Design and implementation of an open-source subtractive synthesizer on the Arduino Due platform

Rodolfo Pedó Pirotti [1], Marcelo Johann [1], Marcelo Pimenta [1]

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

rodolfo.p.pirotti@gmail.com, {johann, mpimenta}@inf.ufrgs.br

ABSTRACT

In this paper we present the design of a digital subtractive synthesizer using fixed-point arithmetic on the Arduino Due platform. Our main contribution is to show that a fully functional instrument of this type can run on a cheap and widely accessible processor. We have implemented oscillators with anti-aliasing algorithms, resonant filters, an envelope generator, a delay effect, a MIDI interface and a keybed scanner, therefore making a complete playable instrument. The implementation uses object orientation to create software modules replicating those classic analog synthesizer functions. With this approach, we have a modular software system that can be easily extended and adapted for new functionalities. An external DAC was used to provide the high-quality audio output of 16 bits at 48KHz. In addition to this, we also included an additive synthesis organ, demonstrating the possibility of having two important synthesis methods at the same time on the Due board. With this open and public design, we intend to contribute to the maker movement and encourage new and innovative implementations in this area.

1. Introduction

A significant number of audio synthesis techniques are used nowadays on professional musical instruments and home-made projects as well. Among these, the most fundamental are the subtractive synthesis and the additive synthesis. Subtractive synthesis became very popular with the analog synthesizers of the 60's and 70's, which used hardware modules that could be connected together, with a modular concept. The introduction of this type of electronic instrument brought new areas of possibility to sound creation and musical performance – the adoption and popularity of analog synthesizers grew up fast with the synthesizers launched on those years.

Nowadays, on the digital domain, virtual analog instruments, which are digital and software emulation of analog synthesizer functions, are common, either as separate instruments or as part of most digital synthesizers. Nevertheless, the prices of these instruments are often high.

The complexity involved in the design of professional digital musical instruments is naturally increasing as the technology progresses. It usually involves the design of ASICS (application-specific integrated circuits), or programming of complex DSPs (Digital Signal Processors), lots of memory and all the sound capture and processing tasks required to prepare samples, design of complex PCBs (Printed Circuit Boards) and so on. In some sense, the complexity usually employed makes us believe that this is absolutely necessary to get a functional instrument.

On the other hand, smaller processors have also evolved tremendously, and nowadays, for a small budget, there is available a large number of open-source platforms. Boards like Arduino, Raspberry Pi and Beagle Bone provide computing, processing power and interfaces on a ready-to-go board that can be used even by people with almost no knowledge of engineering and electronic components, and the number of users of these platforms is growing every year [1]. Part of this increasing number of projects and boards is related to the DIY culture and maker movement, emerging movements in which people aim to create, design, build or modify products, equipment, home items, making art projects, music and several other things.

In the audio realm, there have been many attempts to implement basic audio synthesis using platforms such as the Arduino. But until recently, it was not possible to implement full-featured instruments with an audio quality considered as "professional", here defined as being able to generate and output audio with at least 16-bit resolution, 48 kHz sample rate and no output aliasing, because the basic boards that were widely available lacked resources and computing power for that. In other words, the audio synthesis implementations for the Arduino Uno, for example, are very interesting from the point of view of learning and experimentation, but too limited to be used to implement a "real instrument", something with an audio quality that would be used by a trained musician for traditional performances or studio recording. They present too little resolution, limited bandwidth, lots of noise and aliasing and so on.

In this work, we wanted to investigate the hypothesis that it is possible to design and implement a fully functional subtractive synthesizer with good quality audio output using the still limited processing resources of the Arduino Due platform, in opposition to the premise that a high-quality music synthesizer needs dedicated and expensive processors and electronic components. By achieving this goal, this design could become a reference to be used by others to build their own synthesizer on a low-budget hardware.

The rest of this paper is organized as follows: Section 2 presents a comparison with related work and the specific goals of this paper. Section 3 presents the design of subtractive synthesizer modules, an overview of software modules integration and considerations about the additive synthesizer. Section 4 presents analysis and measurements made with the implementation. Finally, in Section 5 we describe some conclusions and discussions about future ideas.

1.1 The Arduino Due Platform

Arduino is an open-source prototyping platform based on flexible and easy-to-use hardware and software. Its usage has been growing on recent years by students, professionals and amateurs in many areas. Comparing to other open-source platforms listed before (Raspberry PI and Beagle Bone), Arduino is more suitable for low-level applications, and its learning curve is usually faster comparing to others [1].

The Arduino Due has a 32-bit ARM microcontroller running at 84 MHz and 96 kB of RAM memory. With these specifications, unlike simpler boards as Arduino Uno or Nano, we estimated that serious audio applications could run output audio with minimum professional audio quality (16 bits / 48 kHz).

2. Context

In order to provide the context to our contribution, we have listed a few previous related works with implementation of subtractive synthesizers and classified them according to the following criteria:

- Build complexity: how hard it is for someone with small knowledge of electronic and computing to build;
- Total cost: total cost to build and use the synthesizer;
- Usage purpose: if for learning purpose only, with no commitment regarding the output quality; if for usage as a real music instrument;
- Output quality: if the output fits the minimum requirements of 16 bits of audio resolution, sample rate above Nyquist frequency (sample rate at least twice the highest harmonic – around 17 kHz for additive synthesis organ and 44.1 kHz for virtual analog) and no output aliasing.

Table 1 shows a comparison of related works. By looking at their characteristics, we can observe that our proposal differs from others in many aspects. The work of [2] presents a subtractive synthesizer implementation for accessible platforms, but with low audio quality output and no anti-aliasing algorithms. In [3], authors only propose how to design an instrument, without showing the actual implementation, whereas in [4] it is necessary to pay for the project. There are other projects like [5], which presents a design suitable for a PC computer, and [6,7], as examples of complete analog synthesizer designs. In such cases, they are either too expensive and with a high complexity to build, or require a full personal computer to run, different from the purpose of this work.

Author	**[2]**	**[3]**	**[4]**	**This work**
Build complexity	Medium	NA	Easy	Medium
Total cost	Cost of Arduino Uno (aprox. U$ 22), plus few external components	NA	Product sold by U$ 266	Cost of Arduino Due (aprox. U$ 30), plus few external components
Usage purpose	Learning	NA	Learning	Instrument
Output quality	Poor	NA	Poor	High

Table 1. Related work comparison

2.1 Specific goals

After the comparison, we define the specific goals of this work:

- To design and develop a subtractive synthesizer using classic modular analog synthesizers as reference, on a cheap, widely available and easy to use platform – Arduino Due;
- To identify and use efficient algorithms that fit in the limited processing capacity of the chosen platform.
- To have a modular software design that makes it easy to change and add features as needed;
- To add a good-quality audio output, so the synthesizer can be used as a real musical instrument;
- To include on the design an additive synthesis, based on [8], to show that it is possible to have two fundamental synthesis methods programmed at the same time on a small budget platform;
- To contribute to the maker movement by sharing the design, encouraging new and innovative implementations in this area and increasing the usage and access to (electronic) musical instruments and the musical creativity.

3. Synthesizer Design

We started the design by establishing some guidelines so that the implementation could be successful, easily tested, and further extended. We defined that:

- for mathematic calculation, we must use fixed point arithmetic instead of floating point arithmetic, as the Arduino Due processor does not have floating point unit.
- for subtractive synthesis, the software model should use an object-oriented design, making separate classes which mimic the functions of analog synthesizers (their modules);
- for test and simulation, to implement mock or hardware abstraction functions that enable the test and simulation of algorithms on a standard PC;

3.1 Time-critical, time-accurate and housekeeping tasks

This organization was already proposed by [8], and is used in our implementation to efficiently compute the necessary functions according to their criticality. The time-critical tasks include the code for the audio processing, like waveform generation, filter and effect calculation. This is the code that needs to run at the exact sample rate to produce each audio sample.

Time-accurate tasks include tasks whose updates are much less frequent, but depend on correct timing. They do not need to be computed at each sample, but still need to be accurately evaluated over time. They typically include the envelope generator and other low-frequency modulation functions. Part of this code might still be computed at the main loop, but at least the time-accurate interruption needs to keep track of the time elapsed by means of one or more counters, as needed.

It is important to note that the time-accurate tasks must be implemented as an interrupt with a lower priority than the time-critical code, otherwise they will interfere with it.

Finally, housekeeping tasks include the keybed scanning, switches and analog input readings, as well as any general control code. This last set of tasks do not need a time-accurate execution and can be slightly delayed as long as we keep their periodicity under acceptable values so not to impact the playing experience in a bad way.

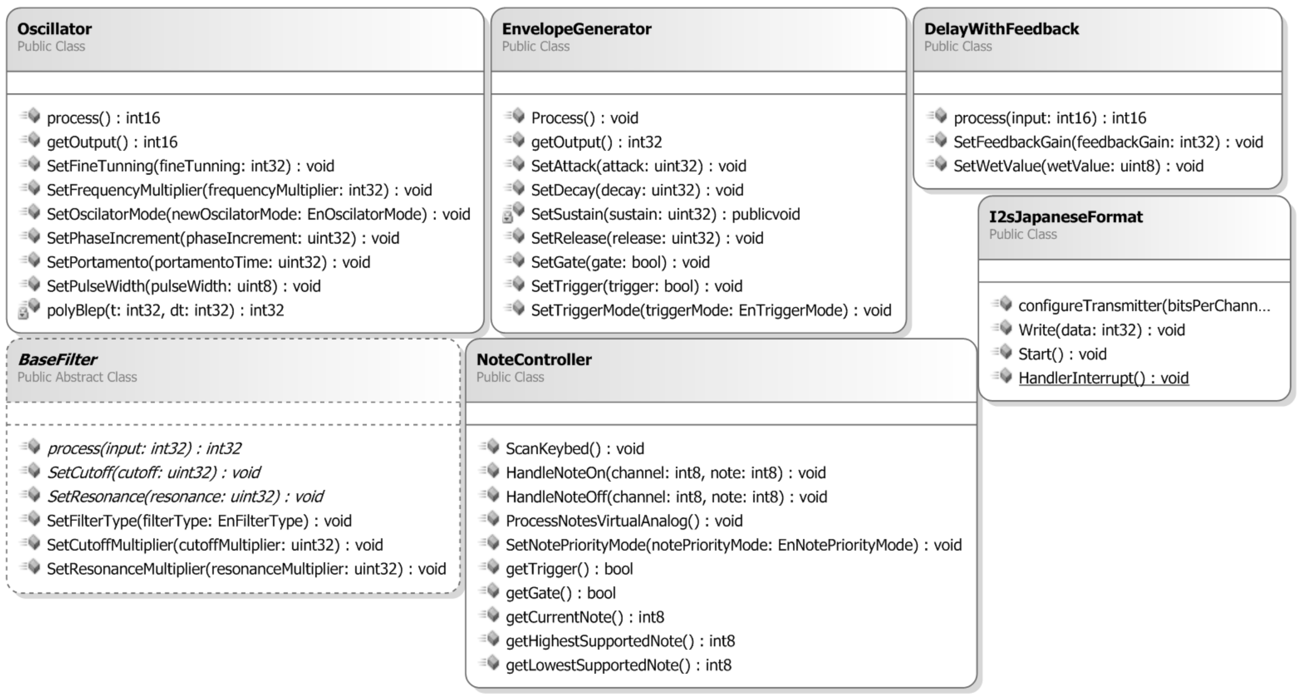
Oscillator
Public Class
process() : int16
getOutput() : int16
SetFineTunning(fineTunning: int32) : void
SetFrequencyMultiplier(frequencyMultiplier: int32) : void
SetOscilatorMode(newOscilatorMode: EnOscilatorMode) : void
SetPhaseIncrement(phaseIncrement: uint32) : void
SetPortamento(portamentoTime: uint32) : void
SetPulseWidth(pulseWidth: uint8) : void
polyBlep(t: int32, dt: uint32) : int32
EnvelopeGenerator
Public Class
Process() : void
getOutput() : int32
SetAttack(attack: uint32) : void
SetDecay(decay: uint32) : void
SetSustain(sustain: uint32) : publicvoid
SetRelease(release: uint32) : void
SetGate(gate: bool) : void
SetTrigger(trigger: bool) : void
SetTriggerMode(triggerMode: EnTriggerMode) : void
DelayWithFeedback
Public Class
process(input: int16) : int16
SetFeedbackGain(feedbackGain: int32) : void
SetWetValue(wetValue: uint8) : void
I2sJapaneseFormat
Public Class
configureTransmitter(bitsPerChann...
Write(data: int32) : void
Start() : void
HandlerInterrupt() : void
BaseFilter
Public Abstract Class
process(input: int32) : int32
SetCutoff(cutoff: uint32) : void
SetResonance(resonance: uint32) : void
SetFilterType(filterType: EnFilterType) : void
SetCutoffMultiplier(cutoffMultiplier: uint32) : void
SetResonanceMultiplier(resonanceMultiplier: uint32) : void
NoteController
Public Class
ScanKeybed() : void
HandleNoteOn(channel: int8, note: int8) : void
HandleNoteOff(channel: int8, note: int8) : void
ProcessNotesVirtualAnalog() : void
SetNotePriorityMode(notePriorityMode: EnNotePriorityMode) : void
getTrigger() : bool
getGate() : bool
getCurrentNote() : int8
getHighestSupportedNote() : int8
getLowestSupportedNote() : int8

Figure 1. Main classes developed for the subtractive synthesizer

With this structure in mind, the code is composed of the classes shown in Figure 1, which will be described in the next sections.

3.2 Oscillator module

According to [9] and [10], the oscillator module converts a voltage value to a waveform output on a specific frequency. Usually it also has control inputs such as to modulate the main frequency and select waveform type.

In our implementation, we've created a software class that implements the oscillator, with features such as waveform type selection, portamento (a feature usually associated to the keybed on classic analog synthesizers, but we chose to move it to the oscillator class), frequency modulation (to use together with an LFO), fine tuning and octave selection.

One of the key points of the oscillator class is the waveform generation algorithm. The Numeric Controlled Oscillator (NCO) [11] is the simplest approach and does not require a lot of calculation, but it produces undesired aliasing in the output signal for pulse and sawtooth waves. A comparison of anti-aliasing algorithms is presented on [12], and based on that we chose to use the Polynomial Bandlimited Step Function (PolyBLEP) algorithm for pulse and sawtooth waves, while the sine wave uses the wavetable method [13], as there is no aliasing on this method for sine waves. The PolyBLEP algorithm was implemented using fixed-point arithmetic and 32-bits integer types.

3.3 Envelope Generator module

According to [9] and [10], the Envelope Generator (EG) module generates a signal used to control other modules in order to give a contour to some parameter, like the signal amplitude or filter cutoff frequency.

Our design has control inputs to configure *Attack*, *Decay*, *Sustain* and *Release* values, plus *Gate* and *Trigger* signal simulation. A seventh input is used to configure the EG behavior (*SetTriggerMode*). The output value is a 32 bits fixed point value, with 16 bits of fractional part. The output value is always less or equal to 65536 (equivalent to 1.0), and can be used as a multiplier to another module parameter. Figure 2 shows the state machine of this module.

3.4 Filter module

For the filter module, we created a base class to be an interface class to allow different filter implementations. On our current prototype, there are two different low pass filters, one based on [5] (a fourth-order filter) and other based on [14] (a fourth-order filter with resonance based on Robert Moog's filter used on Moog synthesizers).

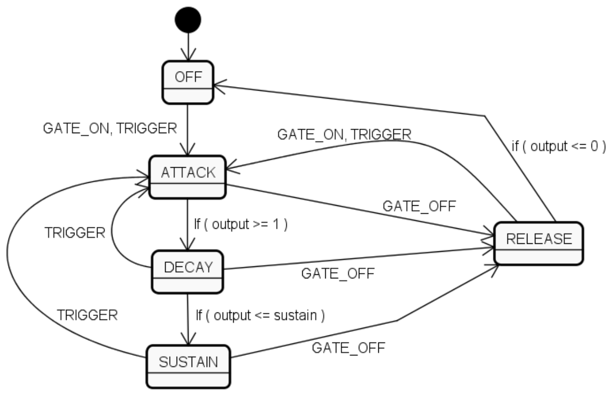

Figure 2: Envelope generator state machine

3.5 Delay effect and note controller modules

The delay effect module is a simple delay with feedback effect, used to create a repeating and decaying echo in the sound. The note controller module was created to be responsible for managing the notes being played and generate the gate and trigger signals. It has methods to process MIDI events (note on and note off) and a method to scan a keybed.

3.6 External DAC

Arduino Due processor has a 12 bits integrated DAC (digital to analog converter), which is good for general applications and even some experimental audio applications, but in order to improve the audio processing and reduce output noise, we chose to use an external 16 bits DAC.

The DAC used is the TDA1543A, a dual-channel 16 bits DAC for usage in hi-fi applications. Using the code provided by [15] as reference, we implemented our own class responsible to configure and send data to the DAC.

3.7 Module integration

Each software module was created as a separate class and can be instantiated as different objects. This approach makes it possible and easy to add (or remove) oscillators, envelope generators, filters and effects (and other features) to the synthesizer, keeping in mind the processing time of each module.

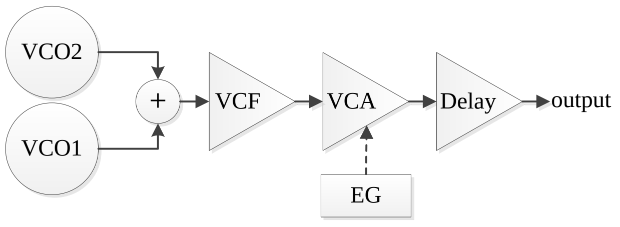

Figure 3: Typical synthesizer diagram

Figure 3 shows an example of a typical connection between analog synthesizer modules. The code below is the equivalent code in our design to implement the same architecture and connections.

```
output = pVCO1->process() + pVCO2->process();
output = pFilter->process( output );
output = output * pVcaEg->getOutput() >> FRACT_WIDTH;
output = pDelay->process( output );
```

3.8 Testing and simulation

This implementation runs on an embedded system with limited resources and interfaces. This would make it harder to debug and validate the algorithms. To improve testing and simulation capabilities, we created empty code definitions and calls to hardware related interfaces. This enables us to build the code using a standard compiler like GCC or MS Visual Studio on a standard PC, taking advantage of all debugging tools.

3.9 Additive synthesis organ

When we started the development of the subtractive synthesizer, we realized that the most critical concern is the processing speed, mainly because of the lack of a hardware floating point unit. After the first code implementations and tests, we observed that the amount of flash and RAM memory available on the Arduino Due processor was much more than needed for our proposal. We decided therefore to implement an additive synthesis organ, based on the implementation described in [8], and integrate it to the subtractive synthesizer.

The implementation, based on classic Hammond organ design, includes the functions of 9 drawbars, 61 notes with full polyphony, 96 oscillators (compared to 91 implemented on Hammond organs), vibrato and tremolo effects. For this implementation, the output sample rate was reduced to 24 kHz, but it is enough for the organ design because its highest oscillator runs at 8372 Hz.

The integration of subtractive and additive synthesis cannot be simultaneously played, as each audio generation tasks takes up almost all the processing power. On the other hand, they can coexist programmed at the Arduino Due board at the same time, and they can be switched on the fly, e.g., the user can select one or the other mode and start playing right away by changing a simple switch.

3.10 Block diagram

Figure 4 shows a block diagram of the final synthesizer. The Arduino Due board is the main component. In addition to this, there's an array of potentiometers, used to control analog parameters in realtime, an array of digital switches, used to enable and disable features, the digital-to-analog converter, the MIDI input circuit, and digital pins connected to the keybed, some configured as output and others configured as inputs (the keybed works as a matrix with 8 columns and 7 rows).

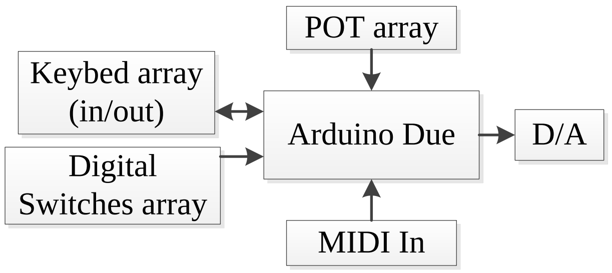

Figure 4: Hardware block diagram

Figure 5 shows one potentiometer input. The POT array shown on Figure 5 is an array of 11 potentiometer inputs, each one connected on a different Arduino analog input.

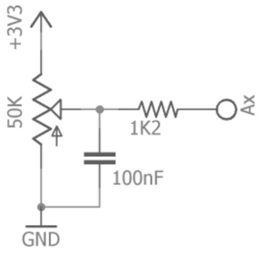

Figure 5: Potentiometer input

The digital switches array is a group of 16 digital switches, connected to Arduino digital inputs. Figure 6 shows an example of 1 input.

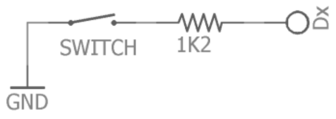

Figure 6: Digital switch input

The D/A is the circuit used for the TDA1543A DAC component, shown on Figure 7.

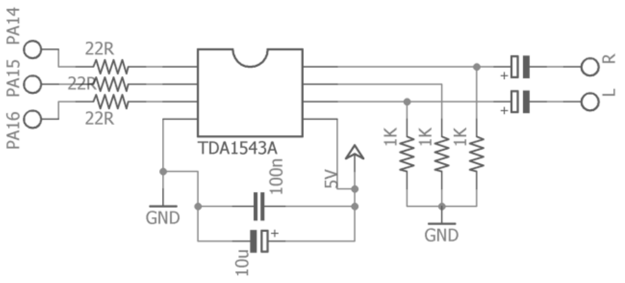

Figure 7: TDA1543A digital-to-analog converter

Figure 8 shows the MIDI input circuit.

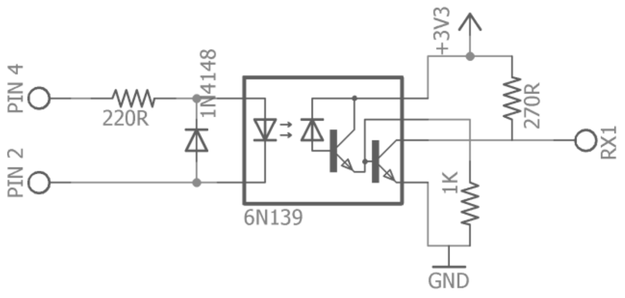

Figure 8: MIDI In circuit

For the keybed scanning, there are 7 digital ports configured as output and 8 digital ports configured as input. Figure 9 shows one of the output digital ports (Dx is Arduino output and Ko is keybed), and Figure 10 shows one of the input digital ports (Dx is Arduino input and Ki is keybed).

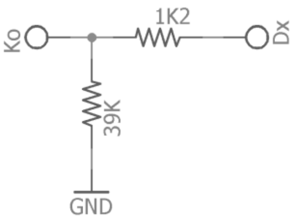

Figure 9: Arduino output for keybed

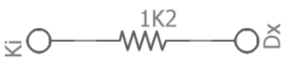

Figure 10: Arduino input for keybed

4. Features and output analysis

We present here some analysis regarding important areas of the implementation.

4.1 Output aliasing check

The anti-aliasing algorithm was chosen based on the analysis of [12]. The PolyBLEP algorithm seemed to be a nice approach, with low processing overhead and good output results. Figure 11 shows the frequency spectrum of the output **without** the PolyBLEP algorithm for a square wave, C note, frequency of 2093 Hz.

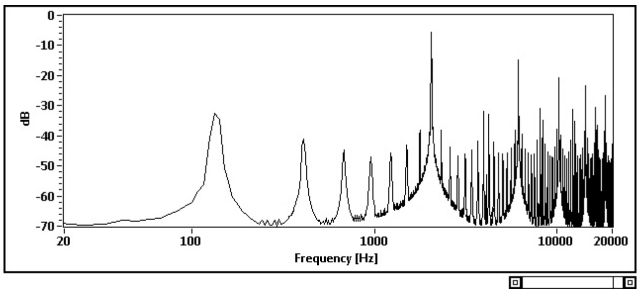

Figure 11: Frequency spectrum without PolyBLEP

Figure 12 shows the frequency spectrum of the output with the PolyBLEP algorithm, for a square wave, C note, frequency of 2093 Hz. It is clearly visible the reduced aliasing caused by the PolyBLEP addition.

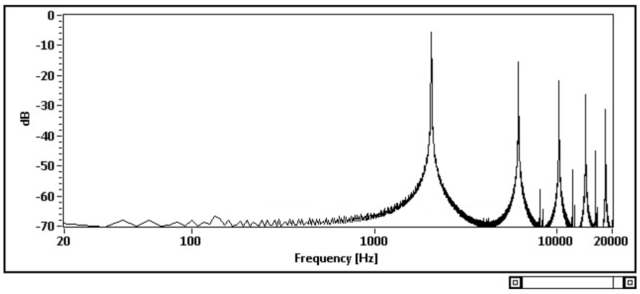

Figure 12: Frequency spectrum with PolyBLEP

4.2 Filter spectrogram

Figure 13 shows the spectrogram of one of the filters implemented [14] for a single note using square wave, while increasing the cutoff frequency and the resonance control.

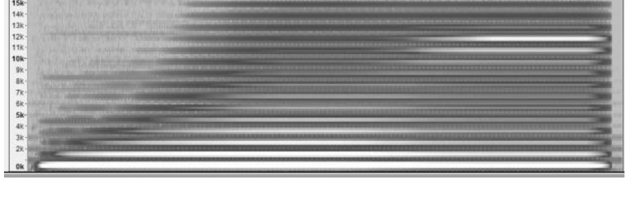

Figure 13: Filter spectrogram

5. CONCLUSIONS

In this paper we presented the design and implementation results of a high quality subtractive synthesizer implemented on the Arduino Due platform. We also demonstrated that it is possible to have an additive synthesis organ inspired on Hammond design on the same platform at the same time. The complete implementation is open, public and focused on low cost platforms. Although the implementation was created on the Arduino Due, we believe it fits also on any other similar board containing an ARM Cortex-M3 processor or other processors with similar architectures (maybe removing some features or modules if using a slower clock than 84 MHz).

The fixed-point approach completely met the expectations for the implemented algorithms. With the 32 bits processor, we could use fixed point calculations with a good resolution.

The required time for processing the main modules allowed us to include two oscillators, two envelope generators with selectable trigger mode, one LFO, one low pass filter, one effect module, a MIDI receiver and a keybed on a 48 kHz sample rate. The PolyBLEP algorithm, implemented with fixed point calculation also presented good results, by removing a significant part of the audible aliasing from the pulse and sawtooth waves.

Regarding the filter implementation, we used two filter designs from [5] and [14] and, although they are good starting points and bring a lot of possibilities to the sound creation, we believe this is one of the key points for further improvement. As future work, we propose the implementation of different filters that could replicate better the filters used by classical synthesizers, such as Moog, ARP or Oberheim.

The object-oriented design made it possible to easily modify the synthesizer features and allowed us to validate the algorithms by running the code on a standard PC. We encourage the usage of this approach by having function mocks or empty function calls for hardware related functions.

Finally, both subtractive and additive synthesis are available on the same code and platform, and the user can select between them with a simple toggle switch connected to an Arduino Due digital pin.

As for next steps, besides a better filter implementation, we intend to implement more effect types, "keyboard tracking" on filters and LFOs, to make the parameter control to be adjustable depending on the note being played (as present on old analog synthesizers), two note polyphony (but this would reduce the total number of modules instantiated in the system) and other waveform types, such as sample-and-hold and noise.

Furthermore, the successful implementation of two synthesizers on the simple Arduino Due board using fixed point arithmetic demonstrates that complex audio functions can be implemented cheaply in this platform and provide a good audio quality. It naturally opens up several other opportunities for implementing other forms of audio synthesis, other functions and modules, and also innovative ideas with the contribution of a larger community.

The source code is available in a public repository [16]. The schematics and some pictures of the synthesizer finished are also included in the repository. A video of the project for sound demonstration is available at *https://youtu.be/asuycIvozhg*.

6. References

[1] P. Jamieson and Jeff Herdtner, "More missing the Boat - Arduino, Raspberry Pi, and small prototyping boards and engineering education needs them" in *Frontiers in Education Conference (FIE), 2015. 32614 2015. IEEE*, 2015. doi: 10.1109/FIE.2015.7344259

[2] T. Barrass. (2016). "Mozzi Sound Synthesis Library for Arduino". Source code available at http://sensorium.github.io/Mozzi/

[3] A. Huovilainen, "Design of a Scalable Polyphony-MIDI Synthesizer for a Low Cost DSP", M. S. thesis, Department of Signal Processing and Acoustics, Aalto University School of Science and Technology, Espoo, Finland, 2010.

[4] L. Biddulph and J. Ziembicki. (2000). "AVRSynth". Available at http://www.elby-designs.com/contents/en-us/d5.html

[5] M. Finke. (2013). "Martin Finke's Blog". Available at http://www.martin-finke.de/blog/articles

[6] Y. Usson, "Yusynth". Available at http://home.yusynth.net/

[7] *modular.br*, Facebook private group, containing 1.108 members on September 2016. Available at https://www.facebook.com/groups/modularbr/

[8] < Omitted for blind review >

[9] D. Crombie, *The Complete Synthesizer: A Comprehensive Guide*, Omnibus Press, 1982.

[10] Devahari, *The Complete Guide To Synthesizers*, Prentice-Hall, Inc., 1982.

[11] E. C. Kisenwether and W. C. Troxell, "Performance Analysis of the Nuumerically Controlled Oscillator" in *40th Annual Symposium on Frequency Control*, 1986. doi: 10.1109/FREQ.1986.200971

[12] V. Valimaki and A. Huovilainen, "Antialiasing oscillators in subtractive synthesis" in *IEEE Signal Processing Magazine*, v. 24, n. 2, p. 116–125, 2007. doi: 10.1109/MSP.2007.323276

[13] W. Pirkle, *Designing Software Synthesizer Plug-Ins in C++*, Burlington, MA: Focal Press, 2014. [E-book] Available: Safari Books Online.

[14] P. Kellett, "Moog VCF, variation 1, 24dB resonant low pass", source code available at http://www.musicdsp.org/archive.php?classid=3#25

[15] Delsauce. (2013). "ArduinoDueHiFi - An I2S audio codec driver library for the Arduino Due board", source code library avaiable at https://github.com/delsauce/ArduinoDueHiFi

[16] https://github.com/rppirotti/rgsynth