Metaprogramming Strategies for AudioWorklets

Charles Roberts
Interactive Media and Game Development Program
Department of Computer Science
Worcester Polytechnic Institute
charlie@charlie-roberts.com

ABSTRACT

The introduction of AudioWorklets to the Web Audio API greatly expands the potential of browser-based audio programming. However, managing state between the various threads AudioWorklets occupy entails a fair amount of complexity, particularly when designing dynamic music programming environments where exact digital signal processing requirements cannot be known ahead of time. Such environments are commonly used for live coding performance, interactive composition, and coding playgrounds for musical experimentation.

Our research explores metaprogramming strategies to create AudioWorklet implementations for two JavaScript libraries, Genish.js and Gibberish.js. These strategies help hide the complexities of inter-thread communication from end-users and enable a variety of signal processing and interaction techniques that would otherwise be difficult to achieve.

1. INTRODUCTION

While the Web Audio API (WAAPI) has seen remarkable adoption and usage over the last six years, it has done so despite only including a minimal set of DSP algorithms, constraining the types of audio synthesis that can be used. The use of a block-based audio graph is another constraint of the WAAPI, precluding per-sample processing techniques such as single-sample feedback loops and audio-rate modulation of scheduling [8].

The ScriptProcessor node the WAAPI offers removes these limitations, but comes with costs that are unacceptable for many applications, including increased latency and decreased efficiency. It also increases the potential for audio interrupts due to running in the main thread alongside many imnportant browser processes such as networking, graphics, and human computer interaction. Despite these limitations the ScriptProcessor has been used in a variety of contexts, from realtime musical performances to computer science education [9]. And running in the main thread of a web application also provides a very simple model for interacting with the audio engine: if your audio engine is built using a system like Flocking [2] or Gibberish.js [8], where all computation is performed in a ScriptProcessor node, you can alter properties synchronously in the main thread. In the case of Gibberish.js, this includes the ability to easily change / manipulate scheduling, while still enabling it to be modulated at audio rate.

While these capabilities of the ScriptProcessor node are exciting, its previously mentioned drawbacks are severe enough to preclude usage by many developers. The introduction of the AudioWorklet, which shipped in Chrome 66 in April of 2018, removes many of these drawbacks by enabling programmers to define custom DSP routines in JavaScript or WebAssembly that run in a separate thread [1]. However, not running in the main thread means that much of the simplicity provided by the ScriptProcessor node is lost; developers are immediately forced to think about inter-thread communication and strategies for sharing state.

The goal of the research presented in this paper is to enable two libraries, Genish.js and Gibberish.js, to use AudioWorklets while hiding the complexity of communicating between the main thread and the audio processing thread from end-users. While this process was fairly straightforward for Genish.js, where code generation is typically isolated to individual synthesis or processing units, it proved more difficult in Gibberish.js, which features a general-purpose music programming API. The challenge is complicated by our interest in dynamic programming environments, where the computational needs of an AudioWorklet cannot be fully known ahead of time. Such environments include systems for live coding performance, interactive composition, and coding playgrounds for musical experimentation.

We begin this paper with a short discussion of important AudioWorklet terminology. We then discuss adding support for AudioWorklets to both libraries, and conclude with a discussion of the limitations we encountered in our research and a brief overview of related work. The synthesis and music programming capabilities of Genish.js and Gibberish.js have been extensively discussed previously [8]; please see our prior work for a more general overview of the libraries.

2. AUDIOWORKLET TERMINOLOGY

This section briefly addresses some terminology that is important for readers who are unfamiliar with AudioWorklets. For a more thorough description of the AudioWorklet architecture, please see [1].

- AudioWorkletNode – A Web Audio API node that can be connected / disconnected / addressed in the same

Web Audio Conference WAC-2018, September 19–21, 2018, Berlin, Germany.

fashion as any other WAAPI nodes, but includes custom DSP written in JavaScript or WebAssembly. This node is primarily scripted and controlled from the main thread, however, it can communicate with its associated audio processing thread using a dedicated *MessagePort* object. Each instance of an AudioWorkletNode is associated with an *AudioWorkletProcessor*.

- AudioWorkletProcessor – Subclasses of the AudioWorkletProcessor are responsible for generating samples and defining inputs to their associated AudioWorkletNode that other WAAPI nodes can connect to. Generating output samples occurs in a different thread than the main browser thread, necessitating inter-thread communication through a MessagePort for many types of messaging.

- MessagePort – The MessagePort enables asynchronous, bi-directional communication between the AudioWorkletNode (operating in the main thread) and the AudioWorkletProcessor, which typically processes samples in a separate thread. Most serializable objects and arrays can be passed between threads using the MessagePort.

3. GENISH.JS

Genish.js is a relatively low-level DSP library inspired by the Gen programming language for Max/MSP [14]. It provides fundamental building blocks for constructing oscillators, filters, and digital audio effects, but does not include these higher-level synthesis objects itself. Genish.js places a heavy emphasis on per-sample processing techniques; for more information please see [8]. Implementing AudioWorklet support for Genish.js was fairly straightforward. In addition to its existing ability to generate DSP functions that can be used inside of a ScriptProcessor node, utility methods were added enabling Genish.js to generate a complete definition of a AudioWorkletProcessor subclass.

Interacting with Genish.js, regardless of whether its output is running in a ScriptProcessor or an AudioWorklet, is primarily accomplished through the use of the `param` unit generator. Each instance of `param` points to a single `Float64` number in a single memory heap used by all unit generators found in Genish.js, and provides the ability to alter it from the main thread. When a function generated by Genish.js runs in a ScriptProcessor node, users access this memory heap directly from the main thread using basic metaprogramming, as illustrated in Listing 1[1]:

```
frequency = param( "frequency", 220 )
callback = play( cycle( frequency ) )

// change the frequency of the running
// sine oscillator from 220 to 440 Hz
frequency.value = 440
```

Listing 1: Changing the value of a property in the AudioWorkletProcessor thread from the main thread

[1]Genish.js and Gibberish.js both include coding playgrounds where synths and ugens are exported to the global namespace. All code examples in this paper will assume the use of the global namespace for the sake of terseness.

The setter method placed on the `value` property looks up the memory index associated with the frequency `param` and then assigns the new value to the associated location in the memory heap. For the AudioWorklet implementation, the process is similar, but instead of directly assigning to the memory heap we instead communicate with the AudioWorkletProcessor through its MessagePort, which then assigns the received value to the heap inside of the AudioWorkletProcessor thread.

The synthesis algorithm running in the AudioWorkletProcessor can also be controlled via the `input` ugen, which creates a Web Audio API `AudioParam` on the resulting AudioWorklet that can be connected to by any other standard WAAPI node (including other AudioWorklets), as shown in Listing 2.

```
// input parameters: name, input #, channel #,
// default value, min, and max
freqMod = input('freqMod',0,0,2,0,50)
sine = cycle( add( 440, freqMod ) )

play( sine ).then( node => {
  const ctx = Gibberish.utilities.ctx

  const osc = ctx.createOscillator()
  osc.frequency.value = 2
  const gain = ctx.createGain()
  gain.gain.value = 50

  osc.connect( gain )
  gain.connect( node.freqMod )
  osc.start()
})
```

Listing 2: Creating a Gibberish.js wavetable oscillator, running in an AudioWorklet, with its frequency modulated by a WAAPI `OscillatorNode`.

3.1 Initialization and Usage

One final difference between the AudioWorklet implementation of Genish.js and its ScriptProcessor implementation is that the utility methods for creating and running graphs are slightly different. In the case of the ScriptProcessor node, the `playGraph` function returns a callback that is immediately played in a `ScriptProcessor` node. In the case of the AudioWorklet, the `playWorklet` function instead returns a promise that resolves when the AudioWorkletProcessor module has been asynchronously loaded; this behavior is illustrated in Listing 2.

The boilerplate for generating the AudioWorkletProcessor subclass is about forty lines of code; the callback function created by Genish.js is injected into this boilerplate which is then converted to a `Blob` that is registered as a module. This is a different approach to what is used by Gibberish.js, as discussed in Section 4.1.

4. GIBBERISH.JS

Like Genish.js, Gibberish.js places a heavy emphasis on the use of per-sample processing techniques, including single-sample feedback loops and audio-rate modulation of scheduling. Gibberish.js creates high-level synthesis objects from the lower-level building blocks found in Genish.js, and then provides affordances for connecting and sequencing them. For more information on Gibberish.js and its relationship to Genish.js, please see [8].

In computer music, there is a history of decoupling audio synthesis engines with systems for scheduling and control. A classic example is the shift from SuperCollider 2, which tightly coupled control and synthesis, to SuperCollider 3, which decoupled them entirely by separating them into different applications, SCLang and SCServer [7]. A more recent multimedia programming environment, LuaAV, used coroutines running in the Lua interpreter thread for scheduling while transferring state to the audio thread via message queues [15]. Similar capabilities are provided to the AudioWorklet via its MessagePort abstraction.

In some ways the transition of Gibberish.js from supporting the ScriptProcessor node to also including AudioWorklet support mirrors the transition from SuperCollider 2 to SuperCollider 3. In the ScriptProcessor version of Gibberish.js, all synthesis routines directly access control and scheduling structures defined in the main thread. In the AudioWorklet version, this is no longer possible as signal processing occurs in a different thread. Our solution is somewhat similar to that taken by LuaAV, where we mirror state between the main thread and the audio processing thread by passing data through each AudioWorklet's `MessagePort`.

This approach leads to many interesting complexities; we have simplified as many of these as possible for users of Gibberish.js through metaprogramming techniques as described in the rest of this section.

4.1 Mirroring of State via Metaprogramming

Our decision to hide the inner workings of state management between the main thread and the AudioWorkletProcessor thread led to the extensive use of metaprogramming to mirror the current state of objects across thread boundaries. This mirroring begins with the loading and instantiation of the Gibberish.js library itself, which occurs both in the main JavaScript runtime thread as well as in each AudioWorkletProcessor thread using Gibberish.js. The build script for Gibberish.js first compiles the library, and then compiles an AudioWorklet module that contains the inlined library and a minimal AudioWorkletProcessor subclass for communicating with the main thread and running audio callbacks generated by Gibberish.js. All DSP code is then generated dynamically in the AudioWorkletProcessor thread as commands from the main thread are received. One limitation of this method is that it is not currently possible to define `AudioParam` inputs to connect external Web Audio API nodes into Gibberish.js. However, Gibberish.js is intended as a standalone library that emphasizes per-sample processing techniques that are largely precluded by processing block-rate inputs; for creating nodes designed work in tandem with other WAAPI nodes Genish.js is a better choice.

Initializing Gibberish.js from the main JavaScript runtime thread loads an associated AudioWorklet module, which in turn initializes a second instance of Gibberish.js inside of the AudioWorkletProcessor thread. These separate instances are then roughly synchronized through MessagePort communication at block rate. When programmers instantiate Gibberish.js objects in the main thread Gibberish.js checks to see if AudioWorklets are being used; if so constructors return a JavaScript *Proxy* object that is responsible for communicating any changes in its underlying state to the AudioWorkletProcessor. This communication primarily consists of three responsibilities:

- Communicate that a new object has been instantiated. Include all parameters used in the instantiation and a unique identification number (UID) to identify the object across thread boundaries. If other Gibberish objects are used to instantiate an object (for example, an oscillator that is used to modulate a parameter) include the UID of each object so that its counterpart in the AudioWorkletProcessor thread can be found.
- Communicate any changes to any properties of the underlying object, by passing the UID of the object that was changed, the name of the property, and the new property value.
- Communicate any method calls that are made by the object. This includes the UID of the object, the name of the method that was called, and any arguments passed to the method.

The use of ES6 `Proxy` objects ensure that this typically happens automatically, without any intervention required by programmers; this communication is shown in Figure 1. However, as described in Section 4.2 this isn't always possible to automate.

In Gibberish.js state transfer needs to be bi-directional between the JavaScript runtime's main thread and the AudioWorkletProcessor thread. One of the primary benefits to using Gibberish.js is its ability to modulate scheduling at audio-rate; this means that all sequencing in Gibberish.js, when using AudioWorklets, *is performed inside of the AudioWorkletProcessor thread, not the main thread.* Because of this, changes to state that are triggered by sequencers in the AudioWorkletProcessor are communicated back to the main thread; this is particularly useful for visualizing / annotating the state of the audio system and musical patterns, for example, as described in [10]. Any changes to state generated by sequencers running in the AudioWorkletProcessor thread are queued until its `.process()` method has completed; the queued changes are then sent asynchronously to the main thread in a single serialized array.

4.2 Functions, Closures, Successes, Failures

Sequencing in Gibberish.js (and Gibber [11], which is built on top of Gibberish.js) is highly dependent on the use of functions. Programmers can pass anonymous functions to sequencers that will determine both their outputted values and their scheduling. However, passing such functions across the boundaries of a thread becomes difficult in JavaScript, where the execution of functions often depends on having access to upvalues captured in a closure, or, even more problematically, in a nested hierarchy of closures. For example, consider the example shown in Listing 3, where two synthesized hihat instruments are created, with the 'open' hihat possessing a longer decay time than the 'closed' one. A `playHat` function is then written, which randomly triggers one of the hihats each time it is called. This function is then passed to a `Sequencer` constructor, which will call the function every `11025` samples.

```
closedHat = Hat({ decay:.05 }).connect()
openHat   = Hat({ decay:.2 }).connect()

playHat = ()=> {
  if( Math.random() > .25 )
    closedHat.trigger( .035 )
  else
    openHat.trigger( .05 )
```

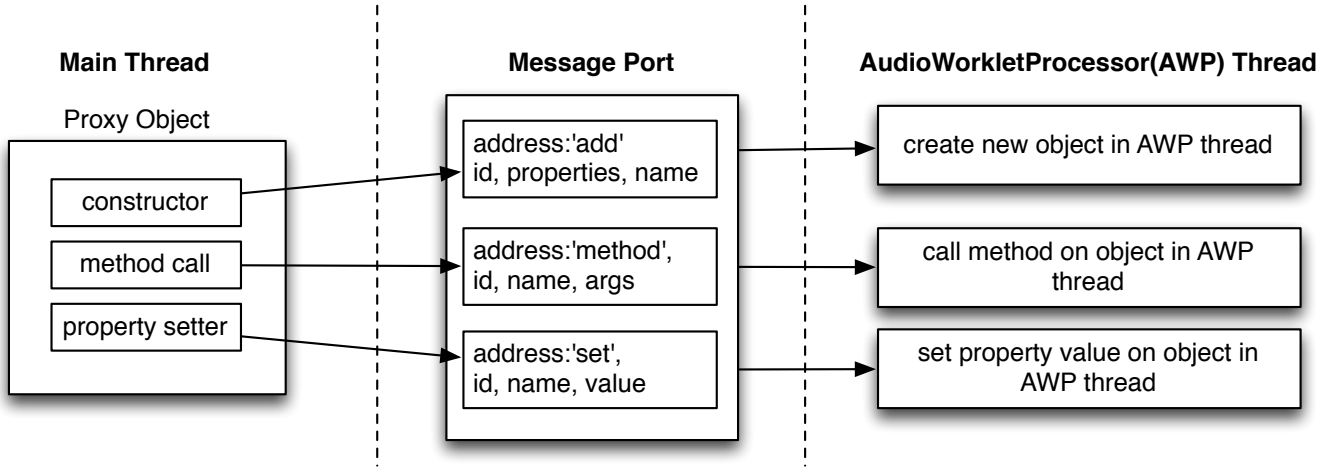

Figure 1: Proxy object communication from the main thread traveling to the AudioWorkletProcessor thread.

```
}

hatSeq = Sequencer.make(
  [ playHat ],
  [ 11025 ]
).start()
```

Listing 3: A an example of a musical function that loses access to its scope after serialization.

In the ScriptProcessor implementation of Gibberish.js this works as expected. However, in the AudioWorklet implementation, the `playHat` function has to be serialized so that it can be transferred to the AudioWorkletProcessor thread. But a serialized function (a function converted to a string) will no longer contain references to the `closedHat` and `openHat` objects after it is transferred, as these objects were created in a different thread. The closure providing the function access to its surrounding scope is lost during serialization and the function will fail to execute.

As one workaround, the `wrap` function enables users to pass objects that are bound as parameters to a function when it is de-serialized inside the AudioWorkletProcessor thread. The code in Listing 3 is converted to a version using `wrap` that is valid with the AudioWorklet implementation in Listing 4. Although the amount of code required to wrap a function in this manner is minimal, it does require that programmers consider inter-thread communication, exposing limits in the complexities we are able to hide from end-users.

```
closedHat = Hat({ decay:.05 }).connect()
openHat  = Hat({ decay:.2 }).connect()

wrappedHatz = wrap( (ch, oh) => {
  if( Math.random() > .25 )
    ch.trigger( .035 )
  else
    oh.trigger( .05 )
  }
}, closedHat, openHat )

hatSeq = Sequencer.make([wrappedHatz],[5512])
hatSeq.start()
```

Listing 4: Using `wrap()` to bundle the scope needed for a function to properly execute.

Other behaviors are more complicated to implement. For example, consider the *temporal recursion*, a staple of many live-coding performance practices [12]. In a temporal recursion, a function calls itself over time; in between iterations, the live coder has the opportunity to redefine the function so that new behaviors occur on subsequent executions. Temporal recursions are trivial to create in the ScriptProcessor implementation of Gibberish.js without providing any extra support for them; in the AudioWorklet implementation they become impossible without adding functionality to Gibberish.js specifically written to address the complexities of communicating across threads. Given that most JavaScript libraries do not perform sequencing from within audio callbacks, perhaps one solution to many of the problems described in this section would be to add a another type of sequencer to Gibberish.js that runs in the main thread with sufficient lookahead to accurately schedule events in the AudioWorkletProcessor. Although a sequencer running in the main thread would not be able to be modulated at audio-rate, it could potentially enable many techniques, like temporal recursion, that rely on functions having access to the scope where they were instantiated.

5. COMPARISON

While there is a growing selection of JavaScript DSP and music programming systems to choose from [6, 13, 5], we limit our discussion here to systems enabling per-sample processing techniques and possessing AudioWorklet implementations. As AudioWorklets only recently shipped in Chrome, programming with them is a relatively new endeavor. However, the developers of the FAUST programming language have already added AudioWorklets as a target of the FAUST compiler; this compiler can itself also be compiled to WebAssembly and embedded directly in web pages so that DSP graphs can be dynamically compiled [4]. In addition, Csound developers have been developing a browser implementation for quite some time [3]; this version now compiles to WebAssembly and runs in an AudioWorklet[2].

[2]https://github.com/kunstmusik/waaw

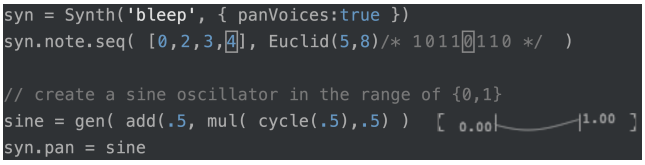

Figure 2: Dynamic annotations and visualizations in Gibber, depicting the state of musical patterns and modulations running in the AudioWorkletProcessor thread.

Both Csound and FAUST are incredibly rich systems for DSP programming; Csound is also built on a long history of music compositional practice. While the capabilities of Genish.js and Gibberish.js are limited in comparison, our libraries do enable JavaScript developers to create new DSP algorithms and perform music programming without requiresiring the use of additional languages. Additionally, embedding Gibberish.js in a web page requires a minimal download, currently thirty-two kilobytes when gzipped; Genish.js is eighteen kilobytes. Finally, the automatic communication of state between the main thread and the AudioWorkletProcessor might be of interest to developers who want to use the current state of unit generators to drive tasks in the main thread, such as visualization. For example, an in-progress update to Gibber makes extensive use of annotations and visualizations reflecting the state of objects in the AudioWorkletProcessor thread, as shown in Fig. 5.

6. CONCLUSIONS

The research presented in this paper enables programmers to use two libraries, Genish.js and Gibberish.js, with AudioWorklet implementations. Through the use of metaprogramming techniques we have hidden the complexities of inter-thread communication from end-users, who are able to use the libraries in almost exactly the same way they would have previously used ScriptProcessor implementations. Using AudioWorklets improves latency, reduces audio dropouts, and prevents audio signal processing from interfering with main thread activities (and vice-versa). Although not all use cases can be accommodated without creating dedicated mechanisms for sharing state between the main thread and the AudioWorkletProcessor thread, such dedicated solutions are often simple to create. We look forward to exploring such solutions further, and to the opportunities that AudioWorklets enable in interactive, audiovisual, browser-based applications.

7. ACKNOWLEDGMENTS

We gratefully acknowledge the W3C Audio Working Group for their work on the AudioWorklet specification, and would like to single out Hongchan Choi for his initial implementation of the AudioWorklet in the Chrome browser.

8. REFERENCES

[1] H. Choi. AudioWorklet: The future of web audio. In *International Computer Music Conference*, 2018.

[2] C. Clark and A. R. Tindale. Flocking: A Framework for Declarative Music-Making on the Web. In *International Computer Music Conference*, pages 1550–1557, 2014.

[3] V. Lazzarini, E. Costello, S. Yi, et al. Csound on the Web. In *Proceedings of the 2014 Linux Audio Conference*, pages 77–84. University of Bath, 2014.

[4] S. Letz, Y. Orlarey, and D. Fober. Faust domain specific audio dsp language compiled to webassembly. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, pages 701–709, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[5] A. Mahadevan, J. Freeman, B. Magerko, and J. C. Martinez. Earsketch: Teaching computational music remixing in an online web audio based learning environment. In *Proceedings of the 1st annual Web Audio Conference*, 2015.

[6] Y. Mann. Interactive Music with Tone.js. In *Proceedings of the 1st annual Web Audio Conference*, 2015.

[7] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[8] C. Roberts. Strategies for per-sample processing of audio graphs in the browser. In *Proceedings of the Web Audio Conference*, 2017.

[9] C. Roberts, J. Allison, D. Holmes, B. Taylor, M. Wright, and J. Kuchera-Morin. Educational design of live coding environments for the browser. *Journal of Music, Technology & Education*, 9(1):95–116, 2016.

[10] C. Roberts, M. Wright, and J. Kuchera-Morin. Beyond editing: extended interaction with textual code fragments. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 126–131, 2015.

[11] C. Roberts, M. Wright, J. Kuchera-Morin, and T. Höllerer. Gibber: Abstractions for creative multimedia programming. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 67–76. ACM, 2014.

[12] A. Sorensen. The Many Faces of a Temporal Recursion. http://extempore.moso.com.au/temporal_recursion.html, 2013.

[13] T. Tsuchiya, J. Freeman, and L. W. Lerner. Data-Driven Live Coding with DataToMusic API. In *Proceedings of the 2nd annual Web Audio Conference*. Georgia Institute of Technology, 2016.

[14] G. Wakefield. *Real-Time Meta-Programming for Interactive Computational Arts*. PhD thesis, University of California Santa Barbara, 2012.

[15] G. Wakefield, W. Smith, and C. Roberts. Luaav: extensibility and heterogeneity for audiovisual computing. In *Proceedings of Linux Audio Conference*, 2010.

APPENDIX

The example below shows a commented example of initializing Gibberish, creating a synthesizer, and creating a sequencer to trigger notes on the sequencer.

```
window.onload = function() {
  // define path to AudioWorklet module.
  Gibberish.workletPath = './gibberish/dist/gibberish_worklet.js'

  // instantiate worklet with call to Gibberish.js.init
  Gibberish.init().then( processorNode => {

    // Create main thread instance of synth instrument. The generated proxy object
    // will send a serialized copy of the synth to AudioWorkletProcessor thread
    // for instantiation.
    const syn = Gibberish.instruments.Synth({
      attack:10,
      decay:22050,
      waveform:'saw'
    }).connect()

    // Create a main thread sequencer instance iterating between playing
    // three different notes every 11025 samples and targeting the synth
    // we just created. The resulting proxy object will create a serialized copy
    // and transfer it to the AudioWorkletProcessor thread for instantiation. The proxy will
    // then send a call to .start() to the sequencer running in the AudioWorkletProcessor thread.
    const seq = Gibberish.Sequencer({
      values:[220,330,440],
      timings:[11025],
      target:syn,
      key:'note'
    }).start()

    // Although the sequencer runs in the AudioWorkletProcessor thread, any changes to the state
    // of objects in the AudioWorkletProcessor thread will be mirrored back to objects
    // in the main thread at the end of each call to AudioWorkletProcessor.process().
    setInterval( ()=> {
      console.log( 'last frequency played:', syn.frequency )
    }, 500 )

  })
}
```