Real Time Audio Digital Signal Processing With Faust and the Teensy

Romain Michon, Yann Orlarey, Stéphane Letz, Dominique Fober

Real Time Audio Digital Signal Processing With Faust and the Teensy

Romain Michon,[1,2] **Yann Orlarey,**[1] **Stéphane Letz,**[1] **and Dominique Fober**[1]
[1] GRAME-CNCMC, 11 Cours de Verdun-Gensoul, 69002 Lyon (France)
[2] Center for Computer Research in Music and Acoustics, 660 Lomita Ct., Stanford CA 94350-8180 (USA)
rmichon@ccrma.stanford.edu

ABSTRACT

This paper introduces a series of tools to program the Teensy development board series with the FAUST programming language. `faust2teensy` is a command line application that can be used both to generate new objects for the Teensy Audio Library and standalone Teensy programs. We also demonstrate how `faust2api` can produce Digital Signal Processing engines (with potential polyphony support) for the Teensy. Details about the implementation and optimizations of these systems are provided and the results of various tests (i.e., computational, latency, etc.) are presented. Finally, future directions for this work are discussed through a discussion on bare-metal implementation of real-time audio signal processing applications.

1. INTRODUCTION

Arduinos [1] contributed to the spreading of microcontrollers by making them more accessible through a high level programming language (which is essentially a subset of C++), various domain-specific libraries, and an Integrated Development Environment (IDE) allowing to export the generated firmware to the board using USB.

The impact of the "Arduino revolution" on the computer music/NIME (New Interfaces for Musical Expression) community has been significant and gave birth to hundreds of new music controllers and instruments.

In parallel of that, the rise of embedded Linux platforms with their potential applications to real-time audio signal processing also impacted the way we approach digital lutherie [1]. A series of tools (both hardware and software) such as Satellite CCRMA [2] and the BELA [2] [3] (to only name a few) have been exploiting the potential of these new technologies. The BELA is especially interesting to us as it adds audio-rate analog I/Os and low-latency audio processing capabilities to the BeagleBone Black. [3] More specific applications and experiments have also been targeting specialized boards such as FPGAs (Field Programmable Gate Array) (e.g., Digilent Zybo Series [4]) [4,5] and DSPs (Digital Signal Processing) (e.g., Analog Devices SHARC Audio Module, [5] etc.). The main drawback of these platforms is their price: fully functional "all-in-one" solutions can't be found for less than 100USD.

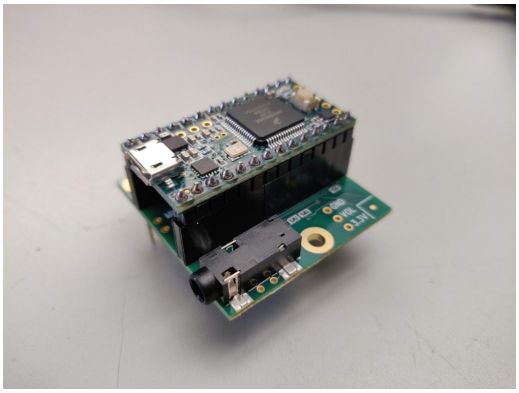

Figure 1. The Teensy 3.2 and its audio shield.

On the other hand, recent microcontrollers are cheap, offer an heightened computational power, and can be used to synthesize/process audio signals. Some of them such as the ARM Cortex-M4F [6] even include a dedicated Floating-Point Unit (FPU) and support for DSP instructions, making them a well-suited platform for sound synthesis/processing.

ARM Cortex-M4 microcontrollers are used at the heart of PJRC's Teensy [7] development board series whose price averages 25USD. Since the Cotex-M4 hosts its own DAC, [8] sound can be synthesized and played directly from the Teensy. PJRC also offers an audio shield for the Teensy that essentially upgrades it with a SGTL5000 Audio Codec providing a 16bits 44.1kHz stereo audio input and output (see Figure 1) for 15USD!

The Teensy comes with an Audio Library [9] where basic DSP objects (e.g., oscillators, effects, Karplus-Strong, etc.) implemented in C++ can be patched with a high level API. This task is facilitated by an online graphical environ-

[1] `https://www.arduino.cc/`. All URLs presented in this paper were verified on Feb. 4, 2019.
[2] `https://bela.io/`
[3] `https://beagleboard.org/bone`

[4] `https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/`
[5] `https://wiki.analog.com/resources/tools-software/sharc-audio-module`
[6] `https://developer.arm.com/products/processors/cortex-m/cortex-m4`
[7] `https://www.pjrc.com/teensy/`
[8] `DigitaltoAudioConverter`
[9] `https://www.pjrc.com/teensy/td_libs_Audio.html`

ment [10] where objects can be connected by drawing patch chords between them.

The standard DSP objects of the Teensy Audio Library are relatively basic. New objects can be implemented in C++, which is often out of reach to people in the DIY (Do It Yourself) community who might use the Teensy. Also, since not all the microcontrollers used in the Teensy series have an FPU, the standard DSP objects of the Teensy Audio Library are all implemented in fixed point.

FAUST [6] is a functional programming language for efficient real-time audio signal processing. The FAUST compiler can generate DSP code/classes in various languages (i.e., C, C++, Java, JS, LLVM, WebAssembly, etc.) from a given FAUST program. The FAUST DSP libraries implement hundreds of algorithms for sound synthesis and processing.

In this paper, we introduce a series of tools to program the Teensy with FAUST. [11] First, we present `faust2teensy`, a command-line application to generate new DSP objects for the Teensy Audio Library. Then, we introduce a new target for `faust2api` [7] allowing us to generate DSP engines (with potential polyphony support) for the Teensy. We also demonstrate how ready-to-use Teensy programs can be written in FAUST. In that case, the parameters of a FAUST program (e.g., the frequency of an oscillator, etc.) can be directly mapped to the analog and digital inputs of the Teensy. Finally, we evaluate the performances of these systems and we present future directions for this type of work.

2. CREATING NEW OBJECTS FOR THE TEENSY AUDIO LIBRARY

2.1 Generating DSP Objects With `faust2teensy`

`faust2teensy` is a command-line tool that can be used to generate new DSP objects compatible with the Teensy Audio Library. [12] For this, the `-lib` option must be provided when calling `faust2teensy`:

```
faust2teensy -lib MyFaustSynth.dsp
```

which will yield a package containing two C++ files: `MyFaustSynth.h` and `MyFaustSynth.cpp` (see Figure 2) implementing a class called `MyFaustSynth`.

These files can either be placed in the source of the Teensy Audio Library or in their own library. In both cases, the `.h` file should be included at the beginning of the Teensy program and then called and connected at least to a DAC (Digital to Audio Converter) just like any other object of the Teensy Audio Library:

```
#include <Audio.h>
#include <MyFaustSynth.h>
MyFaustSynth myFaustSynth;
AudioOutputAnalog dac;
AudioConnection
  patchCord0(myFaustSynth,dac);
void setup() {
  AudioMemory(2);
}
void loop() {
}
```

Listing 1. Simple Teensy program using a FAUST-generated DSP object with the built-in DAC of the Teensy.

`AudioOutputAnalog` corresponds to the built-in DAC of the Teensy but `AudioOutputI2S` could be used instead, in case the Teensy is equipped with an audio shield. In that case, an `AudioControlSGTL5000` object should also be instantiated and multi-channel audio connections can be used:

```
#include <Audio.h>
#include <MyFaustSynth.h>
MyFaustSynth myFaustSynth;
AudioOutputI2S dac;
AudioControlSGTL5000 audioShield;
AudioConnection
  patchCord0(myFaustSynth,0,dac,0);
AudioConnection
  patchCord1(myFaustSynth,0,dac,1);
void setup() {
  AudioMemory(2);
  audioShield.enable();
}
void loop() {
}
```

Listing 2. Simple Teensy program using a FAUST-generated DSP object with the Teensy Audio Shield.

Note that the number of audio inputs and outputs of the generated object depends on the FAUST program, hence `MyFaustSynth` could have more than one output, in which case the wiring of the audio connections could look like:

```
AudioConnection
  patchCord0(myFaustSynth,0,dac,0);
AudioConnection
  patchCord1(myFaustSynth,1,dac,1);
```

The following Code Listing presents an example FAUST program implementing a band-limited sawtooth wave oscillator that could be used as `MyFaustSynth.dsp`:

```
import("stdfaust.lib");
freq = nentry("f",300,50,2000,0.01) :
  si.smoo;
gain = nentry("g",0.5,0,1,0.01) :
  si.smoo;
process = os.sawtooth(freq)*gain;
```

Listing 3. FAUST program implementing a band-limited sawtooth oscillator controllable with some UI elements.

`f` controls the frequency of the oscillator and `g` its gain. Both are processed by `si.smoo` which exponentially interpolates samples, preventing discontinuities.

[10] `http://www.pjrc.com/teensy/gui/index.html`

[11] All the tools presented in this paper are open source and have been integrated to the FAUST distribution which can be found on GitHub: `https://github.com/grame-cncm/faust`.

[12] We'll see in §4 that `faust2teensy` can also be used to generate ready-to-use Teensy programs.

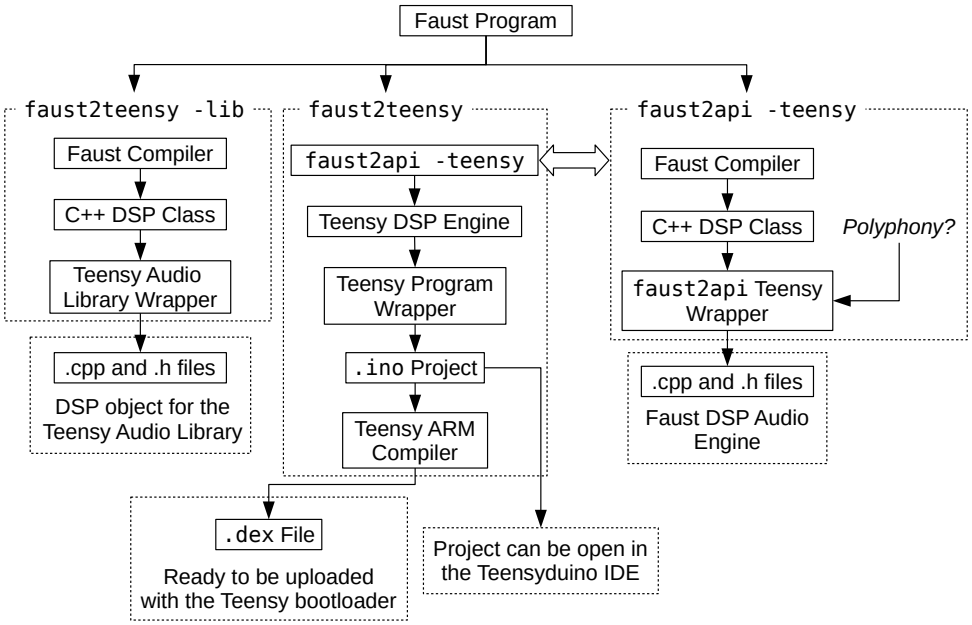

Figure 2. Overview of the various tools to use FAUST on the Teensy.

The value of `f` and `g` can be set at any point by calling the `setParamValue` method which takes the name of the FAUST parameter and its corresponding value as arguments. Note that this method can also be used with `faust2api` (see §3) as it is inherited from the same parent class (`MapUI`) of the FAUST architectures system. Hence, the frequency of the FAUST sawtooth oscillator presented in Code Listing 3 could be set randomly every two-hundred milliseconds by modifying the implementation of the `loop` function [13] of Code Listing 1:

```
void loop() {
  myFaustSynth.setParamValue("f",
    random(50,2000));
  delay(200);
}
```

2.2 Implementation

`faust2teensy` is a simple bash script calling the command-line FAUST compiler to generate the C++ DSP class corresponding to a given FAUST program (see Figure 2). The generated C++ code is pasted into a wrapper C++ file (also called architecture in the FAUST world) implementing a generic object for the Teensy Audio Library. This file and its corresponding header file are then packaged in a zip file which is provided to the user.

[13] `loop` is a standard Arduino function that is repeated until the device running it is powered off. In other words, it corresponds to the main thread of the system. `delay` is also an Arduino function that can be used to pause the thread for a given duration in milliseconds.

3. GENERATING FAUST DSP ENGINES FOR THE TEENSY

3.1 Monophonic DSP Engine

FAUST can also be used to generate ready-to-use DSP engines for the Teensy. In that case, the strategy consists in letting FAUST taking care entirely of the sound synthesis/processing portion of the program.

The FAUST program presented in Code Listing 3 can be turned into a DSP engine using `faust2api` by running the following command line in a terminal:

```
faust2api -teensy MyFaustSynth.dsp
```

Just like `faust2teensy` (see §2), the generated package contains a `.cpp` and a `.h` file that can be called directly in a Teensy program:

```
#include <MyFaustSynth.h>
int SR = 44100; // Sampling Rate
int BS = 128; // Block Size
MyFaustSynth myFaustSynth(SR,BS);
void setup() {
  myFaustSynth.setDevice(0,0);
  myFaustSynth.start();
}
void loop() {
  myFaustSynth.setParamValue(
    "f",random(50,2000));
  delay(200);
}
```

Listing 4. Teensy program using a FAUST-generated monophonic DSP engine.

Here, the `MyFaustSynth` object relies on the Teensy Audio Library which it calls internally. The `setDevice` method allows the programmer to specify the hardware

Device ID	Device Description
0	Teensy Audio Shield
1	Teensy Audio Shield (Quad)
2	Built-In ADC
3	Built-In ADC (Stereo)
4	Teensy Audio Shield (Slave Mode)
5	Pulse Density Modulated Bitstream
6	Time Division Multiplexed Frame
7	USB: receive stereo audio from computer

Table 1. Input devices ID for the `setDevice` method (directly taken from the Teensy Audio Library).

Device ID	Device Description
0	Teensy Audio Shield
1	Teensy Audio Shield (Quad)
2	SPDIF
3	PT8211 DAC
4	Built-In DAC
5	Built-In DAC (Stereo)
6	PWM
7	Teensy Audio Shield (Slave Mode)
8	Time Division Multiplexed Frame
9	ADAT
10	USB: send stereo audio to computer

Table 2. Output devices ID for the `setDevice` method (directly taken from the Teensy Audio Library).

input (see Table 1) and output (see Table 2) of the DSP engine. `start` launches computation and `setParamValue` is used to change the value of a specific parameter (as with `faust2teensy`). All the other standard `faust2api` methods [7] are also available.

3.2 Polyphonic DSP Engine

While the benefits of using a Teensy monophonic DSP engine generated with `faust2api` over `faust2teensy` might be questionable, `faust2api` also allows us to generate polyphonic audio engines that can be used with a specific API.

A FAUST program can be made "polyphony-compatible" simply be declaring the `freq`, `gain`, and `gate` parameters. [14] In that case, an audio effect common to all voices can be declared using the `effect` standard declaration. Hence, Code Listing 3 can be easily modified to make it polyphony-compatible (`MyFaustSynthPoly.dsp`):

```
import("stdfaust.lib");
freq = nentry("freq",300,50,2000,0.01);
gain = nentry("gain",0.5,0,1,0.01);
gate = button("gate");
envelope = en.asr(0.01,gain,0.01,gate);
process = os.sawtooth(freq)*envelope;
effect = +~@(ma.SR*0.15)*0.3; // echo
```

Listing 5. FAUST program implementing a MIDI-controllable polyphonic synthesizer.

The FAUST program presented in Code Listing 5 can be turned into a polyphonic DSP engine by running the following command line:

```
faust2api -teensy -nvoices 4 -effect auto
    MyFaustSynthPoly.dsp
```

Note that `-nvoices` allows us to specify the maximum number of voices of polyphony of the engine and that `-effect auto` connects all the voices to the effect declared in the `effect` standard declaration.

The generated DSP engine allows for the use of polyphony-related methods [7] such as `keyOn`, `keyOff`, `newVoice`, `deleteVoice`, `setVoiceParamValue`, etc. Code Listing 6 demonstrates the use of a polyphonic DSP engine by generating random major chords.

```
#include <MyFaustSynthPoly.h>
MyFaustSynthPoly myFaustSynth(44100,128);
void setup() {
  myFaustSynth.setDevice(0,0);
  myFaustSynth.start();
}
void loop() {
  int root = random(40,80);
  int M3 = root+4; int P5 = root+7;
  myFaustSynth.keyOn(root,100);
  myFaustSynth.keyOn(M3,100);
  myFaustSynth.keyOn(P5,100);
  delay(1000);
  myFaustSynth.keyOff(root,100);
  myFaustSynth.keyOff(M3,100);
  myFaustSynth.keyOff(P5,100);
  delay(500);
}
```

Listing 6. Teensy program using a FAUST-generated polyphonic DSP engine.

Figure 2 gives an overview of the implementation of this system.

4. USING FAUST TO PROGRAM THE TEENSY

`faust2teensy` can be used in "standalone mode" to fully program the Teensy directly from FAUST without writing a single line of Arduino code. This is done through the use of specific metadata in the declaration of the name of the parameters of the FAUST program. Hence, `[io: AN]` can be used to connect the `N` analog pin to the current parameter. The same approach is used for digital pins using the `[io: DN]` metadata.

Global metadata can be declared as well to configure the sampling rate (`declare SR`), the block size (`declare BS`), and the audio input and output (`declare device` – see Tables 1 and 2 for a list of available inputs and outputs) of the Teensy.

Code Listing 7 presents a FAUST program where analog pins 0 and 1 of the Teensy control respectively the fre-

[14] https://faust.grame.fr/doc/manual/index.html#midi-polyphony-support

quency and the gain of a sawtooth oscillator and digital pin 0 the fact that it's on or off.

```
declare SR "44100";
declare BS "128";
declare device "{0,0}";
import("stdfaust.lib");
f = nentry("f[io: A0]",
  300,50,2000,0.01) : si.smoo;
g = nentry("g[io: A1]",
  0.5,0,1,0.01) : si.smoo;
t = nentry("t[io: D0]",
  0,0,1,1) : si.smoo;
process = os.sawtooth(f)*g*t;
```

Listing 7. Standalone FAUST Teensy program.

Note that the range of analog pins on the Teensy is automatically mapped to that of the corresponding FAUST parameter. Hence, in the case of the `f` parameter in Code Listing 7, if the Teensy uses 10 bits integers to store the values of the samples acquired by analog pin 0, 0 will correspond to a value of `f` of 50 and 1023 to a value of `f` of 2000. The same is true for digital pins.

`faust2teensy` can be used in standalone mode simply by running the following command in the terminal:

```
faust2teensy MyFaustSynth.dsp
```

In that case, `faust2teensy` will automatically call the Teensy bootloader to upload the generated firmware, so the Arduino IDE doesn't have to be used at all! Note that the `-vb` option can be added to verbose the output of the compilation process.

5. EVALUATION

The tools presented in §2-4 have been evaluated through the simple FAUST program presented in Code Listing 8 which implements a sawtooth oscillator from scratch.

```
import("stdfaust.lib");
freq = hslider("freq",400,50,2000,0.01);
frac(n) = n - floor(n);
sawtooth(f) = +(f/ma.SR)~frac : *(2)-1;
process = sawtooth(freq) <: _,_;
```

Listing 8. Standalone FAUST Teensy program.

This algorithm was chosen for its simplicity (only three additions/subtractions and one multiplication). The corresponding C++ code generated by the FAUST compiler was used with `faust2api` in polyphonic mode (see §3.2) to measure the number of cycles per seconds on the Teensy under various conditions. This code was re-written "by hand" to use fixed points instead of floating points without changing the algorithm (the FAUST compiler can only generate floating point DSP code).

Tests were carried out on a Teensy 3.2 (MK20DX256VLH7 Cortex-M4 72MHz) and a Teensy 3.6 (MK66FX1M0VMD18 Cortex-M4F 180MHz). Since 3.6 has an FPU, floating point instructions were forced by using the following options during compilation: `-mfloat-abi=hard -mfpu=fpv4-sp-d16` (selects a hardware floating-point unit conforming to the single precision variant of the FPv4 architecture) when testing DSP code using floating points. Since 3.2 doesn't have an FPU, no specific C++ compilation options for floating points were selected (emulated floating points).

The results of our tests are presented in Table 3. All tests were carried out at a sampling rate of 44.1KHz. We chose 128 samples as our maximum test block size since using higher block sizes doesn't seem to impact computation. 8 samples is the smallest block size that we were able to use on the Teensy. "MaxPoly" corresponds to the maximum number of voices of polyphony based on Code Listing 8 that we were able to run.

As expected, the Teensy 3.6 outperforms the 3.2 for all tests. While there's only a gain factor of $\sim$2.2 between these two devices when using fixed point arithmetic, the 3.6 was 15 times more powerful in average than the 3.2 when using floating points. Hence, 63 parallel versions of the algorithm presented in Code Listing 8 (which corresponds to a total of 64 multiplications and 251 additions/-subtractions per sample) could be ran in parallel and added on the 3.6 while the 3.2 only allowed to play 4 voices. Another interesting element to note is that the impact of block size on computation is rather small. Hence, a block size of 8 samples is only 1.2 times more expensive in average than a block size of 128 samples (or greater) in most cases.

These performances could potentially be improved by using CMSIS-DSP instructions,[15] but since all the optimized math function of this library are vector-based, they're only useful for very specific kinds of algorithms. For instance, they would not help make the program presented in Code Listing 8 more officient because of its internal feedback.

More complex algorithms such as the FAUST version of Zita-Verb (stereo feedback delay network) [8] were also ran successfully on the Teensy 3.6.

Finally, audio "round-trip" (analog to digital and then back to analog) latency measurements were carried out (also using a sampling rate of 44.1KHz). When using a block size of 128 samples, a latency of 10.5ms was measured. When using a block size of 8 samples, a latency of 1.2ms was measured!

6. FUTURE DIRECTIONS

The current set of metadata available to produce standalone Teensy programs with FAUST (see §4) is somewhat limited and could be easily extended. For example, it is currently not possible to map sensors using i2s (Integrated Inter-IC Sound Bus) to the parameters of a FAUST program, etc. We plan to expand the scope of these metadata in the future to allow users to program the Teensy in FAUST without making compromises.

Microcontrollers and CPUs for embedded systems now offer enough processing power to implement complex real-time audio signal processing algorithms, but little work has

[15] `http://www.keil.com/pack/doc/CMSIS/DSP/html`

	Block Size	Teensy 3.2	Teensy 3.6
`int16`	128	∼2,004,700 c/s	∼4,801,450 c/s
MaxPoly `int16`	128	34 (∼32,150 c/s)	75 (∼31,350 c/s)
`int32`	128	∼1,113,700 c/s	∼2,402,900 c/s
MaxPoly `int32`	128	20 (∼23,950 c/s)	33 (∼55,550 c/s)
`float32`	128	∼222,650 c/s	∼3,512,600 c/s
MaxPoly `float32`	128	4 (∼25,700 c/s)	63 (∼31,350 c/s)
`int16`	8	∼1,774,000 c/s	∼4,271,300 c/s
MaxPoly `int16`	8	30 (∼49,050 c/s)	67 (∼59,950 c/s)
`int32`	8	∼986,100 c/s	∼2,122,400 c/s
MaxPoly `int32`	8	17 (∼43,200 c/s)	30 (∼28,300 c/s)
`float32`	8	∼196,000 c/s	∼3,109,050 c/s
MaxPoly `float32`	8	3 (∼70,150 c/s)	60 (∼27,900 c/s)

Table 3. Number of cycles per second and maximum number of voices of polyphony for different data types, block sizes and Teensy boards based on the FAUST program presented in Code Listing 8.

been done towards bare-metal implementations on more advanced platforms. Indeed, while we don't think there's more work to be done on the Teensy side, we'd like to implement a series of tools similar to the ones presented in this paper targeting the Raspberry Pi (RPI). [16] The RPI 3 A+ [17] only costs 25USD and is based on a Broadcom BCM2837B0 Cortex-A53 with 4 1.4GHz cores and 512MB of RAM. Beside the fact it offers much more processing power than the Teensy, the Cortex-A53 microarchitecture provides support for Neon, [18] which should allow further optimizations for floating points operations.

While the PI is not a microcontroller like the Teensy and therefore doesn't have any analog inputs for sensors, etc. it can be easily upgraded with an Analog to Digital Converter (ADC) such as an MCP3008 [19] which costs less than 4USD. Similarly, the built-in audio codec of the PI is notorious to be low quality. The Fe-Pi Audio Z V2 [20] is a sister board for the PI using the same SGTL5000 Audio Codec as the Teensy Audio Shield, and its cost is inferior to 12 USD. Hence, the total cost of this set-up is similar to the one of a Teensy 3.6 upgraded with an Audio Shield (∼45USD) but provides much more processing power for potential bare-metal implementations.

7. CONCLUSIONS

Programming the Teensy for custom real-time audio signal processing applications is out of reach to most people in the DIY community. The tools presented in this paper provide a comprehensive way to carry out this type of task at a higher level using the FAUST programming language. Programmers benefit from hundreds of existing DSP functions as well as complex functionalities such as handling polyphony.

More generally, thanks to its high processing power (at least considering that it's a microcontroller) and its FPU, the Teensy 3.6, when combined with its audio shield provides a cheap platform (>50USD) for low-latency real-time audio signal processing involving external sensors control. The lack of operating system allows for the use of low block sizes (eight samples) at a minimal computational cost, and for an extremely fast boot time (less than one second).

[16] `https://www.raspberrypi.org/`

[17] `https://www.raspberrypi.org/products/raspberry-pi-3-model-a-plus/`

[18] `https://developer.arm.com/technologies/neon`

[19] 8-Channel 10-Bit ADC with SPI interface.

[20] `https://fe-pi.com/products/fe-pi-audio-z-v2`

8. REFERENCES

[1] S. Jordà, "Digital lutherie crafting musical computers for new musics' performance and improvisation," Ph.D. dissertation, Universitat Pompeu Fabra, Spain, 2005.

[2] E. Berdahl and W. Ju, "Satellite ccrma: A musical interaction and sound synthesis platform," in *Proceedings of the New Interfaces for Musical Expression (NIME'11)*, Oslo, Norway, June 2011.

[3] A. McPherson, "Bela: An embedded platform for low-latency feedback control of sound," *The Journal of the Acoustical Society of America*, vol. 141, no. 5, pp. 3618–3618, 2017.

[4] E. Motuk, R. Woods, S. Bilbao, and J. McAllister, "Design methodology for real-time fpga-based sound synthesis," *IEEE Transactions on signal processing*, vol. 55, no. 12, pp. 5833–5845, 2007.

[5] F. Pfeifle and R. Bader, "Real-time finite-difference method physical modeling of musical instruments using field-programmable gate array hardware," *Journal of the Audio Engineering Society*, vol. 63, no. 12, pp. 1001–1016, 2016.

[6] Y. Orlarey, S. Letz, and D. Fober, *New Computational Paradigms for Computer Music*. Paris, France: Delatour, 2009, ch. "Faust: an Efficient Functional Approach to DSP Programming".

[7] R. Michon, J. Smith, C. Chafe, S. Letz, and Y. Orlarey, "faust2api: a comprehensive api generator for android

and ios," in *Proceedings of the Linux Audio Conference (LAC-17)*, Saint-Etienne, France, May 2017, submitted for review.

[8] V. Valimaki, J. D. Parker, L. Savioja, J. O. Smith, and J. S. Abel, "Fifty years of artificial reverberation," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 5, pp. 1421–1448, 2012.