DC Motor Speed Controller
Final Written Report

Mason Nixon & Jake Bosarge

TO: Christopher Wilson

FROM: Mason Nixon & Jake Bosarge (THUR 3:30-5:20 p.m.)

DATE: 05/03/2010

SUBJECT: Final Written Report

Design Decisions

When designing our motor controller system, it was necessary to make several design decisions about primary systems of the motor controller. One such system is the input from the motor for determining speed of which we could choose between Analog-to-Digital conversion (ADC) or Input Capture. Aside from just being simpler, we determined that the best method would be the ADC method. If we were to employ the input capture method it would take much more code than with the ADC. Also, the ADC system on the HCS12 is built for this exact purpose: to take an analog input and convert it to a usable form for our coding. One major limitation however is that the ADC input can only capture with up to 10-bit resolution.

Another decision we had to make was how to determine the rpm speed: either the comparator or use a conversion factor from measured frequencies. One major difference is cost. By not using the comparator we eliminated the additional cost of a comparator and all of the hardware additions it would require. This, however, was at the expense of accuracy. By not knowing the exact frequency in our calculations, we introduced some error. We believed that this was justified since our speed display only showed the first two digits, and since our method could presumably only be flawed by the resolution of the ADC, we determined that we could have a minimum error of around 18 rpm since every ADC value could change the rpm speed in increments of 18 (See the *Software Design* section below for more details).

Hardware Design

For the motor control hardware design (See the attached figure) we were able to make a compact design for the two main systems: the speed display and the motor drive system. To power the motor, the 2N2222 NPN BJT was employed as a switch. We used the fact that the speed of a DC motor is directly proportional to the voltage applied across its windings. We send in a pulse width modulated signal from Port T3 which will drive the motor. By varying the pulse width duty cycle, we are able to create variable speed motor. Unfortunately, the motor requires 12VDC to be driven and the microcontroller can only output a maximum of 5VDC. By tying the emitter to ground and the collector to the negative terminal of the motor, then by tying the positive terminal to the 12V source, we were able to implement this variable speed control. Because of the switching mechanism of the transistor, it was also necessary to protect it against the transients during the "turn-off" mode. To do this, we employed an anti-parallel or "freewheeling" diode, the 1N4001, wired in parallel with the motor.

For the feedback loop of our control system, we needed some way to measure the motor speed. Since the Buehler motor came equipped with a built-in tachometer, we were able to use its output to measure the motor speed. The tachometer would produce an approximately 30V sine wave whose frequency (and voltage amplitude) is proportional to the speed of the motor. Since the HCS12 could only handle inputs no higher than about 4V, we needed to step this voltage down. Also, we determined that it would be most practical to use the voltage amplitude instead of the

frequency, so we employed a full-wave rectifier to convert the periodic signal to a constant dc signal. This signal was then the input into the Analog-to-Digital Port AD1 and the amplitude was then used to determine the motor speed (which is described in more detail in the *Software Design* section.

Software Design

The software code was uploaded to the HCS12 by using the *Freescale CodeWarrior* IDE. The code was written in the C programming language and a flow chart and the final code itself can be found as an attachment to this report. The code can be broken down into four I/O systems: *the output PWM motor driver, the display output, the motor speed input, and the compensator*. The PWM, whose function was described in the Hardware Design section, was set up with a period set to 150 with a prescale of 2. We used clock A and a begin high polarity on timer channel 0. Our MODRR bit 0x08 (bit 3) was set high to correspond to Port T3 as the PWM output. For controlling the speed, as mentioned, we would adjust the duty cycle: a lower duty cycle corresponding to a lower speed and a higher duty cycle for a higher speed. We could write in values up to 150 (the total period of the PWM), where at 150 we would be driving the motor at maximum speed (About 5000rpm). Since we only needed 3 different speeds (High, medium, and low), we stored 3 values for 75, 62.5 and 50% in an array called *refvalues*[], indicating that these were the values of reference we were using. By using the 2 switches as a two-bit binary number, we made the switch settings (called *swstate* in the program—see the attached source code) the index of the *refvalues* array.

The motor speed input

The speed display output was created using 11 different output pins: PT0-PT2, PT5-PT7, and PTAD3-PTAD7. The display consists of 3 elements: two hex-to-decimal 7-segement displays for the first two digits of the rpm speed, and an 3 digit array of leds. We first needed to derive a method of determining the rpm speed. We decided to use our ADC value (obtained from the motor speed input as described in the previous paragraph) to derive the rpm speed with the following equations:

$$rpmspeed = \frac{rotations}{minute} = frequency\left(\frac{cycles}{second}\right)\left(\frac{60\ seconds}{minute}\right)\left(\frac{rotation}{8\ cycles}\right)$$

$$rpmspeed = frequency\ x\ 7.5$$

By dividing the rpm speed by our ADC value for each switch setting and then taking an average of these slopes, we calculate about 2.41. Then we are able to get a constant scalar of the ADC value of 2.41*7.5=18.1.

$$rpmspeed \approx ADC\ x\ 18.1$$

Although this method is said to be a little less accurate than say for instance a comparator circuit, we found it to be very accurate when testing the motor frequency and calculating the speed directly. The only variation seemed to be since every ADC value would cause an rpm speed jump of 18 rpm, which was still fine since we only needed to display the first two digits of the speed.

After calculating the speed it was necessary to convert these values into displayable forms. For the first two digits we were able to divide the rpm speed by ten until we had a number less than 10; this was the first digit. By taking the number of times we divided by ten to get the first digit and

subtracting one iteration, we then had the first two digits—from these two digits, we subtracted 10 times the first digit and were able to have just the second digit by itself. The number of times we divided by ten the first time was the power of ten needed for the scientific notation. These values were then easily sent to our display using bit-shifting. It is also worth mentioning that we utilized a timer overflow interrupt that would occur every 1ms to allow our display to update every 250ms by using a decremented counter of 250 (aptly dubbed *counter*).

The final module of our program was the compensator. By taking samples of our open-loop response, we were able to construct a transfer function of our plant using MATLAB. We determined by simulation is MATLAB that an improved steady-state response, a faster rise time, and an overall more stable system could be created using a PID compensator. To compensate, it is necessary to calculate an error from the feedback loop. We took our reference value and subtracted to obtained sample from the ATD input from the motor tachometer. The P (or Proportional) compensator is simply the product of a scalar gain quantity and the error term. The I (or Integrator) compensator is a summation of the error over all time of the motor operation multiplied by the sample time and then scaled by another gain factor. The D (or Differentiator) compensator is the average of the current error and the previous error multiplied by the sample time and then multiplied by yet another gain factor.

The sample time was a little more difficult to calculate, so we decided that a more accurate measure could be employed by utilizing our 1ms timer interrupt. We used the onboard ATD system to average 8 samples every time the 1ms interrupt occurred, which effectively forces our sample time to be approximately 1ms (As accurate as the interrupt time is to 1ms). The gain factors for the P-I-D compensators were determined mostly by trial and error to see what combination would get the best performance.

Problems

Many problems were encountered while creating our project, and in several instances, their solutions eluded us for quite a while. To start out with, we found that if we increased our Kp, Ki, and Kd values enough to achieve good performance, our program would malfunction when switching from high to low speeds. Under these circumstances, the motor would increase to full speed (100% duty cycle), and it would not return to any other speed setting without turning it off and re-initializing our code. This problem baffled us for a long time because its cause was not easy to identify. After extensively tweaking and testing portions of our code, we discovered that under certain circumstances, certain variables such as Gc and Ierror (see attached code at the end of this report) would increase rapidly to some huge value, and not come down. This is why re-initializing our code was necessary to correct this problem, since in doing so all variables were reset. It turns out that having a large Kp or Ki value coupled with a large error value would result in an overflow. This was corrected by instituting a saturation block. In doing so, not only could we switch speeds without problem, our results were considerably improved. For example, overshoot was reduced and our closed loop design was noticeably faster than open loop.

Another problem we faced was that starting from rest, our motor occasionally experienced a slight delay in reaching its set speed. After running through our program step-by-step, we found that our Ierror variable, which acted as running tally of recent errors, had to deplete itself before it was able to increase. As it goes, we had written our program so that by turning the switches off, the program would automatically set the motor speed to 0 instead of calculating that it should be 0. This meant that the next time the motor speed was increased, our program had to deplete the

previous values before it could likewise increase. We removed this pre-set value and from then on out, our program never delayed between any switch settings.

There were many minor problems encountered that we corrected along the way as well. These included using an interrupt and timer to regulate our compensator instead of a counter, changing our compensator code to correctly implement derivative control by taking into account the present error in relation to the previous one, and repeatedly altering our Kp, Ki, and Kd values (Kp=1, Ki=1 and Kd=5 performed best).

Conclusions

In completing our project, we look back at the work we have done and appreciate all of the things we have learned along the way. Not only have we learned more about the hardware and software used, but many tools as well, such as Excel, MATLAB, and the oscilloscope. More importantly, we learned many practical things about problem solving and engineering in general. This includes avoiding assumptions and checking all parts of a design to ensure it is working as expected. For example, we assumed our controller worked correctly since the math worked out on paper, but in actuality the computer calculated things differently than we did. So, by going through our program line-by-line, we were able to determine the disparities and correct them. All of this will serve us well down the road since in the real world, we will be designing, building, testing, and debugging things just like we did during our project. Furthermore, these kinds of problem solving skills are useful in all spheres of life. Finally, by working together we learned how to better collaborate and developed a greater appreciation of the importance of teamwork. With regards to our project, while there are several applications for it, such as cruise control, there are even more with respect to the controller itself, such as mine detecting robots and magnetic signature management on “stealth” ships.

The following is our final code:

```
// EE Lab IV – Final Project Code
#include <hidef.h> // common defines and macros
#include <MC9S12C32.h>  // derivative information
#pragma LINK_INFO DERIVATIVE "MC9S12C32"
#define TOTALATD
(ATDDR0L+ATDDR1L+ATDDR2L+ATDDR3L+ATDDR4L+ATDDR5L+ATDDR6L+ATDDR7L)
#define DS_1    150
#define DS_2                    1000 //Number of clocks for 1ms at 1MHz

char bit0, bit1, bit2, swstate, Kp, Ki, Kd, flag;
unsigned char sample, a;
int error, preverror, refvalues[4], i, rpmspeed, Gp, Gd, Gi, Gc, sec, counter;
signed long Ierror, Derror, temp;

void interrupt timer_ISR(void){  //Vector 8
//Update the display at 250ms intervals
            unsigned int f, rpmtemp, tenpower, g, msb1, msb2;
            f=0; rpmtemp=0; tenpower=0; g=0; msb1=0; msb2=0;

  if(counter==0){ //Update display after 250 ms
    sec++;
    counter=250;
              rpmtemp=rpmspeed; //current value
              if(rpmtemp>500){
              for(f=0;rpmtemp>1;f++){
                        rpmtemp=rpmtemp/10;  //shift decimal to the left
            }
            tenpower=f-1;  //number of leds lit
            g=tenpower;
            rpmtemp=rpmspeed;
            for(g;g>0;g--){
                        rpmtemp=rpmtemp/10;
            }
            msb1=rpmtemp;
            g=tenpower-1;
            rpmtemp=rpmspeed;
            for(g;g>0;g--){
                        rpmtemp=rpmtemp/10;
            }
            msb2=rpmtemp-(10*msb1);

            //Output to Ports
            //3 digit binary
            if(tenpower==0){
              PTT_PTT0=0;
            PTT_PTT1=0;
            PTT_PTT2=0;
            }
            else if(tenpower==1){
                        PTT_PTT0=1;
            PTT_PTT1=0;
            PTT_PTT2=0;
            }
```

```
            else if(tenpower==2){
                    PTT_PTT0=1;
            PTT_PTT1=1;
            PTT_PTT2=0;
            }
            else if(tenpower==3){
                    PTT_PTT0=1;
            PTT_PTT1=1;
            PTT_PTT2=1;
            }

        //First digit of HEX display
            PTT_PTT5=(msb1 & 0x08)>>3;
            PTT_PTT6=(msb1 & 0x04)>>2;
            PTT_PTT7=(msb1 & 0x02)>>1;
            PTAD_PTAD3=msb1 & 0x01;

        //Second digit of HEX display
            PTAD_PTAD4=(msb2 & 0x08)>>3;
            PTAD_PTAD5=(msb2 & 0x04)>>2;
            PTAD_PTAD6=(msb2 & 0x02)>>1;
            PTAD_PTAD7=msb2 & 0x01;
                    //update_display(rpmspeed);
            }
            else{ //For the 0 rpm case
                    PTT_PTT0=0;
                    PTT_PTT1=0;
                    PTT_PTT2=0;
                    PTT_PTT5=0;
                    PTT_PTT6=0;
                    PTT_PTT7=0;
                    PTAD_PTAD3=0;
                    PTAD_PTAD4=0;
                    PTAD_PTAD5=0;
                    PTAD_PTAD6=0;
                    PTAD_PTAD7=0;
             }
  }
  else{
            --counter;
            /* Initiates conversion of pin AN1 and return converted data */
            ATDCTL5 = 0x81; // write to ATDCTL5 starts conversion
    while (ATDSTAT0_SCF == 0); // wait for conversion sequence complete
    sample=(TOTALATD/8); // return average value from ATDDR's
  }

            TC0+=DS_2; //Set-up TC0
            TFLG1_C0F=1;//Latch needs 1 to reset
            TIE_C0I=0; //Disable the interrupt
            flag=1;
}

void init_ATD() {
char i;
/* Power up the ATD module */
ATDCTL2_ADPU = 1; //Activate power up
```

```
ATDCTL2_AFFC = 0; //Normal flag clear (default)
ATDCTL2_AWAI = 0; //Run in WAIT mode (default)
ATDCTL2_ETRIGE = 0; //No external trigger (default)
ATDCTL2_ASCIE = 0; //No interrupts (default)
/* Short delay (> 20us) for power up */
for (i = 0; i < 20; i++);
/* Set up ATDCTL3 for conversion sequence */
ATDCTL3_S8C = 0; //one conversion per sequence
ATDCTL3_S4C = 0;
ATDCTL3_S2C = 0;
ATDCTL3_S1C = 0;
ATDCTL3_FIFO = 0; //result in consecutive registers (default)
ATDCTL3_FRZ = 0; //continue in freeze mode (default)
/* Set up ATDCTL4 for conversion timing */
ATDCTL4_SRES8 = 1; //8-bit resolution (0 = 10-bit)
ATDCTL4_SMP = 3; //16-clock (max) sample time for accuracy
ATDCTL4_PRS = 1; //prescale fbus(4MHz) by 4 for range [2MHZ..500KHz]
}

void main(void){
PTT=0; DDRT=0xE7; DDRAD=0xF8; bit0=0; bit1=0; bit2=0; swstate=0; i=0;
Gp=0; Gi=0; Gc=0; Gd=0; Kp=1; Ki=1; Kd=5; Derror=0; Ierror=0; error=0; temp=0; sec=0;
refvalues[0]=0;
refvalues[1]=128;//75;//128; //50%
refvalues[3]=159;//94;//159; //62.5%
refvalues[2]=191;//113;//191; //75%

   //*****Timer Setup
TIOS_IOS0=1; //Enable Output compare channel 1
TFLG1_C0F=0; //Clear C0F flag
TSCR2_TOI=1; //enable timer overflow interrupt
counter=250;
TC0=TCNT; //Set-up TC0 - Now have 8ms to set up system without an interrupt
TIE_C0I=1; //Enable the interrupt

   //*****PWM Setup
   PWMPRCLK=0x20; //Prescale of 16 [first 3 LSB 000 to 111 is 0,2,4,8,16,32,64,128]
   PWMSCLA = 0x04;//don't care, since using A and not SA
   PWMPOL = 0x08; //Polarity - Begin high
   PWMPER3=DS_1;
   PWMDTY3=25*swstate;
   PWME_PWME3=1; //PWM enable for channel 0
   MODRR=0x08; //MODRR bit k controls bit k of Port T
   init_ATD(); // set up the ATD module
   TSCR2 = 2;                    //SET BUS Clock prescaler to 4 /4= (1MHz)
   TSCR1_TEN = 1 ;                // ENABLE the timer
   EnableInterrupts ;            // ENABLE global interrupts

          while(1){ // infinite loop for the led counter
   bit0=PORTA & 0x01; //SW1=PA0
   bit1=(PORTB & 0x10)>>3; //SW2=PB4
                  swstate = bit1 | bit0;
                  error=refvalues[swstate]-sample;
                  Gp=Kp*error;
                  if(swstate==0 && Ierror<0){
                         Ierror=0;
```

```
            }
            else{
              Ierror+=error;
            }
            temp=Ki*Ierror;
            Gi=temp/1000;
            Derror=(preverror-error)/2;
            temp=Kd*Derror;
            Gd=temp/1000;
            Gc=Gp+Gi+Gd;
            preverror=error;
            temp=0;
            if(Gc>150){
              Gc=150;
            }
            else if(Gc<0){
              Gc=0;
            }
            //PWMDTY3=Gc;

        //Open loop case
        PWMDTY3=refvalues[swstate];

            rpmspeed=(sample*18)+(sample/10);
            if(flag==1){
              TFLG1_C0F=0; //Clear C0F flag
              TIE_C0I=1; //Enable the interrupt
              flag=0;
            }
        }
} //End main
```

Figure 1: Circuit Schematic

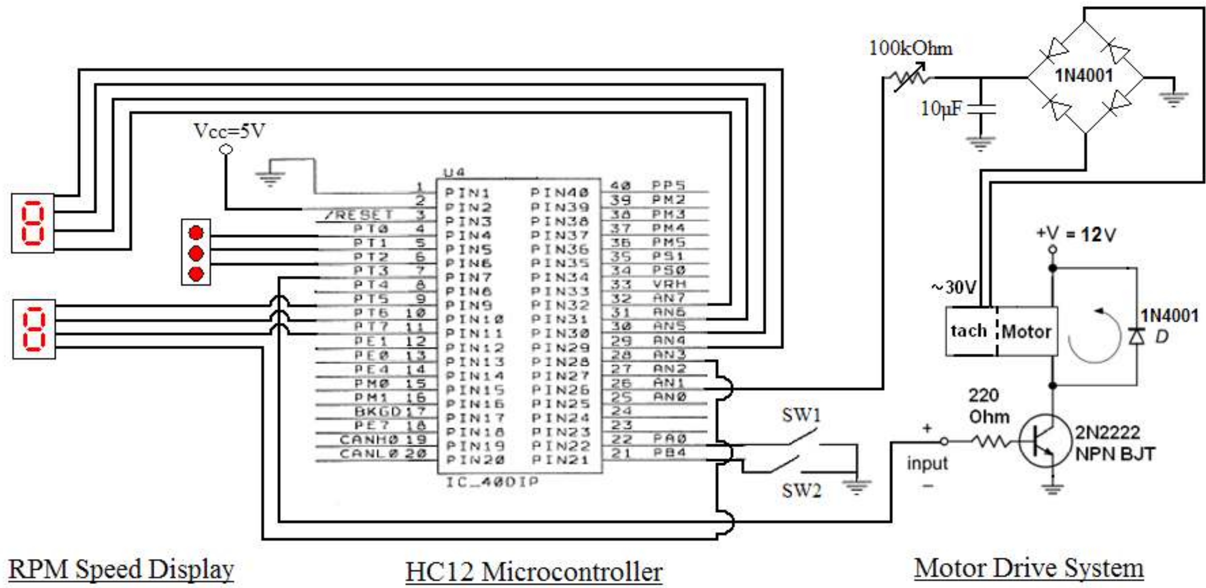

Figure 2: Software Flow Chart

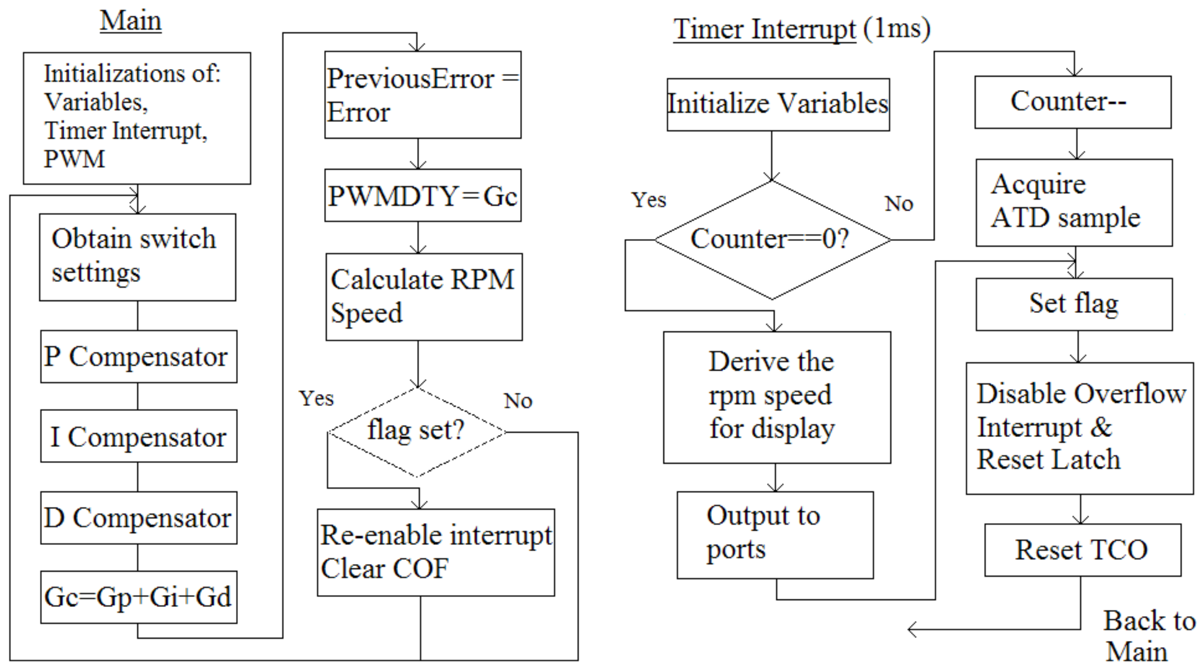

Figure 3: Hardware Cost

Hardware	Description	Quantity	Unit Cost
Dragonfly	Microcontroller Development Board	1	$18
Breadboard	Jameco BreadBoard	1	$5.37
Potentiometer	100KOhm, 1/2W	1	$0.83
Diode	1N4001, 50V, 1A	5	$0.10
Capacitor	10microF, 50V	1	$0.21
Resistor	220Ohm, 5%, 1/4W	1	$0.05
Transistor	2N2222A NPN BJT	1	$0.41
Total Cost			**$25.37**

Note: All prices obtained from the AU Chemical Supply Store, except the Dragonfly, which can be found here: http://www.evbplus.com/c32_modules/DIP40.html

Figure 4: Oscilloscope Output of Low, Medium, and High Speeds

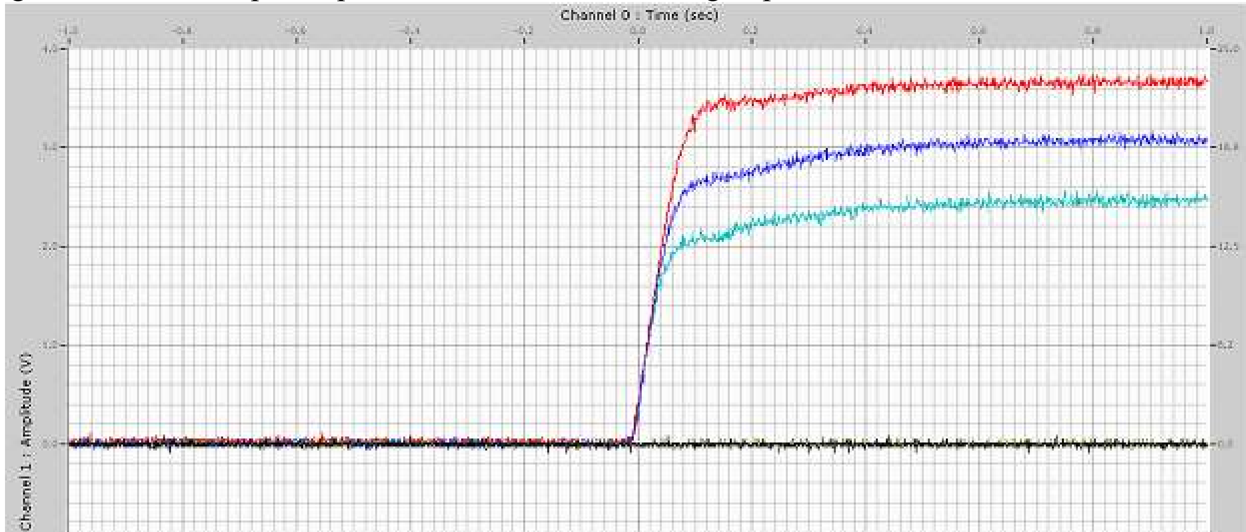

Figure 5: Overshoot as calculated by the oscilloscope

Speed	Overshoot
Low	0.2941 %
Medium	0.2703 %
High	1.8595 %

Figure 6: Oscilloscope output of closed loop vs. open loop for low speed (closed loop in light blue)
Time in seconds vs. Voltage amplitude

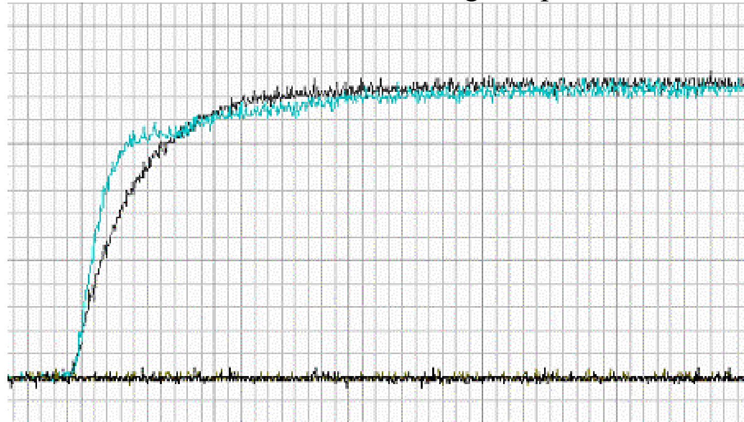

Figure 7: Oscilloscope output of closed loop vs. open loop for medium speed (closed loop in blue)
Time in seconds vs. Voltage amplitude

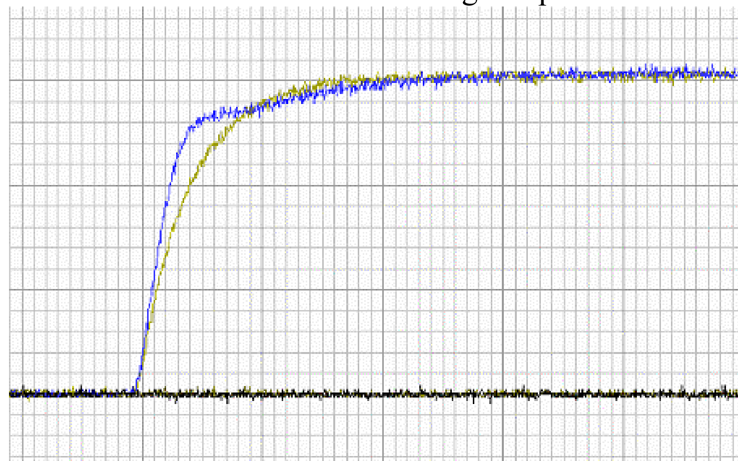

Figure 8: Oscilloscope output of closed vs. open loop for high speed (closed loop in red)
Time in seconds vs. Voltage amplitude