Embedded Real-Time Audio Signal Processing With Faust

Romain Michon, Yann Orlarey, Stéphane Letz, Dominique Fober, Dirk Roosenburg

Proceedings of the 2nd International Faust Conference (IFC-20), Maison des Sciences de l'Homme Paris Nord, France, December 1-2, 2020

EMBEDDED REAL-TIME AUDIO SIGNAL PROCESSING WITH FAUST

Romain Michon,[a,b] Yann Orlarey,[a] Stéphane Letz,[a] Dominique Fober,[a] and Dirk Roosenburg[d,e]

[a]GRAME – Centre National de Création Musicale, Lyon, France
[b]Center for Computer Research in Music and Acoustics, Stanford University, USA
[d]TIMARA, Oberlin Conservatory of Music, USA
[e]Department of Physics, Oberlin College, USA
michon@grame.fr

ABSTRACT

FAUST has been targeting an increasing number of embedded platforms for real-time audio signal processing applications in recent years. It can now be used to program microcontrollers, mobile platforms, embedded Linux systems, Digital Signal Processors (DSPs), and more. This paper gives an overview of existing targets, presents ongoing research, and gives potential avenues for future developments in this field.

1. INTRODUCTION

The computer music and music technology communities have been greatly impacted by the advent of the makers/DIY[1] culture [1] in recent years. In this context, embedded audio systems have been increasingly used to create musical instruments, sound effect processors, art installations, or even to prototype and commercialize products in the music technology industry (e.g., MOD Duo,[2] etc.). In parallel of that, mobile devices (e.g., smartphones, tablets, etc.) – which in some respects can also be considered as embedded systems – have been used for similar types of applications [2].

Embedded Linux Systems (ELS) such as the Raspberry Pi[3] (RPI) or the Beagle Board,[4] etc., were the only available solutions accessible to the aforementioned communities for a long time. Linux allowed them to deal with embedded audio processing in the same way than on a desktop computer through the use of high-level tools like PureData [3], ChucK [4], SuperCollider [5], etc. Specialized Linux distributions with specific configurations for real-time audio processing such as Satellite CCRMA [6] were also developed in this context. However, the ease of use offered by operating systems comes at the cost of lightness/simplicity (both in terms of hardware and software), audio latency, efficiency, etc. Moreover, most of these boards were often not designed with real-time audio applications in mind and lack a proper audio input/output, implying the use of an external audio interface (e.g., USB, hat/sister board with an I2S[5] audio codec, etc.).

More recently, these issues were addressed by specialized hardware/software solutions such as the BELA [7] (first released in 2014) and the Elk.[6] In both cases, dedicated hardware (hat/sister board) is added to "standard" ELS (the RPI for the Elk and the BeagleBone Black for the BELA) to improve their performances for real-time audio processing applications (i.e., audio quality, multi-channel audio, etc.). These products work in conjunction with specialized Linux distributions where audio processing tasks are carried out outside of the operating system, allowing for the use of significantly smaller buffer sizes and therefore lower audio latency. While this type of solution offers a good compromise in terms of ease of use and performances, it remains expensive ($239 for the Elk and $200 for the BELA) and relatively heavy for the kind of application it usually targets.

Even more recently, new generations of microcontrollers such as the ARM Cortex-M family[7] or the ESP32[8] have been offering extended computational power (e.g., 600MHz for the Cortex-M7), suitable for real-time audio processing applications. Many of these microcontrollers also host a Floating Point Unit (FPU), greatly simplifying the implementation of DSP algorithms. Some microcontroller-based boards such as the LilyGO TTGO TAudio[9] provide a comprehensive solution for real-time audio processing applications by hosting an audio codec and an external RAM module. Other brands/boards like the Teensy[10] distribute an "audio shield," which is essentially a breakout board for an audio codec with the correct form factor to be mounted on the main board. Programming microcontrollers for real-time audio signal processing applications is more complex than ELS because the use of C++ is always required at the end of the chain.

Specialized chips such as Digital Signal Processors (DSP) also became more accessible in recent years by allowing the use of floating-points and C++ for their programming while assembly and fixed-points used to be the norm. The Analog Devices SHARC processor family[11] has been dominating the floating-point DSP market for many years now and can be found in a wide range of commercial synthesizers (e.g., Korg, Roland, etc.). Evaluation boards for this type of chip such as the SHARC audio module[12] now also try to target the makers/hobbyist community by making their toolchain higher-level and more accessible.

Beside ELSs which provide a comprehensive set of standard high-level audio programming tools, all the aforementioned platforms remain relatively hard to program and inaccessible to non-specialized communities. While some tools such as libpd [8] and Embedded ChucK [9] address this issue, none of them provide

[1]Do It Yourself
[2]https://www.moddevices.com/ – All URLs in this paper were verified on Sep. 3, 2020.
[3]https://www.raspberrypi.org/
[4]https://beagleboard.org/
[5]*Inter-IC Sound*
[6]https://elk.audio/
[7]https://developer.arm.com/ip-products/processors/cortex-m
[8]https://www.espressif.com/en/products/hardware/esp32/overview
[9]https://github.com/LilyGO/TTGO-TAudio
[10]https://www.pjrc.com/teensy/
[11]https://www.analog.com/en/products/processors-dsp/dsp/sharc.html
[12]https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/sharc-audio-module.html

a solution as comprehensive as FAUST [10]. Indeed, FAUST has been active on all these fronts for the past ten years, allowing for the programming of most of these platforms from a single standpoint. This paper provides an overview of the work that has been done around FAUST in the field of embedded systems for real-time audio processing. It also presents ongoing research in this field as well as avenues for potential future developments.

2. EMBEDDED LINUX SYSTEMS

Embedded Linux systems offer more or less the same type of features as desktop Linux distributions. Thus, real-time audio signal processing tasks can be carried out in many different ways on this type platform. For instance, standalone applications can be made and connected directly to various audio engines such as Alsa, Jack, etc. Open-source computer music environments like PureData [3], ChucK [4], SuperCollider [5], etc. are also available. Various audio plugin hosts (e.g., Ardour, Audacity, etc.) can be used to execute audio plugins, etc. Finally, while not necessarily the best/most effective option in that context, web apps can potentially be used as well.

FAUST is extremely well supported on Linux and can target most of the existing standards and environments available on this platform through a wide range of architectures[13] (see Table 1). While most Linux-compatible `faust2...` compilation scripts can be used on ELS (with some adjustments in the C++ compilation flags), platform-specific scripts such as `faust2rpialsaconsole` or `faust2rpinetjackconsole` which target the Raspberry Pi (see Figure 1) were implemented.

Even though FAUST can generate applications with a user interface (e.g., Qt, GTK, etc.), these don't necessarily make sense in the context of embedded systems where screens are not always used/available. Hence, functionalities are usually accessed through SSH, making command line applications more appropriate in that context.

Designing musical instruments around ELS also implies the use of an external microcontroller (either connected through USB or I2C[14]) since the CPUs used on these platforms don't have built-in ADCs.[15] External controllers (i.e., MIDI controllers) or communication protocols such as OSC[16] can be used as well with FAUST, but it defeats the purpose in this context by externalizing some parts of the instrument.

3. EMBEDDED LINUX SYSTEMS WITH SPECIAL HARDWARE

A series of projects/commercial products such as the BELA and the Elk (see §1) have been trying to provide dedicated Linux-based platforms for musical instrument design/real-time audio signal processing applications. Both are based on existing ELS (the BeagleBone Black and the RPI, respectively) and involve the use of a sister board/hat and of a specialized Linux distribution in order to provide the following features:

[13] Architectures in the FAUST vocabulary refer to wrappers allowing to turn a FAUST program into a specific object such as a standalone desktop program, an audio plugin, a smartphone app, an audio engine for a specific platform, etc.

[14] *Inter-Integrated Circuit*

[15] Analog to Digital Converter

[16] Open Sound Control

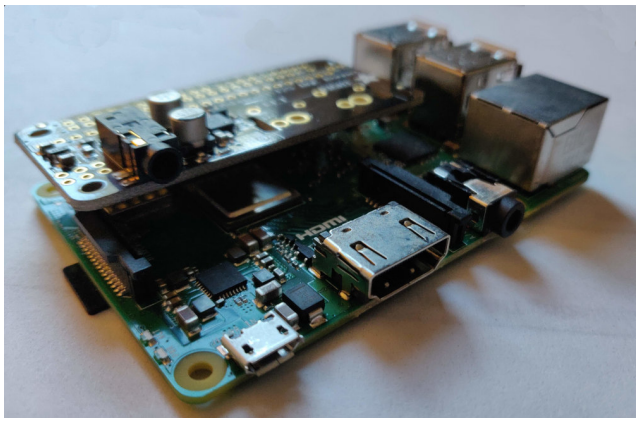

Figure 1: *The Raspberry Pi 3B equipped with an I2S audio hat providing a stereo PCM audio output.*

- high priority to audio processing bypassing the operating system through the use of Xenomai[17] [11] for reduced audio latency,
- multichannel (more than stereo) audio codec,
- built-in microcontroller for analog sensor inputs,
- etc.

3.1. Using FAUST on the BELA

FAUST can be used for audio programming on the BELA (see Figure 2) with `faust2bela`.[18] This command line tool takes a FAUST program as its main argument and executes it on this board by running:

```
faust2bela -tobela faustProgram.dsp
```

MIDI and polyphony support can be enabled by using the `-midi` and `-nvoices` options, respectively. The parameters of the FAUST program can also be accessed through a web interface running on a dedicated built-in web server by using the `-gui` option.

3.2. Using FAUST on the Elk

While Elk (see Figure 3) is not officially supported by FAUST through an architecture file, FAUST objects can be compiled as VST plugins compatible with the Elk. `faust2faustvst`[19] can be used for that purpose. Elk also provides a Cmake file to carry out this task.[20]

4. MOBILE PLATFORMS

Mobile devices can be considered as embedded systems as well. In particular, musical instruments can be easily constructed around

[17] `https://gitlab.denx.de/Xenomai/xenomai/-/wikis/home`

[18] `https://github.com/BelaPlatform/Bela/wiki/Compiling-Faust-code-for-Bela`

[19] `https://bitbucket.org/agraef/faust-vst/src/master/`

[20] `https://github.com/elk-audio/faust-vst-template`

Architecture	Description
Standalone With User Interface	
`faust2alqt`	Standalone application with an Alsa audio engine and a Qt UI
`faust2alsa`	Standalone application with an Alsa audio engine and a GTK UI
`faust2jack`	Standalone application with a Jack audio engine and a GTK UI
`faust2jaqt`	Standalone application with a Jack audio engine and a Qt UI
`faust2netjackqt`	Standalone application with a NetJack audio engine and a Qt UI
Standalone Without User Interface	
`faust2alsaconsole`	Command line application with an Alsa audio engine
`faust2jackconsole`	Command line application with a Jack audio engine
`faust2netjackconsole`	Command line application with NetJack audio engine
Computer Music Environment External	
`faust2ck`	ChucK Chugin
`faust2csound`	CSound Opcode
`faust2puredata`	PureData External
`faust2sc`	SuperCollider External
Audio Plug-In	
`faust2juce`	Can be used to generate VST plugins
`faust2ladspa`	LADSPA audio plugin
`faust2lv2`	LV2 audio plugin
Web	
`faust2wasm`	WebAudio wasm module
`faust2webaudiowasm`	WebAudio wasm HTML pages
RPI-Specific	
`faust2rpialsaconsole`	Command line application with an Alsa audio engine
`faust2rpinetjackconsole`	Command line application with a NetJack audio engine

Table 1: *Linux-compatible* FAUST *architectures.*

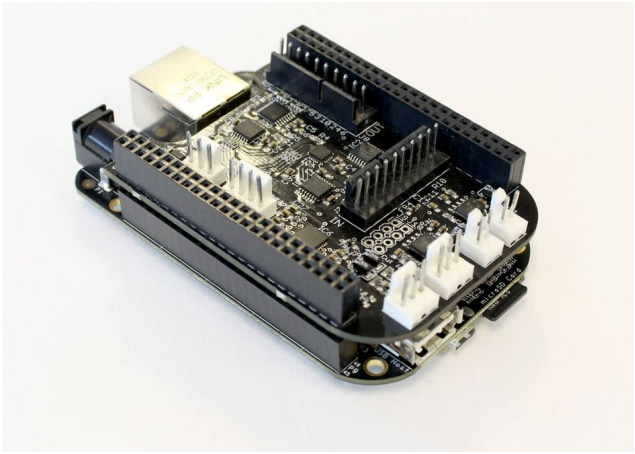

Figure 2: *The BELA.*

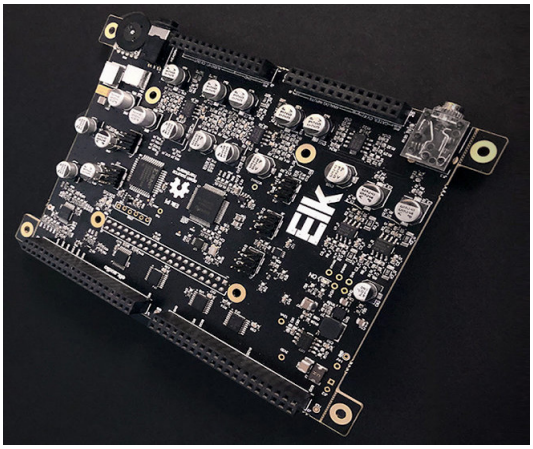

Figure 3: *The Elk.*

these platforms through the use of microcontrollers connected to them via MIDI (USB or Bluetooth) [12]. FAUST played a pioneering role in the field of mobile music [13] by allowing for the programming of iOS and Android devices for real-time audio signal processing applications.

4.1. faust2ios

`faust2ios` is a tool to convert FAUST programs into ready-to-use iOS applications whose user interface is based on the UI description provided in the FAUST code (just like most FAUST architectures). UIs are therefore typically made out of sliders, knobs, buttons, groups, etc.

Figure 4 presents a screenshot of *sfCapture*[21] which is an app made with `faust2ios` as part of the SmartFaust project [13].

`faust2ios` works as a command line tool taking a FAUST program as its main argument and producing either a ready-to-install iOS app or the Xcode project corresponding to this app in return. For example, running the following command in a termi-

[21] `https://itunes.apple.com/us/app/sfcapture/id799532659?mt=8`

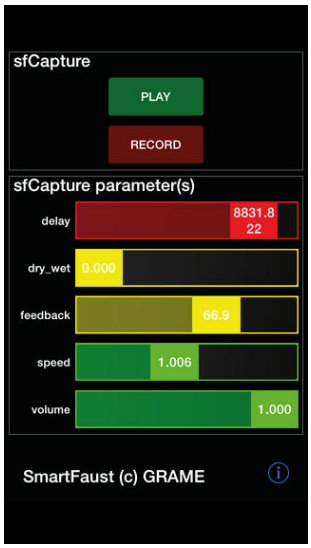

Figure 4: *Screen-shot of sfCapture, an app made with* `faust2ios`.

nal:

```
faust2ios faustProgram.dsp
```

will produce an iOS app corresponding to the FAUST program implemented in `faustProgram.dsp`.

Various features can be added to the generated app such as MIDI, OSC, and polyphony support simply by using specific flags (options) when running `faust2ios`. Regular FAUST options are also available to generate parallelized DSP[22] code, change sample resolution, etc. Any parameter of a FAUST program can be assigned to a specific axis of a built-in motion sensor (i.e., accelerometer, gyroscope, etc.) of the smartphone simply by using metadata. Complex non-linear mappings can be implemented using this mechanism.[23]

4.2. faust2android

`faust2android` [14] is the equivalent of `faust2ios` for the Android platform and can also be used as a command line tool:

```
faust2android faustProgram.dsp
```

will generate a read-to-use Android application from `faustProgram.dsp` (see Figure 5).

Unlike `faust2ios`, standard FAUST user interfaces can be replaced by advanced interfaces more usable in a musical context such as piano keyboards, X/Y controllers, etc. with `faust2android` [15].

4.3. faust2smartkeyb

`faust2smartkeyb` [15] is a command line tool that can be used to generate iOS or Android apps where the standard FAUST UI is replaced by a more advanced interface, better adapted to a use in

[22] *Digital Signal Processing*

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

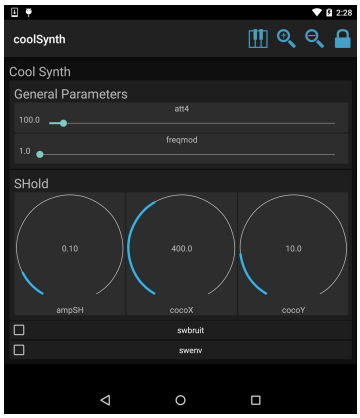

Figure 5: *Example of interface generated by* `faust2android` *containing groups, sliders, knobs, and checkboxes.*

a live music performance context and to touch-screens (see Figure 6). This type of interface uses the SMARTKEYBOARD system which is a highly configurable keyboards matrix where keys can be seen both as discrete buttons and continuous X/Y controllers. For example, a keyboard matrix of size 1x1 (a single keyboard with a single key) will fill up the screen which can then be used as a multi-touch X/Y controller. The interface can be configured directly from the FAUST code using a metadata.[24]

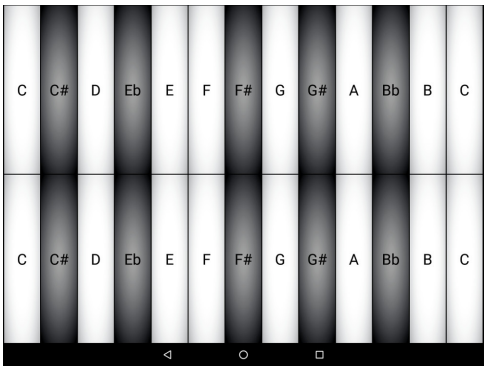

Figure 6: *Simple* SMARTKEYBOARD *interface.*

5. MICROCONTROLLERS

5.1. The OWL

The OWL (see Figure 7) is a series of programmable hardware (i.e., guitar effect pedals and modular synth modules) developed by Rebel Technology[25] for real-time audio signal processing. The OWL is based on an ARM Cortex M4 (STM32F4)[26] which is a microcontroller (168MHz 32bit) providing a FPU. It also hosts additional RAM (1Mb 10nS SRAM) – which is usually limited on this kind of chip – as well as a high quality stereo audio codec (24 bits, up to 96 kHz).

[24] `https://ccrma.stanford.edu/~rmichon/smartKeyboard/`

[25] `https://www.rebeltech.org`

[26] `http://www.arm.com/Arm-Cortex/CPU`

Figure 7: *The OWL pedal.*

The OWL is the first microcontroller-based platform that was programmable with FAUST. FAUST support is relatively basic on this platform and running `faust2owl` will generate C++ code usable directly within the OWL toolchain.

Rebel Technology developed other Cortex M4-based products programmable with FAUST since the OWL such as *Alchemist*, *Magus*, and *Wizard*.[27]

5.2. The Teensy

Teensy is a family of microcontroller-based prototyping boards developed and distributed by PJRC[28] and targeting the makers community. Just like the OWL (see §5.1), Teensys are based on ARM Cortex M microcontrollers which in some cases (i.e., Teensy 3.6 and 4.0) host an FPU, ensuring compatibility with FAUST. PJRC also distributes an "audio shield" which is essentially a breakout board for an audio codec (SGTL5000[29]) compatible with most Teensy boards. This allows us to add a proper stereo input and output to the Teensy.

FAUST has been supporting the Teensy for over a year now [16] through `faust2teensy`.[30] This command-line tool can be used to generate DSP objects compatible with the Teensy Audio Library.[31] E.g.:

```
faust2teensy -lib faustProgram.dsp
```

will turn `faustProgram.dsp` into a ready-to-use DSP object for the Teensy Audio Library.

Various optimizations are carried out to take advantage of the Cortex-M architecture. Because of the limited amount of RAM on the Teensy, FAUST only provides basic features on this platform (e.g., polyphonic objects cannot be created, etc.).

The Teensy 3.6,[32] which has been available for a couple of years now, is built around a 180MHz ARM Cortex-M4 offering plenty of computational power (up to 60 wave-table-based FAUST sine waves can be run in parallel at 48KHz with a block size of 8 samples). However, its limited RAM (256 kBytes) disqualifies it for applications with large memory footprints (e.g., long echos, advanced reverbs, etc.).

The Teensy 4.0 on the other hand is based on a 600 MHz Cortex-M7 with 1024 kBytes of RAM. Up to 650 wave-table-based FAUST sine waves can be run on it in parallel in similar conditions than in the previous example, providing enough computational power to run most standard DSP algorithms.

The Teensy offers plethora of advantages over other microcontroller-based boards. Indeed, it is relatively cheap (i.e., $35 for the 3.6, $19 for the 4.0, $13 for the audio shield) and it is designed and made in the USA, guarantying a certain level of quality.

5.3. The ESP32

The ESP32 is a microcontroller designed and produced in China by Espressif.[33] Its dual core architecture based on an Xtensa 32bits LX6 microprocessor operating at 240 MHz and with 512 kBytes of RAM also provides built-in WiFi and Bluetooth support for an unparalleled price ($3). The ESP32 has been used at the heart of a series of prototyping boards targeting real-time audio processing applications (i.e., audio codec, SRAM module, etc.) such as the LilyGO TTGO TAudio, etc. (see Figure 8), which can all be programmed with FAUST through the use of `faust2esp32`.[34] This command line tool can be used to generate a C++ DSP engine taking care of the entire audio processing chain from a FAUST program (unlike the Teensy where an external audio library has to be used – see §5.2). E.g.:

```
faust2esp32 -lib faustProgram.dsp
```

Drivers for various audio codecs are supported by `faust2esp32` and can be targeted through the use of options (i.e., `-wm8978` for the Cirrus Logic WM8978, `-ac101` for the AC101, etc.). Generated DSP engines are compatible both with the C++ (i.e., Python/makefile toolchain) and the Arduino programming environment of the ESP32.

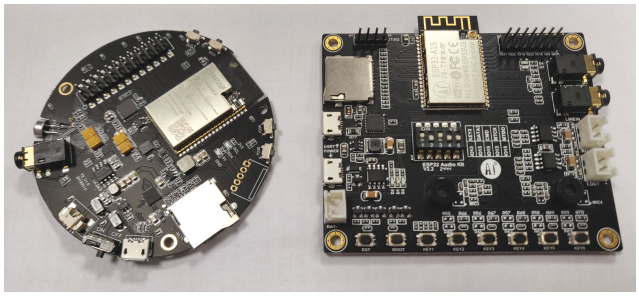

Figure 8: *ESP32-based audio Processing boards (the TTGO TAudio on the left and the ESP32 Audio Dev Kit on the right).*

Recent developments involve MIDI and polyphony support. Using the onboard WiFi, OSC over UDP can be implemented.

[27] `https://www.rebeltech.org/products/`
[28] `https://www.pjrc.com/teensy/`
[29] `https://www.pjrc.com/teensy/SGTL5000.pdf`
[30] `https://faust.grame.fr/doc/tutorials/index.html#dsp-on-the-teensy-with-faust`
[31] `https://www.pjrc.com/teensy/td_libs_Audio.html`
[32] `https://www.pjrc.com/store/teensy36.html`
[33] `https://www.espressif.com/en/products/hardware/esp32/overview`
[34] `https://faust.grame.fr/doc/tutorials/index.html#dsp-on-the-esp32-with-faust`

Additionally, the LilyGO TTGO TAudio can now be fully programmed with FAUST without writing a single line of C++ or Arduino code. In that case, metadata can be used to access the various analog inputs and GPIOs[35] of the board. These developments are being carried out in the context of the Amstramgrame project.[36]

The ESP32 offers a wide range of advantages over similar types of board. The principal one is its cost. For example, the LilyGO TTGO TAudio provides a comprehensive programmable platform fo real-time audio processing for less than $15! Another advantage of the ESP32 is its exhaustivity (e.g., WiFi, Bluetooth, etc.). They all come to the cost of quality of production and of tiny bugs that tend to pop up from time to time.

6. DIGITAL SIGNAL PROCESSORS: THE SHARC AUDIO MODULE

The SHARC audio module (see Figure 9) can be targeted with FAUST using the `faust2sam` command line tool which converts a FAUST program into a C++ package which can be inserted into a project on the Cross Core Embedded Studio (CCES) bare-metal framework.[37] Despite some efforts in that direction, this platform doesn't necessarily target the hobbyist/maker or computer music communities and its toolchain remains relatively cumbersome. For example, a USB JTAG emulator such as the Analog Devices ICE-1000[38] must be used for its programming. Moreover, FAUST objects running on the SHARC audio module can only be controlled using MIDI (through a physical MIDI connector) which can be very limiting.

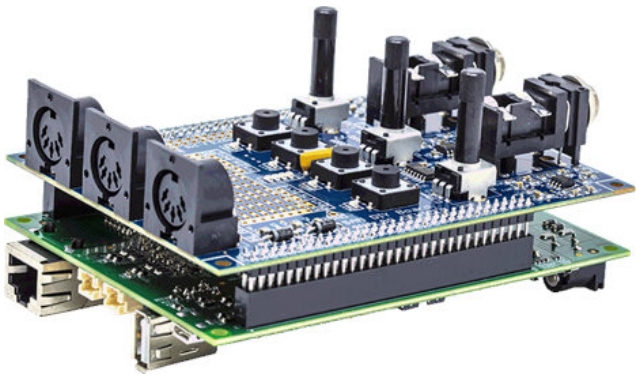

Figure 9: *The Analog Devices SHARC Audio Module.*

We'd like to support other types of Digital Signal Processors in the future. In particular, being able to target fixed-point chips would be a huge step forward towards making this type of platform more accessible (see §7.2).

7. FUTURE DIRECTIONS

While microcontrollers existed for decades before Arduinos, it's only when this platform appeared about fifteen years ago that they became fully accessible. The Arduino IDE[39]/programming environment played a huge role in that context by offering a high-level way to program these low-level platforms. Our conviction is that Domain Specific Languages (DSLs) such as FAUST can make low-level platforms with potential applications for audio processing (e.g., FPGAs, GPUs, bare-metal CPU-based embedded systems, etc.) more accessible to the makers, audio DSP, music technology, etc. communities. Such systems can target ultra-low latency applications and provide extended computational power for specific DSP algorithms. Our future efforts will focus on this topic.

7.1. Bare-Metal on the Raspberry Pi

CPU-based embedded Linux systems such as the RPI provide extended computational power and memory (e.g., the PI 3 A+[40] is based on a Broadcom BCM2837B0 Cortex-A53 with 4 1.4GHz cores and 512MB of RAM) compared to simpler architectures such as microcontrollers (see §5). Running real-time DSP algorithms in a bare-metal environment (without an OS) on these platforms should allow for further optimizations (e.g., by taking advantage of the Neon technology[41] on Cortex A processors, etc.) and reduced audio latency.

Some work has already been done in that direction by implementing the prototype of a FAUST architecture for Circle[42] which is a C++ bare metal programming environment for the RPI. Generated objects consist of kernels that can be placed on the PI's SD card. MIDI and polyphony support has been implemented as well.

The current toolchain remains relatively cumbersome though and the SD card has to be taken off of the RPI every time a new program is created: the system is completely static. Furthermore, the RPI doesn't give access to proper audio inputs and outputs (i.e., audio codec) and to sensor inputs (i.e., microcontroller), making it unusable in the context of embedded musical instrument design.

Our plan is to develop a bare metal kernel turning the RPI into a WiFi hotspot running a small web server. FAUST programs will be sent on the Pi remotely using this system and will then be compiled using the on-the-fly embedded FAUST compiler [17]. In parallel of that, an RPI hat/sister board adding a microcontroller and an audio codec could be developed.

7.2. FPGAs

We've been exploring for the past few months the idea of programming FPGA boards with FAUST in the framework of the SyFaLa project[43] which associates INSA Lyon's Citi Lab (Lyon, France) and Xilinx's expertise around FPGAs to FAUST (Grame's research team). While we're still in the preliminary stage of this project, we managed to run FAUST DSP objects on a Digilent Zybo Z7[44] which is a development board hosting a Xilinx Zynq-7000 FPGA (see Figure 10). This is currently done through High-Level Synthesis (HLS) by compiling FAUST programs to C++ and then to an IP core (Intellectual Property Core). A specific FAUST C++

[35] *General Purpose Inputs/Outputs*

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

[37] `https://wiki.analog.com/resources/tools-software/sharc-audio-module`

[38] `https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/emulators.html#eb-overview`

[39] *Integrated Development Environment*

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

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

[42] `https://github.com/rsta2/circle`

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

[44] `https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/`

backend had to be developed to make this possible. While our system works, it is currently hard to control audio processing parameters in real-time. Moreover, generated IPs are relatively inefficient, probably due to the use of floating points.

Next steps consist in generating fixed-point DSP objects to further optimize produced IPs and improving the control of processing parameters. The ability to produce fixed-point DSP objects could also be exploited in the context of Digital Signal Processors (see §6). In a more distant future, we'd also like to be able to generate VHDL code directly from FAUST. The results of this preliminary work are presented in a companion paper [18].

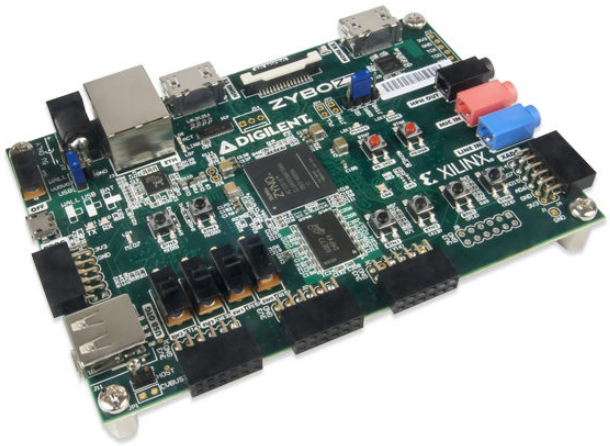

Figure 10: *The Digilent Zybo Z7.*

7.3. GPUs

Graphics Processing Units (GPUs) have been increasingly used in recent years for real-time audio processing applications, taking advantage of their high degree of parallelization to run DSP algorithms that can be easily divided into multiple processing units such as modal reverbs [19], etc.

We believe that FAUST has a role to play in that context by facilitating the programming of this type of platform. We did some experiment in 2010 by developing OpenCL and CUDA backends. At that time, results were not really convincing. Now that GPUs are becoming much more powerful, and with a better understanding of the class of DSP algorithms that can take benefit of their massive data parallelism capabilities, we plan to work again on this subject in the future.

8. CONCLUSIONS

For a long time, FAUST had been mostly targeting desktop applications such as standalones, plugins, externals, etc., but it took a huge step towards embedded hardware platforms in recent years. In many cases, programming these systems require specialist skills, making them out of reach to the audio programming community. We believe that Domain Specific Languages such as FAUST have a huge role to play in that context by making these platforms more accessible. This involves complex research challenges, especially in the case of ultra-low-level systems such as FPGAs. We plan to dedicate a lot of time in the future at tackling these problems.

9. REFERENCES

[1] Chris Anderson, *Makers: The New Industrial Revolution*, Currency Press, Sydney, Australia, 2014.

[2] Georg Essl, Ge Wang, and Michael Rohs, "Developments and challenges turning mobile phones into generic music performance platforms," in *Proceedings of the Mobile Music Workshop*, Vienna, Austria, May 2008.

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

[4] Ge Wang and Perry Cook, "ChucK: A concurrent, on-the-fly, audio programming language," in *Proceedings of the International Computer Music Conference (ICMC-03)*, Singapore, 2003.

[5] Scott Wilson, David Cottle, and Nick Collins, *The SuperCollider Book*, The MIT Press, 2011.

[6] Edgar Berdahl and Wendy 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.

[7] Andrew 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.

[8] Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner, "Embedding PureData with libpd," in *Proceedings of the Pure Data Convention*, Weinmar, Germany, 2011.

[9] Jack Atherton and Ge Wang, "Chunity: Integrated audiovisual programming in unity," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME-18)*, Blacksuburg, USA, 2018.

[10] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[11] Antonio Barbalace, A Luchetta, G Manduchi, M Moro, A Soppelsa, and C Taliercio, "Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application," *IEEE Transactions on Nuclear Science*, vol. 55, no. 1, pp. 435–439, 2008.

[12] Romain Michon, Julius O. Smith, Matt Wright, Chris Chafe, John Granzow, and Ge Wang, "Mobile music, sensors, physical modeling, and digital fabrication: Articulating the augmented mobile instrument," *Applied Sciences*, vol. 7, no. 12, pp. 1311–1320, 2019.

[13] Romain Michon, Yann Orlarey, Stéphane Letz, Domique Fober, and Catinca Dumitrascu, "Mobile music with the Faust programming language," in *Proceedings of the International Symposium on Computer Music Multidisciplinary Research (CMMR-19)*, Marseille, France, 2019.

[14] Romain Michon, "faust2android: a Faust architecture for Android," in *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx-13)*, Maynooth, Ireland, September 2013.

[15] Romain Michon, Julius Orion Smith, and Yann Orlarey, "MobileFaust: a set of tools to make musical mobile applications with the Faust programming language," in *Proceedings of the Linux Audio Conference (LAC-15)*, Mainz, Germany, April 2015.

[16] Romain Michon, Yann Orlarey, Stéphane Letz, and Dominique 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.

[17] Sarah Denoux, Stéphane Letz, Yann Orlarey, and Dominique Fober, "Faustlive: Just-in-time Faust compiler... and much more," in *Proceedings of the Linux Audio Conference (LAC-12)*, Karlsruhe, Germany, 2014.

[18] Tanguy Risset, Romain Michon, Yannn Orlarey, Stéphane Letz, Gero Müller, Adeyemi Gbadamosi, Luc Forget, and Florent de Dinechin, "faust2fpga for ultra-low audio latency: Preliminary work in the syfala project," in *Proceedings of the International Faust Conference (IFC-20)*, Paris, France, 2020, *Paper submitted for review*.

[19] Travis Skare and Jonathan Abel, "Gpu-accelerated modal processors and digital waveguides," in *Proceedings of the Linux Audio Conference (LAC-19)*, Stanford, USA, 2019.