Embedded Group Project

Project 2 - Speed Controller

Steinarr Hrafn Höskuldsson
Arnþór Gíslason
Andrew Madden

Reykjavik University

September 2022

1 Part 1

> Implement the Initialization and Operational states and the reset command. You can use the Arduino Serial class to issue simple commands from the keyboard to trigger a state transition.

A communication scheme was devised. To send a command, e.g. an order to switch between states, only a single upper case letter should be sent. To set a variable a string on the form '(lower case letter)=(floating point value)' shall be sent, the letter indicates which value to be set. Things sent can not be longer than 10 characters and should be separated by a newline character.
Functions to receive Serial input were added to our `HackySerial` module. A function was written that parses a string on the form: `p=3.45` to a `"Parsed"` struct, which contains the character, 'p' and the floating point number 3.45.
A program was written based on Refactoring Guru's state machine example. In the main loop a section was written that asynchronously receives data on the Serial port according to the communication scheme and switches between states on appropriate commands.

A lot of work was put into moving the programming of each state into it's own file. However that work was concurrent with the debugging of the control code and we ran out of time and were unable to merge the divergent branches.

A state diagram was created using PlantUML. The State Diagram can be seen in Figure 1

2 Part 2

Brake was implemented in the `motor_controller` class where two methods were tried. First method was setting the set point to 0 and have the motor controller apply active breaking and second method was using the h bridge to short the poles of the motors effectively providing passive brake.
For this application, having active brakes wasn't needed to stop the motor in a suitable time and there wasn't any active load on the system which could've caused the motor to creep forwards. Adding active braking would also have introduced complexity in the program which might cause safety issues.

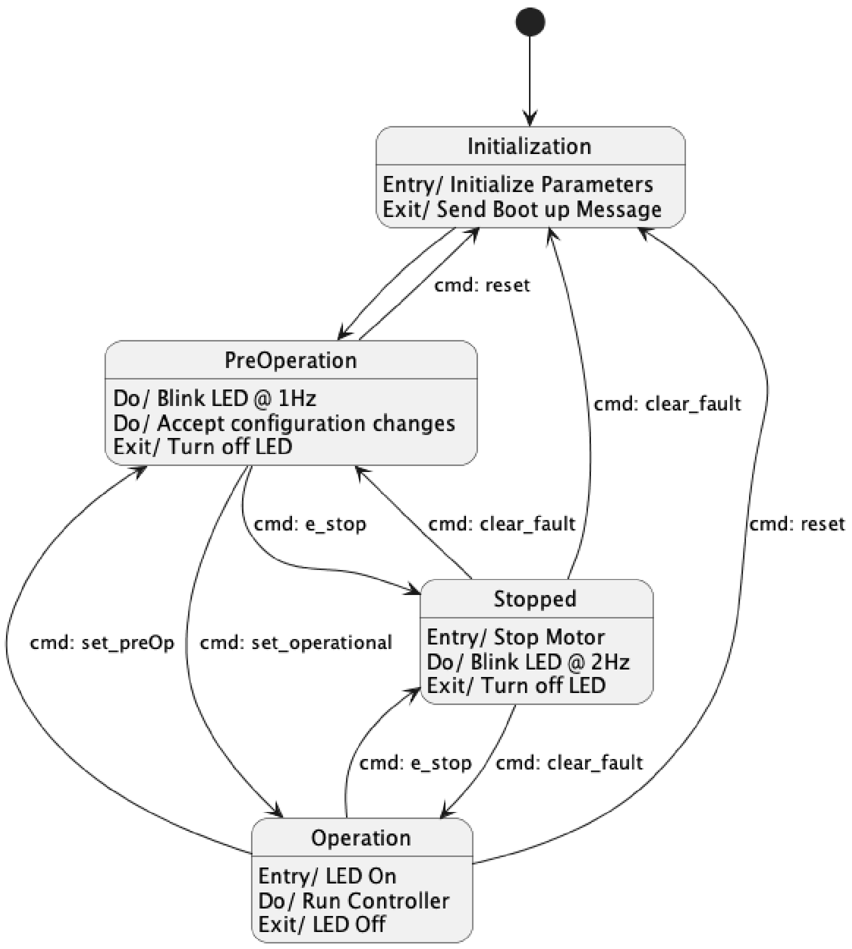
Initialization
Entry/ Initialize Parameters
Exit/ Send Boot up Message
cmd: reset
PreOperation
Do/ Blink LED @ 1Hz
Do/ Accept configuration changes
Exit/ Turn off LED
cmd: clear_fault
cmd: e_stop
cmd: clear_fault
cmd: reset
Stopped
Entry/ Stop Motor
Do/ Blink LED @ 2Hz
Exit/ Turn off LED
cmd: set_preOp
cmd: set_operational
cmd: e_stop
cmd: clear_fault
Operation
Entry/ LED On
Do/ Run Controller
Exit/ LED Off

Figure 1: UML

3 Part 3

3.1 PI_controller

Instead of having `PI_controller` inherit from `P_controller` a new base class, `controller` , was made that housed the common variables. Update was then implemented in `PI_controller` and `P_controller`. Alternatively, `PI_controller` could've inherited from `PI_controller` and overridden update().
The `PI_controller` subsists of both proportional gain and integral gain where the proportional gain is the error between set point and actual multiplied by an adjustable scalar, and the integral gain is the accumulated error over time.
In order to overcome losses in a system, integral gain is a vital factor of the control system. In the case where there's no integral factor, the controller will not reach the desired set point and any attempts to reach the desired set point by increasing the proportional gain will result in the system becoming unstable.
With an integral factor the accumulated error will result in an ever increasing output until the actual value reaches the set point and the integral starts to become stable.

3.1.1 Windup Guard

When the actual value either takes too long to reach the setpoint or the load on the system is enough to prevent the motor from reaching the set point a condition may arise where the integral grows excessively large and it may take the system some time to wind the integral down. A simple method to prevent this condition is to limit the internal integral value to not exceed the output limits.

3.2 RPM Gauge

The first iteration of the rpm metering module consisted of a timer running at a set period, `delta_t`, and each cycle the interrupts were counted between cycles. This method resulted in a fairly accurate value that was mostly immune to inaccuracies in the encoder wheel. However, the resolution of the measured rpm was around 400rpm,it was deemed too inaccurate and a new method was implemented.
With the new method, a counter was set up that incremented at $500ns$ intervals and instead of counting the amount of interrupts during a set period the time between pulses was measured. When the timer overflows it's safe to assume that the motor is stationary and rpm is zero.

$$rpm = \frac{60\cdot}{PPR} \cdot t$$

Where 60 is seconds to a minute, ppr is pulses per revolution and t is time it took between pulses.
Absolute time can be found with $t = \frac{t_c}{counts}$ where t_c is time/count and counts is ultimate count between pulses
At first, both phases of the encoder were used but what's believed to be inaccuracies in the positioning of the sensors of the encoder resulted in the measured value jumping 1000rpm often during the revolution, hardly any better than the previous method.
After changing to using a single phase of the encoder, the measured value reached acceptable repeatability, however, a repeating pattern may be seen in the waveform which is likely attributed to manufacturing tolerances in the magnetic encoder wheel. See 2.

The rpm was stabilized further by taking a rolling average of the measured values and it was determined that the average of 10 samples would be adequate to reach a clean waveform.

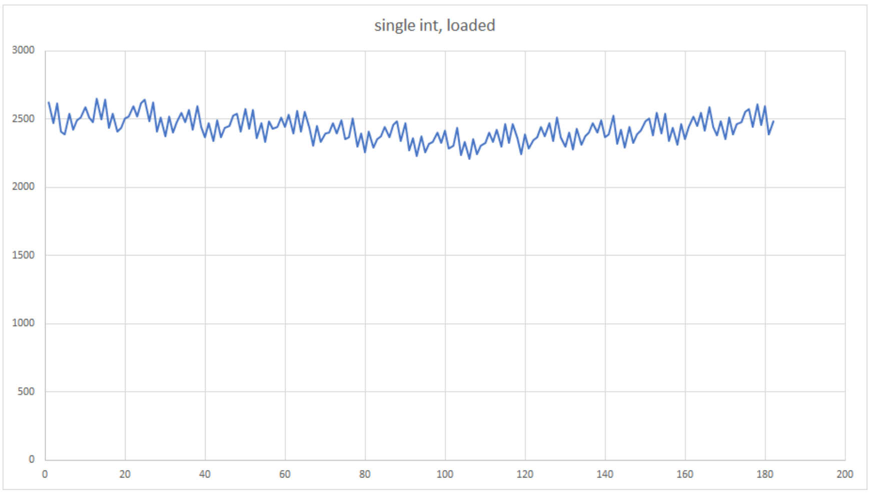

Figure 2: repeating pattern indicating inaccuracies in the encoder wheel

4 Part 4

The Ziegler-Nichols method was used to tune the system. The integral gain was set to zero and proportional gain was increased until stable oscillations could be seen in the output. A graph of the oscillations can be seen in Figure 3. The ultimate gain was found to be $K_u = 7.2$, $T_u = 26ms$, At that point a suitable values for K_p and K_i were fetched from a table according to the Ziegler-Nichols tuning method giving $K_p = 3.45$ and $K_i = 150$

With the new values the system behaved quite nicely as can be seen in the step response in Figure 4

A good load reponse pattern may be seen in figure 7

With the new rpm method counting the pulses of the rotary encoder is strictly unnecessary but it was necessary to monitor both phases to determine which direction the motor was turning. Having the counts could also help for future use.

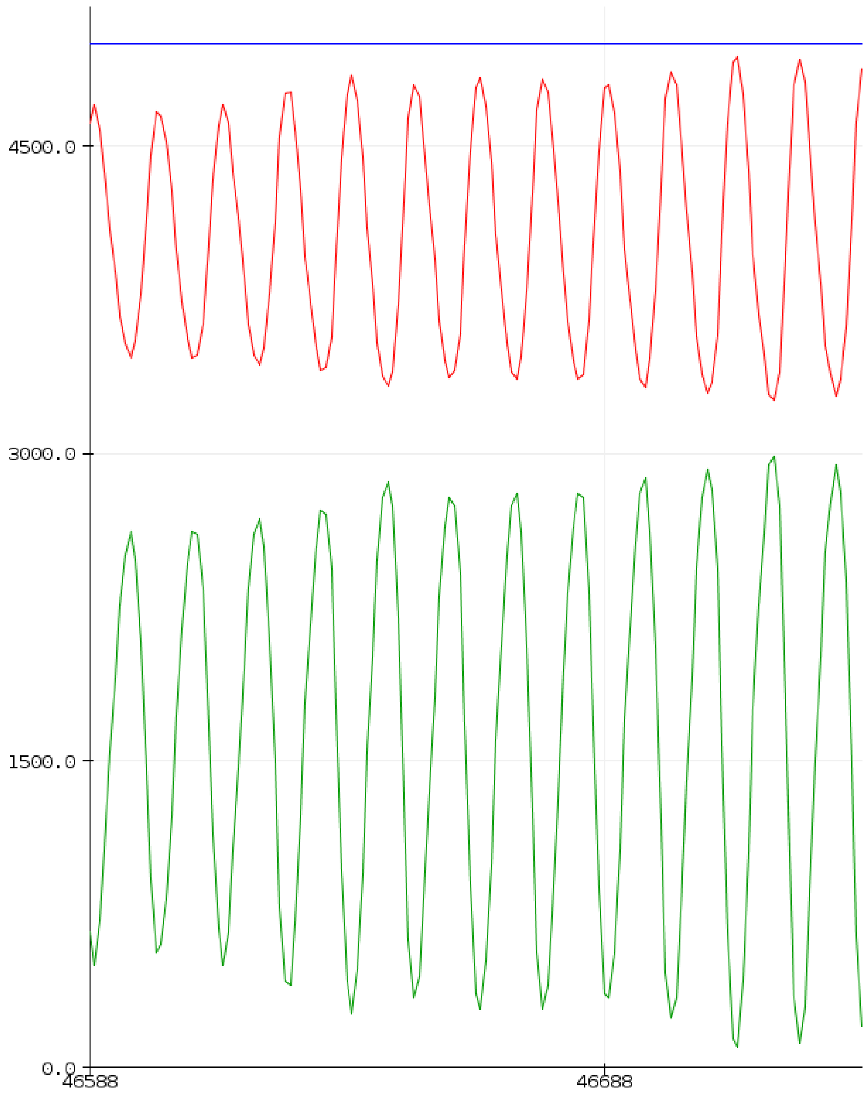

Figure 3: Stable Oscillations were observed with $K_p = 7.2, K_i = 0$, Set point is in blue, the observed rpm is in red and the scaled up pwm duty is in green

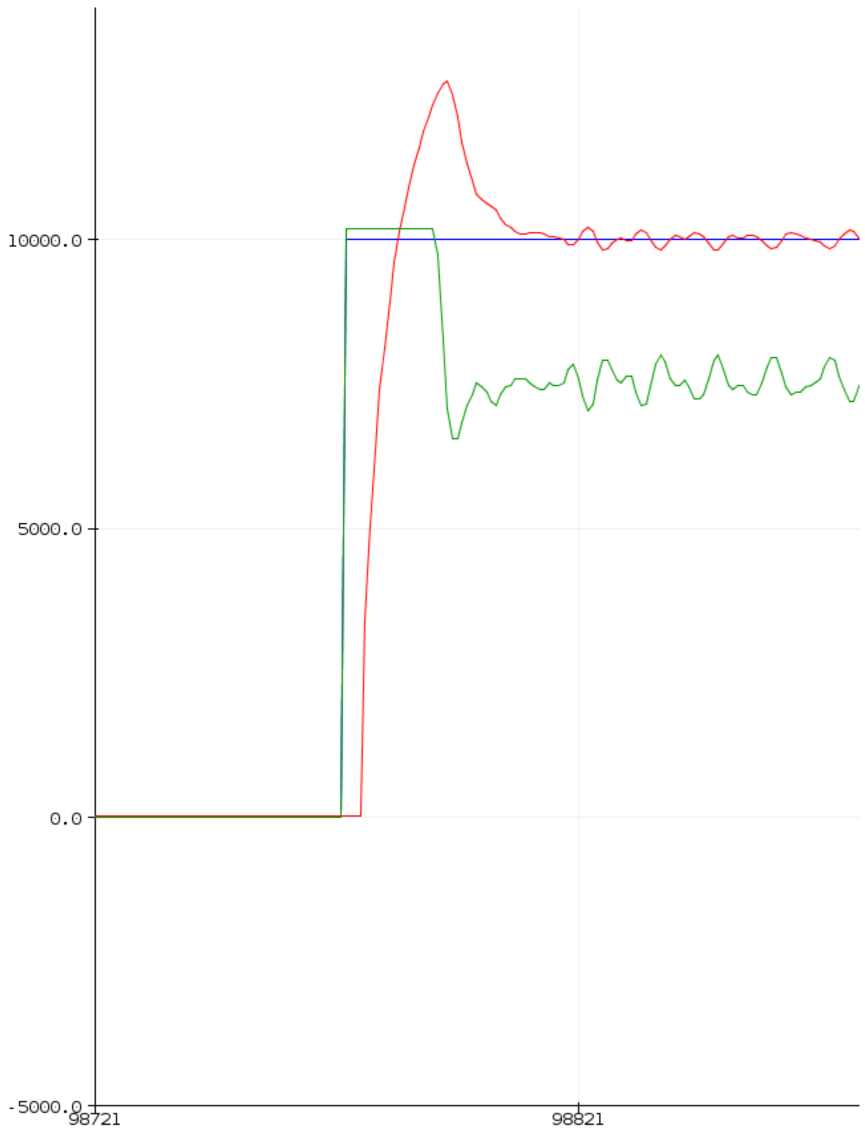

Figure 4: step response from 0 to 10krpm with $K_p = 3.24, K_i = 150$, Set point is in blue, the observed rpm is in red and the scaled up pwm duty is in green

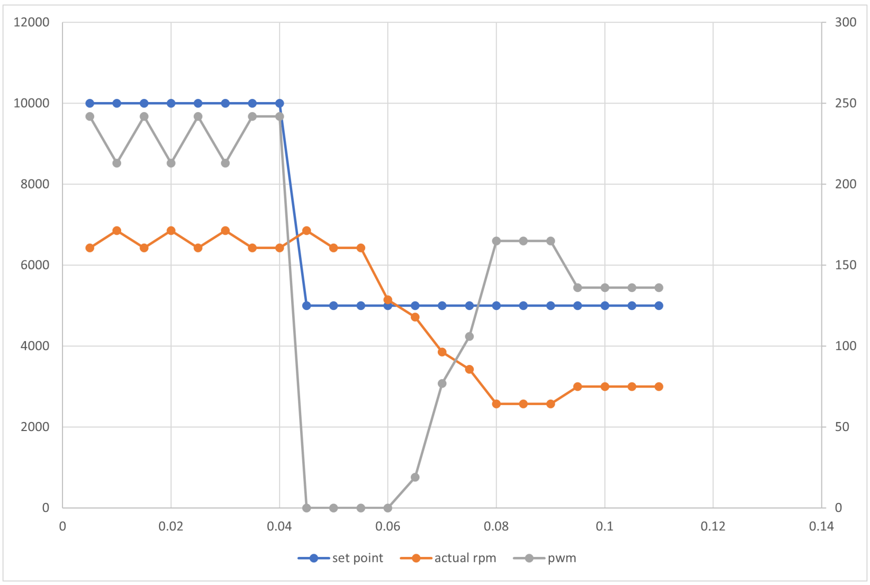

Figure 5: Step response for old p controller

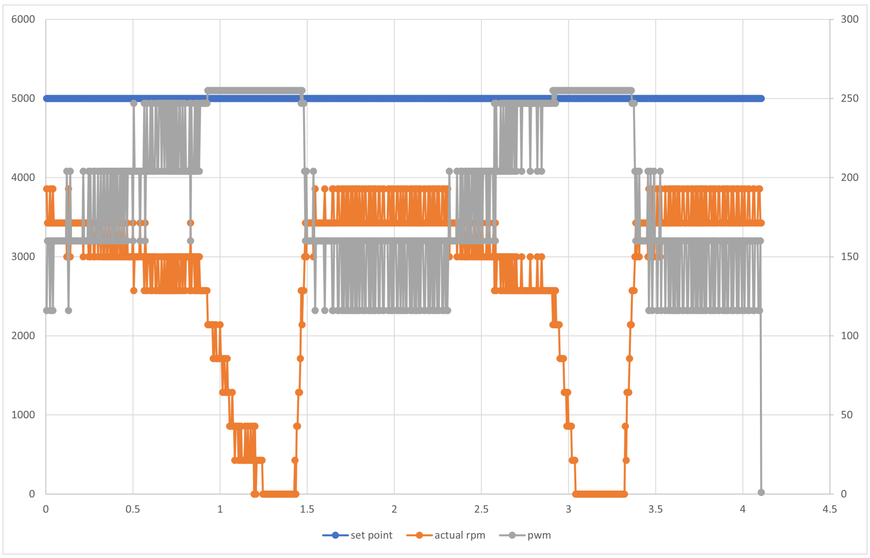

Figure 6: load response for old p controller

A video showing the motor controller in operation can be seen at `https://youtu.be/YvaofKchtCM` The code used can be found on github at `https://github.com/Steinarr134/EmbeddedGroupProjects/tree/main/Project3ControllerStateMachine`

5 Discussion

Initially, there were problems with timer0 and timer1.
Timer0 wouldn't set a compare match interrupt until TCNT0 register was manually set to 0 in the interrupt routine. Additionally, timer0 would get glitched and not interrupt at the correct period rather interrupt at delta_t plus time. This proved to happen exclusively when timer1 overflow interrupt ran, however, the timer1 overflow interrupt wasn't being used so disabling it ameliorated the issue.
The encoder on the motor seems to be rather poorly made, the position of the magnetic sensors is not controlled accurately which causes the pulse width, between phases, at a constant rpm, to be inconsistent. Too inconsistent for it to be able to use the separate phases for higher resolution in rpm values. Although, 14 pulses per revolution is very adequate and it's certainly better than the previous method to measure rpm and the previous method seemed to be good enough.

Appendix

A Code

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <digital_out.h>
#include <timer_msec.h>
#include <encoder_simple.h>
#include <hackySerial.h>

#include <P_controller.h>
#include "PWM2.h"
#include <encoder_interrupt.h>
#include "speedometer.h"

#define DELTA_T (int32_t)5
#define INV_DELTA_T (int32_t)1.0 / ((float)DELTA_T / 1000.0)
#define PPR (int32_t)(7 * 2 * 2) // 7 pulses per phase and triggering on falling
    as well
#define GEAR_REDUCTION (int32_t)101
#define MAX_PWM 255
#define MAX_RPM 15000

uint16_t duty = 0;
int16_t set_point = 5000;
float kp = 6; // gain

Encoder_interrupt encoder;
Digital_out led(5);
Timer_msec timer;
PWM2 pwm;
P_controller controller(kp, MAX_RPM, MAX_PWM);
volatile int32_t counter = 0;
volatile int32_t delta_counts = 0;
volatile bool flag = false;
volatile uint16_t time = 0;
int32_t rpm = 0; // initialize just cause
unsigned int count = 0;
int32_t dc;

int main()
{
  USART_Init(0); // 1megabaud
  // USART_Init(MYUBRR); // 9600
  timer.init(DELTA_T);
  pwm.init();
  pwm.set(0);
  encoder.init();
  while (true)
  {
    if (flag)
    { // if there's a new measurement available
      dc = delta_counts;
      if (dc < 0)
      {
        dc = -dc;
      }
      rpm = dc * (int32_t)60 * INV_DELTA_T / PPR;
      duty = (int16_t)controller.update(set_point, rpm); // RPM of input shaft,
    not rpm of output shaft!!
      print_3_numbers(set_point, rpm, duty);
      pwm.set(duty);
      flag = false;
      count++;
    }

    if (count > 1000)
    { // to vary the set point
      count = 0;
      if (set_point == 5000)
      {
        set_point = 10000;
      }
      else
      {
        set_point = 5000;
      }
    }
  }
}

ISR(INT0_vect)
{
  encoder.pin1();
}

ISR(INT1_vect)
{
  encoder.pin2();
}

ISR(TIMER1_COMPA_vect)
{
  delta_counts = encoder.position();
  encoder.reset();
  flag = true;
}
```

Listing 1: main program used to produce the step response

```
#include <encoder_simple.h>
#include <avr/delay.h>
#include <digital_out.h>
#include <avr/io.h>

Encoder_simple::Encoder_simple(){}

void Encoder_simple::init()
{    //PB1, pin 9 on Arduino Nano
    input1.init(1, true);          // use internal pull up
    input2.init(2, true);
}

long Encoder_simple::position()
{
    return counter;
}

void Encoder_simple::monitor()
{
    if (input1.is_hi())
    {
        if (last_state1 == 0)
        {
            PORTB ^= (1<<5);
            if (input2.is_hi())
            {
                counter++;
            }
            else
            {
                counter--;
            }
            last_state1 = 1;
        }
    }
    else
    {
        last_state1 = 0;
    }

    if (input2.is_hi())
    {
        if (last_state2 == 0)
        {
            PORTB ^= (1<<5);

            if (input1.is_hi())
            {
                counter--;
            }
            else
            {
                counter++;
            }

            last_state2 = 1;
        }
    }
    else
    {
        last_state2 = 0;
    }
}
```

Listing 2: timer_msec.cpp

```
#include <avr/io.h>
#include <avr/delay.h>
#include <avr/interrupt.h>
#include <digital_out.h>
#include <timer_msec.h>
#include <encoder_simple.h>

// #include <digital_in.h>


// for printing to serial::
// USART code taken from datasheet for printing to serial
#define FOSC 16000000 // Clock Speed
#define BAUD 9600
#define MYUBRR FOSC / 16 / BAUD - 1

// ints to store last state of A and B
volatile uint8_t last_stateA;
volatile uint8_t last_stateB;


void USART_Init(unsigned int ubrr)
{
  /*Set baud rate */
  UBRR0H = (unsigned char)(ubrr >> 8);
  UBRR0L = (unsigned char)ubrr;
  // Enable receiver and transmitter * /
      UCSR0B = (1 << RXEN0) | (1 << TXEN0);
  /* Set frame format: 8data, 2stop bit */
  UCSR0C = (3 << UCSZ00);
}

void USART_Transmit(unsigned char data)
{
  /* Wait for empty transmit buffer */
  while (!(UCSR0A & (1 << UDRE0)))
    ;
  /* Put data into buffer, sends the data */
  UDR0 = data;
}

// helper function to print an integer
void print_i(int i)
{
  // speial case for when i is 0
  if (i == 0){
    USART_Transmit('0');
    USART_Transmit((unsigned char)10);
    return;
  }
  // if negative print minus sign and then just as if it were positive
  if (i < 0){
    USART_Transmit('-');
    i = -i;
  }
  // don't print leading zeros, however,do print zeros that are not leading
  bool leading_zeros_done = false;
  for (int j = 4; j>= 0;j--) // supports 5 digit numbers (ints can't be larger)
  {
    int m = pow(10, j);
    if (leading_zeros_done || i/m > 0){
      USART_Transmit((unsigned char)(((i/m)%10) + 48));
      leading_zeros_done = true;
    }
  }
  // finally print \n
  USART_Transmit((unsigned char)10);
}

// part 2 with interrupts

void set_interrupt_d2()
```

```
{
  DDRD &= ~(1 << DDD3);
  PORTD |= (1 << PORTD3);
  last_stateB = PIND & (1<<PIND3);
  EICRA |= (1 << ISC00);  // set INT0 to trigger on ANY logic change
  EIMSK |= (1 << INT0);   // Turns on INT0
  sei();                  // turn on interrupts
}
void set_interrupt_d1()
{
  DDRD &= ~(1 << DDD2);   // set the PD2 pin as input
  PORTD |= (1 << PORTD2);
  last_stateA = PIND & (1<<PIND2);
  EICRA |= (1 << ISC10);  // set INT1 to trigger on ANY logic change
  EIMSK |= (1 << INT1);   // Turns on INT1
  sei();                  // turn on interrupts
}


// for part 1
uint32_t counter = 0; // counts the pulses
volatile bool time_to_print = false; // printing flag
Timer_msec timer;  // timer to print every now and then
Encoder_simple encoder; // create the encoder

ISR(TIMER1_COMPA_vect)
{
  time_to_print = true;
  // led.toggle();
}
// Part 1


Digital_out led(5);


int main()
{
  bool part1 = false;
  USART_Init(MYUBRR);
  timer.init(50);
  led.init();
  // input.init(1,true);

  if (part1)
  {
    encoder.init();
    bool last = false;
    int c = 0;
    while (true){
      encoder.monitor();
      if (time_to_print)
      {
        time_to_print = false;
        print_i(encoder.position());
        // print_i(c);
        //led.toggle();

      }
    }
  }
  else
  {


    set_interrupt_d1();
    set_interrupt_d2();

    while (true){
    _delay_ms(100);
    USART_Transmit('#');
    print_i(counter);
    }

  }
}

// only counts up. never down
ISR(INT0_vect)
{
  // phase A
  last_stateA = PIND & (1<<PIND2);

  if(PIND & (1<<PIND2)) {
    if(last_stateB) {
      counter--;
    }
    else {
      counter++;
    }
  }
  else {
    if(last_stateB) {
      counter++;
    }
    else {
      counter--;
    }
  }
  PORTB ^= (1<<5);

}

ISR(INT1_vect)
{
    // phase B
  last_stateB = PIND & (1<<PIND3);
  if(PIND & (1<<PIND3)) {
    if(last_stateA) {
      counter++;
    }
    else {
      counter--;
    }
  }
  else {
    if(last_stateA) {
      counter--;
    }
    else {
      counter++;
    }
  }
  PORTB ^= (1<<5);
}
```

Listing 3: main.cpp

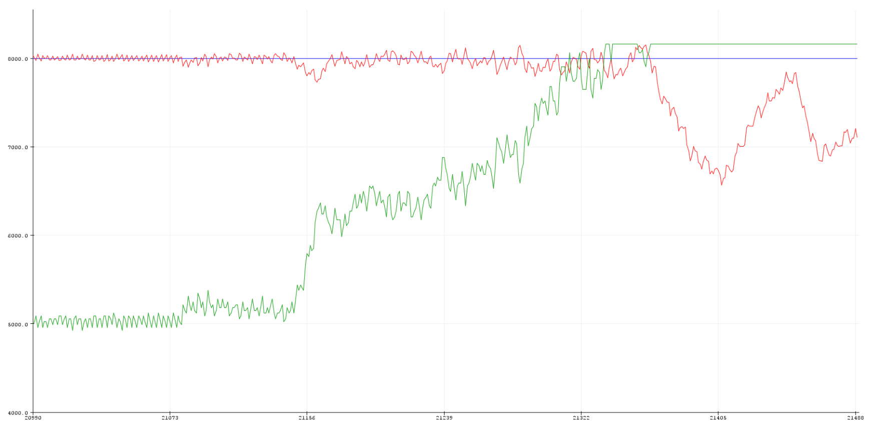

Figure 7: load response for the new PI controller. Set point is in blue, the observed rpm is in red and the scaled up pwm duty is in green