University of Colorado - Boulder

ECEN 2270
Electronics Lab | Spring 2024

ECEN 2270 Electronics Lab: Lab 4

Team Papa:
Gabriel Agostine
Sam Walker
Julian Werder
Jonah Yunes

Lab Instructor:
Steven Dunbar

Lab: Section 12

Sunday, March 21, 2024

College of Engineering & Applied Science
UNIVERSITY OF COLORADO **BOULDER**

I. Introduction

Experiment A dives into the integration of Arduino Nano Every for controlling robotic movements, highlighting the practical application of interrupts in microcontroller programming. We will build a compensator and tachometer circuit on the other side of the robot, which are crucial for balancing motion control and for precise speed measurements. This experiment serves as a fundamental introduction to Arduino and interrupts, focusing on their utility for handling time-sensitive tasks efficiently without constant polling. We also explore position control using Arduino's time delay capabilities, which allows for refined maneuvering of the robot through timed programming adjustments.

In Experiment B, our focus shifts to more advanced applications of interrupts in robotics, particularly for position control. We implement interrupts to accurately count encoder pulses, a vital aspect of gauging the robot's position and movement. Additionally, we use comparators as level shifters to facilitate this process. This setup allows us to employ encoder pulses for precise position control, enhancing the robot's ability to navigate and perform tasks with higher accuracy. Later, advanced position control strategies push the boundaries of what can be achieved with extra fine tuning of our encoder parameters to test our robots control capabilities.

II. Experiment A

A. Exploration Topic: Interrupts

1) **What are hardware and software interrupts, and how do they differ?**
   Hardware Interrupts:
   - Signals from hardware devices to CPU.
   - Asynchronous events triggered by hardware.
   - Temporarily halts current CPU operation.
   - Handled by interrupt service routines.
   - Managed by hardware, often via interrupt controllers.

   Software Interrupts:
   - Generated by the software itself.
   - Synchronous events are initiated by executing specific instructions.
   - Switches CPU to kernel mode.
   - Handled by OS interrupt handlers.
   - Used for system calls, error handling, and signaling within programs.

2) **What is polling?**
   Polling is the process where the computer or controlling device waits for an external device to check for its readiness or state, often with low-level hardware. For example, when a printer is connected via a parallel port, the computer waits until the printer has received the next character.

3) **How does an interrupt work?**
   An interrupt is an event that alters the sequence in which the processor executes instructions.

4) **What happens in the microcontroller program execution when an interrupt occurs?**
   The program stops executing and the microcontroller begins to execute the ISR.

5) **What is the interrupt structure of the Arduino Nano Every?**

```
attachInterrupt(iPin,service,FALLING); // Falling edge at iPin
```

6) **Write a simple Arduino program that demonstrates the use of interrupts.**

```
const int buttonPin = 2; // the pin number of the pushbutton
volatile int buttonState = 0; // variable for reading the pushbutton status

void setup() {
  pinMode(buttonPin, INPUT);
  attachInterrupt(digitalPinToInterrupt(buttonPin), buttonInterrupt, CHANGE);
  Serial.begin(9600);
}

```

```
void loop() {
  // Main program will continue running while the interrupt is waiting
}

void buttonInterrupt() {
  buttonState = digitalRead(buttonPin);
  Serial.print("Button state changed: ");
  Serial.println(buttonState);
}
```

B. 4.A.2

Here is the Feedback controller, tachometer, and direction control builts on both sides:

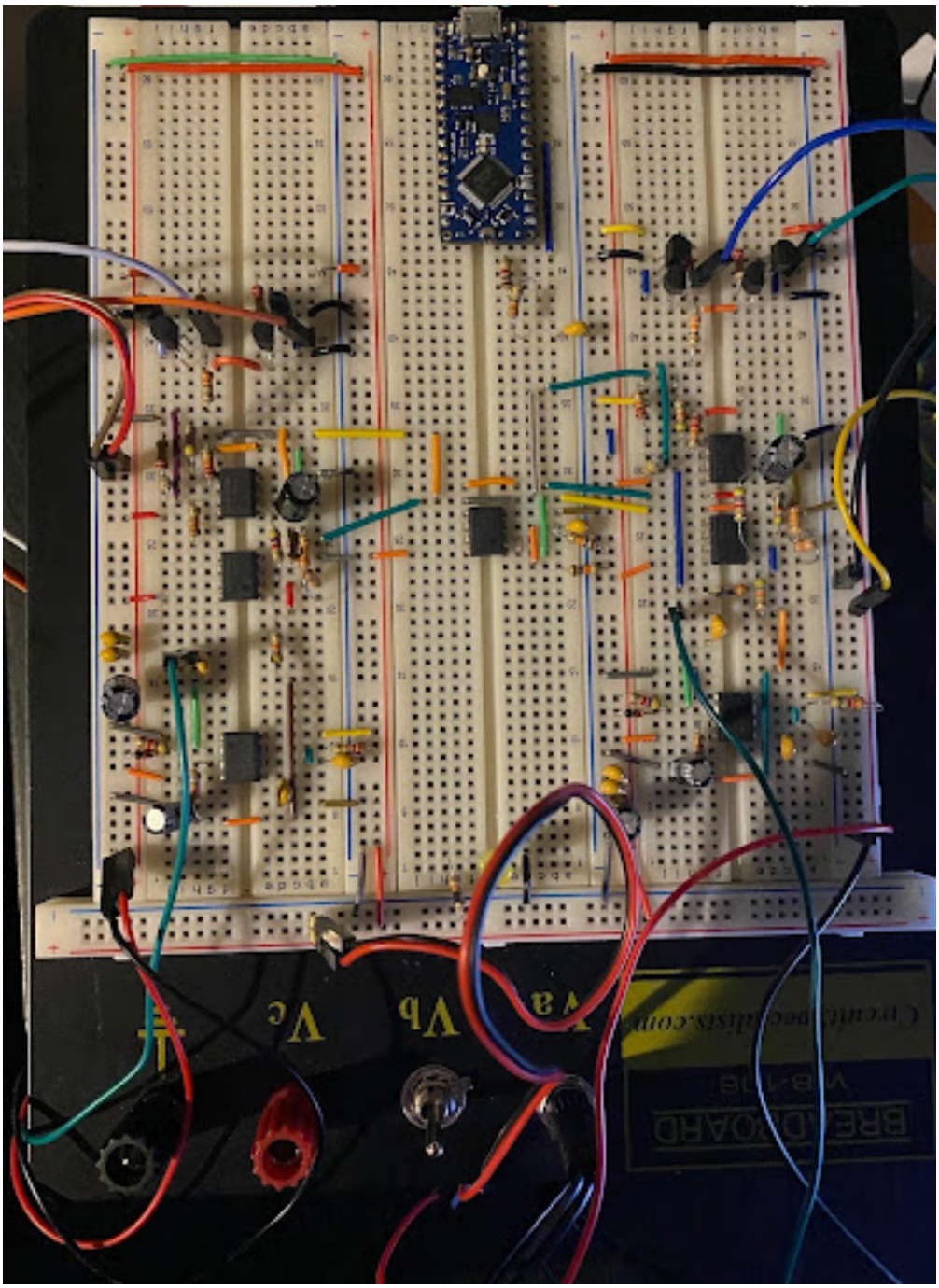

Fig. 1

C. 4.A.3

```
// 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 a second
  digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
  delay(200);                      // wait for a second
}
```

D. 4.A.4

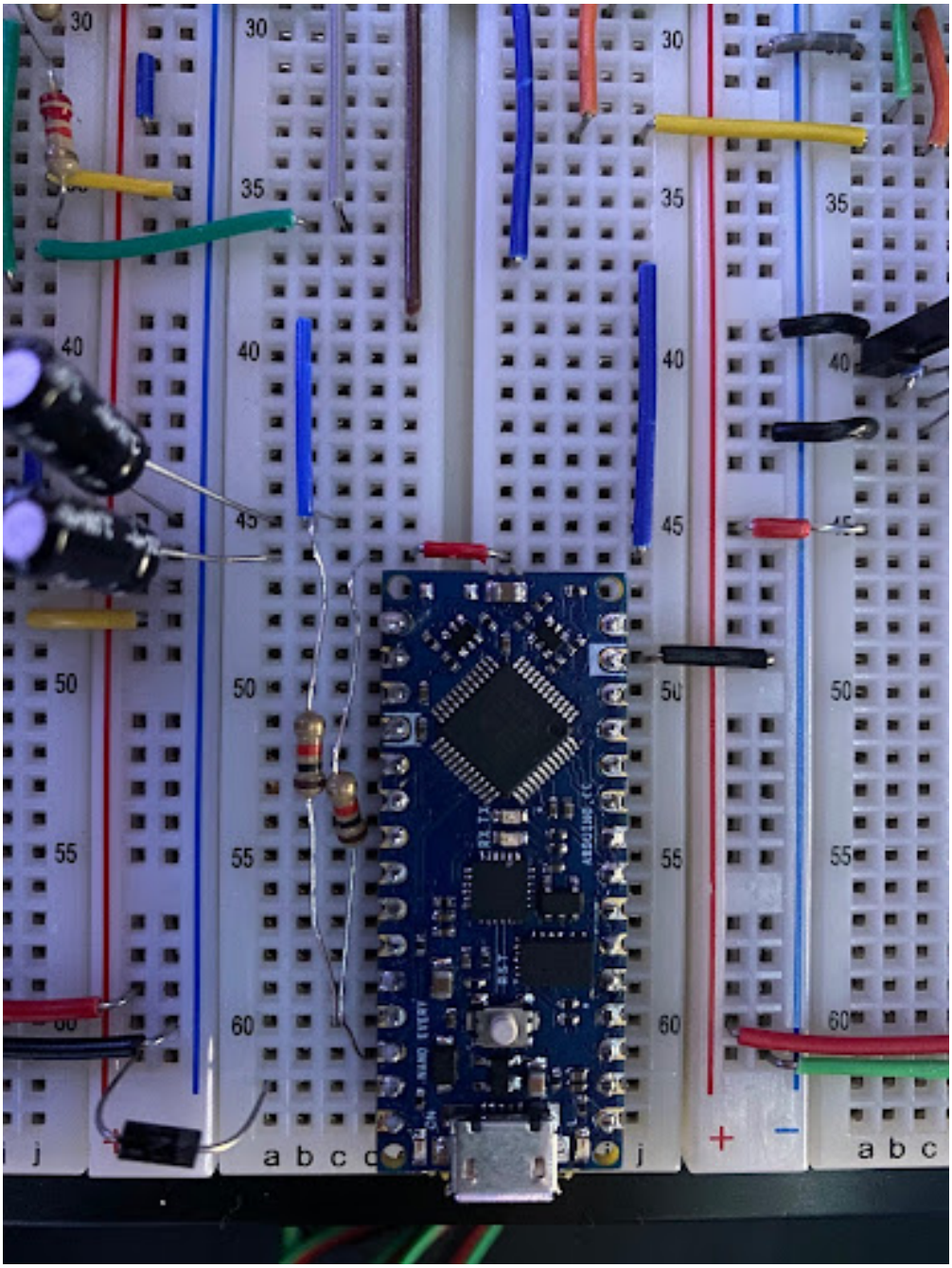

Fig. 2

E. 4.A.5

We use an Arduino Nano Every in order to command and control our robot and its direction control. As documented in previous reports, our closed-loop feedback control system has a reference voltage input. We can vary this reference voltage by varying the duty cycle of an outputted PWM signal from an Arduino. Thus, we must design a Low Pass Filter so that the output waveform is a constant DC voltage with a ripple voltage less than 100 mV.

We can define ripple voltage for a square wave as the difference between the maximum amplitude (5 V) of the squarewave and the voltage across a capacitor t_{off} seconds after the falling edge of the PWM signal. This can be expressed as:

$$V_{rip} = V_i \left(1 - exp\left(\frac{\%_{DC}}{100fRC}\right)\right)$$

For V_i = 5 V, V_{rip} = 0.1 V, f = 970 Hz, 50% Duty Cycle Wave and we assume C = 10µF; R = 2550Ω.

F. 4.A.6

Using the following code, we are able to confirm that both the left and right sides of our robots (specifically their corresponding circuits and motors) behave as expected.

```
// pin definitions
const int pinRightPWM = 9;
const int pinLeftPWM = 10;
const int value = 204;

void setup() {
  // put your setup code here, to run once:
  pinMode(pinRightPWM, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);
  analogWrite(pinRightPWM, value); // Vref = 5*(value/255)
  analogWrite(pinLeftPWM, value); // Vref = 5*(value/255)
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

We also changed the values of R_2 and C_2 within our speed control circuit so that at maximum speed it outputs 5V, to be consistent with our Arduino's PWM waveform. This was achieved by changing R_2 to 30kΩ and C_2 to 10nF.

V_{ref} [V]	$f_{enc_{Left}}$ [Hz]	$f_{enc_{Right}}$ [Hz]
1	278	353
2	524	649
3	783	950
4	1027	1257
5	1369	1585

Table 1 Wheel spin speed confirmation

The frequency values seen in **Table 1** show that the spin rates of the two motors are indeed not the same. While they are somewhat close, this means that we will likely have to account for positional error between the motors within the algorithm.

G. 4.A.7

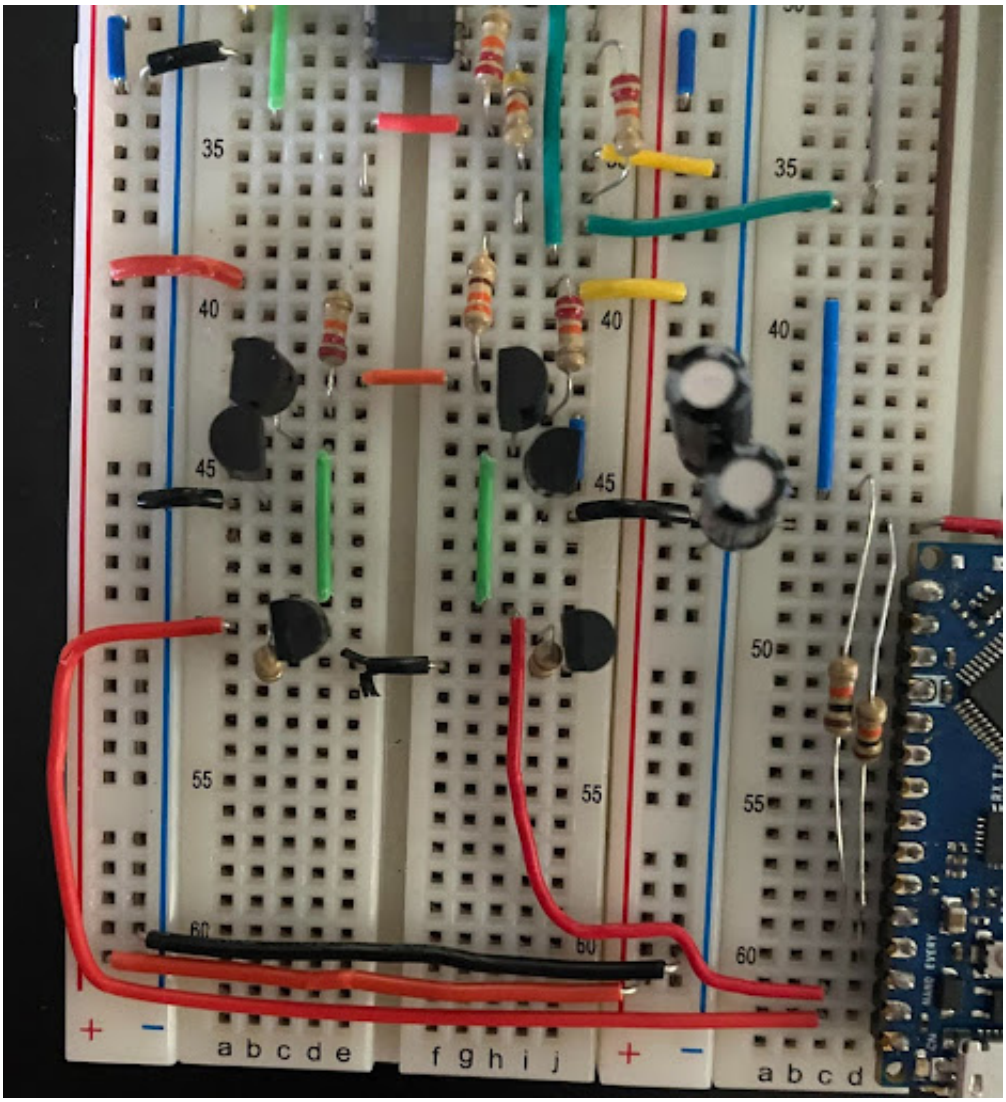

Fig. 3

H. 4.A.8

```
const int pinON = 6;
const int pinLeftForward = 11;
const int pinLeftBackward = 12;
const int pinLeftPWM = 10;
const int pinRightForward = 7;
const int pinRightBackward = 8;
const int pinRightPWM = 9;

void setup() {
  // put your setup code here, to run once:
 pinMode(pinON,INPUT_PULLUP);
 pinMode(pinLeftForward, OUTPUT);
 pinMode(pinLeftBackward, OUTPUT);
 pinMode(pinLeftPWM, OUTPUT);
 pinMode(13, OUTPUT);
 digitalWrite(pinLeftForward, LOW);
```

```
 digitalWrite(pinLeftBackward, LOW);
 analogWrite(pinLeftPWM, 4.5*51);


  pinMode(pinRightForward, OUTPUT);
  pinMode(pinRightBackward, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);
  digitalWrite(pinRightForward, LOW);
  digitalWrite(pinRightBackward, LOW);
  analogWrite(pinRightPWM, 4.5*51);
}

void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(13, LOW);
  do {} while (digitalRead(pinON) == HIGH);
  digitalWrite(13, HIGH);

  // Wait 1 second
  delay(1000);

  // Perform 360 deg clockwise rotation of the robot
  rotateClockwise();

  // Stop and wait 1 second
  stopMotors();
  delay(1000);

  // Perform 360 deg counter-clockwise rotation of the robot
  rotateCounterClockwise();

  // Stop and wait 1 second
  stopMotors();
  delay(1000);

}
void rotateClockwise() {
  digitalWrite(pinRightForward, LOW);
  digitalWrite(pinLeftBackward, LOW);
  digitalWrite(pinLeftForward, HIGH);
  digitalWrite(pinRightBackward, HIGH);

  delay(4000); // Adjust the delay according to your robot's rotation speed
}

void rotateCounterClockwise() {
  digitalWrite(pinLeftForward, LOW);
  digitalWrite(pinRightBackward, LOW);
  digitalWrite(pinLeftBackward, HIGH);
  digitalWrite(pinRightForward, HIGH);

  delay(4000); // Adjust the delay according to your robot's rotation speed
}

void stopMotors() {
  digitalWrite(pinLeftForward, LOW);
  digitalWrite(pinRightForward, LOW);
  digitalWrite(pinLeftBackward, LOW);
  digitalWrite(pinRightBackward, LOW);
}
```

I. 4.A.9

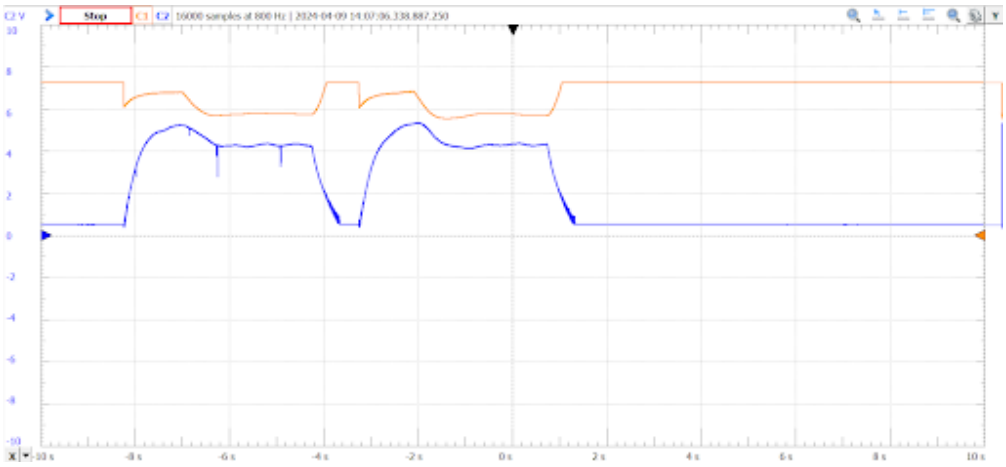

Fig. 4

```
const int pinON = 6;
const int pinLeftForward = 11;
const int pinLeftBackward = 12;
const int pinLeftPWM = 10;
const int pinRightForward = 7;
const int pinRightBackward = 8;
const int pinRightPWM = 9;


void setup() {
  // put your setup code here, to run once:
 pinMode(pinON,INPUT_PULLUP);
 pinMode(pinLeftForward, OUTPUT);
 pinMode(pinLeftBackward, OUTPUT);
 pinMode(pinLeftPWM, INPUT);
 pinMode(13, OUTPUT);
 digitalWrite(pinLeftForward, LOW);
 digitalWrite(pinLeftBackward, LOW);
 analogWrite(pinLeftPWM, 5*51);


  pinMode(pinRightForward, OUTPUT);
  pinMode(pinRightBackward, OUTPUT);
  pinMode(pinRightPWM, INPUT);
  digitalWrite(pinRightForward, LOW);
  digitalWrite(pinRightBackward, LOW);
  analogWrite(pinRightPWM, 4.3*51);
}

void loop() {
  digitalWrite(13, LOW);
  do {} while (digitalRead(pinON) == HIGH);
  digitalWrite(13, HIGH);

  goForward(1875);

  delay(1000);

  rotateClockwise(1900);

  delay(1000);

  goForward(1875);

```

```
  rotateCounterClockwise(1900);


  // Stop and wait 1 second
  stopMotors();
  delay(1000);
}


void goForward(int tim) {
  digitalWrite(pinLeftBackward, LOW);
  digitalWrite(pinRightBackward, LOW);
  digitalWrite(pinRightForward, HIGH);
  digitalWrite(pinLeftForward, HIGH);
  delay(tim);
}
void rotateClockwise(int tim) {
  digitalWrite(pinRightForward, LOW);
  digitalWrite(pinLeftBackward, LOW);
  digitalWrite(pinLeftForward, HIGH);
  digitalWrite(pinRightBackward, HIGH);

  delay(tim); // Adjust the delay according to your robot's rotation speed
}

void rotateCounterClockwise(int tim) {
  digitalWrite(pinLeftForward, LOW);
  digitalWrite(pinRightBackward, LOW);
  digitalWrite(pinLeftBackward, HIGH);
  digitalWrite(pinRightForward, HIGH);

  delay(tim); // Adjust the delay according to your robot's rotation speed
}

void stopMotors() {
  digitalWrite(pinLeftForward, LOW);
  digitalWrite(pinRightForward, LOW);
  digitalWrite(pinLeftBackward, LOW);
  digitalWrite(pinRightBackward, LOW);
}
```

The robot did indeed move forwards by roughly the commanded amount. However, without any gradual rise and fall in motor speed, the robot was inevitably going to overshoot its target distance if not accounted for. Similarly, without accounting for the differing rotation speeds of the motors on the left and right sides, the robot was also likely to veer to one side or the other. Both of these problems will however be addressed and rectified in Experiment B.

III. Experiment B

A. 4.B.2

First starts to skip at around 45 kHz, slowly becomes more consistent until around 65 kHz, where it becomes very obvious...

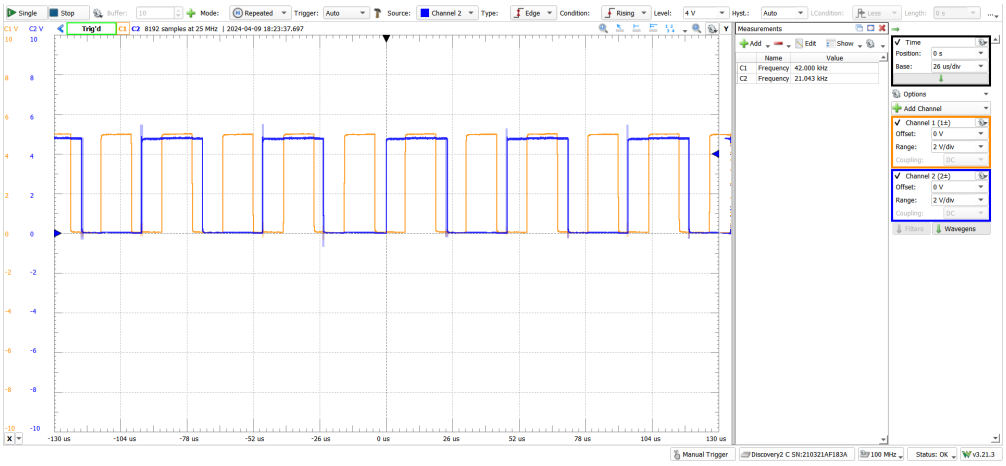

Fig. 5 First instance of bit skipping

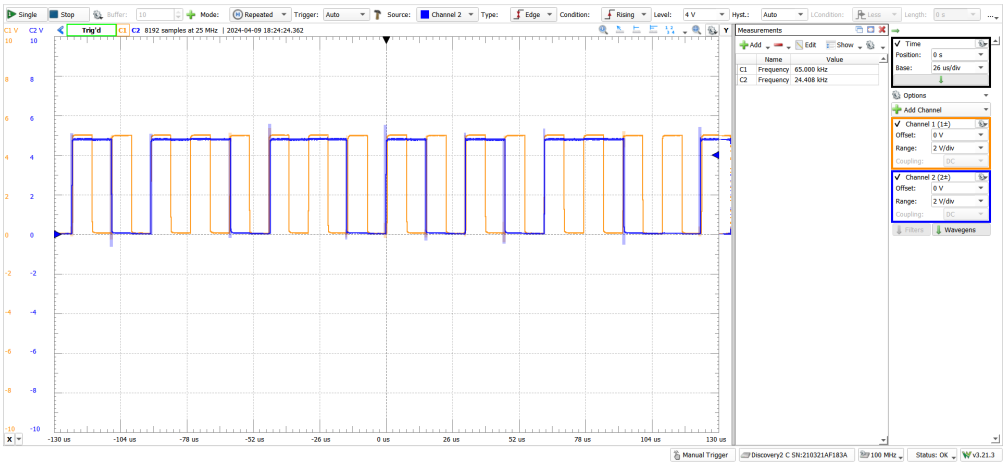

Fig. 6 Significant bit skipping

```
volatile bool var=LOW;

void setup() {
  pinMode(6, INPUT);
  attachInterrupt(digitalPinToInterrupt(6), service6, RISING);
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, var);
}

// Interrupt Service Routine
void service6() {
  var = ! var;
}
```

The overhead of the Interrupt Service Routine (ISR) for the Arduino Nano Every was determined to be about 15.4-22.2µs. This was determined by finding the first frequency that bit skipping starts to occur and the frequency that it becomes more apparent and finding the time in seconds from these frequencies. This assumes that the process "var = !var;" in the above code has negligible computation time.

B. 4.B.3

```
volatile float pL = 0;
volatile float pR = 0;
const float it = 0.00021; // [m]
const float target = 0.305; // [m]

const int pinON = 6;
const int pinRightForward = 7;
const int pinRightBackward = 8;
const int pinRightPWM = 9;
const int pinLeftPWM = 10;
const int pinLeftForward = 11;
const int pinLeftBackward = 12;

void setup() {
  pinMode(pinON, INPUT_PULLUP);

  pinMode(pinLeftForward, OUTPUT);
  pinMode(pinLeftBackward, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);

  pinMode(pinRightForward, OUTPUT);
  pinMode(pinRightBackward, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);

  digitalWrite(pinLeftForward, LOW); // Stop Forward
  digitalWrite(pinLeftBackward, LOW); // Stop Backward
  digitalWrite(pinRightForward, LOW); // Stop Forward
  digitalWrite(pinRightBackward, LOW); // Stop Backward
  analogWrite(pinLeftPWM, 200); // Vref, Duty Cycle of 200/255
  analogWrite(pinRightPWM, 200); // Vref, Duty Cycle of 200/255

  attachInterrupt(digitalPinToInterrupt(2), counterR, RISING);
  attachInterrupt(digitalPinToInterrupt(4), counterL, RISING);
  Serial.begin(9600);
}

void loop() {
  pL = 0;
  pR = 0;
  do {} while (digitalRead(pinON) == HIGH); // Wait for ON button
  digitalWrite(pinLeftForward, HIGH); // Go clockwise
  digitalWrite(pinLeftBackward, LOW); // Go clockwise
  digitalWrite(pinRightForward, HIGH); // Go clockwise
  digitalWrite(pinRightBackward, LOW); // Go clockwise
  do {} while (pL * it <= target || pR * it <= target);
  digitalWrite(pinLeftForward, LOW); // Go clockwise
  digitalWrite(pinLeftBackward, LOW); // Go clockwise
  digitalWrite(pinRightForward, LOW); // Go clockwise
  digitalWrite(pinRightBackward, LOW); // Go clockwise
  Serial.print(pL);
  Serial.print(" || ");
  Serial.println(pR);
}

void counterL() {
  pL++;

}
void counterR() {
  pR++;

}
```

In order to properly code our ISR's, we must determine the number of CPU cycles in each ISR (left and right motors) for programming and processes. This can be determined by first defining the maximum motor encoder frequency of 2250 Hz. This maximum frequency corresponds to the smallest timeframe for processes to occur. The way our code is

formatted causes the program to do nothing but run the service routine, meaning we may assume all resources (such as CPU cycles and time) are allocated to the ISR's. The minimum timeframe must then be $0.44\frac{ms}{routine}$, which is divided by two (one for each ISR) to be be $0.22\frac{ms}{routine}$ each.

Since the Arduino Nano Every has a built-in 20 MHz clock, this means that each CPU cycle is (regardless of Rising of Falling Edge logic) $50\frac{ns}{cycle}$. Dividing $22\frac{ms}{routine}$ by $50\frac{ns}{cycle}$ gives us $440000\frac{cycles}{routine}$. This assumes no other processes are occurring while waiting for an ISR.

C. 4.B.4

In order to test the correct counting of pulses, we may set the robot to move forwards a set distance and count the number of pulses (for each side) that were detected. Ideally these numbers are perfectly related and the distance that the robot moves (in meters) is defined by:

$$distance = 0.00021 N_{pulse}$$

The value of 0.00021 $\frac{m}{pulse}$ (or 0.21 $\frac{mm}{pulse}$) was derived from the "ideal" physical system parameters. Thus, we can command the robot to drive 1 meter forwards and count the number of pulses as well as the actual distance forwards it moved. Using the above equation, we can plug in the distance moved and the number of pulses and output the value of $\frac{m}{pulse}$. This value can be compared to our ideal value and changed if necessary.

Upon testing, we commanded the robot to move 1 foot (0.305 m) forwards. The robot itself moved 14 inches (1 foot and 2 inches) forwards and the left and right encoders experienced 1466 and 1453 pulses respectively.

Distance commanded [m]	Recorded distance [m]	Pulses_{left}	Pulses_{right}	$\frac{m}{\text{pulse}_{left}}$	$\frac{m}{\text{pulse}_{right}}$
0.305	0.36	1466	1453	0.000246	0.000248

Table 2 Distance Testing

From **Table 2** we can see that the error between the left and right $\frac{m}{pulse}$ values are on average 0.865%. We can then surmise that this is insignificant enough so that we may assume they are the same. For future calculations we will use the left encoder pulse value of 0.000246 $\frac{m}{pulse}$.

With this, we may determine an error of 16.94% from our ideal case. We can account for this error by changing the incremental value in the code or by accounting for error in the measurements. It should also be noted that the reasoning for this error and overshoot of the robot is due to the immediate spin-up and stop of the motors. This will later be accounted for by slowly spinning the motors up and down upon start and reaching the target value using Control Law. Similarly, in code processes, different values for $\frac{m}{pulse}$ will be used in order to account for manufacturing errors in the motors as well as environmental conditions such as the friction of the floor.

D. 4.B.5

By changing specific lines (such as which left and right pins are forwards and backwards) we may command the robot to move forwards, backwards and or spin left and right. We may also change the value of the variable "target" in order to change the distance (or number of pulses the ISR has to receive) the robot needs to move.

```
#include <math.h>

volatile float pL = 0;
volatile float pR = 0;
const float itL = 0.000248; // [m]
const float itR = 0.000238; // [m]
float target = 0; // [m]
int i = 0;
float distL = 0; // [m]
float distR = 0; // [m]

const int pinON = 6;
const int pinRightForward = 7;
```

```
const int pinRightBackward = 8;
const int pinRightPWM = 9;
const int pinLeftPWM = 10;
const int pinLeftForward = 11;
const int pinLeftBackward = 12;

void setup() {

  pinMode(pinON, INPUT_PULLUP);

  pinMode(pinLeftForward, OUTPUT);
  pinMode(pinLeftBackward, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);

  pinMode(pinRightForward, OUTPUT);
  pinMode(pinRightBackward, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);

  digitalWrite(pinLeftForward, LOW); // Stop Forward
  digitalWrite(pinLeftBackward, LOW); // Stop Backward
  digitalWrite(pinRightForward, LOW); // Stop Forward
  digitalWrite(pinRightBackward, LOW); // Stop Backward

  analogWrite(pinLeftPWM, 200);
  analogWrite(pinRightPWM, 200);

  attachInterrupt(digitalPinToInterrupt(2), counterR, RISING);
  attachInterrupt(digitalPinToInterrupt(4), counterL, RISING);
  Serial.begin(9600);

}

void loop() {

  target = 0.61; // [m]
  i = 0;
  pL = 0;
  pR = 0;
  distL = 0;
  distR = 0;

  do {} while (digitalRead(pinON) == HIGH); // Wait for ON button

  digitalWrite(pinLeftForward, HIGH); // Go clockwise
  digitalWrite(pinLeftBackward, LOW); // Go clockwise
  digitalWrite(pinRightForward, HIGH); // Go clockwise
  digitalWrite(pinRightBackward, LOW); // Go clockwise

  while (i <= 10) {
    analogWrite(pinLeftPWM, 20*i); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 20*i); // Vref, Duty Cycle of 100x/255
    i++;
    delay(10);
  }

  while (pL * itL <= target || pR * itR <= target) {
    distL = target - pL * itL;
    if (distL < 0) {
      distL = 0;
    }
    distR = target - pR * itR;
    if (distR < 0) {
      distR = 0;
    }
    analogWrite(pinLeftPWM, 255 * (1 - exp(-(distL * 100) / 3))); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 255 * (1 - exp(-(distR * 100) / 3))); // Vref, Duty Cycle of 100x/255
  }

```

```
  digitalWrite(pinLeftForward, LOW); // Go clockwise
  digitalWrite(pinLeftBackward, LOW); // Go clockwise
  digitalWrite(pinRightForward, LOW); // Go clockwise
  digitalWrite(pinRightBackward, LOW); // Go clockwise

}

void counterL() {
  pL++;
}
void counterR() {
  pR++;
}
```

Next we wish to test the accuracy of our code by commanding the robot to move forwards and backwards a commanded amount. The results of these tests are as follows:

Commanded movement	Measured movement
0.61 m	0.61 m
-0.61 m	-0.61 m
360°	361°

Table 3 Movement Testing

As seen in **Table 3**, our tests confirm that our robot moves as we would expect. For the forwards and backwards movement tests the average error is 0% and for the rotational tests the average error is 0.278%. This tells us the positional movement tests are very accurate, with the rotation tests being slightly less accurate. This is understandable however as rotational motion has more complexities and sources of error to be accounted for than linear motion.

E. 4.B.6

Lastly, we wish to test the repeatability and linearity of our code by commanding the robot to undergo a series of processes such as moving forwards 2 feet, spinning 180°, moving 2 feet forwards again and then spinning around 180° one more time. The code to execute that process is seen below:

```
#include <math.h>

volatile float pL = 0;
volatile float pR = 0;
const float itL = 0.000248; // [m]
const float itR = 0.000238; // [m]
float target = 0; // [m]
int i = 0;
float distL = 0; // [m]
float distR = 0; // [m]

const int pinON = 6;
const int pinRightForward = 7;
const int pinRightBackward = 8;
const int pinRightPWM = 9;
const int pinLeftPWM = 10;
const int pinLeftForward = 11;
const int pinLeftBackward = 12;

void setup() {

  pinMode(pinON, INPUT_PULLUP);

  pinMode(pinLeftForward, OUTPUT);
  pinMode(pinLeftBackward, OUTPUT);
  pinMode(pinLeftPWM, OUTPUT);

  pinMode(pinRightForward, OUTPUT);
```

```
  pinMode(pinRightBackward, OUTPUT);
  pinMode(pinRightPWM, OUTPUT);

  digitalWrite(pinLeftForward, LOW); // Stop Forward
  digitalWrite(pinLeftBackward, LOW); // Stop Backward
  digitalWrite(pinRightForward, LOW); // Stop Forward
  digitalWrite(pinRightBackward, LOW); // Stop Backward

  analogWrite(pinLeftPWM, 200);
  analogWrite(pinRightPWM, 200);

  attachInterrupt(digitalPinToInterrupt(2), counterR, RISING);
  attachInterrupt(digitalPinToInterrupt(4), counterL, RISING);
  Serial.begin(9600);

}

void loop() {

  target = 0.61; // [m]
  i = 0;
  pL = 0;
  pR = 0;
  distL = 0;
  distR = 0;

  do {} while (digitalRead(pinON) == HIGH); // Wait for ON button

  digitalWrite(pinLeftForward, HIGH); // Go clockwise
  digitalWrite(pinLeftBackward, LOW); // Go clockwise
  digitalWrite(pinRightForward, HIGH); // Go clockwise
  digitalWrite(pinRightBackward, LOW); // Go clockwise

  while (i <= 10) {
    analogWrite(pinLeftPWM, 20*i); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 20*i); // Vref, Duty Cycle of 100x/255
    i++;
    delay(10);
  }

  while (pL * itL <= target || pR * itR <= target) {
    distL = target - pL * itL;
    if (distL < 0) {
      distL = 0;
    }
    distR = target - pR * itR;
    if (distR < 0) {
      distR = 0;
    }
    analogWrite(pinLeftPWM, 255 * (1 - exp(-(distL * 100) / 3))); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 255 * (1 - exp(-(distR * 100) / 3))); // Vref, Duty Cycle of 100x/255
  }

  target = 0.12*3.14; // [m]
  i = 0;
  pL = 0;
  pR = 0;
  distL = 0;
  distR = 0;

  digitalWrite(pinLeftForward, LOW); // Go clockwise
  digitalWrite(pinLeftBackward, HIGH); // Go clockwise
  digitalWrite(pinRightForward, HIGH); // Go clockwise
  digitalWrite(pinRightBackward, LOW); // Go clockwise

  while (i <= 10) {
    analogWrite(pinLeftPWM, 20*i); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 20*i); // Vref, Duty Cycle of 100x/255
```

```
    i++;
    delay(10);
  }

  while (pL * itL <= target || pR * itR <= target) {
    distL = target - pL * itL;
    if (distL < 0) {
      distL = 0;
    }
    distR = target - pR * itR;
    if (distR < 0) {
      distR = 0;
    }
    analogWrite(pinLeftPWM, 255 * (1 - exp(-(distL * 100) / 3))); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 255 * (1 - exp(-(distR * 100) / 3))); // Vref, Duty Cycle of 100x/255
  }

  target = 0.61; // [m]
  i = 0;
  pL = 0;
  pR = 0;
  distL = 0;
  distR = 0;

  digitalWrite(pinLeftForward, HIGH); // Go clockwise
  digitalWrite(pinLeftBackward, LOW); // Go clockwise
  digitalWrite(pinRightForward, HIGH); // Go clockwise
  digitalWrite(pinRightBackward, LOW); // Go clockwise

  while (i <= 10) {
    analogWrite(pinLeftPWM, 20*i); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 20*i); // Vref, Duty Cycle of 100x/255
    i++;
    delay(10);
  }

  while (pL * itL <= target || pR * itR <= target) {
    distL = target - pL * itL;
    if (distL < 0) {
      distL = 0;
    }
    distR = target - pR * itR;
    if (distR < 0) {
      distR = 0;
    }
    analogWrite(pinLeftPWM, 255 * (1 - exp(-(distL * 100) / 3))); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 255 * (1 - exp(-(distR * 100) / 3))); // Vref, Duty Cycle of 100x/255
  }

  target = 0.12*3.14; // [m]
  i = 0;
  pL = 0;
  pR = 0;
  distL = 0;
  distR = 0;

  digitalWrite(pinLeftForward, HIGH); // Go clockwise
  digitalWrite(pinLeftBackward, LOW); // Go clockwise
  digitalWrite(pinRightForward, LOW); // Go clockwise
  digitalWrite(pinRightBackward, HIGH); // Go clockwise

  while (i <= 10) {
    analogWrite(pinLeftPWM, 20*i); // Vref, Duty Cycle of 100x/255
    analogWrite(pinRightPWM, 20*i); // Vref, Duty Cycle of 100x/255
    i++;
    delay(10);
  }

```

```
            while (pL * itL <= target || pR * itR <= target) {
              distL = target - pL * itL;
              if (distL < 0) {
                distL = 0;
              }
              distR = target - pR * itR;
              if (distR < 0) {
                distR = 0;
              }
              analogWrite(pinLeftPWM, 255 * (1 - exp(-(distL * 100) / 3))); // Vref, Duty Cycle of 100x/255
              analogWrite(pinRightPWM, 255 * (1 - exp(-(distR * 100) / 3))); // Vref, Duty Cycle of 100x/255
            }

            digitalWrite(pinLeftForward, LOW); // Go clockwise
            digitalWrite(pinLeftBackward, LOW); // Go clockwise
            digitalWrite(pinRightForward, LOW); // Go clockwise
            digitalWrite(pinRightBackward, LOW); // Go clockwise

          }

          void counterL() {
            pL++;
          }
          void counterR() {
            pR++;
          }
```

The robot comes back to the starting point as we would expect it to. The code used for this section results in a more accurate path than what was used in 4.A.9. This is expected as the code used in section A did not use a Control Law in order to maintain wheel travel distance was the same by damping the spin speed.

Interestingly, we notices that between consecutive resets of the motor undergoing the same test that sometimes the process would run accurately, however other times it would deviate slightly from the expected path and from other generations. This however can be attributed to motor error and assumptions that we neglected in our models such as non-ideal resistors, capacitors and IC's. Similarly, environmental conditions such as the floor texture can lead to inaccuracies, as this same model deviates more when allowed to run on carpet for example.

IV. Conclusion

With most projects, there is commonly one element of software that must be implemented, as the physical hardware needed to complete most complex processes are beyond the complexity that is expected to be manually completed. Thus, software implementation devices such as Arduino's and Raspberry Pi's are valuable for most projects. These devices are very valuable in order to implement a wide range of software.

Devices like Arduino's are particularly useful for these kind of projects as they can support a wide range of features, most importantly for this lab being hardware interrupts. Hardware interrupts allow one to interrupt any process in the software in order to complete a different process. This is used in our project in order to count encoder pulses more efficiently so that we don't miss pulses and cause errors in measuring travel distances.

Therefore, it is important to familiarize ourselves with coding in the Arduino IDE and using hardware to implement and elevate our systems complexity. This lab helped us to better understand Arduino processes, familiarizing ourselves with its inner workings and understanding processes such as clock cycling. Experiment A leaned towards helping us learn how to implement hardware onto our board effectively, while Experiment B helped us learn how to implement the code onto our Arduino board to accomplish mathematical processes for our system. These two Experiments helped us learn how to use hardware and implement it into our physical system as well as code in C++ languages.