Lab #2: Analog Output (PWM), Timers, and Analog Input

1. Background

Computers operate on digital logic, and as such use digital input and output signals to communicate. However, the world at-large is analog, with most things having a range of values rather than two distinct states. To translate between the analog physical world and the digital world of computers, analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) are used.

The microcontroller used in this lab, the Atmel ATMEGA32u4, contains a built in ADC, but not a DAC. To replicate the functionality of a DAC, the pulse width modulation (PWM) functionality of the inbuilt timers can be used.

1.A. Analog-to-Digital Converter (ADC)

The ADC uses an op-amp, counter, and DAC. The counter is controlled by the op-amp and continues to count until the output voltage of the DAC is greater than the input voltage to the ADC. The value at which the counter stops is the digital numeric value corresponding to the analogue input voltage.

1.B. Digital-to-Analog Converter (DAC) using PWM

Since there is no built-in user accessible DAC on the Atmel ATMEGA32u4, the timers in PWM mode are used to implement one. While there are other ways to implement a DAC, using the PWM mode of the timers requires only a single pin and moves much of the design process to software. The only extra hardware required is a low-pass filter on the PWM output to remove the PWM signal itself from the output of the DAC.

2. Experimental Procedure

2.A. Direct Digital Synthesis (DDS)

For this lab, the analog waveform used is a rectified 1Hz sine wave with a 3V peak-to-peak amplitude. This signal can be constructed using the millis() function in the Arduino library, a function which returns the number of milliseconds elapsed since the start of running the program, along with the sin() and abs() functions included in the same library.

The sin wave has the mathematical expression of $-3\sin\left(\frac{2\pi t}{1000}\right)$ which can be written as the following C code:

```
// Include Arduino Library
#icloud <Arduino.h)
// Inside loop of the main program
long x = millis();
float aval =
abs(3*sin(2*3.141592654*x/1000));
```

The above code will store the analog voltage value produced at time x in the aval variable. Those values can then be passed to the DAC to produce the desired waveform.

2.B. PWM DAC

For this lab, the desired PWM output pin is D3 on the Playground classic, corresponding to the timer pin 0C0B on the microcontroller. To create a PWM signal on OC0B, timer 0 must be set to PWM mode and OCR0B must be configured as the compare register. To adjust the frequency of the signal, the OCR0A will be used as the top register and hold the value at which the timer will reset back to zero.

Given the 8MHz clock of the microcontroller, the PWM frequency is given by the following formula (1):

$$f_{PWM} = \frac{8MHz}{N(OCR0A+1)} \quad (1)$$

In Equation 1, N is the clock pre-scaler value, and OCR0A is the value in that register. Using a prescaler of divide-by 64, and the target frequency of 1KHz, the OCR0A value needed is 124, but 125 was used in practice as it was calculated by simply dividing the clock frequency by the prescaler and gave a frequency of almost exactly 1KHz to within less than 1Hz.

The control registers for timer 0 are TCCR0A and TCCR0B. The control registers were set to Fast PWM mode, clear-on-match, with OCR0A as top, and a clock prescaler division of 64. The code below shows the exact value used to setup the timer in this manner.

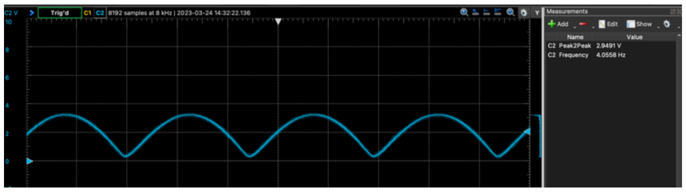
```
DDRD |= 1;
 /*
00 = Channel A pin is disconnected
   10 = Channel B Pin connected, clear on
          compare match, set at top
       xx = N/A
          11 = Bits 1:0 of wave type, Fast PWM
                 OCR0A TOP
  */
  TCCR0A=0b00100011;

/*
xx = In PWM Mode
    xx = N/A
        1 = Bit 2 of wave type, Fast PWM
              OCR0A TOP
             011 = Clk prescaler of 64
  */
  TCCR0B=0b00001011;

//8MHz/64=125KHz ->1KHz=125*125KHz
OCR0A = 125;
OCR0B = 0;
TCNT0 = 0;
```

The setup code starts the waveform duty-cycle at 0%. To generate the desired waveform, the duty-cycle, controlled by the value in OCR0B, is varied proportionally to the calculated sin wave value. This was done in the main loop using the following code:

```
void loop() {
x = millis();
aval = abs(3*sin(2*3.141592654*x/1000));
OCR0B = (uint8_t) 125 * aval / 3;
}
```

As can be seen in the code above, the value of the sin wave is divided by three to get the current sin wave voltage as a percent of the maximum three volts. This percentage is then multiplied by 125, the value of OCR0B which represents 100% duty cycle, to map the percent of maximum voltage to the duty-cycle percentage.

The constructed low-pass filter consisted of a 10kΩ resister and 1µF capacitor in series. The PWM signal was passed through this filter and the output was measure by the oscilloscope. Figure 1 shows the scope measurement for the filtered PWM DAC output signal:



Name	Value
C2 Peak2Peak	2.9491 V
C2 Frequency	4.0558 Hz

Figure 1: Oscilloscope Measurement of filtered PWM signal

As can be seen in Figure 1, some sort of error occurred as the frequency measured by the scope was 4.0558Hz, rather than the 2Hz expected of a rectified 1Hz sine wave. From this, we can reasonably deduce that there is a factor of 2 error somewhere in the digital synthesis calculations.

2.C. ADC

The desired ADC input pin for this lab was D12 on the Playground Classic, which is Port D Pin 6 on the microcontroller and Pin 9 of the ADC.

The ADC was run in free running mode, with a prescaler division of 128, 3.3V refrence, right shifted, and high-speed mode. The code for configuring the ADC is shown below.

```
DDRD &= ~(1 << 6);

/*Set 3.3V reference, Set MUX[4:0] to 1 to
read ADC9 */
ADMUX = (1<<REFS0) | 1;

//Enable ADC, set prescaler division to 128
ADCSRA = (1<<ADEN) | (1<<ADATE) | 7;

DIDR0 = 0; //Disable Digital Input
DIDR1 = 0; //Disable Digital Input

/*Enable High Speed Mode, Set trigger
source to free running mode, Set MUX[5] to
1 to read ADC0 */
ADCSRB = (1<<ADHSM) | (1 << MUX5);

//Start conversion
ADCSRA |= (1 << ADSC);
```

To read the value of the ADC, the ADCL and ADCH registers at addresses 0x78 and 0x79 respectivley are read as one 16-bit value. This value is anded with 1023 to cutoff any values over 1023, and then converted to the voltage measured by dividing by the number of values of the ADC, in this case 1023, and multiplying by the maximum voltage of three volts. The following code was used for the ADC read and conversion process.

```
ADCData=(unsigned short *)0x78;
ADCVal=(*ADCData & 0x3FF);
fADCVal=((float)ADCVal)/1023 * 3;
```

2.D. Data Analysis

To check the results, the Serial.print() lines below were added to the main loop of the program to compare the original rectified sinusoid value to the analog voltage measured by the ADC.

```
Serial.println(abs(aval));
Serial.println(fADCVal);
Serial.print("\n");
```

For voltages around 1V or higher, the two data points were essentially identical. However, for voltages smaller than 1V, a discrepancy opened up between the two values, growing larger as the voltage dropped. This error is likely the result of excessive filtering and could possibly have been avoided by using a lower value resister in the low-pass filter.

This behavior is somewhat unexpected however, as given that the frequency of the analog signal is on the order of 1Hz, and the frequency of the PWM signal is on the order of 1KHz, the lowpass filter used should be adequate with a cutoff frequency of 100Hz. The 100Hz cutoff frequency easily puts the original signal in the pass band and the PWM signal in the reject band.

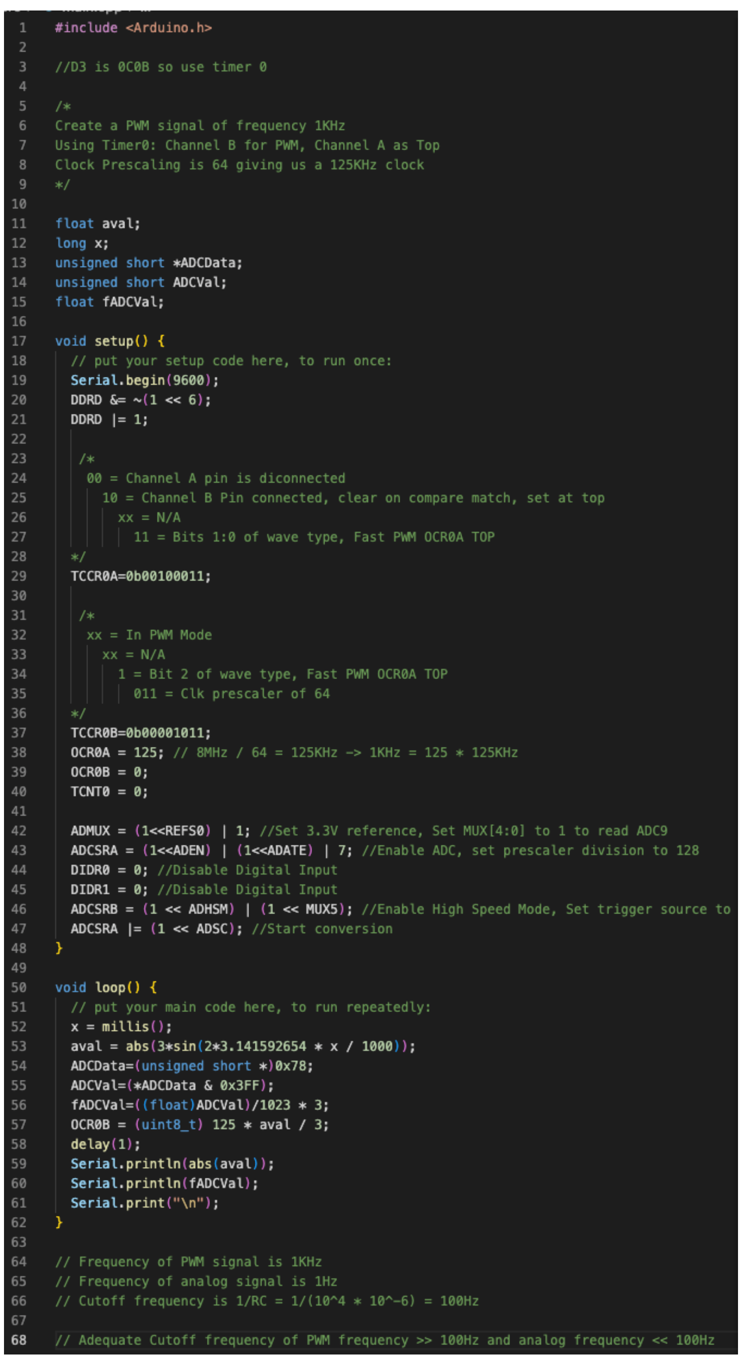
This discrepancy between theory and experiment could be an indication that the tolerances of the resistor and/or capacitor used was too high. The large tolerance might have led to significantly lower than advertised resistance and/or capacitance, resulting in the filter failing to properly cutoff the PWM signal.

3. Conclusion

In this lab, a DDS waveform was generated using the millis() function. The waveform was then sent to a DAC created using timer 0 in PWM mode. The built-in ADC was used to measure the waveform once it was passed through a low-pass filter.

There were two major unresolved errors. The first produced a roughly 4Hz frequency instead of the expected 2Hz for the rectified sine wave, and the second resulted in excessive filtering of the PWM output signal.

4. Complete Code Listing



```
#include <Arduino.h>

//D3 is 0C0B so use timer 0

/*
Create a PWM signal of frequency 1KHz
Using Timer0: Channel B for PWM, Channel A as Top
Clock Prescaling is 64 giving us a 125KHz clock
*/

float aval;
long x;
unsigned short *ADCData;
unsigned short ADCVal;
float fADCVal;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  DDRD &= ~(1 << 6);
  DDRD |= 1;

   /*
    00 = Channel A pin is diconnected
      10 = Channel B Pin connected, clear on compare match, set at top
        xx = N/A
          11 = Bits 1:0 of wave type, Fast PWM OCR0A TOP
  */
  TCCR0A=0b00100011;

   /*
    xx = In PWM Mode
      xx = N/A
        1 = Bit 2 of wave type, Fast PWM OCR0A TOP
          011 = Clk prescaler of 64
  */
  TCCR0B=0b00001011;
  OCR0A = 125; // 8MHz / 64 = 125KHz -> 1KHz = 125 * 125KHz
  OCR0B = 0;
  TCNT0 = 0;

  ADMUX = (1<<REFS0) | 1; //Set 3.3V reference, Set MUX[4:0] to 1 to read ADC9
  ADCSRA = (1<<ADEN) | (1<<ADATE) | 7; //Enable ADC, set prescaler division to 128
  DIDR0 = 0; //Disable Digital Input
  DIDR1 = 0; //Disable Digital Input
  ADCSRB = (1 << ADHSM) | (1 << MUX5); //Enable High Speed Mode, Set trigger source to
  ADCSRA |= (1 << ADSC); //Start conversion
}

void loop() {
  // put your main code here, to run repeatedly:
  x = millis();
  aval = abs(3*sin(2*3.141592654 * x / 1000));
  ADCData=(unsigned short *)0x78;
  ADCVal=(*ADCData & 0x3FF);
  fADCVal=((float)ADCVal)/1023 * 3;
  OCR0B = (uint8_t) 125 * aval / 3;
  delay(1);
  Serial.println(abs(aval));
  Serial.println(fADCVal);
  Serial.print("\n");
}

// Frequency of PWM signal is 1KHz
// Frequency of analog signal is 1Hz
// Cutoff frequency is 1/RC = 1/(10^4 * 10^-6) = 100Hz

// Adequate Cutoff frequency of PWM frequency >> 100Hz and analog frequency << 100Hz
```