Lab 3 - Motors

Justin Fortner & Katelyn Young

Part 1 - Driving an RC Servo

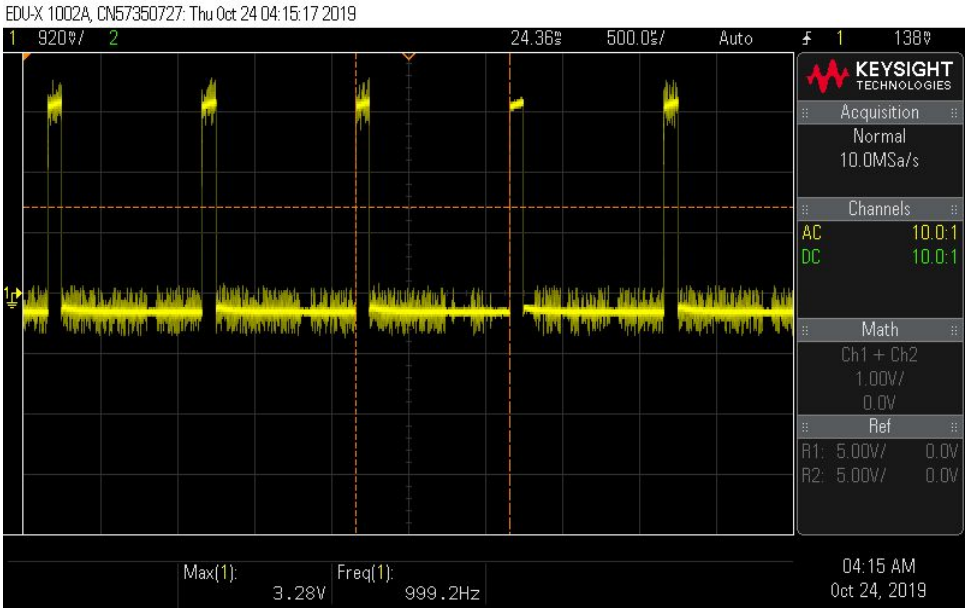
Part 1 requires the driving of a 5V powered RC servo with the use of a PWM modulated 50Hz signal. The board, RC_Servo and LED files all needed to be initialized and set to proper values as follows.

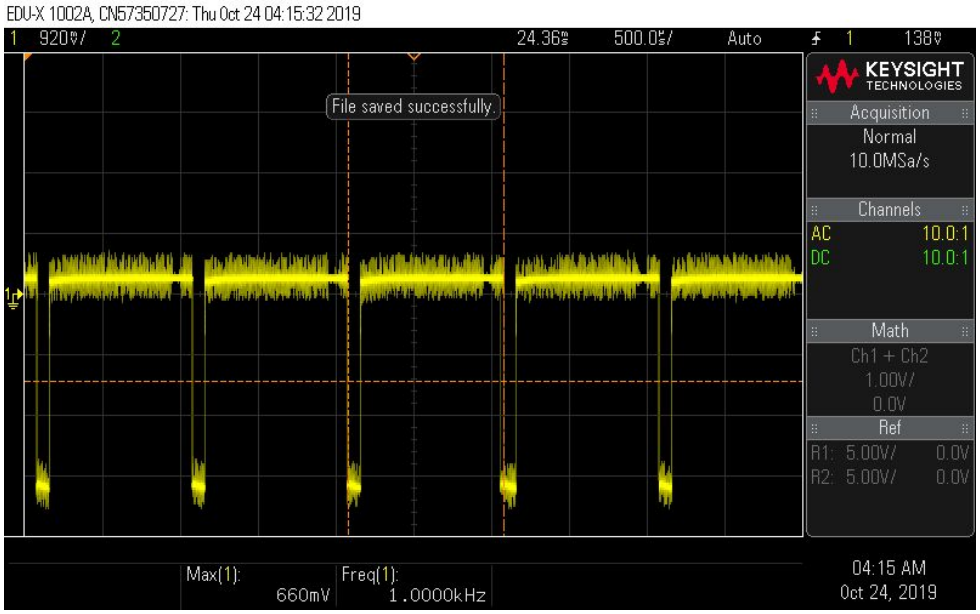
```
BOARD_Init();
    AD_Init();
    AD_AddPins(AD_PORTW3);
    LED_Init();
    LED_AddBanks(LED_BANK1 | LED_BANK2 | LED_BANK3);
    PWM_Init();
    PWM_AddPins(PWM_PORTZ06);
    PWM_SetFrequency(1000);
```

A potentiometer feeds a voltage, regulated from 0-3.3V, to analog pin W3 on the Uno32. The input read by this voltage ranged from 0-800. In order to receive the full PWM modulation scale we had to scale the potentiometer input by 1.25. Thus allowing the potentiometer to reach the full 0%-100% modulation possibility. Allowing the servo motor to reach its full +/- 60 degrees.

```
PWM_SetDutyCycle(PWM_PORTZ06, pot * 1.25);
```

This is shown by the servo control pin output traces below:





The LEDs indicate the value given by the potentiometer. As the value of the potentiometer increased more LEDs will light up. We had difficulties implementing the bit shifting method of controlling the LEDs so we had to use 12 if/else statements to control each LED individually. These statements are as follows.

```
if (pot >= 61) {
            LED_OnBank(LED_BANK1, 0b0001);
        } else if (pot < 61) {
            LED_OffBank(LED_BANK1, 0b0001);
        }
        if (pot >= 122) {
            LED_OnBank(LED_BANK1, 0b0010);
        } else if (pot < 122) {
            LED_OffBank(LED_BANK1, 0b0010);
        }
        if (pot >= 183) {
            LED_OnBank(LED_BANK1, 0b0100);
        } else if (pot < 183) {
            LED_OffBank(LED_BANK1, 0b0100);
        }
        if (pot >= 244) {
            LED_OnBank(LED_BANK1, 0b1000);
        } else if (pot < 244) {
            LED_OffBank(LED_BANK1, 0b1000);
        }


        if (pot >= 305) {
            LED_OnBank(LED_BANK2, 0b0001);
```

```
        } else if (pot < 305) {
            LED_OffBank(LED_BANK2, 0b0001);
        }
        if (pot >= 366) {
            LED_OnBank(LED_BANK2, 0b0010);
        } else if (pot < 366) {
            LED_OffBank(LED_BANK2, 0b0010);
        }
        if (pot >= 427) {
            LED_OnBank(LED_BANK2, 0b0100);
        } else if (pot < 427) {
            LED_OffBank(LED_BANK2, 0b0100);
        }
        if (pot >= 488) {
            LED_OnBank(LED_BANK2, 0b1000);
        } else if (pot < 488) {
            LED_OffBank(LED_BANK2, 0b1000);
        }


        if (pot >= 549) {
            LED_OnBank(LED_BANK3, 0b0001);
        } else if (pot < 549) {
            LED_OffBank(LED_BANK3, 0b0001);
        }
        if (pot >= 610) {
            LED_OnBank(LED_BANK3, 0b0010);
        } else if (pot < 610) {
            LED_OffBank(LED_BANK3, 0b0010);
        }
        if (pot >= 671) {
            LED_OnBank(LED_BANK3, 0b0100);
        } else if (pot < 671) {
            LED_OffBank(LED_BANK3, 0b0100);
        }
        if (pot >= 732) {
            LED_OnBank(LED_BANK3, 0b1000);
        } else if (pot < 732) {
            LED_OffBank(LED_BANK3, 0b1000);
                                            }
```

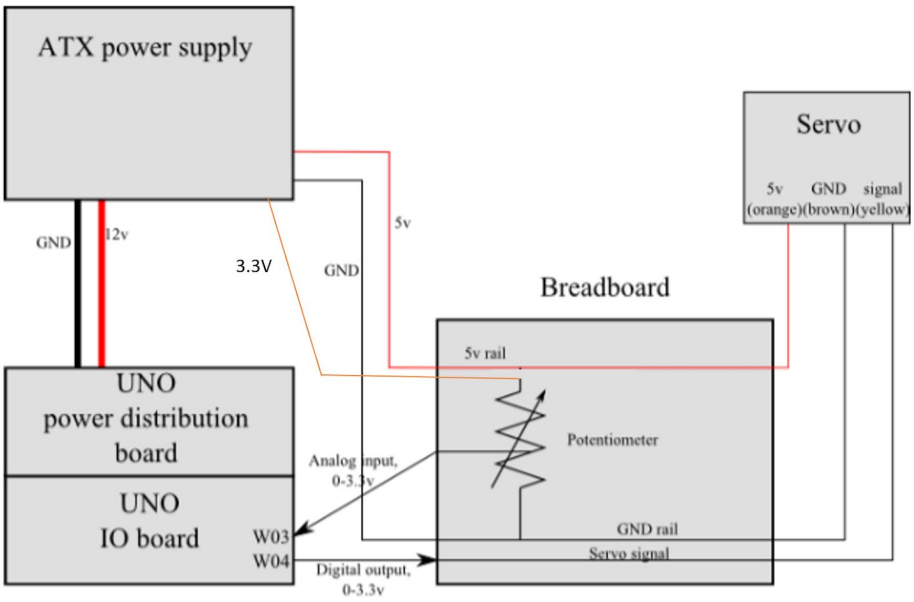
This was by far the largest portion of our code.

Overall we were able to test and verify several aspects of our RC servo motor. Through the use of a protractor we were able to verify our RC servo could reach the full

=/- 60 degrees of the minimum and maximum pulse width values. We could change our servo with as small as an 8% change in the PWM. The maximum angular velocity is 60 degrees in .17 seconds or 6.16 radians/s. The minimum angular velocity for smooth motion was calculated at a 1100 microsecond high time. 5 degrees in .15 seconds or .582 radians/s.

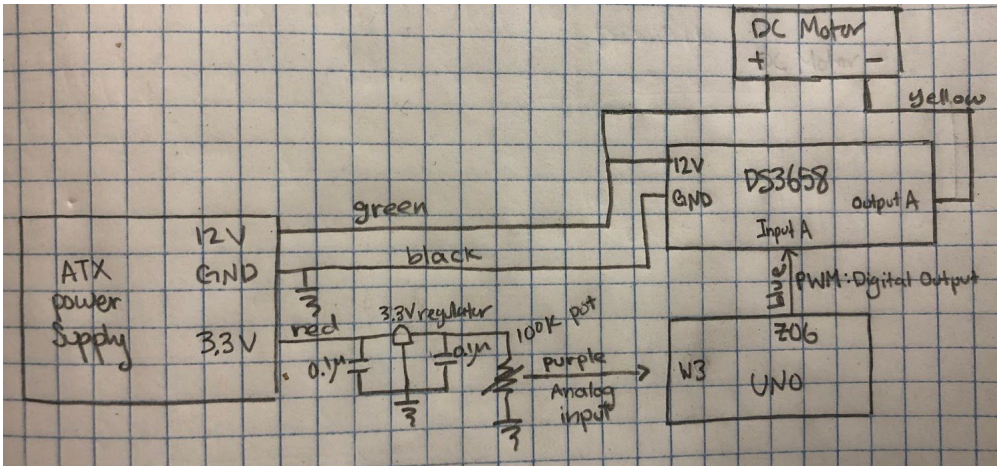
The only difficulty we encountered with this lab was when driving the LEDs as explained above.

The final block diagram is as follows:



Part 2 - Unidirectional Drive of a DC Motor

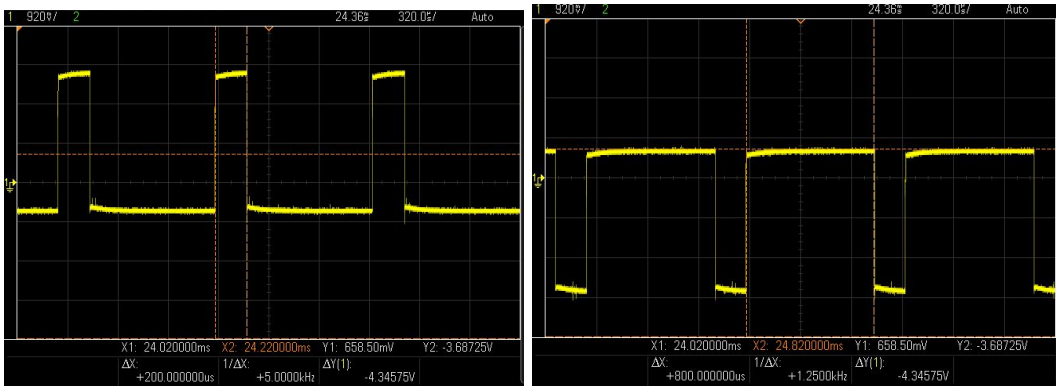
In part 2, we utilized the DS3658 high current peripheral driver to control a DC motor. A PWM signal from the Uno32 board drove the board, and the duty cycle was controlled by a 100k potentiometer (0V → 0% duty cycle and V_{max} → 100% duty cycle). By increases the duty cycle, the average DC voltage across the motor increases, consequently increasing speed of the DC motor. The LEDs on the Uno32 board also indicated the potentiometer reading level (0V → 0 LEDS and V_{max} → 12 LEDS). The following block diagram displays the constructed system.



Above: Part 2 Block Diagram

The ATX power supply provided a common ground rail, as well as a 12V and 3.3V power rail. The Uno32 pins have a 3.3V limit, so the 3.3V rail powered the regulator circuit (output voltage of about 2.85V) and the 100k potentiometer. The resulting voltage across the pot was sent to pin W3 on the Uno32. The Uno32 read this analog input, using the function "*AD_ReadADPin(AD_PORTW3)*." Port Z6, listed in pwm.h, was set up to send a 1kHz digital PWM with a duty cycle dependent on the potentiometer, using the function "*PWM_SetDutyCycle(PWM_PORTZ06, pot * 1.25)*." The "pot" variable stores the potentiometer value, read in by the previous function. The range for the duty cycle input of *PWM_SetDutyCycle()* is from 0-1000. Since our potentiometer ranges from 0-800, we multiplied the pot value by 1.25 (or 1000/800) to map our readings to the duty cycle range. The code for controlling the LEDs is the same as part 1.

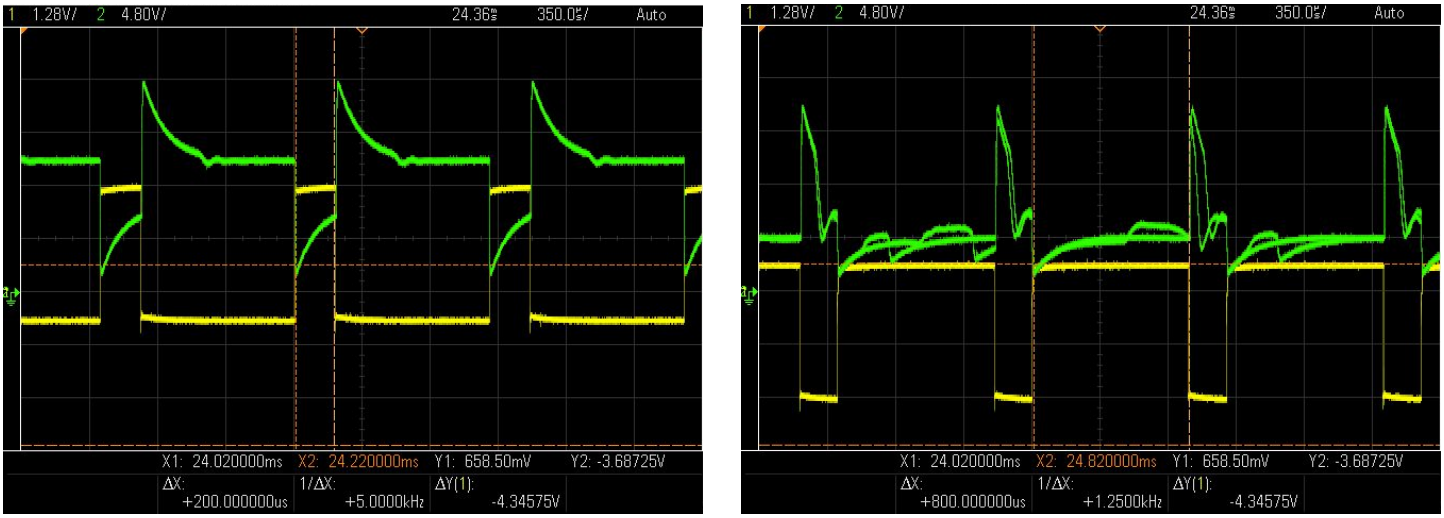
Before connecting the Uno32 to the DS3658 board, we scoped the output to verify that the PWM signal is as we expected. The following scope trace shows a duty cycle of 20% and 80%.



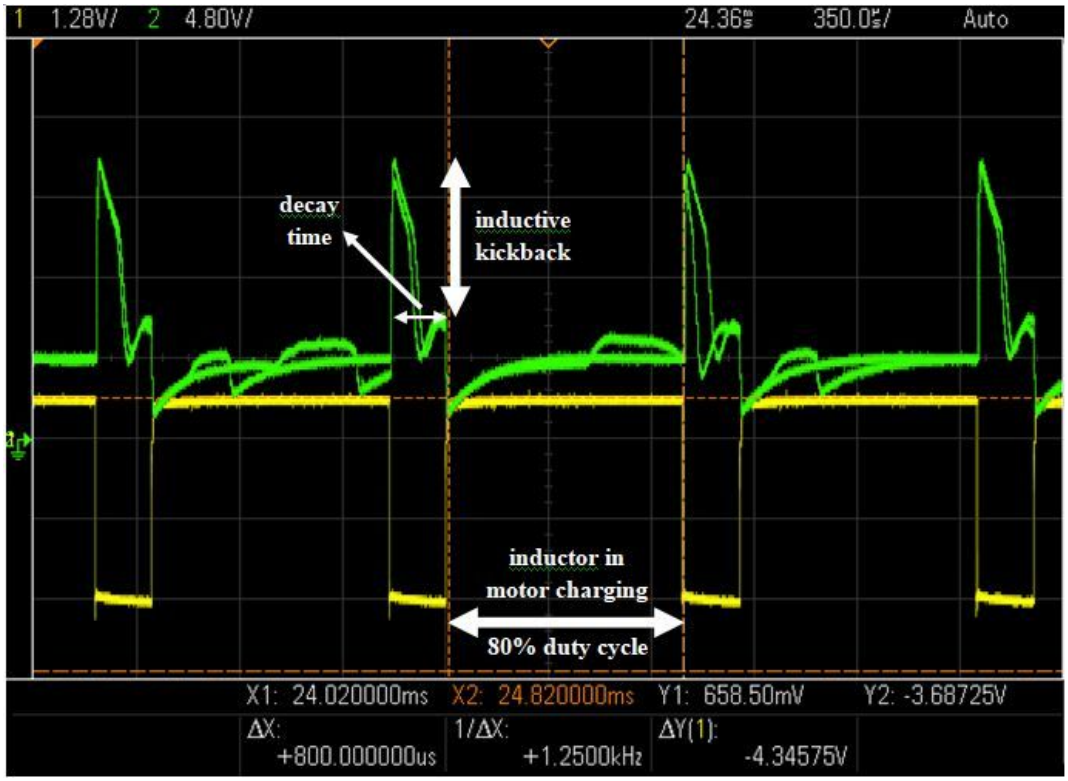
Above: 20% duty cycle PWM (LEFT) and 80% duty cycle PWM (RIGHT)

The DS3658 Driver, operating in CLAMPED mode (J3 - closed), receives the PWM signal through Input A (J2:3). The datasheet for this driver details that the power supply requirements are 5.2-30V due to its own voltage regulation. Thus, we used our 12V ATX rail to power the rail (J4:1 - Vcc and J4:2 - GND). This board only sinks current (max 600mA per output), so Output A (J1:1) was connected to the negative terminal of the DC motor, while the positive terminal was connected to the 12V source of ATX power supply. Our first trial with the full circuit setup was unsuccessful. Since the Uno32 was outputting the right PWM signal, we hypothesized that either the board or motor was malfunctioning. We were not given power specifications for the DC motor, so we originally used the 5V rail. We decided to test if the voltage was insufficient by switching to the 12V rail, and this fixed our problem. The motor would rotate when the duty cycle was 30.75% or greater. This suggests that lower duty cycles are not high long enough to generate the current needed to drive motor.

The output trace of the DS3658 board for a 20% and 80% duty cycle PWM is displayed below. Probing across the DC motor resulted in a short through the o-scope, so we probed the output of the board with reference to common ground.

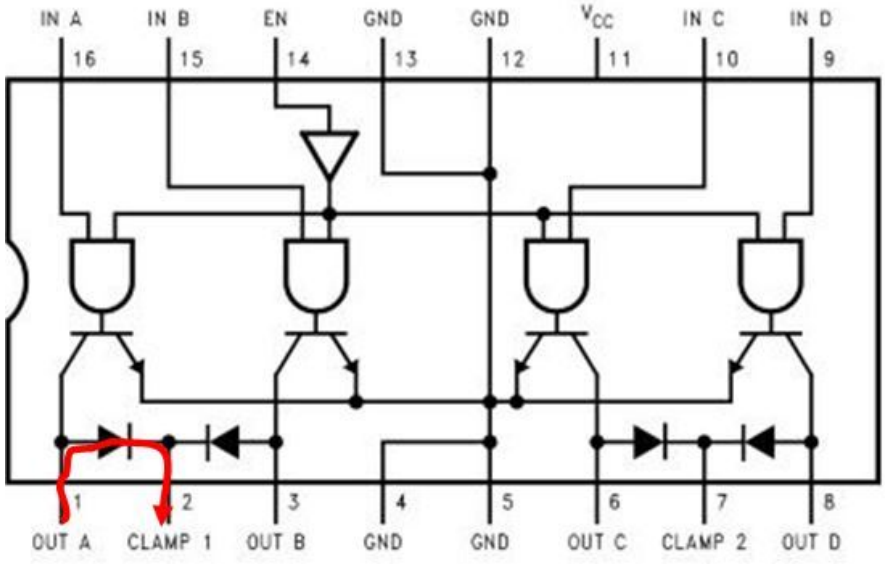


Above: Scopes of PWM signal (YELLOW) and DS3658 output (GREEN)
(LEFT: 20%. RIGHT: 80%)



Above: 80% duty cycle PWM and DS3658 output with labels

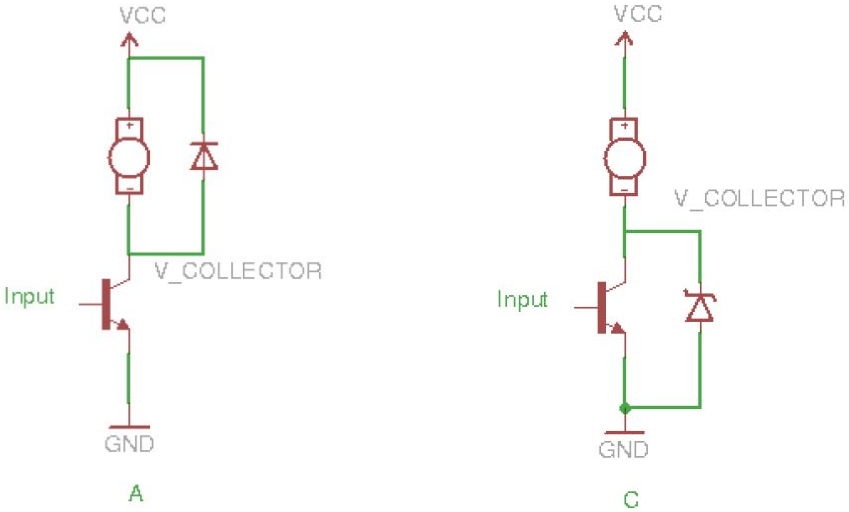
In the scope trace above, we can clearly see the different areas of the motor voltage, driven by an 80% duty cycle PWM. The decay time refers to the time needed for the inductor in the motor to completely discharge. The inductive kickback refers to large peak that occurs right after the PWM signal switches low. This voltage spike is due to the charged-up inductor leaking current back through the circuit after the transistor within the DS3658 chip switches off. This can damage the circuit components, specifically the chip. Fortunately, the chip itself has internal clamp diodes that protect the chip by providing a separate path for the leaking current to flow (shown below in the diagram). However, these clamping diodes are not enough to counteract the full effects of the inductive kickback.



Above: DS3658 chip with current path drawn through clamping diode

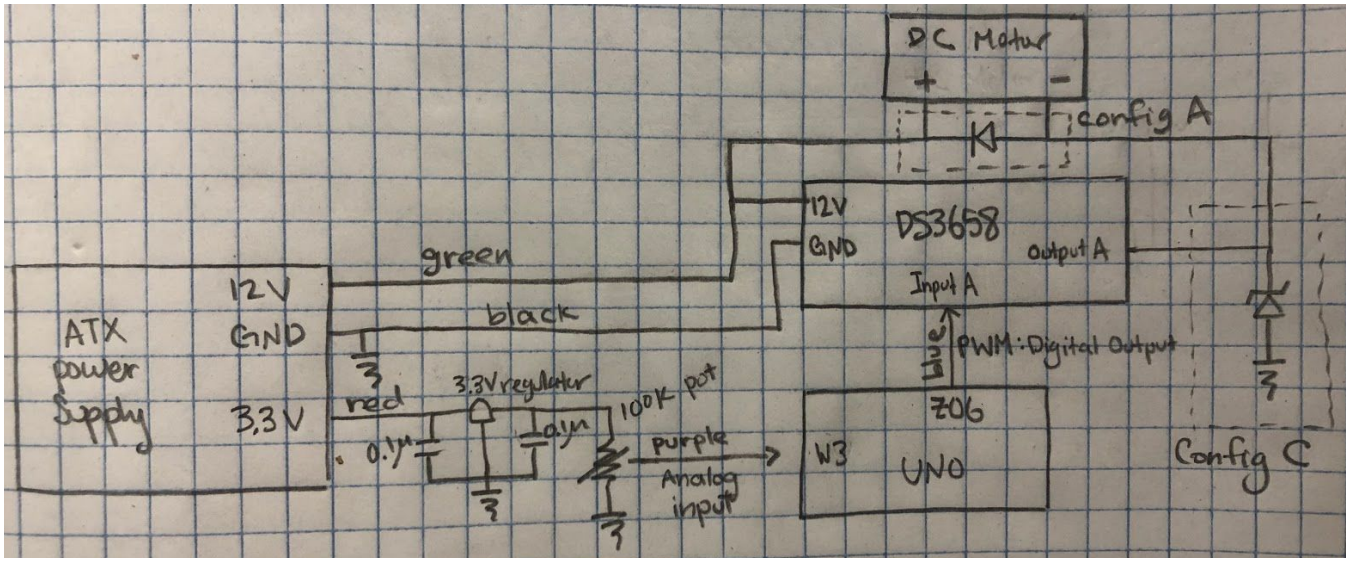
Part 3 - Snubbing the Inductive Kickback

In part 3, we took the circuit from part 2 and explored different methods of snubbing the inductive kickback discussed towards the end of the section. The software was the same, as well as most of the hardware. The only difference was incorporating the following snubbing configurations between the DS3658 chip and the DC motor.



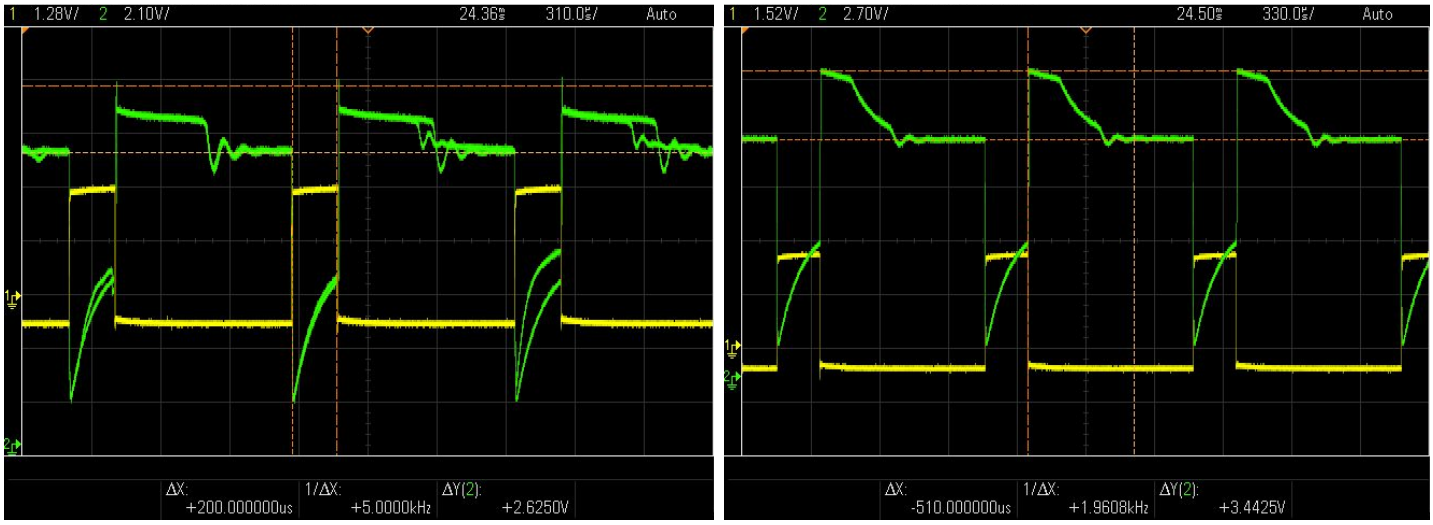
Above: Configuration A (LEFT: normal diode) and Configuration B (RIGHT: zener diode)

The resulting block diagram is shown below.



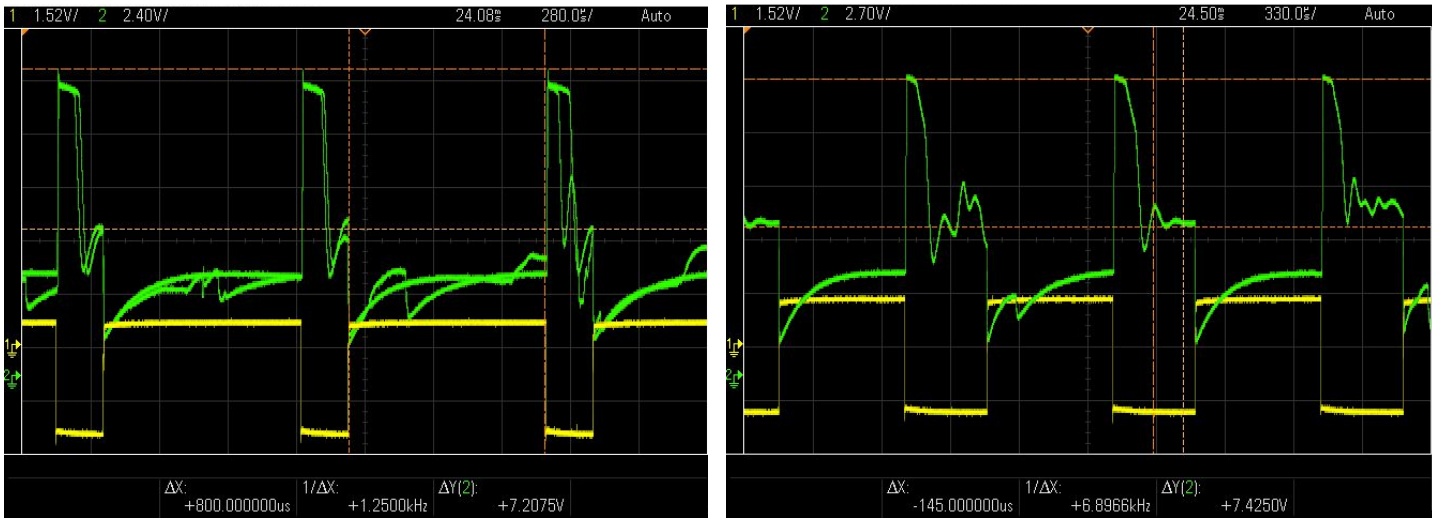
Above: Part 3 Block Diagram

To examine the difference between configuration A and C, we scoped the output of the DS3658 board and measured the kickback voltage and the decay time at low, medium, and high duty cycles.



Above: 20% Duty Cycle PWM (LEFT: config A. RIGHT: config B)

Above: 60% Duty Cycle PWM (LEFT: config A. RIGHT: config B)



Above: 80% Duty Cycle PWM (LEFT: config A. RIGHT: config B)

Configuration A Results

Duty Cycle [%]	Kickback voltage [V]	Decay time [microsec]
20	2.625	652
60	6.195	400
80	7.2075	200

Configuration C Results

Duty Cycle [%]	Kickback voltage [V]	Decay time [microsec]
20	3.4425	510
60	7.29	330
80	8.3	176

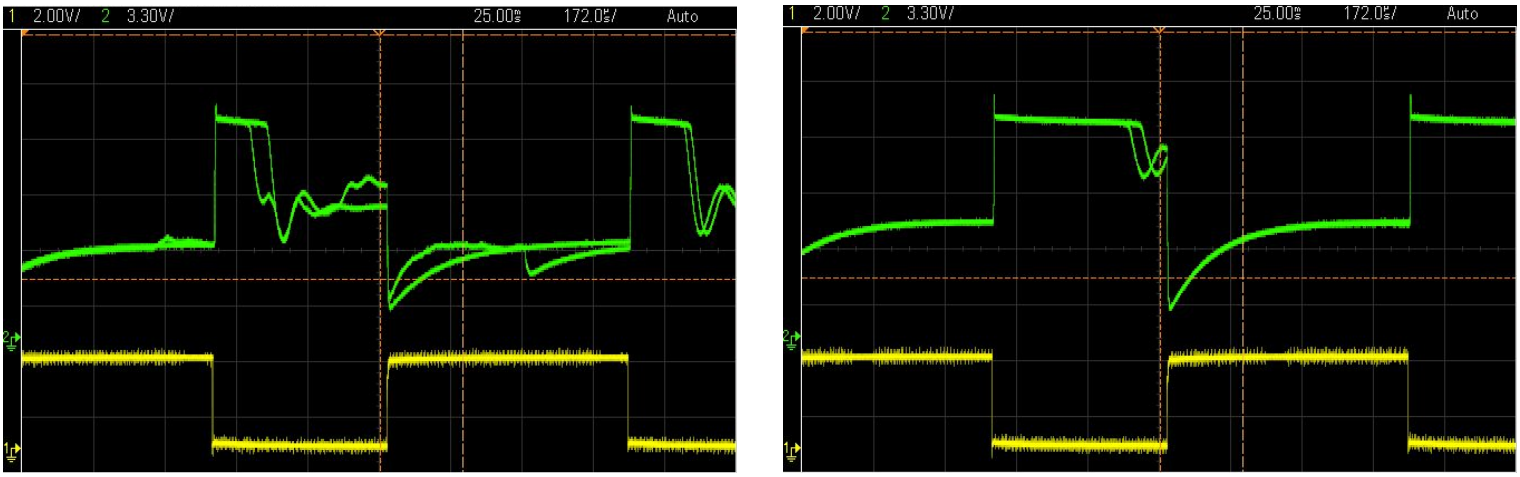
We can see that the kickback voltage is higher for Config C with the zener diode, while the decay time is higher in Config A. These results can be explained when we consider the characteristics of these diodes as well as their placement within the circuit. In Config A, the diode is placed across the motor, which forms a path through the diode back to the motor when the switch is opened and limits the voltage across motor (diode turn-on voltage). In Config C, the zener diode is placed from the output of the DS3658 driver/input of the motor to ground. When the kickback voltage reaches the zener diode breakdown voltage, the zener will allow leaking current to flow to ground. Thus, the kickback voltage of Config C will be higher than that of Config A, whose turn-on voltage is typically lower than the zener diode breakdown voltage. Because of the diode's position in Config A, the current is recirculating through the inductor, unlike in Config C, where the current is flowing to ground. Additionally, zener diodes have higher switching speeds than normal diodes. This all results in Config A having higher decay times than Config C.

A frequency and duty cycle limit analysis was also performed to find the optimal ranges for these PWM signal characteristics. We found that the motor was functional through entire range of frequencies detailed in the pwm.h file (500Hz-100kHz). There was no upper limit for the duty cycle. However, similar to part 2, the motor would not turn on until it reached a certain duty cycle. We found that as we increased the frequency, this lower limit for the duty cycle slightly increased as well, and vice versa. This happens because as the frequency increases, there is less time in between cycles for the current to rise. Thus, duty cycle need to higher to compensate.

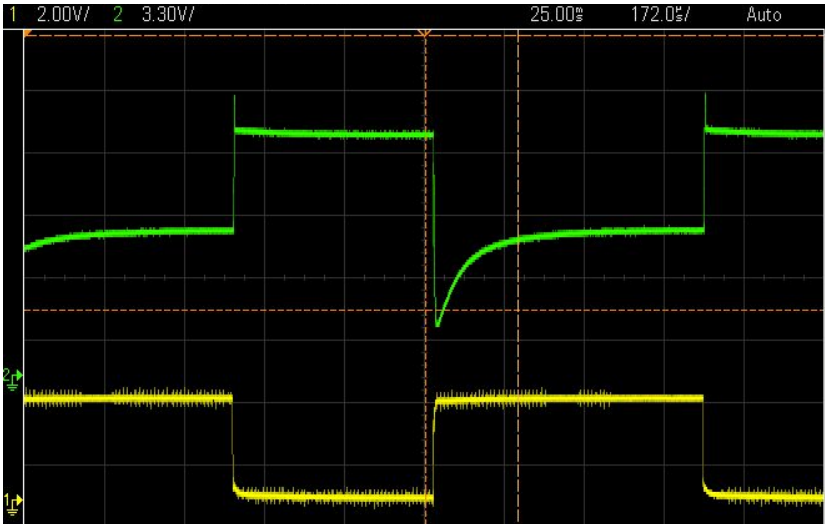
Frequency	Duty Cycle (Turn On)
500	25%
1k	27.50%
2k	28.80%
5k	30%
10k	34.20%

For the range of duty cycle in which the motor was on, the motor's speed was roughly a linear function of the duty cycle. We tested this by listening to the motor as we slowly cranked the potentiometer at constant rate. The sound of the motor would raise in pitch in parallel with the rate of change of the potentiometer, suggesting a linear function between the motor speed and potentiometer voltage. Since the potentiometer and readings and duty cycle are linearly related, this would mean the motor speed and duty cycle are also linearly related.

When a resistive load is added to the motor (i.e. fingers/pliers holding the shaft), torque/current ripple decreased significantly, as shown in the scope traces on the next page.



Above: No Resistive Torque (LEFT). Fingers Holding Motor Shaft (RIGHT)



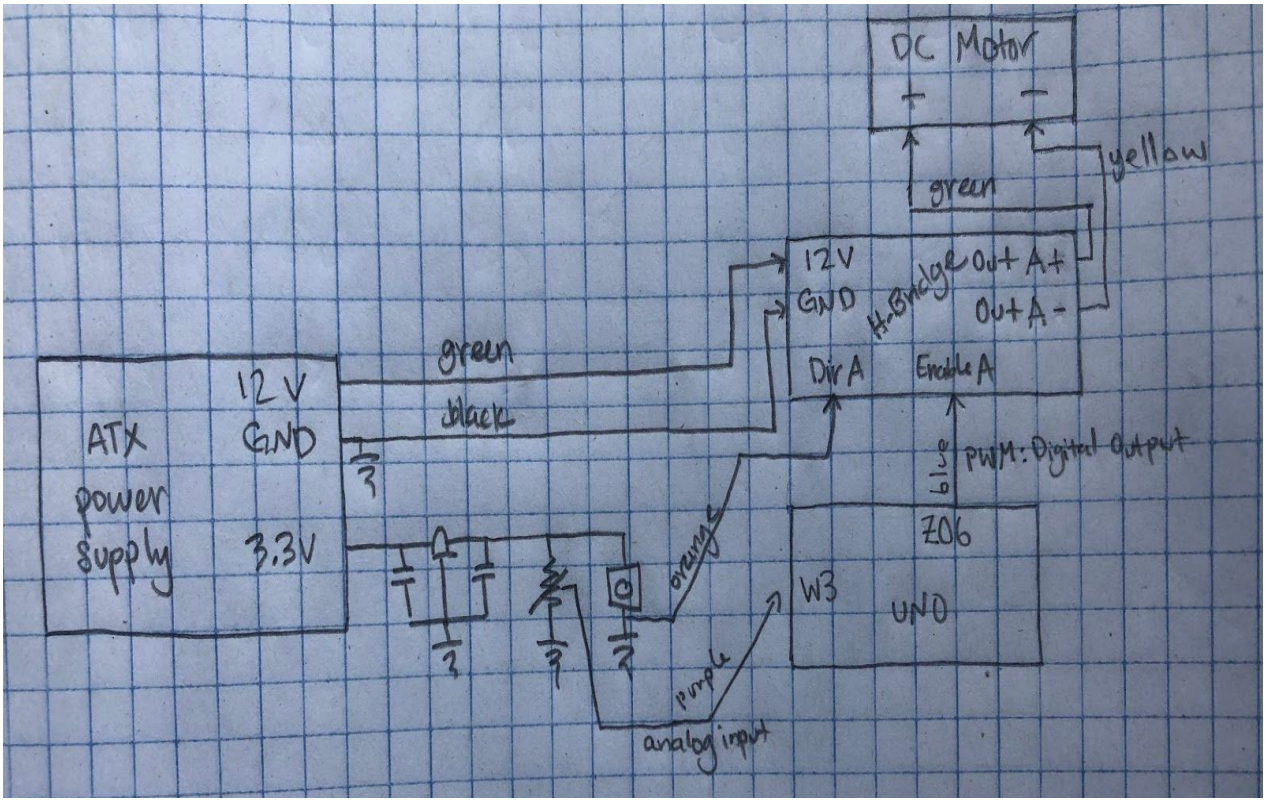
Above: Pliers Holding Motor Shaft (No Rotation)

This phenomenon can be explained by the general equation for DC motors, $V = \frac{T}{K_T}R + K_e\omega$, where: V = supply voltage, T = applied torque, K_T = torque constant
(unique to every motor), R = resistance of motor coils, K_e = voltage constant (unique to every motor), and ω = rotational speed of the motor. $K_e\omega$ is also known as the back EMF. When the motor is stalled by the resistive load, the back EMF decreases to zero, and the DC motor
equation becomes $V = \frac{T}{K_T}R$. This is essentially Ohm's law, indicating that the stalled motor
is acting like a large resistor. The absence of back EMF eliminates any ripple.

Part 4 - Bidirectional Control of a DC Motor

In part 4, we switched out the DS3658 board for an H-bridge to allow bidirectional control of the DC motor. A PWM signal from the Uno32 board drove H-bridge, and the duty cycle was controlled by a 100k potentiometer (0V → 0% duty cycle and V_{max} → 100% duty cycle). By increases the duty cycle, the average DC voltage across the motor increases, consequently increasing speed of the DC motor. The LEDs on the Uno32 board also indicated the potentiometer reading level (0V → 0 LEDS and V_{max} →

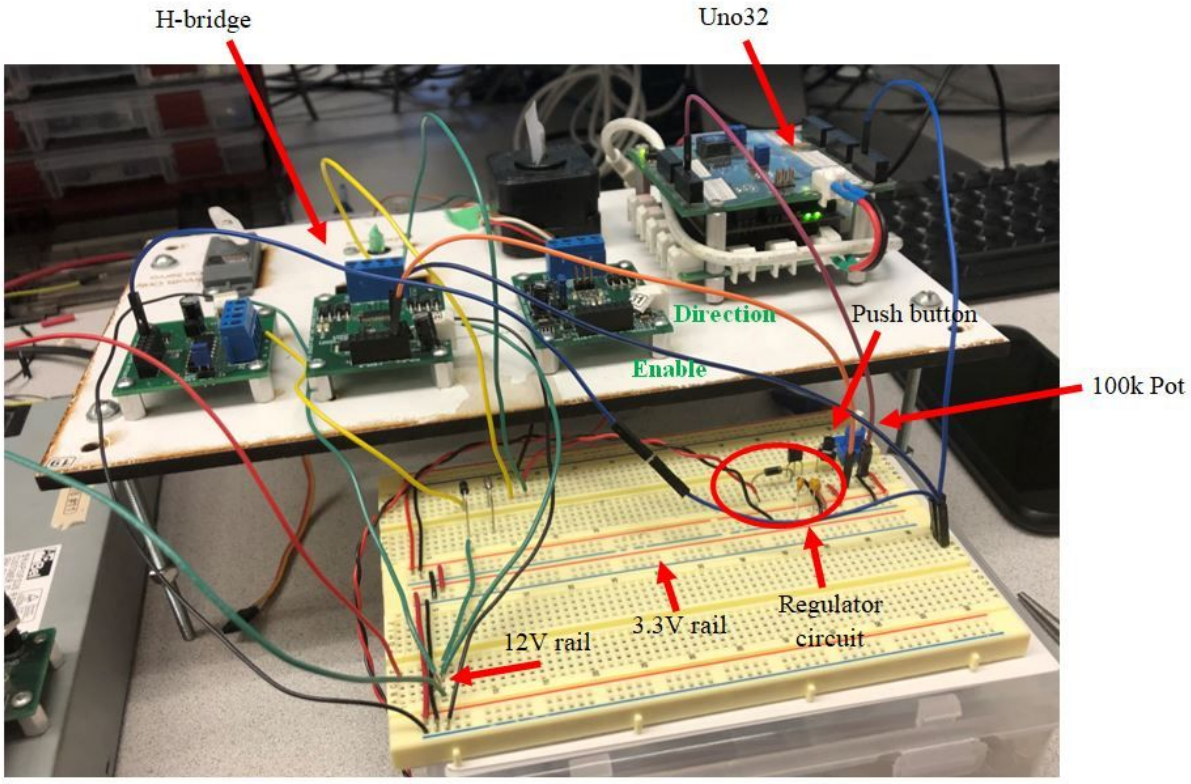
12 LEDS). A push button was added to the circuit to allow for switching directions. The following block diagram displays the constructed system.



Above: Part 4 Block Diagram

The software is the same from parts 2 and 3. A push button was added to the breadboard to control direction A (J1:11) on the H-bridge. This button provided a mostly digital output on its own so we did not need to route the signal through the Uno32 and instead connected it directly to the H-bridge. The PWM signal from the Uno32 was connected to enable A (J1:12) on the H-bridge. We then connected the outputs for A to both the positive (J2:1) and negative (J2:2) terminals of the DC motor. The H-bridge called for a power voltage range of 8V-30V, so we used our 12V ATX rail to power the

rail (J3:1 - Vcc and J3:2 - GND). We scoped the PWM output of the Uno32 before connecting to the H-bridge to verify the signal. The results were the same as in part 2, as expected. Our first attempt was unsuccessful as we included the flyback diode (Config A) from part 3, as we thought we still needed to snub the inductive kickback. However, this created a short across the motor, so the current never went through the motor. After we removed the diode, our circuit worked perfectly with the motor being driven at speeds controlled by the potentiometer and the direction changing when the button is held down and when released. The photo of our setup is included on the next page.



Above: Part 4 Circuit Setup

We performed a frequency and duty cycle analysis like in part 3. We came to the same conclusions. All the frequencies within the given range (500Hz-100kHz) were able to drive the motor without failure. The lower limit of the duty cycle was again the only value that changed.

Frequency	Duty Cycle (Turn On)
500	26.3%
1k	28%
2k	29.5%
5k	32%
10k	36.7%

We also performed the same resistive load test and also arrived at the same conclusions. The motor has the same properties so changing the direction of the flow of current through it will not alter its response to adding a resistive load. It will continue to act like a large resistor when the motor shaft is held in place.

Part 5 - Control of a Stepper Motor

Part 5 requires the control of a stepper motor using the DRV8814 H-Bridge. The stepper motor will be driven by three separate methods, Full Step, Wave Step and Half Step while finding the maximum stepper rates of each driver method. The wires are paired as white (A+) to green (A-) and red (B+) and brown (B-).

Full Step State Machine:

```
        void FullStepDrive(void) {
  switch (coilState) {
     case step_one
        printf(rnStep 1rn);
         coil drive both forward
        COIL_A_DIRECTION = 1;
        COIL_B_DIRECTION = 1;
        if (stepDir == FORWARD) {
```

```
        coilState = step_two;
    } else {
        coilState = step_four;
    }
    break;

case step_two
    printf(rnStep 2rn);
     coil drive A forward, B reverse
    COIL_A_DIRECTION = 1;
    COIL_B_DIRECTION = 0;
    if (stepDir == FORWARD) {
        coilState = step_three;
    } else {
        coilState = step_one;
    }
    break;

case step_three
    printf(rnStep 3rn);
     coil drive both reverse
    COIL_A_DIRECTION = 0;
    COIL_B_DIRECTION = 0;
    if (stepDir == FORWARD) {
        coilState = step_four;
    } else {
        coilState = step_two;
    }
    break;

case step_four
    printf(rnStep 4rn);
     coil drive A reverse, B forward
    COIL_A_DIRECTION = 0;
    COIL_B_DIRECTION = 1;
    if (stepDir == FORWARD) {
        coilState = step_one;
    } else {
        coilState = step_three;
```

```
        }
        break;
    }
}
```

Wave Step State Machine:

```
        void HalfStepDrive(void) {
    switch (coilState) {
        case step_one
            printf(rnStep 1rn);
             coil drive both forward
            COIL_A_DIRECTION = 1;
            COIL_B_DIRECTION = 0;
            COIL_A_ENABLE = 1;
            COIL_B_ENABLE = 0;
            if (stepDir == FORWARD) {
                coilState = step_two;
            } else {
                coilState = step_eight;
            }
            break;

        case step_two
            printf(rnStep 2rn);
             coil drive A forward, B reverse
            COIL_A_DIRECTION = 1;
            COIL_B_DIRECTION = 1;
            COIL_A_ENABLE = 1;
            COIL_B_ENABLE = 1;
            if (stepDir == FORWARD) {
                coilState = step_three;
            } else {
                coilState = step_one;
            }
            break;

        case step_three
            printf(rnStep 3rn);
             coil drive both reverse
```

```
    COIL_A_DIRECTION = 0;
    COIL_B_DIRECTION = 1;
    COIL_A_ENABLE = 0;
    COIL_B_ENABLE = 1;
    if (stepDir == FORWARD) {
        coilState = step_four;
    } else {
        coilState = step_two;
    }
    break;

case step_four
    printf(rnStep 4rn);
     coil drive A reverse, B forward
    COIL_A_DIRECTION = 0;
    COIL_B_DIRECTION = 1;
    COIL_A_ENABLE = 1;
    COIL_B_ENABLE = 1;
    if (stepDir == FORWARD) {
        coilState = step_five;
    } else {
        coilState = step_three;
    }
    break;

case step_five
    printf(rnStep 5rn);
     coil drive A reverse, B forward
    COIL_A_DIRECTION = 0;
    COIL_B_DIRECTION = 0;
    COIL_A_ENABLE = 1;
    COIL_B_ENABLE = 0;
    if (stepDir == FORWARD) {
        coilState = step_six;
    } else {
        coilState = step_four;
    }
    break;
```

```
case step_six
    printf(rnStep 6rn);
     coil drive A reverse, B forward
    COIL_A_DIRECTION = 0;
    COIL_B_DIRECTION = 0;
    COIL_A_ENABLE = 1;
    COIL_B_ENABLE = 1;
    if (stepDir == FORWARD) {
        coilState = step_seven;
    } else {
        coilState = step_five;
    }
    break;
case step_seven
    printf(rnStep 7rn);
     coil drive A reverse, B forward
    COIL_A_DIRECTION = 0;
    COIL_B_DIRECTION = 0;
    COIL_A_ENABLE = 0;
    COIL_B_ENABLE = 1;
    if (stepDir == FORWARD) {
        coilState = step_eight;
    } else {
        coilState = step_six;
    }
    break;
case step_eight
    printf(rnStep 8rn);
     coil drive A reverse, B forward
    COIL_A_DIRECTION = 1;
    COIL_B_DIRECTION = 0;
    COIL_A_ENABLE = 1;
    COIL_B_ENABLE = 1;
    if (stepDir == FORWARD) {
        coilState = step_one;
    } else {
        coilState = step_seven;
    }
    break;
```

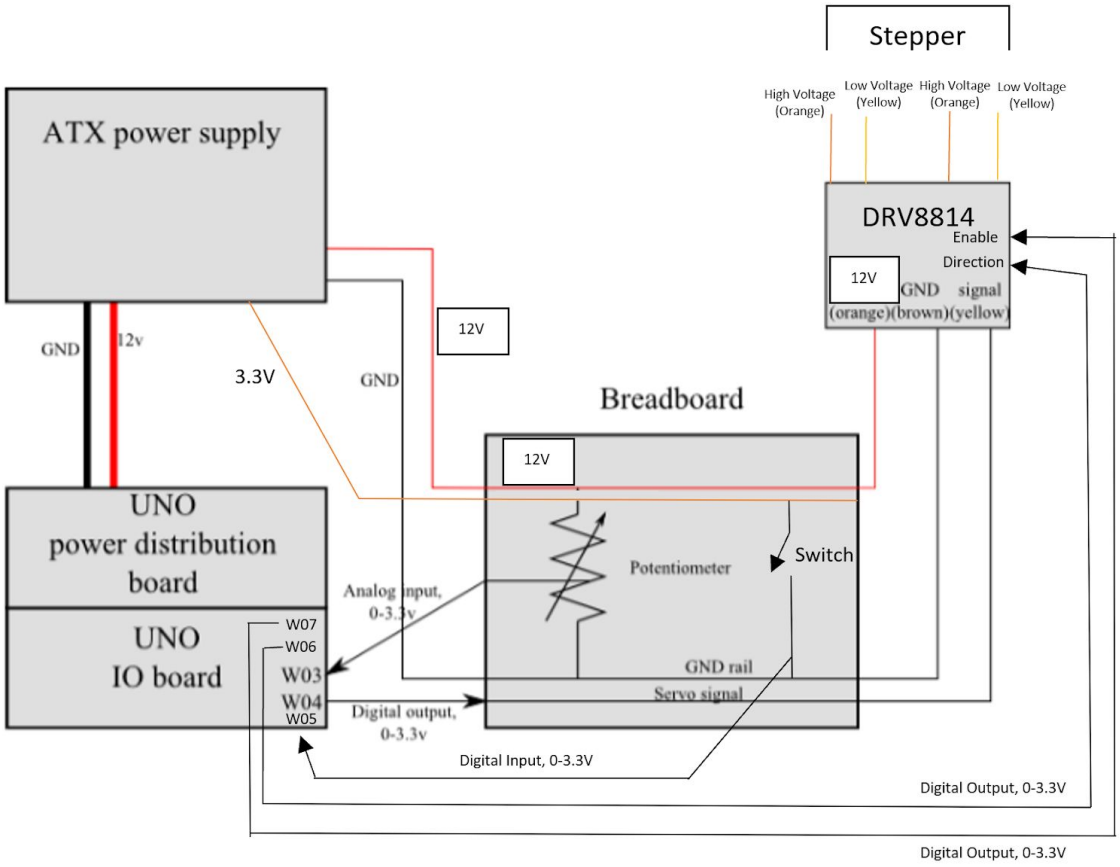
```
    }
}
```

Half Step State Machine:

```
        void WaveStepDrive(void) {
    switch (coilState) {
        case step_one
            printf(rnStep 1rn);
             coil drive both forward
            COIL_A_DIRECTION = 1;
            COIL_B_DIRECTION = 0;
            COIL_A_ENABLE = 1;
            COIL_B_ENABLE = 0;
            if (stepDir == FORWARD) {
                coilState = step_two;
            } else {
                coilState = step_four;
            }
            break;

        case step_two
            printf(rnStep 2rn);
             coil drive A forward, B reverse
            COIL_A_DIRECTION = 0;
            COIL_B_DIRECTION = 1;
            COIL_A_ENABLE = 0;
            COIL_B_ENABLE = 1;
            if (stepDir == FORWARD) {
                coilState = step_three;
            } else {
                coilState = step_one;
            }
            break;

        case step_three
            printf(rnStep 3rn);
             coil drive both reverse
            COIL_A_DIRECTION = 0;
            COIL_B_DIRECTION = 0;
```

```
            COIL_A_ENABLE = 1;
            COIL_B_ENABLE = 0;
            if (stepDir == FORWARD) {
                coilState = step_four;
            } else {
                coilState = step_two;
            }
            break;

        case step_four
            printf(rnStep 4rn);
             coil drive A reverse, B forward
            COIL_A_DIRECTION = 0;
            COIL_B_DIRECTION = 0;
            COIL_A_ENABLE = 0;
            COIL_B_ENABLE = 1;
            if (stepDir == FORWARD) {
                coilState = step_one;
            } else {
                coilState = step_three;
            }
            break;
    }
}
```

The final block diagram is as follows:



The various stepper methods have different maximum stepper values under no load as well as under load. Those are displayed in the following chart.

	Max w/o Loss	Max w/ Loss	Max w/o Loss Under Load	Max w/ Loss Under Load
Full Step	800 steps/sec	680 steps/sec	725 steps/sec	650 steps/sec
Wave Step	700 steps/sec	550 steps/sec	650 steps/sec	500 steps/sec
Half Step	1650 steps/sec	1250 steps/sec	1600 steps/sec	1225 steps/sec

Through these limits we discovered that the more steps within a diver the faster the stepper rate can be, both under load and not under load.

We initially had an issue driving our stepper motor. We realized that our initial stepper speed was too high. After lowering it to 1 we were able to run all the driver steps and increase as appropriate.

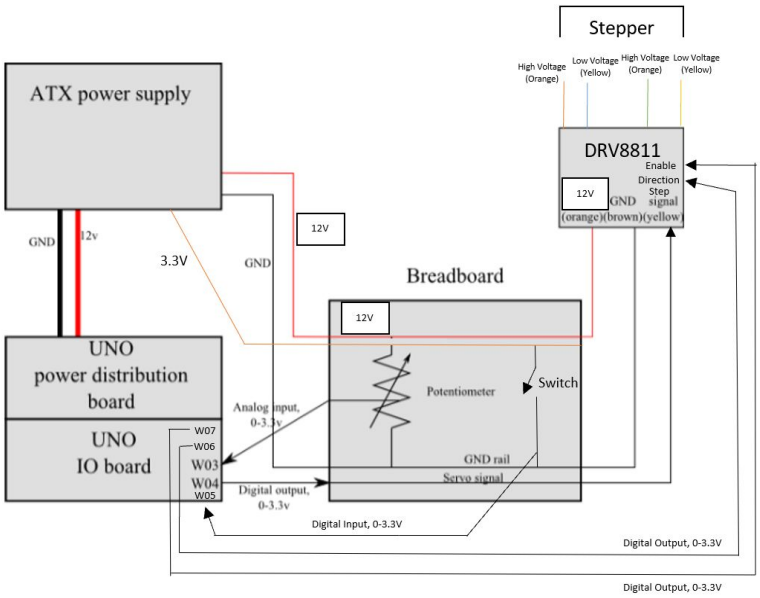
Part 6 - Stepper Motor Using Dedicated Board

Part 6 requires the control of a stepper motor using the dedicated stepper driver board DRV8811. This board allows control over the stepper motor using full, step half step, quarter step and eighth step. Because the board handles all of this internally, all that is needed is steps, direction and enable. The code is also very simple as well as there are no state machines required and only code to enable the hardware. The max stepper raste under load and no load can be seen on the following chart.

	Max w/o Loss	Max w/ Loss	Max w/o Loss Under Load	Max w/ Loss Under Load
Full Step	0 steps/sec	0 steps/sec	0 steps/sec	0 steps/sec
Half Step	100 steps/sec	10 steps/sec	80 steps/sec	7 steps/sec
Quarter Step	100 steps/sec	25 steps/sec	80 steps/sec	21 steps/sec
Eighth Step	115 steps/sec	35 steps/sec	90 steps/sec	29 steps/sec

The observation made above, the more steps within a diver the faster the stepper rate can be, both under load and not under load, held true during this part. However, we did notice that the overall step count was drastically lower than those observed during part 5. We also initially had the same issue driving our stepper motor. Luckily the solution was the same as in part 5, so we did not need to spend much time on this.

The final block diagram is as follows:



Conclusion

All in all we had a very successful lab. We were able to complete the lab and understand all of its concepts with very few difficulties. This lab was completed within a timely matter and we feel well prepared to take these skills into the final project.

CHECKOFF AND TIME TRACKING

Student Name: Justin Fortner CruzID jfortner@ucsc.edu

Time Spent out of Lab	Time Spent in Lab	Lab Part - Description
0	1	Part 0 – Preparation for Lab
0	1	Part 1 – Driving an RC Servo
0	2	Part 2 – Unidirectional Drive of a DC Motor
0	1	Part 3 – Subbing the Inductive Kickback
0	1	Part 4 – Bidirectional Control of a DC Motor
0	4	Part 5 –Control of a Stepper Motor
0	1	Part 6 –Stepper Motor Using Dedicated Board

ZP | Pre-Lab

Checkoff: TA/Tutor Initials	Lab Part - Description
CD	Part 1 – Driving an RC Servo
CD	Part 2 – Unidirectional Drive of a DC Motor
CD	Part 3 – Subbing the Inductive Kickback
CD	Part 4 – Bidirectional Control of a DC Motor
ORC	Part 5 –Control of a Stepper Motor
CD	Part 6 –Stepper Motor Using Dedicated Board

CHECKOFF AND TIME TRACKING

Student Name: Katelyn Young CruzID kalyoung @ucsc.edu

Time Spent out of Lab	Time Spent in Lab	Lab Part - Description
		Part 0 – Preparation for Lab
		Part 1 – Driving an RC Servo
		Part 2 – Unidirectional Drive of a DC Motor
		Part 3 – Subbing the Inductive Kickback
		Part 4 – Bidirectional Control of a DC Motor
		Part 5 –Control of a Stepper Motor
		Part 6 –Stepper Motor Using Dedicated Board

Checkoff: TA/Tutor Initials	Lab Part - Description
CD	Part 1 – Driving an RC Servo
CD	Part 2 – Unidirectional Drive of a DC Motor
CD	Part 3 – Subbing the Inductive Kickback
CD	Part 4 – Bidirectional Control of a DC Motor
DRC	Part 5 –Control of a Stepper Motor
CD	Part 6 –Stepper Motor Using Dedicated Board
DB	Prelab