An additive synthesis organ with full polyphony on Arduino Due

Marcelo Johann[1]

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

johann@inf.ufrgs.br

Abstract. *In this paper we present the implementation of an organ by classic harmonic addition, with full 5-octave polyphony, on the Arduino Due platform, which has an 84MHz RISC processor. With this widely available and easily programmable system, we were able to implement such instrument including all needed controllers, keyboard scan and MIDI input, delay and a simple rotary speaker effect, while serially communicating with a high-quality external DAC. It demonstrates the power of recent microcontrollers in performing complex music and audio duties and opens up many other opportunities for ubiquitous music application.*

1. Introduction

The Arduino Platform is causing a revolution in how people creatively learn and use computing technology to interact with real life situations. Arduino is a microcontroller. Microcontrollers are small processors integrated with a simple but comprehensive set of input/output ports, including basic analog and digital interfaces, their own RAM and program memories. But microcontrollers in the past were more or less restricted to engineering experts, as they frequently required special hardware to be programmed, used proprietary technology from each vendor, and different set of languages, libraries and programming tools, with all associated compatibility and learning challenges. Yet the Arduino project has succeeded in providing a standard, homogeneous platform that caught the world. As an open plataform it managed to create a huge community of fans right from the beginning.

People reach to use an Arduino microcontroller when they need a portable and small device that can interact with the environment through sensors and actuators in a simple way, what is complicated and expensive to achieve with standard computers or smartphones. The Arduino has been used in many applications related to creative industry, performing arts, installations, computer music, processing MIDI and digital audio. But because it uses a relatively simple processor, it cannot easily handle more computing-intensive tasks, as audio with CD or studio-quality sample rates and bit depths, multi-channel audio and so on. Nevertheless, people have managed to create many types of works with audio either with raw features or with more elaborate libraries like Mozzi [Barrass 2014], using PWM, 8-bit sound, limited polyphony, very low sampling rates, and this was already exciting. In 2012, however, a new platform was introduced, the Arduino Due, using a much more powerful RISC processor, what opens up the opportunity for more serious audio works.

1.1. The Arduino Due Platform

The Arduino Due Platform is quite recent. It was introduced in late 2012, and is based on an ARM RISC processor (Atmel SAM3X8E, Cortex-M3-based) running at 84 MHz. The architecture is 32 bit natively, and is able to perform both additions and multiplications, as well as most of its other operations, within one clock cycle each.

The processor on the Arduino Due has also two 12-bit DACs, what makes it possible to output analog signals with resolution of 4096 levels from the first time in the platform without additional components, at high sample rates. Additionally, it includes a hardware module dedicated to I2S communication, a serial protocol used by many audio chips (DACs and ADCs) that can be used to connect to such components and transmit/receive audio with up to 32-bits per sample, per channel.

This looks very promising to audio and music applications, but because it is quite new, there seems to be little work already done in the platform. People still concentrate on the Arduino Uno, for it is more widely accessible an accepted. As an example, during the development of this work, it was very difficult to find references or reports of successful communication with the Due's I2S protocol. Most of the searches returned either questions or failure reports. There is only one library provided by [Delsauce 2013] that reported partial success, but limited to the operation as a "slave" device (clocked by an external signal). It uses code that is not easily traceable to its origins, some custom code, and reports supposed errors in both libraries and documentation. Therefore, it is clear that additional work is possible and needed to explore the platform for audio processing.

1.2. Objectives

Based on previous works with audio on the Arduino Uno, and the attractiveness of the Due Platform, we wanted to investigate what could be done with it. In a short course given at the V Ubimus in 2013, we have demonstrated how to implement basic oscillators and frequency modulation with the 8-bit Arduino Uno, which is not any novelty. For that, we have used a discrete R2R DAC created with a set of resistors, look-up table-based sine and interrupt-based synchronism, like in [Ghassaei 2013]. So, although the audio was generated with only 8 bits and presented a lot of noise, the sampling frequency was high, and the resulting sound was interesting, with frequencies resembling those of analog circuits.

Any other improvement in terms of sound quality and polyphony, however, would be very difficult, if not impossible to achieve. For higher bit depths, not only its 8-bit architecture becomes slow, but external DACs are needed. They also need to have parallel interface, as the Uno is not able to serially communicate with them because of its 16 MHz clock. That was the main motivation for trying the Arduino Due. We expected to be able to implement audio with professional sampling rates and bit depths, also achieving usable polyphony and more sophisticated sounds when implementing sound synthesis.

Two obvious choices for sound generation were frequency modulation an additive synthesis. Frequency Modulation is the most efficient way of generating complex sounds with few computer instructions [Chowning 1973], and was intensively used by Yamaha for synthesizers, video games and computer sound cards for such reason. On the other hand, the pleasant sound resulting from additive synthesis

employed at the Hammond electro-mechanical organs [Hammond 1966] have had and still has a tremendous impact in music of many styles. Additive synthesis can be used to generate all sorts of sounds, provided that an unlimited set of oscillators each with appropriate envelopes and modulators is provided. In this work, however, we are particularly interested in the "organ" concept and sounds, as it was employed in traditional instruments from pipe-organs, Hammonds and electronic combo organs in general.

We have chosen to experiment additive sound synthesis first. The goal is to implement some sort of organ, not necessarily to accurately replicate any particular instrument. We shall take inspiration and use the Hammond organ as reference, but it was not the main objective to make detailed replications of their particularities, imperfections, sound signatures, effects. The objective, however, was to create a playable and pleasant instrument. And for that, some of Hammond's distinct characteristics turned out to be almost mandatory, as they have a logical meaning, a natural reason of existence, as it will be highlighted later.

1.3. Organization

The rest of this paper is organized as follows. Section 2 presents the Hammond organ as a reference, reviewing its main characteristics, such as number of notes, oscillators, harmonics and other features. Section 3 represents the core of this work, in which we evaluate how to implement all needed oscillators in real time on the Due platform, and develop an efficient architecture that is able to do that. It ends by defining three classes of time-criticality and the tasks that lie on each of them. Section 4 describes the integration of an external DAC along the modifications in the core audio generation that are required for that. Section 5 shows how simple audio effects as delay, vibrato and tremolo are implemented in general, and particularly a very simple emulation of the effect produced by rotary speakers. Section 6 discusses the achieved results and presents a set of interesting possibilities that can be explored in the platform from this starting point. Finally, section 7 concludes the paper.

2. The Hammond Organ

The Hammond Organ is an electromechanical instrument created by Laurens Hammond in 1932 and produced until mid-60's, where a motor rotates a set of gears and *tonewheels* close to magnet pickups in different speeds so as to generate basic sine waves covering about 8 octaves [Hammond 1984]. Those almost pure sine partials are then added according to harmonic progression (first integer multiples) to compose different waveforms, thus implementing additive sound synthesis. Such organ follows the more traditional pipe-organs and the first electronic instrument, the Telharmonium, and was followed by electronic organs (also known as theater organs and combo organs), but it has a unique place in music history and aesthetics, not only for its sound, but also for its systematic approach to sound synthesis.

A typical Hammond organ (variations exist) has 61-note keyboards, with each key (with some exception at the extremes) being able to play 9 harmonics. The harmonic series, however, is not a simple integer series from 1 to 9. It is composed of the following harmonics: 1 (fundamental), 2nd, 3rd, 4th, 5th, 6th, 8th harmonics, one octave below the fundamental, and its third harmonic, which is the fifth of the

fundamental. For this project, we first experimented with the simple series of 1 to 8, 1 to 9 and 1 to 10 harmonics, and we clearly identify the advantages the particular arrangement defined by the Hammond Company. The fifth combines so much with a fundamental note that we do not need to introduce it only at the second octave. They sound much more coherent closer in frequency than farther apart. The seventh harmonic, however, plays a minor seventh, which is not part of a major scale, and is not present on the organ's harmonics for that.

In terms of semitone displacement, the series represents the sequence {+0,+19,+12,+24,+31,+36,+40,+43,+48}. In the conception of the Hammond organs, the designer correctly identified that the harmonics of one note are fundamentals of others, and the same oscillators could be used provided that they can be added somehow. For the 61-note instrument, oscillators would need to cover a range of 61+48, or 109 semitones, but the last waves would have frequencies so high that they not add too much to the sound, and might be difficult to be built. The designers decided therefore to stop at the 91th oscillator, what means that the last notes can only have their first four harmonics. In the implementation developed in this work, we decided to slightly extend the range, implementing 96 oscillators, so that the last notes are able to play five harmonics instead.

There are many subtleties in the construction and operation of the Hammond organ that affect its emblematic sound. A large set of gears is used to move *tonewheels* with a single synchronous motor, and the ratio of gear rotation speeds to number of *tonewheel* teeth determine available frequencies, which are not exactly the ones needed for the target equal temperament [Wiltshire, 2008]. The harmonics are added up with some weighting, carefully chosen for musical soundness, and many vibrations, leakage and other imperfections contribute to its distinctive and organic sound. One characteristic that stands out, however, is the foldback. At the same time that the designers decided to stick with a limited number of oscillators, they wired the missing high frequency harmonics to the ones available one octave below. This seems strange at first, but when implementing a similar instrument, we quickly realize why. If a user programs a sound that has the fifth harmonic at some octave, whenever he plays a note for which this harmonic is not present, the fifth is suddenly missing compared to other notes being played, and it sounds quite different from the others. If a fifth is introduced one octave below, it helps in keeping the sound similar to the others.

For this work we did not take time to closely match most of these imperfections and subtleties. We did not even implemented foldback, even though this is straightforward. On the other hand we attempted to achieve a high quality basic sound, and successfully implemented sound effects, so that we can demonstrate that a complete playable instrument is possible with the embedded platform, as described below.

3. Implementing 96 oscillators at 24 KHz rate

To generate audio in the digital domain, a Numerically Controlled Oscillator (NCO) consists of a phase accumulator (PA) followed by a phase to amplitude converter (PAC). The PA is a simple counter that counts forward progressing in steps computed from the desired frequency and sampling rate. Yet the most efficient implementation of the PAC to generate a sine wave uses a look-up table containing a quarter or a full wave sample. The PA should be able to increment fractions of the table index in order to

generate low frequency, and this can be accomplished by floating-point or fixed point arithmetic. The resulting value of the phase accumulator must match the table size, but as in software we have only few options of bit widths, we need a specific operation to take only the bits corresponding to the table range. The following code is a basic oscillator that progresses in steps *s*, which is once computed from frequency *f*, sampling rate *r* and number of fixed-point binary digits *d* for a table size that is power of two.

```
phase = phase + s;
output = sineTable[(t>>d) & tableSize-1];
```

To implement an additive organ with five octaves and nine harmonics as it was done in the traditional Hammond organs, we need in principle only 96 oscillators, covering the 61 fundamentals plus the difference between first and last harmonic possible for the last note, which is 35 semitones in our implementation, and 30 in actual Hammonds. Therefore, oscillator's phase and output are vectors that needed to be indexed. Considering the ARM processor in question, each oscillator as implemented above needs around 8 cycles to be computed, and it is very reasonable to implement all 96 oscillators with a comfortable sampling rate.

The Hammond company has optimized its electro-mechanical organ by observing that the same oscillator which is a fundamental of one key pressed is the harmonic of another note that we want to play, and they created a matrix of contacts that separate the harmonic series in different buses. In this way, an oscillator goes to one bus as a fundamental, and to several others as harmonics, each bus having the n^{th} harmonic of a series of keys pressed. The buses go in the end to an early form of mixer, where the user selects with which level it should add each harmonic, therefore shaping the sound by selecting the harmonic series. The developed architecture was key to make that instrument viable, as it minimized the number of oscillators and allowed a simple control to act on the harmonics of all notes being played.

If we try to implement this in software for the Arduino Due, however, we quickly realize that there are too many operations to perform. Each one of the nine contacts under each key would need to add the output of one oscillator to its respective bus. That alone would account for 9 * 61 = 549 additions per sample rate, each one taking many cycles, as we need to iterate over them, index, identify if the contact is on or off, find/access the correct oscillator and finally make the addition. If we consider something like 8 to 10 cycles for each of these operations, and a 44.1 KHz sampling rate, we get a total of between 193 and 260 millions of cycles per second, way beyond the 84MHz clock of the Arduino Due, and this is just to sum the output of the oscillators to the harmonic buses.

A smaller sample rate helps only a little, but is not able to solve the problem. Implementing less than full polyphony does not help too much either, as we need many oscillators to generate the needed harmonics, and sparsity would only complicate control and updating operations, probably being even slower for a reasonable polyphony. In [Teichman 2011] the author tries to achieve a very ambitiuos goal of implementing a polyphonic Hammond simulation on the more limited 8-bit Arduino Uno. For that, it keeps a list of active tonewheels, implementing variable polyphony, which depends on the drawbar settings. As a consequence, it can play more notes with

simple harmonics or just a few notes when more complex registrations are set. That, coupled with the limited dynamic ranges that come from all the needed mixing, makes such implementation not usable as a real instrument. In this work we seek to reach a playable instrument with full polyphony. What we need is a completely different architecture, which is optimized for the software domain, instead of the one developed for the legendary Hammond organs, and we shall present it in the next section.

3.2. An architecture for efficient implementation of additive synthesis

In this section we present an architecture that allows us to compute all oscillators and harmonics as in the original Hammond organs, with full polyphony, under the constraints of the Arduino Due platform running at 84MHz. Instead of adding the sound of oscillators to harmonic buses, we investigated the possibility of accumulating the desired intensity needed from each oscillator, and then apply these intensities at each oscillator's outputs, finally summing all of them together. Considering a table of 1024 positions for the sine function and 14 bits of fixed-point precision for the PA, the code for the oscillators (gears) then becomes:

```
for (gear=0; gear<96; ++gear)
  {
  phase[gear] = phase[gear] + steps[gear];
  output +=  wave[(phase[gear]>>14) & 1023] * intensity[gear];
  }
```

This might look subtle at first, but there is a fundamental difference in the proposed new model. The accumulated values of intensity are control parameters, slowly varying data that is changed only when the user changes the sound, and they can also be slowly updated in the software implementation, whereas in the architecture directly derived from the mechanical implementation, it is the sound that is added, what needs to be recomputed at each sample step.

This was apparently already used in [Teichman 2011], but we find no other reference systematically describing such approach. There is a paper entitled “Computationally Efficient Hammond Organ Synthesis” [Pekonen, 2011], with the corresponding code and sound samples available. The article provides a comprehensive model for the instrument, but regardless of the title, it does not make any assessment on efficiency for computation or implementation. The software implementation of organ simulations developed for commercial products are not open-source, and it is difficult to determine if the same technique is already employed. Therefore, we consider this architecture as one significant contribution of this paper, together with the other technical considerations, and the statement that complex instruments are possible with the Arduino Due.

At this time, it is important to better define the different levels of timing needed to operate a digital instrument, as described in the next section.

3.2. Time-critical, time-accurate and Housekeeping tasks

There are three levels of timing needed to appropriately implement a real-time instrument. At the core of the process lies the sound generation which must be

performed at each sample interval. In our case, this includes basically the code provided above for all oscillators, and a few other instructions that will be identified later. This is the most CPU-intensive task, and the bottleneck for sound synthesis on an embedded processor. In this work, we firstly determine that it is indeed possible to implement a Hammond-like organ with nine harmonics and full 61-note polyphony. Secondly, in order to do that, we identified that a few adaptations are necessary, considering the number of bits involved in the operations of phase accumulation and output mix.

In a first attempt of implementing the oscillators we could only compute less than half of them at a conventional sampling rate of 48 KHz, what indicates that it would not be possible to achieve full polyphony and compute all harmonics at such rates. But as the organ sound is based on pure sine waves, and no other high frequencies are necessary, we lowered the sampling rate to 24 KHz. According to the Nyquist theorem, that is enough to compute frequencies of up to around 11 KHz. The highest needed frequency for the Hammond's 91th oscillator is around 6 KHz, and for our 96 oscillators is 8 KHz. Based on the maximum required frequencies of 8 or 6 KHz, it would be still possible to lower the sampling rate further. But on the other hand there are some practical considerations that come into account. As we shall address ahead, one of the quality external DAC chips we wish to use has a lower limit on sampling rate of 20 KHz, and therefore it is not good to impose a limit on our implementation that prevents it to use such converters.

There is, however, another significant optimization possible in the code above. Considering that the code inside the loop is very short, estimated to take at around 8 to 10 cycles to execute, the overhead imposed by the loop control is considerable. The loop control has to increment, compare and jump back at each iteration, 96 times at each sample interval, and as such, is consuming a considerable amount of the time. To solve this, we employ the known optimization technique of loop unrolling. Instead of completely eliminating the loop, however, we iterate over the 12 semitones, and for each of them we generate all the corresponding oscillators for the 8 octaves needed.

The second level of timing priority will be used to compute modulation. For an organ instrument, this will be used to implement tremolo, vibrato or rotary speaker effects. This activity is slower. It only provides updates to values used in the core sound generation. Such updates must be time-accurate, not variable or dependent on the work load resulting from input/output operations, for instance, but need not be too fast. In our implementation, we choose to use 1000 Hz as the timer for such activities. This is enough to compute quick and smooth modulation parameters. The code in this part still needs to be short, however, so as to not interfere on the critical code generation, as it is only 24 times slower.

The last part of the code consists of all other housekeeping activities, including I/O, which can be slightly delayed without any other consequences. The implementation must ensure that there is enough time to perform this frequently, e.g., that the core audio generation is not so close to eating up all CPU time that the housekeeping code hardly advances. That is particularly important for keyboard scanning or MIDI input, but is not hard to achieve. In our implementation, MIDI processing is performed by the *Arduino MIDI Library*, of [Best 2011].

Besides keyboard scan and MIDI input, this part encompasses other controller's readings, as well as all the parameter updates needed to implement the new input and

controller's settings. Care should be taken not to generate unsynchronized updating problems or inconsistencies. The main example is the computation of intensity for each oscillator. It is defined as the sum of all the individual intensities desired for a given oscillator, coming from different keys pressed that require it as some harmonic. As such, the intensity values should be all reset and iteratively added up. This code and intermediate values should not interfere and be visible to the core sound generation. But they can be instantly updated at any time after being accumulated. So, the housekeeping code operates on separate arrays for resetting and accumulating those values, and then copy them to the arrays used by the sound generation in a separate loop, both of them in this slow part of the code. That same procedure is used for other parameters where the same problem arises.

4. Using an External DAC

For the development of this project we initially used the integrated 12-bit DACs as audio outputs, but after some playing tests, we concluded that their inherent noise was too high to be used for any professional application. That might be a particular problem with the specific unit we were using, or may be compensated somehow, but in general it is expected that the noise generated on an integrated processor would severely influence a generic DAC in a design that is not targeted to high quality audio applications.

To cope with a significant amount of noise, the generated signal could be programmed to be very high at all times, but this is not easy to accomplish with fixed-point arithmetic, and is not even natural for the additive synthesis of an organ sound. In other terms, every time we add signals together, we need to increase the number of bits that represent them, which is a typical problem of fixed-point arithmetic in DSPs. And as we have a provision to add a lot of notes and harmonics, already translated into oscillators and intensities, we must have headroom for all this, even though it will be rarely fulfilled. Most of the times, the system will be outputting much smaller signals. That means that one or two nodes being played with few harmonics will be ever very close to the noise levels.

A better solution for serious audio works in the platform is to abandon the embedded DACs and make the Arduino work with external converters that have higher resolution and smaller noise figures. Fortunately, the SAM3X8E chip has an integrated I2S communication module. This protocol is used by many converters to serially communicate stereo audio data of up to 32-bit words per channel. As it was already mentioned in the introduction, we have only identified one successful report of using the protocol to connect an external DAC for Hi-Fi sound generation with the Due [Delsauce 2013], and it operated in slave mode. Slave-mode operation requires an external clock, what can be derived from digital audio reception, but is not natural for sound synthesis.

For our project we modified the available libraries based on the SAM3X8E chip documentation and successfully made it communicate in master-mode with the simple 16-bit TDA1543 DAC chip at the desired 24 KHz sampling frequency. In the master-mode the Arduino processor generates the clock, using its internal programmable timer mechanisms. This internal timing will dispatch the communication code, which must be perfectly synchronized with the audio generation. Therefore, for this option to work, the core generation cannot be called from any independent clock or timer, but instead from the digital audio serial communication itself. The I2S serial communication can

generate interrupts to accomplish such tasks of reading and writing of audio data. We need to use one of such interrupts to synchronize audio generation, or in fact, to call it.

The I2S protocol, as S/PDIF and many other serial digital audio protocols, transmits two channels on the same data lines, implementing stereo sound in each connection. For that, each time frame is composed of two parts, one for each channel. The I2S module embedded on the SAM3X8E chip generates an interrupt for each of these parts, as it needs the next sample, be it from the left or the right channel. Apparently, it cannot be programmed to generate a single interrupt at the end of each frame. As a consequence, we have to split the sound generation for our organ synthesizer in two parts. Otherwise, the sound generation code would have not been finished executing when the next interrupt for I2S communication is called, what makes the processor to halt.

The split code works as follows. It identifies with a flag if it is being called for the left or right channel, and computes the first or the second half of oscillators respectively. In the first half it resets the output to zero, to start accumulating the instantaneous amplitude of the wave. At the end of the second part, it updates an output buffer with the final wave amplitude, as a separate variable, that can be used on either half to output the same mono-aural sound, or processed differently to implement additional sound effects as described in the next section.

5. Audio Effects

For a pleasant sound, a synthesizer typically needs some sort of effect, able to generate some kind of modulation and ambience to a basic raw sound. For the ambience, the first simple thing to try is a delay line with feedback, which partially sustain played notes after they are released. Thanks to the Due's 96 KBytes of SRAM, a long delay line can be internally implemented. One second of delay at the 24 KHz sampling rate for 16 bit samples will take 48 KBytes, or half the RAM memory. This is perfectly acceptable, as there are not other memory hungry processes needed for this synthesizer. In our implementation, we used a slightly smaller total delay of about half a second. When combined with the feedback, it can produce longer sustained sounds. The code to compute the delay with feedback is simply:

```
delayLine[delayLast] = (delayLine[delayLast] + rawOutput) >> 2;
delayLast++;
if (delayLast == DELAY_SIZE)
   delayLast = 0;
```

Another typical effect present on the Hammond organ and some other instruments is the key click, an initial hit noise that is produced whenever each key is pressed. Despite the fact that this is a much sough after feature of the famous Hammond sound, it is in fact an intrinsic implementation problem. When a running oscillator is instantly activated electronically, its sound wave becomes present at an arbitrary phase position, causing an abrupt level change from 0 to its value at the current phase. In terms of frequency spectrum, this fast transition corresponds to an instantaneous appearance of wide spectrum energy, perceived as noise, or click. The Hammond company has attempted many ways of reducing those transients, but never eliminated

them completely. On the contrary, as it consists of an attack sound that is present in many other acoustic instruments, it complemented the pure organ sounds and became a desirable characteristic of those organs. In our software implementation, a filter can be employed to reduce those transients to some extent, which are more pronounced in the digital domain. We tested a very simple filter, which slightly reduces the strong initial clicks, but ended up choosing not to use it, as it is much simpler to implement an RC filter externally if needed, letting our implementation to output those transitions untouched.

Other basic sound effects for organs include tremolo (here defined as a repetitive variation in amplitude) and vibrato (regular variation in pitch). Both can be easily implemented on the Due, on top of all the features already presented. Both monophonic or polyphonic (independent) modulation on the oscillator's frequencies can be implemented by slowly and slightly changing the *s* (steps) parameters for each of them, which control the speed of the oscillators. For typical Low-Frequency Modulation, this falls in the time-accurate tasks: they must be controlled accurately, but not computed at each sample interval. In order to avoid numerical cumulative errors, however, it is advised to keep original copies of the target (center) values for steps, and compute oscillation around them in absolute form, not incrementally. The amplitude of the whole sound output or individual oscillators can also be changed in a similar manner, controlled at the time-accurate section, computed at the housekeeping section, and regularly used in the core audio generation.

As a final touch, we wanted to implement some sort of effect that resembles the equally famous Leslie Speakers, or formally rotary speaker emulation. There are many digital and analog emulations of such speakers, and even small mechanical models with real rotating horns and microphones enclosed in soundproof cabinets. They are employed by many musicians that seek the most authentic and live sound of the traditional rotating horns of Leslie speakers. There are as well many works that analyze and provide models to digitally compute a similar sound, such as [Penniman 2014]. But any attempt to more closely emulate it would stress the limits of our simple embedded target processor. As stated in the beginning, however, our objective was not to make perfect emulations of a real instrument, but provide a consistent and complete instrument that demonstrates its potential of being playable and enjoyable. For that, we just need to provide a modulation effect that combines tremolo, vibrato, stereo imaging and the liveness that results from the rotating horns speeding up and slowing down.

Therefore, we have implemented a combined change in amplitude, pitch, and stereo position, controlled between two speeds, slow and fast, with the accompanying transitions between them. Transition times are different for bass and treble, so that treble speeds up and slows down more quickly than bass. The stereo position is computed by taking the tremolo modulation and making it 90 degrees out of phase between left and right. The change on vibrato (pitch modulation) is not dependent on the stereo position, as it needs to be computed on the unique PA steps, one for each oscillator. Surprisingly, this simple model of a rotary speaker, despite using only a few lines of code, provides the desired life and dynamic expected from a usable instrument.

6. Discussion and Perspectives

In this work we have already determined that an additive organ with full polyphony and nine harmonics per key can be efficiently implemented and operated on the readily available Arduino Due platform. The implementation includes high quality sound output with external DAC, keyboard and parameter control, and attractive audio effects. This is an interesting statement, as previously only professionals could design complex hardware and musical instruments with such capability, a task that is both time consuming and expensive. It opens up a lot of opportunities for independent developers, artists, DIYers, to access software models such as this one and run on small Arduinos for serious applications.

It is indeed desirable to continue this trend and check how to implement other known synthesis techniques with the platform. The author is already in collaboration for the development of an FM synthesizer with characteristics similar to other iconic machines.

But there is much more, and we would like to take this time for a few prospective notes. The first thing to note is that the look-up table based approach for oscillator implementation, although selected for being the most efficient way of implementing the sine function, is much more powerful, and should not be limited to that. We can implement any desirable wave shape in such table, provided that it is used accordingly. Complex waves like saw tooth would not be too pleasant perhaps with additive synthesis. But how about samples of real pipe organs? With 512 KBytes of persistent Flash memory on the Arduino Due, it is perfectly capable of storing many samples from different pipes, or from *toneweels* in an actual Hammond, among others.

It can easily deal with multi-samples as well, which would probably be needed to better reproduce sampled instruments. In fact, the more we unroll the loop to access specific tables for each note range, the faster is the code. The wave tables can also be dynamically changed, in two ways. First, the code can use a pointer, which can be quickly changed to access different tables at the user's command. But the table can also be updated in RAM, creating evolving sound waves with unlimited consequences, and it does not affect too much the code speed, as those changes can be slowly computed on the housekeeping part of the code.

So, in the end there seems to be some exciting possibilities to test with this platform for sound synthesis. We are interested in trying some of them in the near future, as well as encourage others to try it as well.

7. Conclusions

In this paper we presented an efficient architecture for implementing an organ by classic harmonic addition, with full 5-octave polyphony, on the Arduino Due platform. We demonstrate that a playable and complete instrument implementation is possible with this widely available and easily programmable system, and described the main strategies employed to achieve that. The instrument included all needed controllers, keyboard scan and MIDI input, delay and a simple rotary speaker effect, while serially communicating with high-quality 16-bit or 24-bit external DACs. It demonstrates the power of recent microcontrollers in performing complex music and audio duties and opens up many other opportunities for ubiquitous music applications.

References

Barrass, Tim. (2014) “Mozzi - sound synthesis library for Arduino”. <http://sensorium.github.io/Mozzi/>

Best, François (2011) Arduino MIDI Library. <https://github.com/FortySevenEffects/arduino_midi_library/tree/dev>

Chowning, John. (1973). “The Synthesis of Complex Audio Spectra by Means of Frequency Modulation”. Journal of the Audio Engineering Society 21 (7). 526-534.

Delsauce. (2013) “ArduinoDueHiFi - An I2S audio codec driver library for the Arduino Due board”. <https://github.com/delsauce/ArduinoDueHiFi>

Ghassaei, Amanda. (2013) “Arduino Audio Output”. <http://www.instructables.com/id/Arduino-Audio-Output/>.

Goodeve, Pete. (2009) “Rotor Organ - A Csound simulation oriented to live playing”. <http://www.goodeveca.net/RotorOrgan/ToneWheelSpec.html>

Hammond Organ Company. (1984) “The Hammond Story - Fifty years of musical excellence”. 50th Anniversary 1934-1984.

Hammond Organ Company. (1966) “When Electrons Sing: The Story of Hammond Organ Company”. <https://books.google.com.br/books?id=cXOPGQAACAAJ>

Pekonen, J., Pihlajamäki, T. and Välimäki, V. (2011) “Computationally Efficient Hammond Organ Synthesis”. In: 14th International Conference on Digital Audio Effect (DAFx-11), Paris, France, September 19–23, 2011.

Penniman, Ross. (2014) “A Rotary Speaker Modeling Plug-In”. In: Will Pirkle. App Notes. App Note AN-9. <http://www.willpirkle.com/app-notes/> Access on 05/03/2015.

Teichman, Pete. (2011) “Roto - Hammond Organ Simulation”. <https://teichman.org/blog/2011/05/roto.html>

Wiltshire, Tom. (2008) “The Hammond Organ”. In: Electric Druid - Synth DIY pages. <http://www.electricdruid.net/?page=info.hammond>