HAL open science

Modernized Toolchains to Create JSPatcher Objects and WebAudioModules from Faust Code

Shihong Ren, Stéphane Letz, Yann Orlarey, Dominique Fober, Romain Michon, Michel Buffa, Laurent Pottier

To cite this version:

Shihong Ren, Stéphane Letz, Yann Orlarey, Dominique Fober, Romain Michon, et al.. Modernized Toolchains to Create JSPatcher Objects and WebAudioModules from Faust Code. WAC 2022 - 7th International Web Audio Conference, Jul 2022, Cannes, France. 10.5281/zenodo.6767596 . hal-03812938

HAL Id: hal-03812938

https://inria.hal.science/hal-03812938v1

Submitted on 13 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modernized Toolchains to Create JSPatcher Objects and WebAudioModules from Faust Code

Shihong Ren
Shanghai Conservatory of Music, SKLMA, China, Univ Jean Monnet, ECLLA Lab, France
shihong.ren@univ-st-etienne.fr

Stéphane Letz
Yann Orlarey
Univ Lyon, GRAME-CNCM, INSA Lyon, INRIA, CITI, EA3720, 69621 Villeurbanne, France
{letz, orlarey}@grame.fr

Dominique Fober
GRAME-CNCM, 11, cours de Verdun (gensoul) Lyon, 69002 France
fober@grame.com

Romain Michon
INRIA, INSA Lyon, Univ Lyon, CITI, EA3720, 69621 Villeurbanne, France
romain.michon@inria.fr

Michel Buffa
Univ Côte d'Azur, I3S Lab, INRIA, France
michel.buffa@univ-cotedazur.fr

Laurent Pottier
Univ Jean Monnet, ECLLA Lab, France
laurent.pottier@univ-st-etienne.fr

ABSTRACT

Faust, as an audio domain specific language (DSL) for DSP, has different compilation targets including WebAudio nodes [7, 4]. An AudioWorklet [2] processor wrapping a WebAssembly DSP can be generated statically from native platform compilers as a JavaScript module with `wasm` files, or dynamically in a browser using a WebAssembly version of the compiler [5]. The WebAssembly compiler has been used in various WebAudio projects through a JavaScript module `faust2webaudio`, such as the Faust IDE [6] and JSPatcher [8].

In the paper, we present recent work on a new version of the Faust WebAssembly compiler for the WebAssembly target and its JavaScript wrapper module: `faustwasm`. Using modern JavaScript tools, the module is designed to be cross-platform and runnable under both Node.js and browser environment to generate, compile, and instantiate WebAssembly binary code from Faust code and wrap the binary as an AudioWorkletProcessor. Based on this module, we created `faust2wam`, a WebAudio plugin generator for the WebAudioModule standard with an automatically generated user interface. We also created a tool as an external package for JSPatcher to generate JSPatcher DSP objects in bulk.

1. INTRODUCTION

Since the support of the WebAssembly features in the AudioWorklet, high performance DSPs can be programmed using low-level code in a WebAudio processor running in a dedicated thread. Despite the complexity of the workflow and the implementation of the JavaScript-side interface, the possibilities of bringing DSP modules from C/C++ code to the Web becomes a hot research topic, as these DSPs initially written for native platforms can be ported to the web using cross-compiler systems like Emscripten [11]. To provide maximum cross-platform compatibility, developers of audio domain specific languages (DSLs), such as Csound [3, 9, 10], Faust or Soul[1] contributed tools, compilers, and IDEs to facilitate the compilation of code from these DSLs to web platforms based on WebAssembly and AudioWorklet.

The Faust language has started to support `asm.js` as a compiler backend since 2015, allowing developers to compile Faust DSPs to JavaScript binary code. Using Emscripten, the Faust compiler itself is also transpiled to a JavaScript module that can dynamically compile and run Faust DSP in the browser. In 2017, a WebAssembly backend has been added in the compiler (`libfaust`) and additional glue code to transform the DSP to fully functional WebAudio nodes has been developed. `faust2webaudio`, a modularized version of the Faust WebAssembly compiler and the WebAudio node wrapper written in TypeScript, has been designed in 2019. It is used in multiple Faust projects such as the Faust IDE and JSPatcher.

In this paper, we will present three JavaScript modules related to the Faust language: an updated version of the Faust WebAssembly compiler `faustwasm`,[2] and two DSP generation tools based on this compiler `faust2wam`[3] and `@jspatcher/package-dsp`.[4]

`faustwasm` is an updated version of the previous JavaScript interface of the Faust WebAssembly compiler `faust2webaudio`. The structure of the module - classes and functions that are usable in the WebAssembly compiler - is refactored and aligned with the Faust C++ compiler. Thanks to the version 2 of the Emscripten tools, the compiler is now compatible with both Node.js and the web environment, while `faustwasm` provides a more flexible solution for DSP compilation, packaging, and executing in different contexts and scenarios. The `faustwasm` module is presented

Licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). **Attribution**: owner/author(s).

Web Audio Conference WAC-2022, December 6–8, 2022, Cannes, France.

© 2022 Copyright held by the owner/author(s).

[1] https://soul.dev/
[2] https://github.com/Fr0stbyteR/faustwasm
[3] https://github.com/Fr0stbyteR/faust2wam
[4] https://github.com/jspatcher/package-dsp

in §2.

`faust2wam` is a JavaScript tool to generate WebAudio plugins in the WebAudioModules format, with an automatically-generated user interface, supporting parameter automation, and MIDI messages. Additional information on this topic can be found in §3.1. `@jspatcher/package-dsp` (The DSP package of JSPatcher) contains a large set of modular DSPs allowing programming audio processor graphically in JSPatcher web application. All the DSPs in the package are in static WebAssembly file format generated using a build system with `faustwasm` (see §3.2).

2. THE FAUSTWASM MODULE

A complete rewrite of the WebAssembly `libfaust` glue code has been started at GRAME at the end of 2020 with the following design principles:

1. using a cleaned version of the C++ low-level interface to be compiled by the Emscripten compiler;
2. using TypeScript as the implementation language, to be directly used by TypeScript-based projects, or compiled as a JavaScript library used by JavaScript-based projects;
3. allowing for a pure static model, where the DSP code is separately precompiled as wasm and JSON files by the FAUST compiler, then loaded and instantiated using the WebAssembly APIs, or dynamically compiled with the libfaust library and then instantiated;
4. allowing the FAUST DSP wasm module to be used completely outside of the WebAudio model, for example as a DSP processing piece of code used in a non-real-time context (like processing an audio file) or wrapped as a WebAudio node,
5. allowing for monophonic and polyphonic MIDI controllable nodes;
6. providing some additional capabilities like SVG diagram rendering, for instance.

The new design will help WebAudio and JavaScript developers to easily create audio effects or instruments from FAUST source code. The generated DSPs are fully modularized and suitable for various JavaScript environments. The generation is platform-independent and only relies on the browser or Node.js.

`faustwasm` is a JavaScript module compatible with recent Node.js versions and web browsers. The majority of the codes has been taken from the GRAME branch `wasm2` from another JavaScript package `@grame/libfaust`[5] from the FAUST repository,[6] revised to have more compatibility on different environments. Multiple classes are provided to cover complete workflows from the import of the FAUST WebAssembly compiler files generated from Emscripten, to the generation of WebAudio nodes or WebAssembly DSP files.

To cite some important classes, FaustCompiler can use the FAUST WebAssembly compiler to generate a FAUST DSP binary code from its source code. The binary code can be wrapped with the FaustWebAudioDsp class for audio processing in different contexts. A utility class FaustDspGenerator can be used to quickly obtain processors from the FAUST code or from a precompiled WebAssembly DSP. Figure 1 shows the workflow of the compilation with or without FaustDspGenerator. Table 1 shows the description of these classes.

Table 1: Classes and their description in `faustwasm`

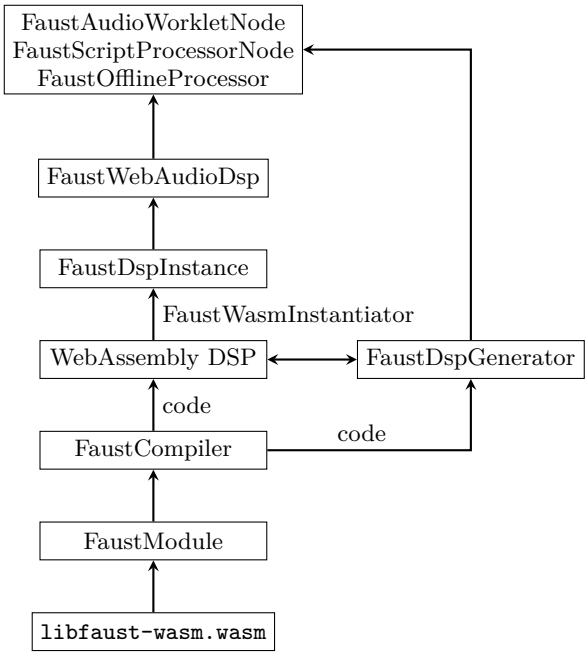
Class	Description
FaustModule	The class exported from the Emscripten compiler, containing raw memory states, function callers and an in-memory file system aligned with C/C++ interface.
LibFaust	A simple JavaScript wrapper class around FaustModule containing only necessary methods.
FaustCompiler	An important class handling the full lifecycle of a FAUST DSP binary code generation process from its source code, including caching, error handling, cleanups, and providing polyphonic instrument mixers.
FaustDsp-Instance	A class interfacing single FAUST DSP instance processing audio buffers, providing information, and handling parameter changes.
FaustWasm-Instantiator	A helper class instantiating DSP instances from its WebAssembly binary code.
FaustWeb-AudioDsp	A higher-level class wraping one or more FaustDspInstances, providing more detailed descriptor on the user interface, and features for JavaScript callbacks, MIDI messages, and parameters.
FaustOffline-Processor	The class provides a `render` method for processing "offline" (as fast as possible) an audio buffer through a FAUST DSP.
FaustDsp-Generator	The class provides a nearly one-stop solution providing functions to create WebAssembly binary DSP codes from FAUST source codes, and generating WebAudio AudioWorkletNodes or ScriptorProcessorNodes using WebAudioDsp interface.
FaustSvg-Diagrams	A utility class that can be used to generate FAUST DSP block diagrams from source code.

The module distribution contains a version of the original FAUST WebAssembly compiler emitted from Emscripten, which includes a JavaScript wrapper file `libfaust-wasm.js`, a WebAssembly binary file `libfaust-wasm.wasm` and a complete FAUST standard library that is loaded at initialization and accessible in the compiler's in-memory file system `libfaust-wasm.data`.

Due to a compatibility issue[7] with Emscripten and its

[5] https://www.npmjs.com/package/@grame/libfaust
[6] https://github.com/grame-cncm/faust
[7] https://github.com/emscripten-core/emscripten/issues/11792

Figure 1: The complete workflow to generate Faust WebAudio nodes from source code using `faustwasm`.



generation of ECMAScript modules, we are still generating the `libfaust-wasm.js` file in CommonJS format. The `faustwasm` module can load correctly the `libfaust-wasm.js` file, and is built for three different formats to maximize its compatibility with different JavaScript environments:

1. CommonJS for legacy Node.js packages. With the same format as the `libfaust-wasm.js` file, it is meant to be used under legacy Node.js environment. When imported in the web environment using the `<script>` tag, it will automatically assign the module under the `faustwasm` object in the global scope.

2. ECMAScript (ES) module for both Node.js and the web environment. It includes an adapter to load the `libfaust-wasm.js` file.

3. ES module bundled with the three files that Emscripten emitted. It is easier to use compared to the previous module, but the file size is larger.

In addition, a command line interface (CLI) is provided for Node.js, the following subsections demonstrate different usages of the module according to Faust's various use cases and scenarios in JavaScript environments.

2.1 Static WebAssembly DSP Generation in Node.js

Before this project, static DSP in WebAssembly format could only be generated using a native distribution of the Faust compiler. Now, it can be done also using `faustwasm`'s Node.js CLI `scripts/faust2wasm.js` under the project repository, which emits `wasm` files along with DSP metadata within `json` files.

The CLI accepts an input source code file path `<input.dsp>` and an output directory path `<output>` with two possible options `[-poly]` and `[-standalone]`:

```
node scripts/faust2wasm.js <input.dsp> <output> [-
    poly] [-standalone]
```

- `[-poly]` option activates the polyphonic mode of the DSP (instrument) and emits additional WebAssembly files that include a voice mixer and a global effect DSP after mixing if it is specified in the DSP source code.

- `[-standalone]` option puts HTML-related files in the output directory including a UI module `faust-ui` and a `faustwasm` distribution to make it independently runnable as a WebAudio node on a web page. The user can open the emitted `index.js` to test the DSP.

2.2 Process Audio Using Faust DSP in Node.js

The `scripts/faust2sndfile.js` CLI can be used to process or generate audio files in Node.js using Faust DSPs by specifying an input source code file path `<input.dsp>` and an output `wav` file path `<output.wav>`. More options can be added to the command using the syntax:

```
node scripts/faust2sndfile.js <input.dsp> <output.
    wav> [-bs <buffer\_size>] [-bd 16|24|32] [-c <
    samples>] [-in <input.wav>] [-sr <sample\_rate
    >]
```

- `[-bs <buffer_size>]` option specifies the buffer size in samples (64 by default).

- `[-bd 16|24|32]` option determines the encoding bit depth of the output `wav` file, possible values are 16, 24 and 32 (16 by default).

- `[-c <samples>]` determines the final length of the audio file generated in samples, ($5 * sample_rate$ by default).

- `[-in <input.wav>]` allows us to specify an input audio file in `wav` format that can be processed using the DSP.

- `[-sr <sample_rate>]` option determines the encoding sample rate of the output `wav` file (44100 by default).

2.3 Generate Block Diagram Files in Node.js

The `scripts/faust2svg.js` CLI can be used to generate Faust block diagram files in `svg` format. The syntax is as follows:

```
node scripts/faust2svg.js <input.dsp> <output>
```

where `<input.dsp>` is the input Faust DSP source code and the `<output>` is the output directory for the `svg` files.

2.4 Statically or Dynamically Generate Faust DSPs on the Web

With `faustwasm`, we can generate *statically* WebAudio nodes from the precompiled WebAssembly DSP binary code, or *dynamically* generate these nodes starting from compiling the Faust DSP source code.

Once a FaustDspGenerator instance is initialized using the compiler's WebAssembly files, a Faust DSP source code

can be compiled into a WebAssembly binary code. Still using the FaustDspGenerator, the binary code - provided either from ArrayBuffers or from precompiled `wasm` files with its `json` metadata - can be used to create higher-level DSPs (WebAudio nodes or FaustOfflineProcessors) runnable in the web environment.

3. USAGES AND EXAMPLES

3.1 The faust2wam Module

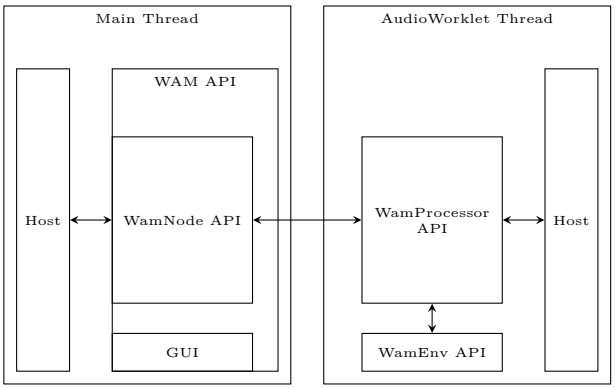
WebAudioModule (WAM) standard is a web-based audio plugin format first released in 2015. In 2021, a new version of the WAM standard has been redesigned to reflect recent technological developments [1]. A WAM plugin can be fetched from the web using a URI, and initialized using the current WebAudio context. The plugin comes with a standardized API that can create a user interface, get or adjust parameters, schedule events such as parameter automation or MIDI messages, and connect with other WAMs or native WebAudio nodes.

The `faustwasm` and the `faust-ui` modules already provide the interface needed for a WAM. Using these modules, we created the `faust2wam` module to automatically generate a WAM from a Faust code. For any web-based DAW with WAM support, a large amount of audio plugins can be quickly integrated with less effort.

3.1.1 Faust WAM Structure

Figure 2 shows the structure of a WAM plugin, in which the AudioNode of the WAM should be able to interact with the host in both the main and the audio thread.

Figure 2: The WebAudioModule API structure.



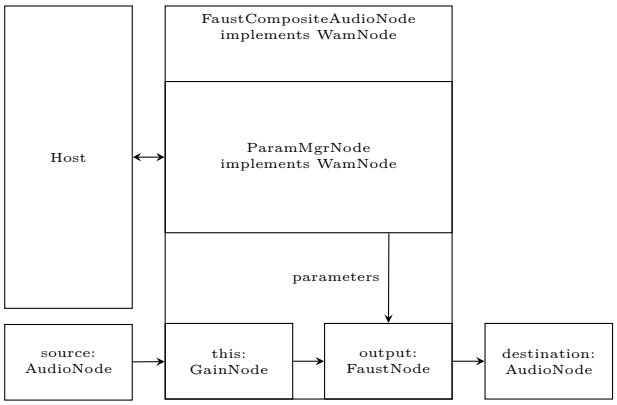
Since the WamNode API is different from the AudioWorkletNode API - especially for event handling and parameter automation - it needs to be implemented separately. From the WAM SDK, we imported the Parameter Manager (ParamMgr) and the CompositeNode as a WamNode API proxy to wrap the FaustAudioWorkletNode. The ParamMgr will be able to automate AudioParams inside the FaustAudioWorkletNode, and to send MIDI messages from the WAM API. The node is generated from precompiled static WebAssembly Faust DSP or from Faust source code in the WAM's initialization phase using `faustwasm`. The WAM also provides a user interface generated using the `faust-ui` modules.

The system actually produces an extended version of the standalone DSP on the web runtime presented in the Subsection 2.1. During the initialization phase, the WAM will execute the following steps to construct an AudioNode that conforms the required WAM API (Figure 3):

1. `json` and `wasm` files containing the DSP's metadata and binary codes will be fetched and compiled to WebAssembly Modules.
2. A FaustAudioWorkletNode (FaustNode) will be created using the FaustDspGenerator.
3. The node's parameters will be used to initialize a ParamMgr that proxies the WamNode API.
4. The FaustCompositeNode that wraps the ParamMgr and the FaustNode will be created as the "main" node of the WAM to connect with other AudioNodes.

In addition, a function that can create a user interface from the DSP's metadata using `faust-ui` in a WebComponent is attached to the WAM.

Figure 3: The FaustCompositeNode structure.



3.1.2 Generate from Node.js

A Node.js CLI is provided to generate a WAM from the Faust source code. Similarly to the `faustwasm` CLI, an input source code file path `input.dsp` and an output directory path `output` are required, but with one possible option `[-poly]` to activate the polyphonic mode:

```
node faust2wam.js <input.dsp> <output> [-poly]
```

The generated files inside the `output` directory will be self-contained, including all the modules required by the WAM.

3.1.3 Generate on the Web

Thanks to the Faust WebAssembly compiler in the `faustwasm` module, it is possible to generate a Faust WAM directly in the web environment.

A `compile` function is provided in the `faust2wam` package to compile Faust code to the WebAssembly binary code with the DSP metadata. Then, users can call the `generateWam` function to generate the FaustAudioWorkletNode and wrap it into a WAM, which will be the same as from static files.

Figure 4 shows a WAM generated by `faust2wam` module.

Figure 4: A Faust polyphonic WAM generated by `faust2wam` and loaded in JSPatcher, connected with a virtual keyboard and an oscilloscope.

3.2 The DSP Package of JSPatcher

JSPatcher is a visual programming language (VPL) on the web written in TypeScript in the style of Max or PureData allowing users to create interactive online applications by building a graph - connecting ports (inlets and outlets) of the "box objects" with virtual patch chords.

A box object can be functional, accepting data as function inputs from its inlets connected from other box objects. Then, after applying the function to the input, it will output data to other box objects from its connected outlets.

Meanwhile, if supported, a box object can be used as a WebAudio DSP node, while its ports represent audio connections as in a WebAudio graph. Cables will be colored in this case.

Since an SDK for creating third-party box objects is available publicly, based on the `faustwasm` module, a set of JSPatcher DSP objects for basic signal calculations is created as a package that is added into JSPatcher, to extend its possibilities for real-time signal processing.

In the design, this statically compiled Faust DSP box object can have several options during runtime, including setting initial values of most of the inputs and setting interpolation time when these inputs are changed by messages (which is 0.01 second by default).

A Faust library `maxmsp.lib`[8] has implemented in the Faust language a part of the DSPs in Max. Our work reused some functions from the library and quickly prepared for each DSP box object a dedicated Faust source code file. For example, to prepare the source code of the low-pass filter (LPF), we only need to write the following:

```
declare argsOffset "1";
declare description "Low Pass Filter";
declare inputsDescription "[``,`f0`,`gain`,`Q`]";
import("maxmsp.lib"); // Import the Library
process = LPF; // Set as the main process
```

Here, the first three lines allow providing additional metadata with the compiled DSP, to be injected into the JSPatcher box object. `argsOffset` is declared to be 1, which means that the arguments of the resulting box object will be applied to the inputs with an offset of 1 (i.e., the first argument to the second input). The `description` and the `inputsDescription` will be showed with the box object as its metadata. Due to the limitations of the compiler, in `inputsDescription`, backquotes are used as delimiters for strings and will be replaced as quotes in order to be parsed as a JSON array.

To support the initial input value when the box object's inlet is not connected, we may also need to declare the value through the metadata. The code below is another example showing how we declare initial input values in the Faust source code.

```
declare defaultInputs "[0, 0.5]";
declare description "Sawtooth waveform oscillator
    between 0 and 1 with phase control";
declare inputsDescription "[`freq`, `phase`]";
import("stdfaust.lib");
process = os.lf_sawpos_phase;
```

The `defaultInputs` indicates (in order) the default input values as a JSON array of numbers.

Once the source files are prepared, we use the `faustwasm` CLI to compile all the files to its WebAssembly binary and JSON metadata files.

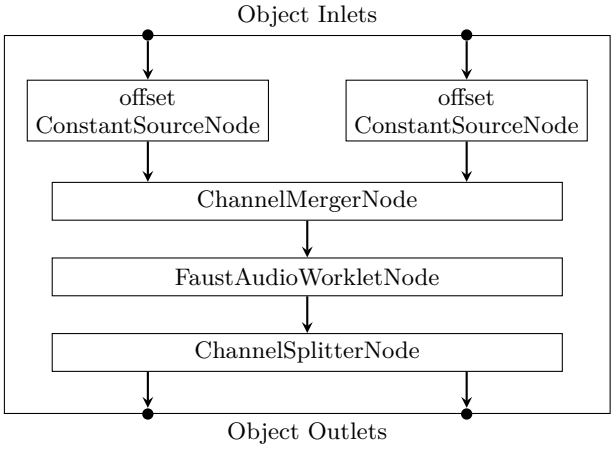
Then, a prototype JSPatcher box object is created as a class that extends (`DefaultObject`). It includes the FaustDspGenerator for compiling the WebAssembly binary to the FaustAudioWorkletNode, taking into account the metadata declared in the source code. In fact, to implement the default argument values, for each Faust DSP input (which is actually different FaustAudioWorkletNode channels of its first input), a ConstantSourceNode is initialized with the declared value as its `offset` AudioParam value and connected to a ChannelMergerNode which connected to the FaustAudioWorkletNode. To allow us to separate connections from each DSP output, a ChannelSplitterNode is connected to the first output of the Faust node. (Figure 5)

With this setup, the box object's inlets can be connected with audio connections for *a-rate* parameter changes on the `offset` AudioParam, which is eventually treated as signal inputs. Without audio connections, they can also handle data messages to set discretely input values with a smooth factor, which can be set through the box object's `smoothInput` property.

Figure 6 shows that when the user added a Faust DSP box object in JSPatcher, the detailed descriptions declared

[8] https://github.com/grame-cncm/faustlibraries/blob/master/maxmsp.lib

Figure 5: The internal WebAudio graph in a Faust DSP box object.



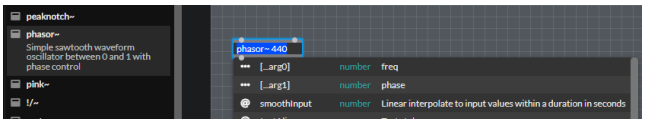


Figure 6: The compiled phasor box object in JSPatcher

in the metadata will show up. In this case, a simple phasor waveform generator is created using the `phasor~` box object. It has two inputs, `freq` and `phase`, which can be initialized with arguments. Figure 7 shows the output signal or the `phasor~` box object with the frequency set to 110Hz. We connect the output to another FAUST DSP box object subtracting the signal by 0.5, then to an oscilloscope to visualize the signal.

4. DISCUSSION

The `faustwasm` module uses several recently implemented Web APIs, including ES module, WebAssembly, AudioWorklet, etc. Its browser compatibility need to be investigated. According to caniuse.com, WebAssembly and ES modules with dynamic import support are available on popular browsers (Chromium, Firefox, and Safari) since May 2020. AudioWorklet API is supported on Chromium since 2018, on Firefox 2020. Safari started supporting since 2021, but not allowing passing constructor options to the processor. In 2022, Safari's Technology Preview has full support for AudioWorklet.

After releasing the first version of `faustwasm`, we have a feature request for sample-accurate handling for MIDI messages and parameter changes. A solution is proposed by the community and will be addressed soon. We are also expecting to see more use cases from the FAUST community. In theory, multi-rate audio processing or non-audio data stream processing (i.e. frequency domain data calculation) will be possible using this module on the web. We will keep the module up to date to keep the FAUST language usable in new audio projects and new web technologies.

5. CONCLUSION

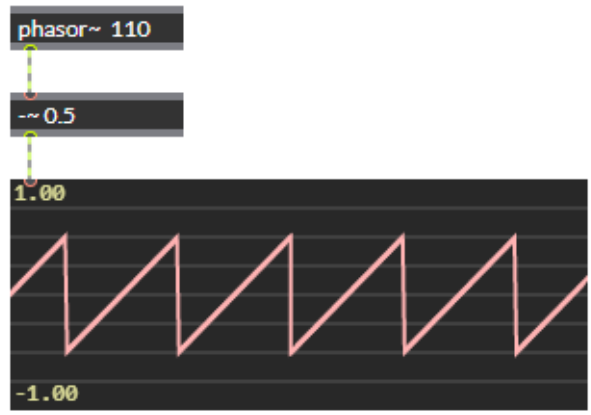


Figure 7: The signal that the phasor box object generated.

Cross-platform compatibility is one of the main design features of the FAUST language. It is also one of the capabilities provided by the web standards and browsers. Different modern Web APIs and web-related programming workflows, including new languages, building systems, and rendering frameworks require the FAUST compiler and DSPs to be highly flexible and modularized for easier integration into third-party projects.

Both `faust2wam` and the DSP package for JSPatcher are proofs of concept of using `faustwasm` as a NPM (Node Package Manager) module for WebAudio related projects. We are going to merge the module into FAUST official repository and replace old web compilers in our online tools such as the FAUST Editor, the FAUST Playground and the FAUST IDE. With the next iteration of these tools, the FAUST compilation service on a remote server is no longer needed for JavaScript targets, WAMs with a UI can be efficiently generated in a browser.

Web-based audio application like JSPatcher often need a set of DSPs implemented as AudioWorkletNodes. `faustwasm` could help developers in this case concentrate on the DSP algorithm and realize the JavaScript programs for them. With less coding efforts, we successfully created around 50 DSPs for JSPatcher and will add more in the future.

6. ADDITIONAL AUTHORS

Additional author: Yang Yu (Shanghai Conservatory of Music, SKLMA, China, email: `yuyang@shcmusic.edu.cn`).

7. REFERENCES

[1] M. Buffa, S. Ren, O. Campbell, T. Burns, S. Yi, J. Kleimola, and O. Larkin. Web Audio Modules 2.0: an Open Web Audio Plugin Standard. In *Companion Proceedings of the Web Conference*, Lyon, France, Apr. 2022.

[2] H. Choi. Audioworklet: the future of web audio. In *Proceedings of the International Computer Music Conference*, pages 110–116, Daegu, South Korea, Aug. 2018.

[3] V. Lazzarini, S. Yi, J. Ffitch, J. Heintz, O. Brandtsegg, and I. McCurdy. *Csound: A Sound and Music Computing System*. Springer Publishing Company, Incorporated, 1st edition, 2016.

[4] S. Letz, S. Denoux, Y. Orlarey, and D. Fober. Faust audio DSP language in the Web. In *Proceedings of the Linux Audio Conference*, pages 29–36, Mainz, Germany, Apr. 2015.

[5] S. Letz, Y. Orlarey, and D. Fober. Compiling faust audio dsp code to webassembly. In F. Thalmann and S. Ewert, editors, *Proceedings of the International Web Audio Conference*, London, United Kingdom, Aug. 2017. Queen Mary University of London, Queen Mary University of London.

[6] S. Letz, S. Ren, Y. Orlarey, R. Michon, D. Fober, E. Aamari, M. Buffa, and J. Lebrun. Faust online ide: dynamically compile and publish faust code as webaudio plugins. In A. Xambó, S. R. Martín, and G. Roma, editors, *Proceedings of the International Web Audio Conference*, WAC '19, pages 71–76, Trondheim, Norway, December 2019. NTNU.

[7] Y. Orlarey, D. Fober, and S. Letz. FAUST : an Efficient Functional Approach to DSP Programming. In E. D. France, editor, *New Computational Paradigms for Computer Music*, pages 65–96. Paris, France, Jan. 2009.

[8] S. Ren, L. Pottier, and M. Buffa. Build webaudio and javascript web applications using jspatcher: A web-based visual programming editor. In L. Joglar-Ongay, X. Serra, F. Font, P. Tovstogan, A. Stolfi, A. A. Correya, A. Ramires, D. Bogdanov, A. Faraldo, and X. Favory, editors, *Proceedings of the International Web Audio Conference*, WAC '21, Barcelona, Spain, July 2021. UPF.

[9] S. Yi, V. Lazzarini, and E. Costello. Webassembly audioworklet csound. In J. Monschke, C. Guttandin, N. Schnell, T. Jenkinson, and J. Schaedler, editors, *Proceedings of the International Web Audio Conference*, WAC '18, Berlin, Germany, September 2018. TU Berlin.

[10] S. Yi, H. Sigurðsson, and E. Costello. Csound web-ide. In *Proceedings of the International Web Audio Conference*, pages 92–97. Trondheim Norway, 2019.

[11] A. Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312, 2011.