2301: Define when and how regular tasks are processed wrt the processing model
- PR 2286: Clean up ABSN start algorithm
- PR 2277: Fix compression curve diagram
- PR 2273: Clarify units used in threshold & knee value calculations
- PR 2256: ABSN extrapolates the last output
- PR 2250: Use FrozenArray for AudioWorkletProcessor process()
- PR 2298: Bikeshed HTML validation issues

§ 11.4. Since Candidate Recommendation of 11 June 2020

- PR 2202: Fixed wrong optionality of IIRFilterNode options
- Issue 2191: Restrict sounds beyond normal hearing
- PR 2210: Return rejected promise when the document is not fully active, for operations returning promises
- Issue 2191: Destination of request created by `addModule`
- Issue 2213: The message queue is for message running on the rendering thread.

Error preparing HTML-CSS output (preProcess) usive language in the spec

- PR 2219: Update more terminology in images and markdown documents
- Issue 2206: PannerNode.rollOffFactor with "linear" distance model is not clamped to [0, 1] in main browser engines
- Issue 2169: AudioParamDescriptor has member constraints that are redundant
- Issue 1457: [privacy] Exposing data to an origin: fingerprinting
- Issue 2061: Privacy re-review of latest changes
- Issue 2225: Describe "Planar versus interleaved buffers"
- Issue 2231: WaveShaper [[curve set]] not defined
- Issue 2240: Align with Web IDL specification
- Issue 2242: LInk to undefined instead of using `<code>`
- Issue 2227: Clarify buffer.copyToChannel() must be called before source.buffer = buffer else nothing is played
- PR 2253: Fix duplicated IDs for decode callbacks
- Issue 2252: When are promises in "[[pending resume promises]]" resolved?
- PR 2266: Prohibit arbitrary termination of AudioWorkletGlobalScopes

§ 11.5. Since Candidate Recommendation of 18 September 2018

- Issue 2193: Incorrect azimuth comparison in spatialization algorithm
- Issue 2192: Waveshaper curve interpolation algorithm incorrect
- Issue 2171: Allow not having get parameterDescriptors in an AudioWorkletProcessor
- Issue 2184: PannerNode refDistance description unclear
- Issue 2165: AudioScheduledSourceNode start algorithm incomplete
- Issue 2155: Restore changes accidentally reverted in bikeshed conversion
- Issue 2154: Exception for changing channelCountMode on ScriptProcessorNode does not match browsers
- Issue 2153: Exception for changing channelCount on ScriptProcessorNode does not match browsers
- Issue 2152: close() steps don't make sense
- Issue 2150: AudioBufferOptions requires throwing NotFoundError in cases that can't happen
- Issue 2149: MediaStreamAudioSourceNode constructor has weird check for AudioContext
- Issue 2148: IIRFilterOptions description makes impossible demands
- Issue 2147: PeriodicWave constructor examines lengths of things that might not be there
- Issue 2113: BiquadFilter gain lower bound can be lower.
- Issue 2096: Lifetime of pending processor construction data and exceptions in instantiation of AudioWorkletProcessor
- Issue 2087: Minor issues with BiquadFilter AudioParams
- Issue 2083: Missing text in WaveShaperNode?
- Issue 2082: WaveShaperNode curve interpolation incomplete
- Issue 2074: Should the AudioWorkletNode constructor invoke the algorithm for initializing an object that inherits from AudioNode?
- Issue 2073: Inconsistencies in constructor descriptions and factory method initialization
- Issue 2072: Clarification on `AudioBufferSourceNode` looping, and loop points
- Issue 2071: cancelScheduledValues with setValueCurveAtTime
- Issue 2060: Would it be helpful to restrict use of `AudioWorkletProcessor.port().postMessage()` in order to facilitate garbage collection?
- Issue 2051: Update to constructor operations
- Issue 2050: Restore ConvolverNode channel mixing configurability (up to 2 channels)
- Issue 2045: Should the check on `process()` be removed from `AudioWorkletGlobalScope.registerProcessor()`?
- Issue 2044: Remove `options` parameter from `AudioWorkletProcessor` constructor WebIDL

Error preparing HTML-CSS output (preProcess)

- Issue 2036: Remove `options` parameter of `AudioWorkletProcessor` constructor
- Issue 2035: De-duplicate initial value setting on AudioWorkletNode AudioParams
- Issue 2027: Revise "processor construction data" algorithm
- Issue 2021: AudioWorkletProcessor constructor leads to infinite recursion
- Issue 2018: There are still issues with the setup of an AudioWorkletNode's parameters
- Issue 2016: Clarify `parameters` in AudioWorkletProcessor.process()
- Issue 2011: AudioWorkletNodeOptions.processorOptions should not default to null.
- Issue 1989: Please update to Web IDL changes to optional dictionary defaulting
- Issue 1984: Handling of exceptions in audio worklet is not very clear
- Issue 1976: AudioWorkletProcessor's [[node reference]] seems to be write-only
- Issue 1972: parameterDescriptors handling during AudioWorkletNode initialization is probably wrong
- Issue 1971: AudioWorkletNode options serialization is underdefined
- Issue 1970: "active source" flag handling is a weird monkeypatch
- Issue 1969: It would be clearer if the various validation of AudioWorkletNodeOptions were an explicit step or set of steps
- Issue 1966: parameterDescriptors is not looked up by the AudioWorkletProcessor constructor
- Issue 1963: NewTarget check for AudioWorkletProcessor isn't actually possible with a Web IDL constructor
- Issue 1947: Spec is inconsistent about whether parameterDescriptors is an array or an iterable
- Issue 1946: Population of "node name to parameter descriptor map" needs to be defined
- Issue 1945: registerProcessor is doing odd things with threads and JS values
- Issue 1943: Describe how WaveShaperNode shapes the input with the curve
- Issue 1935: length of AudioWorkletProcessor.process() parameter sequences with inactive inputs
- Issue 1932: Make AudioWorkletNode output buffer available for reading
- Issue 1925: front vs forward
- Issue 1902: Mixer Gain Structure section not needed
- Issue 1906: Steps in rendering algorithm
- Issue 1905: Rendering callbacks are observable
- Issue 1904: Strange Note in algorithm for swapping a control message queue
- Issue 1903: Funny sentence about priority and latency
- Issue 1901: AudioWorkletNode state property?
- Issue 1900: AudioWorkletProcessor NewTarget undefined
- Issue 1899: Missing synchronous markers
- Issue 1897: WaveShaper curve value setter allows multiple sets
- Issue 1896: WaveShaperNode constructor says curve set is initialized to false
- Issue #1471: AudioNode Lifetime section seems to attempt to make garbage collection observable
- Issue #1893: Active processing for Panner/Convolver/ChannelMerger
- Issue #1894: Funny text in PannerNode.orientationX
- Issue #1866: References to garbage collection
- Issue #1851: Parameter values used for BiquadFilterNode::getFrequencyResponse
- Issue #1905: Rendering callbacks are observable
- Issue #1879: ABSN playback algorithm offset
- Issue #1882: Biquad lowpass/highpass Q
- Issue #1303: MediaElementAudioSourceNode information in a funny place
- Issue #1896: WaveShaperNode constructor says curve set is initialized to false

Error preparing HTML-CSS output (preProcess)

- Issue #1897: WaveShaper curve value setter allows multiple sets.

- Issue #1880: setOrientation description has confusing paragraph
- Issue #1855: createScriptProcessor parameter requirements
- Issue #1857: Fix typos and bad phrasing
- Issue #1788: Unclear what value is returned by AudioParam.value
- Issue #1852: Fix error condition of AudioNode.disconnect(destinationNode, output, input)
- Issue #1841: Recovering from unstable biquad filters?
- Issue #1777: Picture of the coordinate system for panner node
- Issue #1802: Clarify interaction between user-invoked suspend and autoplay policy
- Issue #1822: OfflineAudioContext.suspend can suspend before the given time
- Issue #1772: Sorting tracks alphabetically is underspecified
- Issue #1797: Specification is incomplete for AudioNode.connect()
- Issue #1805: Exception ordering on error
- Issue #1790: Automation example chart has an error (reversed function arguments
- Fix rendering algorithm iteration and cycle breaking
- Issue #1719: channel count changes in filter nodes with tail time
- Issue #1563: Make decodeAudioData more precise
- Issue #1481: Tighten spec on ABSN output channels?
- Issue #1762: Setting convolver buffer more than once?
- Issue #1758: Explicitly include time-domain processing code for BiquadFilterNode
- Issue #1770: Link to correct algorithm for StereoPannerNode, mention algorithm is equal-power
- Issue #1753: Have a single `AudioWorkletGlobalScope` per `BaseAudioContext`
- Issue #1746: AnalyserNode: Clarify how much time domain data we're supposed to keep around
- Issue #1741: Sample rate of AudioBuffer
- Issue #1745: Clarify unit of fftSize
- Issue #1743: Missing normative reference to Fetch
- Use "get a reference to the bytes" algorithm as needed.
- Specify rules for determining output chanel count.
- Clarified rendering algorithm for AudioListener.

§ 11.6. Since Working Draft of 19 June 2018

- Minor editorial clarifications.
- Update implementation-report.html.
- Widen the valid range of detune values so that any value that doesn't cause 2^(d/1200) to overflow is valid.
- PannerNode constructor throws errors.
- Rephrase algorithm for setting buffer and curve.
- Refine startRendering algorithm.
- Make "queue a task" link to the HTML spec.
- Specify more precisely, events overlapping with SetValueCurveAtTime.
- Add implementation report to gh-pages.
- Honor the given value in `outputChannelCount`.
- Initialize bufferDuration outside of process() in ABSN algorithm.
- Rework definition of ABSN output behavior to account for playbackRate's interaction with the start(…duration) argument.
- Add mention of video element in ultrasonic attack surface.

Error preparing HTML-CSS output (preProcess)

§ 11.7. Since Working Draft of 08 December 2015

- Add AudioWorklet and related interfaces to support custom nodes. This replaces ScriptProcessorNode, which is now deprecated.
- Explicitly say what the channel count, mode, and interpretation values are for all source nodes.
- Specify the behavior of Web Audio when a document is unloaded.
- Merge the proposed SpatialListener interface into AudioListener.
- Rework and clean up algorithms for panning and spatialization and define "magic functions".
- Clarify that AudioBufferSourceNode looping is limited by duration argument to start().
- Add constructors with options dictionaries for all node types.
- Clarify parameter automation method behavior and equations. Handle cases where automation methods may interact with each other.
- Support latency hints and arbitrary sample rates in AudioContext constructor.
- Clear up ambiguities in definitions of start() and stop() for scheduled sources.
- Remove automatic dezippering from AudioParam value setters which now equate to setValueAtTime().
- Specify normative behavior of DynamicsCompressorNode.
- Specify that AudioParam.value returns the most recent computed value.
- Permit AudioBufferSourceNode to specify sub-sample start, duration, loopStart and loopEnd. Respecify algorithms to say exactly how looping works in all scenarios, including dynamic and negative playback rates.
- Harmonized behavior of IIRFilterNode with BiquadFilterNode.
- Add diagram describing mono-input-to-matrixed-stereo case.
- Prevent connecting an AudioNode to an AudioParam of a different AudioContext.
- Added Audioparam cancelAndHoldAtTime
- Clarify behaviour of AudioParam.cancelScheduledValues().
- Add playing reference to MediaElementAudioSourceNodes and MediaStreamAudioSourceNodes.
- Refactor BaseAudioContext interface out of AudioContext, OfflineAudioContext.
- OfflineAudioContext inherits from BaseAudioContext, not AudioContext.
- "StereoPanner" replaced with the correct "StereoPannerNode".
- Support chaining on AudioNode.connect() and AudioParam automation methods.
- Specify behavior of events following SetTarget events.
- Reinstate channelCount declaration for AnalyserNode.
- Specify exponential ramp behavior when previous value is 0.
- Specify behavior of setValueCurveAtTime parameters.
- Add spatialListener attribute to AudioContext.
- Remove section titled "Doppler Shift".
- Added a list of nodes and reason why they can add latency, in an informative section.
- Speced nominal ranges, nyquist, and behavior when outside the range.
- Spec the processing model for the Web Audio API.
- Merge the SpatialPannerNode into the PannerNode, undeprecating the PannerNode.
- Merge the SpatialListener into the AudioListener, undeprecating the AudioListener.
- Added latencyHint(s).
- Move the constructor from BaseAudioContext to AudioContext where it belongs; BaseAudioContext is not constructible.
- Specified the Behavior of automations and nominal ranges.
- The playbackRate is widened to +/- infinity.

Error preparing HTML-CSS output (preProcess) is modified so that an implicit call to setValueAtTime is made at the end of the curve duration.

- Make setting the `value` attribute of an `AudioParam` strictly equivalent of calling setValueAtTime with AudioContext.currentTime.
- Add new sections for AudioContextOptions and AudioTimestamp.
- Add constructor for all nodes.
- Define ConstantSourceNode.
- Make the WaveShaperNode have a tail time, depending on the oversampling level.
- Allow collecting MediaStreamAudioSourceNode or MediaElementAudioSourceNode when they won't play ever again.
- Add a concept of 'allowed to start' and use it when creating an AudioContext and resuming it from resume() (closes #836).
- Add AudioScheduledSourceNode base class for source nodes.
- Mark all AudioParams as being k-rate.

§ 12. Acknowledgements

This specification is the collective work of the W3C Audio Working Group.

Members and former members of the Working Group and contributors to the specification are (at the time of writing, and by alphabetical order):
Adenot, Paul (Mozilla Foundation) - Specification Co-editor; Akhgari, Ehsan (Mozilla Foundation); Becker, Steven (Microsoft Corporation); Berkovitz, Joe (Invited Expert, affiliated with Noteflight/Hal Leonard) - WG co-chair from September 2013 to December 2017); Bossart, Pierre (Intel Corporation); Borins, Myles (Google, Inc); Buffa, Michel (NSAU); Caceres, Marcos (Invited Expert); Cardoso, Gabriel (INRIA); Carlson, Eric (Apple, Inc); Chen, Bin (Baidu, Inc); Choi, Hongchan (Google, Inc) - Specification Co-editor; Collichio, Lisa (Qualcomm); Geelnard, Marcus (Opera Software); Gehring, Todd (Dolby Laboratories); Goode, Adam (Google, Inc); Gregan, Matthew (Mozilla Foundation); Hikawa, Kazuo (AMEI); Hofmann, Bill (Dolby Laboratories); Jägenstedt, Philip (Google, Inc); Jeong, Paul Changjin (HTML5 Converged Technology Forum); Kalliokoski, Jussi (Invited Expert); Lee, WonSuk (Electronics and Telecommunications Research Institute); Kakishita, Masahiro (AMEI); Kawai, Ryoya (AMEI); Kostiainen, Anssi (Intel Corporation); Lilley, Chris (W3C Staff); Lowis, Chris (Invited Expert) - WG co-chair from December 2012 to September 2013, affiliated with British Broadcasting Corporation; MacDonald, Alistair (W3C Invited Experts) — WG co-chair from March 2011 to July 2012; Mandyam, Giridhar (Qualcomm Innovation Center, Inc); Michel, Thierry (W3C/ERCIM); Nair, Varun (Facebook); Needham, Chris (British Broadcasting Corporation); Noble, Jer (Apple, Inc); O'Callahan, Robert(Mozilla Foundation); Onumonu, Anthony (British Broadcasting Corporation); Paradis, Matthew (British Broadcasting Corporation) - WG co-chair from September 2013 to present; Pozdnyakov, Mikhail (Intel Corporation); Raman, T.V. (Google, Inc); Rogers, Chris (Google, Inc); Schepers, Doug (W3C/MIT); Schmitz, Alexander (JS Foundation); Shires, Glen (Google, Inc); Smith, Jerry (Microsoft Corporation); Smith, Michael (W3C/Keio); Thereaux, Olivier (British Broadcasting Corporation); Toy, Raymond (Google, Inc.) - WG co-chair from December 2017 - Present; Toyoshima, Takashi (Google, Inc); Troncy, Raphael (Institut Telecom); Verdie, Jean-Charles (MStar Semiconductor, Inc.); Wei, James (Intel Corporation); Weitnauer, Michael (IRT); Wilson, Chris (Google,Inc); Zergaoui, Mohamed (INNOVIMAX)

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words "for example" or are set apart from the normative text with `class="example"`, like this:

Error preparing HTML-CSS output (preProcess)

EXAMPLE 22 ¶

This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

Note, this is an informative note.

§ Conformant Algorithms

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant. Implementers are encouraged to optimize.

§ Index

§ Terms defined by this specification

Error preparing HTML-CSS output (preProcess)

Error preparing HTML-CSS output (preProcess)

Error preparing HTML-CSS output (preProcess)

Error preparing HTML-CSS output (preProcess)

Error preparing HTML-CSS output (preProcess)

Error preparing HTML-CSS output (preProcess)

Error preparing HTML-CSS output (preProcess)

§ Terms defined by reference

[] defines the following terms:

- getUserMedia()

[DOM] defines the following terms:

- Event
- EventInit
- EventTarget
- fire an event

[ECMASCRIPT] defines the following terms:

- data block

[HR-TIME-3] defines the following terms:

- DOMHighResTimeStamp

[HTML] defines the following terms:

- ErrorEvent
- EventHandler
- HTMLMediaElement
- MessageChannel
- MessagePort
- StructuredDeserialize
- StructuredSerialize
- StructuredSerializeWithTransfer
- Worklet
- WorkletGlobalScope
- addModule(moduleURL, options)
- associated document
- audio
- clean up after running a callback
- clean up after running script
- close()
- current settings object
- event handler
- fully active
- perform a microtask checkpoint
- port1
- port2
- prepare to run a callback
- prepare to run script
- queue a task
- relevant global object
- sticky activation
- terminate a worklet global scope
- video
- worklet destination type
- worklet global scope type

[INFRA] defines the following terms:

- code unit
- exist
- item
- ordered map
- struct

[MEDIACAPTURE-STREAMS] defines the following terms:

- MediaDevices
- MediaStream
- MediaStreamTrack
- enumerateDevices()

[WEBIDL] defines the following terms:

- ArrayBuffer
- DOMException
- DOMString
- DataCloneError
- EncodingError
- Exposed
- Float32Array
- FrozenArray
- Global
- IndexSizeError
- InvalidAccessError
- InvalidStateError
- NotAllowedError
- NotFoundError
- NotSupportedError
- Promise
- RangeError
- SameObject
- SecureContext
- TypeError
- Uint8Array
- UnknownError
- a promise rejected with
- boolean
- byte length
- conforming implementation
- construct
- detach
- detached
- double
- float
- get a copy of the buffer source
- long
- object
- record
- sequence
- this
- undefined
- unsigned long
- unsigned long long
- write

§ References

§ Normative References

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

Error preparing HTML-CSS output (preProcess)

[ECMASCRIPT]
ECMAScript Language Specification. URL: https://tc39.es/ecma262/multipage/

[FETCH]
Anne van Kesteren. *Fetch Standard*. Living Standard. URL: https://fetch.spec.whatwg.org/

[HR-TIME-3]
Yoav Weiss. *High Resolution Time*. URL: https://w3c.github.io/hr-time/

[HTML]
Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Anne van Kesteren; Domenic Denicola. *Infra Standard*. Living Standard. URL: https://infra.spec.whatwg.org/

[MEDIACAPTURE-STREAMS]
Cullen Jennings; et al. *Media Capture and Streams*. URL: https://w3c.github.io/mediacapture-main/

[MIMESNIFF]
Gordon P. Hemsley. *MIME Sniffing Standard*. Living Standard. URL: https://mimesniff.spec.whatwg.org/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[SECURITY-PRIVACY-QUESTIONNAIRE]
Theresa O'Connor; Peter Snyder. *Self-Review Questionnaire: Security and Privacy*. URL: https://w3ctag.github.io/security-questionnaire/

[WEBIDL]
Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

[WEBRTC]
Cullen Jennings; et al. *WebRTC: Real-Time Communication in Browsers*. URL: https://w3c.github.io/webrtc-pc/

§ Informative References

[2DCONTEXT]
Rik Cabanier; et al. *HTML Canvas 2D Context*. URL: https://www.w3.org/html/wg/drafts/2dcontext/html5_canvas_CR/

[MEDIASTREAM-RECORDING]
Miguel Casas-sanchez. *MediaStream Recording*. URL: https://w3c.github.io/mediacapture-record/

[WEBAUDIO-USECASES]
Joe Berkovitz; Olivier Thereaux. *Web Audio Processing: Use Cases and Requirements*. 29 January 2013. NOTE. URL: https://www.w3.org/TR/webaudio-usecases/

[WEBCODECS]
Paul Adenot; Bernard Aboba; Eugene Zemtsov. *WebCodecs*. URL: https://w3c.github.io/webcodecs/

[WEBGL]
Dean Jackson; Jeff Gilbert. *WebGL 2.0 Specification*. 12 August 2017. URL: https://www.khronos.org/registry/webgl/specs/latest/2.0/

[XHR]
Anne van Kesteren. *XMLHttpRequest Standard*. Living Standard. URL: https://xhr.spec.whatwg.org/

§ IDL Index

```
enum AudioContextState {
    "suspended",
    "running",
    "closed"
};

callback DecodeErrorCallback = undefined (DOMException error);

callback DecodeSuccessCallback = undefined (AudioBuffer decodedData);
```

Error preparing HTML-CSS output (preProcess)

```
interface BaseAudioContext : EventTarget {
```

```
    readonly attribute AudioDestinationNode destination;
    readonly attribute float sampleRate;
    readonly attribute double currentTime;
    readonly attribute AudioListener listener;
    readonly attribute AudioContextState state;
    [SameObject, SecureContext]
    readonly attribute AudioWorklet audioWorklet;
    attribute EventHandler onstatechange;

    AnalyserNode createAnalyser ();
    BiquadFilterNode createBiquadFilter ();
    AudioBuffer createBuffer (unsigned long numberOfChannels,
                              unsigned long length,
                              float sampleRate);
    AudioBufferSourceNode createBufferSource ();
    ChannelMergerNode createChannelMerger (optional unsigned long numberOfInputs = 6);
    ChannelSplitterNode createChannelSplitter (
        optional unsigned long numberOfOutputs = 6);
    ConstantSourceNode createConstantSource ();
    ConvolverNode createConvolver ();
    DelayNode createDelay (optional double maxDelayTime = 1.0);
    DynamicsCompressorNode createDynamicsCompressor ();
    GainNode createGain ();
    IIRFilterNode createIIRFilter (sequence<double> feedforward,
                                   sequence<double> feedback);
    OscillatorNode createOscillator ();
    PannerNode createPanner ();
    PeriodicWave createPeriodicWave (sequence<float> real,
                                     sequence<float> imag,
                                     optional PeriodicWaveConstraints constraints = {});
    ScriptProcessorNode createScriptProcessor(
        optional unsigned long bufferSize = 0,
        optional unsigned long numberOfInputChannels = 2,
        optional unsigned long numberOfOutputChannels = 2);
    StereoPannerNode createStereoPanner ();
    WaveShaperNode createWaveShaper ();

    Promise<AudioBuffer> decodeAudioData (
        ArrayBuffer audioData,
        optional DecodeSuccessCallback? successCallback,
        optional DecodeErrorCallback? errorCallback);
};

enum AudioContextLatencyCategory {
        "balanced",
        "interactive",
        "playback"
};

enum AudioSinkType {
    "none"
};

[Exposed=Window]
interface AudioContext : BaseAudioContext {
    constructor (optional AudioContextOptions contextOptions = {});
    readonly attribute double baseLatency;
    readonly attribute double outputLatency;
    [SecureContext] readonly attribute (DOMString or AudioSinkInfo) sinkId;
    [SecureContext] readonly attribute AudioRenderCapacity renderCapacity;
    attribute EventHandler onsinkchange;
    AudioTimestamp getOutputTimestamp ();
    Promise<undefined> resume ();
    Promise<undefined> suspend ();
    ned> close ();
    [SecureContext] Promise<undefined> setSinkId ((DOMString or AudioSinkOptions) sinkId);
```

Error preparing HTML-CSS output (preProcess)

```
    MediaElementAudioSourceNode createMediaElementSource (HTMLMediaElement mediaElement);
    MediaStreamAudioSourceNode createMediaStreamSource (MediaStream mediaStream);
    MediaStreamTrackAudioSourceNode createMediaStreamTrackSource (
        MediaStreamTrack mediaStreamTrack);
    MediaStreamAudioDestinationNode createMediaStreamDestination ();
};

dictionary AudioContextOptions {
    (AudioContextLatencyCategory or double) latencyHint = "interactive";
    float sampleRate;
    (DOMString or AudioSinkOptions) sinkId;
};

dictionary AudioSinkOptions {
    required AudioSinkType type;
};

[Exposed=Window]
interface AudioSinkInfo {
    readonly attribute AudioSinkType type;
};

dictionary AudioTimestamp {
    double contextTime;
    DOMHighResTimeStamp performanceTime;
};

[Exposed=Window]
interface AudioRenderCapacity : EventTarget {
    undefined start(optional AudioRenderCapacityOptions options = {});
        undefined stop();
        attribute EventHandler onupdate;
};

dictionary AudioRenderCapacityOptions {
        double updateInterval = 1;
};

[Exposed=Window]
interface AudioRenderCapacityEvent : Event {
    constructor (DOMString type, optional AudioRenderCapacityEventInit eventInitDict = {});
        readonly attribute double timestamp;
        readonly attribute double averageLoad;
        readonly attribute double peakLoad;
        readonly attribute double underrunRatio;
};

dictionary AudioRenderCapacityEventInit : EventInit {
    double timestamp = 0;
    double averageLoad = 0;
    double peakLoad = 0;
    double underrunRatio = 0;
};

[Exposed=Window]
interface OfflineAudioContext : BaseAudioContext {
    constructor(OfflineAudioContextOptions contextOptions);
    constructor(unsigned long numberOfChannels, unsigned long length, float sampleRate);
    Promise<AudioBuffer> startRendering();
    Promise<undefined> resume();
    Promise<undefined> suspend(double suspendTime);
    readonly attribute unsigned long length;
    attribute EventHandler oncomplete;
};
```

Error preparing HTML-CSS output (preProcess)

```
dictionary OfflineAudioContextOptions {
```

```
    unsigned long numberOfChannels = 1;
    required unsigned long length;
    required float sampleRate;
};

[Exposed=Window]
interface OfflineAudioCompletionEvent : Event {
    constructor (DOMString type, OfflineAudioCompletionEventInit eventInitDict);
    readonly attribute AudioBuffer renderedBuffer;
};

dictionary OfflineAudioCompletionEventInit : EventInit {
    required AudioBuffer renderedBuffer;
};

[Exposed=Window]
interface AudioBuffer {
    constructor (AudioBufferOptions options);
    readonly attribute float sampleRate;
    readonly attribute unsigned long length;
    readonly attribute double duration;
    readonly attribute unsigned long numberOfChannels;
    Float32Array getChannelData (unsigned long channel);
    undefined copyFromChannel (Float32Array destination,
                               unsigned long channelNumber,
                               optional unsigned long bufferOffset = 0);
    undefined copyToChannel (Float32Array source,
                             unsigned long channelNumber,
                             optional unsigned long bufferOffset = 0);
};

dictionary AudioBufferOptions {
    unsigned long numberOfChannels = 1;
    required unsigned long length;
    required float sampleRate;
};

[Exposed=Window]
interface AudioNode : EventTarget {
    AudioNode connect (AudioNode destinationNode,
                       optional unsigned long output = 0,
                       optional unsigned long input = 0);
    undefined connect (AudioParam destinationParam, optional unsigned long output = 0);
    undefined disconnect ();
    undefined disconnect (unsigned long output);
    undefined disconnect (AudioNode destinationNode);
    undefined disconnect (AudioNode destinationNode, unsigned long output);
    undefined disconnect (AudioNode destinationNode,
                          unsigned long output,
                          unsigned long input);
    undefined disconnect (AudioParam destinationParam);
    undefined disconnect (AudioParam destinationParam, unsigned long output);
    readonly attribute BaseAudioContext context;
    readonly attribute unsigned long numberOfInputs;
    readonly attribute unsigned long numberOfOutputs;
    attribute unsigned long channelCount;
    attribute ChannelCountMode channelCountMode;
    attribute ChannelInterpretation channelInterpretation;
};

enum ChannelCountMode {
    "max",
    "clamped-max",
    "explicit"
```

Error preparing HTML-CSS output (preProcess)

```
enum ChannelInterpretation {
    "speakers",
    "discrete"
};

dictionary AudioNodeOptions {
    unsigned long channelCount;
    ChannelCountMode channelCountMode;
    ChannelInterpretation channelInterpretation;
};

enum AutomationRate {
    "a-rate",
    "k-rate"
};

[Exposed=Window]
interface AudioParam {
    attribute float value;
    attribute AutomationRate automationRate;
    readonly attribute float defaultValue;
    readonly attribute float minValue;
    readonly attribute float maxValue;
    AudioParam setValueAtTime (float value, double startTime);
    AudioParam linearRampToValueAtTime (float value, double endTime);
    AudioParam exponentialRampToValueAtTime (float value, double endTime);
    AudioParam setTargetAtTime (float target, double startTime, float timeConstant);
    AudioParam setValueCurveAtTime (sequence<float> values,
                                    double startTime,
                                    double duration);
    AudioParam cancelScheduledValues (double cancelTime);
    AudioParam cancelAndHoldAtTime (double cancelTime);
};

[Exposed=Window]
interface AudioScheduledSourceNode : AudioNode {
    attribute EventHandler onended;
    undefined start(optional double when = 0);
    undefined stop(optional double when = 0);
};

[Exposed=Window]
interface AnalyserNode : AudioNode {
    constructor (BaseAudioContext context, optional AnalyserOptions options = {});
    undefined getFloatFrequencyData (Float32Array array);
    undefined getByteFrequencyData (Uint8Array array);
    undefined getFloatTimeDomainData (Float32Array array);
    undefined getByteTimeDomainData (Uint8Array array);
    attribute unsigned long fftSize;
    readonly attribute unsigned long frequencyBinCount;
    attribute double minDecibels;
    attribute double maxDecibels;
    attribute double smoothingTimeConstant;
};

dictionary AnalyserOptions : AudioNodeOptions {
    unsigned long fftSize = 2048;
    double maxDecibels = -30;
    double minDecibels = -100;
    double smoothingTimeConstant = 0.8;
};

[Exposed=Window]
interface AudioBufferSourceNode : AudioScheduledSourceNode {
                                  aseAudioContext context,
                                  optional AudioBufferSourceOptions options = {});
```

Error preparing HTML-CSS output (preProcess)

```
    attribute AudioBuffer? buffer;
    readonly attribute AudioParam playbackRate;
    readonly attribute AudioParam detune;
    attribute boolean loop;
    attribute double loopStart;
    attribute double loopEnd;
    undefined start (optional double when = 0,
                     optional double offset,
                     optional double duration);
};

dictionary AudioBufferSourceOptions {
    AudioBuffer? buffer;
    float detune = 0;
    boolean loop = false;
    double loopEnd = 0;
    double loopStart = 0;
    float playbackRate = 1;
};

[Exposed=Window]
interface AudioDestinationNode : AudioNode {
    readonly attribute unsigned long maxChannelCount;
};

[Exposed=Window]
interface AudioListener {
    readonly attribute AudioParam positionX;
    readonly attribute AudioParam positionY;
    readonly attribute AudioParam positionZ;
    readonly attribute AudioParam forwardX;
    readonly attribute AudioParam forwardY;
    readonly attribute AudioParam forwardZ;
    readonly attribute AudioParam upX;
    readonly attribute AudioParam upY;
    readonly attribute AudioParam upZ;
    undefined setPosition (float x, float y, float z);
    undefined setOrientation (float x, float y, float z, float xUp, float yUp, float zUp);
};

[Exposed=Window]
interface AudioProcessingEvent : Event {
    constructor (DOMString type, AudioProcessingEventInit eventInitDict);
    readonly attribute double playbackTime;
    readonly attribute AudioBuffer inputBuffer;
    readonly attribute AudioBuffer outputBuffer;
};

dictionary AudioProcessingEventInit : EventInit {
    required double playbackTime;
    required AudioBuffer inputBuffer;
    required AudioBuffer outputBuffer;
};

enum BiquadFilterType {
    "lowpass",
    "highpass",
    "bandpass",
    "lowshelf",
    "highshelf",
    "peaking",
    "notch",
    "allpass"
};
```

Error preparing HTML-CSS output (preProcess)

```
[Exposed=Window]
```

```
interface BiquadFilterNode : AudioNode {
    constructor (BaseAudioContext context, optional BiquadFilterOptions options = {});
    attribute BiquadFilterType type;
    readonly attribute AudioParam frequency;
    readonly attribute AudioParam detune;
    readonly attribute AudioParam Q;
    readonly attribute AudioParam gain;
    undefined getFrequencyResponse (Float32Array frequencyHz,
                                    Float32Array magResponse,
                                    Float32Array phaseResponse);
};

dictionary BiquadFilterOptions : AudioNodeOptions {
    BiquadFilterType type = "lowpass";
    float Q = 1;
    float detune = 0;
    float frequency = 350;
    float gain = 0;
};

[Exposed=Window]
interface ChannelMergerNode : AudioNode {
    constructor (BaseAudioContext context, optional ChannelMergerOptions options = {});
};

dictionary ChannelMergerOptions : AudioNodeOptions {
    unsigned long numberOfInputs = 6;
};

[Exposed=Window]
interface ChannelSplitterNode : AudioNode {
    constructor (BaseAudioContext context, optional ChannelSplitterOptions options = {});
};

dictionary ChannelSplitterOptions : AudioNodeOptions {
    unsigned long numberOfOutputs = 6;
};

[Exposed=Window]
interface ConstantSourceNode : AudioScheduledSourceNode {
    constructor (BaseAudioContext context, optional ConstantSourceOptions options = {});
    readonly attribute AudioParam offset;
};

dictionary ConstantSourceOptions {
    float offset = 1;
};

[Exposed=Window]
interface ConvolverNode : AudioNode {
    constructor (BaseAudioContext context, optional ConvolverOptions options = {});
    attribute AudioBuffer? buffer;
    attribute boolean normalize;
};

dictionary ConvolverOptions : AudioNodeOptions {
    AudioBuffer? buffer;
    boolean disableNormalization = false;
};

[Exposed=Window]
interface DelayNode : AudioNode {
    constructor (BaseAudioContext context, optional DelayOptions options = {});
    readonly attribute AudioParam delayTime;
```

Error preparing HTML-CSS output (preProcess)

```
dictionary DelayOptions : AudioNodeOptions {
    double maxDelayTime = 1;
    double delayTime = 0;
};

[Exposed=Window]
interface DynamicsCompressorNode : AudioNode {
    constructor (BaseAudioContext context,
                 optional DynamicsCompressorOptions options = {});
    readonly attribute AudioParam threshold;
    readonly attribute AudioParam knee;
    readonly attribute AudioParam ratio;
    readonly attribute float reduction;
    readonly attribute AudioParam attack;
    readonly attribute AudioParam release;
};

dictionary DynamicsCompressorOptions : AudioNodeOptions {
    float attack = 0.003;
    float knee = 30;
    float ratio = 12;
    float release = 0.25;
    float threshold = -24;
};

[Exposed=Window]
interface GainNode : AudioNode {
    constructor (BaseAudioContext context, optional GainOptions options = {});
    readonly attribute AudioParam gain;
};

dictionary GainOptions : AudioNodeOptions {
    float gain = 1.0;
};

[Exposed=Window]
interface IIRFilterNode : AudioNode {
    constructor (BaseAudioContext context, IIRFilterOptions options);
    undefined getFrequencyResponse (Float32Array frequencyHz,
                                    Float32Array magResponse,
                                    Float32Array phaseResponse);
};

dictionary IIRFilterOptions : AudioNodeOptions {
    required sequence<double> feedforward;
    required sequence<double> feedback;
};

[Exposed=Window]
interface MediaElementAudioSourceNode : AudioNode {
    constructor (AudioContext context, MediaElementAudioSourceOptions options);
    [SameObject] readonly attribute HTMLMediaElement mediaElement;
};

dictionary MediaElementAudioSourceOptions {
    required HTMLMediaElement mediaElement;
};

[Exposed=Window]
interface MediaStreamAudioDestinationNode : AudioNode {
    constructor (AudioContext context, optional AudioNodeOptions options = {});
    readonly attribute MediaStream stream;
};
```

Error preparing HTML-CSS output (preProcess)

```
interface MediaStreamAudioSourceNode : AudioNode {
```

```
    constructor (AudioContext context, MediaStreamAudioSourceOptions options);
    [SameObject] readonly attribute MediaStream mediaStream;
};

dictionary MediaStreamAudioSourceOptions {
    required MediaStream mediaStream;
};

[Exposed=Window]
interface MediaStreamTrackAudioSourceNode : AudioNode {
    constructor (AudioContext context, MediaStreamTrackAudioSourceOptions options);
};

dictionary MediaStreamTrackAudioSourceOptions {
    required MediaStreamTrack mediaStreamTrack;
};

enum OscillatorType {
    "sine",
    "square",
    "sawtooth",
    "triangle",
    "custom"
};

[Exposed=Window]
interface OscillatorNode : AudioScheduledSourceNode {
    constructor (BaseAudioContext context, optional OscillatorOptions options = {});
    attribute OscillatorType type;
    readonly attribute AudioParam frequency;
    readonly attribute AudioParam detune;
    undefined setPeriodicWave (PeriodicWave periodicWave);
};

dictionary OscillatorOptions : AudioNodeOptions {
    OscillatorType type = "sine";
    float frequency = 440;
    float detune = 0;
    PeriodicWave periodicWave;
};

enum PanningModelType {
        "equalpower",
        "HRTF"
};

enum DistanceModelType {
    "linear",
    "inverse",
    "exponential"
};

[Exposed=Window]
interface PannerNode : AudioNode {
    constructor (BaseAudioContext context, optional PannerOptions options = {});
    attribute PanningModelType panningModel;
    readonly attribute AudioParam positionX;
    readonly attribute AudioParam positionY;
    readonly attribute AudioParam positionZ;
    readonly attribute AudioParam orientationX;
    readonly attribute AudioParam orientationY;
    readonly attribute AudioParam orientationZ;
    attribute DistanceModelType distanceModel;
    attribute double refDistance;
    attribute double maxDistance;
    attribute double rolloffFactor;
```

Error preparing HTML-CSS output (preProcess)

```
    attribute double coneInnerAngle;
    attribute double coneOuterAngle;
    attribute double coneOuterGain;
    undefined setPosition (float x, float y, float z);
    undefined setOrientation (float x, float y, float z);
};

dictionary PannerOptions : AudioNodeOptions {
    PanningModelType panningModel = "equalpower";
    DistanceModelType distanceModel = "inverse";
    float positionX = 0;
    float positionY = 0;
    float positionZ = 0;
    float orientationX = 1;
    float orientationY = 0;
    float orientationZ = 0;
    double refDistance = 1;
    double maxDistance = 10000;
    double rolloffFactor = 1;
    double coneInnerAngle = 360;
    double coneOuterAngle = 360;
    double coneOuterGain = 0;
};

[Exposed=Window]
interface PeriodicWave {
    constructor (BaseAudioContext context, optional PeriodicWaveOptions options = {});
};

dictionary PeriodicWaveConstraints {
    boolean disableNormalization = false;
};

dictionary PeriodicWaveOptions : PeriodicWaveConstraints {
    sequence<float> real;
    sequence<float> imag;
};

[Exposed=Window]
interface ScriptProcessorNode : AudioNode {
    attribute EventHandler onaudioprocess;
    readonly attribute long bufferSize;
};

[Exposed=Window]
interface StereoPannerNode : AudioNode {
    constructor (BaseAudioContext context, optional StereoPannerOptions options = {});
    readonly attribute AudioParam pan;
};

dictionary StereoPannerOptions : AudioNodeOptions {
    float pan = 0;
};

enum OverSampleType {
    "none",
    "2x",
    "4x"
};

[Exposed=Window]
interface WaveShaperNode : AudioNode {
    constructor (BaseAudioContext context, optional WaveShaperOptions options = {});
    attribute Float32Array? curve;
    SampleType oversample;
};
```

Error preparing HTML-CSS output (preProcess)

```
dictionary WaveShaperOptions : AudioNodeOptions {
    sequence<float> curve;
    OverSampleType oversample = "none";
};

[Exposed=Window, SecureContext]
interface AudioWorklet : Worklet {
  readonly attribute MessagePort port;
};

callback AudioWorkletProcessorConstructor = AudioWorkletProcessor (object options);

[Global=(Worklet, AudioWorklet), Exposed=AudioWorklet]
interface AudioWorkletGlobalScope : WorkletGlobalScope {
    undefined registerProcessor (DOMString name,
                                 AudioWorkletProcessorConstructor processorCtor);
    readonly attribute unsigned long long currentFrame;
    readonly attribute double currentTime;
    readonly attribute float sampleRate;
    readonly attribute MessagePort port;
};

[Exposed=Window]
interface AudioParamMap {
    readonly maplike<DOMString, AudioParam>;
};

[Exposed=Window, SecureContext]
interface AudioWorkletNode : AudioNode {
    constructor (BaseAudioContext context, DOMString name,
                 optional AudioWorkletNodeOptions options = {});
    readonly attribute AudioParamMap parameters;
    readonly attribute MessagePort port;
    attribute EventHandler onprocessorerror;
};

dictionary AudioWorkletNodeOptions : AudioNodeOptions {
    unsigned long numberOfInputs = 1;
    unsigned long numberOfOutputs = 1;
    sequence<unsigned long> outputChannelCount;
    record<DOMString, double> parameterData;
    object processorOptions;
};

[Exposed=AudioWorklet]
interface AudioWorkletProcessor {
    constructor ();
    readonly attribute MessagePort port;
};

callback AudioWorkletProcessCallback =
  boolean (FrozenArray<FrozenArray<Float32Array>> inputs,
           FrozenArray<FrozenArray<Float32Array>> outputs,
           object parameters);

dictionary AudioParamDescriptor {
    required DOMString name;
    float defaultValue = 0;
    float minValue = -3.4028235e38;
    float maxValue = 3.4028235e38;
    AutomationRate automationRate = "a-rate";
};
```

Error preparing HTML-CSS output (preProcess)

Error preparing HTML-CSS output (preProcess)