Final Project Report: *Brushless DC Motor Control*

N. Hoffman [2] N. Klein [2] N. Kosaric [2]
Rowan University

May 9, 2019

1 Design Overview

The purpose of the Brushless DC Motor Control Final Project is to create a control system that carefully and precisely controls the speed of a brushless DC motor by taking in a command signal that dictates the desired speed and a feedback signal which comes from the output of the motor through a frequency to voltage converter. The feedback is used as an error signal to correct the difference between the desired speed and actual speed of the motor.

Brushless DC motors have become more prevalent in every day use. Because of this, accurate speed control to ensure that appliances perform at the proper specifications have become more important than ever. In order to test and achieve this, a brushless DC motor driver, frequency to voltage converter, and PID controller were all built and tuned to ensure accurate and quick speed control.

1.1 Design Features

In order to achieve and test the accuracy of the PID controller, the following design features have been utilized:

These are the design features:

- Brushless DC motor driver
- Hall effect sensor
- Frequency to voltage Converter
- PID controller
- Variable speed control with a potentiometer

1.2 Featured Applications

- Electric vehicles
- Industrial motor control
- Electric power tools

1.3 Design Resources

All code for the motor driver and PID controller can be found here :
bitbucket.org/Gliderman/sac-motor-controller/src/master/

2 Key System Specifications

Parameter	Specifications	Details
Peak Time (Tp)	1.2390s	Peak time is the time it takes for the system to reach its final value
Percent Overshoot (%OS)	4.0735%	Percent overshoot is the amount the wave form overshoots the system's final value
Settling Time (Ts)	1.8922s	Settling time is the time it takes for the response of the system to remain and stay within 2% of its final value
Rise Time (Tr)	0.4481s	Rise time is the time it takes for the system to go from 10% of its final value to 90% of its final value

3 System Description

The motor control system is comprised of a motor who's speed is controlled by a dsPIC33EV256GM102 microcontroller which takes a voltage through a potentiometer to determine the intended power going to the motor as well as a feedback voltage which allows the PIC to adjust its signal based on system error.

3.1 Schematics and Implementation

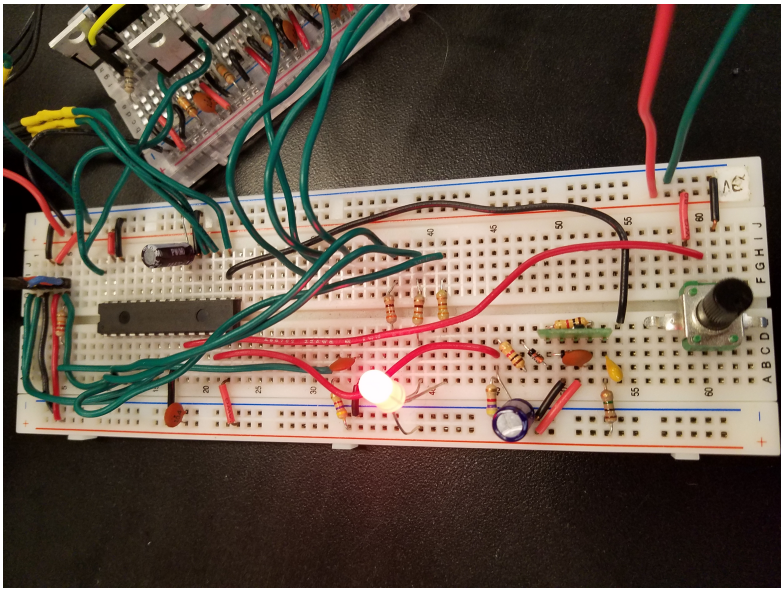

Figure 1: Board controlling the logic of the system

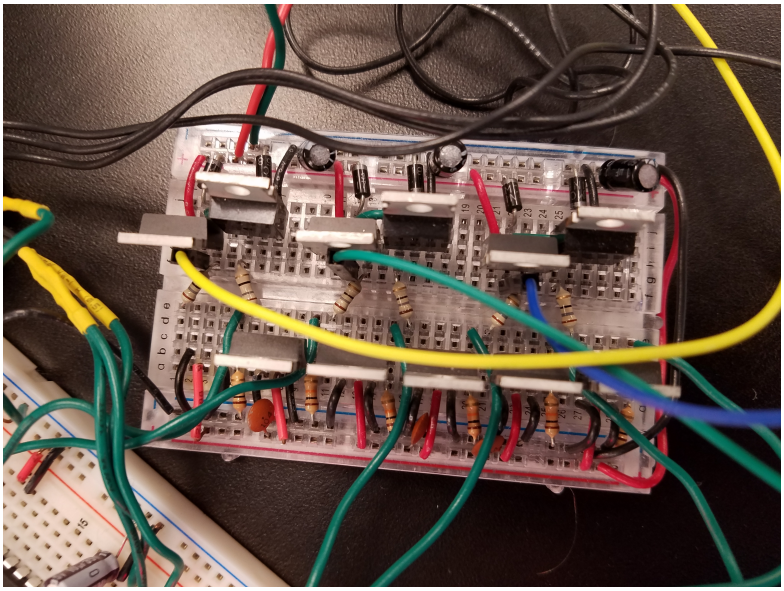

Figure 2: Motor Driver Board

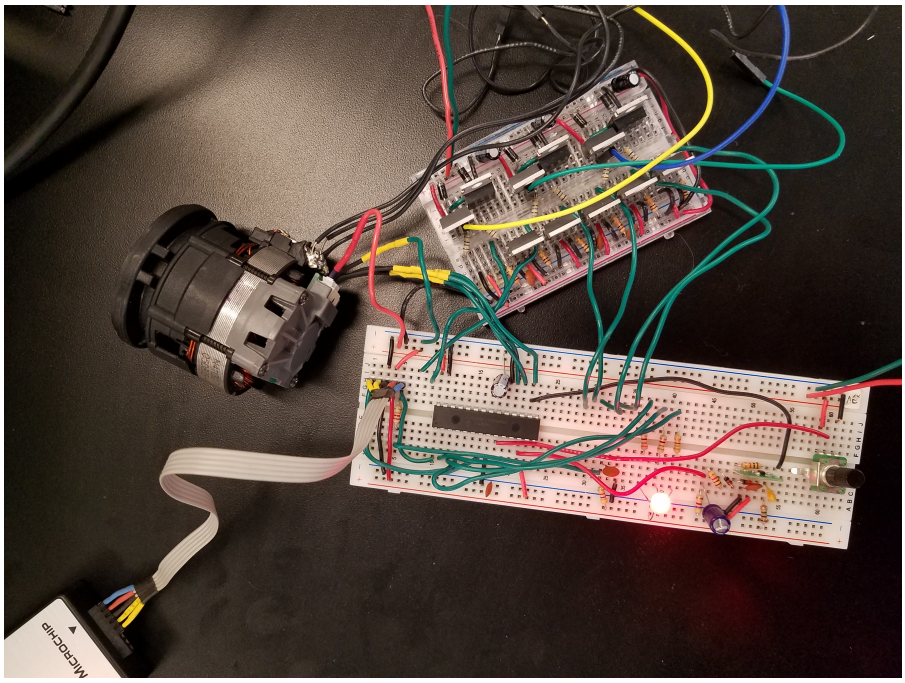

Figure 3: Entire Analog system including Motor

3.2 Detailed Block Diagram

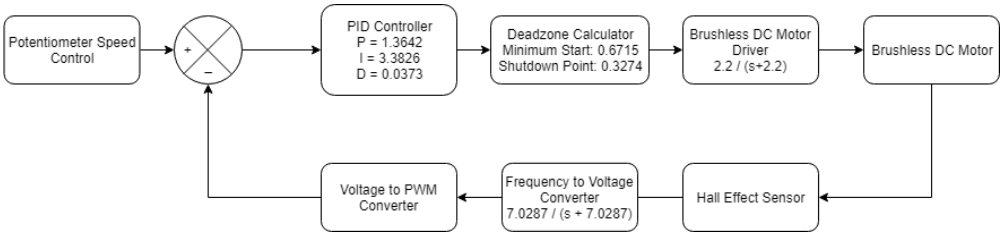

Figure 4: Detailed Block Diagram

3.2.1 Potentiometer Speed Control

The potentiometer speed control is simply a dial to set the desired motor output speed. The potentiometer is used as a variable voltage divider. It's voltage is then read by an ADC of the dsPIC, and the following equation is used to convert the 16 bit adc value to a pwm value:

$$DutyCycle = 0.00011455 * ADCValue + 0.3275$$

3.2.2 PID Controller

The PID controller calculates an error value and applies a correction based on proportional, integral, and derivative terms. Thus, the system receives a correction that applies accurate and response correction.

3.2.3 Brushless DC Motor Driver

The dsPIC is connected to the motor driver using 6 pins for 3 phase PWM. Each phase has a high and a low, thus the need for 6 wires. The motor driver consists of MOSFETS and MOSFET gate drivers. These components transmit a 3 phase PWM signal to the motor. The motor takes in this data and spins at a certain speed. The hall effect sensor tracks every time the motor completes a half a revolution.

3.2.4 Brushless DC Motor

Taken from a DeWalt electric drill, this motor is powered by 3 wires. Internally these wires are connected in a Y configuration,. These lines can be driven in a certain sequence that produces a rotation in the desired direction. The more current flowing through the coils produces a higher torque, while a higher voltage is needed for a higher speed.

3.2.5 Hall Effect Sensor

Built onto the back (opposite from the drive shaft output) is the hall effect sensor array. These allow a microcontroller to read the rotational position of the output shaft so the correct signals can be driven to the 3 input phases. Each time to motor rotates 1 half turn, it outputs a signal. The rotations per second (RPS) of the motor can be derived by dividing the frequency of the hall effect sensor by two.

3.2.6 Frequency to Voltage Converter

The frequency of the hall effect sensor must then be input into microcontroller so the speed of the motor can be derived. In order to achieve this, the frequency of the motor is converted to a voltage. This voltage is then read by an ADC on the microcontroller and an equation is used to convert the frequency into a desired PWM value.

3.2.7 Voltage to PWM Converter

The voltage from the frequency to voltage converter is then input into the following equation and a decimal PWM value is derived:

$$DutyCycle = -0.016*voltage^4 - 0.027*voltage^3 + 0.170*voltage^2 + 0.1553*voltage + 0.341$$

3.3 Highlighted Devices

- dsPIC33EV256GM102 Microcontroller
- Brushless DC Motor
- TC4422A MOSFET Gate Driver
- FQP27P06 PMOS
- FQP30N06L NMOS
- MIC1555 555timer

3.4 dsPIC33EV256GM102

A 16-bit microcontroller that can operate at up to 70 MIPS. As this project is in conjunction with the Rowan University Formula Electric Clinic, a versatile microcontroller was chosen that has many more features than were needed in this motor controller. Notable features are:

- Internal 7.37 MHz clock accurate to within 1%.
- Built-in clock switching with PLL
- Low-power modes, operating down to 50 μA
- 6 PWM outputs with 7.14 ns precision
- 12-bit ADC up to 500 ksps
- 5 16-bit timers, which can be paired to create 2 32-bit timers
- 2 UART up to 6.25 Mbps
- 2 SPI up to 15 MHz
- 1 IIC up to 1 Mbaud
- 2 SENT modules
- 1 CAN module with 32 buffers, 16 filters, and 3 masks

The three dual-output PWM modules are designed to be used to control complex motor systems like BLDC motors. This PWM module has a resolution of 7.14 ns, the chosen operation mode was a 10-bit output at 133 kHz. If a higher frequency was desired, then there would be a loss of precision.

The Enhanced CAN (ECAN) peripheral provides 32 receive buffers, of which 8 can be used for transmission. The different filters and masks can be used to sort the incoming messages into different buffers for easy processing.

3.5 Brushless DC Motor

Taken from a DeWalt electric drill, this motor is powered by 3 wires. Internally these wires are connected in a Y configuration, as shown in figure 5. These lines can be driven in a certain sequence that produces a rotation in the desired direction. Figure 6 shows an example sequence of powering the coils. The more current flowing through the coils produces a higher torque, while a higher voltage is needed for a higher speed.

Over time, the amount of current flowing through an inductor increases while voltage is applied. By controlling the PWM at 133 kHz, the current fluctuates less than if the current was slowly toggled. A higher frequency could be used at the loss of resolution.

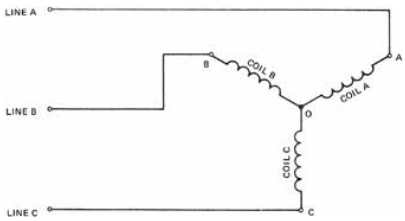

Figure 5: BLDC Motor Y configuration

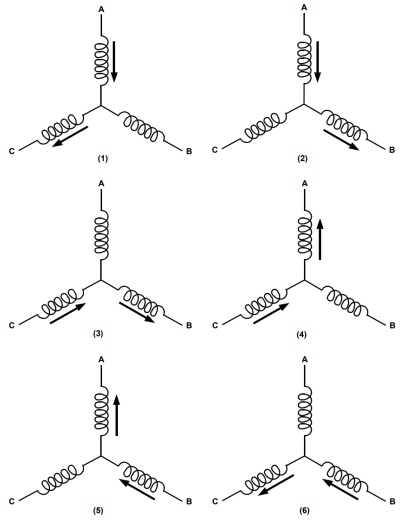

Figure 6: BLDC Motor Winding Energization

Built onto the back (opposite from the drive shaft output) is the hall effect sensor array. These allow a microcontroller to read the rotational position of the output shaft so the correct signals can be driven to the 3 input phases.

3.6 TC4422A MOSFET Gate Driver

To achieve the full amperage of the MOSFETs for driving the motor, a higher voltage than the microcontroller can output must be used. The gate driver performs this action by stepping up the voltage from the microcontroller to the 15 V supply for the motors. It is also able to source up to 10 A of current, which provides fast (¡20 ns) switch times for MOSFETs, as typically the larger the MOSFET, the higher the gate capacitance. It is not good to leave a MOSFET partially on, as that increases its internal resistance while it is still conducting, which generates excess heat.

There is a limitation of max 18 V supply to the gate drivers. Should higher motor voltages be needed in the future (which require higher gate voltages to be applied to the MOSFETs) an alternate circuit can be created. This alternate circuit uses inductors to control the voltages at the MOSFET gates. Figure 7 shows an example of what this circuit could look like.

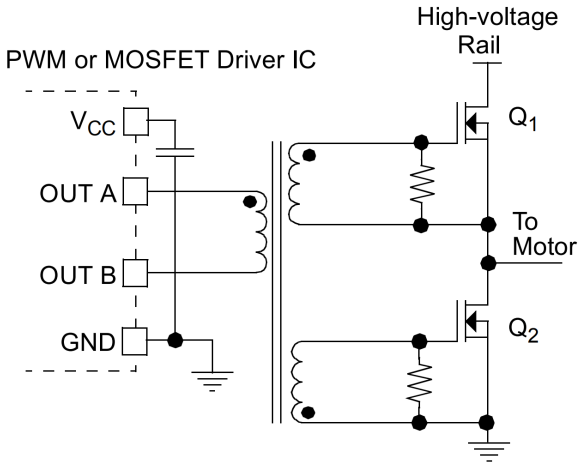

Figure 7: Double-Ended Gate Drive Transformer

3.7 FQP27P06 PMOS

Allows a motor phase to get driven to 15 V. The source is connected to 15 V, the gate is connected to the output of a gate driver, and the drain is connected to a single wire of the motor. When not activated, the drain floats. To activate the MOSFET, the V_{GS} must be less than -4 V, but for the full amperage rating of the MOSFET, -7 V is recommended.

The drain is also connected to the drain of the NMOS. The idea is that either the PMOS or the NMOS is activated at once.

3.8 FQP30N06L NMOS

Allows a motor phase to get driven to ground. The source is connected to ground, the gate is connected to the output of a gate driver, and the drain is connected to a single wire of the motor. When not activated, the drain floats. To activate the MOSFET, the V_{GS} must be greater than 2.5 V, but for the full amperage rating of the MOSFET, -7 V is recommended.

3.9 MIC1555 555timer

A 5 pin 555 timer that provides the logic for creating simple RC timer or oscillator circuits. In this case the MIC1555 recieves a square wave from the encoder where it may may be triggered by the AC-coupled falling edge. The RC time constant of the input capacitor and pull-up resistor is less than the output pulse time, to prevent multiple output pulses. A diode across the timing resistor provides a fast reset at the end of the positive timing pulse. The output goes through a low-pass filter to remove noise so the final output can be received as a three phase PWM voltage.

4 SYSTEM DESIGN THEORY

4.1 Closed Loop System Design

In order to ensure that the speed output of the brushless DC motor was accurate, a closed loop system utilizing a PID controller was built and tested. A visualization can be seen in Figure 4 above. A potentiometer was used to set the desired PWM value, and thus the speed of the brushless motor. This PWM value then goes into a summing block, where the error is derived (Desired PWM Value - Current PWM Value). The error is then input into a PID controller, where they system is corrected to provide an accurate PWM value to send to the brushless DC motor driver. This new PWM value is then sent to the motor driver, which results in the motor spinning. The back of the motor contains a hall effect sensor, where the speed of the motor is read as the frequency of a rotation. This frequency is then converted into a voltage that can be read by the microcontroller. The microcontroller then converts the voltage into a PWM value, that is plugged into the summing value to provide the current PWM value.

4.1.1 Simulink Simulation

The entire system was simulated in Simulink in order to test P, I, and D values prior to testing with the motors. The configuration of the Simulink model can be seen below in Figure 8.

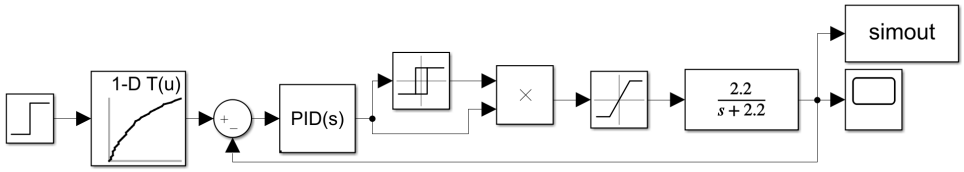

Figure 8: Simulink Configuration

In order to determine initial P, I, and D values, the PID tuner was utilized to provide these values. The PID tuner was able to adjust the values based on the response time and transient response of the system. The response of the PID controller can be seen below in FIgure 9.

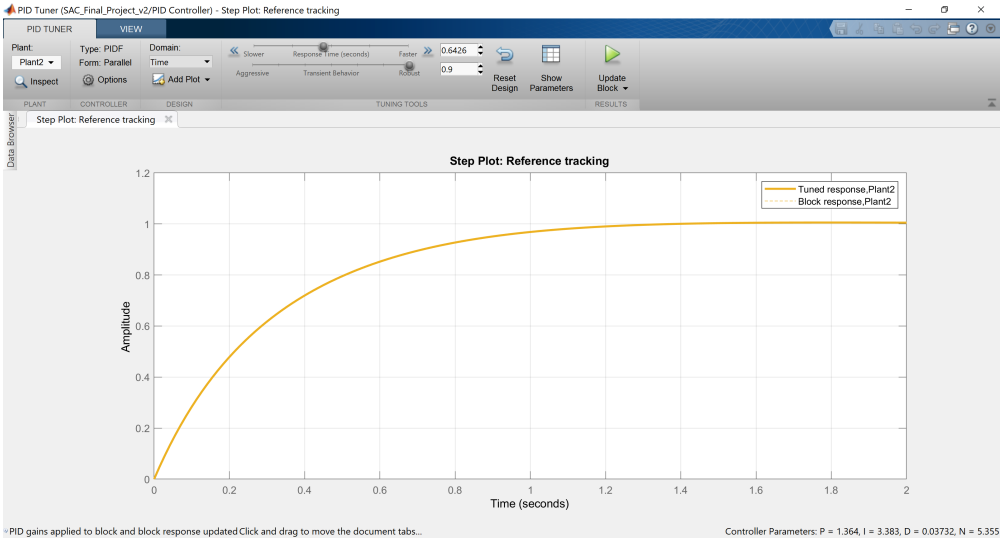

Figure 9: Simulink Tuner Step Response

The tuned PID values were as follows:
P = 1.3642407803804
I = 3.38261365385595
D = 0.0373242127602274

4.1.2 Tuning and Testing

After significant testing, it was found the the P, I, and D values found using Simulink were extremely effective and accurate. Changes to those values during testing only had a negative affect on the system.

4.1.3 Final Design

After simulation and testing were completed, the tuned PID values were as follows:
P = 1.3642407803804
I = 3.38261365385595
D = 0.0373242127602274
System specifications can be seen in section 2.

5 Code Review

5.1 Initialization

On startup the 12 bit ADC module is initialized with two input channels. One channel is to read the desired speed input dial, and the other is to read the feedback from the frequency to voltage converter. The PWM module is initialized as a 135 kHz (10 bit resolution) PWM. This provides the output to the motor driver that will power the motor. A 1 kHz timer is also initialized to provide continuous PID calculations at a fixed rate.

5.2 Sensor Acquisition

In order to determine the motors current speed (RPS), a hall effect was utilized to determine the frequency at which the motor completes a rotation. The frequency of the hall effect sensor is then input into microcontroller so the speed of the motor can be derived. In order to achieve this, the frequency of the motor is converted to a voltage. This voltage is then read by an ADC on the microcontroller. The voltage from the frequency to voltage converter is then input into the following equation and a decimal PWM value is derived:

$$DutyCycle = -0.016*voltage^4 - 0.027*voltage^3 + 0.170*voltage^2 + 0.1553*voltage + 0.341$$

5.2.1 Signal Conditioning

The frequency to voltage converter contains a lowpass filter which filters out any outliers in the motor frequency. Based on the motors top speed, the frequency had a maximum value of 450 Hz, and thus a lowpass filter was designed to filter any frequencies greater than 450 Hz.

5.3 Error Calculation

Error is found by checking the difference between the desired PWM duty cycle set by the voltage from the potentiometer to the PIC and the actual PWM duty cycle detected as a feedback voltage from the frequency to voltage converter, both of which are input to the PIC where the PID makes the relevant adjustment to the voltage output going

through the motor. The new duty cycle is calculated using the equation presented in Section 3.2.7. This corrects the error detected by the PIC.

5.4 Control Calculation

The calculations started with the integral calculation, the previous integral is added to a new integral value that represents just the span of time since the last update. It uses the change in the error multiplied by the Ki value. To prevent wind-up where if there is continuous error the integral value will keep accumulating, its value is capped at 1000. The proportional calculation simply multiplies Kp by the new error. The derivative control multiplies the change in error by Kd, all divided by the sampling time. The output value of the PID controller is the sum of the three parts. For use in the next pass through the PID calculations, the new integral value is stored, and the new error value is stored.

5.5 Actuation

After the new duty cycle is calculated, it is scaled back up to reach the full range of the PWM duty cycle output (0-1023). This value can be updated as often as the processor will allow. This duty cycle cannot go over 79%, otherwise the MOSFETs will not have enough off time to prevent the motor from drawing too much power through its inductor. Limits are added into the code to prevent this situation from happening. The motor has a wide range of operation which allows it to rotate slowly, however it has a high turn on point to get the output shaft spinning. This is factored into the simulation and the code to prevent the PID controller from trying to start the motor at too low of a duty cycle.

In addition to the PID controller that runs off the 1 kHz timer, a control loop that checks the rotation angle of the motor is used to adjust which MOSFETs get switched on by the PWM signal. This does not affect the duty cycle of the signal, rather it actuates them in the right sequence to produce the spinning motion. In fact, the duty cycle can be updated separately from this loop, providing isolated control.

6 Design Files

6.1 Schematics

6.1.1 3x Motor Driver

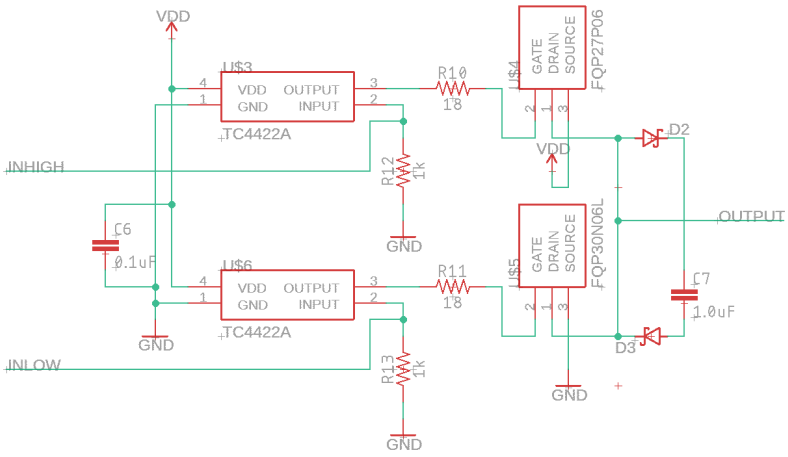

Figure 10: Motor Driver Schematic

6.1.2 Motor Controller

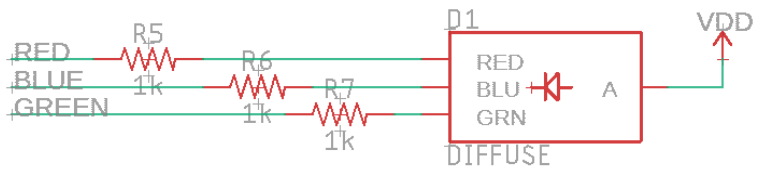

Figure 11: Motor controller schematic

6.1.3 Speed Input

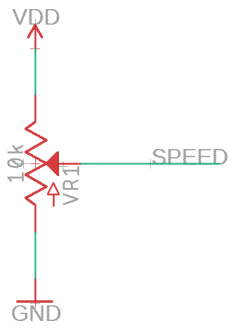

Figure 12: Speed Input Schematic

6.1.4 Frequency to Voltage Converter

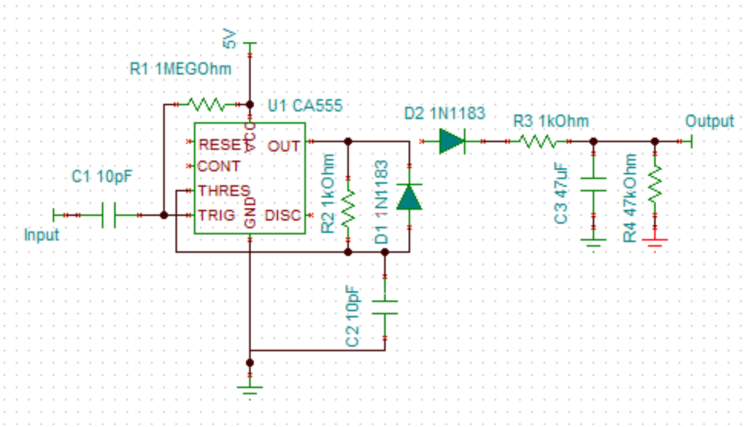

Figure 13: Frequency to Voltage Converter

6.2 Bill of Materials

The parts list is broken down into multiple lists to categorize the parts used for easy grouping. These lists go with the corresponding schematics.

6.2.1 Motor Driver

- 6x TC4422A MOSFET Gate Driver
- 3x FQP27P06 PMOS
- 3x FQP30N06L NMOS
- 6x 1N5817 Schottky Diode
- 6x 18 Ω Resistor
- 6x 10 $k\Omega$ Resistor
- 3x 1 μF Capacitor
- 3x 0.1 μF Capacitor

6.2.2 Microcontroller

- 1x dsPIC33EV256GM102 Microcontroller
- 1x 22 $k\Omega$ Resistor
- 1x 4.7 $k\Omega$ Resistor
- 1x Pushbutton
- 1x 10 μF Capacitor
- 2x 0.1 μF Capacitor

6.2.3 Motor Controller

- 1x 3-Phase Brushless DC Motor with Hall Effect Sensors
- 1x RGB LED
- 3x 330 Ω Resistor

6.2.4 Speed Input

- 1x 10 $k\Omega$ Linear Potentiometer

6.2.5 Frequency to Voltage Converter

- 1x MIC1555 555timer
- 1x 10 ρF Capacitor
- 1x 1 Meg Ω Resistor
- 2x 1 $k\Omega$ Resistor
- 2x 1n4148 Diode
- 1x 47 μF Capacitor
- 1x 47 $k\Omega$ Resistor

6.3 Code

6.3.1 Main Code File

main.c file

```
/**
  Generated main.c file from MPLAB Code Configurator

  @Company
    Microchip Technology Inc.

  @File Name
    main.c

  @Summary
    This is the generated main.c using PIC24 / dsPIC33 / PIC32MM MCUs.

  @Description
    This source file provides main entry point for system intialization and
    application code development.
    Generation Information :
        Product Revision  :  PIC24 / dsPIC33 / PIC32MM MCUs - 1.75.1
        Device            :  dsPIC33EV256GM102
    The generated drivers are tested against the following:
        Compiler          :  XC16 v1.35
        MPLAB             :  MPLAB X v5.05
*/

/*
    (c) 2016 Microchip Technology Inc. and its subsidiaries. You may use this
    software and any derivatives exclusively with Microchip products.

    THIS SOFTWARE IS SUPPLIED BY MICROCHIP "AS IS". NO WARRANTIES, WHETHER
    EXPRESS, IMPLIED OR STATUTORY, APPLY TO THIS SOFTWARE, INCLUDING ANY
    IMPLIED
    WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A
    PARTICULAR PURPOSE, OR ITS INTERACTION WITH MICROCHIP PRODUCTS,
    COMBINATION
    WITH ANY OTHER PRODUCTS, OR USE IN ANY APPLICATION.
```

```
32
33      IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE,
34      INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND
35      WHATSOEVER RELATED TO THE SOFTWARE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS
36      BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE
37      FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS
        IN
38      ANY WAY RELATED TO THIS SOFTWARE WILL NOT EXCEED THE AMOUNT OF FEES, IF
        ANY,
39      THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THIS SOFTWARE.
40
41      MICROCHIP PROVIDES THIS SOFTWARE CONDITIONALLY UPON YOUR ACCEPTANCE OF
        THESE
42      TERMS.
43  */
44
45  /**
46    Section: Included Files
47  */
48  #include "mcc_generated_files/system.h"
49  #include "mcc_generated_files/pwm.h"
50  #include "mcc_generated_files/pin_manager.h"
51  #include "mcc_generated_files/tmr2.h"
52  #include "mcc_generated_files/adc1.h"
53  #include "mcc_generated_files/ecan1.h"
54  #include "mcc_generated_files/clock.h"
55
56  /*
57                              Main application
58   */
59  //#define FCY _XTAL_FREQ / 2
60  //#define BAUDRATE 115200
61  //#define BRGVAL ((FCY / BAUDRATE) / 16) - 1
62
63  void initUART() {
64      // Reset UART
65      U1MODE = 0;
66  //    U1BRG = BRGVAL;
67
68      // Pin connections
69      RPINR18bits.U1RXR = 38;
70  //    RPOR0bits.RP20R = 1;
71      RPOR1bits.RP37R = 1;
72
73      // Enable receive interrupt
74      IPC2bits.U1RXIP = 0x01;
75      IFS0bits.U1RXIF = 0;
76      IEC0bits.U1RXIE = 1;
77
78      // Turn on UART
79      U1MODEbits.UARTEN = 1;
80      U1STAbits.UTXEN = 1;
81  }
82
83  int main(void)
84  {
85      CNPDBbits.CNPDB4 = 1;
```

```
86      // initialize the device
87      SYSTEM_Initialize();
88  //    initUART();
89      printf("start\r\n");
90
91      // Set the status LEDs
92      Stat1_SetLow();
93      Stat2_SetHigh();
94      Stat3_SetHigh();
95
96      // Start the PWM
97      MDC = 0;
98      PWM_FaultInterruptStatusClear(PWM_GENERATOR_1);
99      PWM_FaultInterruptStatusClear(PWM_GENERATOR_2);
100     PWM_FaultInterruptStatusClear(PWM_GENERATOR_3);
101
102     // Track the rotation of the motor
103     int graycode = 0;
104     uint16_t h1, h2, h3;
105
106     // Track the RPS of the motor
107     bool first = false;
108     double timerPeriod = 1000; // ns
109     double desiredRPS = 100;
110
111     // Initialize CAN
112 //    uCAN1_MSG msg;
113 //    ECAN1_ReceiveEnable();
114
115     while (1)
116     {
117         // Read any available CAN message
118 //        bool received = ECAN1_receive(&msg);
119 //        if (received) {
120 //            desiredRPS = msg.frame.data0;
121 //        }
122
123         // Read the hall effect sensor
124         h1 = Hall1_GetValue();
125         h2 = Hall2_GetValue();
126         h3 = Hall3_GetValue();
127         graycode = (h3 << 2) | (h2 << 1) | h1;
128
129         // Enable the PWM lines based on the position of the motor
130         switch (graycode) {
131             case 1:
132                 // Recalculate PWM duty cycle to run at a desired speed
133 //                if (first) {
134 //                    first = false;
135 //
136 //                    // Set the speed based on the time taken to loop once
137 //                    double period = TMR2_SoftwareCounterGet();
138 //                    TMR2_SoftwareCounterClear();
139 //                    // Twice as fast due to two cycles of the hall effect
    sensor per revolution
140 //                    double calculatedRPS = 5000.0 / period;
141 //                    if (calculatedRPS < 1) {
```

```
142 //                          calculatedRPS = desiredRPS;
143 //                      }
144 //                      double differencePercent = ((desiredRPS * 1.0) /
    calculatedRPS) - 1.0;
145 //                      uint32_t newValue = MDC + ((MDC * 0.01) *
    differencePercent);
146 //                      if (newValue > 0x37F) {
147 //                          newValue = 0x2FF;
148 //                      } else if (newValue < 0x0FF) {
149 //                          newValue = 0x0FF;
150 //                      }
151 //                      MDC = newValue;
152 //                  }
153                     Stat2_SetLow();
154                     Stat3_SetHigh();
155                     PWM_OverrideHighEnable(PWM_GENERATOR_1); // Yellow float
156                     PWM_OverrideLowEnable(PWM_GENERATOR_1);
157                     PWM_OverrideHighEnable(PWM_GENERATOR_2); // Green low
158                     PWM_OverrideLowDisable(PWM_GENERATOR_2);
159                     PWM_OverrideHighDisable(PWM_GENERATOR_3); // Blue high
160                     PWM_OverrideLowEnable(PWM_GENERATOR_3);
161                     break;
162                 case 5:
163                     Stat2_SetHigh();
164                     Stat3_SetHigh();
165                     PWM_OverrideHighEnable(PWM_GENERATOR_1); // Yellow low
166                     PWM_OverrideLowDisable(PWM_GENERATOR_1);
167                     PWM_OverrideHighEnable(PWM_GENERATOR_2); // Green float
168                     PWM_OverrideLowEnable(PWM_GENERATOR_2);
169                     PWM_OverrideHighDisable(PWM_GENERATOR_3); // Blue high
170                     PWM_OverrideLowEnable(PWM_GENERATOR_3);
171                     break;
172                 case 4:
173                     Stat2_SetHigh();
174                     Stat3_SetHigh();
175                     PWM_OverrideHighEnable(PWM_GENERATOR_1); // Yellow low
176                     PWM_OverrideLowDisable(PWM_GENERATOR_1);
177                     PWM_OverrideHighDisable(PWM_GENERATOR_2); // Green high
178                     PWM_OverrideLowEnable(PWM_GENERATOR_2);
179                     PWM_OverrideHighEnable(PWM_GENERATOR_3); // Blue float
180                     PWM_OverrideLowEnable(PWM_GENERATOR_3);
181                     break;
182                 case 6:
183                     first = true;
184 //                  ADC1_SamplingStop();
185                     Stat2_SetHigh();
186                     Stat3_SetHigh();
187                     PWM_OverrideHighEnable(PWM_GENERATOR_1); // Yellow float
188                     PWM_OverrideLowEnable(PWM_GENERATOR_1);
189                     PWM_OverrideHighDisable(PWM_GENERATOR_2); // Green high
190                     PWM_OverrideLowEnable(PWM_GENERATOR_2);
191                     PWM_OverrideHighEnable(PWM_GENERATOR_3); // Blue low
192                     PWM_OverrideLowDisable(PWM_GENERATOR_3);
193                     break;
194                 case 2:
195                     Stat2_SetHigh();
196                     Stat3_SetHigh();
```

```
197                 PWM_OverrideHighDisable(PWM_GENERATOR_1); // Yellow high
198                 PWM_OverrideLowEnable(PWM_GENERATOR_1);
199                 PWM_OverrideHighEnable(PWM_GENERATOR_2); // Green float
200                 PWM_OverrideLowEnable(PWM_GENERATOR_2);
201                 PWM_OverrideHighEnable(PWM_GENERATOR_3); // Blue low
202                 PWM_OverrideLowDisable(PWM_GENERATOR_3);
203                 break;
204             case 3:
205                 Stat2_SetHigh();
206                 Stat3_SetHigh();
207                 PWM_OverrideHighDisable(PWM_GENERATOR_1); // Yellow high
208                 PWM_OverrideLowEnable(PWM_GENERATOR_1);
209                 PWM_OverrideHighEnable(PWM_GENERATOR_2); // Green low
210                 PWM_OverrideLowDisable(PWM_GENERATOR_2);
211                 PWM_OverrideHighEnable(PWM_GENERATOR_3); // Blue float
212                 PWM_OverrideLowEnable(PWM_GENERATOR_3);
213                 break;
214             default:
215                 first = false;
216                 MDC = 0x2FF;
217                 Stat2_SetHigh();
218                 Stat3_SetLow();
219                 PWM_OverrideHighEnable(PWM_GENERATOR_1); // All float
220                 PWM_OverrideLowEnable(PWM_GENERATOR_1);
221                 PWM_OverrideHighEnable(PWM_GENERATOR_2);
222                 PWM_OverrideLowEnable(PWM_GENERATOR_2);
223                 PWM_OverrideHighEnable(PWM_GENERATOR_3);
224                 PWM_OverrideLowEnable(PWM_GENERATOR_3);
225         }
226     }
227     return 1;
228 }
229 /**
230  End of File
231 */
```

6.3.2 PID Code File

```
1
2  /**
3    TMR2 Generated Driver API Source File
4
5    @Company
6      Microchip Technology Inc.
7
8    @File Name
9      tmr2.c
10
11   @Summary
12     This is the generated source file for the TMR2 driver using PIC24 /
       dsPIC33 / PIC32MM MCUs
13
14   @Description
15     This source file provides APIs for driver for TMR2.
16     Generation Information :
17         Product Revision  :  PIC24 / dsPIC33 / PIC32MM MCUs - 1.75.1
```

```
18          Device            :  dsPIC33EV256GM102
19      The generated drivers are tested against the following:
20          Compiler          :  XC16 v1.35
21          MPLAB             :  MPLAB X v5.05
22  */
23
24  /*
25      (c) 2016 Microchip Technology Inc. and its subsidiaries. You may use this
26      software and any derivatives exclusively with Microchip products.
27
28      THIS SOFTWARE IS SUPPLIED BY MICROCHIP "AS IS". NO WARRANTIES, WHETHER
29      EXPRESS, IMPLIED OR STATUTORY, APPLY TO THIS SOFTWARE, INCLUDING ANY
        IMPLIED
30      WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A
31      PARTICULAR PURPOSE, OR ITS INTERACTION WITH MICROCHIP PRODUCTS,
        COMBINATION
32      WITH ANY OTHER PRODUCTS, OR USE IN ANY APPLICATION.
33
34      IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE,
35      INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND
36      WHATSOEVER RELATED TO THE SOFTWARE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS
37      BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE
38      FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS
        IN
39      ANY WAY RELATED TO THIS SOFTWARE WILL NOT EXCEED THE AMOUNT OF FEES, IF
        ANY,
40      THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THIS SOFTWARE.
41
42      MICROCHIP PROVIDES THIS SOFTWARE CONDITIONALLY UPON YOUR ACCEPTANCE OF
        THESE
43      TERMS.
44  */
45
46  /**
47    Section: Included Files
48  */
49
50  #include <xc.h>
51  #include "tmr2.h"
52  #include "pwm.h"
53  #include "adc1.h"
54  #include "clock.h"
55  #define FCY _XTAL_FREQ/2
56  #include <libpic30.h>
57  #include <stdio.h>
58
59  /**
60    Section: Data Type Definitions
61  */
62
63  /** TMR Driver Hardware Instance Object
64
65    @Summary
66      Defines the object required for the maintainence of the hardware instance.
67
68    @Description
69      This defines the object required for the maintainence of the hardware
```

```
70      instance. This object exists once per hardware instance of the peripheral.
71
72    Remarks:
73      None.
74  */
75
76  typedef struct _TMR_OBJ_STRUCT
77  {
78      /* Timer Elapsed */
79      bool                                                    timerElapsed;
80      /*Software Counter value*/
81      uint8_t                                                 count;
82
83  } TMR_OBJ;
84
85  static TMR_OBJ tmr2_obj;
86
87  /**
88    Section: Driver Interface
89  */
90
91  void TMR2_Initialize (void)
92  {
93      //TMR3 0;
94      TMR3 = 0x00;
95      //PR3 1;
96      PR3 = 0x01;
97      //TMR2 0;
98      TMR2 = 0x00;
99      //Period = 0.0010000036 s; Frequency = 69093750 Hz; PR2 3558;
100     PR2 = 0xDE6;
101     //TCKPS 1:1; T32 32 Bit; TON enabled; TSIDL disabled; TCS FOSC/2; TGATE
        disabled;
102     T2CON = 0x8008;
103
104
105     IFS0bits.T3IF = false;
106     IEC0bits.T3IE = true;
107
108     tmr2_obj.timerElapsed = false;
109
110 }
111
112
113 void __attribute__ ( ( interrupt, no_auto_psv ) ) _T3Interrupt (  )
114 {
115     /* Check if the Timer Interrupt/Status is set */
116
117     //***User Area Begin
118
119     // ticker function call;
120     // ticker is 1 -> Callback function gets called everytime this ISR
        executes
121         TMR2_CallBack();
122
123     //***User Area End
124
```

```
125     tmr2_obj.count++;
126     tmr2_obj.timerElapsed = true;
127     IFS0bits.T3IF = false;
128 }
129
130
131
132
133 void TMR2_Period32BitSet( uint32_t value )
134 {
135     /* Update the counter values */
136     PR2 = (value & 0x0000FFFF);
137     PR3 = ((value & 0xFFFF0000)>>16);
138 }
139
140 uint32_t TMR2_Period32BitGet( void )
141 {
142     uint32_t periodVal = 0xFFFFFFFF;
143
144     /* get the timer period value and return it */
145     periodVal = (((uint32_t)PR3 <<16) | PR2);
146
147     return( periodVal );
148
149 }
150
151 void TMR2_Counter32BitSet( uint32_t value )
152 {
153     /* Update the counter values */
154    TMR3HLD = ((value & 0xFFFF0000)>>16);
155    TMR2 = (value & 0x0000FFFF);
156
157 }
158
159 uint32_t TMR2_Counter32BitGet( void )
160 {
161     uint32_t countVal = 0xFFFFFFFF;
162     uint16_t countValUpper;
163     uint16_t countValLower;
164
165     countValLower = TMR2;
166     countValUpper = TMR3HLD;
167
168     /* get the current counter value and return it */
169     countVal = (((uint32_t)countValUpper<<16)| countValLower );
170
171     return( countVal );
172
173 }
174
175 // Performs PID calculations
176 void __attribute__ ((weak)) TMR2_CallBack(void)
177 {
178     // PID configuration
179     double Kp = 1.3642407803804;
180     double Ki = 3.38261365385595;
181     double Kd = 0.0373242127602274;
```

```
182     double tSampling = 0.001; // 1 kHz
183     double integralMin = -1000;
184     double integralMax = 1000;
185     double switchOn = 0.671554252;
186     double switchOff = 0.327468231;
187     uint16_t pwmMax = 0x3FF;
188 
189     // Get the dial
190     ADC1_ChannelSelectSet(ADC1_DIAL);
191     ADC1_SamplingStart();
192     __delay_us(100);
193     uint16_t dial = ADC1_Channel0ConversionResultGet();
194 
195     // Get the speed
196     ADC1_ChannelSelectSet(ADC1_SPEED);
197     ADC1_SamplingStart();
198     __delay_us(100);
199     uint16_t speed = ADC1_Channel0ConversionResultGet();
200 
201     // Normalize the input signals
202     double signalDial = (0.00011455 * dial) + 0.327468230694037;
203 //    MDC = signalDial * 0x3ff;
204     double speedVoltage = 0.001220703125 * speed;
205     double signalSpeed = (-0.015504900608 * (speedVoltage * speedVoltage *
        speedVoltage * speedVoltage)) +
206             (-0.02650420896 * (speedVoltage * speedVoltage * speedVoltage)) +
207             (0.170362883294 * (speedVoltage * speedVoltage)) +
208             (0.1552907996399 * speedVoltage) +
209             0.341281524152;
210 
211     // Calculate the error
212     double errorNew = signalDial - signalSpeed;
213 
214     // PID calculations
215     static double errorOld = 0;
216     static double integralOld = 0;
217     double integralNew = integralOld + ((Ki * tSampling * (errorNew - errorOld
        )) / 2);
218     if ((integralNew > integralMax) || (integralNew < integralMin)) {
219         integralNew = integralOld;
220     }
221     double pidOutput = (Kp * errorNew) + ((Kd * (errorNew - errorOld)) /
        tSampling) + integralNew;
222     integralOld = integralNew;
223     errorOld = errorNew;
224 
225     // Handle the switch on and switch off points of the motor
226     if (pidOutput > switchOn) {
227         // PID output doesn't change, motor will always spin
228     } else if (pidOutput < switchOff) {
229         // Turn off PWM, motor can't actually spin
230         pidOutput = 0;
231     } else if (MDC > 0) {
232         // PID output doesn't change, motor already spinning
233     } else {
234         // Motor can't spin yet, don't turn on PWM
235         pidOutput = 0;
```

```
236     }
237
238     // Limit the PID output and set the PWM output value
239     if (pidOutput <= 0) {
240         MDC = 0;
241     } else if (pidOutput > 0.8) {
242         MDC = 0x32F;
243     } else {
244         MDC = pidOutput * pwmMax;
245     }
246
247     // Logging
248     static int iterCounter = 0;
249     if (iterCounter > 100) {
250         iterCounter = 0;
251         printf("e %f pid %f mdc %d\r\n", errorNew, pidOutput, MDC);
252     } else {
253         iterCounter++;
254     }
255 }
256
257 void TMR2_Start( void )
258 {
259     /* Reset the status information */
260     tmr2_obj.timerElapsed = false;
261
262     /*Enable the interrupt*/
263     IEC0bits.T3IE = true;
264
265     /* Start the Timer */
266     T2CONbits.TON = 1;
267 }
268
269 void TMR2_Stop( void )
270 {
271     /* Stop the Timer */
272     T2CONbits.TON = false;
273
274     /*Disable the interrupt*/
275     IEC0bits.T3IE = false;
276 }
277
278 bool TMR2_GetElapsedThenClear(void)
279 {
280     bool status;
281
282     status = tmr2_obj.timerElapsed;
283
284     if(status == true)
285     {
286         tmr2_obj.timerElapsed = false;
287     }
288     return status;
289 }
290
291 int TMR2_SoftwareCounterGet(void)
292 {
```

```
293     return tmr2_obj.count;
294 }
295 
296 void TMR2_SoftwareCounterClear(void)
297 {
298     tmr2_obj.count = 0;
299 }
300 
301 /**
302  End of File
303 */
```

6.4 Other Files

The remainder of the code can be view at the following BitBucket Repository: bitbucket.org/Gliderman/sac-motor-controller/src/master/