Contents

Introduction

The project segment titled "Closed Loop Motor Control" involves developing a motor control system capable of functioning in both open and closed loop configurations. In open loop operation, the motor's speed is regulated by varying the PWM duty cycle, with the motor speed correlating to the set duty value. For closed loop control, a PI controller maintains the motor's speed at a steady RPM, with the target motor speed in RPM being the input parameter. The realization of this control system incorporates the utilization of various pins, functions, and peripherals of the PIC16F46K22, alongside additional components and devices. This report aims to comprehensively detail every facet of the motor design to a degree that enables a team of engineers to faithfully execute or construct the design. It also encompasses the documentation of completed tests and their corresponding outcomes.

Block Diagram

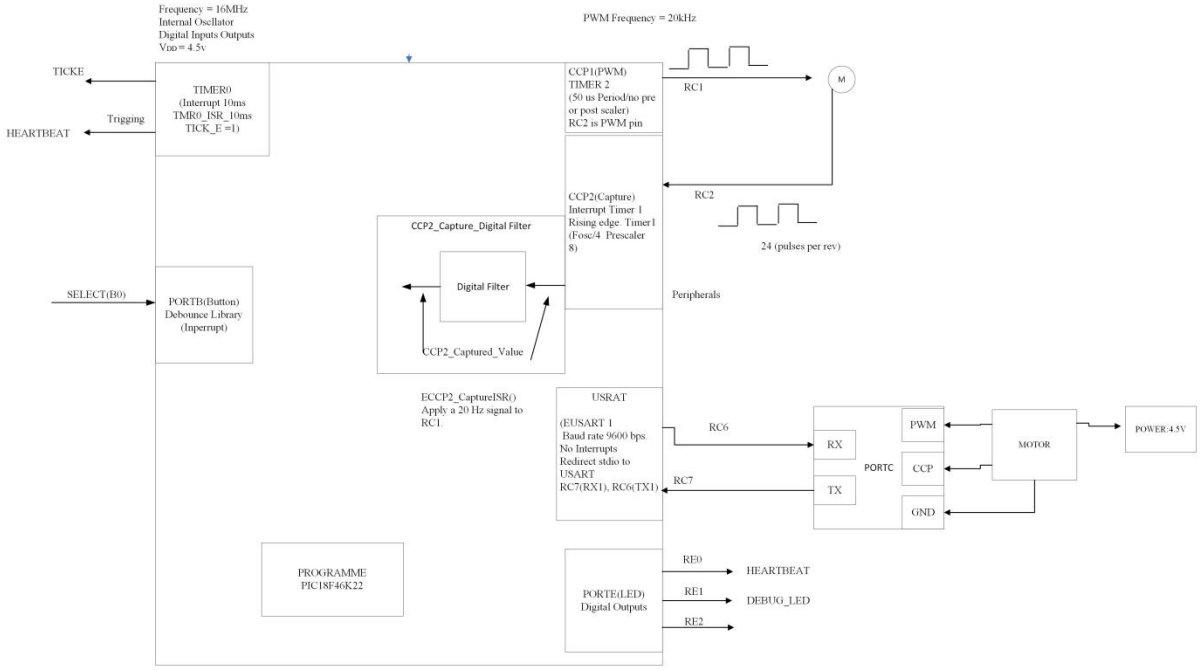

Figure 1: Closed Loop Motor Control Block Diagram

The figure1 shows the block diagram of the motor control project, showing the various pins, interrupts, peripherals, external circuitry and components used.

Detailed Schematic

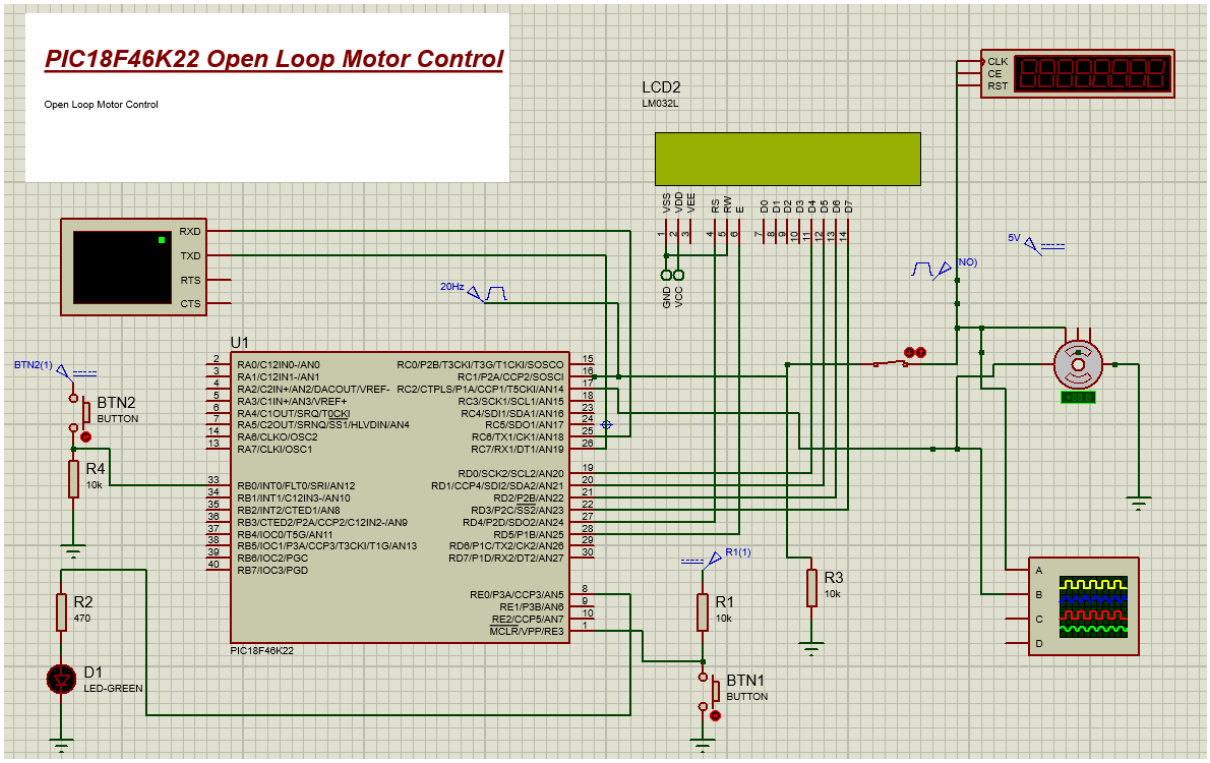

Figure 2: Motor Simulation Schematic

The figure2 shows a schematic of the motor control system, along with the buttons, leds, and motor that are connected to various pins of the PIC16F46K22. The LCD screen, oscilloscope, and counter on the top right are not used in this motor project, but the rest of the components are.

System configuration

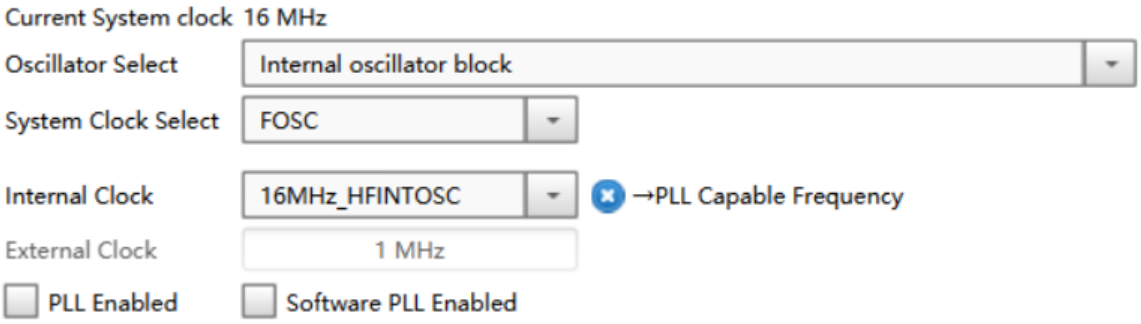

Figure 3: System Configuration

System configuration include: internal oscillator, FOSC, 16MHz, digital inputs (PORTB) and outports (PORTE).

Peripherals Used and Configurations

The various peripherals used in this motor control system and their relevant configurations include:

ECCP 1 (PWM) : Enhanced PWM. Timer 2 (50 us Period/no pre or post scaler). RC2 is PWM pin.

ECCP2 2(Capture) : Interrupt Timer 1. Rising edge. Timer1 (Fosc/4 Prescaler 8)

EUSART1: Asynchronous mode. Baud rate: 9600. No Interrupts. Redirect stdio to USART. RC7(RX1), RC6(TX1)

TIMER0: Clock source is FOSC/4 (Interrupt 10ms TMR0_ISR_10ms TICK_E =1)

Overview of software tasks performed and the use of interrupts

The various software tasks performed by this motor control system include:

1) Initialising the peripherals with a function
2) Declaring states and variables
3) FLASH LED
4) Transitioning between different states of the motor control{SETUP_S, OPEN_S, CLOSED_S } States_T
5) Printing texts and messages using printf() command
6) Scanning in values using scanf() command
7) Using counting functions using variable i
8) Calculating values such as frequency and RPM
9) Loading values into the motor with EPWM1_LoadDutyValue() command
10) Loading sets of values into the PI controller with Controller_Func_PI () command

The interrupts used in this system include:

11) Timer0 interrupt for TICK_E event: This interrupt occurs every 10ms so that a delay function can be implemented in the motor control system. For example, 100 TICK_E interrupt means that 1 second has passed.
12) Button interrupt for SELECT_F event: This interrupt is on Portb.0 connected to a button so that when the button is pressed, an interrupt is generated which brings the motor control program from the open or closed loop state back to the setup state.

State machine diagram

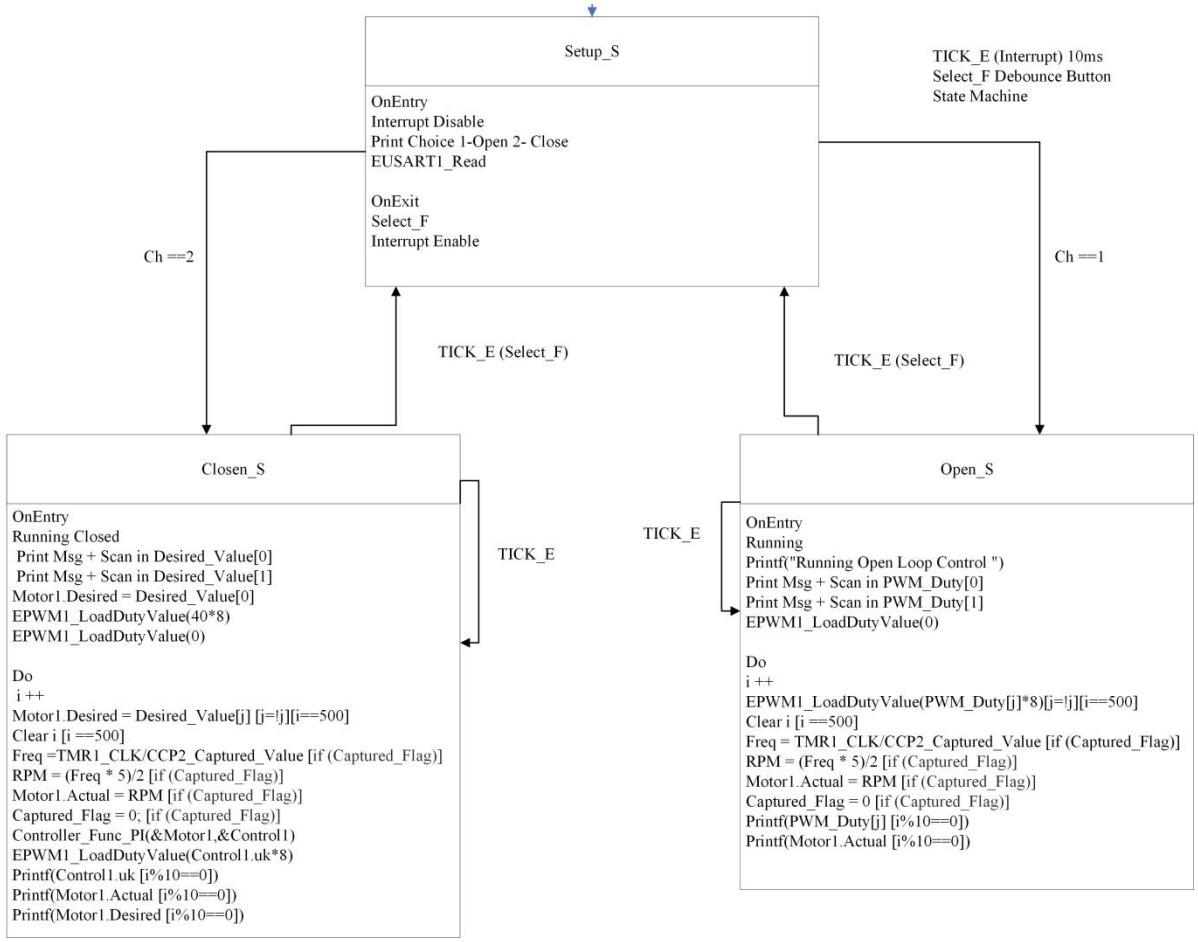

Figure 4: Motor Control State Machine Diagram

The Figure3 above shows the state machine diagram of the motor control system which describes what events would allow the motor to transition between states and what the motor would do upon entering each state, as well as what the motor would do continuously as it stays in one state.

Function call graph

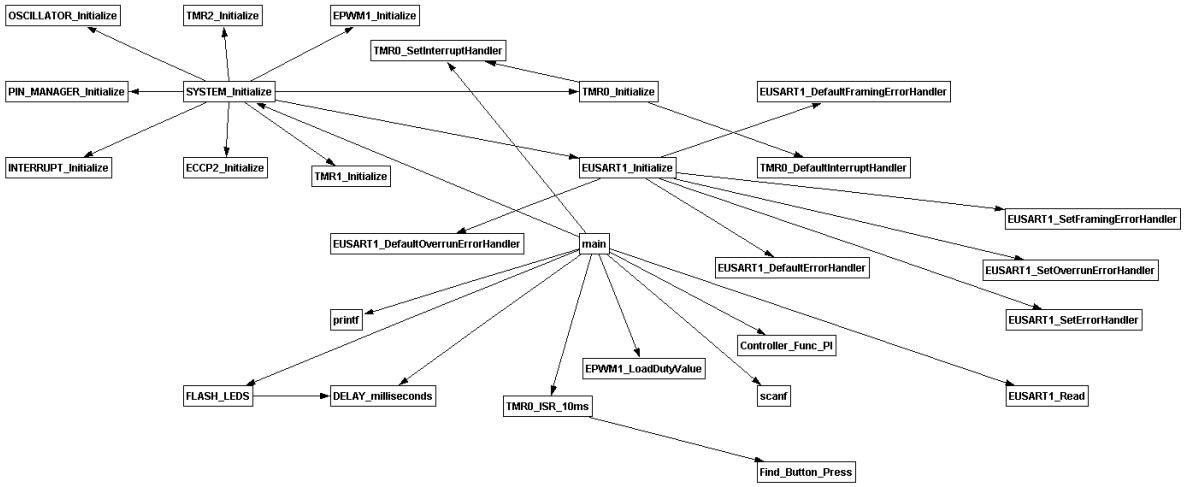

Figure 5: Function Call Graph

The figure above shows the functions called by the motor control program.

main() Function:

This is the entry point of the program. It initializes the system, configures interrupts, and enters a loop where it handles the motor state machine and user inputs.

Initialization:

SYSTEM_Initialize() likely calls several other initialization functions such as OSCILLATOR_Initialize, TMR0_Initialize, TMR1_Initialize, CCP2_Initialize, EUSART1_Initialize etc.

Interrupt Service Routines (ISR):

TMR0_ISR_10ms(): This function is set as the interrupt handler for the Timer0 interrupts and is responsible for toggling the heartbeat LED, checking button presses, and maintaining a count.

Find_Button_Press(): Called within TMR0_ISR_10ms(), relates to debouncing and reading button states.

Peripheral Handling:

FLASH_LEDS(): Directly controls the LEDs based on the given duration, which toggles the state of the LEDs on PORTE.

EUSART1_Read(): Reads data from USART1, which is typically initialized in the EUSART1_Initialize() function.

PWM and Motor Control:

EPWM1_LoadDutyValue(): Configures the duty cycle of PWM used for motor control.

Controller_Func_PI(): This function calculates the proportional-integral (PI) control values for the motor speed.

Utility Functions:

printf(): Used for debugging and interaction with the user through the console.

DELAY_milliseconds(): Provides delays in milliseconds, useful for timing and control.

Data Flow and Data Types

The data flow is implemented by declaring all the variables as global variables in the main function. The various variables and their data types are shown below:

```
unsigned char count = 0;
unsigned char ch = '1';   //declare another variable for convenience
unsigned int Choice = 0;
unsigned int PWM_Duty[2] = {50,50};
unsigned int Desired_Value[2] = {0,0};
unsigned int i=0;
unsigned char j = 0;
Motor Motor1;
Motor1.Desired =50;    //some initial values
Motor1.Actual = 10;
unsigned int Freq=0;
unsigned int RPM =0;
Controller Control1;
Control1.kp = 0.9;
Control1.ki = 0.1;
Control1.kpki = Control1.kp + Control1.ki;
Control1.ek = 0;
Control1.ek_1 = 0;
Control1.uk = 30;
Control1.uk_1 = 30;
Motor1.Desired =120;    //RPM
Motor1.Actual = 10;
```

Code Written

The code below is the one written to implement the motor control system:

```
#include "mcc_generated_files/mcc.h"
#include"../Buton_Debounce_and_LibraryFolders/Button_Debounce_Library/Buttons_Debounce.h"
#include "../motort.h"
//Function Phototype
void FLASH_LEDS(unsigned char secs);


//MAcros
#define HEARTBEAT PORTEbits.RE0
#define DEBUG_LED PORTEbits.RE1
#define SELECT_F Button_Press.B1
#define TMR1_CLK 500000


//Global Variables

__bit TICK_E = 0;
Button_Type Button_Press;
unsigned int CCP2_Captured_Value = 0;
__bit Captured_Flag = 0;


//ISR for Timer0 configured to execute every 10 milliseconds, involving user inputs and heartbeat flash.
void TMR0_ISR_10ms(void)
{
   static unsigned char count = 0;
   TICK_E = 1;
   Find_Button_Press();
   count++;
   if (count == 100)
   {
      count = 0;
      HEARTBEAT = ~HEARTBEAT;
   }
}

void main(void)

{
   // Initialize the device

   typedef enum {SETUP_S, OPEN_S, CLOSED_S } States_T;
```

```
States_T State_Mch = SETUP_S;

unsigned char count = 0;
unsigned char ch = '1';   //declare another variable for convenience

unsigned int Choice = 0;
unsigned int PWM_Duty[2] = {50,50};
unsigned int Desired_Value[2] = {0,0};
unsigned int i=0;
unsigned char j = 0;
Motor Motor1;
Motor1.Desired =50;    //some initial values
Motor1.Actual = 10;
unsigned int Freq=0;
unsigned int RPM =0;
Controller Control1;
Control1.kp = 0.9;
Control1.ki = 0.1;
Control1.kpki = Control1.kp + Control1.ki;
Control1.ek = 0;
Control1.ek_1 = 0;
Control1.uk = 30;
Control1.uk_1 = 30;
Motor1.Desired =120;    //RPM
Motor1.Actual = 10; //just start

SYSTEM_Initialize();
EPWM1_LoadDutyValue(0);
FLASH_LEDS(3);

TMR0_SetInterruptHandler(TMR0_ISR_10ms);
INTERRUPT_GlobalInterruptEnable();
INTERRUPT_PeripheralInterruptEnable();

printf ("\r\nMotor Controller Program Starting \r\n");   //Takes ~1 ms
DELAY_milliseconds(3000);

 while (1)
   {
    // external event sets
     while (!TICK_E);
      switch (State_Mch)
      {
         case SETUP_S:

            //OnEntry actions for the SETUP_S common from any state
            INTERRUPT_GlobalInterruptDisable();
```

```
    EPWM1_LoadDutyValue(0);
    //input to choose between open loop and closed loop control
    printf ("\n Enter Choice. 1 - Open Loop , 2 - Closed Loop \r\n");
    ch = EUSART1_Read();
    if (ch == '1')
    {
       State_Mch = OPEN_S;    //OnEntry Actions here when going to OPEN_S
       printf("\n Running Open Loop Control\r\n");
       printf("Enter PWM value 1 (40 to 80)> \r\n");
       scanf("%d",&PWM_Duty[0]);
       printf("Enter PWM value 2 (40 to 80)> \r\n");
       scanf("%d",&PWM_Duty[1]);
       j=0;
       EPWM1_LoadDutyValue(PWM_Duty[j]*8);
    }
    else
    {
          State_Mch = CLOSED_S;  //OnEntry Actions here when going to
   CLOSED_S
       printf("\nRunning Closed Loop PI Control\r\n");
       printf ("\n Enter Desired RRM Value 1 (100 to 250)> \r\n");
       scanf("%d",&Desired_Value[0]);
       printf ("\n Enter Desired RRM Value 2 (100 to 250)> \r\n");
       scanf("%d",&Desired_Value[1]);
       Motor1.Desired = Desired_Value[0];
       j=0;i=0;
       EPWM1_LoadDutyValue(40*8);  //40% duty cycle to start
    }
    SELECT_F = 0;   //OnExit Action
    INTERRUPT_GlobalInterruptEnable();
    break;
 case OPEN_S:
    if (SELECT_F)
    {
       State_Mch = SETUP_S;
       EPWM1_LoadDutyValue(0);   //Turn off Motor
     }
  break;
  case CLOSED_S:
    if (SELECT_F)
    {
       State_Mch = SETUP_S;
       EPWM1_LoadDutyValue(0);   //Turn off Motor
    }
    break;
 default:
```

```
        State_Mch = SETUP_S;
        break;
}
//Do actions
switch (State_Mch)
{
   case SETUP_S:
      //no Do actions since never stay in this state
      break;
   case OPEN_S:
      i++;
      if (i==1000)  //100 is 1 sec, 500 is 5 sec, 1000 is 10 sec
      {
         j=!j;
         EPWM1_LoadDutyValue(PWM_Duty[j]*8);
         i=0;
      }
      if (Captured_Flag)
         {
              //Freq value could be int but not long
              Freq = (unsigned int)(TMR1_CLK/CCP2_Captured_Value);
              //multiply by 2.5 since multiply by 60(secs to minutes)/24 (pulses per
              rev)
              RPM = (Freq * 5)/2;
              Motor1.Actual = RPM;
              Captured_Flag = 0;
         }
       if (i%10 == 0) //every 100 ms
         {
            printf("%d",PWM_Duty[j] ); //PWM
            printf("\t\t");
            printf("%d\r\n",Motor1.Actual);
         }
      break;

   case CLOSED_S:
          i++; // timer counter
        if (i==500)  //100 is 1 sec, 500 is 5 sec, 1000 is 10 sec
        {
         j=!j; //j between 0 and 1. change of state options
        Motor1.Desired = Desired_Value[j];
        i=0;
   }
           if (Captured_Flag)
        {
          //Freq value could be int but not long
```

```
                    Freq = (unsigned int)(TMR1_CLK/CCP2_Captured_Value)
                    //multiply by 2.5 since multiply by 60(secs to minutes)/24 (pulses per rev)
                    RPM = (Freq * 5)/2;
                    //Sets Motor1.Actual to the newly calculated RPM.
                    Motor1.Actual = RPM;
                    Captured_Flag = 0; //the processed state
                    Controller_Func_PI(&Motor1,&Control1);
                    //Sets the new duty cycle for the PWM signal controlling the motor speed.
                    EPWM1_LoadDutyValue(Control1.uk*8);
                  }
                 if (i%10 == 0) //every 100 ms
                  {
                    printf("%d",Control1.uk ); //PWM
                    printf("\t");
                    printf("%d",Motor1.Actual);
                    printf("\t");
                    printf("%d\r\n",Motor1.Desired);
                  }
                break;
        }
        TICK_E = 0;
    }
}
void FLASH_LEDS(unsigned char secs)
{
    unsigned char i;
    for (i = 0; i < secs; i++)
    {
        PORTE = 0xFF;  // Turn all LEDs on
        DELAY_milliseconds(1000);  // Delay for 3000 milliseconds
        PORTE = 0x00;  // Turn all LEDs off
          }
}
```

Test results

Open Loop

When the motor is operated in an open loop state, the program uses two PWM duty value to set the motor speed. For example, the PWM value used is 40 and 60. The motor control switches between these two values every few seconds, and different motor speeds occur.

When entering the open loop motor control state, the program asks the user to enter two PWM values, and reads it using the scanf() function. From Figure 6 input PWM Value at 70. Actual RPM: Shows variability but generally stabilizes around specific values like 405, 407, and occasionally peaking or dropping slightly (e.g., 415, 400). This indicate how the motor reacts to a higher PWM setting, where the RPM fluctuates before stabilizing.

```
13:21:25.276 -> 70              405
13:21:25.369 -> 70              405
13:21:25.461 -> 70              407
13:21:25.539 -> 70              407
13:21:25.678 -> 70              407
13:21:25.770 -> 70              415
13:21:25.862 -> 70              407
13:21:25.955 -> 70              407
13:21:26.079 -> 70              407
13:21:26.173 -> 70              407
13:21:26.281 -> 70              405
13:21:26.374 -> 70              407
13:21:26.468 -> 70              407
13:21:26.560 -> 70              400
13:21:26.639 -> 70              400
13:21:26.761 -> 70              412
13:21:26.839 -> 70              410
```

Figure 3: Open loop motor at 70 PWM and output RPM

From Figure 7 input PWM Value at 40. Actual RPM: Decreases consistently as the time progresses, starting from around 292 RPM and stabilizing around 182 RPM. This gradual decrease could be due to several factors including mechanical characteristics of the system, such as thermal effects or changes in load.

```
l3:21:50.177 -> 40              292
l3:21:50.256 -> 40              265
l3:21:50.394 -> 40              245
l3:21:50.457 -> 40              230
l3:21:50.595 -> 40              215
l3:21:50.674 -> 40              210
l3:21:50.784 -> 40              202
l3:21:50.892 -> 40              197
l3:21:50.986 -> 40              192
l3:21:51.079 -> 40              190
l3:21:51.172 -> 40              190
l3:21:51.296 -> 40              187
l3:21:51.374 -> 40              187
l3:21:51.498 -> 40              182
l3:21:51.591 -> 40              182
l3:21:51.685 -> 40              185
l3:21:51.778 -> 40              182
```

Figure 7: Open loop motor at 40 PWM and output RPM

From Figure 8 shows RPM over time with a PWM value of 70. The RPM remains relatively steady with minor fluctuations, which is typical in open-loop systems where the control does not adjust in response to RPM deviations.

From Figure 9 shows RPM decline over time with a PWM value of 40. The steady decline followed by stabilization is indicative of the system reaching a new steady state at a lower energy input.

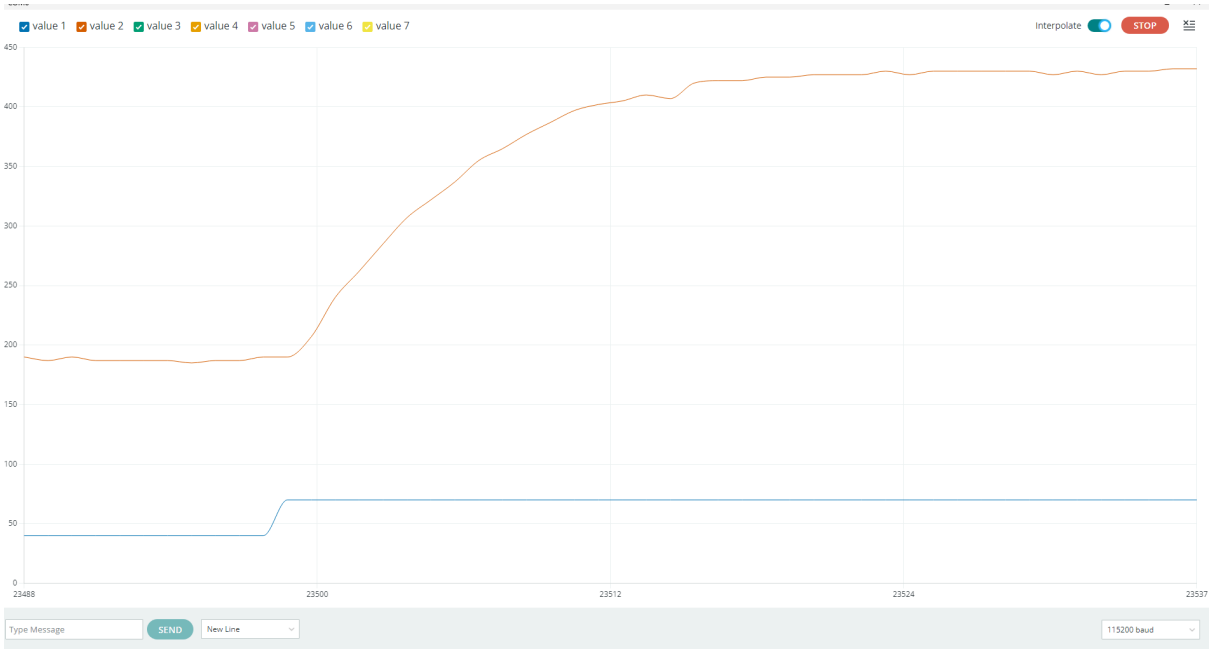

Figure 4: Graph of open loop motor transitioning between 40 and 60 PWM and output RPM

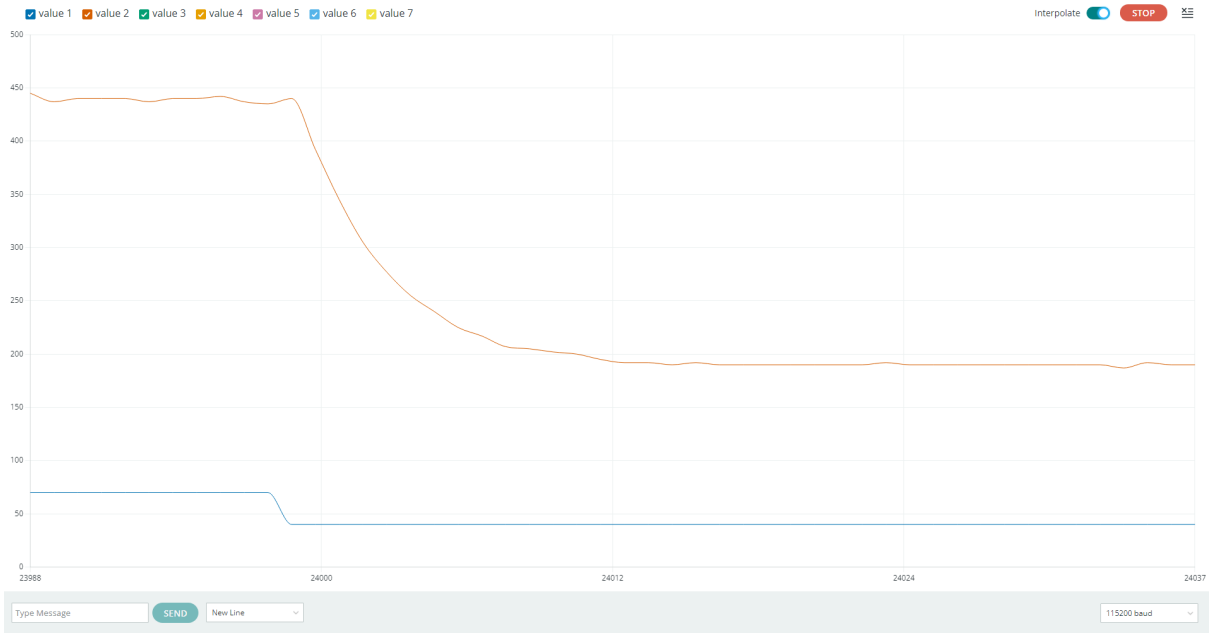

Figure 9: Graph of open loop motor transitioning between 40 and 60 PWM and output RPM

Closed Loop:

When the motor is operated in an close loop state, the program uses two desired RPM value to operate the motor speed. In this example, the desired RPM value used is 190 and 230. The motor control switches between these two values every few seconds, and the motor tries to maintain the constant desired speeds.

When entering the closed loop motor control state, the program asks the user to enter two desired RPM values, and reads it using the scanf() function. From Figure 10, enter 190.

PWM Value: Fluctuates slightly between 38 and 40, indicating minor adjustments by the control system to maintain the desired RPM.

Actual RPM: Varies around 185 to 187 RPM, consistently staying slightly below the desired 190 RPM.

Desired RPM: Constant at 190 RPM.

```
13:28:01.869 -> 40      185     190
13:28:01.933 -> 38      187     190
13:28:02.071 -> 38      187     190
13:28:02.165 -> 38      187     190
13:28:02.257 -> 40      185     190
13:28:02.350 -> 38      187     190
13:28:02.474 -> 38      187     190
13:28:02.552 -> 40      185     190
13:28:02.676 -> 40      185     190
13:28:02.739 -> 40      185     190
13:28:02.864 -> 38      187     190
13:28:02.958 -> 40      185     190
13:28:03.051 -> 40      185     190
13:28:03.175 -> 40      185     190
13:28:03.269 -> 38      187     190
13:28:03.360 -> 40      185     190
```

Figure 5: Closed loop motor at 190 desired RPM

From Figure 11, enter 230.

PWM Value: Shows more variability between 43 and 45. This increased range suggests that the controller is making more significant adjustments, possibly due to greater system demands or a higher setpoint.

Actual RPM: Mostly around 225 to 227 RPM, again slightly below the target of 230 RPM.

Desired RPM: Constant at 230 RPM.

```
13:27:25.029 -> 45      225     230
13:27:25.154 -> 45      225     230
13:27:25.246 -> 45      225     230
13:27:25.339 -> 43      227     230
13:27:25.432 -> 45      225     230
13:27:25.544 -> 43      227     230
13:27:25.637 -> 45      225     230
13:27:25.715 -> 43      227     230
13:27:25.840 -> 45      225     230
13:27:25.933 -> 45      225     230
13:27:26.056 -> 45      225     230
13:27:26.147 -> 45      225     230
```

Figure 6: Closed loop motor at 230 desired RPM

The two figures above show the PWM, Actual RPM, and desired RPM values as it transitions between the two given RPM values. From Figure 12. The graph displays a stable Desired RPM line at two levels, corresponding to 190 and 230 RPM as setpoints. The Actual RPM line tries to track these setpoints but consistently shows a lag or a lower performance than the desired state.

The PWM signal fluctuates and appears to respond to the discrepancies between actual and desired RPMs, which is expected in a closed-loop system trying to minimize error.

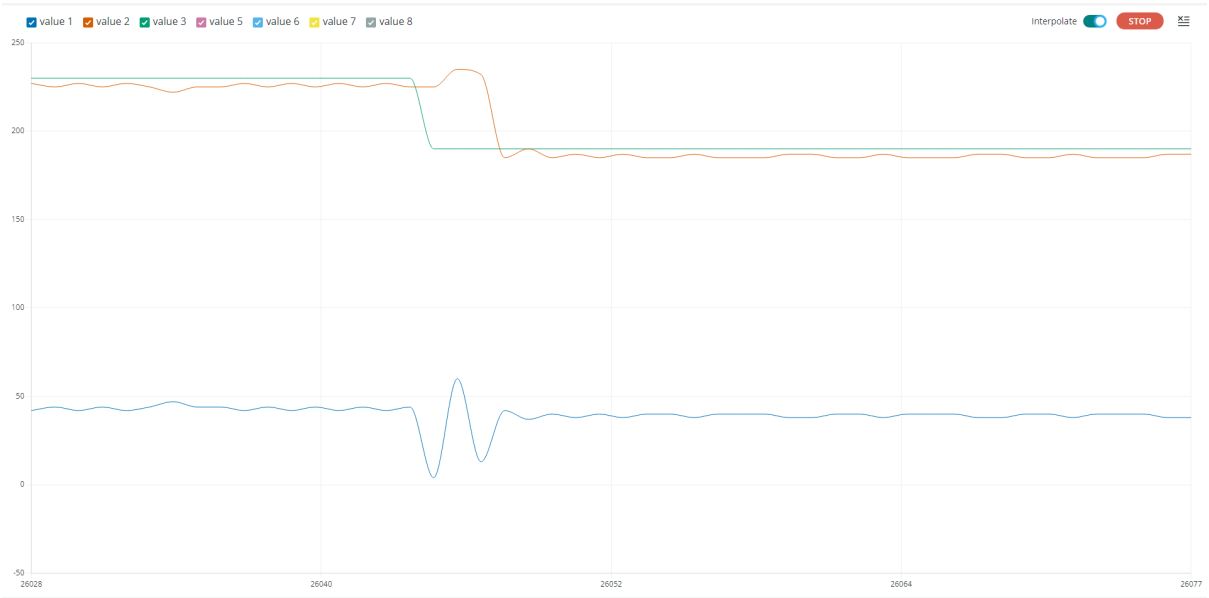

Figure 7: Graph of closed loop motor transitioning between 190 and 230 RPM

From Figure 13, This graph offers a clearer view of the transition period where the Desired RPM shifts from 190 to 230. There's a visible delay in the Actual RPM response, indicating the system's response time and dynamics.

The PWM line shows a sharp increase as the control system reacts to the new higher RPM requirement, stabilizing as the Actual RPM approaches the new desired level.

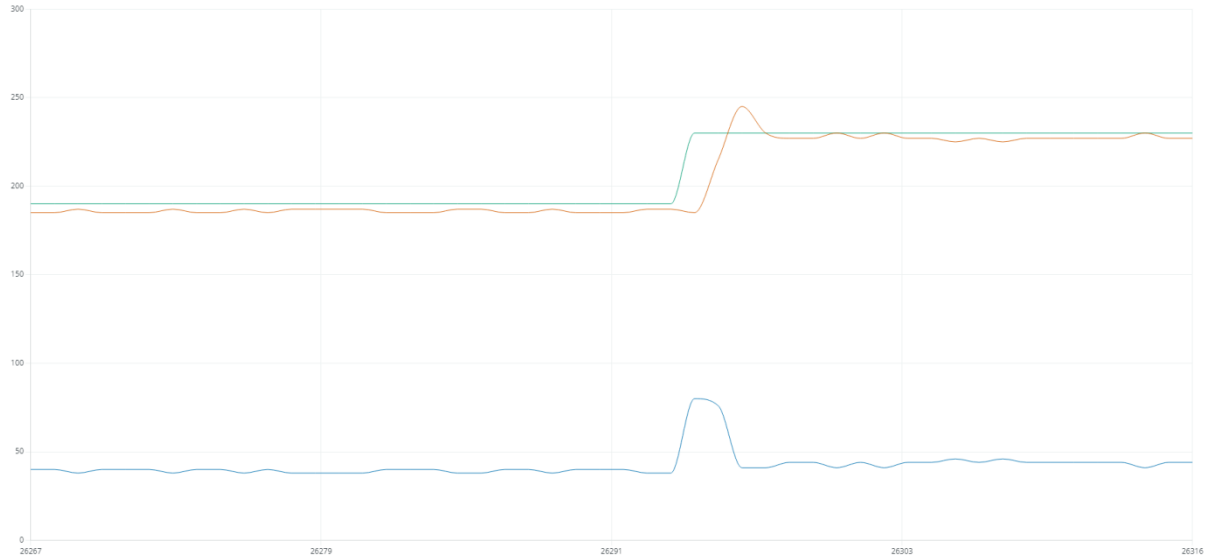

Figure 83: Graph of closed loop motor transitioning between 190 and 230 RPM

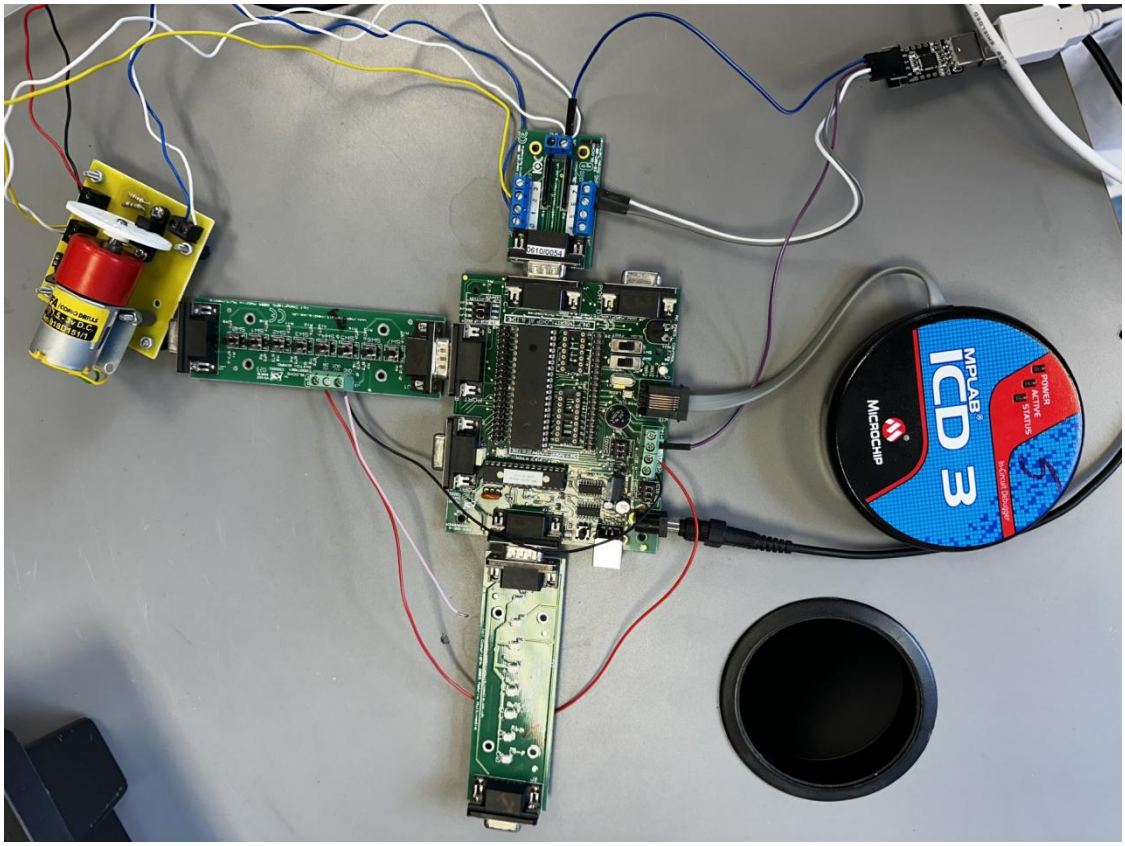

Figure 9: Motor control hardware

The figure above shows the motor control system connected to the PIC16F46K22 along with the various peripheries and components.

Conclusion

In this project, the aim was to evaluate and compare the performance of open-loop and closed-loop control systems in motor control applications. By observing how each control scheme manages motor RPM under varying PWM inputs.

Open-Loop System Summary:

In the open-loop control configuration, the motor's behavior was controlled solely by predetermined PWM settings without any feedback to adjust these settings in response to actual motor performance. The system was straightforward and relied entirely on the assumption that the input settings would achieve the desired output.

The actual RPM of the motor was influenced by fixed PWM values, which did not adjust in response to any discrepancies between desired and actual RPM. This led to predictable but rigid performance, where any external disturbances or variations in load were not compensated.

Closed-Loop System Summary:

The closed-loop control, in contrast, utilized a feedback mechanism to continuously adjust the PWM based on the difference between the desired RPM and the actual RPM. This adaptive approach aimed to minimize the error and maintain the motor RPM as close to the target as possible. The system offered a higher level of precision and adaptability. By constantly adjusting the control inputs in response to actual performance, the closed-loop system could effectively manage disturbances and changes in operating conditions, maintaining the motor's RPM closer to the desired setting even under varying conditions.

Open-Loop Controls are simpler and more cost-effective but lack the ability to adapt to changes in the environment or load, making them suitable for applications where conditions are constant and predictable.

Closed-Loop Controls, while more complex and costly, provide significant advantages in terms of accuracy, adaptability, and efficiency, making them ideal for applications where precision and reliability are critical.