Lab 4

ECEN 2270
University of Colorado Boulder

April 15, 2022

Luke Hanley
Nicholas Haratsaris
Ginn Sato

Introduction

In this Lab, we implement an Arduino Nano Every to control our previously designed circuits. The bulk of our time was spent in the Arduino IDE software, which was a nice change of pace from the previous labs. The chief goal of this lab is to control our robots' movement using the Arduino microcontroller. In order to do this, we must first replicate the circuits we have built so that all motors are being powered. We need to set up and power the Arduino, and then code the Arduino output pins to send specific voltages to our direction control circuits. From there we implement two different software methods to control the movement of the robot– time delays and interrupts.

Equipment List

- Texas Instruments TLC3702 Operational Amplifier
- Diodes Incorporated ZVN2106A N-Channel MOSFET
- 4 pin push button
- Sparkfun Resistor Kit
- Sparkfun Capacitor Kit
- Digilent Analog Discovery 2
- Fully Assembled ROB 0025 Robot
- Breadboard Wires and Jumper Wires
- LTSpice Software
- Arduino Nano Every with USB connector
- Arduino IDE Software
- Microsemi Corp. 1N5818 diode

4.A Lab Exploration Topic

1) Hardware interrupts occur when an external source sends a specific voltage signal to the microcontroller. Software interrupts occur from internal processes of the microcontroller, such as internal clocks or specific registers having specific values. In our lab we will be dealing with external interrupts.
2) Polling is actively checking for a specific signal within our software program. Instead of having an interrupt tell us when we see the desired signal, we have our main looping program check for the signal.
3) In general, an interrupt occurs, a flag is raised, some code is executed, and then the flag is cleared and then the execution jumps back to where it left off.
4) In the context of the Arduino IDE software, interrupts are relatively straightforward. In our setup code we define what (external) interrupt we are looking for, such as a rising edge on a specific pin. When this signal happens, we stop executing the main looping code, and jump to the "Interrupt Service Routine". This is a specifically defined segment of code written to deal with the interrupt. We can only get to the ISR if an interrupt has been raised, and in jumping to the ISR we clear said interrupt.
5) In the Arduino Nano Every, the ATmega4808 processor is used, which is what defines the use of interrupts over the entire microcontroller. An interrupt request is sent to the processor when the corresponding interrupt is enabled, and this request raises the interrupt flag. The main assembly execution then jumps to the area of memory that

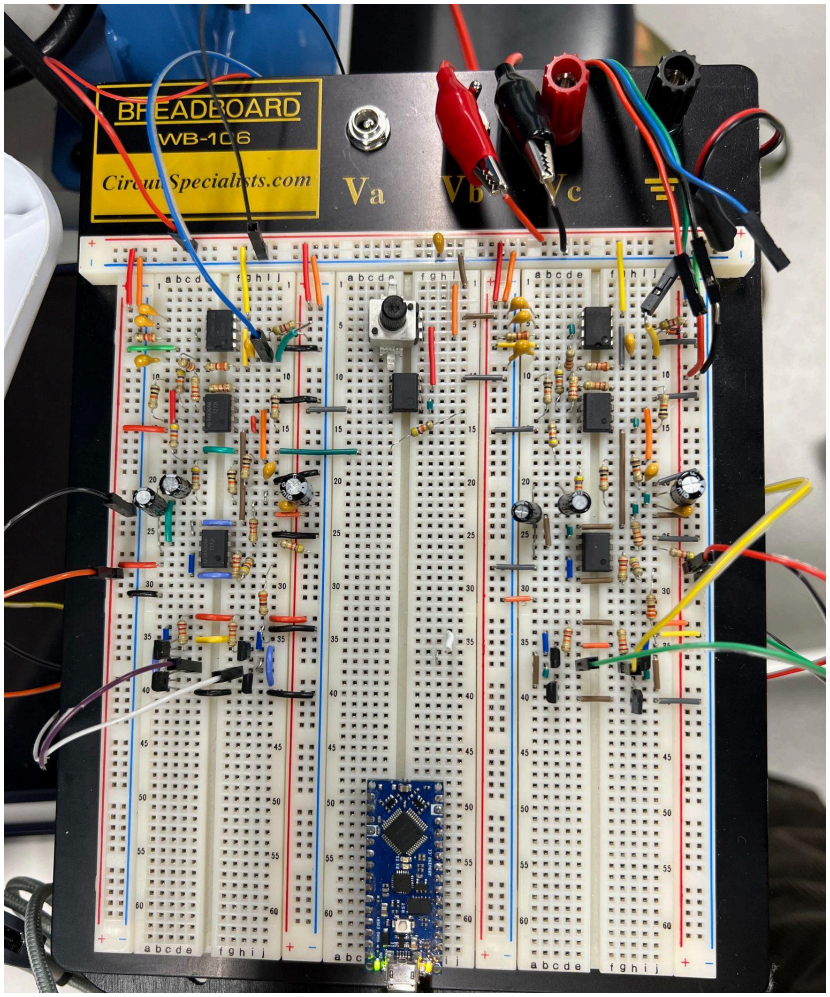
contains the interrupt service routine, defined as the ISR. The ISR then clears the interrupt flag, executes the code, then jumps back to where it was before the interrupt happened. Clearing the flag is necessary to be able to get back to main execution, otherwise, the execution will be stuck looping through the ISR.

6)

```
volatile bool x = LOW;

void setup() {
  pinMode(10, INPUT);
  attachInterrupt(digitalPinToInterrupt(10), _ISR, RISING);
}

void loop() {
  //any code here
}


void _ISR() {
  x = !x;
}
```

4.A.2 Speed Controller for Second Side of Robot

Below shows our completed breadboard circuit. The left and right speed controller circuits are identical, however some resistor and capacitor values may be tweaked later on in the lab to attain similar wheel speeds at the same speed voltage.



Completed Speed Controller Circuit for Both Sides of Robot

This circuit design was built and tested incrementally using some of the same experimental techniques as in earlier labs from this class. A successful build was indicated by being able to control both the left and right wheels using a potentiometer. All group members were able to successfully complete this step and control all 4 wheels.

4.A.3 Test Arduino

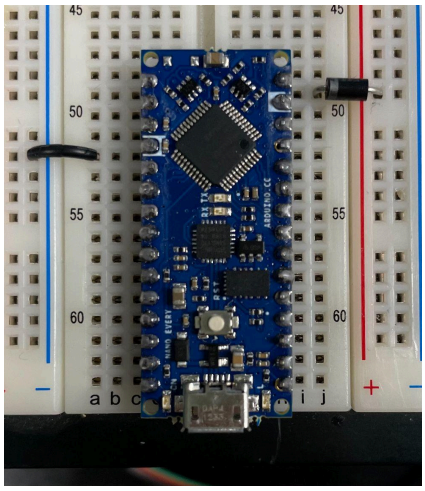
Below shows our code used to blink the LED for 2 seconds on and 0.2 seconds off. The input parameter for the delay function is in units of milliseconds.

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(2000);                       // wait for 2 seconds
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(200);                       // wait for 0.2 seconds
}
```

Blink LED Example with Specified Modifications

4.A.4 Power Arduino

Below is a picture of our power setup for the Arduino, which is very straightforward. We simply connect one of two available ground pins to our ground rail, as well as powering 8V into our Vin pin. We use a diode for this connection so that, when our 8V supply is turned off, our Arduino doesn't try to power the entire circuit here. The diode has a breakdown voltage high enough to ensure this will never happen.



4.A.5 Connect Vref Outputs

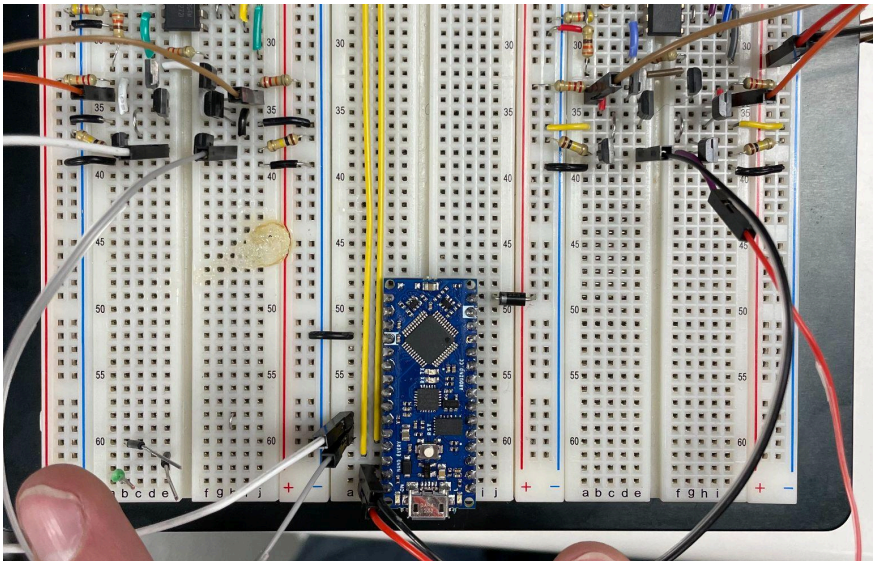
In order to convert our PWM signals to analog signals we use an RC LPF. We want a peak to peak voltage of about 0.1 V with a PWM standard frequency of 970 Hz. We know that the cutoff frequency for an RC LPF is $\omega = \frac{1}{RC}$. At a max voltage of 5V, we want to attenuate our signal by a factor of 0.02 = 1/50. Thus we take the ratio $\frac{1}{50\omega} = \frac{1}{RC}$ and solve for RC. This gives us values of approximately C = 1μF and R = 10$k\Omega$, since we have both of these values in our kits.

4.A.6 Adjust the Speed Control Feedback Circuits

In order to check that we attain a maximum motor speed of $V_S \approx 5V$, we run a V_{REF} of 5V from our Arduino Nano Every using the given code. This V_S is dependent on both the V_{REF} and our C2, R2 values. After much experimentation we found that an R2 = $3.3k\Omega$ and C2 = $100nF$ gives the best results. We then want to test that our motors run at the same speed for different values of V_{REF}. We found that there are some inconsistencies between the left and right wheels so in order to account for this we adjust the output of our V_{Ref} at the corresponding Pin number in our Arduino code. For example if our V_{enc} frequency for our right wheels is 1.5kHz at a $V_{ref} = 4$ V and our V_{enc} of our left wheels is 1.8kHz, we create a ratio and multiply it by our V_{ref} at the right wheels so that both F_{enc} are around 1.8 kHz. To confirm that these adjustments are accurate, we place our robot on a flat surface and ensure that it moves forward at these different V_{REF}'s.

4.A.7 Connect Motor Control Direction to Arduino

Now that we have our Arduino wired up, we can use the direction control circuit we have already built to implement direction control. We do so by adding a new set of NMOS transistors to each circuit, which we can use to ground either VB1 or VB2 by opening or closing the gates. Thus we will need to make two Arduino connections for each set of wheels. We make these connections to the digital out pins, specifically, D7 and D8 for the left wheels, and D11 and D12 for the right wheels. A picture is shown below.



Direction Control and PWM Arduino Outputs Shown on Breadboard

4.A.8 Robot Speed Control with Arduino

In order to test the functionality of the motor direction control circuit with the Arduino, a simple robot rotation test will be conducted. We will write an Arduino script that will wait for a button to be pressed, delay for 1 second, rotate the robot 360 degrees clockwise, stop and wait another second, then rotate 360 degrees counter clockwise, then stop. A successful test will have the robot back at the origin after both of the rotations.

To change the direction of the wheels, we first make every wheel direction pin low, then write the desired direction pin high. After this is done, there is a time delay while the robot moves. After trial and error, we finalized the code for a successful motor direction test:

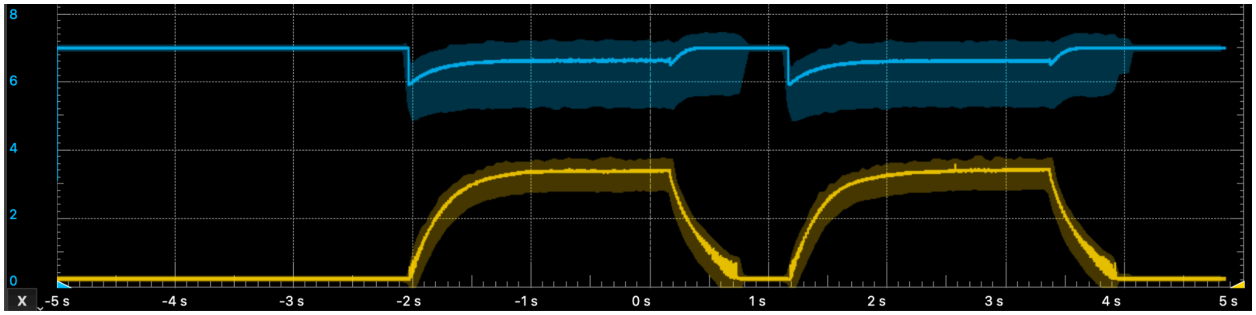
```
// define pins
const int pinON = 6;
const int FWD_Right = 7;
const int BACK_Right = 8;
const int pinRightPWM = 9;
const int pinLeftPWM = 10;
const int FWD_Left = 11;
const int BACK_Left = 12;

void setup() {
  // put your setup code here, to run once:
  pinMode(pinON, INPUT_PULLUP); //Set up button mode
  pinMode(FWD_Right, OUTPUT); //Set direction pins and PWM as outputs
  pinMode(BACK_Right, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);
  pinMode(FWD_Left, OUTPUT);
  pinMode(BACK_Left, OUTPUT);

  digitalWrite(FWD_Right, LOW); //Set all directions modes intially to low
  digitalWrite(BACK_Right, LOW);
  digitalWrite(FWD_Left, LOW);
  digitalWrite(BACK_Left, LOW);

  analogWrite(pinRightPWM, 5*51); //Vref = 5 V
  analogWrite(pinLeftPWM, 5*51); //Vref = 5 V

}
```

Code for Motor Direction Control With Arduino Test 1

```
void loop() {
  // put your main code here, to run repeatedly:
  do {} while (digitalRead(pinON) == HIGH); //Polling loop to check for button press

  delay(1000); //Delay 1 second after button press

  digitalWrite(BACK_Right, HIGH); //Set direction mode pins for clockwise rotation
  digitalWrite(FWD_Left, HIGH);

  delay(2200); //Delay during clockwise rotation

  digitalWrite(BACK_Right, LOW); //Stop robot after rotation
  digitalWrite(FWD_Left, LOW);

  delay(1000); //Delay 1 second

  digitalWrite(FWD_Right, HIGH); //Set direction mode pins for counter clockwise rotation
  digitalWrite(BACK_Left, HIGH);

  delay(2200); //Dealy during counter clockwise rotation

  digitalWrite(FWD_Right, LOW); //Stop robot and enter loop to poll for another button press
  digitalWrite(BACK_Left, LOW);
}
```

Code for Motor Direction Control With Arduino Test 2

Another way to verify this test is to look at the voltage waveforms of speed and controller output while the code is running. By scoping these waveforms with the Analog Discovery 2, the following graph is produced:



Button Press at t=-3s, Yellow = Vs, Blue Line = Vo

For this graph, the button was pressed at t=-3 seconds, after the one second delay Vs starts to increase at t=-2. Then, the robot spins while at maximum speed, before completing the spin and stopping at a little over zero seconds. There is a bit of a delay while Vs decays to zero, but this is not an issue because at this speed the wheels can not overcome friction and move. So even though Vs is at zero volts for only about a half second, the difference between where the robot stops the first spin and starts again for the second spin is the desired 1 second.

4.A.9 Robot Movement Repeatability Using Speed Control

Now that the robot has been tested for direction control while rotation, the next step is to add forward movements. Both of these things will ensure proper control when moving the robot. During this test, the robot will take the path described by the Arduino script, eventually returning to its original location.

In the Arduino script, we want to robot to follow these movements:

1. Stop and wait for button press on D6 switch
2. Wait 1 second
3. Move forward ~2 feet
4. Rotate 180 degrees clockwise
5. Move forward ~2 feet
6. Rotate 180 degrees counterclockwise
7. Stop in starting position

```
// define pins
const int pinON = 6;
const int FWD_Right = 7;
const int BACK_Right = 8;
const int pinRightPWM = 9;
const int pinLeftPWM = 10;
const int FWD_Left = 11;
const int BACK_Left = 12;

void setup() {
  pinMode(pinON, INPUT_PULLUP); //Set mode for pushbutton
  pinMode(FWD_Right, OUTPUT); // Set direction pins and PWM pins as output
  pinMode(BACK_Right, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);
  pinMode(FWD_Left, OUTPUT);
  pinMode(BACK_Left, OUTPUT);

  digitalWrite(FWD_Right, LOW); //Initially write all direction pins low
  digitalWrite(BACK_Right, LOW);
  digitalWrite(FWD_Left, LOW);
  digitalWrite(BACK_Left, LOW);

  analogWrite(pinRightPWM, 5*51); //Vref = 5 V
  analogWrite(pinLeftPWM, 5*51); //Vref = 5 V
}
```

Code for Robot Movement Repeatability Using Speed Control 1

```
void loop() {

  do {} while (digitalRead(pinON) == HIGH); //Polling loop to wait for button press on pin D6

  delay(1000); //Delay 1 second after button press

  digitalWrite(FWD_Right, HIGH); //Change wheel directions to move foward
  digitalWrite(FWD_Left, HIGH);

  delay(2000); // Delay during foward movement

  digitalWrite(FWD_Right, LOW); //Change directions for clockwise turn
  digitalWrite(BACK_Right, HIGH);

  delay(1250); // Delay during 180 deg clockwise turn

  digitalWrite(BACK_Right, LOW); //Change wheel directions to move foward after turn
  digitalWrite(FWD_Right, HIGH);

  delay(2000); // Delay during foward movement

  digitalWrite(FWD_Left, LOW); //Change directions for clockwise turn
  digitalWrite(BACK_Left, HIGH);

  delay(1200); // Delay during 180 deg counter clockwise turn

  digitalWrite(FWD_Right, LOW); //Stop movement to allow entry into putton push polling loop
  digitalWrite(BACK_Left, LOW);
}
```

Code for Robot Movement Repeatability Using Speed Control 2

After adjusting our delays a few times in order to achieve accurate results. We were able to get our robot back to its starting position almost perfectly. We look forward to using more strategic and accurate techniques in the upcoming part of the lab.

4.B Lab Exploration Topic

N/A

4.B.2 Estimating Interrupt Overhead

In part A of this lab, we executed all movement control of the robot through hard coded instructions and polling loops. To further increase our control of the robot, we will count the number of pulses coming from the motor encoder. Then we can employ the use of software interrupts to measure distance traveled and control movements. This will make it easier to control the robot compared to the time delays from part A.

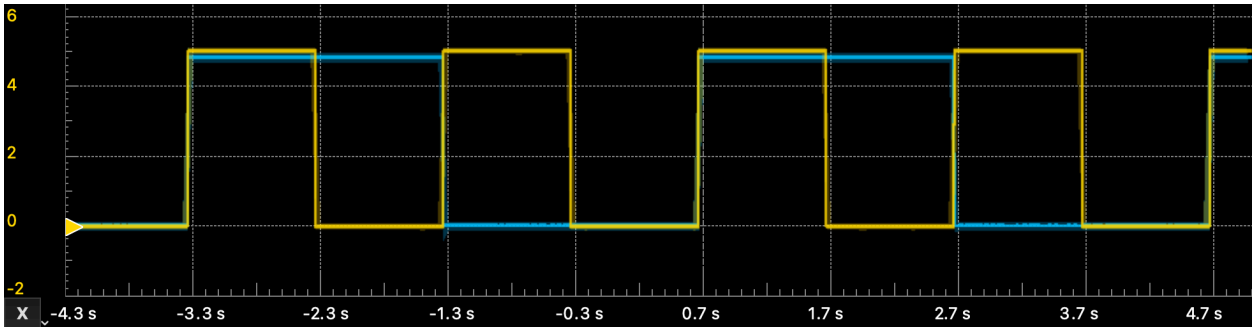
The first step to adding an interrupt service routine to the Arduino script is to estimate the number of clock cycles the interrupt overhead takes. When controlling the robot, it needs to be known how long the switch from the main loop to the interrupt service routine takes.

In order to test this, we use a rising edge triggered interrupt on one of the Arduino input pins. The input voltage will be a square waveform provided by the AD2. The code for this test is as follows:

```
volatile bool var=LOW;

void setup() {
  pinMode(6, INPUT); //Arduino D6 pin as input
  attachInterrupt(digitalPinToInterrupt(6), service6, RISING); //Attach rising edge interrupt
  pinMode(13, OUTPUT); //Arduino D13 pin as output
}
void loop() {
  digitalWrite(13, var); //Write value of var to D13 output pin
}
// Interrupt Service Routine
void service6() {
  var = ! var; //Flip value of var when interupt is triggered
}
```

This code takes an input on pin D6, and at each rising edge of the input waveform, toggles the output pin D13 on and off. We will increase the frequency of the input voltage waveform, looking at if there are any pulses skipped in the output. Once the frequency is high enough that pulses are skipped, we know that the period of that wave is equal to the amount of time the interrupt overhead takes since it can no longer keep up.

When looking at the input and output waveforms, an example of no skipped pulses is shown below. Notice how on each rising edge of the D6 input, the output D13 toggles.

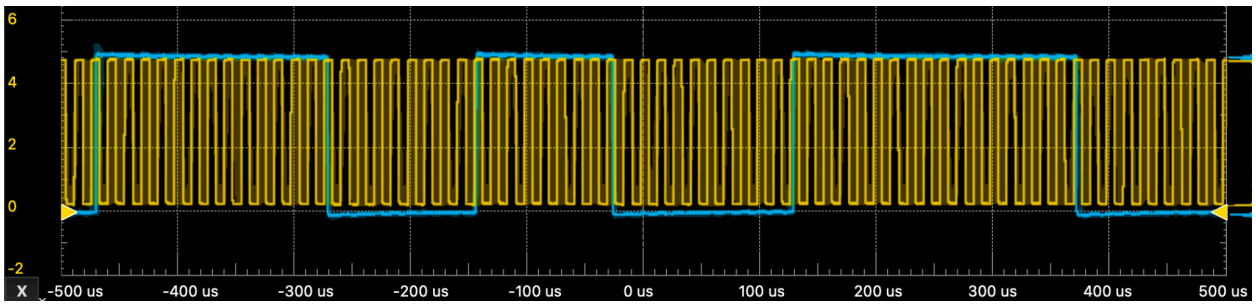


Output with no Skips, Yellow: Input Voltage, 500 mHz, Blue: Pin D13 Output

We did many tests increasing the input frequency, eventually skipping too far, then having to go back in smaller increments to find the first frequency when pulses were skipped. The following table summarizes this test.

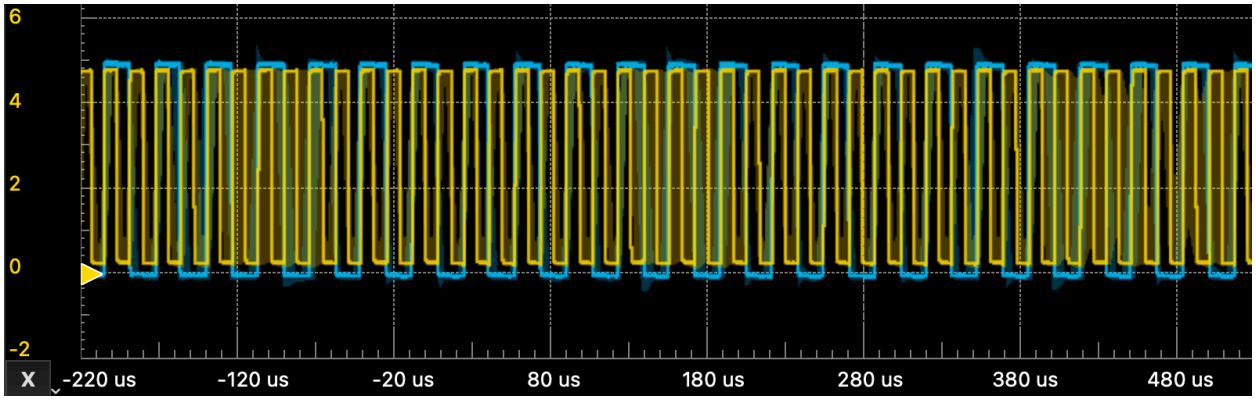
Input Frequency	Pulse Skip	Notes
500 mHz	No	
500 Hz	No	
1kHz	No	
10kHz	No	Short delay
20kHz	No	Short delay
30kHz	No	Short delay
40kHz	No	~0.5 period delay
50kHz	No	~0.75 period delay
60kHz	No	
70kHz	Yes	Large skips, Try smaller increments
61kHz	No	
62kHz	Yes	Lowest skipping frequency

When incrementing the frequency by 10 kHz each test, we accidentally overshot the smallest frequency. This can be seen in the large amount of skips in the following graph:



Interrupt Overhead Test: 70 kHz, Yellow: D6 input, Blue: D13 output

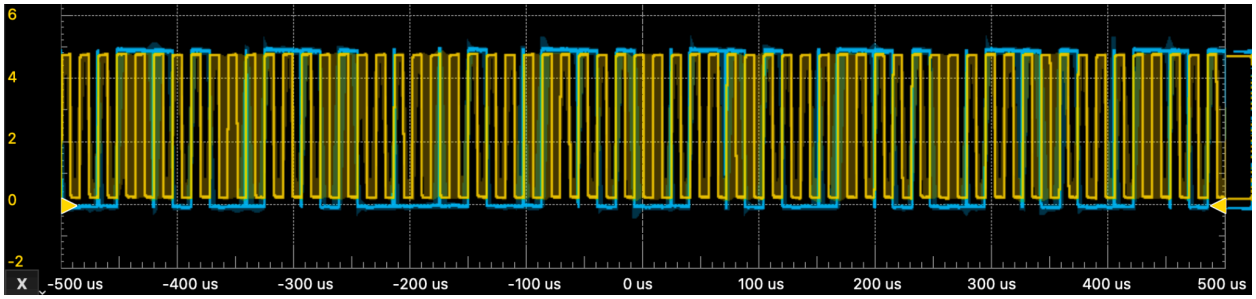
70kHz was too fast, while there still was no skipping at 60kHz. In order to get a more exact frequency, we then tested 1kHz increments starting at 61kHz.



Interrupt Overhead Test: 61 kHz, Yellow: D6 input, Blue: D13 output

61kHz works as intended, with no skipped pulses.

After another increment, 62kHz is where pulses are first skipped:



Interrupt Overhead Test: 62 kHz, Yellow: D6 input, Blue: D13 output

This frequency is used to estimate the overhead of the ISR, or the number of clock cycles used by the CPU for the Interrupt Service Routine . The first skipping of pulses occurred at 62kHz, so we will assume 61kHz is the largest frequency that works. To find the number of clock cycles this takes, we need to use the period of this last non skipping frequency, and the clock speed of the megaAVR processor on the Arduino Nano Every.

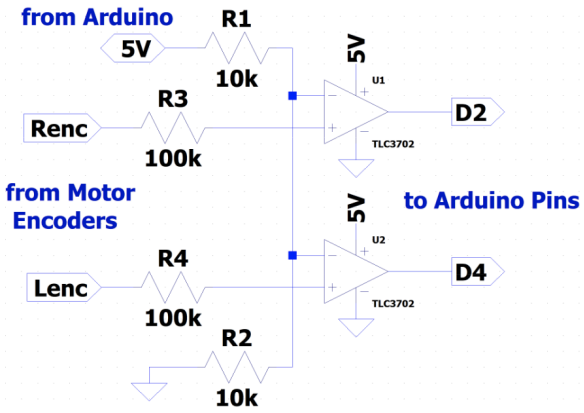
Since f = 1 / T, it can be calculated that 61kHz relates to a period of 0.01639 ms. The processor has a clock frequency of 16MHz, meaning that in one second, 16 million clock cycles occur. Knowing this, a ratio relation finds the number of clock cycles for the interrupt overhead.

(0.01639 ms) / (ISR overhead clock cycles) = (1000 ms) / (16 000 000 clock cycles)

The ISR overhead is estimated to be **262 clock cycles.**

4.B.3 Using Comparators as Level Shifters

Since we are going to use interrupts and distance based control, we will need to use the encoder pulse signal as our Arduino reference. We know that the Arduino pins cannot take more than 5V in, however the encoder pulse is roughly 8V. Thus, we will use the comparator circuit shown below to step down these signals to 5V. We used our TLC3702 opamp and four resistors from our kit to implement the schematic shown below.

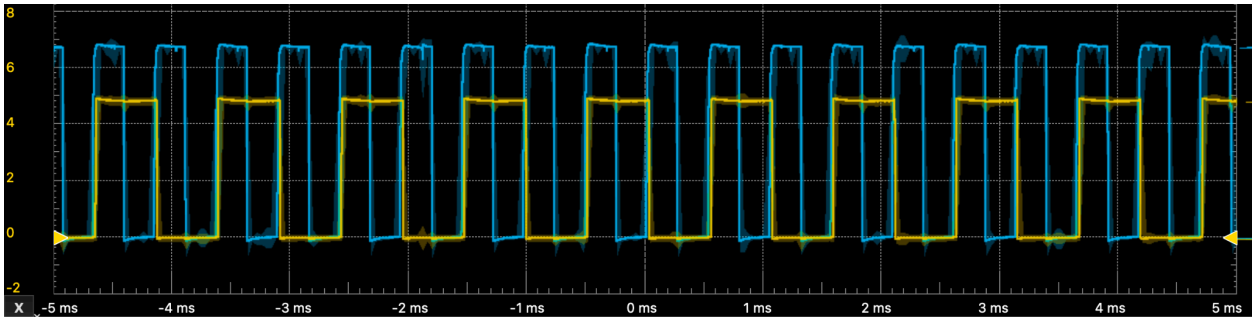


Comparator Circuit Schematic

Something else that needs to be done is ensure that these encoder pulses can be counted for later use in distance tracking ISRs. In order to do that, we run an edited version of the test in 4.B.2, using encoder frequency instead of a generated square waveform, and new input pins D2 and D4 for the scaled down voltages from R_{enc} and L_{enc} respectively.

```
volatile bool varL = LOW;
volatile bool varR = LOW;

//define pins
const int pinON = 6;              //D6 on/off switch, active low
const int pinLeftForward = 11;    //D11 left forward
const int pinLeftBackward = 12;   //D12 left backward
const int pinLeftPWM = 10;        //D10 left vref

const int pinRightForward = 7;    //D7 right forward
const int pinRightBackward = 8;   //D8 right backward
const int pinRightPWM = 9;        //D9 right vref

void setup() {
  //interrupts
  attachInterrupt(digitalPinToInterrupt(4), countLeft, RISING);
  attachInterrupt(digitalPinToInterrupt(2), countRight, RISING);

  //pin setup
  pinMode(13, OUTPUT);
  pinMode(pinLeftForward, OUTPUT);
  pinMode(pinLeftBackward, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);
  pinMode(pinRightForward, OUTPUT);
  pinMode(pinRightBackward, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);

  //start movement
  analogWrite(pinLeftPWM, 5*51);
  analogWrite(pinRightPWM, 5*51);
  digitalWrite(pinLeftForward, HIGH);
  digitalWrite(pinRightForward, HIGH);

  //ISR checks
  pinMode(3, OUTPUT);
  pinMode(5, OUTPUT);
}
void loop() {
  digitalWrite(3, varL);
  digitalWrite(5, varR);

}
void countLeft() {
  varL = !varL;
}
void countRight() {
  varR = !varR;
}
```

Example Code for Determining CPU Cycles Available to Program ISR with Wheels at Full Speed



Right Wheels ISR Test, Blue: Encoder Frequency at Full Speed, Yellow: D13 Output

The voltage output as adjusted by the ISR is able to keep up with the motor encoder frequency when running at the maximum Vref of 5 V, indicating that one period is how long the CPU has

available to program the ISR when the wheels are at full speed. An identical test was run on the left wheels, with the same results.

We also need to understand how much overhead in the code that we have in order to write programs that don't interfere with our distance/interrupt based code. Assuming that our encoder frequency runs at 50kHz, and that our ATmega4808 processor operates with a 16MHz clock cycle, the calculations are relatively simple. The encoder frequency has a period of 20us, which is the amount of time we have to execute code. Each clock cycle is 62.5ns long. Thus we divide the time we have to execute code, 20us, by the time it takes to execute one clock cycle, 62.5ns, which yields 320 clock cycles.

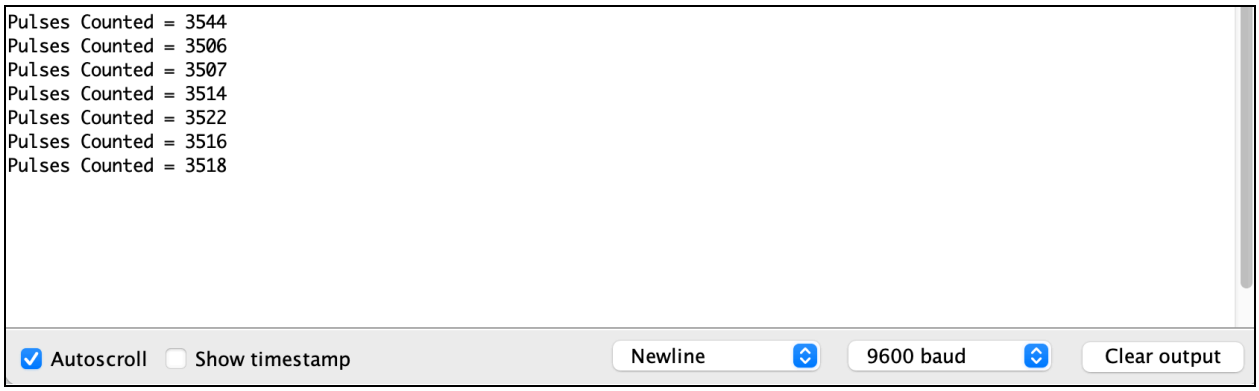
When programming, we need to remember that it takes a few clock cycles to get in and out of the ISR, and that we will be executing lines of code within our main loop as well as the ISR.

4.B.4 Reading Encoder with Arduino

We have now devised a method to trigger an interrupt on the rising edge of every other encoder pulse. This extends well to devising a method to find the relation between number of pulses and distance traveled by the robot. In order to do this, we will let the robot travel straight for a certain amount of time at full speed, and measure the distance traveled. Then, the distance traveled can be divided by the number of encoder pulses in that timeframe to find the relationship between time, distance, and encoder pulses. In order to find the number of encoder pulses in that timeframe we will increment a counter each time the Encoder ISR is triggered on the f_{enc} rising edge. This interrupt counter, when mixed with the button press polling loop and a time delay, will count how many pulses occur with the motors running for a given amount of time.

It’s important to note that in this lab when we refer to measuring the number of pulses we are strictly measuring the number of pulses generated for one side. In reality, the number of pulses in total is roughly doubled considering we have two sides of our robot running at a time.

For a two second delay, our robot traveled 2 feet, 3 inches, which is 685.5 mm. For this same time period we measured on average 3519 pulses. This gives us a ratio of **0.195 mm/pulse**.



Terminal Output Showing Number of Pulses Running for 2 Seconds Each Loop

After attaining this ratio, we can confirm that the pulse counting is correct by allowing our robot to move forward for a different amount of time, say 3 seconds. Given this timeframe we can once again see how many pulses were generated. Multiplying this pulse count by our ratio found previously, we ought to get an accurate measurement for the distance traveled by the robot during the new timeframe. If this result is indeed accurate, we can confirm that our pulse counter is correctly counting the number of pulses.

Running the test again for a timeframe of 3 seconds, we generate an average of 5253 pulses. Thus we expect our robot to travel roughly 40.3 inches. After performing the actual test on the robot we found that it did indeed travel ~40 inches. This confirms the correct counting of encoder pulses.

Our Arduino code is shown below for performing these tests.

```
volatile int enc_pulse_count;

// define pins
const int pinON = 6;
const int FWD_Right = 7;
const int BACK_Right = 8;
const int pinRightPWM = 9;
const int pinLeftPWM = 10;
const int FWD_Left = 11;
const int BACK_Left = 12;

void setup() {
  //pinMode(2, INPUT);
  pinMode(4, INPUT);
  //attachInterrupt(digitalPinToInterrupt(2), encoder_irq, RISING); //2 for right side, 4 for left side
  attachInterrupt(digitalPinToInterrupt(4), encoder_irq, RISING); //2 for right side, 4 for left side

  pinMode(pinON, INPUT_PULLUP); //Set mode for pushbutton
  pinMode(FWD_Right, OUTPUT); // Set direction pins and PWM pins as output
  pinMode(BACK_Right, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);
  pinMode(FWD_Left, OUTPUT);
  pinMode(BACK_Left, OUTPUT);

  digitalWrite(FWD_Right, LOW); //Initially write direction foward for both R and L
  digitalWrite(BACK_Right, LOW);
  digitalWrite(FWD_Left, LOW);
  digitalWrite(BACK_Left, LOW);

  analogWrite(pinRightPWM, 5*51); //Vref = 5 V, Run right wheels at maximum speed
  analogWrite(pinLeftPWM, 5*51); //Vref = 5 V, Run left wheels at maximum speed

  Serial.begin(9600); //Enable serial terminal to output counted pulses to
}
```

```
void loop() {
  enc_pulse_count = 0; //Restart pulse count for next loop iteration
  delay(25);

  do {} while (digitalRead(pinON) == HIGH); //Polling loop to wait for button press on pin D6

  delay(1000); //Delay 1 second after button press

  digitalWrite(FWD_Right, HIGH); //Adjust direction to foward
  digitalWrite(FWD_Left, HIGH);

  delay(2000);

  digitalWrite(FWD_Right, LOW); //Stop robot
  digitalWrite(FWD_Left, LOW);

  Serial.print(F("Pulses Counted = ")); //Print number of counted pulses
  Serial.println(enc_pulse_count);
}

// Interrupt Service Routine to count number of encoder pulses
void encoder_irq() {
  enc_pulse_count++;
}
```

Arduino Script Outputting Encoder Pulses in 2 Seconds to Serial Window

4.B.5 Position Control by Counting Encoder Pulses

Now that we have a ratio for the distance traveled per pulse, ideally we can use this value and multiply it by a number of pulses until we reach a threshold distance and then turn off our wheels and we should be at a given threshold distance away from where we began. Unfortunately, the world is not that simple since we must account for the inertia in the wheels.

To do this we gradually increase and decrease the start-up and slow-down speeds respectively. We use a for loop to change our Vref from 0 to 4 when increasing and from 4 to 0 when decreasing our speed with a delay of 100 ms at each iteration. Then, using a serial print we can figure out how many encoder pulses on average these processes take. We find the total number of pulses for speeding up is about 300 encoder pulses and 700 encoder pulses when slowing down. For this reason we expect our robot to travel 1000*0.195 = 195 mm as we start up and slow down.

Between the two start up and slow down for loops we have a do while loop that tells our robot to continue running at full speed as long as our encoder pulses * distance per pulse remains below a threshold distance. We add 1000 to the encoder pulse count within this condition due to the fact that this is the encoder pulses not accounted for within the other for loops. Using this method we can input any distance and have our robot attain it very accurately.

Testing this with a distance of 2 feet, we found that it is almost perfect.

We can do similar tests to find a ratio for the angle turned per encoder pulse. We could do this by trial and error, simply setting a threshold encoder pulse until our robot travels 360°. This way we then multiply by any desired degree and set this as the threshold value of encoder pulses. After performing this procedure we found that on one robot, a counter clockwise degree turn of 180° takes ~ 1925 encoder pulses where a clockwise turn of 180° takes only 1600 encoder pulses. This is due to inconsistencies in each side's motors and wheels.

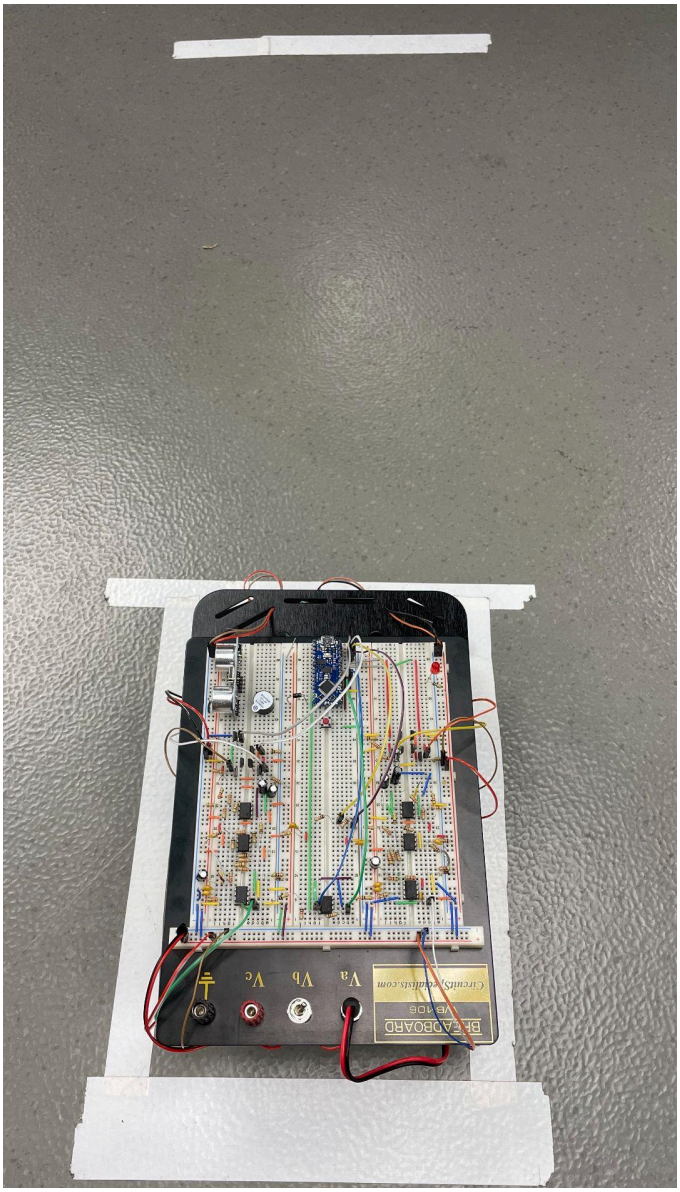
The main body of the code used to travel a distance of two feet is shown below. The setup code has already been shown in previous parts therefore we only show the loop section here.

```
//----------- Gradually Increase Speed ----------------------------------

  for(float i=0; i < 5; i++){
    analogWrite(pinRightPWM, i*51);
    analogWrite(pinLeftPWM, i*51);
    digitalWrite(FWD_Right, HIGH);
    digitalWrite(FWD_Left, HIGH);
    delay(100);
  }

//----------- Run a main speed for specified distance ---------------------------------
  do {
    digitalWrite(FWD_Right, HIGH); //Adjust direction to foward
    digitalWrite(FWD_Left, HIGH);

  } while(((enc_pulse_count+1000.0)*0.195) < Distance);


//----------- Gradually Decrease Speed ----------------------------------
  for(float i=4; i >=0; i--){
    analogWrite(pinRightPWM, i*51); //Vref = 4 V, Run right wheels at maximum speed
    analogWrite(pinLeftPWM, i*51); //Vref = 4 V, Run left wheels at maximum speed
    digitalWrite(FWD_Right, HIGH);
    digitalWrite(FWD_Left, HIGH);
    delay(100);
  }
```

Void Loop Section of Code for Traveling any given "Distance" in mm

4.B.6 Position Control Tests

Now that we have implemented position control using the encoder pulses, we are ready to put our robot to the test. Using the code shown below, we were able to move forward two feet, spin 180 degrees, move forward two feet, then spin 180 degrees back to the starting position. We were able to repeat this test successfully, the start of each test being a button press. Note that the picture below contains two elements that are not discussed in this lab- a piezo speaker and an ultrasonic sensor. These devices have not been integrated into our circuits and are entirely unrelated.



Position Control Test Setup

```
volatile int enc_pulse_count;

// define pins
const int pinON = 6;
const int FWD_Right = 7;
const int BACK_Right = 8;
const int pinRightPWM = 9;
const int pinLeftPWM = 10;
const int FWD_Left = 11;
const int BACK_Left = 12;

void setup() {
  pinMode(4, INPUT);
  attachInterrupt(digitalPinToInterrupt(4), encoder_irq, RISING); //Left side pulse
  pinMode(2, INPUT);
  attachInterrupt(digitalPinToInterrupt(2), encoder_irq, RISING); //Right side pulse

  pinMode(pinON, INPUT_PULLUP); //Set mode for pushbutton
  pinMode(FWD_Right, OUTPUT); // Set direction pins and PWM pins as output
  pinMode(BACK_Right, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);
  pinMode(FWD_Left, OUTPUT);
  pinMode(BACK_Left, OUTPUT);

  digitalWrite(FWD_Right, LOW); //Initially write direction foward for both R and L
  digitalWrite(BACK_Right, LOW);
  digitalWrite(FWD_Left, LOW);
  digitalWrite(BACK_Left, LOW);

  analogWrite(pinRightPWM, 5*51); //Vref = 5 V, Run right wheels at maximum speed
  analogWrite(pinLeftPWM, 2.5*51); //Vref = 5 V, Run left wheels at maximum speed
}
```

```
void loop() {
  enc_pulse_count = 0;
  delay(25);

  do {} while (digitalRead(pinON) == HIGH); //Polling loop to wait for button press on pin D6

  delay(1000); //Delay 1 second after button press

//---------- Move Foward 2 Feet ----------------------------------
  do {
    digitalWrite(FWD_Right, HIGH); //Adjust direction to foward
    digitalWrite(FWD_Left, HIGH);

  } while(enc_pulse_count<6000);

//---------- 180 Clockwiswe Spin ----------------------------------
    do {
    digitalWrite(FWD_Right, LOW); //Adjust direction
    digitalWrite(BACK_Right, HIGH);
  } while( (enc_pulse_count>=6000) && (enc_pulse_count < 9300) );

 //---------- Move Foward 2 Feet ----------------------------------
  do {
    digitalWrite(BACK_Right, LOW); //Adjust direction to foward
    digitalWrite(FWD_Right, HIGH);

  } while( (enc_pulse_count>=9300) && (enc_pulse_count < 16100) );

//---------- 180 Counter Clocksise Spin ----------------------------
    do {
    digitalWrite(FWD_Left, LOW); //Adjust direction
    digitalWrite(BACK_Left, HIGH);
  } while( (enc_pulse_count>=16100) && (enc_pulse_count < 21000) );
 //----------------------------------------------------------------

  digitalWrite(FWD_Right, LOW); //Stop Robot
  digitalWrite(BACK_Left, LOW);
```

```
// Interrupt Service Routine to count number of encoder pulses
void encoder_irq() {
  enc_pulse_count++;
}
```

Arduino Script for Final Position Control Test

Conclusion

This was probably the most satisfying lab we completed. Building upon all of our previous hard work, we achieved all of the objectives we outlined in the introduction, and were successfully able to control the robot's movement using the Arduino Nano Every. The main conclusions drawn were software related, and these ideas will help us in the next lab, where we make some custom modifications to the robot. We learned about polling, interrupts, and clock cycles, to name a few.

One limitation of the lab was the battery supply. The battery supply was relatively unreliable, and the longevity of the batteries was another problem. In previous labs, we could just power our robot with one of the lab's DC power supplies, but actually testing our movement on the ground required a reliable battery supply. Another limitation was the fact that the Arduino sits at the bottom of the breadboard and a lot of the voltage nodes it needs to "talk to" were at opposite ends of the breadboard, requiring us to use some long, loopy wires. After practicing clean breadboard circuit design all semester long, using these long wires just didn't feel right. Another, somewhat smaller, limitation is that we didn't have a ruler or measuring tape. That would probably be pretty helpful to have in the kit, although having one handy in any toolkit is probably good practice.

The two most important things we learned were how to implement interrupts in the Arduino IDE software and how assembly code works in correlation with a processor's clock cycle. We will likely use interrupts when implementing our custom modifications, and they are an important feature of any software, so we are glad we understand how they work in the Arduino IDE software. Additionally, understanding the processing power of our Arduino, as well as the coding overhead will be particularly helpful in the next lab. With any microcontroller, it is important to understand what is going on behind the scenes.

Some improvements could be made in regards to adjusting for mechanical inconsistencies. The lab report pretty much just said to make sure the robot goes straight, but didn't explicitly state that this could need to be done electrically or in the software. Another improvement that could be made would be more time to rebuild the second half of the breadboard circuits. We spent the entire week for part A rebuilding the circuits for the other side of the robot and debugging them. If we had known about this earlier we probably would have started building these circuits over spring break, so that we wouldn't have wasted precious lab time on it. You don't really want to tell students to do work over spring break, but I think a lot of people would have completed this part of the lab over spring break if they had known, and the timing works out pretty well in that regard.