A FAUST ARCHITECTURE FOR THE ESP32 MICROCONTROLLER

Romain Michon,[a,b] **Daniel Overholt,**[c] **Stéphane Letz,**[a] **Yann Orlarey,**[a] **Dominique Fober,**[a] **and Catinca Dumitrascu**[a]
[a] GRAME – Centre National de Création Musicale, Lyon, France
[b] Center for Computer Research in Music and Acoustics, Stanford University, USA
[c] Aalborg University Copenhagen, Denmark
michon@grame.fr

ABSTRACT

This paper introduces `faust2esp32`, a tool to generate digital signal processing engines for the ESP32 microcontroller family. It can target both the C++ and the Arduino ESP32 programming environment and it supports a wide range of audio codecs, making it compatible with most ESP32-based prototyping boards. After demonstrating how to use `faust2esp32` and providing technical details about its implementation, we evaluate its performances and we present the FAUST Gramophone which is a programmable instrument taking advantage of this technology. Finally, future plans around embedded systems for real-time audio processing and FAUST are presented.

1. INTRODUCTION

For a long time, embedded Linux systems such as the Raspberry Pi [1] (RPI) or the Beagle Bone [2] were the only embedded platforms for real-time audio Digital Signal Processing (DSP) accessible to members of the music technology community without a programming background [1]. The Operating System (OS) running on this kind of platform makes the on-board implementation of audio processing algorithms through high-level graphical patching environments such as PureData [2] possible. Specialized Linux distributions targeting this type of application such as *Satellite CCRMA* [3] were developed in that context. However, the ease of use provided by the OS – Linux, for instance – comes at the cost of simplicity and efficiency, both in terms of hardware and software. Additionally, since embedded Linux systems are typically used by the music technology community to create musical instruments, art installations, etc., [4] the lack of a proper audio input/output and of Analog to Digital Converters (ADCs) for sensors are often compensated by the use of external hardware (e.g., USB audio interface, USB microcontroller like the Arduino, etc.), significantly increasing the cost and complexity of such setups.

These issues were addressed by the BELA [3] [5, 6] for a subset of the Beagle Bone boards, or the Elk [4] for the RPI. These systems take the form of sister boards/hats providing high-quality audio inputs and outputs (audio codec) as well as sensor analog inputs from a single standpoint. They also run audio DSP tasks outside of the OS, which translates into a faster access to the audio inputs and outputs for a reduced audio latency. The overall design of such systems remains quite complex though for what they're actually used for and their price is relatively high (>$150).

On the other hand, microcontrollers offer a promising lightweight, efficient, and cheap platform for real-time audio DSP applications. While microcontrollers were primarily designed for data/sensor acquisition and basic processing, new generations such as the ARM Cortex-M [5] provide extended computational power and memory. This combines with the fact that an increasing number of microcontrollers host a Floating Point Unit (FPU), greatly simplifying the implementation of DSP algorithms. The Teensy microcontroller family [6] played a pioneering role in this field by distributing an Audio Shield [7] (which is essentially a breakout board for an audio codec) and providing an "audio library [8] " for high-level audio DSP programming. The Teensy 3.6 and 4.0 (180MHz and 600MHz, respectively), when combined with the Audio Shield provide a powerful platform for real-time audio applications (70 sine waves can be ran in real-time on the 3.6 and up to 600 on the 4.0) [7].

While tools such as the Teensy Audio Library or libpd [8] allow for the programming of DSP algorithms at a high-level, they are limited to their built-in/pre-implemented DSP objects (e.g., filters, oscillators, etc.) and custom elements must be written in C++. In 2019, we introduced `faust2teensy` [7] which is a command-line tool that can be used to generate custom objects for the Teensy audio library using the FAUST programming language [9]. FAUST [9] is a functional Domain Specific Language (DSL) for real-time audio DSP that can target a wide range of

[1] `https://www.raspberrypi.org/` All URLs presented in this paper were verified on April 24, 2020.
[2] `https://beagleboard.org/bone`

[3] `https://bela.io/`
[4] `https://elk.audio/`
[5] `https://developer.arm.com/ip-products/processors/cortex-m`
[6] `https://www.pjrc.com/teensy/`
[7] `https://www.pjrc.com/store/teensy3_audio.html`
[8] `https://www.pjrc.com/teensy/td_libs_Audio.html`
[9] `https://faust.grame.fr`

platforms (e.g., Linux, macOS, Windows, Android, iOS, embedded systems, etc.) and standards (e.g., audio plug-ins, standalone applications, externals, etc.). While FAUST can easily add various features to generated objects such as polyphony for synthesizers, MIDI and OSC (Open Sound Control) support, etc., `faust2teensy` only provides basic functionalities because of the limited amount of memory available on the Teensy – i.e.,:

- control of the parameters of a FAUST DSP object (e.g., declared using UI elements such as `hslider`, `nentry`, etc.),
- stereo audio input and output.

FAUST-generated objects for the Teensy Audio Library can be combined with built-in objects, playing to the strengths of both approaches. Since its release, `faust2teensy` has been used for teaching (e.g., *Music 250a – Physical Interaction Design for Music*, [10] the *Embedded DSP With Faust Workshop* [11] at Stanford University, *Low Latency Embedded DSP for New Musical Instruments* at Aalborg University Copenhagen, etc.) and as a basis for many projects.

While the Teensy gained a special status in the makers community in recent years because of its power, reliability, and affordability, the ESP32 microcontroller [12] has been increasingly used as well, offering a solid alternative to the Teensy for real-time audio processing applications.

In this paper, we introduce `faust2esp32`, a tool to generate DSP engines for the ESP32 microcontroller with FAUST. After giving an overview of this type of board, we demonstrate how `faust2esp32` can be used to generate DSP objects and we provide technical details about the implementation of this system. We then evaluate its performance and present the FAUST Gramophone which is a programmable instrument based on the platform. Finally, we give future directions for this type of work.

2. THE ESP32

The ESP32 is a series of low-cost, low-power embedded systems with integrated WiFi, Bluetooth, and microcontroller distributed by Espressif Systems. It is based on a 32 bits dual core Tensilica Xtensa LX6 microprocessor operating at 240MHz. It provides 520 KiB of internal RAM, but almost all ESP32-based boards host an external SRAM module expanding its memory. The main asset of the ESP32 is its cost since basic ESP32-based boards can be found for less than $3.

The ESP32 is used at the heart of a wide range of more specialized boards targeting specific applications such as real-time audio processing (see Figure 1). In particular, a series of development boards to prototype smart speakers (e.g., Google Home, Amazon Alexa, etc.) have been available for about a year now and are particularly well suited

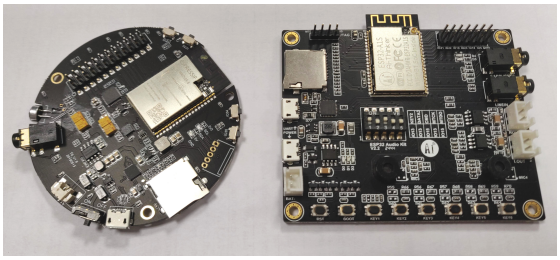

Figure 1. ESP32-based audio Processing boards (the TTGO TAudio on the left and the AI-Thinker Audio Dev Kit on the right).

Board	Features
LilyGO TTGO-TAudio	4MB PSRAM, motion sensors, audio in/out (codec WM8978), battery management, speaker amplifier
AI-Thinker ESP32-A1S	8MB PSRAM, audio in/out (codec AC101), speaker amplifier
Espressif ESP32-LyraT	4MB PSRAM, audio in/out (codec ES8388), battery management, speaker amplifier
Espressif ESP32-LyraT Mini	8MB PSRAM, audio in (codec ES8311), battery management, speaker amplifier

Table 1. Examples of ESP32-based boards for real-time audio signal processing.

for embedded musical instrument design by providing audio inputs and outputs as well as additional RAM required to implement algorithms with a large memory footprints such as echos, reverbs, etc. Table 1 gives a few examples of such boards and of their corresponding features.

While ESP32 boards can be used bare-metal (without an OS), the most convenient and effective way to program them is FreeRTOS [13] which is a low-level real-time OS system kernel for embedded devices. It is accessible through a C++/MakeFile environment built on top of a Python tool-chain to communicate with the board. The ESP32 can also be programmed through the Arduino environment/IDE. FreeRTOS' scheduler/task manager is very convenient in the context of real-time audio DSP applications.

Espressif also recently released a series of ESP32-based boards hosting a Digital Signal Processors (DSP) such as the ESP32-LyraTD-DSPG, the ESP32-LyraTD-SYNA, and the ESP32-LyraTD-MSC [14] which are not yet supported by FAUST (see §6).

[10] `https://ccrma.stanford.edu/courses/250a-winter-2020/`
[11] `https://ccrma.stanford.edu/workshops/faust-embedded-19/`
[12] `https://www.espressif.com/en/products/hardware/esp32/overview`
[13] `https://www.freertos.org/`
[14] `https://www.espressif.com/en/products/hardware/development-boards`

3. USING FAUST2ESP32

3.1 Generating DSP Engines

`faust2esp32` is a command line tool that can be used to generate ready-to-use C++ DSP engines from a FAUST program. Given a simple FAUST program such as:

```
import("stdfaust.lib");
f = nentry("freq",440,50,2000,0.01);
g = nentry("gain",1,0,1,0.01);
t = button("gate");
process = os.sawtooth(f)*g*t;
```

which just implements a band-limited sawtooth oscillator whose frequency (`freq`) and gain (`gain`) can be controlled. Additionally, the `gate` parameter allows us here to turn it on and off.

Running:

```
faust2esp32 -lib FaustObject.dsp
```

will produce a package containing two C++ files: `FaustObject.cpp` and `FaustObject.h` which can be imported into a C++ or an Arduino ESP32 project. In both cases, `FaustObject.h` should be included at the beginning of the main program:

```
#include "FaustObject.h"
```

The corresponding FAUST DSP object can be instantiated with:

```
FaustObject* faustObject = new
   FaustObject(48000,8);
```

where 48000 is the sampling rate and 8 the block size. Any value can be chosen for these parameters as the long as the hardware can handle it. Note that block size has very little impact on hardware performances (see Table 2).

Audio computing can be started by calling the `start` method:

```
faustObject->start();
```

Audio computing is taking place in its own high-priority thread (FreeRTOS task). Audio samples are retrieved from and transmitted to the audio codec of the board using I2S. This implies that the audio codec must be configured before using a driver.

`faust2esp32` comes with a series of drivers supporting some audio codecs such as the Wolfson WM8978 or the X-Powers AC101. These can be integrated to the generated package using the `-wm8978` and the `-ac101` options (respectively) when calling `faust2esp32`. Since functionalities differ between audio codecs, they are not presented here and we advise you to refer to their corresponding documentation.

The parameters of the FAUST DSP object (i.e., `freq`, `gain`, and `gate` here) can be updated at any time by calling the `setParamValue()` method which takes the name of the parameter and its corresponding value as arguments. E.g.:

```
faustObject->
    setParamValue("freq",1000);
```

will set the frequency of the generated sawtooth wave to 1000 Hz.

This procedure works both in the Arduino and native ESP32 C++ environments.

3.2 MIDI Support

MIDI support can be easily added to the generated DSP engine by using the `-midi` option when calling `faust2esp32`. MIDI events are received directly from the corresponding UART (Universal Asynchronous Receiver-Transmitter) serial interface on the ESP32 (see §4). Standard FAUST MIDI metadata [15] can be used as well to configure the MIDI behavior of the DSP engine.

3.3 Polyphony

FAUST synthesizers declaring the `freq`, `gain`, and `gate` parameters (such as the one presented at the beginning of §3.1) can be automatically turned into polyphonic synths by using the `-nvoices` option when calling `faust2esp32`. Hence,

```
faust2esp32 -lib -midi -nvoices 12
   FaustObject.dsp
```

will produce a polyphonic synth with MIDI support with a maximum number of twelve voices of polyphony. Standard FAUST polyphony metadata [16] can be used to further configure the behavior of this feature.

4. IMPLEMENTATION

Unlike the Teensy, the ESP32 doesn't provide a high-level audio library, so we had to implement almost everything from scratch here.

Audio DSP is running in its own high-priority FreeRTOS task (see Figure 2) on the first core. Control-rate computations (like MIDI, sensor, etc.) happen in a separate task on the second core of the ESP32.

External events are handled in an asynchronous manner, hence their timestamp is not taken in account. The generated C++ code hosts the DSP `compute` method which samples the values of all controllers at the beginning of the code, and use the same values during the entire audio block. Thus, only the most recent values are used.

This simple model may have to be improved with more sophisticated queue-based event handling code, if we want to ensure all events are taken in account (which is mandatory in some cases like MIDI clock-based synchronization, for instance).

External PSRAM can be integrated to the heap to make it available to the audio DSP object. This can be configured from the ESP32 project settings (typically accessible by running `make menuconfig`). Audio samples are received and sent to the audio codec using I2S through the corresponding ESP32 API. [17] The I2S configuration has to

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

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

[17] https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/i2s.html

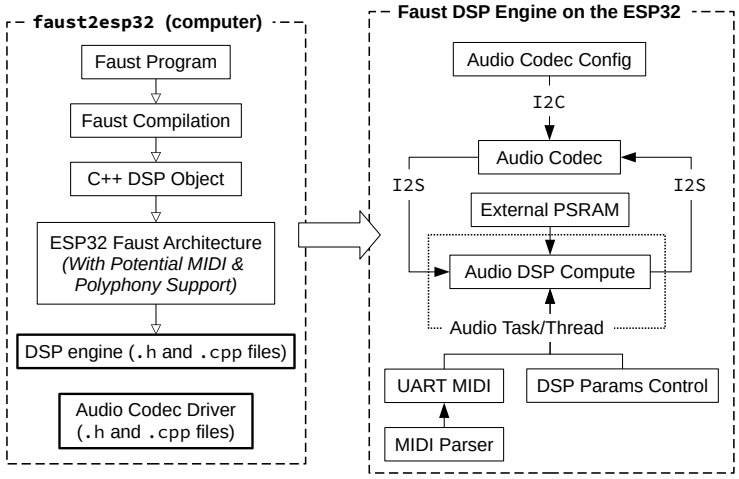

Figure 2. Overview of the implementation of `faust2esp32`. Thick continuous frames on the left represent objects generated by `faust2esp32` on the user's computer. Empty arrows on the left link the various steps in generating the DSP engine. Full arrows on the right represent streams computed in real-time.

match that of the audio codec. The audio codec is configured through I2C using the corresponding driver provided as part of the package generated by `faust2esp32`. Only a few audio codecs are currently supported (see §3).

MIDI events are received through the corresponding UART interface and are parsed using `jdksmidi`, [18] then transmitted to the Faust MIDI handling code with a `esp32_midi` glue class.

The general implementation of `faust2esp32` is similar to most other FAUST architectures and is based on a script (i.e., BASH) taking care of assembling the various constituent elements of the generated DSP engine. `faust2esp32` is now fully integrated to the main FAUST distribution. [19]

5. EVALUATION

Running DSP code on microcontrollers presents two main advantages compared to using a more general board and operating system:

- very small buffer size (i.e., 8 samples) can be used, minimizing latency;
- the OS is minimal and is usually more real-time robust, diminishing the risk for audio glitches to happen.

5.1 Performance

The ESP32 offers plenty of computational power to run most "standard" DSP algorithms. Table 2 presents a comparison of the performance in terms of processing power of the ESP32 with other microcontrollers supported by FAUST. The ESP32 used for these tests was a WROVER-B on a LyliGO TTGO-TAudio board. All tests were carried out at a sampling rate of 48KHz with the following FAUST program implementing a series of sine wave oscillators in parallel:

Block Size	ESP32	Teensy 3.6	Teensy 4.0
8	143	75	611
128	145	73	620
256	145	74	621

Table 2. Maximum number of FAUST sine waves ran in parallel on various boards for different block sizes.

```
import("stdfaust.lib");
N = 600;
tablesize = 16384;
time = (+(1)~_) - 1;
sinwaveform(tablesize) =
    time*(2.0*ma.PI)/tablesize : sin;
decimal(n) = n - floor(n);
phasor(tablesize,freq) =
    freq/ma.SR : (+ : decimal)~_ :
    *(tablesize) : int;
osc(freq) =
    rdtable(tablesize,
    sinwaveform(tablesize),
    phasor(tablesize,freq));
process = par(i,N,osc(50+i));
```

Each oscillator is using a 2^{14} samples-long table read directly from memory (the `sin` function is only called during initialization). The table is read using a phasor which corresponds to one multiply, one add, one subtract, and one call of the math `floor` function.

As expected, the ESP32 clearly outperforms the Teensy

[18] https://github.com/jdkoftinoff/jdksmidi
[19] https://github.com/grame-cncm/faust

Figure 3. The FAUST Gramophone.

3.6, but it is much less powerful than the Teensy 4.0 (which runs at 600MHz).

Since most ESP32-based boards host an external PSRAM module, running DSP algorithms with a large memory footprints is not an issue in most cases on the ESP32 when it can be problematic on most Teensy boards. For example, while `dm.zita_light` [20] (which is a feedback delay network reverb that can be found in the FAUST libraries) can't be ran on any Teensy board without making some modifications to the algorithm, the LyliGO TTGO-TAudio can "digest" it without any issue.

5.2 Use Case: the Gramophone

The Gramophone (see Figure 3) is a programmable musical instrument designed as part of the Amstramgrame project, [21] which is currently in its pilot phase. Amstramgrame aims at teaching physics and math through programming exercises with the Gramophone. The Amstramgrame website hosts pedagogical content as well as code examples that can be run directly in the FAUST Web IDE. FAUST programs prototyped in the web can then be exported to the Gramophone directly from the Amstramgrame website.

The Gramophone is based on a LilyGO TTGO-TAudio board which hosts a single 9-DoF sensor (comprising an accelerometer, a gyroscope, and a magnetometer), a lithium battery manager, an audio codec, etc. Additional sensors (e.g., photoresistor, potentiometers, buttons, etc.) are mounted on the body of the Gramophone. The Gramophone can be powered by a lithium battery directly connected to the board. The speaker of the instrument is directly connected to the built-in amplifier of the TTGO-TAudio's audio codec.

The Gramophone can be entirely programmed in FAUST without writing a single line of C++ or Arduino code. The `-gramophone` option can be used for that when calling `faust2esp32`. The built-in motion sensors (a full 9-axis IMU, Inertial Measurement Unit) can be mapped using the standard FAUST metadata system, [22] and transmitted to the FAUST DSP using the `Esp32SensorUI` class. Gramophone-specific FAUST metadata were created as well to access the instrument's built-in sensors (i.e., potentiometers, buttons, etc.) and transmitted to the FAUST DSP using the `Esp32ControlUI` class.

Multiple FAUST programs can be installed in the Gramophone using the `-multi` option when calling `faust2esp32`, by following a specific coding convention in the DSP source code to describe them all in a single file. FAUST programs can then be selected using an encoder mounted on the body of the instrument. To save memory, only a single DSP is active at one point, so the old one is deleted before a new one is instantiated.

For less than $40 (all parts included), the ESP32 through the TTGO-TAudio board allowed us to make a fully embedded programmable instrument with more than seven hours of battery life that can run most existing FAUST programs. We believe that this would not have been possible with any other board/type of system.

6. FUTURE DIRECTIONS

6.1 Around the ESP32

The ESP32 FAUST support could be extended in many ways. In particular, since the ESP32 provides built-in WiFi support, we hope to be able to improve the existing architecture by enabling OSC support.

Finally, we'd like to develop architectures for the ESP32 DSP boards that have been recently introduced by Espressif such as the LyraTD-DSPG, the LyraTD-SYNA, and the LyraTD-MSC. Since they are based on floating-point DSPs this should be pretty straight forward.

6.2 Towards More Advanced FAUST Architectures for Embedded Processing

We recently started investigating the idea to use FAUST to program more advanced systems such as GPUs and FPGAs. The latter is particularly challenging because it is usually programmed at an extremely low-level using hardware description languages such as VHDL or Verilog. While we managed to program Xilinx FPGAs from FAUST through C++ using HLS (High Level Synthesis), [23] this solution remains relatively heavy and slow and we hope to be able to generated VHDL code directly from FAUST for real-time DSP applications.

In parallel to that, we've been working on a RPI bare-metal FAUST architecture for low-latency applications. While we have a working prototype, many issues remain to be solved so we'd like to keep working on that in the future.

7. CONCLUSIONS

We believe that FAUST greatly simplifies the programming of embedded systems for real-time audio DSP applications by providing a high-level environment with hundreds of pre-implemented functions/algorithms. Systems such as

[20] `https://faust.grame.fr/doc/libraries#dm.zita_light`

[21] `https://www.amstramgrame.fr`

[22] `https://faust.grame.fr/doc/manual#sensors-control-metadatas`

[23] `https://faust.grame.fr/syfala`

the EP32 are currently revolutionizing the field of music technology by providing a comprehensive platform for musical instrument design/audio processing at a very low cost. Moreover, their simplified architecture (bare-metal or close to bare-metal) make them more stable and reliable than OS-based systems. We hope that in a near future, Domain-Specific Languages such as FAUST will make all types of embedded/specialized systems such as FPGAs accessible to makers, hobbyist, and members of the music technology community.

8. REFERENCES

[1] E. Berdahl, "How to make embedded acoustic instruments." in *Proceedings of the New Interfaces for Musical Expression Conference (NIME-14)*, London, UK, 2014.

[2] M. Puckette, "Pure data: another integrated computer music environment," in *Proceedings of the Second Intercollege Computer Music Concerts*, Tachikawa, Japan, 1996.

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

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

[5] G. Moro, A. Bin, R. H. Jack, C. Heinrichs, A. P. McPherson *et al.*, "Making high-performance embedded instruments with bela and pure data," in *Proceedings of the International Conference of Live Interfaces*, Brighton, UK, 2016.

[6] 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.

[7] R. Michon, Y. Orlarey, S. Letz, and D. Fober, "Real time audio digital signal processing with Faust and the Teensy," in *Proceedings of the Sound and Music Computing Conference (SMC-19)*, Malaga, Spain, 2019.

[8] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner, "Embedding PureData with libpd," in *Proceedings of the Pure Data Convention*, Weinmar, Germany, 2011.

[9] 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".