Real time digital audio processing using Arduino

André Jucovsky Bianchi
Computer Science Department
University of São Paulo
ajb@ime.usp.br

Marcelo Queiroz
Computer Science Department
University of São Paulo
mqz@ime.usp.br

ABSTRACT

In the search for low-cost, highly available devices for real time audio processing for scientific or artistic purposes, the Arduino platform comes in as a handy alternative for a chordless, versatile audio processor. Despite the fact that Arduinos are generally used for controlling and interfacing with other devices, its built-in ADC/DAC allows for capturing and emitting raw audio signals with very specific constraints. In this work we dive into the microcontroller's structure to understand what can be done and what are the limits of the platform when working with real time digital signal processing. We evaluate the behaviour of some common DSP algorithms and expose limitations and possibilities of using the platform in this context.

1. INTRODUCTION

Arduino is the name of a hardware and software project started in 2005 which aims to simplify the interface of electric-electronic devices with a microcontroller [1]. It evolved from the *Processing* software IDE [1] (2001) and the *Wiring* software and hardware prototyping platform [2] (2003). Hardware, software and documentation designs are published under free licenses (Creative Commons BY-SA 2.5, GPL/LGPL and CC BY-SA 3.0, respectively) and a large community has grown to provide code and support for newcomers. Nowadays, many Arduino hardware designs are available and range from more limited 8-bit microcontrollers to fully featured 32-bit ARM CPUs. Besides, other advantages of Arduino for academic and artistic use are its mobility (because of its low power needs and possibility of running on batteries for hours, if not days depending on the use), expandability (because of its standardized interface for attaching so called hardware *shields*) and price (selling for under 20 US dollars online).

Despite all these advantages, the Arduino platform has a somewhat limited processing power when compared to standard processors available in the market, as for example DSP chips such as Analog Device's Blackfin 32-bit RISC processors [3] and FPGA-based processors such as Xilinx Virtex-7 family [4]. Research and industry advances have led to optimized computational performance and power consumption for these platforms [2], but we could not find a thorough examination of the use of a low-tech device such as the Arduino.

[1] http://www.processing.org/
[2] http://wiring.org.co/

In this work, we aim to systematically expose the microcontroller-based Arduino platforms' possibilities for carrying real time digital audio processing tasks so there can be more accurate elements to be taken into account when making the choice for a platform. Code examples can be downloaded from the IME/USP Computer Music Group webpage [5].

1.1 Related work

Arduino has been experimentally used as a real time audio processor for sampling audio and control signals with an effective rate of 15.125 KHz [3], which provided the base for our investigation. Also, an ALSA audio driver was implemented to use the Arduino Duemilanove [4] as a full-duplex, mono, 8-bit 44.1 KHz sound card under GNU/Linux.

2. METHODS

In order to meet the needs for real time audio processing, the microcontroller has to be tweaked so we can capture, process and output analog audio. Each of these tasks can be performed in a variety of ways, and for this examination we chose to go with the basic functionalities of the platform.

In this investigation, we used an Arduino Duemilanove with an ATmega328P microcontroller from Atmel, a very modest version of the platform. It has an 8-bit RISC central processor, operates with a base frequency of 16 MHz, and has memory capacity of 32 KB for program storage and 2 KB for random access [5]. From now on, whenever we refer to *the microcontroller*, we are in fact talking about this specific model from this specific manufacturer.

2.1 Microcontroller's elements

To be able to know how to configure the platform to suit our needs, a general understanding of the inner workings of a microcontroller is needed. The Atmel megaAVR series

[3] http://www.analog.com/en/processors-dsp/blackfin/products/index.html
[4] http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm
[5] http://compmus.ime.usp.br/en/arduino

microcontroller is comprised of several components, some of which are fundamental for our investigation and so will be briefly covered in this section.

2.1.1 Clocks

Many *clocks* provide the frequencies in which the different parts of the microcontroller work. They are basically either emitters or dividers of square wave signals that provide the frequency of operation of the CPU, the ADC, the memory access and other components of the microcontroller. Possible sources of clock frequencies are crystal and RC oscillators.

A useful concept associated with clocks is the one of a *prescaler*. Prescalers are dividers for clock frequencies that either actually lower the frequency of a clock or at least trigger specific interrupts on a (power of two) fraction of a clock's frequency.

The *system clock* provides the system's base frequency of operation. Other important clocks are the *I/O clock* and the *ADC clock* used for feeding a frequency to most of the input/output mechanisms. It is possible to choose which clock will feed a frequency to some parts of the system, as well as select prescaler values independently. In our study, we make use of the *timer clock* prescaler to control the PWM frequency that drives our DSP mechanism, as we will see in Section 2.3.

2.1.2 Registers and interrupts

The microcontroller's CPU is comprised of an arithmetic logic unit that works with 32 *registers* – portions of memory that provide data for computation as well as determine the execution flow of the program. An *interrupt* is an attempt of deviation from the current execution flow that can be triggered by a variety of events in the system, usually by setting reference values on specific registers.

In our case, interrupts are of extreme value as they are the low level structures that allow us to execute code with a somewhat fixed frequency (at least if we assume that the clock frequencies are indeed constant in relation with real time).

2.1.3 Timers/counters

A *timer*, or *counter*, is a register whose value is automatically incremented according to a specific clock. When a counter hits its maximum value it is reset to zero and signals an overflow interrupt, which may cause a certain function to be called.

Timers are important in the context of DSP because they provide a natural way to perform many of the DSP chain tasks, as for example to periodically launch the input signal sampling function (that fills the input buffer) and to emit a PWM square wave which, after analog low-pass filtering (through an integrator), corresponds to a smooth analog signal. The ATmega328P has two 8 bit counters and one 16 bit counter, each having different sets of features but all being capable of doing PWM.

2.1.4 Input and output pins

Microcontrollers can receive and emit digital signal through *I/O pins*, which in the case of the Arduino board are conveniently mounted in such a way that it is easy to plug other components and boards. These pins are read from and written to according to frequencies governed by different clocks (I/O, ADC and others).

In principle, the microcontroller pins are designed to work with binary signals represented by two different voltages (0 V and 5 V with a threshold value to account for small deviations). Despite that, I/O pins come equipped with handy mechanisms for sampling band limited input signals whose voltages vary between the reference extremes, and also for generating waveforms that, after being filtered, output varying signals of the same nature. These mechanisms are, respectively, the analog-to-digital converter (ADC) and the pulse-width modulation (PWM), which will be seen in the next sections.

2.1.5 Memory

The microcontroller has 3 manageable memory spaces for storing the program and working data, and the following table summarizes the different characteristics and purposes for each type of memory:

Type	Size (KB)	Data persistency	Write time (clock ticks)	Endurance (write/erase cycles)
Flash	32	yes	1	10,000
SRAM	2	no	2	n/a
EEPROM	1	yes	30	100,000

Usually, the Flash memory stores the program, the SRAM memory stores volatile data used along the computation, and the EEPROM is used for longer-term storage between working sessions. Notice that the amount of SRAM memory represents a hard limit for many DSP algorithms. A 512 point lookup table filled with precalculated sinewave bytes, for example, represents 25% of all available working space. Thus, it might be interesting to store hardcoded data in the program memory whenever possible if memory working space is lacking.

2.2 Audio in: ADC

Data can flow into the microcontroller in a variety of ways, the most basic being embedded mechanisms for digital serial communication and analog-to-digital conversion using the input pins. The former mechanism can feed digital data directly into memory, while the latter can either read 1 bit from an input pin (as explained in the last section) or sample an analog value between the reference voltages using 8 or 10 bits resolution.

Rather than providing the microcontroller with digital data, our setup uses the embedded analog-to-digital conversion to sample an audio signal using the microcontroller pins' ADC mechanism. This choice was made so the signal can be directly connected to the microcontroller (i.e. no external device has to be used for sampling) and we can study the device's performance taking into account this crucial step in the digital audio processing chain.

The ADC uses a *Sample and Hold* circuit that holds the input voltage at a constant level until the end of the conversion. This fixed voltage is then successively compared with reference voltages to obtain a 10 bit approximation. If a faster conversion is desired, precision can be sacrificed and the first 8 bits can be read before the last 2 are computed. Conversion time takes between 13 and 250 μs, depending on several configuration parameters that influence the precision of the result.

As noted before, the ADC mechanism has a dedicated clock to ensure conversion can occur independently of other microcontroller parts. Also, the mechanism can be triggered manually (on demand) or automatically (a new conversion starts as soon as the last one has finished).

2.3 Audio out: PWM

Once the input signal has been sampled and processed, one way to convert it back to analog is to use the embedded *pulse-width modulation* (PWM) mechanism that is available in some of the output pins of the microcontroller, followed by an analog filtering stage. A PWM wave encodes a determined value in the width of a square pulse. In order to do this, it defines a *duty cycle* as the percentage of time that the square wave has its maximum value in relation to the total time between square pulses (see Figure 1). The encoding of a value x ranging from X_1 to X_2 is just the enforcement of a duty cycle with a percentage equal to $\frac{x-X_1}{X_2-X_1}$.

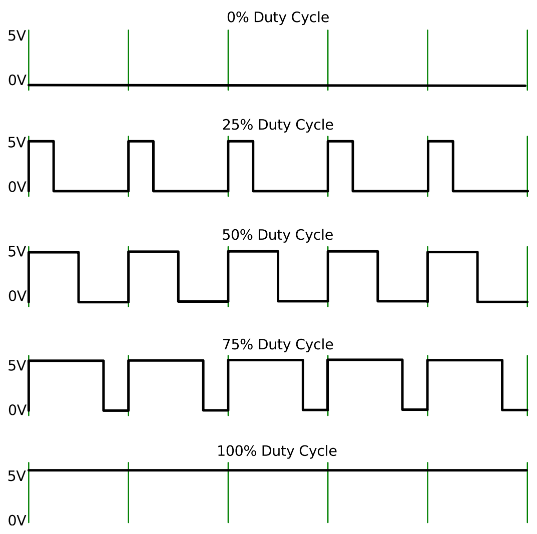

Figure 1. Examples of PWM waves with different duty cycles. The left alignment of the waves corresponds to the Fast PWM mode.

The final analog filtering stage is needed to remove high frequency components present in the square wave spectrum to reconstruct a band limited signal. In our case, this filtering is made from a simple RC integrator circuit that stands between the output pin and a normal speaker.

The PWM mechanism can operate in different modes which vary according to how the reference value to be encoded relates with a counter's signal to generate the output values of the modulated wave. In *Fast PWM* mode, the output signal is set to 1 in the beginning of the cycle and becomes 0 whenever the reference value becomes smaller than the counter value (see Figure 2). This mode has the disadvantage of outputting the square pulses aligned to the left of the PWM cycle, and so the *Phase correct* mode is available to solve this problem at the expense of cutting the signal generation frequency in half. It works by making the counter count back to zero instead of being reset when it hits its maximum value.

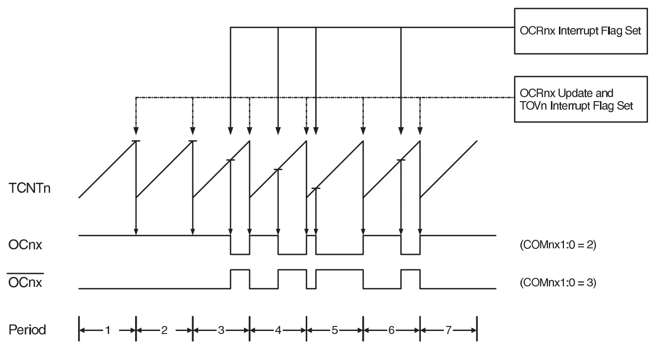

Figure 2. Time evolution of register values in the PWM mechanism. `TCNTn` is the value of the counter and `OCnx` is the value of the output pin. Note how changes in the reference value determine the duty cycle on each wave period.

The output frequency of the PWM signal is a function of the clock selected to be used as input for the counter, the counter prescaler value, the size of the counter (in bits) and the PWM mode. For a b bits counter with input clock of f_{clock} Hz and prescaler value of p, an output pin configured to operate in fast PWM mode overflows with a frequency of $\frac{f_{clock}}{p \times 2^b}$ Hz. This provides us with a way to output the processed signal while using the same infrastructure to schedule periodic actions, such as querying for ADC values and signaling that blocks of samples are ready to be processed.

Also notice that the counter size determines the output signal resolution, as the duty cycles of the square waves correspond to the ratio between the current counter value and its maximum possible value. We will see more about parameters choice for PWM on section 2.5.2.

2.4 Real time processing

The main constraint in real time DSP is, of course, the amount of time available for the computation of output samples: they must be ready to be consumed by the playback hardware or else glitches and other unwanted artifacts will possibly be introduced in the signal. One round of sample analysis, processing and calculation of a new sample is called a *DSP sample cycle*. Many algorithms, though, operate in blocks of samples, consuming and producing a whole block of samples in each round. If the DSP block has N samples and the sample rate is R Hz, then the *DSP block cycle period* is given by $T_{DSP} = \frac{N}{R}$ seconds.

In order to implement this behaviour in the microcontroller, we have to find a way to (1) accumulate input samples in a buffer, (2) schedule a periodic call to a function that will process the samples in this buffer, and (3) output modified samples in a timely fashion. Components are at hand: ADC for reading the input signal, counters and their interrupts for running periodic tasks, and PWM for outputting the resulting signal. In addition, the Arduino library provides a `loop()` function that is called repeatedly which we can use to process the block of samples when it becomes available.

As we saw in section 2.3, the PWM mechanism provides an overflow interrupt frequency that may be used to schedule a function for periodic execution. In our setup we use this mechanism to periodically read samples from the ADC mechanism and accumulate them in an input buffer, while also writing the computed samples from the last DSP cycle to the PWM output buffer. In this same function, whenever the buffer is full and ready to be processed, a flag is set and the `loop()` function is released to work on the samples.

Note that for some critical applications, the term "real time execution" might mean that the application should be interrupted whenever its time for computation is up. In the case of real time digital audio, if no output sample could be computed before the output hardware tries to read it, audible artifacts may be unavoidable had the computation been interrupted or not. Thus, our approach concentrates on measuring the time taken by certain algorithms and comparing it with the DSP cycle period, and does not account for what happens should the time not be sufficient.

2.5 Implementation

Putting all the elements together is a matter of choosing the right parameters for configuring different parts of the microcontroller.

2.5.1 ADC parameters

ADC conversion takes about 14.5 ADC clock ticks, including sample-and-hold time. If the CPU clock frequency is 16 MHz and the ADC prescaler has a value of p, then the ADC clock period is $p/16$ and the conversion period is then $T_{\text{conv}} = (14.5 \times p)/16$. Below we can see a table with the theoretical values for the conversion period T_{conv} for all prescaler values available and also the results $\tilde{T}_{\text{conv}}$ of measured conversion times using each prescaler value. Also depicted in the table are the measured conversion frequencies $\tilde{f}_{\text{conv}} = 1/\tilde{T}_{\text{conv}}$.

ADC prescaler	T_{conv} (μs)	$\tilde{T}_{\text{conv}}$ (μs)	$\tilde{f}_{\text{conv}}$ ($\approx$KHz)
2	1.8125	12.61	79.302
4	3.625	16.06	62.266
8	7.25	19.76	50.607
16	14.5	20.52	48.732
32	29	34.80	28.735
64	58	67.89	14.729
128	116	114.85	8.707

These measurements were made using the `micros()` function of the Arduino library API, which has a resolution of about 4 μs. This might explain part of the deviation of measured values from the expected values for lower values of prescaler. 8 bit approximation was used, and for obtaining a 10 bit approximation we can expect an overhead of about 25% in conversion time.

It is important to note that the choice for ADC prescaler value limits the sampling rate of the input signal. As our setup uses a counter's overflow interrupt to obtain samples from the ADC mechanism, the ADC conversion period must be smaller than the the PWM's cycle period. Any prescaler choice that leads to a frequency higher than the PWM's overflow interrupt frequency is valid, but the lower the prescaler value the lower the quality of the conversion.

2.5.2 PWM

From Section 2.3 we can see that in a 16 MHz CPU, an 8 bit counter with prescaler value of p has an overflow interrupt frequency of $f_{\text{overflow}} = 10^6/(p \times 2^4)$ Hz. Below we can see a table with the overflow interrupt frequency for all possible values of prescaler:

PWM prescaler	f_{incr} (KHz)	f_{overflow} (Hz)
1	16.000	62500
8	2.000	7812
32	500	1953
64	250	976
128	125	488
256	62,5	244
1024	15,625	61

The choice of PWM and ADC prescaler values determine directly the sampling rate of our DSP system. If we set the ADC prescaler in a way that the ADC conversion period is smaller than the PWM overflow interrupt period and synchronize reads from the input with writes to the output, then the PWM overflow interrupt frequency becomes the DSP system's sample rate. We will see this with more details in the next section.

For the PWM mechanism, we chose to use Fast PWM mode on an 8-bit counter with prescaler value of 1. That would give us a sample rate of 62500 Hz, which is enough for representing the audible spectrum. Nevertheless, if we need more time to compute we may artificially lower the frequency by only executing the sampling/outputting bit in a fraction of the interrupts. For our tests, we chose to cut the sample rate in half using the rationale that the payoff of having more time to compute is larger than the one of ensuring we can represent the upper fifth part of the audible spectrum. Therefore, our final choice of sample rate is 31250 Hz, with a sample period of 32 μs.

2.5.3 Putting it all together

Having chosen a value for the PWM counter size and PWM prescaler, we are left with the choice for ADC parameters. As noted, it suffices to choose a value that ensures ADC conversion period is smaller than the desired sample period. We chose to use 8 bit conversion to match the PWM resolution and to provide for a faster conversion time. Also, we chose an ADC prescaler value of 8, with a measured conversion time of 19.76 μs which, when compared with the a sample period of 32 μs ensures that conversion will be finished before the input ADC is queried for the sample.

Below we can see the code for the *interrupt service routine* (ISR) DSP controller function. Variable x is the input buffer, ADCH maps to the ADC register holding the input sample, OCR2A maps to the PWM output register and y is the output buffer. Some of the code is index wizardry and the rest we comment below.

```
// Timer2 Interrupt Service at 62.5 KHz
ISR(TIMER2_OVF_vect) {
  static boolean div = false;
  div = !div; // divide frequency to 31.25 KHz
  if (div){
    // 1. read from ADC input
    x[ind] = ADCH;
    // 2. write to PWM output
    OCR2A = y[(ind-MIN_DELAY)&(BUFFER_SIZE-1)];
    // 3. signal availability of new sample block
    if ((ind & (BLOCK_SIZE - 1)) == 0) {
      rind = (ind-BLOCK_SIZE) & (BUFFER_SIZE-1);
      dsp_block = true;
    }
    // 4. increment read/write buffer index
    ind++;
    ind &= BUFFER_SIZE - 1;
    // 5. start new ADC conversion
    sbi(ADCSRA,ADSC);
  }
}
```

Note that in step 3 we test if the input index is a multiple of the block size and, if it is, we set a read index rind and signal that there is a new DSP block available for calculation. Meanwhile, the loop() function is running concurrently and will eventually catch that signal and start to work on samples. Finally, we increment buffer indexes and perform the call to start a new ADC conversion by calling the sbi() function.

2.6 Benchmarking

We are interested in evaluating the performance of the Arduino board on some common sound processing tasks, in order to gain insight on its real time stream processing capabilities. Note that our interest lies in high-level DSP operations; for instance, we'd prefer to know how many simultaneous sinusoids can be synthesized in real time rather than how many multiplications and additions fit between successive DSP blocks (even though the former follows from the latter).

Some questions arise immediately from the real time constraint:

- What is the maximum amount of DSP operations that can be carried in real time?
- Which implementation details make a difference?

We try to answer these questions by running 3 different DSP algorithms in the microcontroller environment described in the last section. The chosen tasks are additive synthesis, time-domain convolution and FFT computation, and are discussed in the following sections.

2.6.1 Additive synthesis

An additive synthesis is the process of constructing a complex waveform by adding together several basic waveforms (see Figure 3). This technique has been widely used for synthesizing new sounds as well as resynthesizing signals after they have been processed (e.g. via spectral methods).

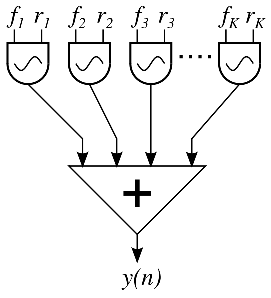

Figure 3. Additive synthesis: many basic oscillators governed by independent phase (f_i) and amplitude (r_i) functions are combined to form a complex signal.

The high level code for a simple additive synthesis can be seen below:

```
for (n = 0; n < N; n++)
{
  angle = 2.0 * M_PI * t;
  y[n] = 0.0;
  for (k = 0; k < numFreqs; k++)
    y[n] += r[k]*sin(f[k] * angle);
  t += 1.0 / SR;
}
```

2.6.2 Time-domain convolution

Frequency-domain multiplication of spectra correspond to time-domain convolution of signals, and such an operation allows for some techniques of frequency filtering. The time-domain implementation of convolution is a widely used technique in many computer music algorithms, being particularly efficient when the filter order N is small. The general scheme can be seen in Figure 4.

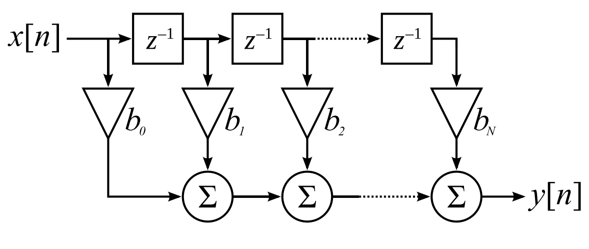

Figure 4. Time-domain convolution: the input signal $x[n]$ is convolved with the filter's impulse response defined by the coefficients b_i to generate the output signal $y[n]$. This is the general scheme for FIR filtering.

The high-level code for a time-domain convolution with a FIR filter of order N is:

```
for (k = 0; k < N; k++)
  y[n] += b[k]*x[n-k];
```

2.6.3 Fast Fourier Transform

The Fast Fourier Transform (FFT) is a clever implementation of the traditional Fourier Transform that brings its complexity down from $O(n^2)$ to $O(n \log(n))$, where n is the number of time-domain digital samples or, equivalently, the number of frequency bins that describe the frequency spectrum of the signal after the Transform computation [6]. The FFT algorithm takes advantage of redundancy and symmetry on intermediary steps of the calculation and is used in many signal processing algorithms. The general scheme of the FFT can be seen in Figure 5.

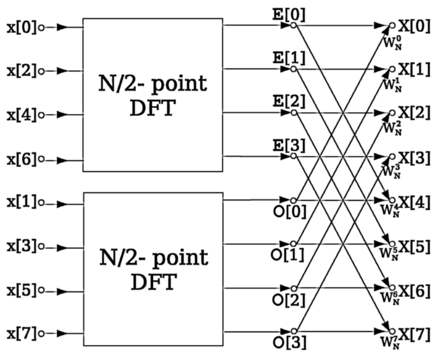

Figure 5. The FFT uses a divide-and-conquer approach and saves intermediate results to accelerate the calculation of a signal spectrum. The figure shows one step of an 8-point FFT calculation and how the results map to frequency bins.

2.6.4 Benchmarking

Each of the algorithms mentioned in the last sections have different computational costs in terms of number of integer and floating-point operations, and quantity/size of memory reads and writes.

In the context of real time audio processing in Arduino, these algorithms bring natural questions regarding feasibility of processing:

- Additive synthesis: what is the maximum number of oscillators that can be used to compute a new waveform in real time?
- Time-domain convolution: what is the maximum length of a filter that can be applied to an audio signal in real time?
- FFT: what is the maximum length of an FFT that can be computed in real time?

3. RESULTS

3.1 Additive synthesis

The first experiment tries to answer the question of how many oscillators can be used when performing real time additive synthesis inside the platform. In the beginning of the DSP cycle, an additive synthesis algorithm is run using a determined number of oscillators and the mean of the synth time is taken over ten million measurements. Block sizes used had 32, 64 and 128 samples (more showed to be unfeasible in real time) and the number of oscillators was increased until the DSP cycle period was exceeded.

The first result has to do with the use of loop structures. Because looping usually requires incrementing and testing a variable in each iteration, the use of one loop structure may have strong influence in the amount of oscillators that can be used in real time for additive synthesis inside the Arduino.

In any DSP algorithm that works over a block of samples there is at least one loop structure, that loops over all samples of the block. This loop could be eliminated at the cost of having to recompile the code every time the length of the block is changed, which is highly inconvenient. Usually more loops will be used, for instance in additive synthesis for summing the result of several oscillators. We investigate the alternative of removing this inner loop, by explicitly writing the sum of oscillators. Figure 6 shows the maximum amount of oscillators feasible in real time by making use of a loop and by making use of inline code. By removing the inner loop we were able to increase from 8 oscillators to 13 or 14 depending on the block size.

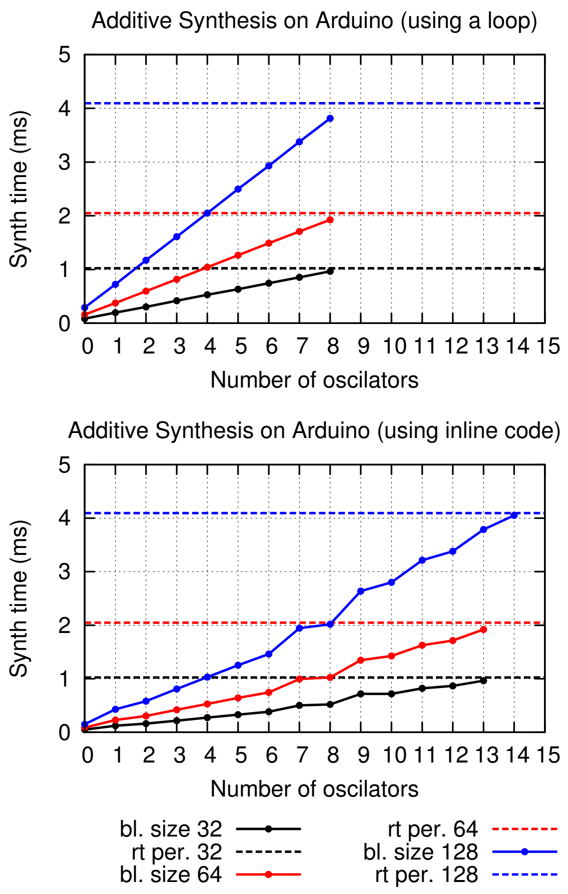

Figure 6. Additive synthesis results using loops (above) and inline code (below).

While implementing this experiment, a first attempt was made using the standard API `sin()` function. As that proved to be unfeasible in real time, we focused on table lookup implementations. At this point we noticed that even the smallest implementation difference can have large impact on the results. Therefore, we decided to test and plot the results for slightly different implementations.

Two parameters are used to calculate the value of each oscillator: phase and amplitude. Phase is handled by updating the index for sine table reads, and then the amplitude has to be multiplied by the value obtained by the lookup. Floating point operations are also extremely expensive in the platform we are using, so we implemented 3 different ways of multiplying the amplitude: (1) by using one integer multiplication and one integer division (2 integer operations), (2) by using only one integer division (1 integer operation), and (3) by using variable bit padding for performing bitwise power of 2 divisions or multiplications. Figure 7 shows the time used by the additive synthesis algorithm using these variants. By making use of lower level operations (that achieve less precise results) and inline coding we were able to raise the number of oscillators from 3 (when using 2 integer operations and a for loop) to 15 (when using a variable pad and inline code).

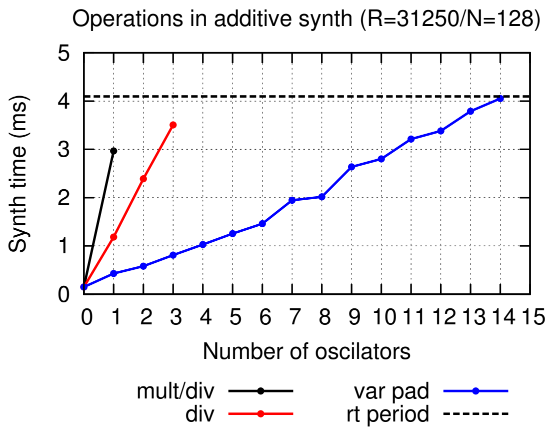

Figure 7. Time taken for additive synthesis algorithm with block size of 128 samples, using different number and kinds of operations and variable number of oscillators.

3.2 Time-domain convolution

Our second experiment tries to clarify what is the maximum size of a FIR filter that can be applied in real time to an input signal by use of time-domain convolution algorithms. Following lessons learned on the first experiment, we implemented the filtering loop using different types of operations for multiplying each coefficient by the sample values: (1) using one integer multiplication and one integer division, (2) using variable pad, and (3) using a constant hardcoded pad. The results for each of these implementations can be seen in Figure 8. This experiment was run with a sample rate of 31250 Hz and block sizes of 32, 64, 128 and 256 samples.

Results again show that small implementation differences make a big difference on computing power. When using integer division, the maximum order obtained for the filter was 1, while by using a variable pad the order raised to 7 and with constant padding we could achieve an order of 13 or 14 depending on the block size.

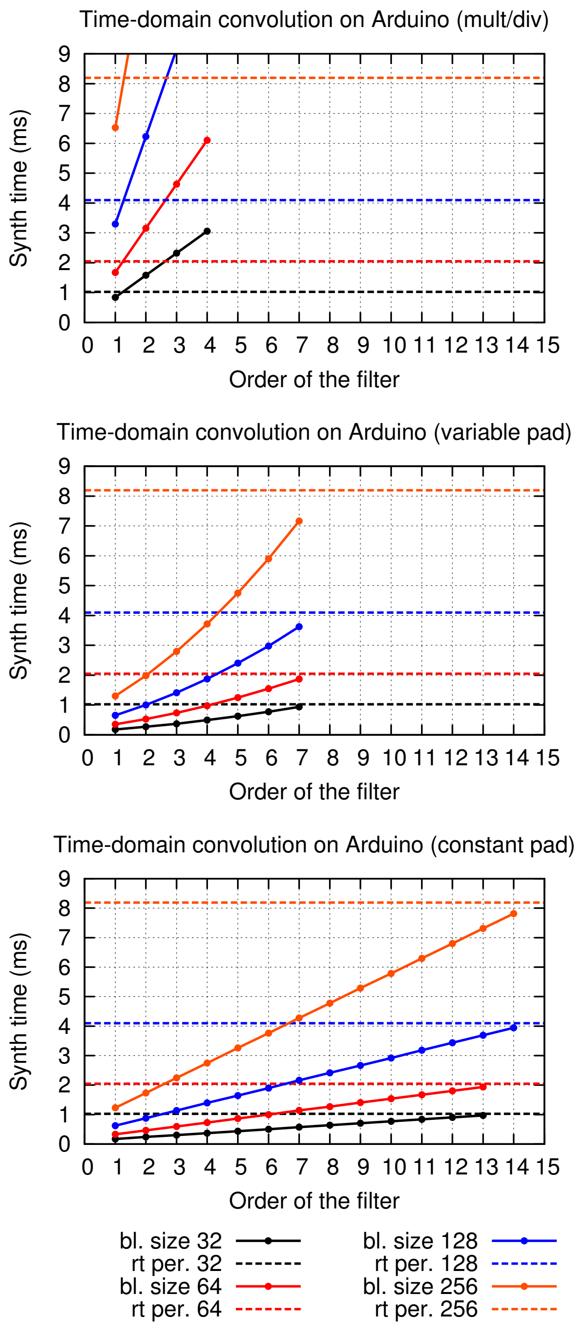

Figure 8. Time-domain convolution using 2 integer operations (top), variable padding (middle) and constant padding (bottom).

3.3 FFT

The third experiment is concerned with the maximum length of an FFT that can be computed in real time inside an Arduino. In this case we chose to evaluate a standard implementation of the FFT without further modifications.

It turned out that calculating an FFT using the same sample rate we used in the other experiments (31250 Hz) was unfeasible, so we had to tweak the microcontroller's parameters to reach a state where we had a longer DSP cycle period for the same amount of samples and the FFT was indeed feasible. By measuring the amount of time taken to compute the FFT given the number of samples, we could determine that the maximum FFT frequency for a 256 samples block is of about 2335 Hz. So by raising the PWM prescaler value to 32, we could reach a sample rate of about 1953 Hz.

Figure 9 shows the FFT analysis time at a sample rate of 1953 Hz for different block sizes and two implementations: using the API `sin()` function and using a lookup table. In this scenario the maximum block size for which an FFT can be computed in real time in our DSP setup in the Arduino is 256 samples. This was expected because we actually forced a sample rate small enough so that the 256 samples FFT was feasible. Note that, even though we can actually perform the FFT for block sizes smaller or equal to 256, there's not much time left for doing anything else with these results. An additive synthesis for reconstructing the signal, for example, is unfeasible as the maximum number of oscillators we could use was 14 (by restricting the type and number of operations), while here we would need the same number of oscillators as the number of samples in the block size.

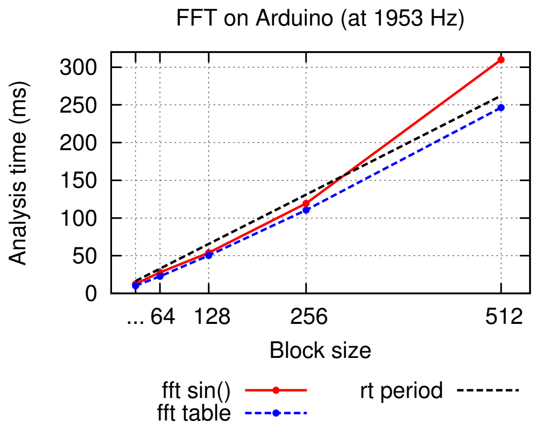

Figure 9. Time taken to compute the Fast Fourier Transform on the Arduino for different block sizes. The red line depicts the implementation using the `sin()` library function and the blue line shows a lookup-table implementation

4. DISCUSSION AND CONCLUSION

From the results of our experiments, it becomes clear that implementation details, such as choice of data type and number and type of operations, make a big difference in the amount and quality of computation, as described in Sections 3.1 and 3.2. Integer multiplication and division, for example, take double the time than integer sum. The amount of loops also proved to make a big difference. In Section 3.1, we nearly doubled the amount of oscillators that can be used in additive synthesis by only substituting one loop with inline code. The mere use of variables also showed to influence performance.

These experiments may serve as illustrations of the type of concern that must be kept in mind when implementing sound processing tasks in Arduino, and also serve as general guidelines for the limitations on the complexity of those tasks when real time functioning is required.

4.1 Future work

There are many possibilities of investigation in the realm of microprocessors like Arduino for real time sound processing, such as:

- Use of 10-bit ADC input and adapting tests for performing 2 byte operations. We should expect each operation to cost much more time because of the 8 bit nature of the processor.
- Determine of the amount of noise introduced in the signal by the ADC sampling/PWM synthesis process.

5. ACKNOWLEDGEMENTS

We would like to thank the members of the Computer Music Group[6] of the University of São Paulo for valuable discussions and contributions. This work has been supported by the funding agencies CAPES and FAPESP (grant 2008/08632-8).

6. REFERENCES

[1] "Arduino homepage," http://www.arduino.cc/, [Online; accessed 12-Jun-2013].

[2] R. Oshana, *DSP for Embedded and Real-Time Systems*. Newnes, 2012.

[3] M. Nawrath, "Arduino realtime audio processing," http://interface.khm.de/index.php/lab/experiments/arduino-realtime-audio-processing/, [Online; accessed 12-Jun-2013].

[4] S. Dimitrov and S. Serafin, "Audio Arduino – an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, A. R. Jensenius, A. Tveit, R. I. Godøy, and D. Overholt, Eds., Oslo, Norway, 2011, pp. 211–216. [Online]. Available: http://www.nime2011.org/proceedings/papers/G01-Dimitrov.pdf

[5] "Atmel ATmega48A/48PA/88A/88PA/168A/328/328P datasheet," http://www.atmel.com/devices/ATMEGA328P.aspx?tab=documents, [Online; accessed 12-Jun-2013].

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical recipes in C: The art of scientific computing. second edition," 1992.

[6] http://compmus.ime.usp.br/en/