AudioWorklet Design Pattern

By Hongchan Choi
Software Engineer working on Web Audio in Chromium

The previous article on AudioWorklets detailed the basic concepts and usage. Since its launch in Chrome 66 there have been many requests for more examples of how it can be used in actual applications. The AudioWorklet unlocks the full potential of WebAudio, but taking advantage of it can be challenging because it requires understanding concurrent programming wrapped with several JS APIs. Even for developers who are familiar with WebAudio, integrating the AudioWorklet with other APIs (e.g. WebAssembly) can be difficult.

This article will give the reader a better understanding of how to use the AudioWorklet in real-world settings and to offer tips to draw on its fullest power. Be sure to check out code examples and live demos as well!

Recap: AudioWorklet

Before diving in, let's quickly recap terms and facts around the AudioWorklet system which was previously introduced in this post.

- BaseAudioContext: Web Audio API's primary object.
- AudioWorklet: A special script file loader for the AudioWorklet operation. Belongs to BaseAudioContext. A BaseAudioContext can have one AudioWorklet. The loaded script file is evaluated in the AudioWorkletGlobalScope and is used to create the AudioWorkletProcessor instances.
- AudioWorkletGlobalScope : A special JS global scope for the AudioWorklet operation. Runs on a dedicated rendering thread for the WebAudio. A BaseAudioContext can have one AudioWorkletGlobalScope.
- AudioWorkletNode : An AudioNode designed for the AudioWorklet operation. Instantiated from a BaseAudioContext. A BaseAudioContext can have multiple AudioWorkletNodes similarly to the native AudioNodes.
- AudioWorkletProcessor : A counterpart of the AudioWorkletNode. The actual guts of the AudioWorkletNode processing the audio stream by the user-supplied code. It is instantiated in the AudioWorkletGlobalScope when a AudioWorkletNode is constructed. An AudioWorkletNode can have one matching AudioWorkletProcessor.

Design Patterns

Using AudioWorklet with WebAssembly

- Example code

WebAssembly is a perfect companion for AudioWorkletProcessor. The combination of these two features brings a variety of advantages to audio processing on the web, but the two biggest benefits are: a) bringing existing C/C++ audio processing code into the WebAudio ecosystem and b) avoiding the overhead of JS JIT compilation and garbage collection in the audio processing code.

The former is important to developers with an existing investment in audio processing code and libraries, but the latter is critical for nearly all users of the API. In the world of WebAudio, the timing budget for the stable audio stream is quite demanding: it is only 3ms at the sample rate of 44.1Khz. Even a slight hiccup in the audio processing code can cause glitches. The developer must optimize the code for faster processing, but also minimize the amount of JS garbage being generated. Using WebAssembly can be a solution that addresses both problems at the same time: it is faster and generates no garbage from the code.

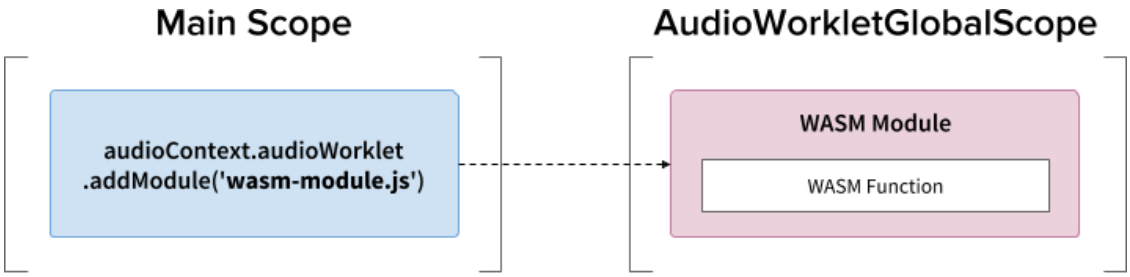
The next section describes how WebAssembly can be used with an AudioWorklet and the accompanied code example can be found here. For the basic tutorial on how to use Emscripten and WebAssembly (especially the Emscripten glue code), please take a look at this article.

Setting Up

It all sounds great, but we need a bit of structure to set things up properly. The first design question to ask is how and where to instantiate a WebAssembly module. After fetching the Emscripten's glue code, there are two paths for the module instantiation:

1. Instantiate a WebAssembly module by loading the glue code into the AudioWorkletGlobalScope via `audioContext.audioWorklet.addModule()`.
2. Instantiate a WebAssembly module in the main scope, then transfer the module via the AudioWorkletNode's constructor options.

The decision largely depends on your design and preference, but the idea is that the WebAssembly module can generate a WebAssembly instance in the AudioWorkletGlobalScope, which becomes an audio processing kernel within an AudioWorkletProcessor instance.

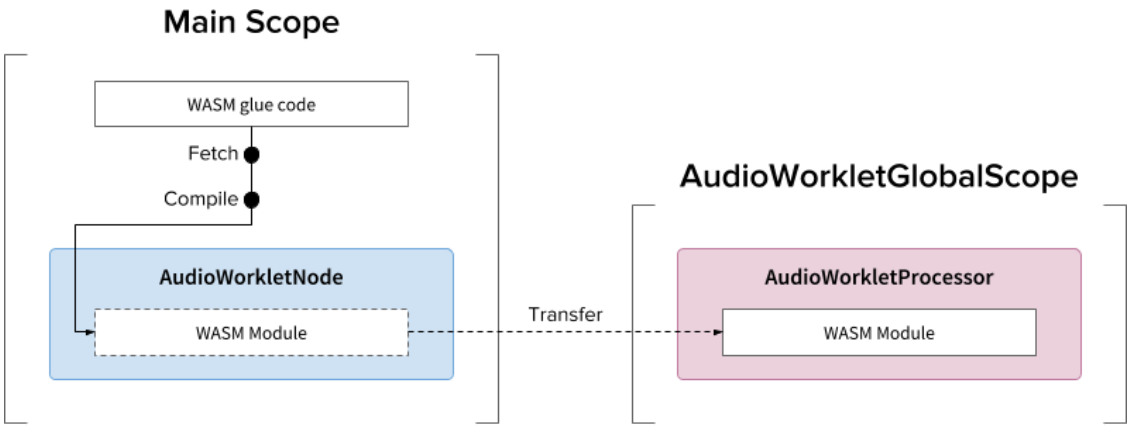


WebAssembly module instantiation pattern A: Using `.addModule()` call

For pattern A to work correctly, Emscripten needs a couple of options to generate the correct WebAssembly glue code for our configuration:

```
-s BINARYEN_ASYNC_COMPILATION=0 -s SINGLE_FILE=1 --post-js mycode.js
```

These options ensure the synchronous compilation of a WebAssembly module in the AudioWorkletGlobalScope. It also appends the AudioWorkletProcessor's class definition in **`mycode.js`** so it can be loaded after the module is initialized. The primary reason to use the synchronous compilation is that the promise resolution of **`audioWorklet.addModule()`** does not wait for the resolution of promises in the AudioWorkletGlobalScope. The synchronous loading or compilation in the main thread is not generally recommended because it blocks the other tasks in the same thread, but here we can bypass the rule because the compilation happens on the AudioWorkletGlobalScope, which runs off of the main thread. (See this for the more info.)



WASM module instantiation pattern B: Using AudioWorkletNode constructor's cross-thread transfer

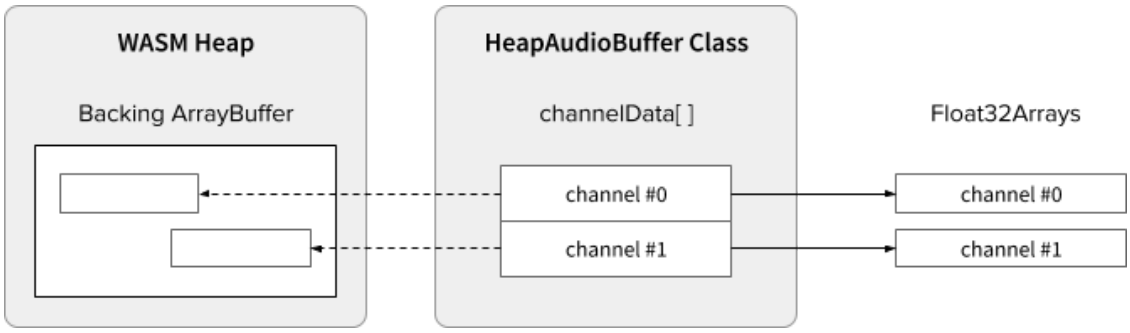
The pattern B can be useful if asynchronous heavy-lifting is required. It utilizes the main thread for fetching the glue code from the server and compiling the module. Then it will transfer the WASM module via the constructor of AudioWorkletNode. This pattern makes even more sense when you have to load the module dynamically after the

AudioWorkletGlobalScope starts rendering the audio stream. Depending on the size of the module, compiling it in the middle of the rendering can cause glitches in the stream.

Currently, the pattern B is only supported behind an experimental flag because it requires WebAssembly structured cloning. (`chrome://flags/#enable-webassembly`) If a WASM module should be a part of your AudioWorkletNode design, passing it through AudioWorkletNode constructor can definitely be useful.

WASM Heap and Audio Data

WebAssembly code only works on the memory allocated within a dedicated WASM heap. In order to take advantage of it, the audio data needs to be cloned back and forth between the WASM heap and the audio data arrays. The HeapAudioBuffer class in the example code handles this operation nicely.



HeapAudioBuffer class for the easier usage of WASM heap

There is an early proposal under discussion to integrate the WASM heap directly into the AudioWorklet system. Getting rid of this redundant data cloning between the JS memory and the WASM heap seems natural, but the specific details need to be worked out.

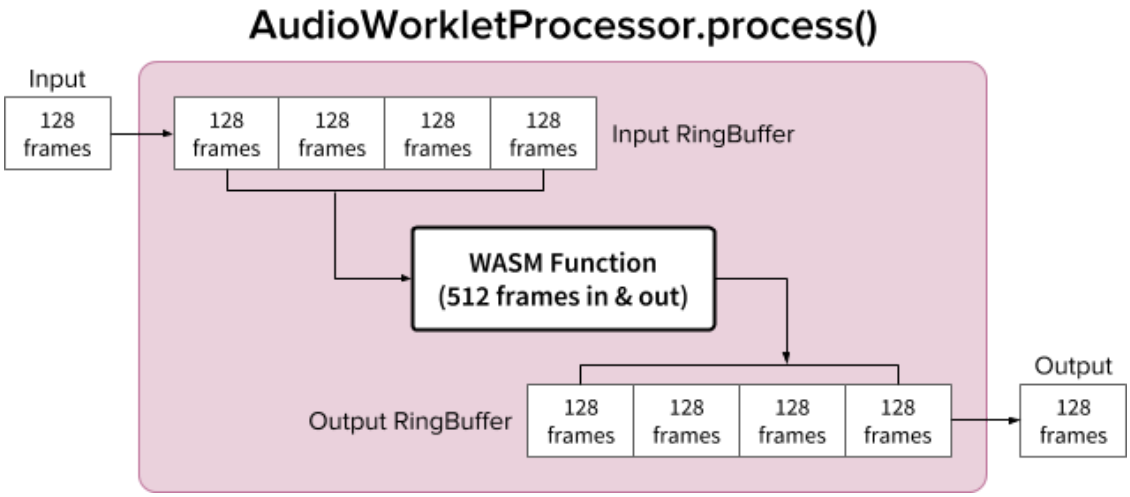
Handling Buffer Size Mismatch

- Example code

An AudioWorkletNode and AudioWorkletProcessor pair is designed to work like a regular AudioNode; AudioWorkletNode handles the interaction with other codes while AudioWorkletProcessor takes care of internal audio processing. Because a regular AudioNode processes 128 frames at a time, AudioWorkletProcessor must do the same to become a first-class citizen. This is one of the advantages of the AudioWorklet design that ensures no additional latency due to internal buffering is introduced within the AudioWorkletProcessor, but it can be a problem if a processing function requires a buffer

size different than 128 frames. The common solution for such case is to use a ring buffer, also known as a circular buffer or a FIFO.

Here's a diagram of AudioWorkletProcessor using two ring buffers inside to accommodate a WASM function that takes 512 frames in and out. (The number 512 here is arbitrarily picked.)



Using RingBuffer inside of AudioWorkletProcessor's `process()` method

The algorithm for diagram would be:

1. AudioWorkletProcessor pushes 128 frames into the **Input RingBuffer** from its Input.
2. Perform the following steps only if the **Input RingBuffer** has greater than or equal to 512 frames.
    a. Pull 512 frames from the **Input RingBuffer**.
    b. Process 512 frames with the given WASM function.
    c. Push 512 frames to the **Output RingBuffer**.
3. AudioWorkletProcessor pulls 128 frames from the **Output RingBuffer** to fill its **Output**.

As shown in the diagram, Input frames always get accumulated into Input RingBuffer and it handles buffer overflow by overwriting the oldest frame block in the buffer. That is a reasonable thing to do for a real-time audio application. Similarly, the Output frame block will always get pulled by the system. Buffer underflow (not enough data) in Output RingBuffer will result silence causing a glitch in the stream.

This pattern is useful when replacing ScriptProcessorNode (SPN) with AudioWorkletNode. Since SPN allows the developer to pick a buffer size between 256 and 4096 frames, so the drop-in substitution of SPN with AudioWorkletNode can be difficult and using a ring buffer

provides a nice workaround. A audio recorder would be a great example that can be built on top of this design.

However, it is important to understand that this design only reconciles the buffer size mismatch and it does not give more time to run the given script code. If the code cannot finish the task within the timing budget of render quantum (~3ms at 44.1Khz), it will affect the onset timing of subsequent callback function and eventually cause glitches.

Mixing this design with WebAssembly can be complicated because of memory management around WASM heap. At the time of writing, the data going in and out of WASM heap must be cloned but we can utilize HeapAudioBuffer class to make memory management slightly easier. The idea of using user-allocated memory to reduce redundant data cloning will be discussed in the future.

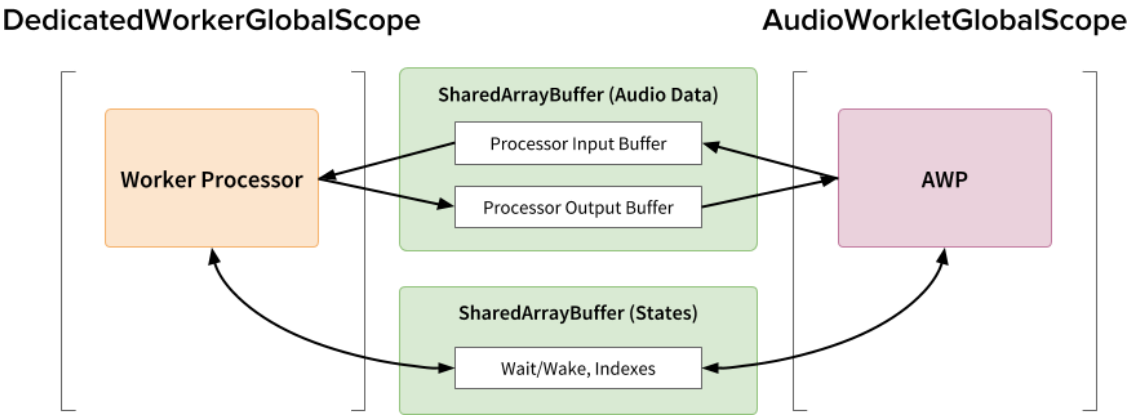
The RingBuffer class can be found here.

WebAudio Powerhouse: AudioWorklet and SharedArrayBuffer

Note: SharedArrayBuffer is disabled by default at the time of writing. Go to `chrome://flags` and enable *SharedArrayBuffer* to play with this feature.

- Example code

The last design pattern in this article is to put several cutting edge APIs into one place; AudioWorklet, SharedArrayBuffer, Atomics and Worker. With this non-trivial setup, it unlocks a path for existing audio software written in C/C++ to run in a web browser while maintaining a smooth user experience.



An overview of the last design pattern: AudioWorklet, SharedArrayBuffer and Worker

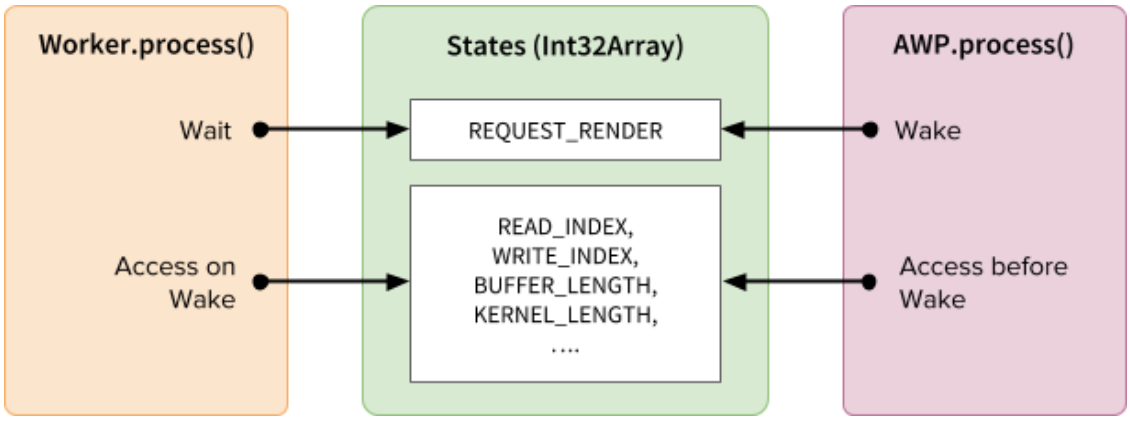
The biggest advantage of this design is being able to use a DedicatedWorkerGlobalScope solely for audio processing. In Chrome, WorkerGlobalScope runs on a lower priority thread than the WebAudio rendering thread but it has several advantages over AudioWorkletGlobalScope . DedicatedWorkerGlobalScope is less constrained in terms of the API surface available in the scope. Also you can expect better support from Emscripten because the Worker API has existed for some years.

SharedArrayBuffer plays a critical role for this design to work efficiently. Although both Worker and AudioWorkletProcessor are equipped with asynchronous messaging (MessagePort), it is suboptimal for real-time audio processing because of repetitive memory allocation and messaging latency. So we allocate a memory block up front that can be accessed from both threads for fast bidirectional data transfer.

From Web Audio API purist's viewpoint, this design might look suboptimal because it uses the AudioWorklet as a simple "audio sink" and does everything in the Worker. But considering the cost of rewriting C/C++ projects in JavaScript can be prohibitive or even be impossible, this design can be the most efficient implementation path for such projects.

Shared States and Atomics

When using a shared memory for audio data, the access from both sides must be coordinated carefully. Sharing atomically accessible states is a solution for such problem. We can take advantage of `Int32Array` backed by a SAB for this purpose.



Synchronization mechanism: SharedArrayBuffer and Atomics

Synchronization mechanism: SharedArrayBuffer and Atomics

Each field of the States array represents vital information about the shared buffers. The most important one is a field for the synchronization (`REQUEST_RENDER`). The idea is that Worker waits for this field to be touched by AudioWorkletProcessor and process the audio

when it wakes up. Along with SharedArrayBuffer (SAB), Atomics API makes this mechanism possible.

Note that the synchronization of two threads are rather loose. The onset of `Worker.process()` will be triggered by `AudioWorkletProcessor.process()` method, but the AudioWorkletProcessor does not wait until the `Worker.process()` is finished. This is by design; the AudioWorkletProcessor is driven by the audio callback so it must not be synchronously blocked. In the worst case scenario the audio stream might suffer from duplicate or drop out but it will eventually recover when the rendering performance is stabilized.

Setting Up and Running

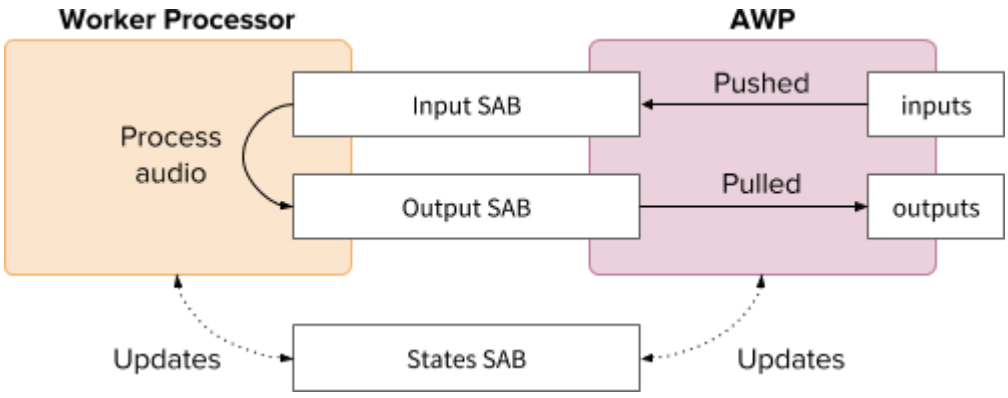
As shown in the diagram above, this design has several components to arrange: DedicatedWorkerGlobalScope (DWGS), AudioWorkletGlobalScope (AWGS), SharedArrayBuffer and the main thread. The following steps describe what should happen in the initialization phase.

Initialization

1. [Main] AudioWorkletNode constructor gets called.
    a. Create Worker.
    b. The associated AudioWorkletProcessor will be created.
2. [DWGS] Worker creates 2 SharedArrayBuffers. (one for shared states and the other for audio data)
3. [DWGS] Worker sends SharedArrayBuffer references to AudioWorkletNode.
4. [Main] AudioWorkletNode sends SharedArrayBuffer references to AudioWorkletProcessor.
5. [AWGS] AudioWorkletProcessor notifies AudioWorkletNode that the setup is completed.

Once the initialization is completed, `AudioWorkletProcessor.process()` starts to get called. The following is what should happen in each iteration of the rendering loop.

Rendering Loop



Multi-threaded rendering with SharedArrayBuffers

1. [AWGS] `AudioWorkletProcessor.process(inputs, outputs)` gets called for every render quantum.
    a. `inputs` will be pushed into **Input SAB**.
    b. `outputs` will be filled by consuming audio data in **Output SAB**.
    c. Updates **States SAB** with new buffer indexes accordingly.
    d. If **Output SAB** gets close to underflow threshold, Wake Worker to render more audio data.
2. [DWGS] Worker waits (sleeps) for the wake signal from `AudioWorkletProcessor.process()`. When it wakes up:
    a. Fetches buffer indexes from **States SAB**.
    b. Run the process function with data from **Input SAB** to fill **Output SAB**.
    c. Updates **States SAB** with buffer indexes accordingly.
    d. Goes to sleep and wait for the next signal.

The example code can be found here, but note that the SharedArrayBuffer experimental flag must be enabled for this demo to work. The code was written with pure JS code for simplicity, but it can be replaced with WebAssembly code if needed. Such case should be handled with extra care by wrapping memory management with HeapAudioBuffer class.

Conclusion

The ultimate goal of the AudioWorklet is to make the Web Audio API truly "extensible". A multi-year effort went into its design to make it possible to implement the rest of Web Audio API with the AudioWorklet. In turn, now we have higher complexity in its design and this can be an unexpected challenge.

Fortunately, the reason for such complexity is purely to empower developers. Being able to run WebAssembly on AudioWorkletGlobalScope unlocks huge potential for high-performance audio processing on the web. For large-scale audio applications written in C or C++, using an AudioWorklet with SharedArrayBuffers and Workers can be an attractive option to explore.

Credits

Special thanks to Chris Wilson, Jason Miller, Joshua Bell and Raymond Toy for reviewing a draft of this article and giving insightful feedback.

Last updated July 12, 2018.