Motor Controller Report

Zedd Serjeant 1261476

27 April 2020

1 Device Purpose

The purpose of this device is to control the speed of a motor using feedback. It must be stable and stand up to system disturbance, such as friction interfering with shaft rotation.

2 Algorithm

2.1 Interrupt

There are three interrupts used to control timing. One occurs approximately every millisecond. This is used to control an LED with a duty cycle. It is set to be proportional to the potentiometer input so that the user knows what setting they have chosen.

The other two serve the purpose of controlling when the system samples the motor speed (*Appendix* 6). Each cycle, as the PWM goes low, the **Timer 2 Interrupt** is triggered. This sets another interrupt to occur after the internal inductance discharges, indicating for the mainline to measure the voltage across the motor terminals.

The discharge of the inductance is the main design factor these interrupts are based around. This system was designed around an inductance of 20 mH, leading to a PWM rate of 280 Hz. A larger inductance requires a slower PWM signal, or a different approach to interrupt timing.

2.2 Mainline

The body of the code is concerned with measuring the motor speed, supply voltage and the potentiometer, and using this information to calculate a duty cycle for the PWM (*Appendix* 6). This in turn sets the motor speed and the LEDs.

The Execution of the mainline is controlled by the interrupts. Each millisecond, a flag is set to measure the supply to ensure it is within the limits specified in the **Specification**(*Section* 5). It also uses this to make the PWM calculations more accurate, as PWM can be seen as setting a fraction of the input voltage. The design is economical for microcontroller pins, as this measurement is made with the same pin as the motor speed measurement. It does this by driving the motor, putting the full input voltage on the terminals, before the measurement.

Each full Period of the PWM, a flag is set to measure the current motor speed. This involves the delay explained in **Interrupts**(*Section* 2.1). After this initial timed measurement, another two are made, separated by the *ADC* clock speed, and a Median Filter is applied for more stability.

This process is then followed by the **Control Algorithm**(*Section* 2.3).

The final task of the Mainline then immediately follows this: The potentiometer is read, and the result interpreted as a voltage to set the motor speed to, a value ranging between 1.47 V to 8 V. This range was chosen empirically and to meet the specification. At 1.47 V, the motor cannot be reliably run without control, and a voltage above 8 V cannot be guaranteed within the specification, so no attempt to support this can be made.

2.3 Control Method

This controller implements **PI** control.

Each cycle of PWM, the output drops low, triggering an interrupt. The system waits a certain amount of time (34 µs) to allow the *motor inductance* to discharge. Three measurements are made and the median taken to give $\mathbf{V_M}$, the voltage of the motor which is representative of shaft speed.

The error is calculated as in *Equation* 1 and is numerically integrated (summed) over time as *Equation* 2

$$\text{error} = \text{set point} - V_M \tag{1}$$

$$\text{error sum} = \int_0^t \mathbf{error}\,\mathrm{d}t = \sum_{i=0}^{t} \mathbf{error}(t) \implies \textit{error sum += error} \tag{2}$$

These values are used in a feedback loop to set the speed as in *Equation* 3, where the values of k_p and k_i are set to lead to a stable and fast response.

$$\text{output} = (\text{set point}) + k_p \cdot (\text{error}) + k_i \cdot (\text{error sum}) \tag{3}$$

3 Hardware Design

The hardware was built with one major constraint: Only components that could be quickly gathered before the initiation of lock down could be used. This limited the resistor and capacitor values that were available, along with forcing the use of a P-type FET.

The circuit sections with interesting design choices are:

- The Programming Header
- The voltage regulator and associated bypass capacitors
- The Feedback voltage divider
- Motor Control

The programming header was implemented to enable *In Circuit Serial Programming*(**ICSP**). This is beneficial as it allows faster prototyping and then fast manufacture.

The voltage regulator is necessary to allow the PIC to function under the fluctuating high range input. This input also has a large level of noise due to the motor being directly connected to this source. Thus Bypass capacitors are necessary, and their values are chosen based on recommended values from the regulators datasheet, and drawn from the limited pool.

To sense the speed of the motor, the voltage across its terminals needs to be read by the ADC. As this voltage can be up to 16 V per the specification, a resistive divider is employed to scale this voltage down to a maximum of 5 V. The first resistor was chosen to be 10 kΩ to limit current though this branch, while the second was chosen to meet the ratio of 5/16.

The Motor, as mentioned is driven by a P-FET rather than the traditional N-FET in a bootstrap configuration. While this made the design simpler it also made the circuit more inefficient. The FET is driven by a BJT controlling the presence of the supply voltage on the gate, allowing the small voltages of the microcontroller to control the motor.

4 Testing and Operating

Whilst the system is running the circuit provides features for user feedback and user adjustment.

There are three LED, one that indicates the current value of the potentiometer as a ratio of duty cycle to period, another that glows when the control loop is offset from the set point by too large a value (currently 1.5 V) and the final LED glows when the circuit is in an error state. The system indicates an error when the supply voltage is outside of specification.

A user knows the system is running well when they can set a motor speed using the POT, causing the POT LED to change, and then the loop-offset LED settles into being off the majority of the time. If the system ever malfunctions, either the loop-offset LED or the error LED will be on permanently. If this is a software error, there is a button attached to induce a soft-reset; pressing the button sets all internal error values to zero and temporarily disables the control loop until the system stabilises.

If the user notices some instability in the control loop, there is a process by which they can adjust the internal control constants:

1. set the motor speed to zero via the potentiometer

2. Press the button. Instead of initiating a reset, this will now adjust the system mode
3. The first mode adjusts the proportion constant. The motor will vary up and down across is full speed range, while the potentiometer now sets the proportion constant
4. Any press of the button will save the new proportion constant, and move the system into integral-constant-set-mode. The motor will continue to vary and the potentiometer will now set the integral constant.
5. Any press of the button will now save the current integral value and move back to proportion set mode
6. To return to speed control, cycle the power.

When these circuits go through manufacture, these features can be used to ensure the product is ready for release.

The circuit is powered and placed into constant set mode. As the motor speed varies, the loop-offset LED is observed (either by a manufacture system or a human). If the system is not responding well enough, the constants can be adjusted until it is.

5 Appendix - Specification

Here are the specifications as laid out by the *Project Manager*

- Goal: keep the rotational speed of the motor shaft constant, as set by the Input Potentiometer.
- Both the code and circuit need to be designed in tandem, with LEDs serving as outputs of useful information.
- The system is required to run on a variable supply of 8 V to 16 V.
- The Control loop may not oscillate around the set point excessively.

6 Appendix - Timing

Various Figures showing the timing of different important parts of the program

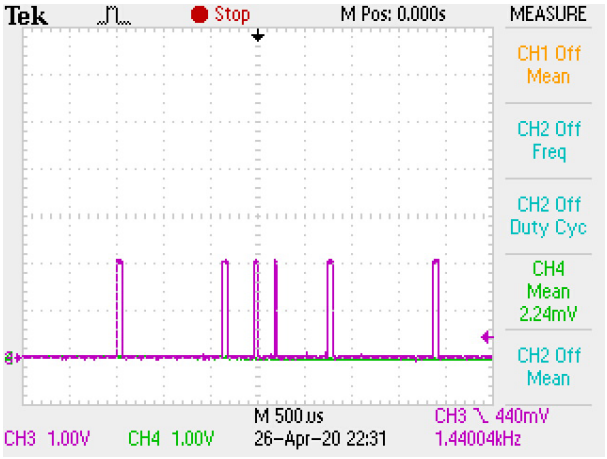

Figure 1: Interrupt Time - high when within the interrupt

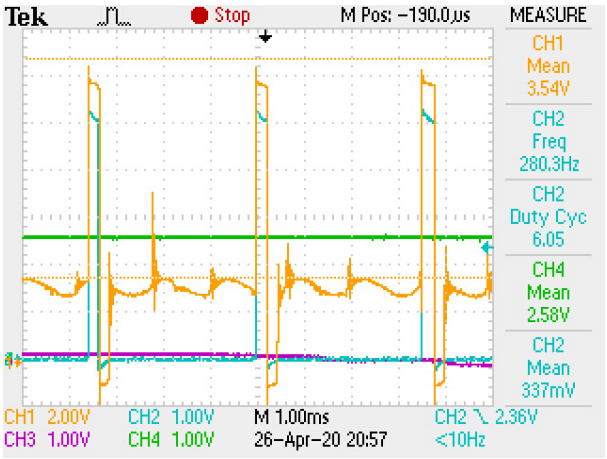

Figure 2: Mainline PWM generation - Orange:Motor, Blue:PWM, Green:Potentiometer

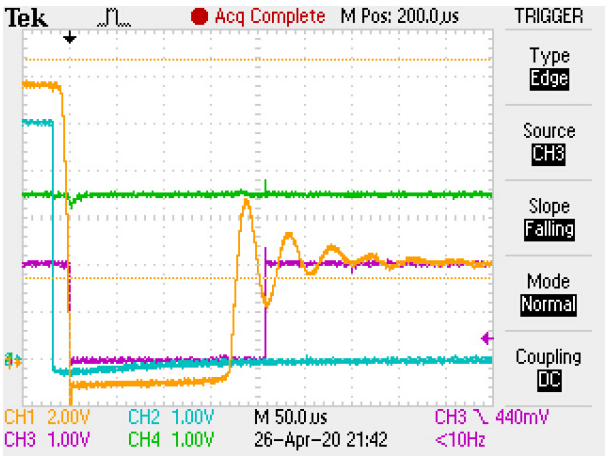

Figure 3: Sampling - Orange: Motor, Blue: PWM, Green: Potentiometer, Purple: Sampling Interrupt

7 Appendix - Software

7.1 head.h

```
#include <xc.h> // include processor files - each processor file is guarded.

enum FLAGS {OFF=0, OUTPUT=0, INTERNAL=0, CLEAR=0, DIGITAL=0, ANALOG=1, INPUT=1, ON=1, RIGHT=1};
#define GLOBAL_INTERRUPTS GIE

//Pins

#define DAT_LED RA0
#define DAT_LED_PIN TRISA0
#define DAT_LED_TYPE ANS0
#define DAT_LED_BIT 0b000001

#define CLK_LED RA1
#define CLK_LED_PIN TRISA1
#define CLK_LED_TYPE ANS1
#define CLK_LED_BIT 0b000010

#define IND_LED RA2
#define IND_LED_PIN TRISA2
#define IND_LED_TYPE ANS2
#define IND_LED_BIT 0b000100

#define PWM_MOTOR RC5
#define PWM_MOTOR_PIN TRISC5

#define MOTOR_READING RC1
#define MOTOR_READING_PIN TRISC1
#define MOTOR_READING_TYPE ANS5

// button
#define BUTTON RA3
#define BUTTON_PIN TRISA3
#define BUTTON_INTERRUPT IOCA3
#define RA_INTERRUPT RAIE
#define BUTTON_INTERRUPT_FLAG RAIF

//POT
#define POT RC0
#define POT_PIN TRISC0
#define POT_TYPE ANS4

//TIMER0
#define PRESCALER PSA
#define TIMER0_COUNTER TMR0
#define TIMER0_CLOCK_SCOURCE T0CS
#define TIMER0_INTERRUPT T0IE
#define TIMER0_INTERRUPT_FLAG TMR0IF

//timer 1
#define TIMER1_H TMR1H
#define TIMER1_L TMR1L
#define TIMER1_INTERRUPT_FLAG TMR1IF
```

```
#define TIMER1_INTERRUPT TMR1IE
#define TIMER1 TMR1ON


//PWM
//parts are specified as an offset within a register
#define PWM_CONTROL CCP1CON
// #define PWM_MODE P1M
#define PWM_MODE (unsigned char)6
// #define PWM_OUTPUT CCP1M
#define PWM_OUTPUT (unsigned char)0
#define PWM_DUTYCYCLE_MSB CCPR1L
// #define PWM_DUTYCYCLE_LSB DC1B
#define PWM_DUTYCYCLE_LSB (unsigned char)4
#define PWM_PERIOD PR2

//timer 2 (needed for PWM)
#define TIMER2 TMR2
#define TIMER2_CONTROL T2CON
#define TIMER2_ON 2
#define TIMER2_INTERRUPT_FLAG TMR2IF
#define TIMER_CLOCK_PRESCALE (unsigned char)0
#define TIMER_CLOCK_POSTSCALE (unsigned char)3
#define PRESCALE_1 (unsigned char)0b00
#define PRESCALE_4 (unsigned char)0b01
#define PRESCALE_16 (unsigned char)0b10

#define PERIPHAL_INTERRUPT PEIE
#define TIMER2_INTERRUPT TMR2IE

// #define TIMER2_PRESCALER T2CKPS

#define ACTIVE_HIGH_ACTIVE_HIGH (unsigned char)0b1100
#define ACTIVE_LOW_ACTIVE_LOW (unsigned char)0b1111
#define SINGLE_OUTPUT (unsigned char)0b00

//ADC
#define ADC_VOLTAGE_REFERENCE VCFG
#define ADC_CLOCK_SOURCE2 ADCS2
#define ADC_CLOCK_SOURCE1 ADCS1
#define ADC_CLOCK_SOURCE0 ADCS0
#define ADC_CHANNEL2 CHS2
#define ADC_CHANNEL1 CHS1
#define ADC_CHANNEL0 CHS0
#define ADC_GODONE GO_DONE
#define ADC_OUTPUT_FORMAT ADFM
#define ADC_RESULT_HIGH ADRESH
#define ADC_RESULT_LOW ADRESL
#define ADC_INTERRUPT ADIE
#define ADC_INTERRUPT_FLAG ADIF
#define ADC_ON ADON
#define ADC_PAUSE asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;")
```

7.2 main.c

```
// CONFIG
#pragma config FOSC = INTOSCIO  // Oscillator Selection bits (INTOSCIO oscillator: I/O function on RA4/OSC2/CLKOUT pin, I/O function on RA5/OSC1/CLKIN)
#pragma config WDTE = OFF       // Watchdog Timer Enable bit (WDT disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable bit (PWRT disabled)
#pragma config MCLRE = OFF      // MCLR Pin Function Select bit (MCLR pin function is digital input, MCLR internally tied to VDD)
#pragma config CP = OFF         // Code Protection bit (Program memory code protection is disabled)
#pragma config CPD = OFF        // Data Code Protection bit (Data memory code protection is disabled)
#pragma config BOREN = OFF      // Brown Out Detect (BOR disabled)
#pragma config IESO = OFF       // Internal External Switchover bit (Internal External Switchover mode is disabled)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enabled bit (Fail-Safe Clock Monitor is disabled)


#include "head.h"

//variables for saving data to the onboard memory
#define PROPORTION_CONSTANT_ADDRESS (unsigned char)0
#define INTEGRAL_CONSTANT_ADDRESS (unsigned char)1

// for timing smaller than a single time loop.
#define TIMER0_INITIAL 118

// represents an LED.
volatile struct LED_TYPE
{
    unsigned short int period;
    unsigned short int duty_cycle;
    unsigned short int counter;
} system_state;


unsigned short int pwm_duty_cycle = 0;//0x3FF; //[ms]
// unsigned short int pwm_period = 500;
volatile unsigned short int speed = 0; // 0->1024, voltage as represented on the POT
unsigned short int in_voltage = 1001; // 16V, the highest this should receive, thus the default
unsigned short int max_voltage = 501; // 7.6V max voltage the motor is allowed to run at (95% of 8V)
unsigned short int min_voltage = 147; // 2.4V min voltage
volatile __bit measure_motor;
volatile __bit measure_supply;
volatile __bit measure_pot;

// when the constants are being set (>1 system mode) then the speed oscillates automatically
//    indicates whether we are setting the proportion constant. This involves moving the speed up and down and reading the
//    constant off of the pot. The error will be periodically reset
#define SPEED_CHANGE_RATE (unsigned short int)1000 //[ms]
volatile unsigned char system_mode = 0;
volatile unsigned short int speed_change_count = SPEED_CHANGE_RATE;
signed short int speed_delta = 30; //[Vadc]
volatile __bit clear_errors;
volatile __bit error_state;

//BUTTON
volatile __bit increment_mode; // indicates whether we should move to the next mode
#define BUTTON_BOUNCE (unsigned char)200
volatile unsigned char button_bounce_count = BUTTON_BOUNCE;
```

```
//control variables
signed long int ratio;
//proportion
signed long int error = 0;
unsigned char proportion_constant;
//integral
signed long int error_sum = 0;
unsigned char integral_constant;

// shadow register for PortA, so as to not suffer from read/modify/write errors
volatile union
{
    unsigned char byte;
    struct
    {
        unsigned RA0:1;
        unsigned RA1:1;
        unsigned RA2:1;
        unsigned RA3:1;
        unsigned RA4:1;
        unsigned RA5:1;
    } bits;
    struct
    {
        unsigned DAT_LED:1;
        unsigned CLK_LED:1;
        unsigned IND_LED:1;
        unsigned _:3;
    };
} PORTA_SH;

union reading // a union allowing byte combination from the adc
{
    // arranged this way for memory locations to match
    unsigned short int readings[3];
    struct
    {
        unsigned char reading3_array[2]; // third adc result
        unsigned char reading2_array[2]; // second adc result
        unsigned char reading1_array[2]; // first adc result.
    };
    struct
    {
        unsigned short int reading3; //reading3_array[1] # reading3_array[0]
        unsigned short int reading2; //reading2_array[1] # reading2_array[0]
        unsigned short int reading1; //reading1_array[1] # reading1_array[0]
    };
} sample;
// Save memory!


void EEPROMWrite(unsigned char address, unsigned char data)
{
    while (WR); // wait for a previous write to finish
    EEADR = address; //load the address
```

```
    EEDATA = data; // load the data
    GLOBAL_INTERRUPTS = OFF;
    WREN = 1; // enable writes to occur
    EECON2 = 0x55; // each of these bytes is a process required by the hardware
    EECON2 = 0xAA;
    WR = 1; // Initiate the write sequence
    WREN = 0; // disable writes
    GLOBAL_INTERRUPTS = ON;
}

unsigned char EEPROMRead(unsigned char address)
{
    EEADR = address; // load the address
    RD = 1; // initiate read
    return EEDATA;
}

//a function (and input defaults) for calculating the the value for the dutycycle register based on the period and a ratio
enum CALC_PWM_PARAMS {ACTIVE_LOW=0, ACTIVE_HIGH=1};
unsigned short int calcPWM(unsigned char period, unsigned short int ratio, unsigned char active_high)
{
    if (active_high)
    {
        return (4*((__uint24)period+1)*ratio)/100;
    }
    else // active low
    {
        return (4*((__uint24)period+1)*(100-ratio))/100;
    }
}

//median of three values
unsigned short int medianValue(unsigned short int samples[]) // returns the index of the middle value
{
   if (samples[0] > samples[1])
   {
        if (samples[1] > samples[2])
        {
            return samples[1];
        }
        else if (samples[0] > samples[2])
        {
            return samples[2];
        }
        else
        {
            return samples[0];
        }
    }
    else
    {
        if (samples[0] > samples[2])
        {
            return samples[0];
        }
        else if (samples[1] > samples[2])
```

```
        {
            return samples[2];
        }
        else
        {
            return samples[1];
        }
    }
}

void __interrupt() ISR()
{
    // turned on by timer 2
    if (TIMER1_INTERRUPT_FLAG)
    {
        GO_DONE = 1;
        measure_motor = 1;

        TIMER1_INTERRUPT_FLAG = 0;
        TIMER1_INTERRUPT = OFF;

    }

    //connected to the PWM
    if (TIMER2_INTERRUPT_FLAG)
    {
        TIMER1_INTERRUPT_FLAG = 0;
        TIMER1_H = 0xFF; TIMER1_L = 0xDD;
        TIMER1_INTERRUPT = ON;

        TIMER2_INTERRUPT_FLAG = 0;
    }

    //millisecond interrupt for LED control
    if (TIMER0_INTERRUPT_FLAG) // if the timer0 interrupt flag was set (timer0 triggered)
    {
        TIMER0_INTERRUPT_FLAG = CLEAR; // clear interrupt flag since we are dealing with it
        TIMER0_COUNTER = TIMER0_INITIAL + 2; // reset counter, but also add 2 since it takes 2 clock cycles to get going
        // move counters, which is the job of this timer interrupt
        system_state.counter++; // increment the led counter
        // PWM_MOTOR = ~PWM_MOTOR;
        // measure_pot = 1;
        // measure_supply = 1;

        if (system_state.counter >= system_state.period)
        {
            system_state.counter -= system_state.period; //reset led counter safely
            measure_supply = 1; // every so often, ensure the supply is at the right level, and use it to set the ratios more accurately
        }
        if (system_state.counter >= system_state.duty_cycle)
        {
            PORTA_SH.DAT_LED = OFF;
        }
        else
        {
            PORTA_SH.DAT_LED = ON; // within On part of duty cycle
```

```
            // PWM_MOTOR = ON;
        }

        //other timings
        if (button_bounce_count)
        {
            button_bounce_count--;
        }
        if (system_mode >= 1)
        {
            if (speed_change_count)
            {
                speed_change_count--;
            }
            else
            {
                speed_change_count = SPEED_CHANGE_RATE;
                speed +=speed_delta;
                if (speed > max_voltage)
                {
                    speed_delta = -1*speed_delta;
                    speed += 2*speed_delta;
                    clear_errors = 1;
                }
                else if (speed < min_voltage)
                {
                    speed_delta = -1*speed_delta;
                    speed += 2*speed_delta;
                }
            }
        }
    }
    if (BUTTON_INTERRUPT_FLAG)
    {
        BUTTON_INTERRUPT_FLAG = CLEAR; // we are dealing with it
        if (!BUTTON && !button_bounce_count) // button was pressed and therefore this will read low (and we avoided bounce)
        {
            increment_mode = 1;
            button_bounce_count = BUTTON_BOUNCE;
        }
    }

}

void main() {
    //set up IO
    DAT_LED_TYPE = DIGITAL;
    DAT_LED_PIN = OUTPUT;

    CLK_LED_TYPE = DIGITAL;
    CLK_LED_PIN = OUTPUT;

    IND_LED_TYPE = DIGITAL;
    IND_LED_PIN = OUTPUT;
```

```
PORTA_SH.byte = 0;

//setup PWM and timer 2
PWM_MOTOR_PIN = OUTPUT;
PWM_MOTOR = OFF;
TIMER2_CONTROL = (ON<<TIMER2_ON) | (PRESCALE_16<<TIMER_CLOCK_PRESCALE); // Timer 2 register
PWM_PERIOD = 222; // 280Hz
pwm_duty_cycle = calcPWM(PWM_PERIOD, 100, ACTIVE_LOW);
PWM_CONTROL = (SINGLE_OUTPUT<<PWM_MODE) | (ACTIVE_LOW_ACTIVE_LOW<<PWM_OUTPUT) | ((pwm_duty_cycle & 0b11)<<PWM_DUTYCYCLE_LSB); //PWM register set
PWM_DUTYCYCLE_MSB = (unsigned char)(pwm_duty_cycle>>2);
PERIPHAL_INTERRUPT = ON;
TIMER2_INTERRUPT = ON;

//timer1
TIMER1_H =0x00; TIMER1_L = 0x00;
// TIMER1_INTERRUPT = OFF;
TIMER1 = ON;


// Set up timer0
// calculate intial for accurate timing $ inital = TimerMax-((Delay*Fosc)/(Prescaler*4))
TIMER0_COUNTER = TIMER0_INITIAL; // set counter
TIMER0_CLOCK_SCOURCE = INTERNAL; // internal clock
PRESCALER = 0; // enable prescaler for Timer0
PS2=0; PS1=1; PS0=0; // Set prescaler to 1:8
TIMER0_INTERRUPT = ON; // enable timer0 interrupts

//Set up ADC
// MOTOR_READING_PIN = INPUT;
// MOTOR_READING_TYPE = ANALOG;
ADC_VOLTAGE_REFERENCE = INTERNAL;
ADC_CHANNEL2 = 1; ADC_CHANNEL1 = 0; ADC_CHANNEL0 = 1; // Set the channel to AN5 (where the motor feedback is)
ADC_CLOCK_SOURCE2 = 0; ADC_CLOCK_SOURCE1 = 0; ADC_CLOCK_SOURCE0 = 1; // Set the clock rate of the ADC
ADC_OUTPUT_FORMAT = RIGHT; // right Shifted ADC_RESULT_HIGH contains the first 2 bits
// ADC_INTERRUPT = OFF; // by default these aren't necessary
ADC_ON = ON; // turn it on

//setup button
// BUTTON_PIN = INPUT;
BUTTON_INTERRUPT = ON;
RA_INTERRUPT = ON;

// load system variables
// proportion_constant = EEPROMRead(PROPORTION_CONSTANT_ADDRESS);
// integral_constant = EEPROMRead(INTEGRAL_CONSTANT_ADDRESS);
proportion_constant = 40; // x/100
integral_constant = 1; // x/100

//turn on interrupts
GLOBAL_INTERRUPTS = ON;

while (1)
{
    if (measure_motor)
    {
        GLOBAL_INTERRUPTS = OFF;
```

```
while(GO_DONE); // wait until the first measurement (initiated by the timer) is made

GO_DONE = 1; //BEGIN second measurement

sample.reading1_array[1] = ADC_RESULT_HIGH; //SAVE first measurement
sample.reading1_array[0] = ADC_RESULT_LOW;

while(GO_DONE);

GO_DONE = 1; //BEGIN third measurement

sample.reading2_array[1] = ADC_RESULT_HIGH; //SAVE second measurement
sample.reading2_array[0] = ADC_RESULT_LOW;

while(GO_DONE);

// CLK_LED = ON;

sample.reading3_array[1] = ADC_RESULT_HIGH; //SAVE third measuement
sample.reading3_array[0] = ADC_RESULT_LOW;

GLOBAL_INTERRUPTS = ON;
measure_motor = 0;

// speed = 250;

error = (signed long int)speed - medianValue(sample.readings);
error_sum += error;
if (clear_errors)
{
    clear_errors = 0;
    error = 0;
    error_sum = 0;
    PORTA_SH.CLK_LED = OFF;
}

if (error > 200 || error < -200)
{
    PORTA_SH.IND_LED = ON;
}
else
{
    PORTA_SH.IND_LED = OFF;
}

if (!error_state)
{
    ratio = (((signed long int)speed+(error*proportion_constant)/100+(error_sum*integral_constant)/100)*22)/in_voltage;
}
else
{
    ratio = 0;
}

if (ratio <= 0) // error magnitude too large
```

```
    {
        ratio = 1;
    }
    else if (ratio > 100)
    {
        ratio = 95;
    }


    pwm_duty_cycle = calcPWM(PWM_PERIOD, ratio, ACTIVE_LOW);
    PWM_DUTYCYCLE_MSB = (unsigned char)(pwm_duty_cycle>>2);
    // reset this so the lsb is in the mix
    PWM_CONTROL = (SINGLE_OUTPUT<<PWM_MODE) | (ACTIVE_LOW_ACTIVE_LOW<<PWM_OUTPUT) | ((pwm_duty_cycle & 0b11)<<PWM_DUTYCYCLE_LSB);


    measure_pot = 1;

}

if (measure_supply)
{

    if (PWM_MOTOR) // if supply is high
    {


        GLOBAL_INTERRUPTS = OFF;
        GO_DONE = 1;
        while (GO_DONE);
        // CLK_LED = 1;
        sample.reading1_array[1] = ADC_RESULT_HIGH;
        sample.reading1_array[0] = ADC_RESULT_LOW;

        if (sample.reading1 > 0)
        {
            if (sample.reading1 < max_voltage || sample.reading1 > 1001) // we are less than the highest setting
            {
                error_state = 1;
                PORTA_SH.CLK_LED = ON;
            }
            else
            {
                error_state = 0;
                in_voltage = sample.reading1;
                PORTA_SH.CLK_LED = OFF;
            }


        }

        GLOBAL_INTERRUPTS = ON;

        measure_supply = 0;
        measure_pot = 0;
        measure_motor = 0;
    }
```

```
}


if (measure_pot)
{
    measure_pot = 0;


    GLOBAL_INTERRUPTS = OFF;
    ADC_CHANNEL2 = 1; ADC_CHANNEL1 = 0; ADC_CHANNEL0 = 0; // Set the channel to AN4 (where the POT is)
    ADC_PAUSE; // wait for small amount of time for the channels to redirect

    GO_DONE = 1; // begin an ADC read

    while(GO_DONE); // wait until the first measurement (initiated by the timer) is made

    GO_DONE = 1; //BEGIN second measurement

    sample.reading1_array[1] = ADC_RESULT_HIGH; //SAVE first measurement
    sample.reading1_array[0] = ADC_RESULT_LOW;

    while(GO_DONE);

    sample.reading2_array[1] = ADC_RESULT_HIGH; //SAVE second measurement
    sample.reading2_array[0] = ADC_RESULT_LOW;

    ADC_CHANNEL2 = 1; ADC_CHANNEL1 = 0; ADC_CHANNEL0 = 1; // Set the channel back to AN5 (where the Motor feedback is)

    GLOBAL_INTERRUPTS = ON;

    if (system_mode == 0) // basic speed set mode
    {
        system_state.period = max_voltage;
        system_state.duty_cycle = speed;

        speed = (unsigned short int)((((unsigned long int)sample.reading1 + sample.reading2) * max_voltage)/2048);

        if (increment_mode)
        {
            increment_mode = 0;

            if (sample.reading1 < 5)
            {
                speed = 150;
                system_mode++;
            }
            else
            {
                clear_errors = 1;
                PORTA_SH.CLK_LED = ON;
            }
        }

    }
    else if (system_mode == 1) //proportion set mode
```

```
            {
                system_state.duty_cycle = 0xFFFF;
                proportion_constant = (unsigned char)(((unsigned long int)sample.reading1*195)/1000);

                if (increment_mode)
                {
                    increment_mode = 0;
                    system_mode = 2;

                    EEPROMWrite(PROPORTION_CONSTANT_ADDRESS, proportion_constant);
                }
            }
            else if (system_mode == 2)
            {
                system_state.duty_cycle = 0;
                integral_constant = (unsigned char)(((unsigned long int)sample.reading1*97)/100);

                if (increment_mode)
                {
                    increment_mode = 0;
                    system_mode = 1;

                    EEPROMWrite(INTEGRAL_CONSTANT_ADDRESS, integral_constant);
                }
            }
        }

        // speed = system_state.duty_cycle;
        PORTA = PORTA_SH.byte; //write out IO register to avoid read-modify-write errors

    }

    return;
}
```

8 Appendix - Hardware

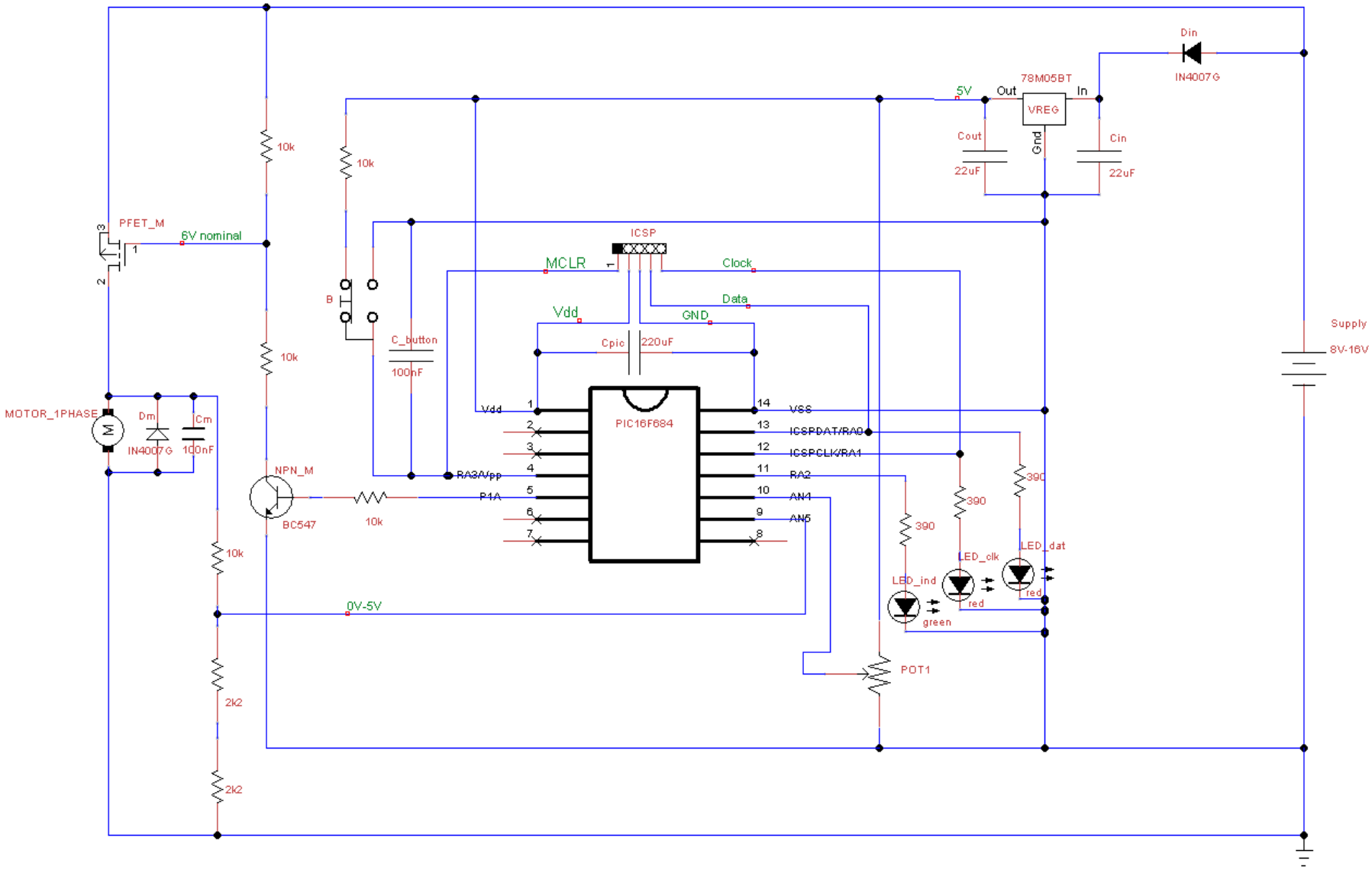

Figure 4: Hardware Schematic

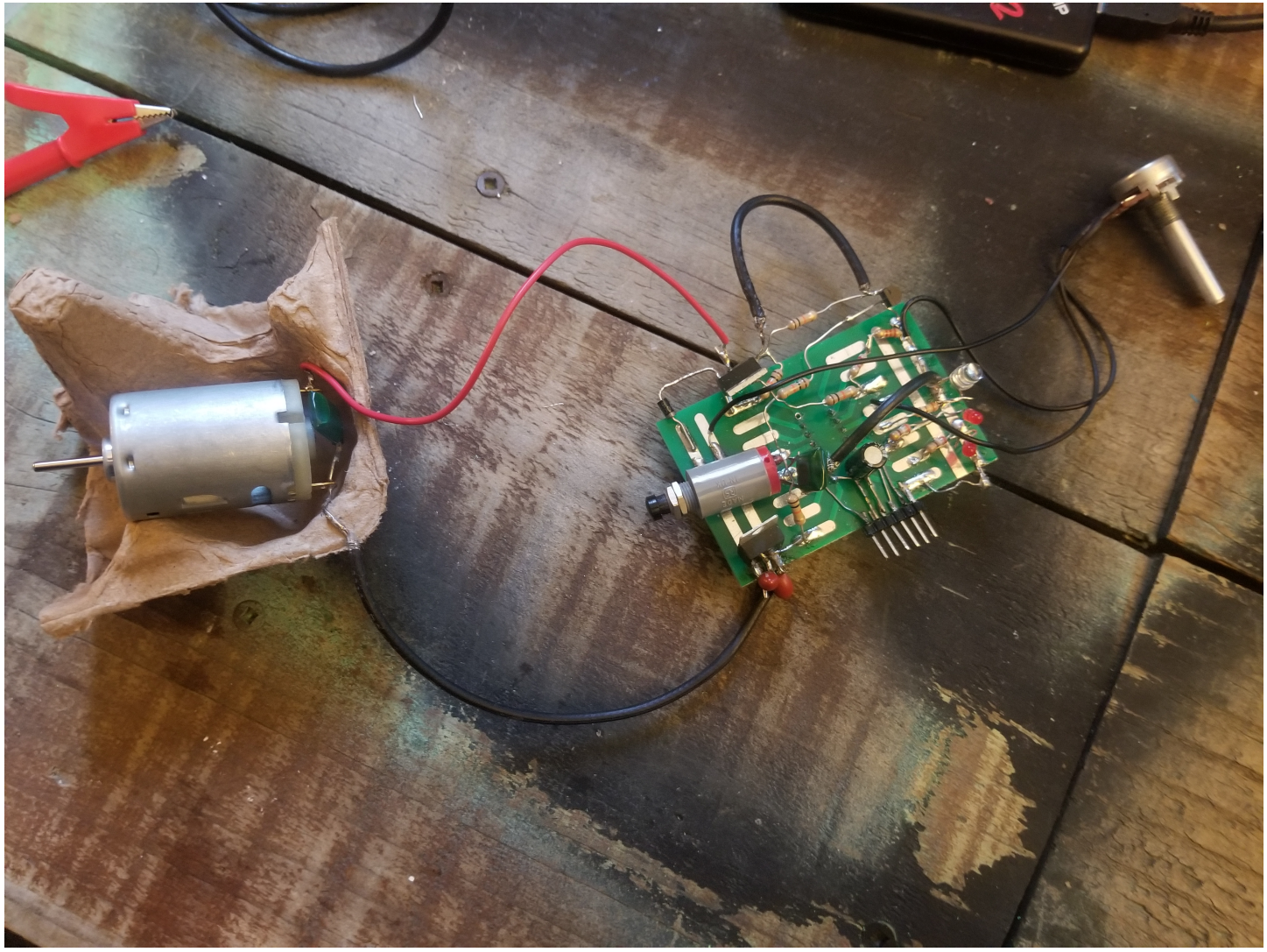

Figure 5: Hardware Prototype Top-down

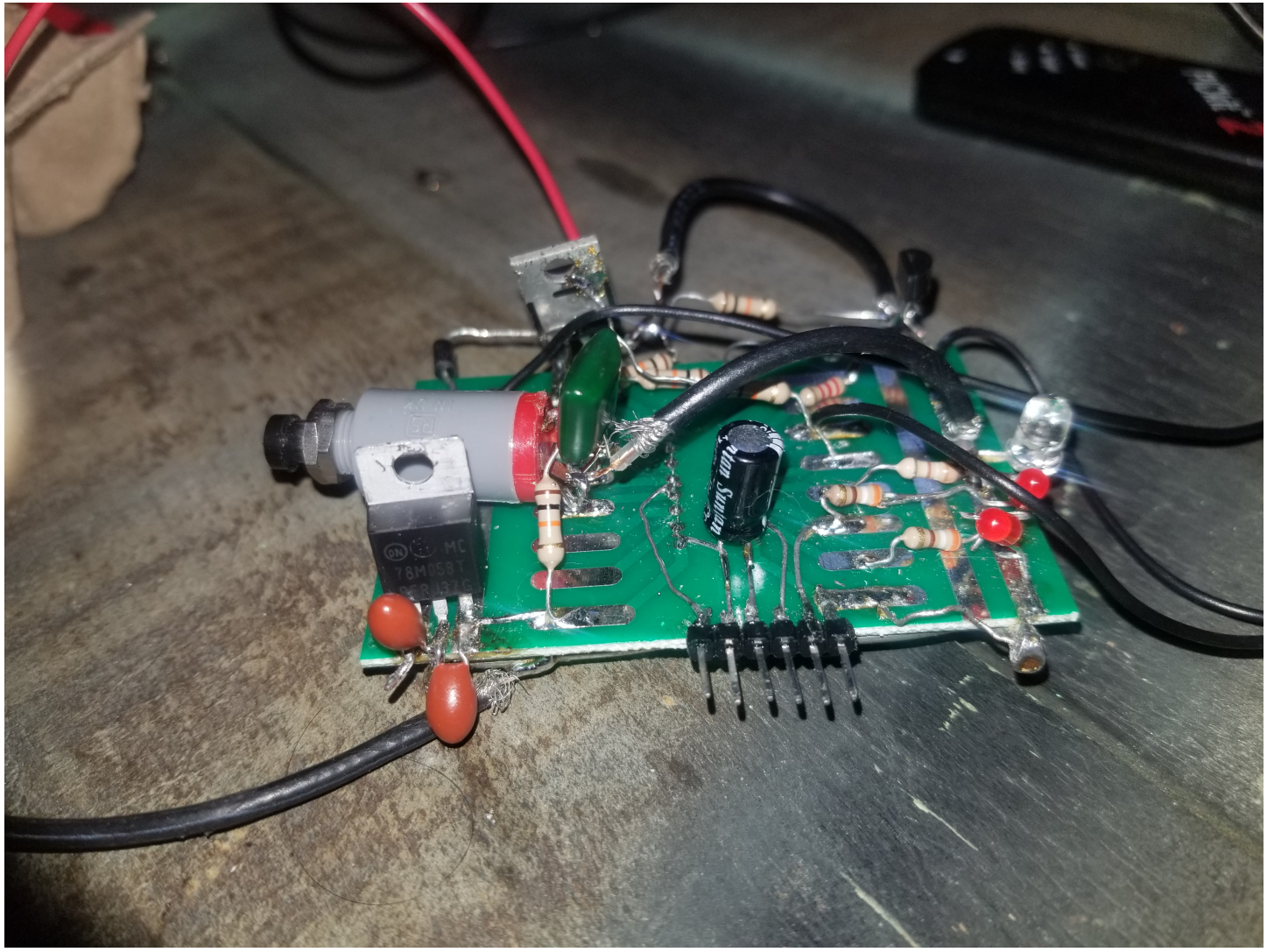

Figure 6: Hardware Prototype Circuit Board