**ECEN301 Assignment 3: Motor Control**

**Abstract**

This assignment develops on laboratory session 5, 6 and 7 to allow the control of a DC motor using PWM. Using an I/O module, a user can input a desired motor speed. Two PCA modules, module 0 and 1 are used for the PWM output and capture of the optical switch interrupt to determine the current motor speed and the error between the target speed and the current motor speed. To control the motor speed, a PID controller is implemented. Through tuning of the constants, the final values were KP = 0.2, KI = 0.1 and KD = 0.3. This resulted in the motor to show some overshoot for a 60 rpm step input, with a steady sate error of essentially 0%.

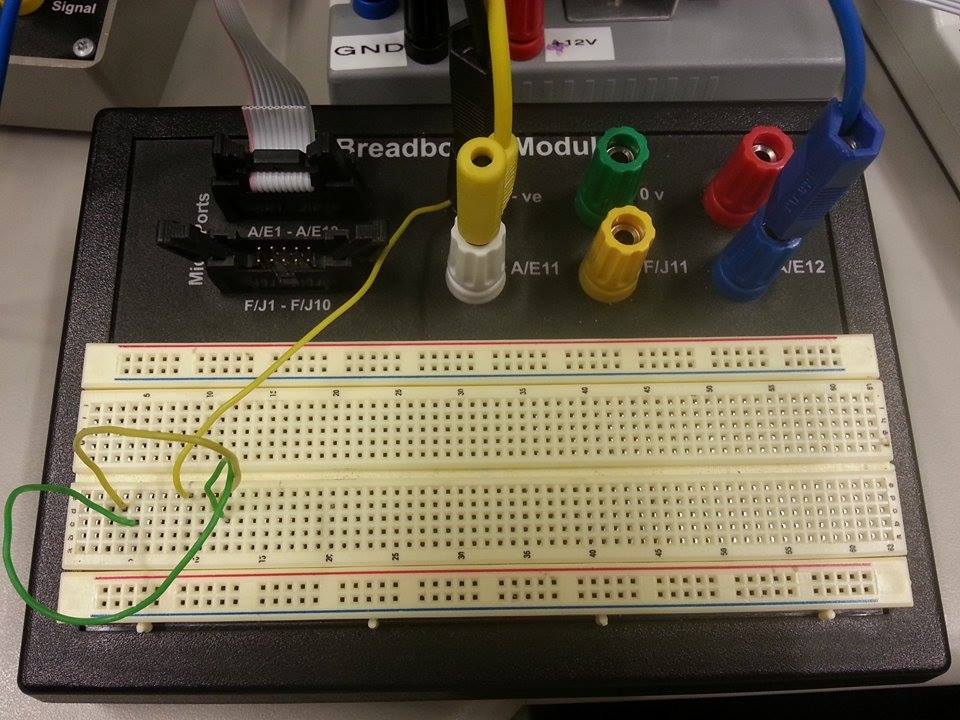
**Hardware Setup**

**Equipment Used**

* Breadboard
* I/O module
* LCD display and keypad
* AT89C51AC3 Microcontroller
* DC Motor
* Power Supply Unit

AT89C51AC3

In port 1, pins 3 and 4, are the PWM output and interrupt capture for module 0 and 1 respectively. Port 1 is connected to the A/E1 – A/E10 port of the breadboard module. A wire is passed from row 4 of the breadboard to row 11. This outputs the PWM signal towards the DC motor module signal input. This allows the motor speed to be controlled by PWM. Another wire is passed from row 5 to row 12. Row 12 contains the optical switch signal from the motor, which is fed back to the pin 4 of port 1 to allow the interrupt flag to be called at a negative edge.



Breadboard wiring

The LCD display has two ports, data (8 lines) and control (3 lines). These are plugged into port 0 and port 4 respectively into the microcontroller.

­Connected to port 2 of the microcontroller is the I/O module. A ribbon cable is connected to the “Digital Out” port of the module, and the switch at the top of the module is flipped to the “Ext” setting to allow input into the module to be read from microcontroller.

The wiring for the motor is shown below:

From “motor out” on motor module

Wiring for motor

+15V

Ground

To A/E12

Ground

**The Control System**

**Calibration between HEX value from I/O and RPM Module**

To get the motor working by pulse width modulation, An I/O module is connected to port 2 of the microcontroller. To set the RPM of the motor, a relationship between the input into the I/O module and output RPM of the motor needs to be determined. This would allow the user to input a desired motor speed into the system.

To obtain the relationship, RPM is measured for each input into the I/O module. 27 samples were taken, in increments of ten and including the RPM value with an input of 255. Each input must be first converted to binary to determine the appropriate bits that needs to be on.

In figure 1 below, is the plot between the input vs the RPM on the motor.

**Figure 1:** Plot between input into I/O module and motor speed (RPM)

Using excel, the relationship between the input and the motor speed is determined to be:

(1)

From this, the input of the I/O module can be used to set the speed of the motor and also allows the user to change the motor speed.

**Implementing of pulse width modulation (PWM)**

PCA module 0 is set as the output of the PWM to the motor. This is done by setting the CCAPM0 module as 0x42. This enables the ECOM0 and PWM0 bit. The ECOM0 enables the compare function which is needed to implement PWM. PWM0 bit is enabled to configure module 0 as an 8bit PWM, outputted on pin 3 of port 1.

**Implementation of dot matrix display**

The dot matrix display allows live information being fed back to the user. The interface displays the current speed of the motor, target motor speed and the difference between the target motor speed and the current motor speed. Functions from the phys340 library are used to initialise the LCD, clear the LCD, set the cursor on the LCD and display information. These functions are *initLCD(), clearLCD(), setLCD() and writeLineLCD()* respectively.

**Interrupt Handler**

The interrupt handler uses the interrupt address of 003H, which corresponds to an interrupt number of 6. Within the interrupt handler, all interrupts are turned off. Firstly, the CF bit is checked to determine if the PCA counter has been overflowed. If it has, the overflow variable is incremented by 1, and the flag is cleared.

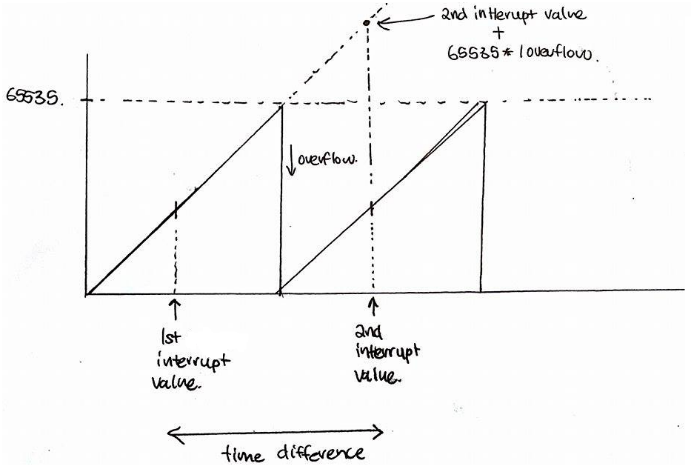
The CCF1 bit is then checked to determine if a hole has passed. If it is set to true, the value in the CCAP1H/L registers are stored. This value correspond how high the PCA counter counted up until the interrupt occurred. From this, the difference between the previous values in these registers can be determined.

At the end, the CCF1 bit is then cleared and interrupts are re-enabled.

**Measuring motor speed in RPM.**

On the DC motor contains a disk with slits. The rate of change of the slits can be determined by an optical switch on board the motor which in turn can be used to determine the RPM of the motor.

Knowing the PCA clock frequency of 1 MHz, the time for the PCA counter to count 1 bit is the inverse of the PCA clock frequency which is 1μs. The difference between the current value stored in the CCAP1H and CCAP1L and the previous value stored in these registers gives us the count difference between the two interrupt events. However, overflows could occur during that time, so the current value stored must be added to the number of overflows that has occurred between the last interrupt multiplied by the 65535. In figure 2 below is a visualisation of determining the difference between the CCAP1H/L values between two interrupts.



**Figure 2:** Visual aid to determine the time difference between two interrupts

To add the two 8 bit values in the CCAP1H and the CCAP1L registers, a bit shift right is done on the value stored in the CCAP1H, and is added to the value stored in CCAP1L. By subtracting the current value stored with the previous value allows us to get the count difference. This difference can be divided by the clock frequency to get the number of seconds that has occurred between the two interrupts. Since the encoder disk has 200 holes per revolution, multiplying the time between the interrupt with 200 holes per revolution gives the time for one revolution. To get the motor speed in terms of RPM, 60 dividing by the time for one revolution is done. This process is simplified with the following equation:

(2)

Where difference is the current value on the CCAP1H/L subtracted from the previous value on the CCAP1H/L register

**Averaging**

Implementation of averaging consists of using an array of size 5. 5 samples of consecutive RPM values are stored into an array. Once this array is full, a variable called *fullFlag* is set to high. The average of these values can be determined by the total value in the array, divided by 5.

After the average is calculated, the current RPM is set as this average, and from this, the error RPM can be determined from the user specified RPM value.

Another method of averaging that could be implemented is a weighted moving average. Instead of averaging the motor speed every 5 samples, a moving average updates the motor speed for every new sample. Values of RPM would be pushed back to the array, and so those values that are in the past might not truly represent the current motor speed accurately. To solve this would require the implementation of weighting to each values of motor speed, where more current values would be weighted more by using a larger constant and older samples would use smaller constants.

**Proportional, integral controller (PID)**

A PID controller is implemented to control the motor speed set by the user through the I/O module. It comprises of three parameters, called the proportional, the integral and derivative values.

**Proportional Controller**

The proportional depends on the present error, the integral on the sum of past errors and derivative is a prediction of future errors.

The size of the controller output is proportional to the size of the present error. The proportional controller has an inherent steady state error. Below in table 1 shows the steady state error for each KP value with a 200 RPM step input. The average steady state for each values of KP is taken from 14 values streamed to the hyper terminal through serial.

|  |  |  |
| --- | --- | --- |
| **Kp** | **Steady State (RPM)** | **Steady State Error (%)** |
| 0.1 | 25 | 87.57% |
| 0.2 | 44 | 78.2% |
| 0.3 | 61 | 69.73% |
| 0.4 | 74 | 62.83% |
| 0.5 | 87 | 56.57% |
| 0.6 | 98 | 50.83% |
| 0.7 | 108 | 45.87% |
| 0.8 | 117 | 47.73% |
| 0.9 | 123 | 38.1% |
| 1.0 | 130 | 35.03% |
| 1.2 | 143 | 28.47% |
| 1.4 | 154 | 22.83% |

**Table 1:** Steady State Error for a step input of 200 RPM

From table 1, increasing the proportional constant decreases the steady state error. However, setting the KP higher than 1.4 resulted in the system being unstable. This seen as the encoder signal fluctuating without going to some steady value.

**Integral Controller**

Taking KP as 0.2, the integral controller is implemented next. Figures 3, 4 and 5 below shows the response of the system for KI values of 0.75, 0.5 and 0.2 with a 60 RPM unit step input into the system. Again, the RPM values are taken through the hyper terminal.

**Figure 3:** Response of system with Kp=0.2 and KI= 0.75

**Figure 4:** Response of system with Kp=0.2 and KI= 0.5

**Figure 5:** Response of system with Kp=0.2 and KI= 0.1

Comparing with the response of the proportional controller, implementing the PI controller results in the steady state error to be effectively zero from the plots.

From By decreasing the value of KI, it is seen that the settling time decreases. Also the number of oscillations decreases as KI decreases.

**Derivative Controller**

Using the same KP value of 0.2, and choosing a KI ­value of 0.1, due to the smallest number of oscillations, a derivative controller is the final controller needed to complete the PID controller for the motor. The responses for in figure 6, 7 and 8 are displayed below.

**Figure 6:** Response of system with KP = 0.2 KI = 0.1 KD = 1

**Figure 7** Response of system with KP = 0.2 KI = 0.1 KD = 0.5

**Figure 8:** Response of system with KP = 0.2 KI = 0.1 KD = 3

For the final parameters for the PID controller is set to KP = 0.2 KI = 0.1 KD = 3 to as it has the fastest settling when comparing figure 8 with figure 6 and 7. Also it has the most minimal overshoot.

To make the system critically damped, KP can be decreased the overshoot. Once a KP value has been determined, the KI value can be further decrease to minimise the overshoot, while at the same time eliminate the steady state. From there, the KD ­can be adjusted to further minimise more overshoot until the response curve has characteristic of a critically damped system.

**Conclusion**

At the end of the assignment, the motor speed was able to be controlled using a PID controller with constants of KP = 0.2 KI = 0.1 KD = 3. A user is able to set the motor speed using the I/O module, and the motor would converge to that value with essentially a steady state error of 0%. However, the step responses of the system could be further improved to make the system critically damped by changing the PID constants. Application for this motor controller could be used in car cruise control. However, the response of the system must be critically damped for the driver to not experience any discomfort.

**Appendix**

#include <t89c51ac3.h>

#include <string.h>

#include <phys340libkeil.h>

#include <stdio.h>

char outputText [33];//for printing out onto LCD or hyperterm

int rpm; //current averaged RPM of motor

int difference=0; //difference between values in CCAP1H/L in two interrupts

unsigned int hi,lo,prev,curr,period; //varaibles for cacluating RPM

unsigned int overflow; // overflow flag counter Flag counter

int rpm; //average RPM

int currRPM = 0; //current RPM

int samples[5]; //for averaging, storage for RPM

int index = 0; //for averaging

int summer = 0; //to calculate the total of the array of RPM

int fullFlag = 0; //flag set when array is full of values

int i = 0; //for loop increment variable

int wantRPM = 0; //target set RPM

int err = 0; //error between set target RPM and current RPM

float kp = 1.5; //proportional constant

float ki = 0; //integral constant

float kd = 0; //derivative constant

float prop = 0; //proportional control

float intCtrl = 0; //integral control

float derCtrl = 0; //derivative control

float correctionTot = 0;

int errSum = 0; //sum of past errors

int oldErr = 0; //rate of change of error

int errDif = 0;//present error

void MyIntHandler(void) interrupt 6

{

EA=0;

EC=0;

if(CF ==1)

{

overflow++;

CF=0;

}

if(CCF1 == 1)

{

hi = (unsigned int)(CCAP1H<<8);

lo = (unsigned int)(CCAP1L);

prev = curr;

curr = hi + lo + ((unsigned int)(65536\*overflow));

difference = curr-prev;

}

CCF1 = 0;

EA=1;

EC=1;

overflow=0;

}

void averaging()

{

summer = 0;

//getting the total within the array

for(i = 0; i < 5; ++i)

{

summer = summer + samples[i];

}

//finding the average

rpm = summer/5;

}

void init()

{

//PCA module 0 as PWM output to motor

CCAPM0 = 0x42;

CMOD = 0x01;

//PCA module 1 as capture for optoswitch

EA=1;

EC=1;

CCAPM1 = 0x11; //Capture mode and enable CCFx interrupt bit

CCON = 0x40; //PCA Counter On

CF=0;

}

void main()

{

init(); //initialise function registers

initLCD(); //initialise LCD

initSerial(1200); //initialise serial communication with 1200 baud rate

while(1)

{

//PID control

prop = 0.2\*err; //proportional controller

intCtrl = 0.1\*errSum; //integral controller

derCtrl = 0.25\*errDif; //differential controller

correctionTot = prop+intCtrl+derCtrl; //sum of the P,I and D

//PWM out

CCAP0H = (393.05-correctionTot)/1.5924; //setting the PWM across the motor

clearLCD();

currRPM = 300000/difference; //calcualting the currrent RPM

samples[index] = currRPM; //storing current RPM into array

index++;//increment array cursor

if (index == 6) //array of samples is full

{

index = 0; //reset array cursor

fullFlag = 1;//array is now full

}

if (fullFlag)

{

averaging();

//prints out current RPM

sprintf(outputText, "%i",rpm);

writeLineLCD(outputText);

writeLineSerial(outputText);

//prints out wanted RPM

setLCDPos(16);

wantRPM = (-1.5924\*P2)+393.05;

sprintf(outputText, "wRPM:%i",wantRPM);

writeLineLCD(outputText);

//prints out error

oldErr = err;

err = wantRPM - rpm; //gets the present error difference

errDif = oldErr - err; //rate of change of error

errSum = errSum+err; //sum of all past erros

setLCDPos(25);

sprintf(outputText, "e:%i",err);

writeLineLCD(outputText);

fullFlag = 0;

}

}

}