Lab 2: Audio Effects

# Introduction

After my failed filtering experiment I moved on to something more achievable, an audio effects box. I was able to find some Arduino realtime audio processing labs online to go off [3]. Unfortunately none of them worked quite right on my Arduino, so I worked to modify these and get them working. Using these I was able to implement a reverb, delay, and phaser audio effect. I’ll go over what I learned through these labs.

# Hardware

The hardware remains mostly the same from the previous filtering example with one big change. I’m not longer using the large 8 bit digital to analog converter. All my audio output is ported through a single digital PWM pin. I’m bypassing the DAC and connecting the PWM output directly to the first set of op amps that perform filtering and then to the voltage followers to power my speaker.

# Setup

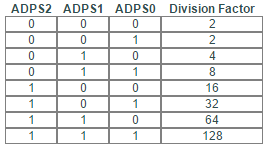
In order to do real time audio sampling certain optimizations have to be made to the Arduino. In my previous experiment I was having trouble due to the speed of sampling and processing. It turns out there are ways to improve this. The ADCs normally records 10 bit precision at 9600Hz [1]. Setting the ADC in fast sampling mode with 8 bit precision can improve the sampling rate to 15625Hz. There is only one ADC unit on the Arduino to handle all 6 inputs, only one channel can be handled at a time. Because the Arduino is an 8bit chip but has 10bit ADC precision the ADC values are stored in two registers ADCH (high) and ADCL (low). The ADC takes 13 ADC clock cycles to perform a conversion [1]. The sampling frequency can be optimized by setting a prescaler value which will speed up the conversion by increasing the ADC clock speed. The prescaler value determines the sampling frequency as a factor of the 16MHz core clock frequency. By default the prescaler is 128 which yields 16MHz/(128prescalar\*13cycles) = 9615Hz. Changing the prescaler to 64 yields a sampling rate of 16MHz/(64prescalar\*13cycles) = 19231Hz. According to the author of the lab, this is done with the following code [3]:

cbi(ADCSRA, ADPS2);

sbi(ADCSRA, ADPS1);

sbi(ADCSRA, ADPS0);

The cbi and sbi methods are used for setting (sbi) or clearing (sbi) bits. These methods are defined at the top of the sketch. In this case its bits of the ADCSRA, the ADC control and status register. The bits ADPS# stand for the ADC prescaler select. By setting bits 0 and 1 and clearing bit 2 this code actually sets the prescaler to 8, which is not what the author intended based on their comments. Here’s a prescaler chart found I was able to find to demonstrate this [2]:



I went ahead and modified the code to set bits 0 and 2, and to clear bit 1 to change the prescaler to 32 for 38kHz sampling. I chose this because I know my ADC will be alternating between sampling from pin 0 and pin 1. That means I can sample audio at 38/2 or 19kHz. According to Nyquist theorem I should theoretically be able to sample up to ~9.5kHz with this configuration.

The next register to set up is the ADMUX or, ADC Multiplexer Selection Register. Here is the code for setup [3]:

sbi(ADMUX,ADLAR); // 8-Bit ADC in ADCH Register

sbi(ADMUX,REFS0); // VCC Reference

cbi(ADMUX,REFS1);

cbi(ADMUX,MUX0); // Set Input Multiplexer to Channel 0

cbi(ADMUX,MUX1);

cbi(ADMUX,MUX2);

cbi(ADMUX,MUX3);

In the previous experiment I divided all inputs by 4 to convert the 10 bit value to an 8 bit value, although this can be done by setting the ADLAR, or ADC Left Adjust Result, bit. Normally the two most significant bits are stored in ADCH, while the lower 8 bits in ADCL. If ADLAR is set the 8 MSB are stored in ADCH and the 2 LSB are in ADCL [1]. In short this allows us to only read ADCH at 8 bit precision to get the 8 MSB, which is faster. The next two lines REFS0 and REFS1 selects the reference voltage for the ADC, which is the voltage it will compare its analog readings to [1]. The final four lines are a multiplexed selection of what input to monitor. In this case clearing all the bits results in monitoring only Analog pin 0.

As I stated earlier, I am no longer using the DAC, but rather using the PWM. PWM stands for pulse width modulation and is another way to create an analog signal from a digital one. It does so using a duty cycle. Digital pins can only be on or off, in the case of the Arduino, 5V or 0V. As the name sounds the duty cycle controls the length the pulse is at 5V. For example a 50% duty cycle would be held at 5V for half the cycle and 0V for the rest of the cycle. Doing this fast enough results in a signal between 0 and 5V, so a 50% duty cycle would be 2.5V, 10% would be 0.5V, etc… In order to use this for outputting the audio signal at a reasonable frequency, we can use the Arduino timer 2 as PWM output. In one of the starter experiments that came with my kit I created a tone generator sketch that was able to output a tone (sine wave) at a specified frequency. I was even able to control the frequency with a light sensor making it fun to play (theremin project in the sketches folder). This tone experiment actually also used timer 2 as a PWM to output the analog signal [1]. All that was necessary was to call the tone method from the built in Arduino library in this case, however, we can gain more control by modifying the Timer2 register directly [3].

// Timer2 PWM Mode set to fast PWM

cbi (TCCR2A, COM2A0);

sbi (TCCR2A, COM2A1);

sbi (TCCR2A, WGM20);

sbi (TCCR2A, WGM21);

cbi (TCCR2B, WGM22);

The first two instructions set the PWM to non-inverting mode. This controls when the pulse happens, either at the beginning of the cycle or the end [1]. Non-inverting or normal mode means the pulse happens at the beginning of the cycle. The last three instructions use the waveform generation bits to set the timer to fast PWM mode.

Just like the ADC the timer also has a prescaler that affects its clock speed. The following code sets this prescaler to 1 which is effectively no prescaling [3]:

// Timer2 Clock Prescaler to : 1

sbi (TCCR2B, CS20);

cbi (TCCR2B, CS21);

cbi (TCCR2B, CS22);

Therefore the timer selects the 16MHz core clock as its clock source.

Port registers are used on the Arduino for faster I/O manipulation. There are 3 port registers: B, C, and D. PORTD controls digital outs 0-7, PORTB digital pins 8-13, and PORTC analog inputs 0-7 [1]. The following code sets digital pin 11 as the output [3].

// Timer2 PWM Port Enable

sbi(DDRB,3); // set digital pin 11 to output

DDRB stands for data direction register for Port B. Since Port B controls pins 8-13 setting bit 3 corresponds to digital pin 11. The final steps to the setup are to disable timer0 and enable the timer2 interrupt responsible for our PWM output.

# Interrupts

One of the most important aspects of these lessons is the use of interrupts. The Arduino lacks threading which would make concurrent operations such as reading the ADC values much easier. Through the flow the main loop the Arduino executes instructions line by line. An interrupts makes it possible to stop the main loop, switch context, execute some other code, and then return to the main loop restoring the previous context. The Arduino can do input interrupts or timer based interrupts, such as in this lab. The lab uses timer two to execute an interrupt service routine at 62.5KHz [3]. The import thing with interrupts is that the number of instructions is limited within them [3]. If the interrupt routine takes too long to complete (longer than the time between them) the interrupts can take over the program leaving no work to be done by the main loop. Here is the code for the interrupt routine from the lab [3]:

ISR(TIMER2\_OVF\_vect) {

PORTB = PORTB | 1 ;

div32 = !div32; // divide timer2 frequency / 2 to 31.25kHz

if (div32) {

div16 = !div16; //

if (div16) { // sample channel 0 and 1 alternately so each channel is sampled with 15.6kHz

byteADC0 = ADCH; // get ADC channel 0

sbi(ADMUX, MUX0); // set multiplexer to channel 1

f\_sample = true;

}

else

{

byteADC1 = ADCH; // get ADC channel 1

cbi(ADMUX, MUX0); // set multiplexer to channel 0

}

\_\_asm\_\_("nop\n\t""nop\n\t""nop\n\t""nop\n\t"); //No operations to create a delay

sbi(ADCSRA, ADSC); // start next conversion

In the setup the prescaler was set to no prescaler or 1. We can figure out the frequency that the code from the interrupt routine is knowing the core clock speed 16MHz, prescaler 1, and the maximum timer value 256. When the timer, which is essentially a counter, reaches 256 cycles the code is executed. That gives a frequency of 16,000,000 / 256, or 62.5kHz. The first instruction in the routine is a toggle to only execute the code every other cycle so 62.5kHz / 2, or 31.25KHz. The next line controls the alternation of sampling from pin 0 and pin 1. The first time through the audio in is sampled on analog pin 0, and the second time the potentiometer input on analog pin 1 is sampled. Then there are several nop, or no operations, performed to give the ADC time before starting a new conversion.

# Reverb and Delay

Reverb, or reverberation, is a sound that occurs as a reflection of an original sound. Thinking of a drum room, the initial impulse of a drum creates a sound. That sound bounces off objects in the room and persists as a reflection of the source. Delay is similar, but not the same. Delay is a repetition of the impulse signal at a later time. The algorithm to implement either reverb or delay isn’t all that complex; it uses a simple buffer to store previous samples and plays them back with the current sample simulating reflection. The Arduino doesn’t have much memory so we use a 512 value circular buffer. After the buffer is full it begins overwriting from the beginning. The reverb algorithm is as follows:

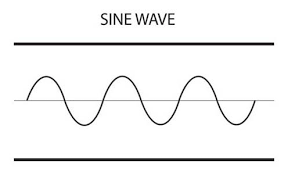
1. Get a previous sample from the delay circular buffer
2. Get the audio in value from the adc
3. Add the audio in and the previous sample and output it to the PWM
4. Store the new audio sample in the buffer in the previous’ place

I have a bit of background in audio design. As a hobby I work with sounds and put effects on them. The code from the lab actually places the addition of the audio in and the previous sample in the buffer, which I don’t think accurately represents reverb. It’s then possible to essentially create an infinite chain of “reverb”. I modified this to only include the new audio to make a more authentic reverb sound. I would consider the lab’s algorithm a delay effect [3]:

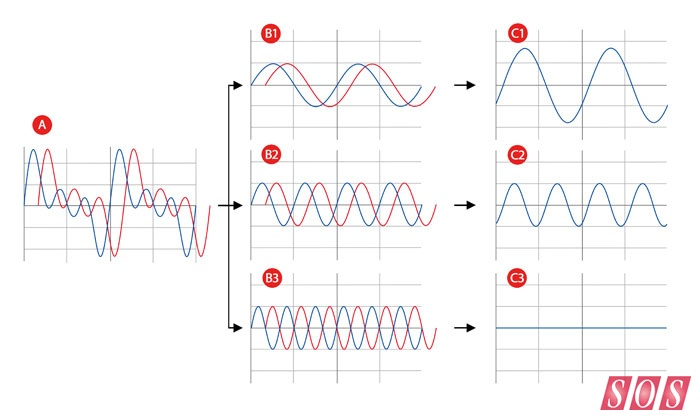
1. Get a previous sample from the delay circular buffer
2. Get the audio in value from the adc
3. Add the audio in and the previous sample and output it to the PWM
4. Store the sum of the audio in and the previous sample in the buffer in the previous’ place

# Phaser

Phaser is an effect that happens when two audio signals conflict with one another. For example, a simple sine wave:



All audio signals oscillate around 0V. The sine wave, like all audio signals has two components; the positive and the negative, which makes its sound. Phasing is the effect where one sine wave is added to another out of phase (their peaks and troughs happen at different times). This image from sound on sound depicts the effect [4]:



When the phase matches the amplitude is increased; however, if the phase doesn’t match the peak and trough cancel each other out. The effect can sound like a change in pitch, timber, or even full cancellation. This is actually one way a person can create an acapella, or vocal track, from an instrumental and track with instruments and vocals. Inverting the instrumental track and overlaying it with the instrumental+vocals track will cancel out all the instruments leaving only the vocals behind. The phaser algorithm is as follows [3]:

1. Use the potentiometer value to select sample n samples before the current sample from the delay circular buffer
2. Get the audio in value from the adc
3. Store the raw audio in into the buffer
4. Add the audio in and the previous sample and divide by two. Division is to keep the output signal in the expected range.
5. Output the phased signal to the PWM

# Results

I actually ran into a few snags going through these lessons. The instructions from the lab aren’t up to date with the code provided. The first thing I noticed were some inconsistencies with the comments in the code and the code itself. Additionally, the code wasn’t initially very readable with some ambiguous choices for variable names. I cleaned these up with more descriptive names. The biggest issue; however, was simply a discrepancy between the hardware diagram and the code. The diagram lists the audio in as pin 0 and effects potentiometer as pin 1. As I was going through the code I noticed every reference to the audio was on pin 1 rather than 0. Simply flipping the input pins on my device solved the issue. I went back into the code and changed this so it's consistent with the hardware diagram.

I was able to gain a lot of knowledge from this lab. This is a more intermediate or advanced application. It hardly uses any of the built in Arduino library and relies almost completely on register manipulation. I was able to understand the ADC, timers, and interrupts. I also enjoyed understanding how these audio effects are created. I attached videos with some sound examples from this experiment.

**Works Cited**

[1] Atmel 8-Bit Microcontroller with 4/8/16/32KBytes In-System Programmable Flash Datasheet(J ed.). (2015). Atmel.

[2] The Analog to Digital Converter (ADC). (n.d.). Retrieved November 20, 2015, from <http://www.avrbeginners.net/architecture/adc/adc.html>

[3]Nawrath, M. (2008). Arduino Realtime Audio Processing. Retrieved November 15, 2015, from http://interface.khm.de/index.php/lab/interfaces-advanced/arduino-realtime-audio-processing/

[4] Senior, M. (2008, April 1). Phase Demystified. Retrieved November 24, 2015, from https://www.soundonsound.com/sos/apr08/articles/phasedemystified.htm