Final Project: PIC Brushless DC Motor Controller

Nate Hoffman and DJ Stahlberger
Rowan University

December 22, 2018

1 Design Overview

The goal of the final project was to design a system that took in real world data, analyzed that data, controlled something as a result of that data, and communicate with either a computer or through wireless means. The system that was created was a Motor Controller for a brushless DC (BLDC) motor capable of communicating over a Controller Area Network (CAN) using Microchip's dsPIC33EV256GM102 and CAN transceivers.

On a top level view, the system reads a potentiometer value and the motor will spin at a speed consistent with that value. This works with the motor under load so that it can actually be used. This can be achieved due to the nature of the closed loop system under which the motor is operating. The motor reports its speed back to the microcontroller and there it is determined whether the motor needs to spin faster or slower. This will be explained in more detail later in the report.

1.1 Design Features

Due to CAN and closed loop system feedback control, here are the design features:

- Closed loop speed control of a BLDC motor
- CAN bus communication
- Variable speed control with a potentiometer

1.2 Featured Applications

- Electric Vehicles
- Industrial Motor control

- CAN communication in automotive use
- Potential Wireless Communication

1.3 Design Resources

Here are a few links to the GitHub repository and a few useful documents:

- GitHub Code Repository
- dsPIC33EV256GM102 Microcontroller DataSheet
- TC4422A MOSFET Gate Driver DataSheet
- FQP27P06 PMOS Datasheet
- FQP30N06L NMOS Datasheet
- MCP2561 CAN Transceiver Datasheet
- Microchip AN885 - Brushless DC (BLDC) Motor Fundamentals
- Microchip AN898 - Determining MOSFET Driver Needs for Motor Drive Applications

1.4 Design Inspiration

The inspiration for this project stems from the Rowan University Formula Electric Clinic. That clinic is student run with the goal of building a fully-functional race ready formula style electric race car. The timeline is 2 years starting in the Fall of 2018. The clinic is split into two parts, a mechanical side and an electrical side. This design was created with research intent towards the electrical side. Since the car is being built from the ground up, some of the parts and components need to be tested in house. This design is one of those test models and attempts to see if a motor controller could be created in house by the members of the team. Hopefully this design will provide useful information that can be used by members of the Formula Electric team.

1.5 Block Diagram

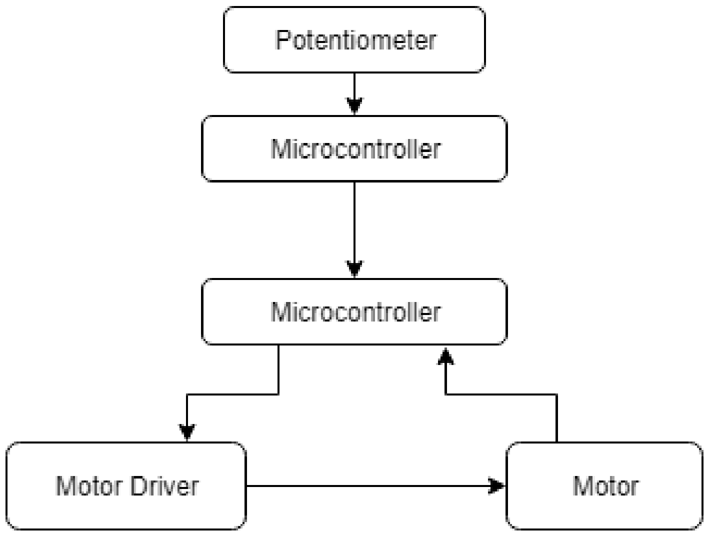

Figure 1: Top level block diagram of the system

1.6 Board Image

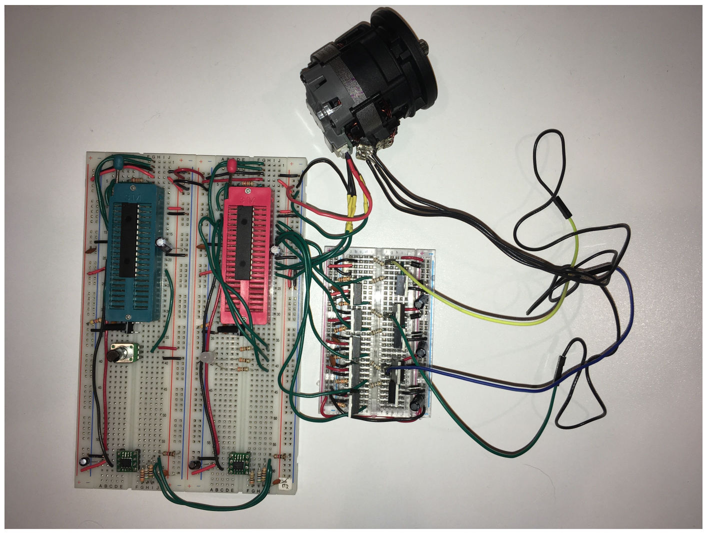

Figure 2: The entire closed loop system

2 Key System Specifications

Table 1: dsPIC33EV256GM102 Specifications

Symbol	Characteristic	Min.	Typ.	Max.	Units
V_{DD}	Voltage Range	4.5	-	5.5	V
I_{DD}	Operating Current	-	28.3	31	mA
I_{Idle}	Idle Current	-	7.25	8.6	mA
I_{IL}	Input Low Voltage	V_{SS}	-	0.2 V_{DD}	V
I_{IH}	Input High Voltage	0.75 V_{DD}	-	5.5	V

Table 2: TC4422A Specifications

Symbol	Characteristic	Min.	Typ.	Max.	Units
V_{DD}	Voltage Range	4.5	-	18	V
V_{IL}	Input Low Voltage	-	1.3	0.8	V
V_{IH}	Input High Voltage	2.4	1.8	-	V
V_{OL}	Output Low Voltage	-	-	0.025	V
V_{OH}	Output High Voltage	V_{DD} - 0.025	-	-	V
I_{DC}	Continuous Output Current	2	-	-	A
I_{PK}	Peak Output Current	-	10	-	A

Table 3: MCP2561 Specifications

Symbol	Characteristic	Min.	Typ.	Max.	Units
V_{DD}	Voltage Range	4.5	-	5.5	V
I_{DDR}	Recessive Supply Current	-	5	10	mA
I_{DDD}	Dominant Supply Current	-	45	70	mA
I_{DDS}	Standby Current	-	5	15	uA
V_{IL}	TXD Input Low Voltage	-0.3	-	0.3 V_{DD}	V
V_{IH}	TXD Input High Voltage	0.7 V_{DD}	-	V_{DD} + 0.3	V
V_{OL}	RXD Output Low Voltage	-	-	0.4	V
V_{OH}	RXD Output High Voltage	V_{DD} - 0.4	-	-	V

3 System Description

The system that was created is a motor controller that communicates over CAN. There are two dsPIC33EV256GM102 microcontrollers set up to communicate over CAN. One reads, analyzes, and transmits data from a potentiometer. The other reads the data from the first microcontroller as well as data from the hall effect sensor in the motor to control the speed of the motor. The second PIC will then determine whether the motor needs to speed up or slow down using an algorithm and transmit data in correspondence to the data received and send it to the gate drivers and motor drivers. These drivers will then spin the motor appropriately.

3.1 Detailed Block Diagram

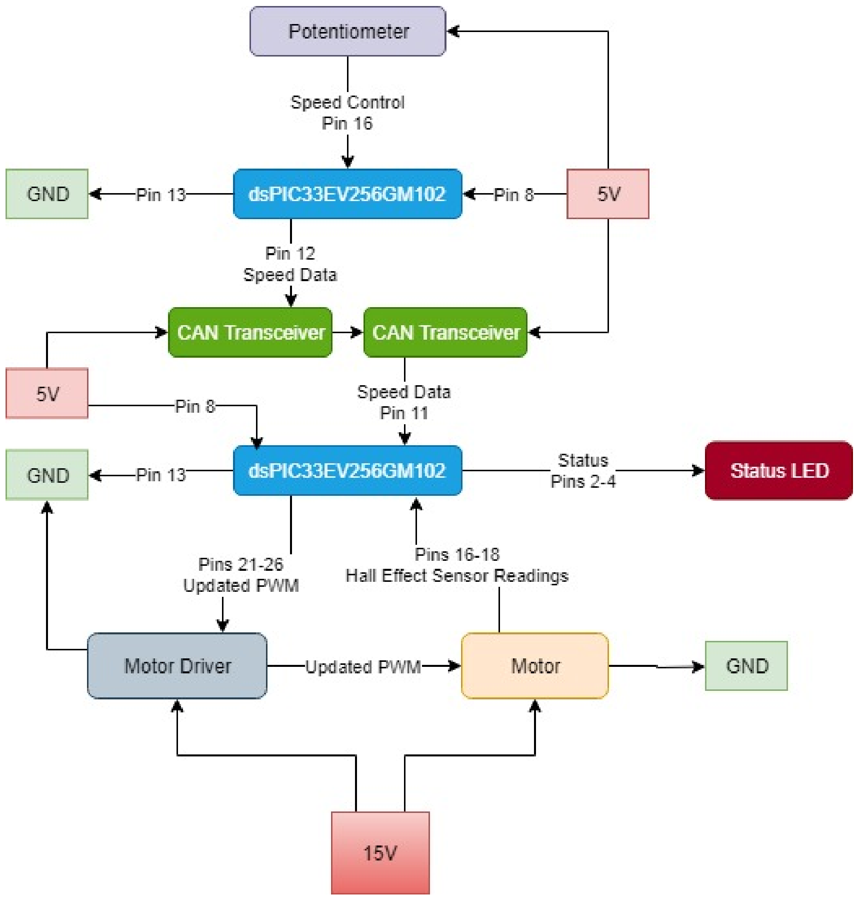

Figure 3: Detailed block diagram of the system

3.2 Highlighted Devices

- dsPIC33EV256GM102 Microcontroller
- Brushless DC Motor
- TC4422A MOSFET Gate Driver
- FQP27P06 PMOS
- FQP30N06L NMOS
- MCP2561 CAN Transceiver

3.3 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.

3.4 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, as shown in figure 4. These lines can be driven in a certain sequence that produces a rotation in the desired direction. Figure 5 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.

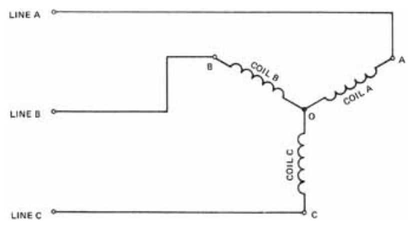

Figure 4: BLDC Motor Y configuration

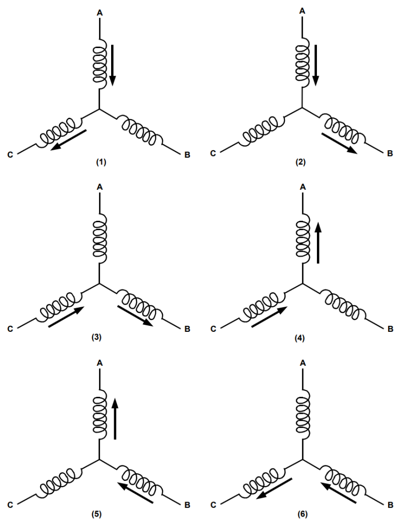

Figure 5: BLDC Motor Winding Energization

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.

3.5 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 6 shows an example of what this circuit could look like.

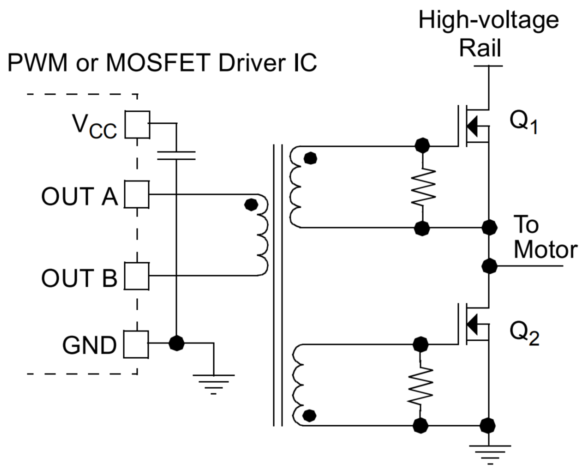

Figure 6: Double-Ended Gate Drive Transformer

3.6 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.

3.7 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.

3.8 MCP2561 CAN Transceiver

While the microcontroller contains a CAN peripheral, it does not have the ability to drive the differential voltage. This chip is designed to take the input TTL signal from the microcontroller and convert it to CAN. It also performs the reverse action by sending the current value from the CAN bus to the microcontroller. This separates the transmit/receive buffers, input filters, and input masks in the microcontroller from the actual noisy signals of the CAN bus.

Features like not driving the CAN lines low when not powered and a split output that allows the bus to still operate correctly even if one of the lines is shorted to high or low make this chip a valuable asset to the system. In addition, it is designed to buffer the sensitive microcontroller from the noisy CAN bus that could provide voltages outside of the microcontroller's operating range. To achieve low power usage, an input pin controls whether the CAN transceiver is disabled. Should the microcontroller error and drive a line to an active 0 output for too long, the CAN transceiver times this out and resets its output to a recessive state. This stops a microcontroller from locking the line, preventing any other data from being sent.

4 Circuit Setup

A 15 Volt power supply is used to power both the motor and the motor driver. A 5 Volt power supply is used to power the Microcontrollers, transceivers, and the potentiometer. These are all hooked up to a common ground. The potentiometer is hooked up to the first PIC. The first potentiometer is then connected to a CAN transceiver which is connected to another transceiver that transmits to the second PIC. This PIC has several purposes. It receives data from the first PIC as well as the motor, and it also transmits data to the motor driver. Going down the line, the second pic 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. This data is reported back into the second PIC where it can be compared to the desired speed and the mismatch can be corrected if there is one. The second PIC also transmits to an LED that is used to determine the status of the entire system.

5 Test Setup

To confirm the accuracy of the system, the circuit setup was followed. The range of speeds was swept through and the hall effect sensor was monitored to make sure the desired speed was being recorded. A load was put on the motor as well to make sure that the speed of the motor was adjusted to make up for the load.

5.1 Test Data

Figure 7 shows a measurement of a BLDC motor's phase voltage with respect to ground. The motor was set to run at 83.3 rps, or 5000 rpm.

It should be noted that due to a broken MOSFET gate driver, only 2.5 of the 3 phases were working, so 1 out of every 6 different states was exclusively driven to ground. This negatively affected the performance of the motor, but it was able to maintain running due to the momentum of the output shaft. Once the replacement part came in, the full 3 phases worked correctly.

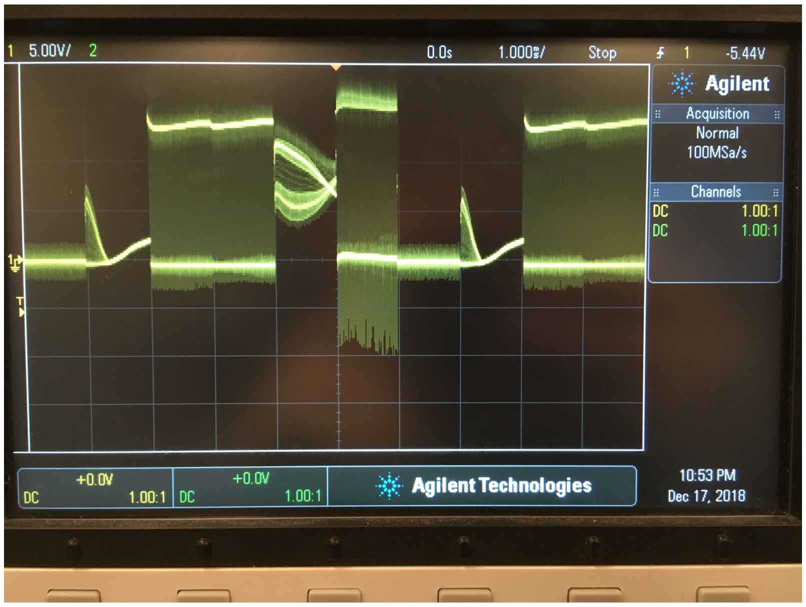

Figure 7: Single phase voltage with respect to ground

Figure 8 is a zoomed-in view of the phases. This shows the PWM cycling on and off to limit the current drawn by the motor and the applied voltage to the motor.

These figures demonstrate the correct operation of the BLDC motor. Motor speed was maintained even though various loads were placed on the output shaft.

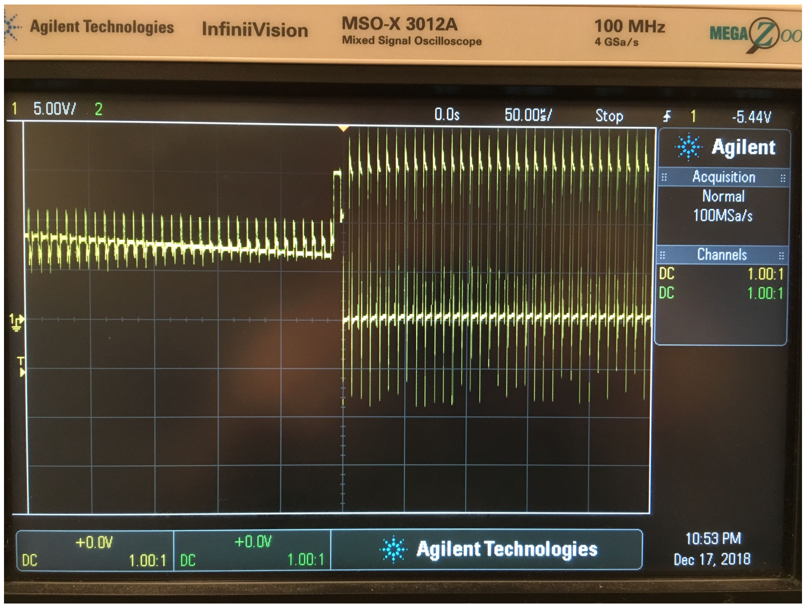

Figure 8: PWM of a single phase with respect to ground

6 Design

6.1 Schematics

6.1.1 3x Motor Driver

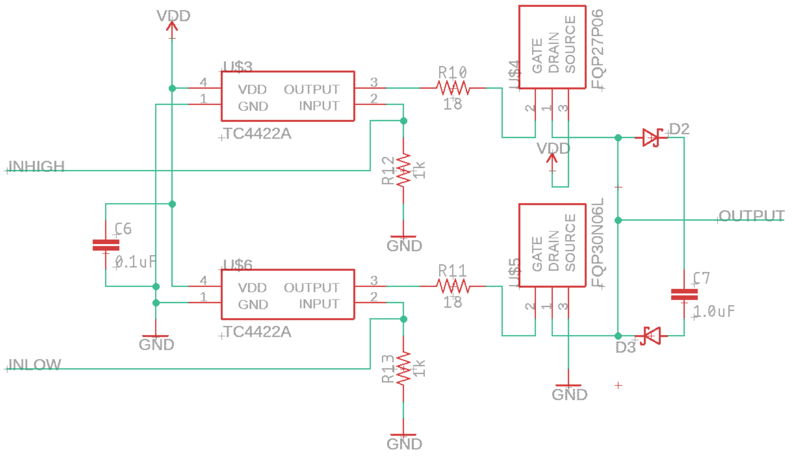

Figure 9: Motor Driver Schematic

6.1.2 2x Microcontroller

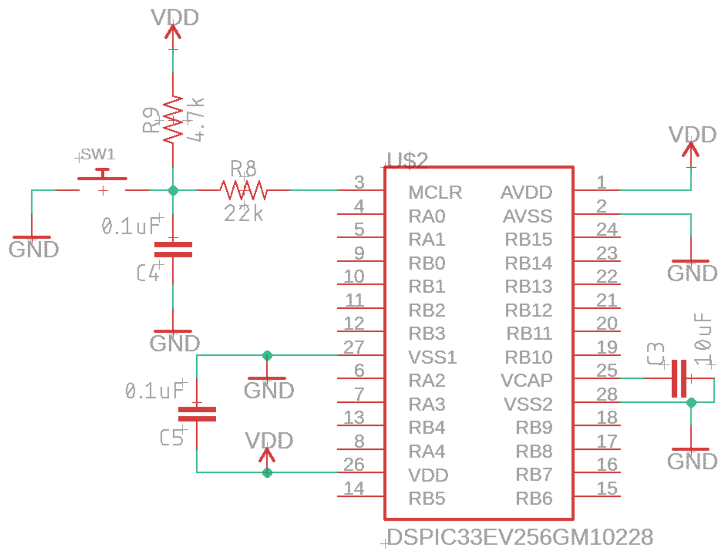

Figure 10: Microcontroller schematic

6.1.3 Motor Controller

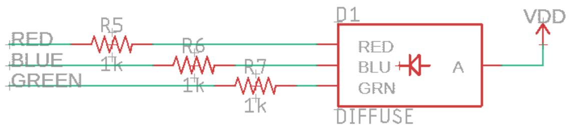

Figure 11: Motor controller schematic

6.1.4 Speed Input

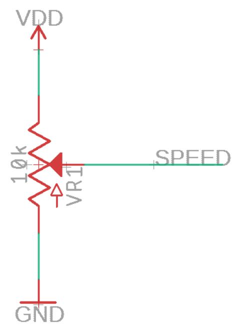

Figure 12: Speed Input Schematic

6.1.5 2x CAN Transceiver

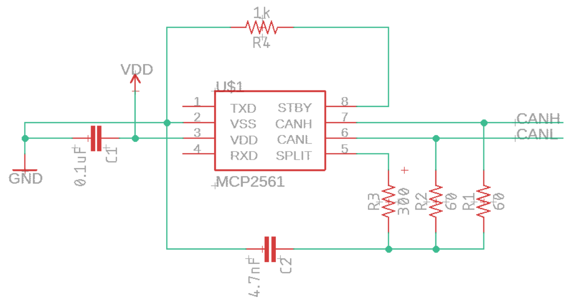

Figure 13: CAN Schematic

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 kΩ Resistor
- 3x 1 uF Capacitor
- 3x 0.1 uF Capacitor

6.2.2 2x Microcontroller

- 1x dsPIC33EV256GM102 Microcontroller
- 1x 22 kΩ Resistor
- 1x 4.7 kΩ Resistor
- 1x Pushbutton
- 1x 10 uF Capacitor
- 2x 0.1 uF 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 kΩ Linear Potentiometer

6.2.5 2x CAN Transceiver

- 1x MCP2561 CAN Transceiver
- 2x 60 Ω Resistor
- 1x 300 Ω Resistor
- 1x 1 kΩ Resistor
- 1x 0.1 uF Capacitor
- 1x 4.7 nF Capacitor

6.3 Code Appendix

Please see the full code listing on the GitHub page here. Shown below are the two main.c files that provide the logic of the two programs running on the microcontrollers.

6.3.1 BLDC Input 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.

    IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY
    INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL OR CONSEQUENTIAL
    LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND WHATSOEVER
    RELATED TO THE SOFTWARE, HOWEVER CAUSED, EVEN IF MICROCHIP
    HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE
    FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW,
    MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY
    RELATED TO THIS SOFTWARE WILL NOT EXCEED THE AMOUNT OF
    FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP
    FOR THIS SOFTWARE.

    MICROCHIP PROVIDES THIS SOFTWARE CONDITIONALLY UPON YOUR
    ACCEPTANCE OF THESE TERMS.
*/

/**
  Section: Included Files
*/
#include "mcc_generated_files/system.h"
#include "mcc_generated_files/clock.h"
#include "mcc_generated_files/adc1.h"
#include "mcc_generated_files/ecan1.h"
#define FCY _XTAL_FREQ/2
#include <libpic30.h>

/*
                         Main application
 */
int main(void)
{
    // initialize the device
    SYSTEM_Initialize();

    // Create CAN message
    uCAN1_MSG msg;
    msg.frame.id = 8;
    msg.frame.idType = CAN1_FRAME_STD;
    msg.frame.msgtype = CAN1_MSG_DATA;
    msg.frame.dlc = 1;

    // Enable transmitting
    ECAN1_TransmitEnable();

    // Start sampling the ADC
    ADC1_ChannelSelectSet(ADC1_SPEED);
    ADC1_SamplingStart();
```

```
        while (1)
        {
            __delay_ms(10);

            // Get a new speed value
            uint16_t newSpeed = ADC1_Channel0ConversionResultGet();
            uint8_t desiredRPS = newSpeed >> 4;
            ADC1_SamplingStart();

            // Send the new speed value
            msg.frame.data0 = desiredRPS;
            ECAN1_transmit(ECAN1_PRIORITY_HIGH, &msg);
        }
        return 1;
    }
    /**
     End of File
    */
```

6.3.2 BLDC Controller 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.

    IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY
    INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL OR CONSEQUENTIAL
    LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND WHATSOEVER
    RELATED TO THE SOFTWARE, HOWEVER CAUSED, EVEN IF MICROCHIP
    HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE
    FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW,
    MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY
    RELATED TO THIS SOFTWARE WILL NOT EXCEED THE AMOUNT OF
    FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP
    FOR THIS SOFTWARE.

    MICROCHIP PROVIDES THIS SOFTWARE CONDITIONALLY UPON YOUR
    ACCEPTANCE OF THESE TERMS.
*/

/**
  Section: Included Files
*/
#include "mcc_generated_files/system.h"
#include "mcc_generated_files/pwm.h"
#include "mcc_generated_files/pin_manager.h"
#include "mcc_generated_files/tmr2.h"
#include "mcc_generated_files/adc1.h"
#include "mcc_generated_files/ecan1.h"

/*
                         Main application
 */
int main(void)
{
    CNPDBbits.CNPDB4 = 1;
    // initialize the device
    SYSTEM_Initialize();

    // Set the status LEDs
    Stat1_SetLow();
    Stat2_SetHigh();
    Stat3_SetHigh();

    // Start the PWM
    MDC = 0x2FF;
    PWM_FaultInterruptStatusClear(PWM_GENERATOR_1);
    PWM_FaultInterruptStatusClear(PWM_GENERATOR_2);
    PWM_FaultInterruptStatusClear(PWM_GENERATOR_3);

    // Track the rotation of the motor
    int graycode = 0;
    uint16_t h1, h2, h3;

    // Track the RPS of the motor
    bool first = false;
    double timerPeriod = 1000; // ns
    double desiredRPS = 100;
```

```
// Initialize CAN
uCAN1_MSG msg;
ECAN1_ReceiveEnable();

while (1)
{
    // Read any available CAN message
    bool received = ECAN1_receive(&msg);
    if (received) {
        desiredRPS = msg.frame.data0;
    }

    // Read the hall effect sensor
    h1 = Hall1_GetValue();
    h2 = Hall2_GetValue();
    h3 = Hall3_GetValue();
    graycode = (h3 << 2) | (h2 << 1) | h1;

    // Enable the PWM lines based on the position of the motor
    switch (graycode) {
        case 1:
            // Recalculate PWM duty cycle to run at a
            // desired speed
            if (first) {
                first = false;

                // Set the speed based on the time taken
                // to loop once
                double period = TMR2_SoftwareCounterGet();
                TMR2_SoftwareCounterClear();
                // Twice as fast due to two cycles of the
                // hall effect sensor per revolution
                double calculatedRPS = 5000.0 / period;
                if (calculatedRPS < 1) {
                    calculatedRPS = desiredRPS;
                }
                double differencePercent = ((desiredRPS * 1.0) /
                calculatedRPS) - 1.0;
                uint32_t newValue = MDC + ((MDC * 0.01) *
                differencePercent);
                if (newValue > 0x37F) {
                    newValue = 0x2FF;
                } else if (newValue < 0x0FF) {
                    newValue = 0x0FF;
                }
```

```
                        MDC = newValue;
                    }
                    Stat2_SetLow();
                    Stat3_SetHigh();
                    PWM_OverrideHighEnable(PWM_GENERATOR_1);  // Yellow
                                                              // float
                    PWM_OverrideLowEnable(PWM_GENERATOR_1);
                    PWM_OverrideHighEnable(PWM_GENERATOR_2);  // Green
                                                              // low
                    PWM_OverrideLowDisable(PWM_GENERATOR_2);
                    PWM_OverrideHighDisable(PWM_GENERATOR_3);  // Blue
                                                               // high
                    PWM_OverrideLowEnable(PWM_GENERATOR_3);
                    break;
                case 5:
                    Stat2_SetHigh();
                    Stat3_SetHigh();
                    PWM_OverrideHighEnable(PWM_GENERATOR_1);  // Yellow
                                                              // low
                    PWM_OverrideLowDisable(PWM_GENERATOR_1);
                    PWM_OverrideHighEnable(PWM_GENERATOR_2);  // Green
                                                              // float
                    PWM_OverrideLowEnable(PWM_GENERATOR_2);
                    PWM_OverrideHighDisable(PWM_GENERATOR_3);  // Blue
                                                              // high
                    PWM_OverrideLowEnable(PWM_GENERATOR_3);
                    break;
                case 4:
                    Stat2_SetHigh();
                    Stat3_SetHigh();
                    PWM_OverrideHighEnable(PWM_GENERATOR_1);  // Yellow
                                                              // low
                    PWM_OverrideLowDisable(PWM_GENERATOR_1);
                    PWM_OverrideHighDisable(PWM_GENERATOR_2);  // Green
                                                              // high
                    PWM_OverrideLowEnable(PWM_GENERATOR_2);
                    PWM_OverrideHighEnable(PWM_GENERATOR_3);  // Blue
                                                              // float
                    PWM_OverrideLowEnable(PWM_GENERATOR_3);
                    break;
                case 6:
                    first = true;
                    ADC1_SamplingStop();
                    Stat2_SetHigh();
                    Stat3_SetHigh();
                    PWM_OverrideHighEnable(PWM_GENERATOR_1);  // Yellow
```

```
                                                  // float
          PWM_OverrideLowEnable(PWM_GENERATOR_1);
          PWM_OverrideHighDisable(PWM_GENERATOR_2);  // Green
                                                  // high
          PWM_OverrideLowEnable(PWM_GENERATOR_2);
          PWM_OverrideHighEnable(PWM_GENERATOR_3);  // Blue
                                                  // low
          PWM_OverrideLowDisable(PWM_GENERATOR_3);
          break;
     case 2:
          Stat2_SetHigh();
          Stat3_SetHigh();
          PWM_OverrideHighDisable(PWM_GENERATOR_1);  // Yellow
                                                  // high
          PWM_OverrideLowEnable(PWM_GENERATOR_1);
          PWM_OverrideHighEnable(PWM_GENERATOR_2);  // Green
                                                  // float
          PWM_OverrideLowEnable(PWM_GENERATOR_2);
          PWM_OverrideHighEnable(PWM_GENERATOR_3);  // Blue
                                                  // low
          PWM_OverrideLowDisable(PWM_GENERATOR_3);
          break;
     case 3:
          Stat2_SetHigh();
          Stat3_SetHigh();
          PWM_OverrideHighDisable(PWM_GENERATOR_1);  // Yellow
                                                  // high
          PWM_OverrideLowEnable(PWM_GENERATOR_1);
          PWM_OverrideHighEnable(PWM_GENERATOR_2);  // Green
                                                  // low
          PWM_OverrideLowDisable(PWM_GENERATOR_2);
          PWM_OverrideHighEnable(PWM_GENERATOR_3);  // Blue
                                                  // float
          PWM_OverrideLowEnable(PWM_GENERATOR_3);
          break;
     default:
          first = false;
          MDC = 0x2FF;
          Stat2_SetHigh();
          Stat3_SetLow();
          PWM_OverrideHighEnable(PWM_GENERATOR_1);  // All
                                                  // float
          PWM_OverrideLowEnable(PWM_GENERATOR_1);
          PWM_OverrideHighEnable(PWM_GENERATOR_2);
          PWM_OverrideLowEnable(PWM_GENERATOR_2);
          PWM_OverrideHighEnable(PWM_GENERATOR_3);
```

```
                PWM_OverrideLowEnable(PWM_GENERATOR_3);
            }
        }
        return 1;
    }
    /**
     End of File
    */
```