4/15/2021

Using Microcontroller Interrupts for Measurement and Control

Georgia Institute of Technology

Electronics II with Professor First

Robert Pierrard

INTRODUCTION

The Arduino development environment provides a simplified interface to several AVR microcontrollers. However, the easier interface often comes with additional computational overhead. One can increase microcontroller speed by directly accessing the various registers onboard. Further, for interrupt services routines where time is valuable, direct access to port registers is usually preferable. The purpose of this lab is to provide exposure to such interrupt routines and coding in what we dub the "AVR" style of programming by directly accessing the registers of the ATmega328P, the microcontroller on the Arduino Uno.

This lab ultimately works towards using the Arduino to create a motor controller with an unconventional method for measuring the motor's rotation rate. But, to get there, we must first learn how to use AVR style programming and interrupt routines. This lab is therefore broken into three sections. Since we learn from each section and apply what we have learned to the next, it is most logical to go through each section (presenting its procedure, results, and discussion) individually

4/15/2021

PART I – WARMUP

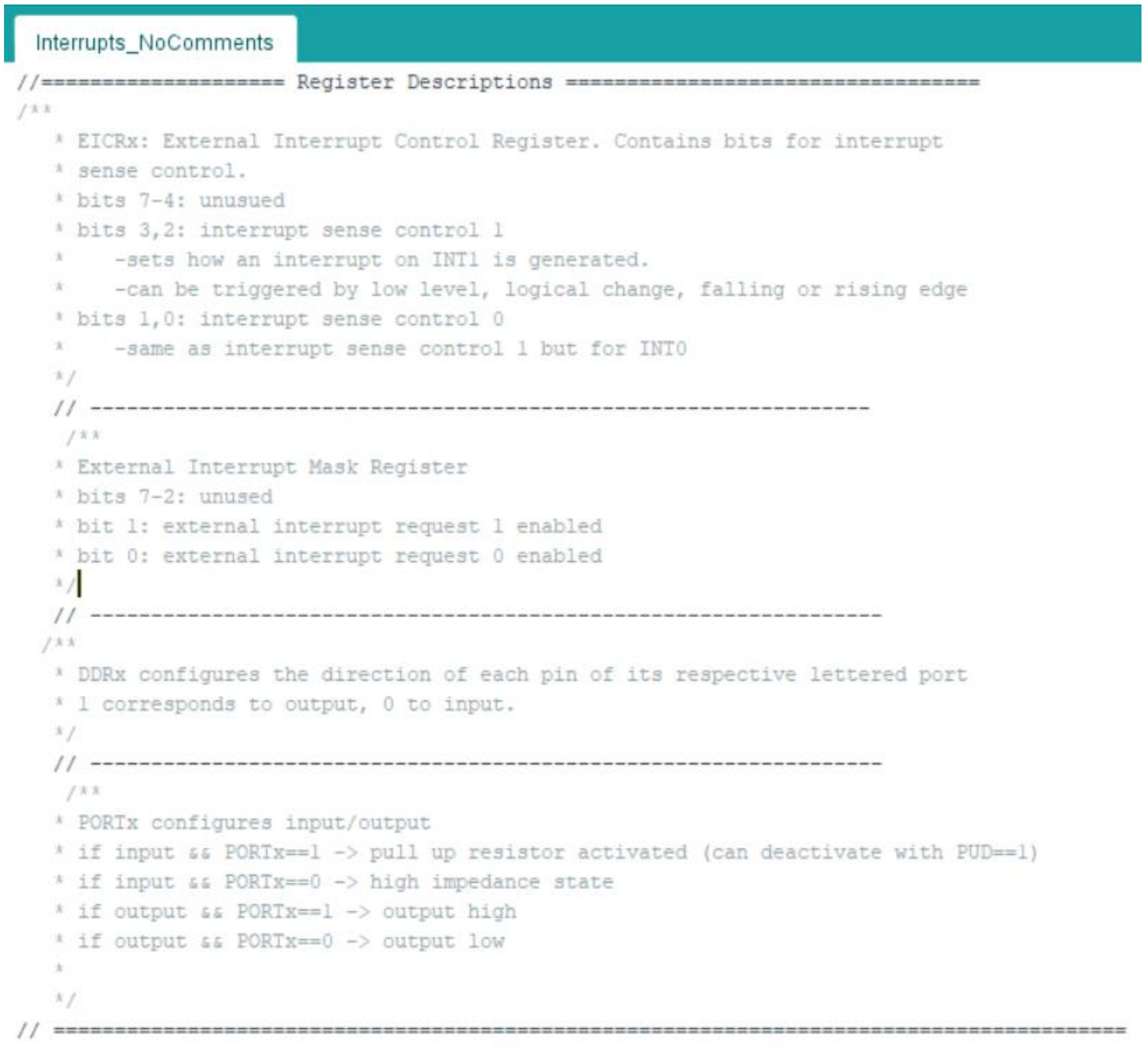
Procedure

For this section of the lab we are simply trying to build our understanding of AVR style programming. We download sample code and comment it line by line to show our understanding. In developing this understanding we must make constant reference to the ATmega328p datasheet: Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf.

Results

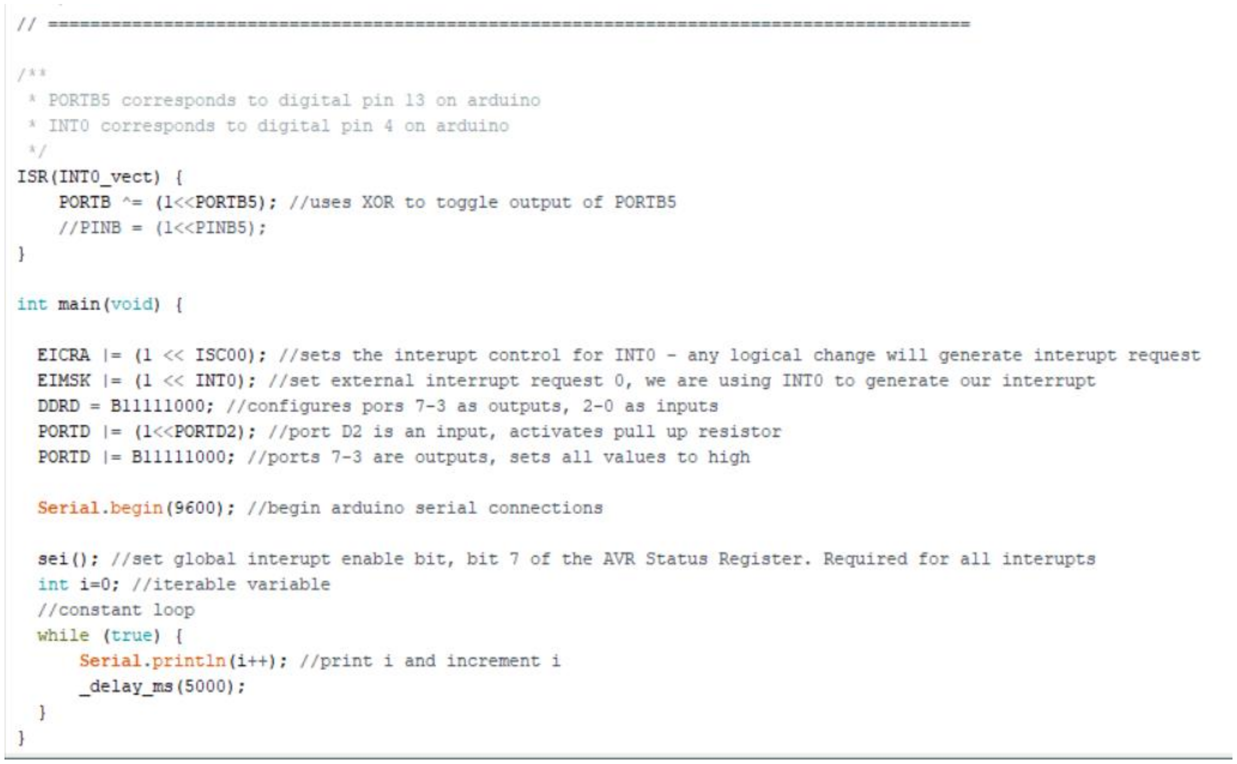
Below you will see our commented code.

Figure 1: Interrupts Code Register Descriptions



```
//===================== Register Descriptions ===================================
/**
   * EICRx: External Interrupt Control Register. Contains bits for interrupt
   * sense control.
   * bits 7-4: unusued
   * bits 3,2: interrupt sense control 1
   *    -sets how an interrupt on INT1 is generated.
   *    -can be triggered by low level, logical change, falling or rising edge
   * bits 1,0: interrupt sense control 0
   *    -same as interrupt sense control 1 but for INT0
   */
   // ----------------------------------------------------------------
    /**
   * External Interrupt Mask Register
   * bits 7-2: unused
   * bit 1: external interrupt request 1 enabled
   * bit 0: external interrupt request 0 enabled
   */
   // -----------------------------------------------------------------
  /**
   * DDRx configures the direction of each pin of its respective lettered port
   * 1 corresponds to output, 0 to input.
   */
   // -----------------------------------------------------------------
    /**
   * PORTx configures input/output
   * if input && PORTx==1 -> pull up resistor activated (can deactivate with PUD==1)
   * if input && PORTx==0 -> high impedance state
   * if output && PORTx==1 -> output high
   * if output && PORTx==0 -> output low
   *
   */
 // ========================================================================================
```

Figure 2: Interrupts Code Comments



```
// ==========================================================================================

/**
 * PORTB5 corresponds to digital pin 13 on arduino
 * INT0 corresponds to digital pin 4 on arduino
 */
ISR(INT0_vect) {
    PORTB ^= (1<<PORTB5); //uses XOR to toggle output of PORTB5
    //PINB = (1<<PINB5);
}

int main(void) {

  EICRA |= (1 << ISC00); //sets the interupt control for INT0 - any logical change will generate interupt request
  EIMSK |= (1 << INT0); //set external interrupt request 0, we are using INT0 to generate our interrupt
  DDRD = B11111000; //configures pors 7-3 as outputs, 2-0 as inputs
  PORTD |= (1<<PORTD2); //port D2 is an input, activates pull up resistor
  PORTD |= B11111000; //ports 7-3 are outputs, sets all values to high

  Serial.begin(9600); //begin arduino serial connections

  sei(); //set global interupt enable bit, bit 7 of the AVR Status Register. Required for all interupts
  int i=0; //iterable variable
  //constant loop
  while (true) {
      Serial.println(i++); //print i and increment i
      _delay_ms(5000);
  }
}
```

Discussion

In Figure 1 you can see that we included a multi-line comment that serves as a reference to each of the relevant registers for our code. This information can be found in the ATmega328p datasheet. But, having it readily accessible makes it much easier to break down the subsequent code line by line. The relevant registers include the EICRx, EIMSK, DDRx, and PORTx registers. Each register's specific use is explained in Figure 1.

In Figure 2 we have commented the code line by line. Now that we understand the code, it seems quite simple. However, when first exposed the code makes no sense at all and so this exercise is very useful before trying to program in the AVR style yourself. In summary, the code activates INT0 to trigger an interrupt request upon any logical change. It configures the D ports as inputs and outputs. Finally, it toggles PORTB5 upon activation of the interrupt request.

PART II – Toggle an LED from an External Event

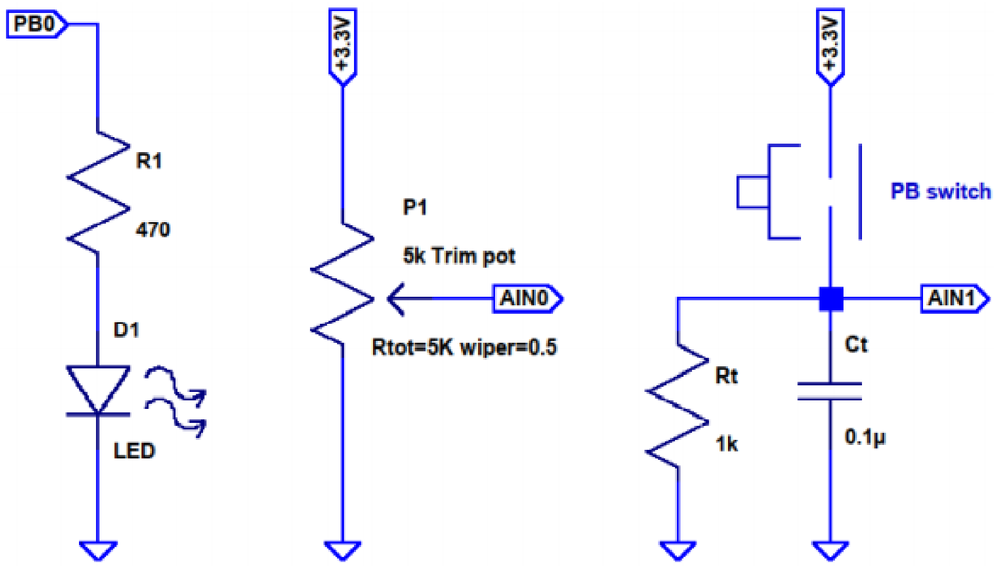
In this part of the lab we will configure the Arduino to trigger an interrupt based on the analog comparator (as opposed to INT0) to toggle an LED on an off. Subsequently, we will learn to use the output of the analog comparator to trigger a "timer capture," which will later be used for measuring rotation periods.

Procedure

2.1 Contact Detection

- First, set up the following circuit:

Figure 3: Circuit for Testing the ANALOG_COMP Interrupt



- Tune the potentiometer to output 1.6 to 1.7V. This will be used as the analog comparator reference voltage. It is tuned midway between the 3.3V supply voltage and ground.
- Use the MyDAQ to confirm that the signal at AIN1 is indeed 0V by default and 3.3V when the button is pressed.
- Now, write a program that responds to the interrupt from the analog comparator when the input signal crosses the reference signal. (We try configuring with the rising edge, the falling edge, and toggle trigger response)
  - Figure 4 in the results section contains this code.

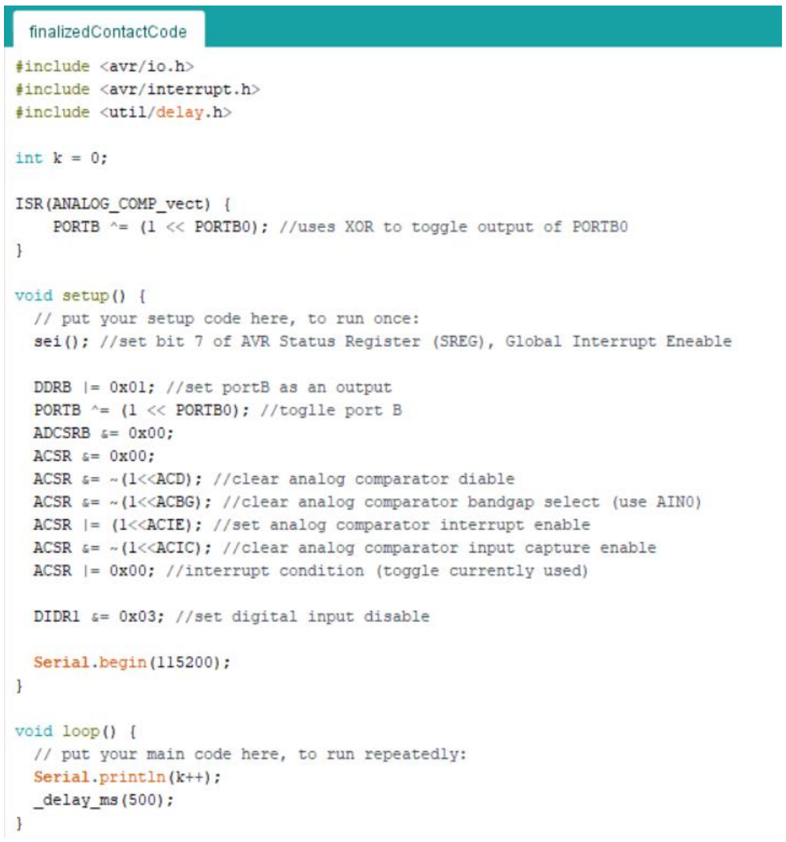
2.2 Interval Timing

- We now familiarize ourselves with Timer1 by reading the appropriate section of the ATmega datasheet.
- Using the same circuit, we will write a new program.
- We copy the configuration of the analog comparator from our previous code (section 2.1).
- Modify the code to include the following changes.
    - Disable the interrupt in the analog comparator configuration and enable the input capture bit.
    - Configure Timer1 with a prescale of 1024, enable the input capture bit, and trigger on the rising edge.
- Next, write a program that outputs on the serial port the time difference between subsequent presses of the push button.
    - By enabling overflow interrupt enable, we can extend the range of our timer from 16 to 31 bits.
    - Figure 5 in the results section contains our code. We will explain the code in our discussion.

Results

2.1 Contact Detection

Figure 4: Contact Detection Code

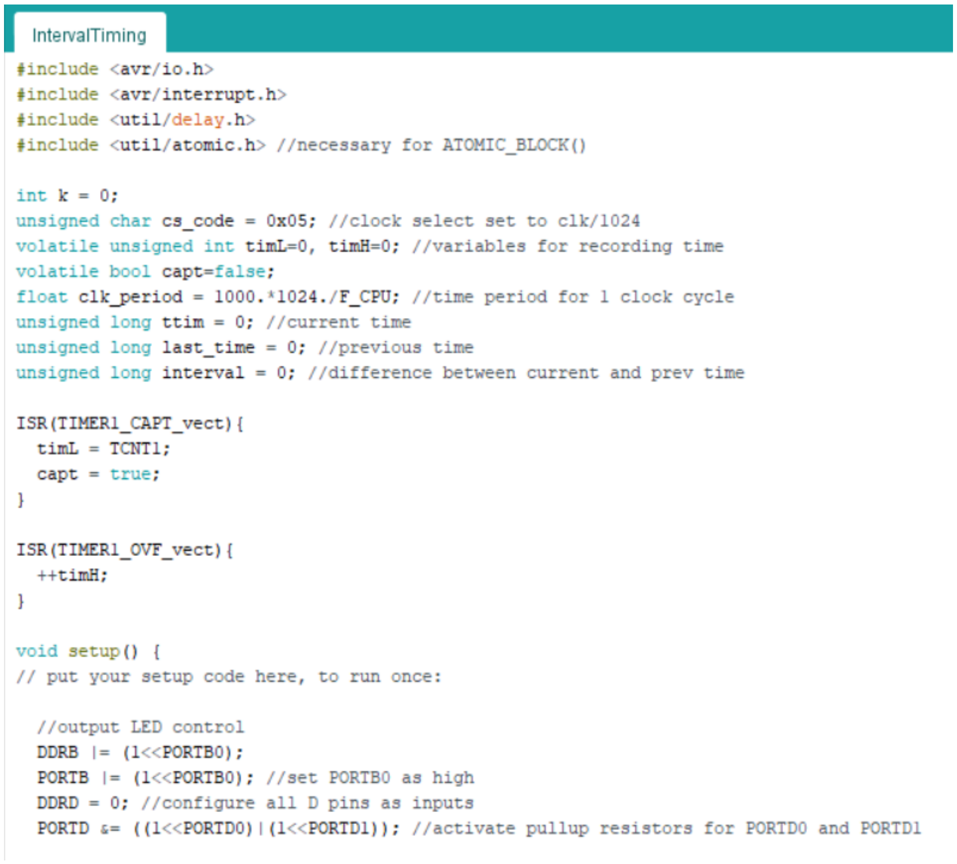


finalizedContactCode

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

int k = 0;

ISR(ANALOG_COMP_vect) {
    PORTB ^= (1 << PORTB0); //uses XOR to toggle output of PORTB0
}

void setup() {
  // put your setup code here, to run once:
  sei(); //set bit 7 of AVR Status Register (SREG), Global Interrupt Eneable

  DDRB |= 0x01; //set portB as an output
  PORTB ^= (1 << PORTB0); //toglle port B
  ADCSRB &= 0x00;
  ACSR &= 0x00;
  ACSR &= ~(1<<ACD); //clear analog comparator diable
  ACSR &= ~(1<<ACBG); //clear analog comparator bandgap select (use AIN0)
  ACSR |= (1<<ACIE); //set analog comparator interrupt enable
  ACSR &= ~(1<<ACIC); //clear analog comparator input capture enable
  ACSR |= 0x00; //interrupt condition (toggle currently used)

  DIDR1 &= 0x03; //set digital input disable

  Serial.begin(115200);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println(k++);
  _delay_ms(500);
}
```

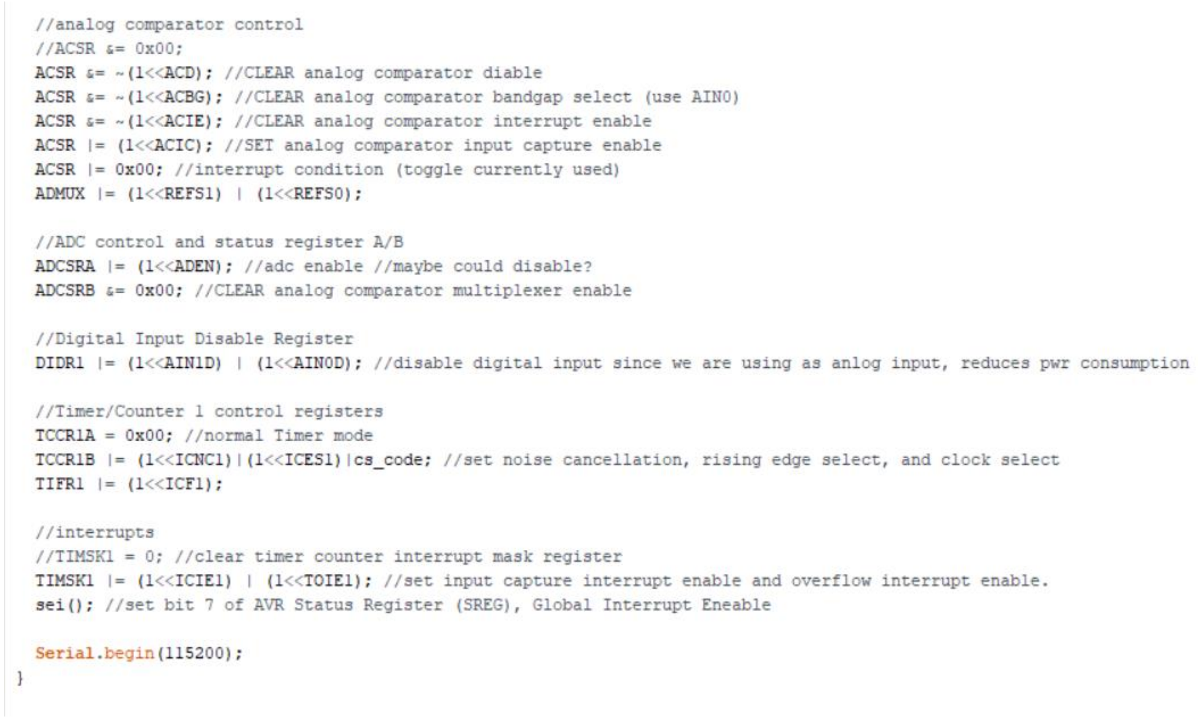
2.2 Interval Timing

Figure 5: Interval Timing Code

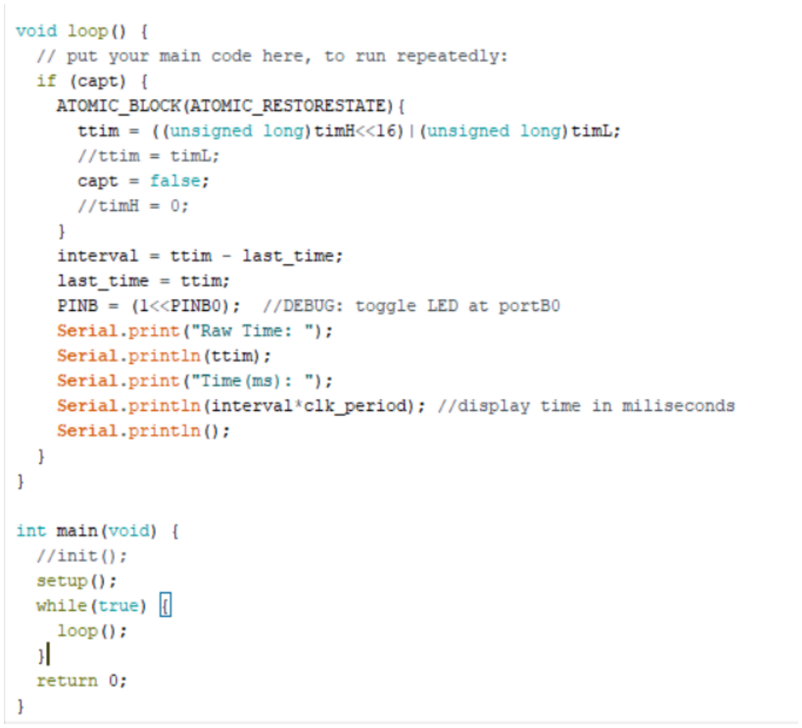


IntervalTiming

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <util/atomic.h> //necessary for ATOMIC_BLOCK()

int k = 0;
unsigned char cs_code = 0x05; //clock select set to clk/1024
volatile unsigned int timL=0, timH=0; //variables for recording time
volatile bool capt=false;
float clk_period = 1000.*1024./F_CPU; //time period for 1 clock cycle
unsigned long ttim = 0; //current time
unsigned long last_time = 0; //previous time
unsigned long interval = 0; //difference between current and prev time

ISR(TIMER1_CAPT_vect){
  timL = TCNT1;
  capt = true;
}

ISR(TIMER1_OVF_vect){
  ++timH;
}

void setup() {
// put your setup code here, to run once:

  //output LED control
  DDRB |= (1<<PORTB0);
  PORTB |= (1<<PORTB0); //set PORTB0 as high
  DDRD = 0; //configure all D pins as inputs
  PORTD &= ((1<<PORTD0)|(1<<PORTD1)); //activate pullup resistors for PORTD0 and PORTD1
```



```
  //analog comparator control
  //ACSR &= 0x00;
  ACSR &= ~(1<<ACD); //CLEAR analog comparator diable
  ACSR &= ~(1<<ACBG); //CLEAR analog comparator bandgap select (use AIN0)
  ACSR &= ~(1<<ACIE); //CLEAR analog comparator interrupt enable
  ACSR |= (1<<ACIC); //SET analog comparator input capture enable
  ACSR |= 0x00; //interrupt condition (toggle currently used)
  ADMUX |= (1<<REFS1) | (1<<REFS0);

  //ADC control and status register A/B
  ADCSRA |= (1<<ADEN); //adc enable //maybe could disable?
  ADCSRB &= 0x00; //CLEAR analog comparator multiplexer enable

  //Digital Input Disable Register
  DIDR1 |= (1<<AIN1D) | (1<<AIN0D); //disable digital input since we are using as anlog input, reduces pwr consumption

  //Timer/Counter 1 control registers
  TCCR1A = 0x00; //normal Timer mode
  TCCR1B |= (1<<ICNC1)|(1<<ICES1)|cs_code; //set noise cancellation, rising edge select, and clock select
  TIFR1 |= (1<<ICF1);

  //interrupts
  //TIMSK1 = 0; //clear timer counter interrupt mask register
  TIMSK1 |= (1<<ICIE1) | (1<<TOIE1); //set input capture interrupt enable and overflow interrupt enable.
  sei(); //set bit 7 of AVR Status Register (SREG), Global Interrupt Eneable

  Serial.begin(115200);
}
```



```
void loop() {
  // put your main code here, to run repeatedly:
  if (capt) {
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
      ttim = ((unsigned long)timH<<16)|(unsigned long)timL;
      //ttim = timL;
      capt = false;
      //timH = 0;
    }
    interval = ttim - last_time;
    last_time = ttim;
    PINB = (1<<PINB0);  //DEBUG: toggle LED at portB0
    Serial.print("Raw Time: ");
    Serial.println(ttim);
    Serial.print("Time(ms): ");
    Serial.println(interval*clk_period); //display time in miliseconds
    Serial.println();
  }
}

int main(void) {
  //init();
  setup();
  while(true) {
    loop();
  }
  return 0;
}
```

Discussion

2.1 Contact Detection

The overall function of this code is simple, toggle an LED on and off. We achieve our goal through an interrupt service routine. This interrupt service routine is triggered by the analog comparator which we configure in our setup function. Our code is commented heavily and so we will not explain line by line. Functionally, however, we set the analog comparator interrupt enable, we set the interrupt condition to toggle, and we set PORTB0 as an output. Then whenever the ANALOG_COMP_vect is set, triggering an interrupt, we toggle PORTB0.

Our main function simply prints an ever-increasing value to the console. This was done to ensure that our code was being uploaded properly.

2.2 Interval Timing

This code is considerably more complex than the previous contact detection code. We will trace through each function called in main(), keeping in mind again that the code is heavily commented already.

We already encounter one key distinction from before. Rather than using the Arduino style of a setup() and loop() function, we override the Arduino main() function (which typically is run in the background and calls these other functions). This is because the Arduino main() function also makes a call to init(), which we see a commented out in the first line of our main(). This init() function make changes to the timer registers and would make our code malfunction. For now, we simply omit it but we will revisit this problem in section 3. In our setup() function we set controls for the output LED port, the analog comparator, the analog to digital converter, the timer/counters, and the interrupts. Our next function in main is a simple loop. In our loop we have a conditional checking if capt is true or false. This variable is set true by the TIMER1_CAPT_vect interrupt service routine. Also in this routine, we record the value of TCNT1, the timer register, into a global variable. We have an additional interrupt service routine for the TIMER1_OVF_vect. When activated, we simply increment another global variable by 1. In this way we have effectively used a 16 bit counter in hardware to create a 32 bit counter in software. By combining out timer variables using bit shifting we can create one 32 bit integer representing the current "time" of our system. We then record this time and subtract it from the previous time to calculate an elapsed time (in factors of 1024 clock cycles). Knowing the period for one clock cycle, we can simply convert this time to millisecond and display to the Serial Monitor.

Many challenges were encountered in writing this code. However, the coding process is difficult (and maybe pointless) to try to explain. In the end we produce the above code following the above outline. The code functions as expected. One thing to note, when pressing the button it was necessary to firmly press down and release, otherwise bouncing effects led to multiple triggers, even with noise cancellation envoked.

PART III – Measure the Period of Rate

In this section we will measure the rotational period of a motor by connecting a light sensor to our microcontroller rather than a push button. We also control the motor itself with the microcontroller and can thus invoke feedback.

Procedure

3.1 and 3.2 – Motor Driver and Rotation Sensor

- We construct the following 2 circuits. For detection, the light sensor on the rotation sensing circuit must be pointed towards the shaft of the motor. Additionally, construct a simple LED circuit which shines on the shaft so that the light reflects into the light sensor. Color half of the shaft black so that when it rotates the light reflecting into the light sensor will alternate between brighter and dimmer.

Figure 6: Motor Driver Circuit

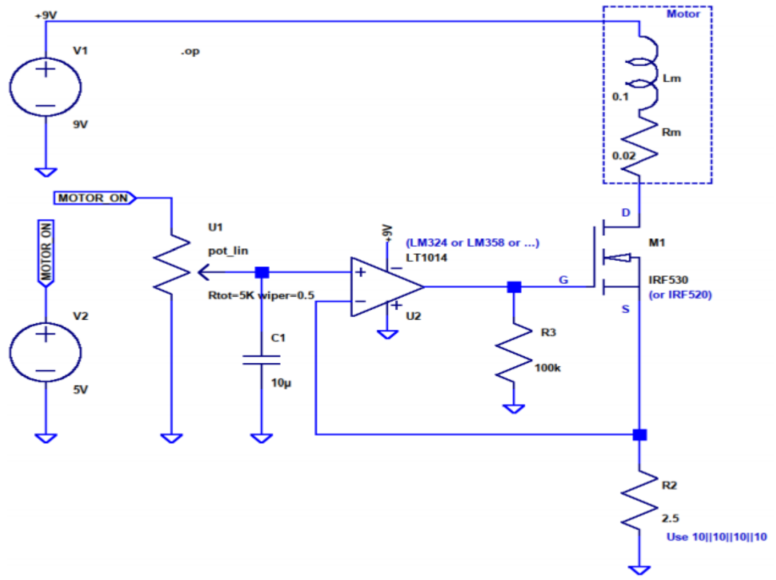
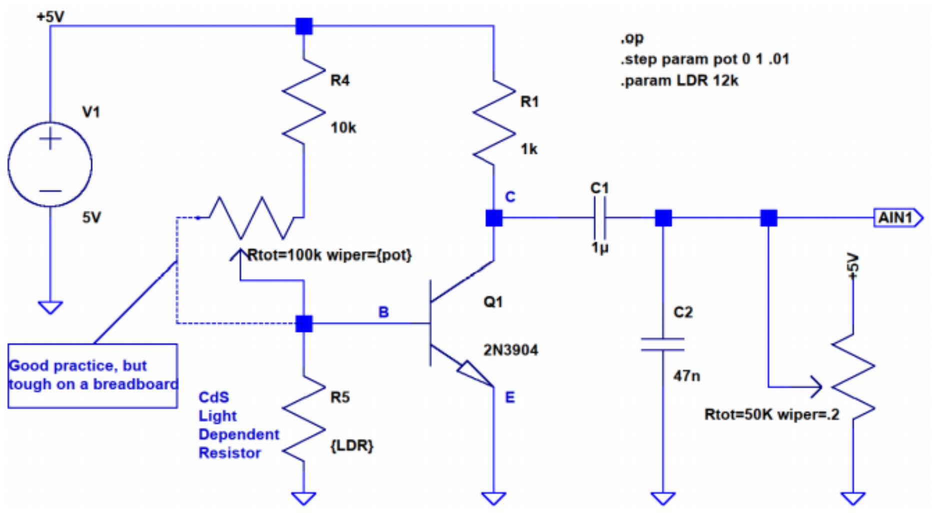


Figure 5: Op-amp front-end for the MOSFET transistor. This dramatically reduces the effect of glitches from the PWM (which may come from noise-induced triggers of the light sensor). Note that the 2.5 Ω resistor should have a power rating of 1 W. The easiest way to obtain the required value and power rating is to use four 10-Ω 1/4-W resistors in parallel. Adjust the 5 kΩ potentiometer to limit the motor current (with a 9 V battery, the adjustment may be very close to the wiper=1 limit).

Figure 7: Rotation Sensing Circuit



Light-sensing circuit. The LDR resistance decreases for higher light intensity. This reduces the transistor base voltage under illumination, turning the transistor off. Under ambient lighting with the green LED on, a typical LDR value is ∼15 kΩ. The 100 kΩ potentiometer is used to tune the transistor base voltage to about 0.65 V, which should put point *C* at 2.5 V to 3.5 V. The output to AIN1 is AC coupled (high-pass filtered) through C_1 ($f_c \sim R_{pot}C_1$) and low-pass filtered by C_2 ($f_c \sim R_1C_2$) to eliminate noise glitches which can cause spurious triggers. The potentiometer at the output is used to offset the AC signal relative to the AIN0 reference voltage. Adjust this to obtain stable triggering of the Analog Compare interrupt when the motor is running.

- It will be quite evident if the motor driver function properly. For this part of the lab rather than altering the signal at MOTOR ON, we provide a constant 5V signal and adjust the power of the motor using the current limiting potentiometer. Note: To save battery life you must cut the 5V signal altogether.
- To configure the Rotation sensing circuit, use the MyDAQ and follow the guidelines in the caption. Note: as you change your ambient light conditions, the light sensor will also change resistance and so the potentiometers will need to be adjusted.

3.3 – Measure!

- After successful setup and confirmation using the MyDAQ, use the interval timer program from section 2.2 to measure the time between light on/off events.
    - The biggest difference in our code from section 2.2 is that the analog comparator must edge-trigger, we can not use toggle otherwise we would get twice the interrupts.
- Turn on the motor and measure its period, compare results with oscilloscope measurements.
- We continue further to alter our code to include a signal averaging feature. This will prove especially useful for later when implementing control features because it produces a smoother output signal.

- Our measurement code continues to adapt throughout section 3. We will present and discuss the final code only since it encapsulates the previous subsections.

3.4 – Output

- We will utilize an application called PuTTY to capture the data being streamed to our serial port which we previously displayed using the Serial Monitor.
- After successfully setting up and confirming that we can stream and store data, we run several test cases to create multiple data files.
- Then, using Python through Jupyter Lab we extract our data from our files to create graphs. Those graphs will be displayed in our results section.

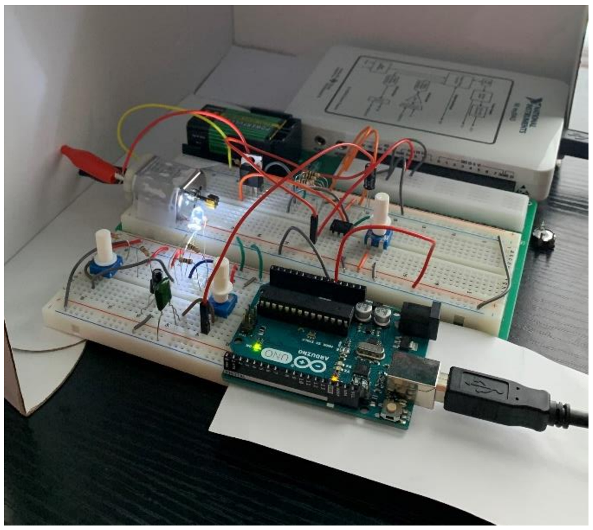
3.5 – Speed Control

- For the final part of this lab, we will establish PWM control and create a PD controller in software. Both these functions will also be controlled by the microcontroller.
- Up until now we have simply powered the motor using a constant MOTOR_ON voltage of 5V. Now, we will use PWM to control MOTOR_ON. To achieve this, we will simply use the analogWrite() Arduino function. As such, we are drifting away from AVR style coding for the sake of time.
- After this is achieved, we create a proportional controller. This is the most basic controller needed to achieve a setpoint frequency.
- Along with the proportional controller, we also include the code to take in an input from the serial port for a new setpoint frequency.
- Finally, we incorporate derivative control to minimize the oscillatory nature of the proportional controller when the setpoint is changed.

Results

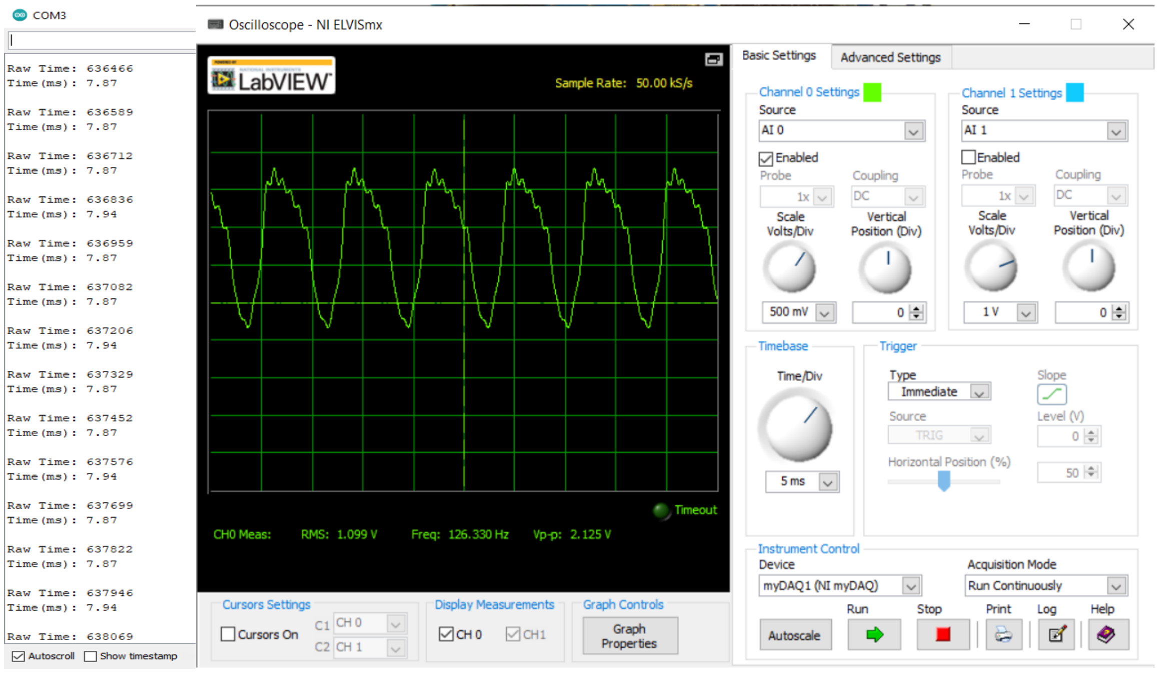
3.1 and 3.2 – Motor Driver and Rotation Sensor

Figure 8: Circuit Setup



3.3 – Measure!

Figure 9: Configuration Measurements on Serial Monitor and Oscilloscope



COM3

```
Raw Time: 636466
Time(ms): 7.87

Raw Time: 636589
Time(ms): 7.87

Raw Time: 636712
Time(ms): 7.87

Raw Time: 636836
Time(ms): 7.94

Raw Time: 636959
Time(ms): 7.87

Raw Time: 637082
Time(ms): 7.87

Raw Time: 637206
Time(ms): 7.94

Raw Time: 637329
Time(ms): 7.87

Raw Time: 637452
Time(ms): 7.87

Raw Time: 637576
Time(ms): 7.94

Raw Time: 637699
Time(ms): 7.87

Raw Time: 637822
Time(ms): 7.87

Raw Time: 637946
Time(ms): 7.94

Raw Time: 638069
```

☑ Autoscroll ☐ Show timestamp

Oscilloscope - NI ELVISmx

Sample Rate: 50.00 kS/s

CH0 Meas: RMS: 1.099 V Freq: 126.330 Hz Vp-p: 2.125 V

Channel 0 Settings: Source AI 0; ☑ Enabled; Probe 1x; Coupling DC; Scale Volts/Div 500 mV; Vertical Position (Div) 0

Channel 1 Settings: Source AI 1; ☐ Enabled; Probe 1x; Coupling DC; Scale Volts/Div 1 V; Vertical Position (Div) 0

Timebase: Time/Div 5 ms

Trigger: Type Immediate; Source TRIG; Level (V) 0; Horizontal Position (%) 50

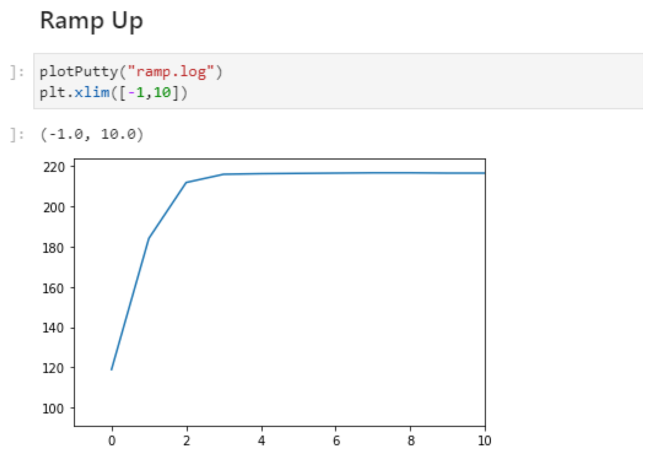
Instrument Control: Device myDAQ1 (NI myDAQ); Acquisition Mode Run Continuously

Cursors Settings: ☐ Cursors On; C1 CH 0; C2 CH 1

Display Measurements: ☑ CH 0 ☑ CH1

3.4 – Output

Figure 10: Frequency (Hz) Data During Motor Ramp Up Period

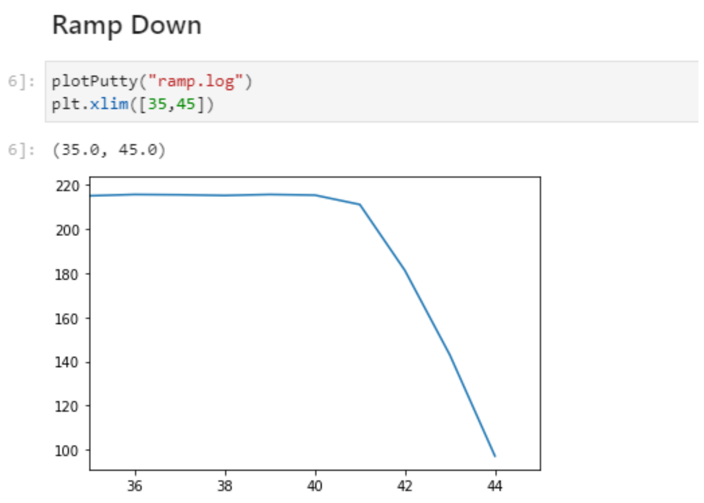


Ramp Up

```
]: plotPutty("ramp.log")
   plt.xlim([-1,10])
```

```
]: (-1.0, 10.0)
```

Figure 11: Frequency (Hz) Data During Motor Ramp Down Period



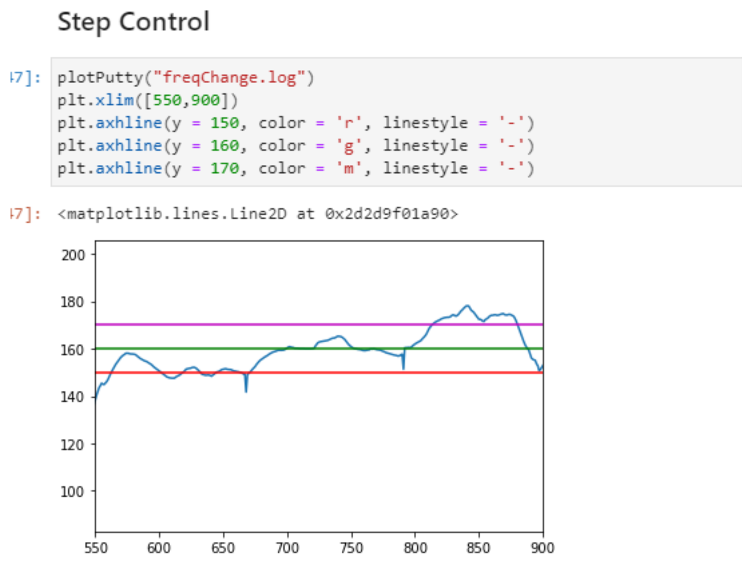
Ramp Down

```
6]: plotPutty("ramp.log")
    plt.xlim([35,45])
```

```
6]: (35.0, 45.0)
```

3.5 – Speed Control

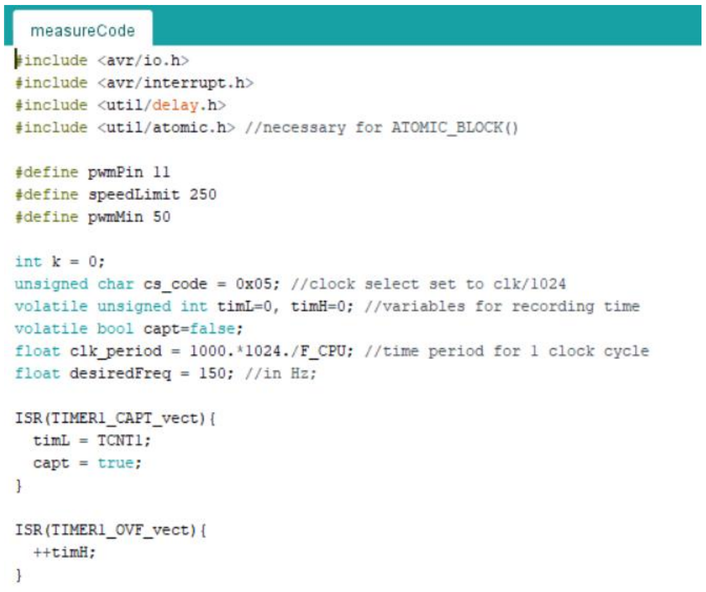
Figure 12: Frequency (Hz) Data Stepping from Freq = 150,160,170



Step Control

```
[47]: plotPutty("freqChange.log")
      plt.xlim([550,900])
      plt.axhline(y = 150, color = 'r', linestyle = '-')
      plt.axhline(y = 160, color = 'g', linestyle = '-')
      plt.axhline(y = 170, color = 'm', linestyle = '-')

[47]: <matplotlib.lines.Line2D at 0x2d2d9f01a90>
```

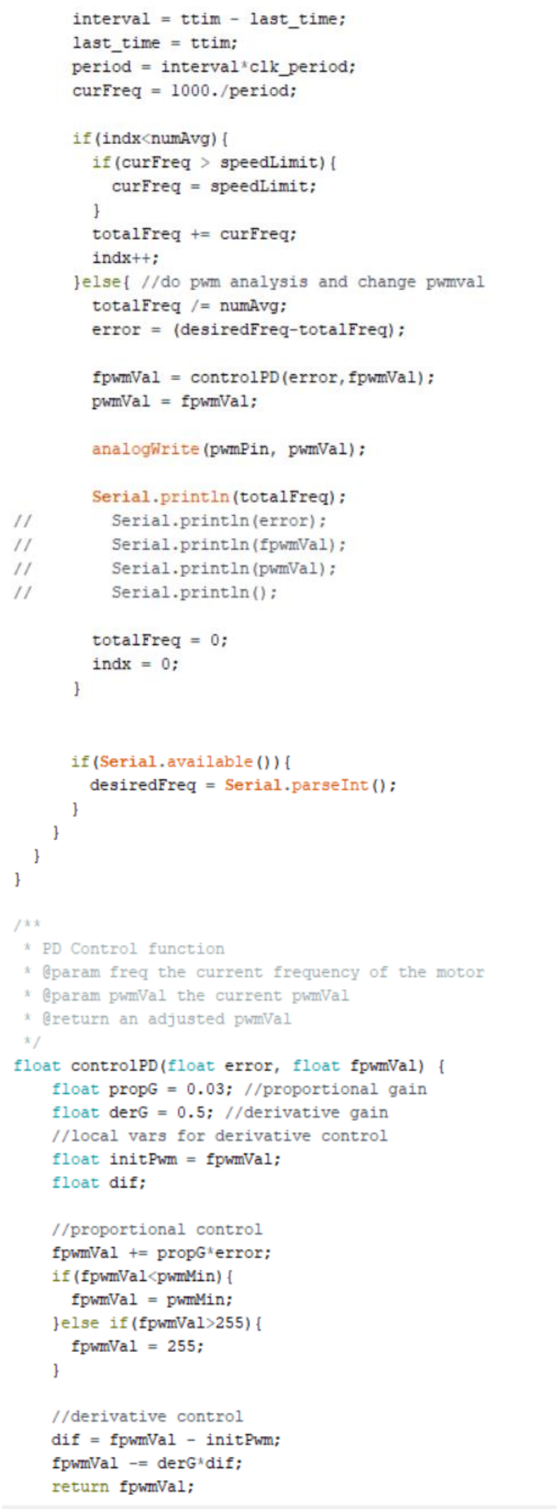
Figure 13: Code for PD Controller



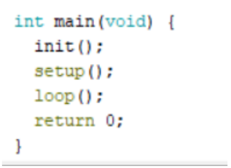
measureCode

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <util/atomic.h> //necessary for ATOMIC_BLOCK()

#define pwmPin 11
#define speedLimit 250
#define pwmMin 50

int k = 0;
unsigned char cs_code = 0x05; //clock select set to clk/1024
volatile unsigned int timL=0, timH=0; //variables for recording time
volatile bool capt=false;
float clk_period = 1000.*1024./F_CPU; //time period for 1 clock cycle
float desiredFreq = 150; //in Hz;

ISR(TIMER1_CAPT_vect){
  timL = TCNT1;
  capt = true;
}

ISR(TIMER1_OVF_vect){
  ++timH;
}
```

```
void setup() {
// put your setup code here, to run once:

  noInterrupts();

  //analog comparator control
  ACSR &= 0x00;
  ACSR &= ~(1<<ACD); //CLEAR analog comparator diable
  ACSR |= (1<<ACBG); //Set analog comparator bandgap select (AIN0 = 1.1V)
  ACSR &= ~(1<<ACIE); //CLEAR analog comparator interrupt enable
  ACSR |= (1<<ACIC); //SET analog comparator input capture enable
  ACSR |= 0x03; //interrupt condition (rising edge)
  ADMUX |= (1<<REFS1) | (1<<REFS0); //use internal 1.1 ref voltage

  //ADC control and status register A/B
  ADCSRA |= (1<<ADEN); //AIN1 is used as negative input (instead of adc0,1,2...)
  ADCSRB &= 0x00; //CLEAR, free running mode

  //Digital Input Disable Register
  DIDR1 |= (1<<AIN1D) | (1<<AIN0D); //disable digital input since we are using as anlog input, reduces pwr consumption

  //Timer/Counter 1 control registers
  TCCR1A = 0; // Set the OCR pins to normal I/O and clear half of the WGM
  TCCR1B = 0; // Stop the timer clock and clear other stuff
  TCCR1B |= (1<<ICNC1)|(1<<ICES1)|cs_code; //set noise cancellation, rising edge select, and clock select
  TIFR1 |= (1<<ICF1);

  //interrupts
  TIMSK1 = 0; // Clear all interrupt enable mask bits
  TIMSK1 |= (1<<ICIE1) | (1<<TOIE1); //set input capture interrupt enable and overflow interrupt enable.
  TIFR1 = 0xFF; // Clear any pending interrupt flag (Yes, write a '1' to CLEAR a flag.  Saves read/modify/write cycles.)
  sei(); //set bit 7 of AVR Status Register (SREG), Global Interrupt Enable


  interrupts();  // Nothing will happen until TCCR1B is written

  Serial.begin(9600);
  pinMode(pwmPin, OUTPUT);
  analogWrite(pwmPin, 255);
}


void loop() {
  unsigned long ttim = 0; //current time
  unsigned long last_time = 0; //previous time
  unsigned long interval = 0; //difference between current and prev time
  float period = 0; //period of rotation
  float curFreq = 0; //frequency of rotation
  float totalFreq = 0; //total freq used for averaging
  float fpwmVal = 100.;
  float error = 0;
  int pwmVal;
  int numAvg = 20;
  int indx = 0;

  while(true){
    // put your main code here, to run repeatedly:
    if (capt) {
      ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
        ttim = ((unsigned long)timH<<16)|(unsigned long)timL;
        //ttim = timL;
        capt = false;
        //timH = 0;
      }
```



```
      interval = ttim - last_time;
      last_time = ttim;
      period = interval*clk_period;
      curFreq = 1000./period;

      if(indx<numAvg){
        if(curFreq > speedLimit){
          curFreq = speedLimit;
        }
        totalFreq += curFreq;
        indx++;
      }else{ //do pwm analysis and change pwmval
        totalFreq /= numAvg;
        error = (desiredFreq-totalFreq);

        fpwmVal = controlPD(error,fpwmVal);
        pwmVal = fpwmVal;

        analogWrite(pwmPin, pwmVal);

        Serial.println(totalFreq);
//        Serial.println(error);
//        Serial.println(fpwmVal);
//        Serial.println(pwmVal);
//        Serial.println();

        totalFreq = 0;
        indx = 0;
      }


      if(Serial.available()){
        desiredFreq = Serial.parseInt();
      }
    }
  }
}

/**
 * PD Control function
 * @param freq the current frequency of the motor
 * @param pwmVal the current pwmVal
 * @return an adjusted pwmVal
 */
float controlPD(float error, float fpwmVal) {
    float propG = 0.03; //proportional gain
    float derG = 0.5; //derivative gain
    //local vars for derivative control
    float initPwm = fpwmVal;
    float dif;

    //proportional control
    fpwmVal += propG*error;
    if(fpwmVal<pwmMin){
      fpwmVal = pwmMin;
    }else if(fpwmVal>255){
      fpwmVal = 255;
    }

    //derivative control
    dif = fpwmVal - initPwm;
    fpwmVal -= derG*dif;
    return fpwmVal;
```



```
int main(void) {
  init();
  setup();
  loop();
  return 0;
}
```

Discussion

3.1 and 3.2 – Motor Driver and Rotation Sensor

We had minimal difficulty constructing our circuits. For operation, constantly using the oscilloscope was required because light conditions were constantly changing. Even the change in brightness from midday to afternoon would change our output.

As seen in Figure 8, we constructed a simple cardboard enclosure to prevent the effects of these light changes.

3.3 – Measure!

We can see from Figure 9 that our calculated periods, which are outputted to the serial monitor and seen on the left of the figure are very near the oscilloscopes measure frequency. 1/f = T , 1/0.008 = 125.

Therefore, we can conclude that our measurements are accurate. Moving forward, it is important to note that the shape of the waveform changes as the frequency changes. Therefore, when drastic frequency changes are made, it is sometimes important to also change the potentiometer values. Otherwise, you may pick up signals such as the AC 60Hz noise, thus triggering additional interrupts leading to higher than expected frequencies.

3.4 – Output

We successfully generate two figures (Figures 10 and 11), one is of the motor turning on to max RPM, and one of it turning off. The Details of these two figures are not particularly pertinent to the purpose of this lab. They simply show that we can record and display our frequency data, which is again used in the next section. A great improvement to these figures would be to add a timescale and axes. For each figure, the y axis represents frequency.

3.5 – Speed Control

We took significantly longer to complete this portion of the lab than anticipated. When it comes to software, small problems can cause great pains. In the end however, we got both our code and hardware to behave as expected. In Figure 12, you can see this behavior. First, we have our motor ramp to 150Hz (out of the figure). After leaving it there for a short time, we ramp to 160Hz, and finally 170Hz. The stepwise nature of our ramping is evident in the behavior of our motor. Although there is still a great deal of oscillations, we can see that our motor quickly approaches the desired output value. We can also clearly see when new data values were set by a vertical spike in the data.

In addition to this graph, we have also created a brief demo video which can be found on YouTube at the following link: https://youtu.be/N4kkcVYROYU

Now that we have clearly demonstrated the functionality of our code and circuit, we must discuss the code itself. Our setup() function is the same as before, we have not altered our triggering or time mechanisms. Alterations start after measuring the frequency in our if statement with the condition (indx<numAvg). Using this if statement, we only continue to our PD control every 20th measurement (we set NumAvg to 20). We then calculate an error function and pass that value to our PD control function.

We track our PWM value as both a float and an integer. We must track it as a float so that our PD controller can make changes smaller than 1, however, we must pass an integer into our PWM function. In our controlPD() function we set our proportional and derivative gain, calculate our proportional control change, use that to calculate our derivative control change, and then return our final value as expected. A key to note is that we set both an upper and lower limit on the frequency input. This is to avoid unexpected behavior. Further we put bounds on the output PWM value. The max value is 255, the max value for analogWrite(). The min value is defined as pwmMin and is set to 50. This is to prevent overshooting which cuts off the motor when trying to achieve low frequency values.

Conclusions

In conclusion, all three parts of this lab had its challenges. At first, it was very difficult to even understand what each control register does. When coding microcontrollers directly, in the "AVR" style, there is much more to keep track of than when using most modern language programming languages.

At this point, we have had a great deal of exposure to constructing circuits. As such, since the circuit diagrams were given to us, we did not have much difficulty building the physical circuit. There were challenges with regards to tuning and changing light conditions, and of course there was the occasional misplaced wire. However, in all, the hardware did not pose as big of a challenge as the software for this lab, if we remained conscious of common hardware issues.

With that said, the greatest challenge throughout this lab was our lack of knowledge of the ATmega328p functionality. However, with constant reference to the datasheet even that was resolved. In the end, we were able to produce a PD controller that functioned very well.