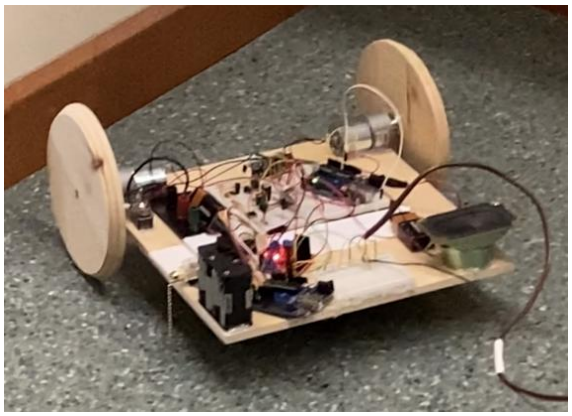
Authors: Max Finch, Tiger Slowinski
Team Members: Max Finch, Tiger Slowinski
05/03/23

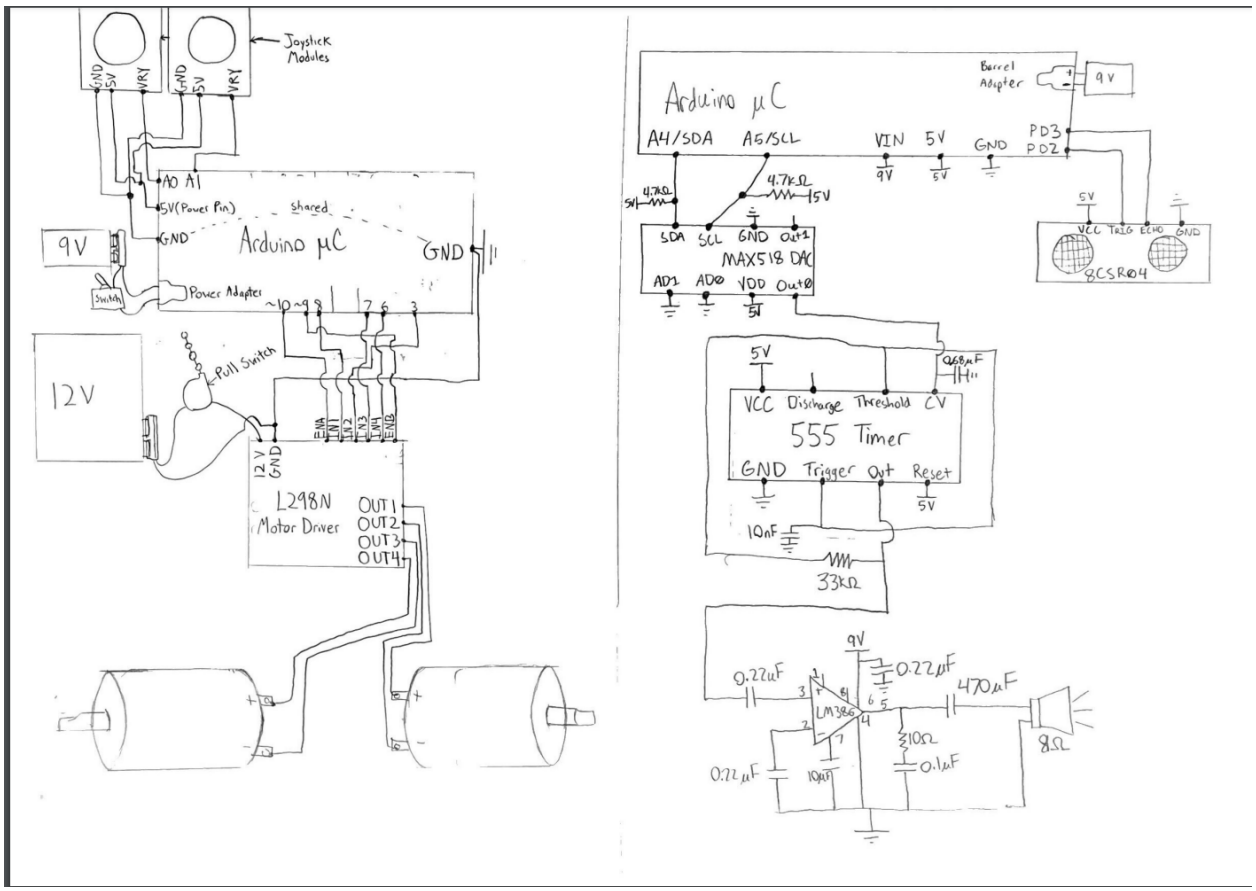
ECE:3360 Embedded Systems
FINAL PROJECT REPORT
Joystick Controlled Car (JSC) MDL1



1. Introduction

The objective of JSC MDL1 was to create a car that can allow users to walk in the dark, using audio based on the distance of the car from an obstacle to navigate. All power supplies have switches to allow for easy configuration. Motors are controlled with very precise PWM signals via a L298N motor driver that are based on the ADC converted position of the joystick. A 555 Timer, amplifier and DAC system are used to create noises between ranges of 3 to 60cm cm based on the 0.7 to 5V input based on distance.

2. Schematic


Joystick Modules
GND
5V
VRY
VRX
A0 A1
5V(Power Pin)
shared
GND
9V
Arduino µC
GND
Power Adapter
Switch
12V
Pull Switch
ENA IN1 IN2 IN3 IN4 ENB
12V GND
L298N Motor Driver
OUT1 OUT2 OUT3 OUT4
Arduino µC
A4/SDA A5/SCL VIN 5V GND PD3 PD2
Barrel Adapter
9V
4.7kΩ
5V
SDA SCL GND Out1
MAX518 DAC
AD1 AD0 VDD Out0
VCC TRIG ECHO GND
HCSR04
VCC Discharge Threshold CV
555 Timer
GND Trigger Out Reset
0.68µF
10nF
33kΩ
0.22µF
LM386
470µF
10Ω
0.1µF
8Ω

3. Discussion

Hardware Explanation

Analog digital conversion (ADC) is to take in a variable analog voltage, and then creating a digital representation of that voltage digitally using a set number of bits, or the bit resolution. In the case of the ATMega328P, the 6 analog pins have a resolution of 10 bits (0 to 1023). By setting our reference voltage (VREF) to our VCC of 5V, any voltage between 0V and 5V can be expressed as an integer value between 0 and 1023. By multiplying the digitally converted voltage value by 5/1024, we can scale the value back down to a value between 0 and 5 (for purposes such as viewing the voltage through the serial monitor) that appears to be analog, but the 1024 points of precision constraint remains.

The ADMUX register is used to configure ADC on the ATMega328P. Bits REFSx are used to set VREF, bit ADLAR can change the bit resolution between 8 and 10 bits, and finally MUXx bits select which analog pin is to be used for conversion. ADCSRA and ADCSRB are control and status registers used for ADC, however only ADCSRA is applicable in the case of this lab. This register contains ADEN (initialized to 1 to enable conversion), ADSC (start conversion bit, which is to be manually set to high when a conversion is to be done, and is set automatically back to low when conversion is complete), ADIE (enables interrupts to be called upon completion of an ADC, not applicable in our case and is set to low), and finally 3 ADPSx bits. These bits determine a prescaler value for the ADC conversion, as the ADC uses the same internal clock as the microcontroller but requires a prescaler as the internal clock speed of 16MHz is too fast for the ADC. A prescaler of 128 is used in our case.

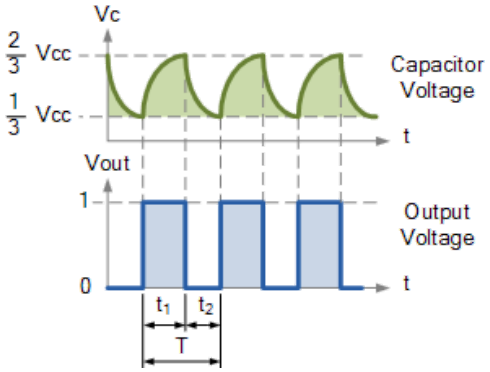
As the ATMega328P has 8 bit registers, but the ADC allows for 10 bit resolution, two registers ADCH and ADCL are used in tandem to store the final ADC value post conversion.

DAC

Our digital analog converter (DAC) device for this lab is the Maxim MAX518, an 8 pin converter that communicates with the microcontroller using two wire I2C protocol. The DAC reads in an 8 bit value (note the decreased resolution from the microcontroller's internal ADC), and can output a corresponding analog voltage out of its two analog out pins. VDD and GND pins on the DAC determine the extremes of the possible voltages out of the OUT pins on the DAC. Although the DAC was used alone for this lab, the pins A1 and A0 can be set high and low to create 4 distinct DAC addresses, making I2C communication with multiple DACs possible. Similarly to the scaling performed for the ADC, the user's desired analog voltage (in this case between 0 and 5), must be scaled to fall within the 8 bit resolution range of 0 to 255, which can be done by multiplying the desired analog voltage by 255/5 and then casting that value as an integer. This final value is sent to the DAC over I2C.

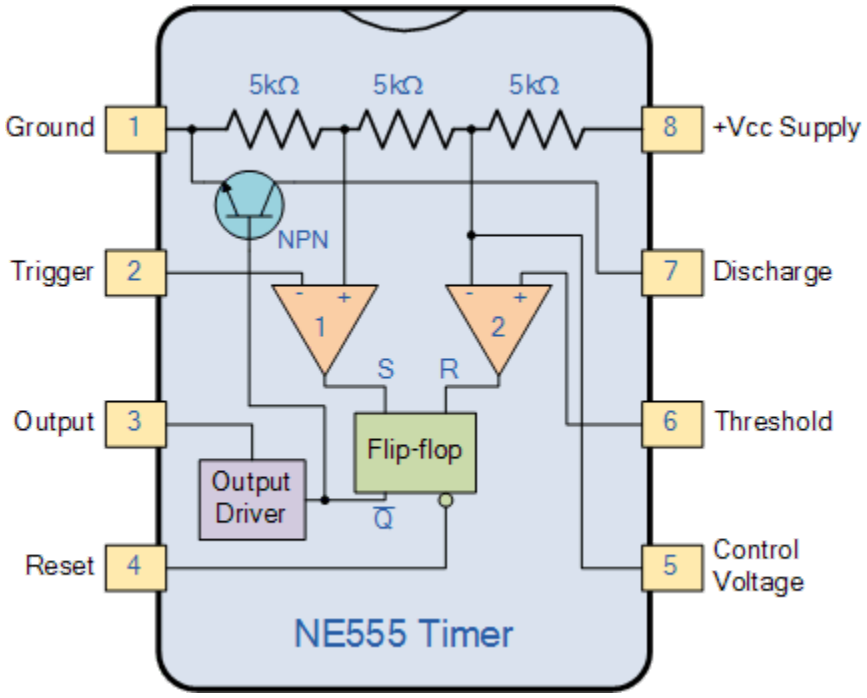
555 Timer Voltage Controlled Oscillator

While the 555 timer has multiple possible configurations just for use as an oscillator alone, our version functions as a PWM circuit. The cycle time spent LOW stays relatively fixed, while the time spent HIGH is modulated with changes in the control voltage (CV). As the time spent HIGH increases, the pulse width as well as the frequency increases as well. The result is a square wave output with fluid changes in frequency possible with continuous change in CV (0V to 5V) sent from the output of a DAC.



This oscillation between two output states is made possible using a resistor capacitor circuit, along with taking advantage of the internal construction of the 555 timer. 3 equal value resistors internal to the IC sent to two comparators allows the output state at pin 3 to go LOW when the threshold pin exceeds 2/3VCC, and to go HIGH when the trigger pin goes below 1/3VCC. This change in the voltage applied to pin 2 and 3 is due to the charging and discharging of a capacitor attached to these pins. Increasing the value of this

capacitor increases the time necessary to charge and discharge the capacitor, thereby lowering the rate of oscillation and the frequency of the output.



Final Amplification

While the output of a 555 timer circuit is quite powerful on its own (especially compared to many other audio sources such as a turntable or electromagnetic pickup), passing a signal through an amplifier circuit provides some benefits. The standard coupling capacitors at the input and output of the LM386 based power amplification circuit remove the DC bias from the square wave signal which protects the speaker, as well as providing some filtering to make the audio more pleasant and less susceptible to noise. With gain pins (pins 1 and 8) of the LM386 open, a gain of roughly 20 is achieved, functioning as a buffer for the signal. The output is passed safely to our 8 ohm speaker load, and the result is a subjectively pleasant analog signal reminiscent of original analog synthesizer sounds from instruments such as the 70s Moog.

Joystick Module

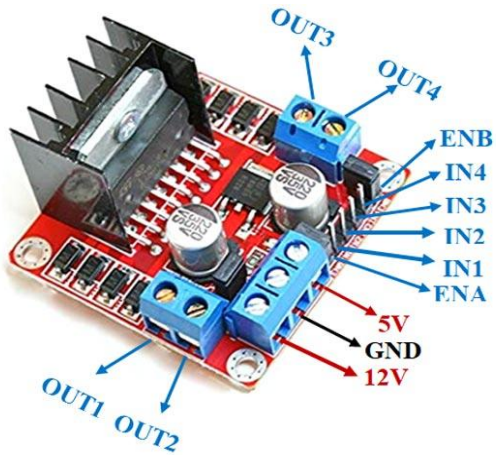
The joystick module for Arduino uses two potentiometers and a switch to measure joystick position and button presses (not used in this project). The Arduino's analog input

pins are connected to the potentiometer output pins to read the voltage changes caused by joystick movement. The analog-to-digital converter (ADC) of the Arduino then converts the voltage to a digital value ranging from 0 to 1023. These digital values can be mapped to control the speed and direction of DC motors via PWM signals sent to the motor driver's input pin.

L298N Motor Driver

The L298N is a dual H-bridge motor driver, which means it has two channels that can control the direction and speed of a motor. It can handle a max voltage of 46V and a maximum current of 2A per channel.The L298N also has built-in protection features, such as thermal shutdown and overcurrent protection.

In our project we needed to control two 12V motors with the L298N, so we connected the motors to the two channels of the motor driver. Each channel has two output pins, labeled OUT1 and OUT2 for channel 1, and OUT3 and OUT4 for channel 2. The L298N has a built-in voltage regulator, so you can use a higher voltage power supply than your motors. For our project an external 12V power supply was required to avoid stall current from the motors causing the microcontroller to turn off. The microcontroller was separately powered with a 9V battery.



To control the speed and direction of the motors, we send signals to the input pins of the L298N. There are four input pins, labeled IN1, IN2, IN3, and IN4. To make a motor spin in one direction, you need to set one input pin to HIGH and the other to LOW. To make it spin in the opposite direction, you need to reverse the states of the input pins.

Software Explanation

I2C

We utilize the I2C protocol via a Two-Wire Interface (TWI) to communicate with the I2C interface on the MAX518 DAC used in this lab. Let's go through the functions used in our code:

[1] i2c_init();

This function handles initializing the TWI clock by configuring the TWSR and TWBR registers of the ATmega328P. Only needs to be called at the start of the program.

[2] i2c_start();

This function issues a start condition (see figure 1) and sends address and transfer direction. We passed in the address of the MAX518 DAC, which based on our configuration was “0x58”. Setting pins A1 and A0 of the MAX518 to ground as one possible address. Since there are 4 different bit combinations of A0 and A1 we can have a max of 4 MAX518’s if we wanted to on the same bus (see figure 2). It will return 0 if the i2c device is accessible and 1 if it has failed to access the device.

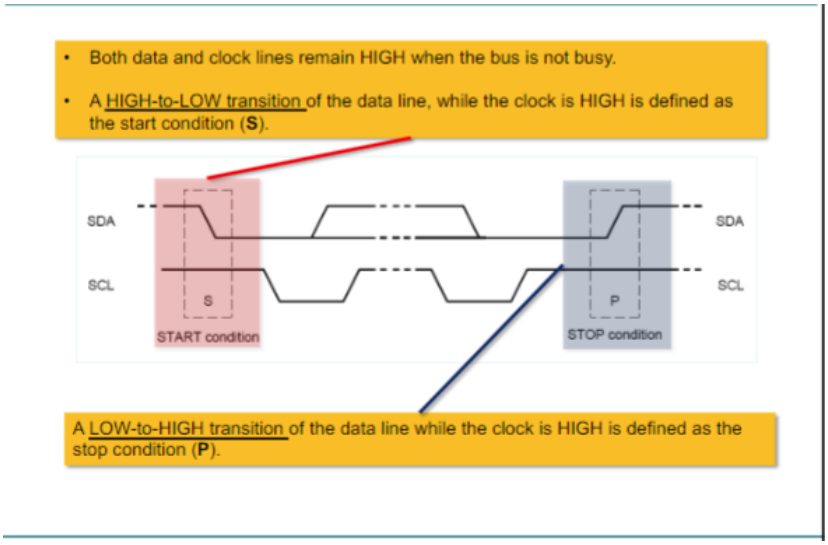


Figure 1

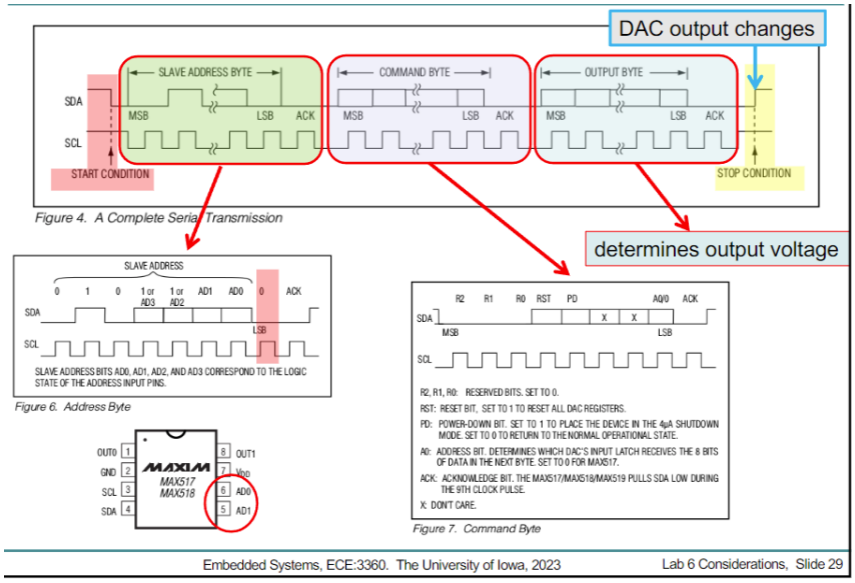


Figure 2

[3] i2c_write();

This function sends a single byte to an I2C device. It sends data to the previously addressed device, waits until the TWINT bit in the TWCR register is set to 1 (waits until transmission of data completes). It will then check the TWI status register to verify whether the byte transfer was successful. It will return 0 for failure and 1 if it was a success.

[3] i2c_stop();

This function sends "1" bits via TWINT, TWEN and TWSTO to "start" the stop condition. It waits until the TWSTO bit is turned back to 0 and the bus released to complete the operation.

To see examples of how I2C is used in process in our code, please see lines (136-141).

Decimal to String, String to Decimal

Multiple standard C libraries were used to perform conversions between C-style string character arrays and numbers of integer, float, and decimal type. *atoi()* can be passed elements of a character array containing ascii characters for integers, and produces an integer with each place value of the integer corresponding to its respective ascii character in the original character array.

dtostrf() was likewise used to convert floating point values to a C-style string, with the function taking in the desired character array to be modified, total desired characters/digits (right justified), and decimal points of precision as parameters.

Using Ultrasonic Sensor Data to Generate CV

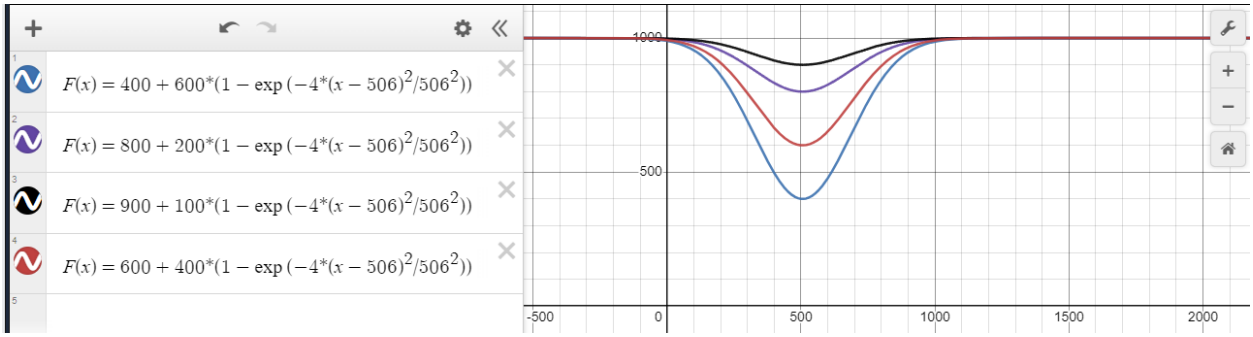
Once the ultrasonic sensor data is converted to a useable distance measurement in centimeters, we first limit our range to the accurate range of 3 to 60cm, with anything measuring greater than 60 centimeters resulting in a control voltage (CV) of 0V corresponding to no frequency (lower CV results in a lower frequency), and measurements of 3 cm resulting in the maximum CV of 5V, and measurements of less than 3 cm resulting in the same 5V CV.

The 555 timer based Voltage Controlled Oscillator (VCO) has a usable CV range of 0.7V to 5V to create usable oscillations, with 0V resulting in no sound and less than 0.7V resulting in unstable oscillations and noise.

To have the 3 cm to 60 cm distance correspond to an output from the DAC of 0.7V to 5V, some scaling and conversions must be done. First, the distance data is subtracted by 3 to have the range be between 0 and 57. This distance is then multiplied by the ratio of 4.3V/57cm, with the 4.3V derived from the 0.7V to 5V range subtracted by 0.7V to have a voltage range of 0 to 4.3V. The result plus 0.7 volts gets us back into the 0.7V to 5V range, and this number is multiplied by 51, or the ratio of the max 8 bit number the DAC can take as an instruction over the max voltage of 5V.

Calculating PWM Values Based on Joystick Input

In this project it was crucial for the joystick to have extremely smooth transitions between speeds. The function used in our project was the one defined in red. Essentially, the x-axis is the Joystick ADC output range (0 - 1023) and the y-axis is the PWM output. See the function implemented at lines 41 and 44. Through various physical tests, we determined that the minimum PWM value to start the motor was around 400 and the PWM value where motor speed flatlines is around 1000. Using this information we concluded that creating an exponential function with a local minimum of (400, 506-510) that smoothly transitioned from inputs of 0-1023 to outputs of 400-1000 would provide the smoothest control. We essentially created a pseudo inverse Normal Distribution curve, which is commonly used by teachers/professors to curve test/exam grades amongst classes.



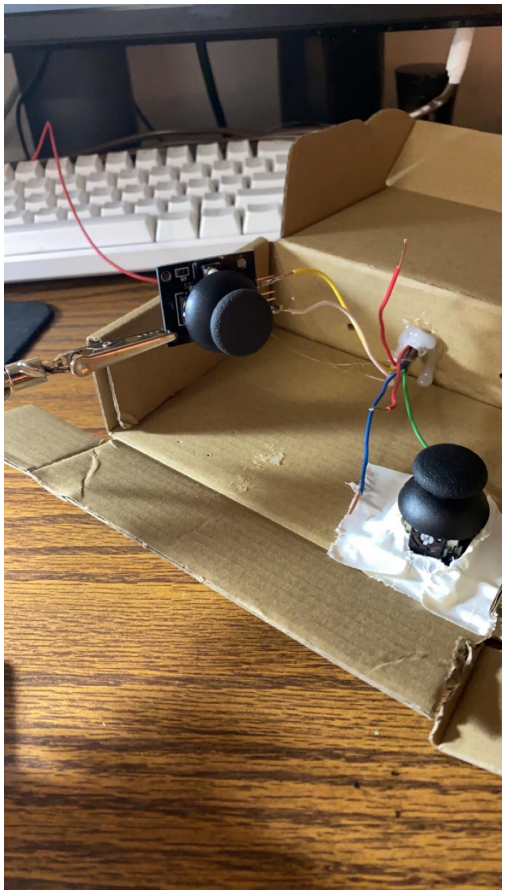
Interrupts

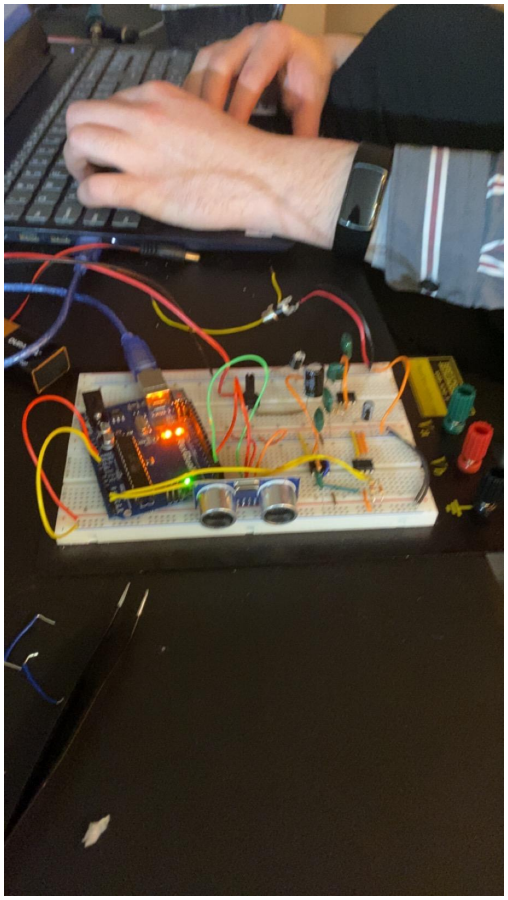
Interrupts allow for freed processing power, responsiveness, and program efficiency at the meager cost of programmer torment. The program counter can be anywhere in the program, and when an interrupt occurs the current instruction will be completed and the program counter will go the address of the appropriate Interrupt Service Routine (ISR), with the return address loaded onto the stack for the program counter to go back to upon completion of the ISR. When an interrupt occurs either in response to a pin change or state change (rising or falling edge), the program counter goes to a defined address in memory corresponding to the type of interrupt called and where it was called from, with the addresses listed in the documentation for the microcontroller Interrupt Vector Table (IVT). While all digital pins of the ATMega328P are associated with a PCINTx value, pin change interrupts can be neatly grouped according to what PORT the pin is a part of, whereas external interrupts have two designated pins (D2 and D3). From there, the program counter may jump to an ISR, and upon completion of the ISR the program returns to the address that was loaded to the stack. In our project, the interrupt simply increments a variable that indicates how many times the timer1 overflow flag has been set. It uses this value in the calculation on line 68.

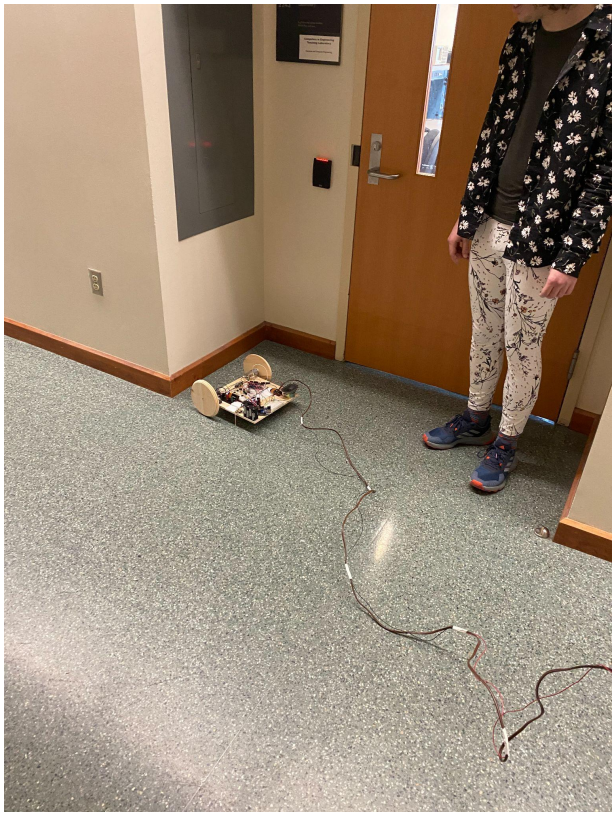
4. Experimental Methods

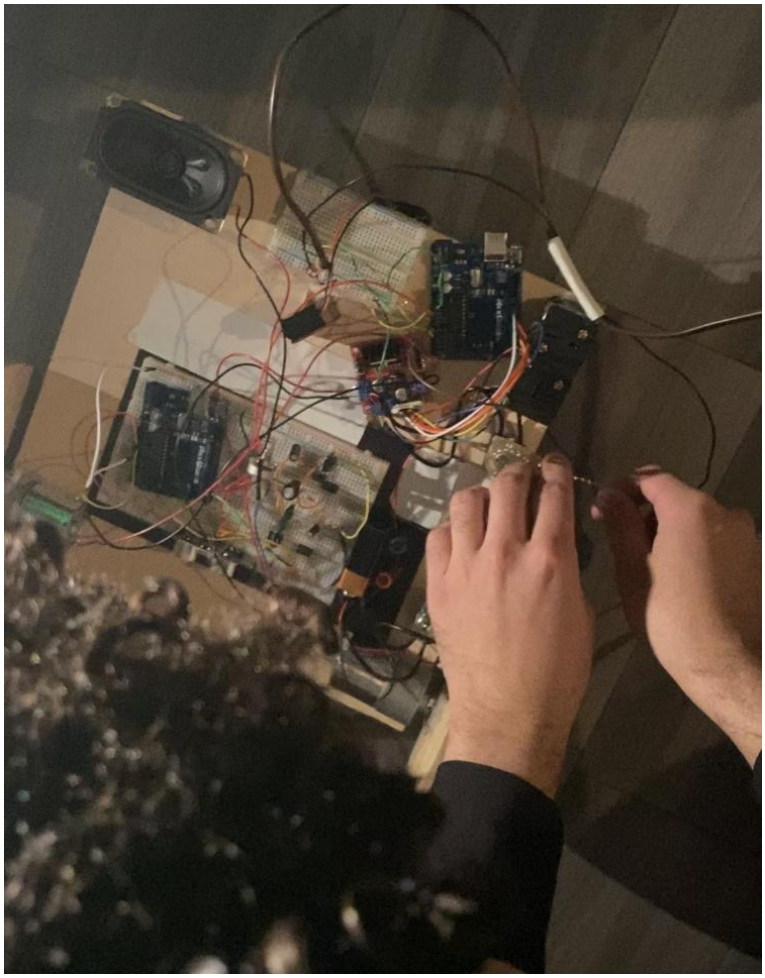
- Constructing the frame of the JSC was rather simple. Adding a caster wheel close to the center of the wooden square frame reduced the strain on the motors which were hot glued to the frame.
- We added mechanical switches to make turning the JSC on a lot easier.
- We used mounting hubs to perfectly align the wheels perpendicular to the motor drive shaft.
- The Greartisan DC motors we used are known to have strong torque, which worked well with our PWM controller design since small changes of PWM equated to small movement of the motors. This is supported by the fact that output speed and output torque are inversely proportional.

5. Results (Pics of hardware, graphs, etc)









6. Discussion of Results (were all design goals met?)

Unfortunately not all design goals were met. The original plan was to control the car using two RF transceiver modules with inputs given by two theremins. Although the proposed module nrf24L01 was well documented in C++, converting it to C would have taken the equivalent time as an independent study. The C++ library implemented the nrf24L01 with an object oriented approach while using other sub C++ libraries, creating an endless rabbit role of translation. There were C libraries but were poorly documented and did not follow guidelines provided by Nordic Semiconductors.

7. Conclusion

This lab provided useful insight into how to get multiple IC's working together even if not in the same physical circuit. We also gained experience in physical circuit layout and now know places where certain components like microcontrollers, power supplies and user interfaces should be on a transportation centered project. We learned how to

properly pick external power supplies based on specific IC requirements and how to avoid one IC from drawing too much current from the rest of the system (such as a motor stalling). One interesting thing was seeing components, like the HC-SR04 ultrasonic sensor, have embedded components such as an ADC which we used in previous labs.

8. Appendix A: Source Code

```c
/*
 * Thermicar.c
 *
 * Created: 4/22/2023 9:23:37 PM
 * Author : Max Finch
 */
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <util/delay.h>
#define F_CPU 16000000


void PWM_Setup(void)
{

    // Set Timer1 for 8-bit Fast PWM mode with ICR1 as TOP and enable PWM outputs on OC1A and OC1B
    TCCR1A |= (1<<COM1B1) | (1<<COM1A1) | (1<<WGM11) | (1<<WGM10);
    TCCR1B |= (1<<WGM12) | (1<<CS10); // set PWM mode with ICR1 as TOP and no prescaling


    // Set the TOP value to 1023
    ICR1 = 400;
    // 1000-> 600 and 400
    //
    // Set the duty cycle of motor 1 to 0%
    OCR1A = 0;
    //Set the duty cycle of motor 2 to 0%
    OCR1B = 0;


}
//80 might be motors min PWM   was 200 + 800 for 1000
int ADC_To_PWM(int x, int v) {

    float result;
    if(v==1)
    {
        result = 600 + 400.0 * (1.0 - exp(-4.0 * pow(x-506.0, 2.0) / pow(506.0, 2.0)));
    }else if (v==2)
    {
        result = 600 + 400.0 * (1.0 - exp(-4.0 * pow(x-510.0, 2.0) / pow(510.0, 2.0)));
    }

    return (int)round(result);
}
int num = 1;
```

```
50  void update_Motors(void)
51  {
52      //A0 and A1 are for joystick control
53      int ADC1;
54      int ADC2;
55      //OCR1A motor 1, OCR1B motor 2
56
57      //read first joystick and get new DC
58      read_ADC(0);
59      ADC1 = ADC;
60      OCR1A = ADC_To_PWM(ADC1, 1);
61
62      //read second joystick and get new DC
63      read_ADC(1);
64      ADC2 = ADC;
65      OCR1B = ADC_To_PWM(ADC2, 2);
66
67      //Update motor spin direction
68      Motor_Direction(1, ADC1);
69      Motor_Direction(2, ADC2);
70
71
72  }
73  void Motor_Direction(int motor_num, int ADCn)
74  {
75      //PB0,PD3 go to motor 1, PD6,PD7 go to motor 2, PB1 & PB2 are CLK pins
76      if(((motor_num==1)&&(ADCn == 506))||((motor_num==2)&&(ADCn == 510)))
77      {
78          //stop motor
79          if(motor_num == 1)
80          {
81
82              PORTB &= ~(1 << PORTB0);
83              PORTD &= ~(1 << PORTD3);
84          }else if(motor_num == 2)
85          {
86
87              PORTD &= ~((1 << PORTD6)|(1 << PORTD7));
88          }
89          //<
90      }else if(((motor_num==1)&&(ADCn < 506))||((motor_num==2)&&(ADCn < 510)))
91      {
92
93          //Go "Forward"
94          if(motor_num == 1)
95          {
96
97              PORTD |= (1 << PORTD3);
98              PORTB &= ~(1 << PORTB0);
99          }else if(motor_num == 2)
100         {
101
102             PORTD |= (1 << PORTD6);
103             PORTD &= ~(1 << PORTD7);
104         }
105
```

```
    }else if(((motor_num==1)&&(ADCn > 506))||((motor_num==2)&&(ADCn > 510)))
    {

        //Go "Backward"
        if(motor_num == 1)
        {
            PORTD &= ~(1 << PORTD3);
            PORTB |=  (1 << PORTB0);

        }else if(motor_num == 2)
        {
            PORTD &= ~(1 << PORTD6);
            PORTD |=  (1 << PORTD7);
        }

    }


}

void ADC_Init(void)
{
     //set to VCC and to A0
    ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);

}
void read_ADC(int PIN_num)
{
    if(PIN_num == 0)
    {
        ADMUX = 0x40;
    }else if(PIN_num = 1)
    {
        ADMUX = 0x41;
    }
    ADCSRA = ADCSRA | (1<<ADSC);

    while (!(ADCSRA & (1 << ADIF)));
    ADCSRA |= 1 << ADIF;

}


int main(void)
{
    ADC_Init();
    DDRB |= 0x3F; //Pin PB0,1,2,3 are outputs
    DDRD |= 0xF8;
    PWM_Setup();
    PORTB = 0x00;
    PORTD = 0x00;

    while(1)
    {
        update_Motors();
```

```
150 int main(void)
151 {
152     ADC_Init();
153     DDRB |= 0x3F; //Pin PB0,1,2,3 are outputs
154     DDRD |= 0xF8;
155     PWM_Setup();
156     PORTB = 0x00;
157     PORTD = 0x00;
158
159     while(1)
160     {
161         update_Motors();
162
163     }
164
165
166 }
167
168
```

```c
/*
 * USSScript.c
 *
 * Created: 4/29/2023 8:15:27 PM
 * Author : Tiger Slowinski
 */

#include <avr/io.h>
#include <stdlib.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include "twimaster.c"

#define F_CPU 16000000

#define TRIG_PIN PD2   // Trigger pin (digital output)
#define ECHO_PIN PD3   // Echo pin (digital input)
#define MAX_DISTANCE 3050 // Maximum distance in centimeters (30.5 cm = 3050)
#define DAC 0x58

volatile uint8_t timerOverflow = 0;
volatile uint32_t pulseDuration = 0;



ISR(TIMER1_OVF_vect) {
    timerOverflow++;

}

void init_HC_SR04() {
    DDRD |= (1 << TRIG_PIN);   // Set trigger pin as output
    DDRD &= ~(1 << ECHO_PIN);  // Set echo pin as input
    TCCR1B |= (1 << CS10);     // Enable timer with prescaler = 1
    TIMSK1 |= (1 << TOIE1);    // Enable timer overflow interrupt
}

uint32_t pulseIn() {
    pulseDuration = 0;
    timerOverflow = 0;
    PORTD &= ~(1 << TRIG_PIN);  // Set trigger pin low
    _delay_us(2);
    PORTD |= (1 << TRIG_PIN);   // Set trigger pin high for 10 us
    _delay_us(10);
    PORTD &= ~(1 << TRIG_PIN);  // Set trigger pin low
    while (!(PIND & (1 << ECHO_PIN))) {
        if(timerOverflow > 1) // Timeout after 100ms
        {
            return 0;
        }
    }  // Wait for echo pin to go high
    TCNT1 = 0;                  // Reset timer count
    TIFR1 |= (1 << TOV1);       // Clear timer overflow flag
    TIMSK1 |= (1 << TOIE1);     // Enable timer overflow interrupt
```

```
    sei();                      // Enable global interrupts
    while ((PIND & (1 << ECHO_PIN))) {  // Wait for echo pin to go low
        if (timerOverflow > 1) {    // If no echo, exit loop after timeout
            TIMSK1 &= ~(1 << TOIE1);  // Disable timer overflow interrupt
            cli();    // Disable global interrupts
            return 0;
        }
    }
    TIMSK1 &= ~(1 << TOIE1);    // Disable timer overflow interrupt
    cli();    // Disable global interrupts
    pulseDuration = timerOverflow * 65536UL + TCNT1;
    return pulseDuration;
}
```

```

int main(void)
{
    //ADC_Init();
    USART_Init(MYUBRR);
    init_HC_SR04();
    float voltage = 0.00;
    int distScaled = 0;

    while (1) {
        uint32_t pulseDuration = pulseIn();
        uint32_t distance = pulseDuration/58/13; //70.6; // Calculate distance in centimeters
        //distance = (distance / 3050.0f) * 12.0f;

        //oscillator uses 0.7 to 5V input voltages from DAC, and range of accurate readings from ultrasonic sensor is 3 to 60cm.
        if(distance >= 60)
        {
            voltage = 0.00;
        }else
        {
            distScaled = distance - 3; //range is now from 0 to 57
            voltage = distScaled * 0.0754; //output voltage to oscillator must be between 0.7 and 5 volts, equivalent to 0 to 4.
            voltage = voltage + 0.7; //voltage is now a range between 0.7 and 5
        }

        if(distance <= 3)
        {
            voltage = 0.70; //if less than 3cm from an object, pitch will not increase further
        }

        int final_int = voltage * 51; //scaling to an 8 bit value to be understood by DAC

        i2c_start(DAC);
        i2c_write(0);   //output channel
        i2c_write(final_int);
        i2c_stop();


        _delay_ms(500);   // Delay between measurements
    }
    return 0;
}

```

9. Appendix B: References

Atmel Corporation. *AVR Instruction Set Manual - Microchip Technology*. <https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-0856-AVR-Instruction-Set-Manual.pdf>.

Arduino. *Arduino UNO Rev3 with Long Pins.*<https://docs.arduino.cc/retired/boards/arduino-uno-rev3-with-long-pins>

Damadmai. "PFLEURY/TWIMASTER.C at Master · Damadmai/Pfleury." *GitHub*, <https://github.com/damadmai/pfleury/blob/master/twimaster.c>

Analog Devices. "MAX517/MAX518/MAX519: Low-Power, Single-Supply, Voltage-Output, 8-Bit DACs in µMAX." Datasheet. Analog Devices, 2004, <https://www.analog.com/media/en/technical-documentation/data-sheets/MAX517-MAX519.pdf.>

AVR-GCC Libraries: I2C Master Library, <https://damadmai.github.io/pfleury/group__pfleury__ic2master.html>

Components101. "L293N Motor Driver Module - Pinout, Specifications, Equivalent & Datasheet." Components101, 23 Mar. 2021, components101.com/modules/l293n-motor-driver-module.

Lab Lecture Slides provided by Professor Beichel