BRUSHLESS DC MOTOR CONTROL USING MSP432

FINAL PROJECT REPORT

ECEN 5613 EMBEDDED SYSTEMS DESIGN

DECEMBER 10, 2018

Submitted By:

Puneet Bansal &Vatsal Sheth

TABLE OF CONTENTS

1) INTRODUCTION

As engineers, Brushless DC motors (BLDC) has always piqued our interest pertaining to its wide variety of use cases in the embedded industry. Be it automotive (Electric Vehicles), medical, HVAC and industrial applications, BLDC is used everywhere for reliable and durable operation. BLDC employs an electronic commutation, doing away with the brushes that eliminates the source of wear and power loss. In addition to this, BLDC offers several other advantages over the conventional DC motors like less noise, better speed vs torque characteristics and faster response.

The BLDC motor commutation can take place either in sensored or sensorless manner. In the sensored approach, the hall sensors are used to provide the current position of rotor. Whereas the sensorless approach depends on the Back EMF generated in the stator. Due to the unavailability of Back EMF, the motor in this case is started from an arbitrary position which results in motor startup with a jerk.

1.1 System Overview

This project aims to create a precise and efficient 3 Phase Brushless DC Motor Controller using the **Closed Loop Technique**. The user turns the motor on/off and enters the desired speed of the motor via an interactive Graphical User Interface. The developed system tries to maintain the desired speed and the current values of speed and current consumption can also be seen on the GUI.

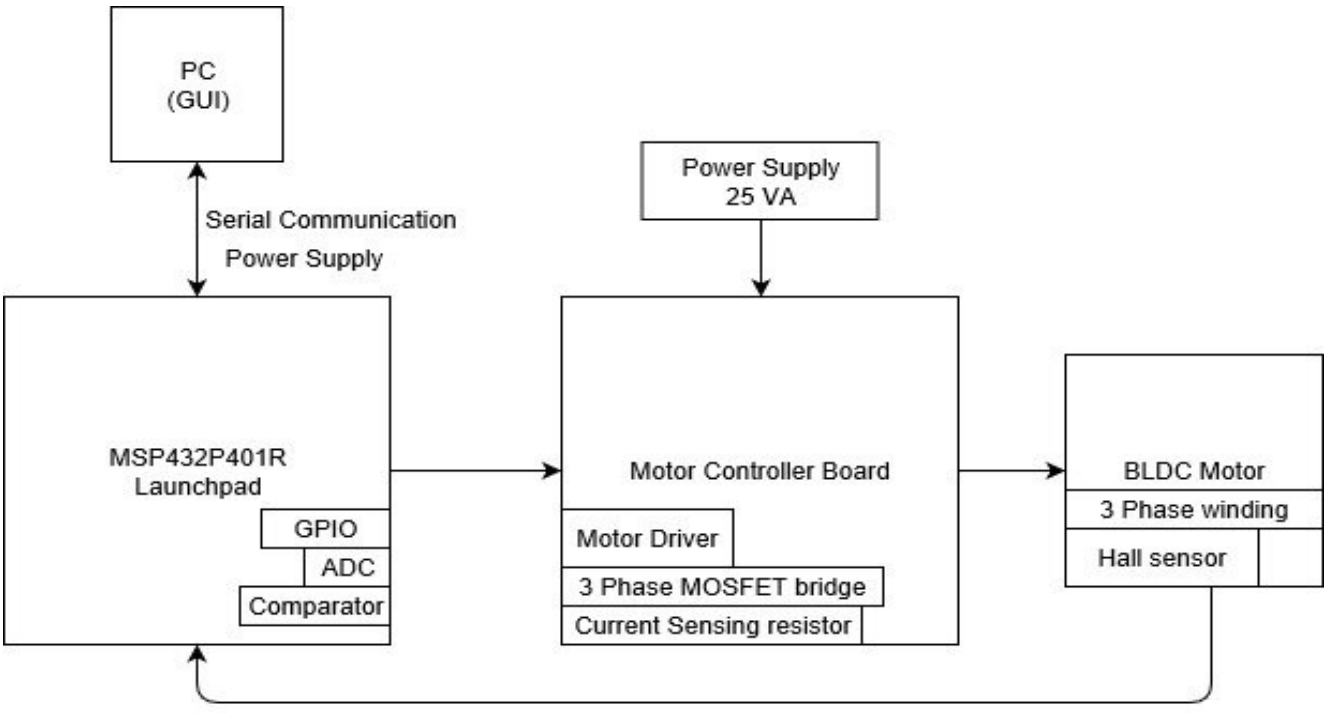

Figure 1.1: System Overview

2) TECHNICAL DESCRIPTION

2.1 BOARD DESIGN

As shown in Fig 1.1 System overview, this system is based on MSP432P401R launchpad which can be referred as MCU board with MSP432 as the MCU. Motor Controller board is responsible for driving the motor at required voltage and current levels. It also provides instantaneous current consumption feedback. Motor controller card uses GPIO on MCU board as an interface.

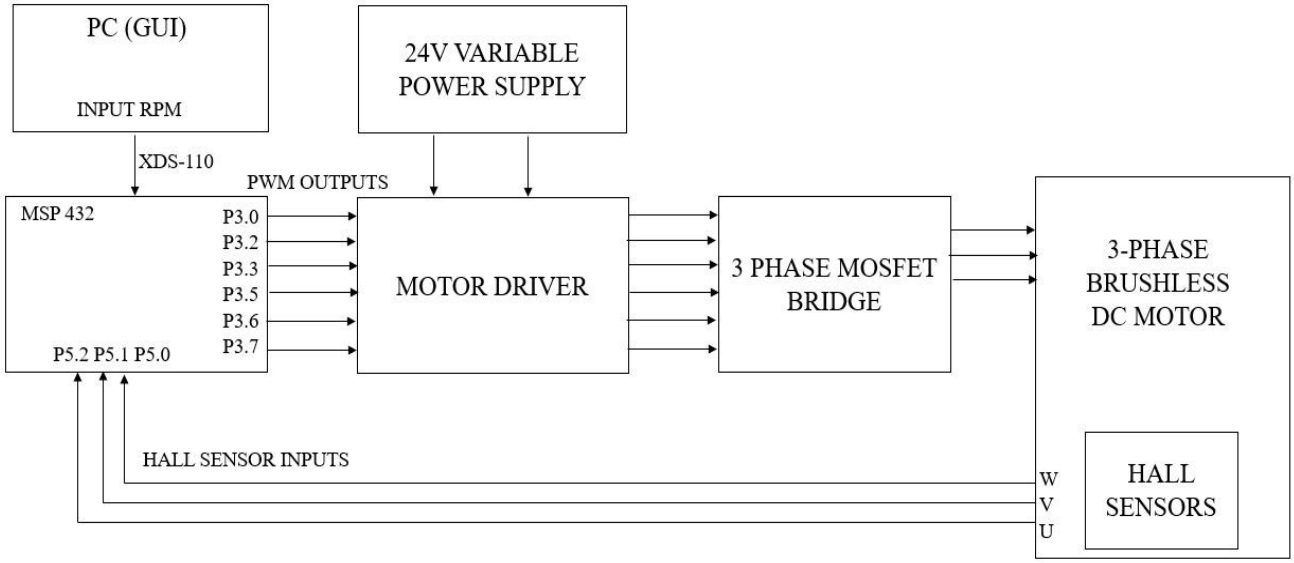

Figure 2.1 Hardware Block Diagram

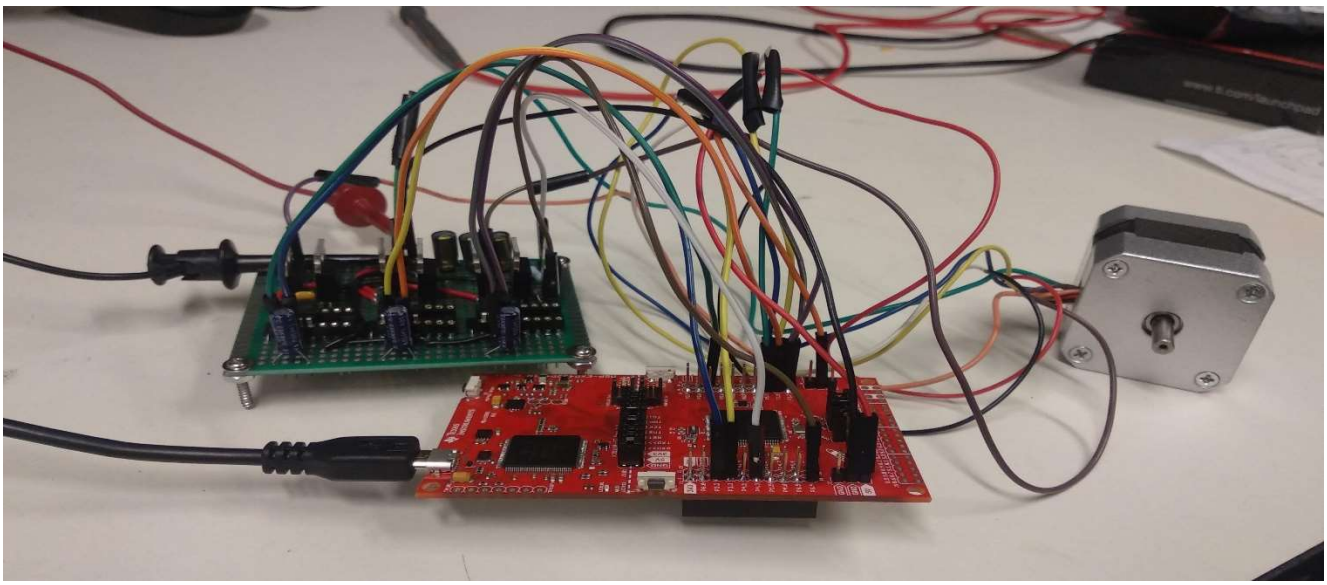

Figure 3.2 Hardware Setup

2.1.1 MCU Board

2.1.1.1 Motor Driver IC (IR2013) Interface:

N-channel MOSFETs are driven using Motor driver IC, to turn on this MOFET they must be driven to saturation and to turn off it needs to be in cut-off region of operations.
Mathematically they are represented as:
Cut-off Vs.<= Vth
Saturation Vgs>Vth & Vds > Vgs-Vth
As per the datasheet for IR2103, Logic input for high side gate driver is active high and for low side gate driver is active low. So to give power to one phase it is required to turn on high side MOSFET and turn off low side MOSFET. In terms of logic it 1 and 0 so in voltage terms it is driving high voltage 3.3V (in this case) on both the GPIOs which are connected to driver. Similarly to turn off the phase of motor it is required to supply ground that is turn off high side mosfet and turn on low side mosfet. In terms of logic it is 1 and 0 so in voltage terms it is driving 0V on both the GPIOs which are connected to driver. Note to maintain the logic levels between boards it is required to short the grounds of both these boards. To sum it up, to drive three phases three driver ICs are required, each on one of them requires 2 GPIO pins as logic inputs for high and low MOSFET gate drivers. So total 6 GPIO pins namely P3.0, P3.2, P3.3, P3.5, P3.6, P3.7 are used as output pins to interface with driver IC IR2103. Only reason behind choosing these set of port pins is availability of 6 pins of same port with headers on MCU board.

2.1.1.2 Hall Sensor Interface:

The hall sensors from BLDC motor are used to determine the current rotor position. Hall sensor consists of a metal strip carrying current which when placed in a transverse magnetic field results in the development of EMF across the edges of the strip. Thus as and when the poles of the rotor pass through hall sensor, it gives a logic high. The three hall sensor values (u,v,w) are interfaced to the 3 GPIO pins (P5.2, P5.1, P5.0) enabled with interrupts. The voltage of hall sensors readings are in the level of millivolts and so internal pull ups are used to bring the voltage to appropriate levels.
2.1.3 Current Sensing Resistor interface
A 0.6 ohm, 1W resistor is interfaced with P6.6 and ground to determine the current drawn by the circuit. The inbuilt Analog to Digital Converter (ADC) on the MSP432 has been utilised to get the appropriate digital value. An internal comparator with reference current 0.8 Amp has also been set up to switch off the circuit when the current exceeds the prescribed limit. The cutoff voltage is:

V(cutoff)= I(cutoff) * R
V(cutoff)=(0.8Amp) * (0.6 Ohm)
V(cutoff)=4.8V

2.1.2 Motor controller Board

This board consists of four main parts main power switch, Motor driver IC IR2103, 3 Phase full bridge using N- MOSFET and current sensing.

Power on Switch:
Switch is just a N-MOSFET placed between current sensing resistor and source of lower MOSFET. This MOSFET's source is practically grounded as source voltage due to current sensing resistor will be max 0.48V. All MOSFETs used are logic MOSFET (i.e low Vth 2.5V max) it's gate can be connected directly to GPIO (P6.4) on MCU board.

Current Sensing Resistor:
Current sensing resistor is of 0.6 Ohm 1W and it is connected between switch and ground so voltage drop across this resistor can be used to calculate current in ADC on MCU.

3-Phase Full Bridge Using N-MOSFET:
Each limb of 3 phase full bridge controls each drive phase, it consists of high side and low side MOSFET. As it can be seen in below figure, low side MOSFET can be turned on to saturation by giving Vgs > Vth where source voltage is near to 0V. But for high side MOSFET it's source is floating or it can be seen as connected to load so in this case source voltage is unknown. Even if we assume that this floating voltage is 0V then also gate voltage of 3.3V from GPIO is not sufficient. To understand this let's consider a hypothetical MOSFET which has only cut off and saturation region and no linear region and rest of the things are same as real one. So initially when 3.3V is applied at gate of high side MOSFET, it will turn on and go in saturation region immediately. As it will be ON so drain and source will be shorted with just Ron = 0.035 Ohms On resistance which means almost 0 voltage drop across it. As drain voltage is 18V from supply so in this ON state source voltage will also be 18V and so now Vgs is 3.3 - 18 = -14.7V which will turn OFF the MOSFET and may even damage it if it is above the tolerance limit for negative gate voltage for MOSFET. In this real scenario it won't even reach till saturation, it will just be in linear region and then again go back to cut off.

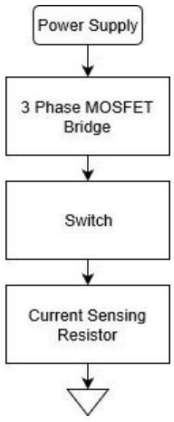

Figure 4.3: 3 Phase Full Bridge using N-MOSFET

Half Bridge Driver IC IR-2103:

To overcome the issue stated in previous section motor driver comes into picture. Motor driver IC has a bootstrapping capacitor mechanism to ramp up voltage by 10-20V depending on capacitor value with respect to reference. In this circuit reference is the source of high side MOSFET. So, as soon as high side MOSFET enters linear region from cut off Vds will increase till Drain voltage that is VCC = 18V and bootstrapping will ensure this 18V + 12V = 30V of gate voltage ensuring high side MOSFET goes to saturation. This IC has a bypass capacitor on its supply of 1uF 50V and bootstrapping capacitor is of 0.22uF 50V value.

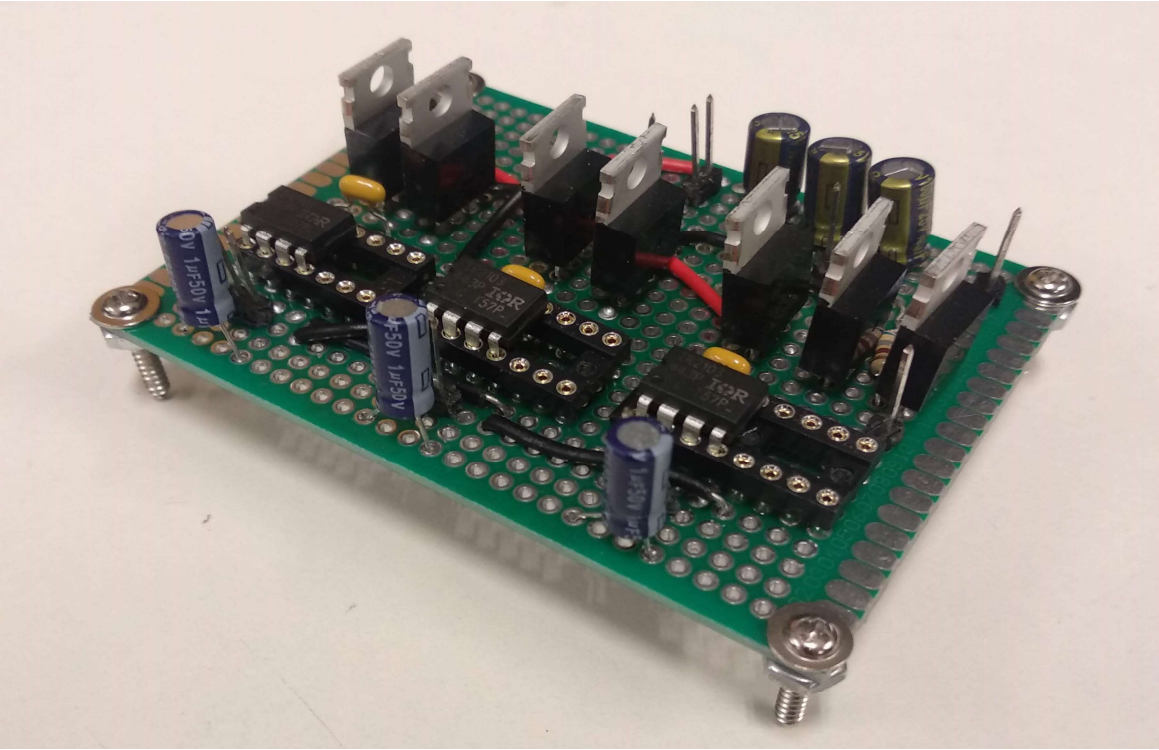

Figure 5.4 Motor Controller Board

2.2 FIRMWARE DESIGN

The entire firmware for this project was developed on TI's Code Composer Studio v 8.2.0 for MSP432P401R. All the unused GPIO pins of the processor are set as input pins with internal pull up to save power.

The Firmware Components for this project involves

1. PWM generation
2. 3 Phase MOSFET Drive
3. Hall sensor interface
4. Current sensing
5. Proportional Control.

2.2.1 PWM Generation

We have generated PWM pulses of 8KHz. We have assumed that at maximum motor speed i.e. 4000 RPM , we'll be able to generate atleast 4 PWM pulses.

Calculations:

Number of Poles in BLDC motor		**Np**
Max RPM		**R**
Number of PWM cycles per hall state		**P**
Max RPS = R/60		
R/60 Mechanical Rotation	**--**	1 Sec
1 Mechanical Rotation	**--**	60/R Sec
1 Mechanical Rotation	**--**	Np/2 Electrical Rotation
Np/2 Electrical Rotation	**--**	60/R Sec
1 Electrical Rotation	**--**	(60/R) * (2/Np) Sec
1 Electrical Rotation	**--**	6 Hall States
1 Hall state change	**--**	(60/R) * (2/Np) * (⅙) Sec
1 Hall state change	**--**	P PWM cycles
1 PWM cycle	**--**	(60/R) * (2/Np) * (⅙) * (1/P) Sec

Motor's stated Max RPM is 4000 ± 10% = 4400 = 4500 (Safe Calculation)
Np = 8 poles
PWM cycles per Hall state = 4
Time period for PWM = 0.138 ms = 0.125 ms (Safe Calculation) = 8Khz frequency

So, now for a 3MHz clock

0.33 microsecods = 1 count
Number of counts in one second = (10^6)/0.33
To get 8KHz PWM signal, time period =⅛ ms
= 0.125ms
Therefore, number of counts in 0.125 sec = 0.125 * (10^6) / 0.33
= 375

We are using PWM signals to drive only the upper MOSFET. Due to the use of PWM signals, the upper MOSFET will switch between on and off states at 8 KHz frequency which will help in averaging out current supplied to the MOSFET.
However, for the lower MOSFET's, the use of PWM signal would result in switching between ground and floating states which will not result in the desired operation. So , a constant logic high or low has been used to drive the lower MOSFET's.

2.2.2 Three Phase MOSFET drive

For clockwise commutation, the following sequence has been utilized

State	Hall Inputs (u|v|w)	Active PWM and Power FET
1	001	PWM3, PWM6 | H_V, L_W
2	011	PWM1,PWM6 | H_U, L_W
3	010	PWM1,PWM4 | H_U, L_V
4	110	PWM5,PWM4 | H_W, L_V
5	100	PWM5,PWM2 | H_W, L_U
6	101	PWM3,PWM2 | H_V, L_U

Table 6.1 Table depicting hall inputs and corresponding FET's that need to be active

Let us understand the above table with an example. Consider the hall sequence 001. Now, the active power FET's in this sequence are H_V and L_W. Which means a logic high needs to be given to H_V. The logic input for the low side gate driver in IR2103 is low therefore, a logic 0 needs to be given to turn it on. Hence, we provide logic level 0 to L_W and supply logic level 1 to the other 2 lower side gate drivers (L-U, L-V)) to turn them off. The same logic can be seen with the help of diagram shown below.

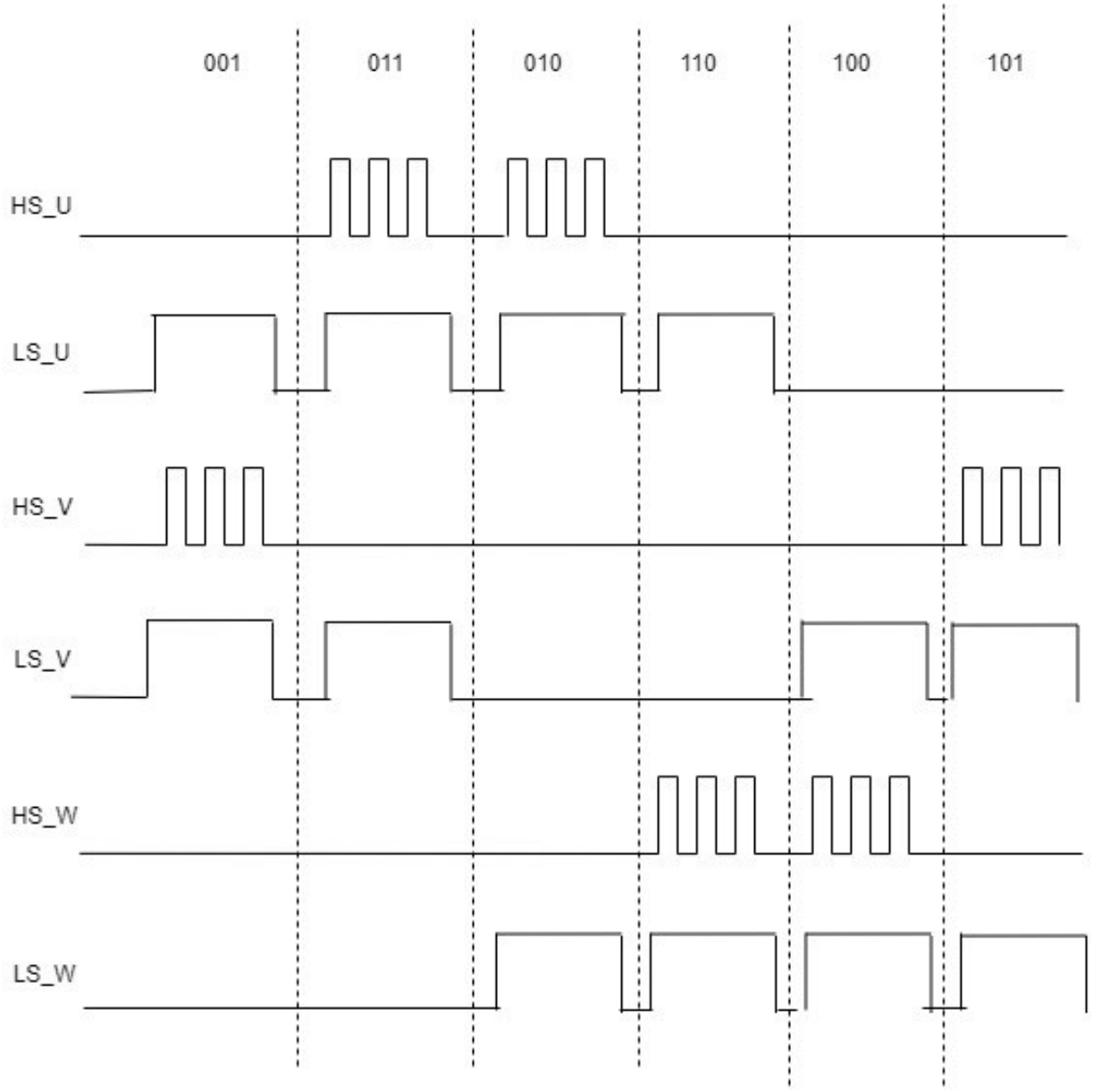

Figure 7.5 Motor Commutation Sequence based on Hall Sensor Inputs (Clockwise)

Motor On Sequence : To start the motor initially, the hall sensor readings are obtained and appropriate MOSFET's are driven so that the initial motor commutation starts. Further addition to this sequence which is currently not implemented in the code is that a timer can be used to check if the motor starts within 1ms .If not, then the motor start sequence will be called again. This condition would be continuously checked in a loop, till the time a motor stop sequence command is received from the GUI or motor starts.

Motor Stop Sequence : First the switch is turned off. Then all the MOSFET's are driven into the cutoff state or OFF state. This is done to ensure that even if due to some reason the switch is not able to turn the system off, there would be no short circuit path on any of the phases since all the MOSFET's will be in cutoff state.

2.2.3 Hall Sensor Interface

The three hall sensor values (u,v,w) are interfaced to the 3 GPIO pins (P5.2, P5.1, P5.0) and port interrupt is enabled for PORT5.
Now, when the hall sensor reading changes, PORT5 interrupt is triggered. In the Interrupt Service Routine, the hall states are stored and based on the hall state, the MOSFET's to be driven are decided. We have implemented a look up table for the same.In the lookup table hall states are mapped to their corresponding index. According to the hall states, the look up table consists information about which MOSFET's to drive and whether the next interrupt will be falling edge or rising edge.

This is the Look-Up table implementation:

```
Hall_commutaion Hall_Lookup[]={
                        {Rising_edge,L_U | L_V | L_W, 0},
                        {Rising_edge,(H_V | L_U | L_V), H_V},
                        {Rising_edge,(H_U | L_U | L_W), H_U},
                        {Falling_edge,(H_U | L_U | L_V), H_U},
                        {Rising_edge,(H_W | L_V | L_W), H_W},
                        {Falling_edge,(H_V | L_V | L_W), H_V},
                        {Falling_edge,(H_W | L_U | L_W), H_W}
                       };
```

Let us consider, the 1st index in the hall lookup, i.e. 0x01. The first entry tells that the next interrupt needs to be rising edge. The second entry tells which all MOSFET's needs to be turned on, which in this case are H_V, L_U, L_V. We have supplied logic high to L_U and L_V to turn them off and to turn L_W on (IR 2103 logic). The last entry explains the MOSFET on which the PWM signal needs to be applied.

2.2.4 Current Sensing

The firmware for current sensing involves taking ADC readings at regular intervals accross the current sensing resistor. The following formula has been used to calculate the current from the obtained voltages.

Current = Voltage at current sense resistor / current sense resistor
I = (Vadc) / R
I = (ADC value * Vref) / ((2^14 -1)*R)
Current across the current sensing resistor= ((Obtained ADC value* 1.2) / (16384 * 0.6))

The current firmware gives the instantaneous value of current which is not accurate. This is so because depending whether one or two phase is being driven at that time, the current may vary. A better way is to measure the average current. Now, the 2 possible solutions which are not currently implemented in our implementation are:

- **Hardware Solution**: Add a small RC circuit before connecting to ADC GPIO to average out the current.
- **Firmware Solution**: Since we are aware about the number of phases the code is driving at the time of ADC conversion, we can divide the measured current by the number of active phases. This will also give average current.

2.2.5 Proportional Control

The proportional controller aims to achieve and maintain the desired RPM input from the user. Proportionality constant is estimated using manual method, where different duty cycle PWM signal is driven to motor and corresponding RPM is measured. This tabular data is available in appendix. So once the set RPM is updated from GUI, duty cycle is directly updated somewhere near to the expected duty cycle as per this table according to below equation.

duty_cycle = (75*RPM_user))/5500

After this the propositional logic takes over and tries to breach the gap and maintain the set speed. Firstly the error is calculated between the expected time between successive hall state changes and the actual value.

expected_time = (3 * 1000000 * 60)/(RPM_user * 24)
error = actual time - expected_time

If the error is positive it will mean that actual speed is slow then the set speed and so duty cycle needs to be increased and vice versa. After this three levels are defined, if error is less then 300 then duty cycle changes are made finer, if it's between 300-500 then little less fine and above that coarse changes are made in duty cycle.

Error	Change in duty cycle
0-300	- (error/700)
300-500	- (error/500)
>500	- (error/300)

At all time it is maintained that duty cycle doesn't go below 5% and is not increased more than 75% as at this duty cycle as per the observation table motor crosses max RPM for no load condition.

2.3) SOFTWARE DESIGN

Main Loop:

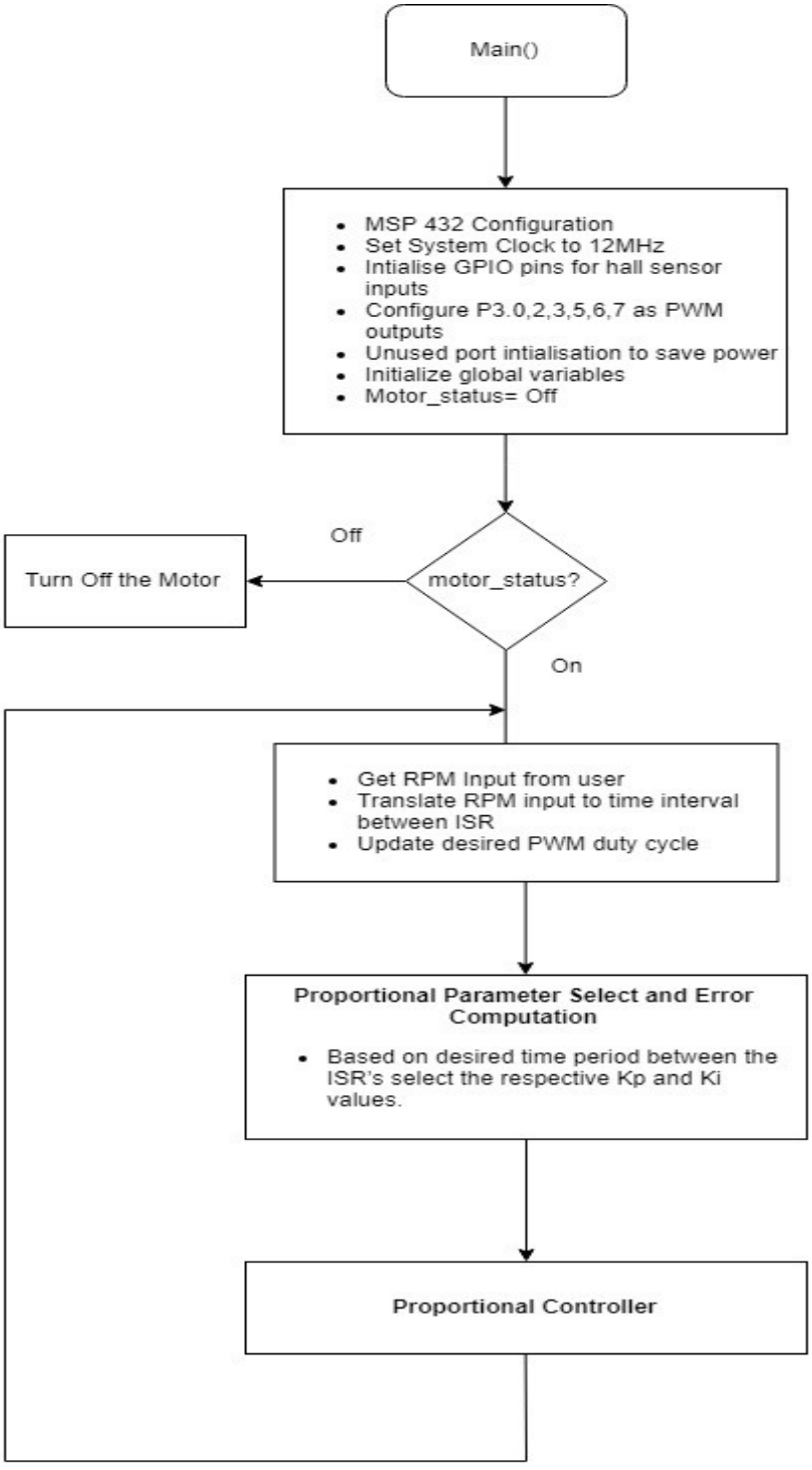

Figure 8.6 Flow chart depicting main loop operation in firmware

Motor On:

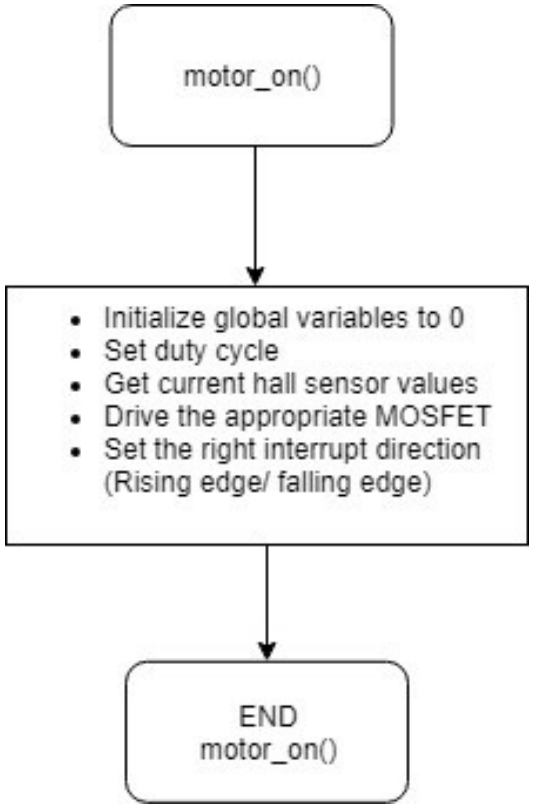

Figure 9.7 Flow chart depicting motor_on() function

GPIO Interrupt :

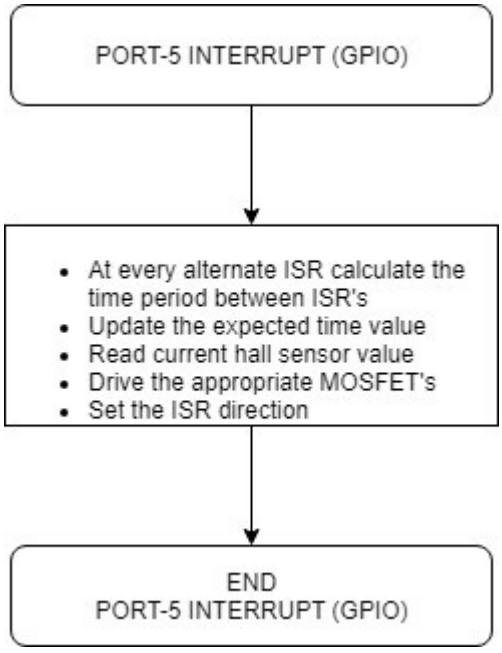

Figure 10.8 Flow chart depicting PORT 5 i.e. GPIO interrupt

2.4) GUI DESIGN

An interactive Graphical User Interface (GUI) has been developed for the project using TI's GUI composer studio. The GUI communicates with MSP432 using XDS-110 interface. The GUI works by linking the global variables with the widgets available in the GUI composer studio. We have made global variables to keep a track of current sensing, motor speed (RPM), to turn on and off the motor and control the motor speed .The image of the GUI can be seen below.

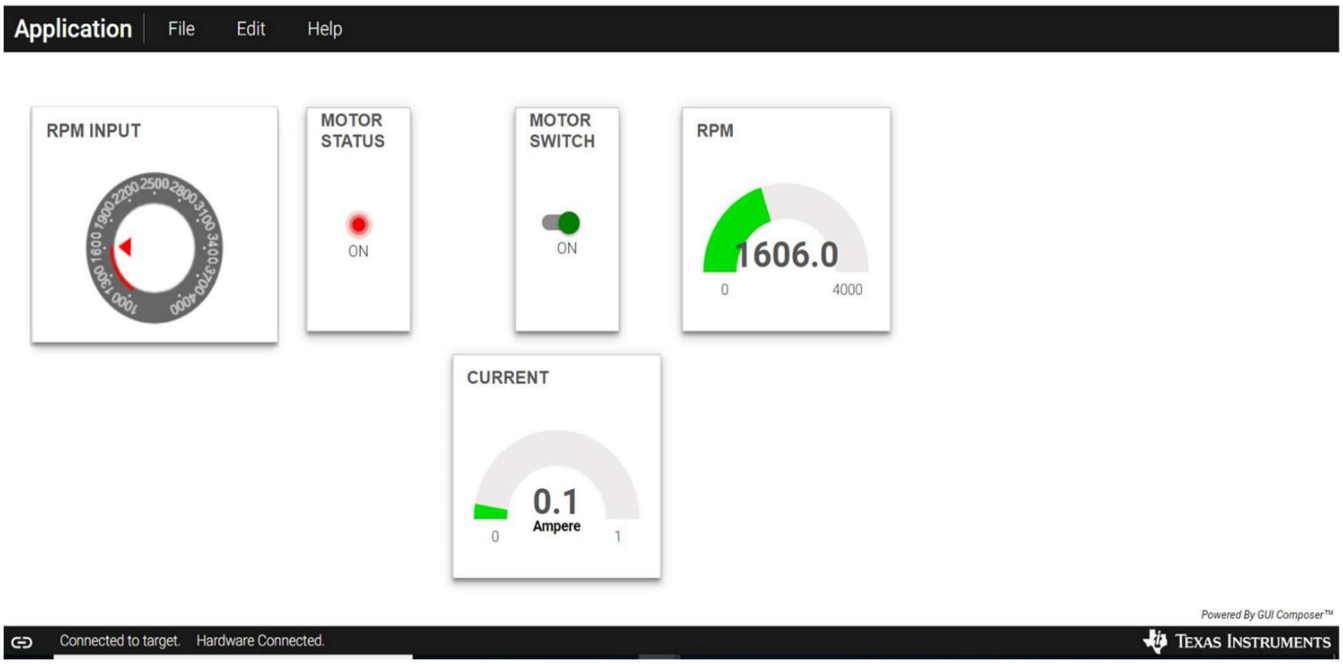

Figure 11.9 Interactive Graphical User Interface

In the above picture, we can see that the motor is turned on using the motor switch. The status of the motor can also be seen using the motor status LED. At the moment, the RPM set by user via the RPM input knob is 1600 RPM. The current speed achieved by the motor i.e. 1606 can be seen in the RPM widget and the current consumed by the motor can be seen in the current widget.

2.5) TESTING AND DEBUGGING PROCESS

1) Testing individual MOSFET's

The first step in our testing process involved testing individual MOSFET's. In order to do this, we drove the gate of the MOSFET via GPIO pin and provided VDD and GROUND. On supplying a logic high to the gate, the drain and source voltage got shorted which could be easily observed on the multimeter. This confirmed that the MOSFET's were working.

2) Connected the hall sensor inputs from the motor to the GPIO pins.

Hooked up logic analyser to the hall inputs and manually rotated the motor shaft. Then we checked the value of the obtained commutation sequence with the commutation sequence table.

3) Without the enabling the PWM signals, ran the code.
Connected the logic analyser on PWM and GPIO pins, rotated the motor shaft manually and checked whether correct MOSFET'S are being driven on the corresponding hall sequences.

4) Enabled PWM logic and observed the signal on logic analyser to check for the duty cycle, period etc.

5) In the difficulties faced, we have discussed that initially, we were not using half bridge driver IC's to drive the MOSFET's. In order to solve this problem, we connected the IR2103 IC. **Then we tried to test the system without supplying power to the motor (to prevent damage).** This testing technique was wrong, since the higher side MOSFET's were not able to turn on. Due to the absence of load, the points were left floating which caused the system to misbehave. Finally, we connected the motor in the circuit, loaded the fully functional firmware and the system operated in a smooth manner.

3) ENGINEERING DIFFICULTIES FACED

1) Insufficient Gate current for high frequency MOSFET switching
For high frequency switching sufficient gate current is required by MOSFET. In our first hardware design motor driver IC IR2103 was not used and GPIO were driving gates directly. GPIO can at max source 15-20 mA current which is not sufficient. Contrary to that IR2103 sources IO+/- 210/360 mA current which solves the gate current issue for MOSFET.

2) High side MOSFET not turning ON
In first developed board MOSFET were driven directly through GPIO which is possible as MOSFET used are logic MOSFET which have low 2V threshold. As high side MOSFET's source is floating or connected to load, once it is turned on source voltage will be equal to drain voltage which is supply of 18V. So Vgs now is 3.3 - 18 = -14.7V which forces the MOSFET to turn off and it is also above the tolerance of negative gate voltage which blew our all three high side MOSFET in first board. To solve this issue driver IC IR2103 is used to bootstrap 10-20V above this source voltage to ensure MOSFET operation in saturation.

3) Rectifier Diodes for bootstrapping circuit limiting switching speed
Wrong part selection is the cause of this issue, and inability to order replacements in time and shipping price are the practical constraints. Use of this diode as it can be seen in the circuit below is to ensure that reverse current doesn't flow back to Vcc once bootstrap voltage goes above Vcc. Depending on the PWM switching speed this capacitor need to charge and discharge which requires this diode to have fast reverse recovery time. As the part used was a 50V 1A rectifier diode instead of Schottky diode, this results in high rise and fall time in MOSFET switching. As a result of this full current or power is not delivered to motor. As motor operation during demo is in no load condition so it has no implications but with load if this doesn't delivers sufficient current then it can result in speed reduction.

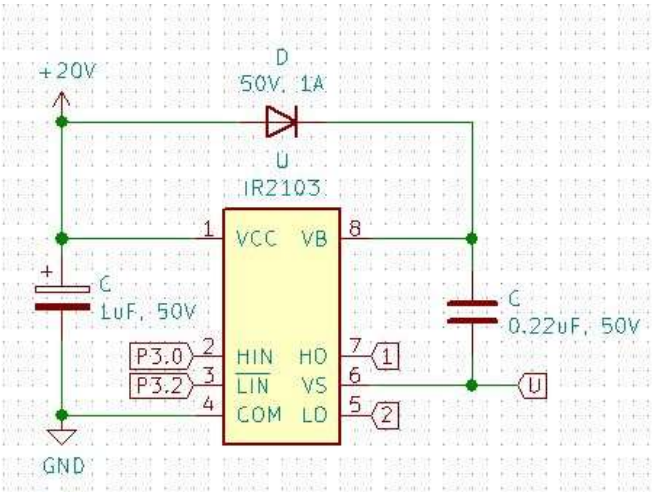

Figure 12.10 ()

4) GPIO high frequency switching glitch for hall sensor causing interrupt to occur twice instead of one.

The interrupts in our program triggers at a very fast rate and a large number of interrupt service routines needs to be handled. The fast switching nature of the program, resulted in high frequency switching glitch which caused the hall sensor interrupt to occur twice instead of once. This problem can be seen in the logic analyzer screenshot given below. For instance, at the hall sequence 4H, we see both rising and falling edge interrupts. This affected the time difference calculations between successive ISR's.

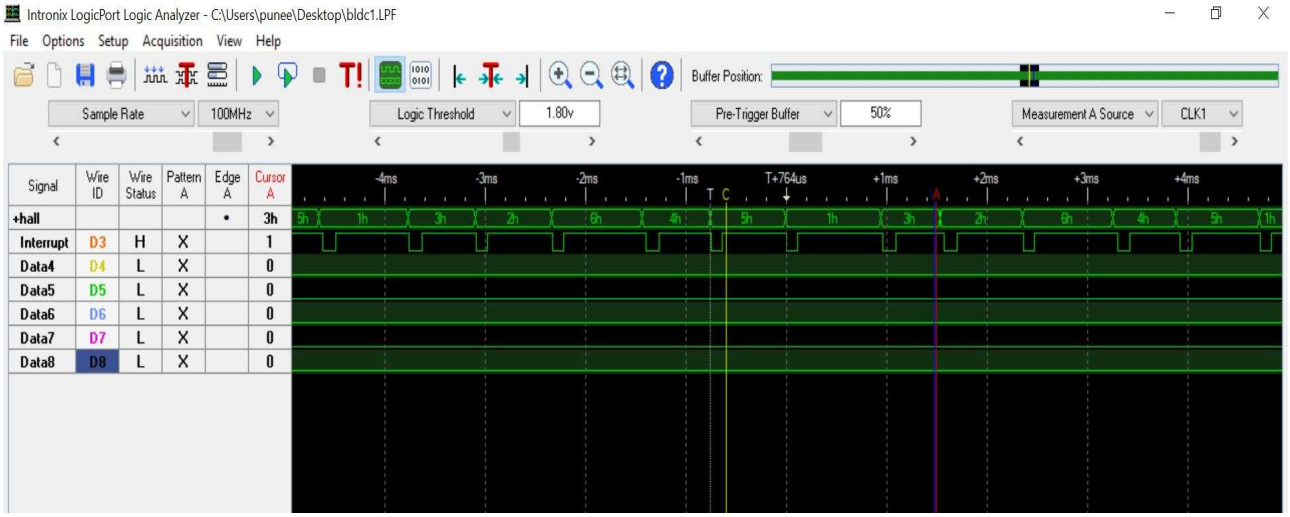

Figure 13.11 Screenshot depicting ISR's occurring twice for a hall state change

Possible Solution: A possible solution to this problem would be the use of internal glitch filters present inside the GPIO. However, the pins that support this glitch filtering are present on separate ports. So, if we use of these pins to remove the glitch, we would have to read the interrupts from 2 separate ports which would have resulted in lag.

Implemented Solution: So, the solution we practically implemented to solve this problem was to bypass every alternate interrupt and calculate the time difference between only these interrupts.

5) Missing ground isolation, results in high ground currents causing the MSP to heat up during development phase.

In our design, we have used separate sources to power up the MSP and the motor. MSP is being powered up through the USB in laptop and the motor is powered up using a 24V variable power supply. Since there was no ground isolation, the ground current used to flow through MSP 432 causing the board to get heated up.
However, as and when we improved our firmware and hardware design, this problem of MSP heating got solved.

6)Default processor clock speed (3MHz) resulted in unsmooth motor operation.

The default processor clock speed at which MSP 432 operates is 3MHz. Initially, we were developing our firmware using this processor speed, which caused the motor to work in an unsmooth or jerky manner.
Due to this, our program would keep on jumping from one ISR to next and would never return to the main loop.

Solution: The solution we developed for this problem, was to increase the clock speed to 12MHz which resulted in motor to run smoothly without any jerks.

4) CONCLUSION

Being able to work on the systems level and develop hardware and firmware from scratch was one of the major motivating factors to chose this project. Although there are already motor driver IC's available in the market, but our curiosity to see how it operates motivated us to take up this project.

Earlier, power electronics concepts always felt daunting. However, going through the project development phase made us come out of our comfort zone and implement something new. Throughout the project we encountered various engineering challenges, solving which gave us deep understanding of the subject.

We developed hardware, firmware and software skills and had great fun developing it as well.

5) FUTURE DEVELOPMENT IDEAS

- Implementation of Proportional Integral controller instead of just Proportional controller to achieve more stability.
- Real time current vs time graph on GUI instead of just displaying the current consumption value.
- Testing and improvements of the system with a load on motor.

6) REFERENCES

- **CONCEPT**: http://www.ti.com/lit/an/slaa503/slaa503.pdf
- **MSP432 REFRENCE MANUAL**: http://www.ti.com/lit/ug/slau356h/slau356h.pdf
- **IR 2103** https://www.infineon.com/dgdl/ir2103.pdf?fileId=5546d462533600a4015355c7b54b166f
- **MOSFET:** https://cdn.sparkfun.com/datasheets/Components/General/FQP30N06L.pdf

7) APPENDICES

7.1) BILL OF MATERIAL

COMPONENT	SPECIFICATIONS	QUANTITY	PRICE ($)
Motor	24V, 1.8A	1	30
Double Sided PCB	-	5	5
N Channel MOSFET	60V , 30A	7	6.65
IR2103 Half Bridge IC	-	3	6
Diode	50V, 1A	3	.45
Capacitors	100µF, 25V	3	1.05
	1µF, 50V	3	1.35
	.22µF, 50V	3	0.88
Current Sensing Resistor	0.6 Ω, 1W	1	1
Shipping Charges	-	-	35
Total Cost			87.38

Table 7.1 Bill Of Material

7.2) SCHEMATICS

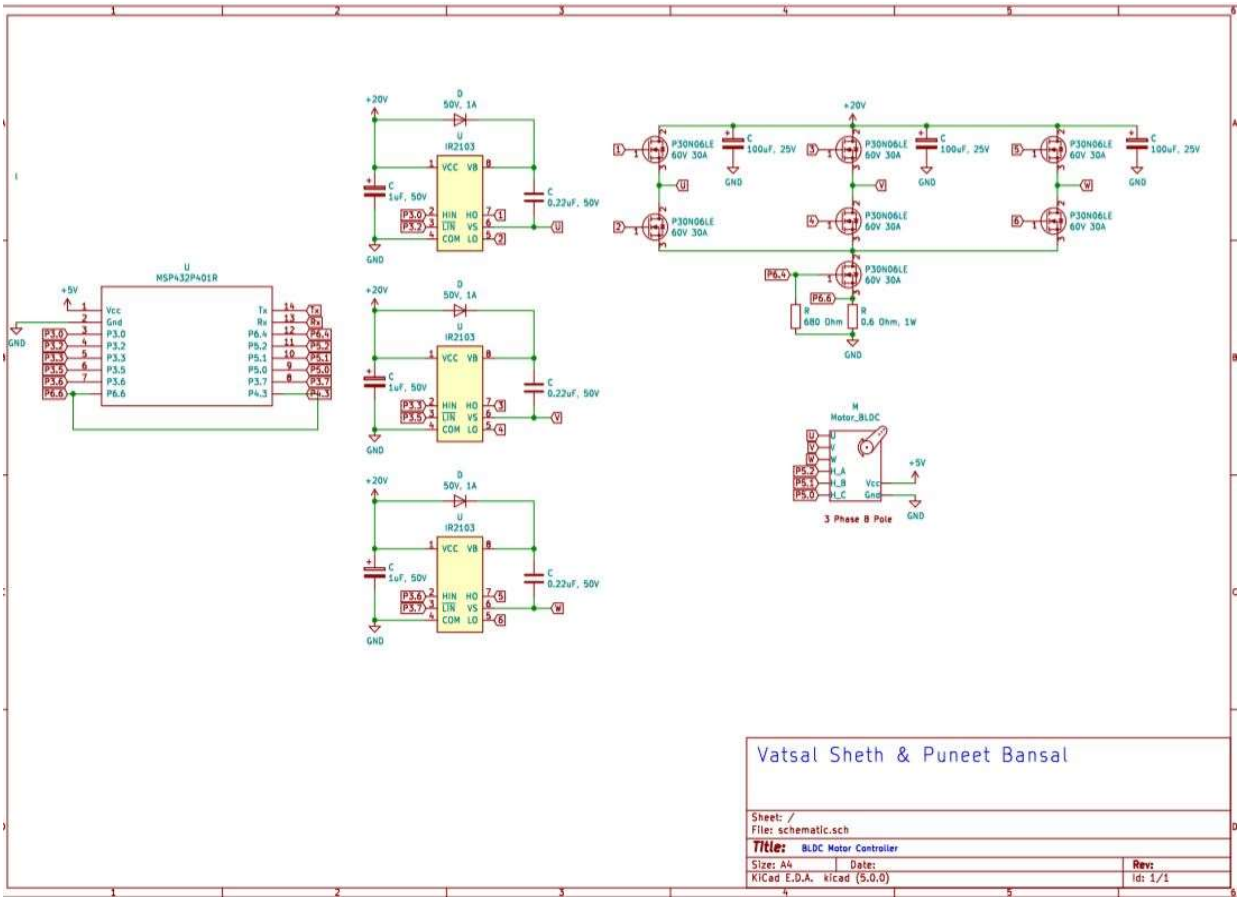

Figure 7.1 Schematics

7.3) PROPORTIONAL CONTROL MANUAL TUNING

DUTY CYCLE	RPM
5	665
10	1000
15	1350
20	1850
30	2700
40	3160
50	3700
60	3941
70	4125
75	4300

Table 7.2 Duty Cycle Vs RPM

7.4) SOURCE CODE

```
/*****************************************************************************
* @main.c
* @This file contains main, port initialization functions. It stops watchdog, updates
* the clock frequency 12Mhz, motor control and calls check for control logic
*
* @author Vatsal Sheth & Puneet Bansal
 *****************************************************************************/

#include "main.h"
void main(void)
{
   WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer

   CS->KEY = CS_KEY_VAL ;                  // Unlock CS module for register access
   CS->CTL0 = 0;                           // Reset tuning parameters
   CS->CTL0 = CS_CTL0_DCORSEL_3;           // Set DCO to 12MHz (nominal, center of 8-     16MHz range)
   // Select ACLK = REFO, SMCLK = MCLK = DCO
   CS->CTL1 = CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3 |    CS_CTL1_DIVS1;
   CS->KEY = 0;                            // Lock CS module from unintended accesses

   unused_port_init();

   RPM_user = 1000;    //Initial start speed
   motor_on_flag = 1;
   motor_off_flag = 0;
   rpm_prev = 0;
   motor_status = Off;

   pwm_port_init();
   hall_port_init();
   motor_init();
```

```
    __enable_irq();

    while(1)
    {
        if(motor_status == On && motor_on_flag==1)
        {
            motor_on_flag = 0;
            motor_off_flag = 1;
            motor_on();
        }
        else if(motor_status == Off && motor_off_flag==1)
        {
            motor_off_flag = 0;
            motor_on_flag = 1;
            motor_off();
        }

        if(RPM_user != rpm_prev)
        {
            update_desired_rpm();
            rpm_prev = RPM_user;
        }

        if(motor_status == On)
        {
            calc_param();
        }
    }
}

//unused port are set as input with internal pull up to consume power
void unused_port_init()
{
```

```
    P1->REN = 0xff;
    P2->REN = 0xff;
    P3->REN = BIT1 | BIT4;
    P4->REN = 0xff;
    P5->REN = BIT3 | BIT4 | BIT5 | BIT6 | BIT7;
    P6->REN = 0x6f;
    P7->REN = 0xff;
    P8->REN = 0xff;
    P9->REN = 0xff;
    P10->REN = 0xff;
}

//initialize GPIO for hall sensor inteface
void hall_port_init()
{
    P5->DIR &= ~(BIT0| BIT1 | BIT2);              //using pins 5.0,5.6,5.7 as input
    P5->REN |= (BIT0 | BIT1 | BIT2);              //Pull up/down enabled
    P5->OUT |= (BIT0 | BIT1 | BIT2);              //For pull up

    P5->IE|=BIT0 | BIT1 | BIT2;                   //enabling interrupt for hall sensor
    P5->IFG = 0X00;                               //clearing interrupt flag initially
}

//initialize GPIO for driving MOSFET
void pwm_port_init()
{
    P3->OUT = 0x00;
    P3->DIR|= BIT0| BIT2| BIT3| BIT5| BIT6| BIT7;   //Setting these pins as outputs for PWM
}
```

```
/*****************************************************************
* @main.h
* @This file contains declaration of functions and variables used in main.c
* motor_status variable is global for GUI to track it.
*
* @author Vatsal Sheth & Puneet Bansal
 *****************************************************************/
#ifndef MAIN_H_
#define MAIN_H_

#include "motor.h"

#define On 0x01
#define Off 0x00

uint8_t motor_status, motor_on_flag, motor_off_flag, rpm_prev;

void unused_port_init();
void pwm_port_init();
void hall_port_init();
#endif /* MAIN_H_ */

/*****************************************************************************
* @motor.c
* @This file contains functions for motor control, feedback and drive.
* It manages ADC, comparator, PWM, GPIO drives, sensor interface, control logic
*
* @author Vatsal Sheth & Puneet Bansal
*****************************************************************************/

#include "motor.h"

//Commutation Sequence lookup table
#ifdef CW
   Hall_commutaion Hall_Lookup[]={
                        {Rising_edge,L_U | L_V | L_W, 0},
                        {Rising_edge,(H_V | L_U | L_V), H_V},
```

```
                        {Rising_edge,(H_U | L_U | L_W), H_U},
                        {Falling_edge,(H_U | L_U | L_V), H_U},
                        {Rising_edge,(H_W | L_V | L_W), H_W},
                        {Falling_edge,(H_V | L_V | L_W), H_V},
                        {Falling_edge,(H_W | L_U | L_W), H_W}
                       };
#endif

//Comarator initializaton
void comp_init()
{
   uint8_t i;

   COMP_E1->CTL0 = COMP_E_CTL0_IPEN | COMP_E_CTL0_IPSEL_1;          // Enable V+, input channel
CE1
   COMP_E1->CTL1 = COMP_E_CTL1_PWRMD_0;    // high speed power mode
   COMP_E1->CTL2 = COMP_E_CTL2_CEREFL_1 |   // VREF 1.2V
               COMP_E_CTL2_RS_2 |      // Ladder enabled by COMP_E_CTL2_RS_2
               COMP_E_CTL2_RSEL|       //VREF is applied to -terminal
               COMP_E_CTL2_REF0_12;    // COMP_E_CTL2_REF0_12 (12/32)*1.2V = 0.45V
  COMP_E1->CTL3 = BIT1;                // Input Buffer Disable @P1.1/CE1
  COMP_E1->CTL1 |= COMP_E_CTL1_ON;        // Turn On Comparator_E
  COMP_E1->INT |= COMP_E_INT_IE;
  NVIC->ISER[0] |= 1 << ((COMP_E1_IRQn) & 31);

  for (i = 0; i < 75; i++);             // delay for the reference to settle
}

//motor start sequence
void motor_on()
{
   uint8_t tmp;

   interrupt_bypass=0;
   pid_flag = 0;
   rpm_count = 0;
   duty_cycle = startup_duty_cycle;
```

```
    current_sensing=0;

    P6->DIR = BIT4;
    P6->OUT|= BIT4; //power switch on

    tmp=(P5->IN & 0x07);
    mosfet_drive(Hall_Lookup[tmp].Mosfet_Value);
    pwm_drive_pin = Hall_Lookup[tmp].Mosfet_H_Value;
    P5->IES = Hall_Lookup[tmp].interrupt_dir;
}

//motor off sequence
void motor_off()
{
    P6->OUT &= ~BIT4;
    P3->OUT = 0xa4; //drive all mosfet to cut off state
    current_sensing=0;
    RPM_avg = 0;
}

//motor initialization
void motor_init()
{
    pwm_init();
    comp_init();
    time_diff_init();
    adc_init();

    NVIC->ISER[1] |= 1 << ((PORT5_IRQn) & 31);
}

//pwm initialization
void pwm_init()
{
    TIMER_A0->CCR[0] = PWM_Period-1;
    TIMER_A0->CCTL[1] = TIMER_A_CCTLN_CCIE;
    TIMER_A0->CCR[1] =  (uint32_t)((PWM_Period*duty_cycle)/100);
```

```
   TIMER_A0->CTL = TIMER_A_CTL_SSEL__SMCLK | TIMER_A_CTL_MC__UP | TIMER_A_CTL_CLR |
TIMER_A_CTL_IE;         //UP mode timer with smclk and overflow interrupt
   NVIC->ISER[0] |= 1 << ((TA0_N_IRQn)&31);
}

//refrence timer for speed and error measurement for motor control
void time_diff_init()
{
   time_diff = 0;
   current_time = 0;
   TIMER_A1->CCR[1] = 30000;   //10ms compare value
   TIMER_A1->CCTL[1] = TIMER_A_CCTLN_CCIE;
   TIMER_A1->CTL = TIMER_A_CTL_SSEL__SMCLK | TIMER_A_CTL_MC__CONTINUOUS |
TIMER_A_CTL_CLR;         //Continuous mode timer with smclk
   NVIC->ISER[0] |= 1 << ((TA1_N_IRQn)&31);
}

//adc initialization in one shot mode
void adc_init()
{
   while(REF_A->CTL0 & REF_A_CTL0_GENBUSY);     // Wait till ref generator busy
   REF_A->CTL0 |= REF_A_CTL0_VSEL_0 | REF_A_CTL0_ON;     // 1.2V reference

   // Configure ADC - Pulse sample mode
   ADC14->CTL0 |= ADC14_CTL0_SHT0_2 | ADC14_CTL0_ON | ADC14_CTL0_SHP;    // SAMPCON from
sampling timer
   ADC14->MCTL[0] = ADC14_MCTLN_VRSEL_1 | ADC14_MCTLN_INCH_10;          // ADC input ch 10
(PIN P4.3)
   ADC14->IER0 = 0x0001;          // ADC_IFG upon conv result-ADCMEM0 because single conversion and
CSTARTADDR reset value is 0. So value will be in mem[0].

   while(!(REF_A->CTL0 & REF_A_CTL0_GENRDY));   // Wait for reference generator to settle
   ADC14->CTL0 |= ADC14_CTL0_ENC;

   NVIC->ISER[0] |= 1 << ((ADC14_IRQn) & 31);
}
```

```
//calculate new duty cycle based on error and calculate speed
void calc_param()
{
   if(pid_flag == 1)
   {
      pid_flag = 0;
      speed(time_diff);
      pid_calc();
      ADC14->CTL0 |= ADC14_CTL0_SC;   //trigger ADC sample and conversion
   }
}

//proportional control logic
void pid_calc()
{
   if(error<300)
   {
      duty_cycle -= (float)((float)error/700);
   }
   else if(error<500)
   {
      duty_cycle -= (float)((float)error/500);
   }
   else
   {
      duty_cycle -= (float)((float)error/300);
   }

   if(duty_cycle > 75) //Ensure duty cycle never exceeds 75%
   {
      duty_cycle = 75;
   }
   else if(duty_cycle<5)   //Ensure duty cycle never goes below 5%
   {
      duty_cycle = 5;
   }
```

```
   TIMER_A0->CCR[1] =  (uint32_t)(((float)(PWM_Period*duty_cycle))/100);   //update compare value to change
duty cycle
}

//calculate average RPM of last 10 RPM values
void speed(uint16_t diff)
{
   uint8_t i=0;
   uint16_t tmp=0;

   tmp = (3 * 1000000 * 60)/(diff * 24);
   if(tmp<6000)
   {
      RPM[rpm_count] = tmp;
      rpm_count = (rpm_count+1)%10;
      for(i=0; i<10; i++)
      {
         tmp += RPM[i];
      }

      RPM_avg = (tmp/10);
   }
}

//Comparator isr for overload protection
void COMP_E1_IRQHandler()
{
   if(COMP_E1->IV & COMP_E_INT_IFG)
   {
      motor_off();
   }
}

//refrence timer isr to generate flags at 10ms to calculate control and speed values
void TA1_N_IRQHandler(void)
{
   TIMER_A1->CCTL[1] &= ~TIMER_A_CCTLN_CCIFG;
```

```
    TIMER_A1->CCR[1] += 30000;
    pid_flag = 1;
}

//pwm generation logic on GPIOs where required as per current hall state
void TA0_N_IRQHandler(void)
{
    if(TIMER_A0->CTL & TIMER_A_CTL_IFG)              //set all R G B bits on overflow
    {
        TIMER_A0->CTL &= ~TIMER_A_CTL_IFG;
        P3->OUT |= pwm_drive_pin;
    }
    if(TIMER_A0->CCTL[1] & TIMER_A_CCTLN_CCIFG)      //clear R bit on compare 1
    {
        TIMER_A0->CCTL[1] &= ~TIMER_A_CCTLN_CCIFG;
        P3->OUT &= ~pwm_drive_pin;
    }
}

//hall sensor GPIO ISR. Alternate ISR are skipped for time difference calculation as a firmware fix for not accessible
GPIO with glitch filter
void PORT5_IRQHandler(void)
{
    uint8_t hall;
    uint16_t tmp;

    if((interrupt_bypass %2 )==0)
    {
        tmp = TA1R;
    }

    //read the hall state and control drive
    if(P5->IFG & (BIT0 | BIT1 | BIT2))
    {
        P5->IFG = 0X00;
        hall = (P5->IN & 0x07);
        mosfet_drive(Hall_Lookup[hall].Mosfet_Value);
```

```
        pwm_drive_pin = Hall_Lookup[hall].Mosfet_H_Value;
        P5->IES = Hall_Lookup[hall].interrupt_dir;
    }

    //alternate time and error measurement
    if((interrupt_bypass %2 )==0)
    {
        if(tmp >= current_time)
        {
          time_diff = tmp-current_time;
        }
        else
        {
          time_diff = 65535 - (current_time - tmp);
        }
        update_error();
        current_time=tmp;
    }

    interrupt_bypass++;
}

//ADC conversion complete isr to calculate current value
void ADC14_IRQHandler(void)     //read adc value from memory and set the local flag
{
    uint16_t tmp;
    tmp= ADC14->MEM[0];
    current_sensing= ((tmp* 1.2) / (16384 * 0.6));
}

//drive required GPIO as per current hall state
void mosfet_drive(uint8_t mv)
{
    P3->OUT=mv;
}

//update values whenever set RPM is changed from GUI
```

```
void update_desired_rpm()
{
   expected_time = (3 * 1000000 * 60)/(RPM_user * 24); //expected time between hall state changes
   duty_cycle = (((float)(75*RPM_user))/5500); //jump duty cycle as per observation in control table for updated set speed
   TIMER_A0->CCR[1] =  (uint32_t)(((float)(PWM_Period*duty_cycle))/100);   //update compare value on duty cycle
}

//update the error
void update_error()
{
   error = expected_time - time_diff;
}

/*****************************************************************************
* @motor.c
* @This file contains functions for motor control, feedback and drive.
* It manages ADC, comparator, PWM, GPIO drives, sensor interface, control logic
*
* @author Vatsal Sheth & Puneet Bansal
*****************************************************************************/

#include "motor.h"

//Commutation Sequence lookup table
#ifdef CW
   Hall_commutaion Hall_Lookup[]={
                        {Rising_edge,L_U | L_V | L_W, 0},
                        {Rising_edge,(H_V | L_U | L_V), H_V},
                        {Rising_edge,(H_U | L_U | L_W), H_U},
                        {Falling_edge,(H_U | L_U | L_V), H_U},
                        {Rising_edge,(H_W | L_V | L_W), H_W},
                        {Falling_edge,(H_V | L_V | L_W), H_V},
                        {Falling_edge,(H_W | L_U | L_W), H_W}
```

```
                        };
#endif

//Comarator initializaton
void comp_init()
{
   uint8_t i;

   COMP_E1->CTL0 = COMP_E_CTL0_IPEN | COMP_E_CTL0_IPSEL_1;          // Enable V+, input channel
CE1
   COMP_E1->CTL1 = COMP_E_CTL1_PWRMD_0;    // high speed power mode
   COMP_E1->CTL2 = COMP_E_CTL2_CEREFL_1 |   // VREF 1.2V
               COMP_E_CTL2_RS_2 |      // Ladder enabled by COMP_E_CTL2_RS_2
               COMP_E_CTL2_RSEL|       //VREF is applied to -terminal
               COMP_E_CTL2_REF0_12;    // COMP_E_CTL2_REF0_12 (12/32)*1.2V = 0.45V
  COMP_E1->CTL3 = BIT1;                // Input Buffer Disable @P1.1/CE1
  COMP_E1->CTL1 |= COMP_E_CTL1_ON;        // Turn On Comparator_E
  COMP_E1->INT |= COMP_E_INT_IE;
  NVIC->ISER[0] |= 1 << ((COMP_E1_IRQn) & 31);

  for (i = 0; i < 75; i++);             // delay for the reference to settle
}

//motor start sequence
void motor_on()
{
   uint8_t tmp;

   interrupt_bypass=0;
   pid_flag = 0;
   rpm_count = 0;
   duty_cycle = startup_duty_cycle;
   current_sensing=0;
```

```
    P6->DIR = BIT4;
    P6->OUT|= BIT4; //power switch on

    tmp=(P5->IN & 0x07);
    mosfet_drive(Hall_Lookup[tmp].Mosfet_Value);
    pwm_drive_pin = Hall_Lookup[tmp].Mosfet_H_Value;
    P5->IES = Hall_Lookup[tmp].interrupt_dir;
}

//motor off sequence
void motor_off()
{
    P6->OUT &= ~BIT4;
    P3->OUT = 0xa4; //drive all mosfet to cut off state
    current_sensing=0;
    RPM_avg = 0;
}

//motor initialization
void motor_init()
{
    pwm_init();
    comp_init();
    time_diff_init();
    adc_init();

    NVIC->ISER[1] |= 1 << ((PORT5_IRQn) & 31);
}

//pwm initialization
void pwm_init()
{
    TIMER_A0->CCR[0] = PWM_Period-1;
```

```
    TIMER_A0->CCTL[1] = TIMER_A_CCTLN_CCIE;
    TIMER_A0->CCR[1] =  (uint32_t)((PWM_Period*duty_cycle)/100);
    TIMER_A0->CTL = TIMER_A_CTL_SSEL__SMCLK | TIMER_A_CTL_MC__UP | TIMER_A_CTL_CLR |
TIMER_A_CTL_IE;         //UP mode timer with smclk and overflow interrupt
    NVIC->ISER[0] |= 1 << ((TA0_N_IRQn)&31);
}

//refrence timer for speed and error measurement for motor control
void time_diff_init()
{
    time_diff = 0;
    current_time = 0;
    TIMER_A1->CCR[1] = 30000;   //10ms compare value
    TIMER_A1->CCTL[1] = TIMER_A_CCTLN_CCIE;
    TIMER_A1->CTL = TIMER_A_CTL_SSEL__SMCLK | TIMER_A_CTL_MC__CONTINUOUS |
TIMER_A_CTL_CLR;         //Continuous mode timer with smclk
    NVIC->ISER[0] |= 1 << ((TA1_N_IRQn)&31);
}

//adc initialization in one shot mode
void adc_init()
{
    while(REF_A->CTL0 & REF_A_CTL0_GENBUSY);     // Wait till ref generator busy
    REF_A->CTL0 |= REF_A_CTL0_VSEL_0 | REF_A_CTL0_ON;     // 1.2V reference

    // Configure ADC - Pulse sample mode
    ADC14->CTL0 |= ADC14_CTL0_SHT0_2 | ADC14_CTL0_ON | ADC14_CTL0_SHP;    // SAMPCON from
sampling timer
    ADC14->MCTL[0] = ADC14_MCTLN_VRSEL_1 | ADC14_MCTLN_INCH_10;          // ADC input ch 10
(PIN P4.3)
    ADC14->IER0 = 0x0001;          // ADC_IFG upon conv result-ADCMEM0 because single conversion and
CSTARTADDR reset value is 0. So value will be in mem[0].

    while(!(REF_A->CTL0 & REF_A_CTL0_GENRDY));   // Wait for reference generator to settle
    ADC14->CTL0 |= ADC14_CTL0_ENC;
```

```
    NVIC->ISER[0] |= 1 << ((ADC14_IRQn) & 31);
}

//calculate new duty cycle based on error and calculate speed
void calc_param()
{
    if(pid_flag == 1)
    {
        pid_flag = 0;
        speed(time_diff);
        pid_calc();
        ADC14->CTL0 |= ADC14_CTL0_SC;   //trigger ADC sample and conversion
    }
}

//proportional control logic
void pid_calc()
{
    if(error<300)
    {
        duty_cycle -= (float)((float)error/700);
    }
    else if(error<500)
    {
        duty_cycle -= (float)((float)error/500);
    }
    else
    {
        duty_cycle -= (float)((float)error/300);
    }

    if(duty_cycle > 75) //Ensure duty cycle never exceeds 75%
```

```
    {
        duty_cycle = 75;
    }
    else if(duty_cycle<5)   //Ensure duty cycle never goes below 5%
    {
        duty_cycle = 5;
    }
    TIMER_A0->CCR[1] =  (uint32_t)(((float)(PWM_Period*duty_cycle))/100);   //update compare value to change duty cycle
}

//calculate average RPM of last 10 RPM values
void speed(uint16_t diff)
{
    uint8_t i=0;
    uint16_t tmp=0;

    tmp = (3 * 1000000 * 60)/(diff * 24);
    if(tmp<6000)
    {
        RPM[rpm_count] = tmp;
        rpm_count = (rpm_count+1)%10;
        for(i=0; i<10; i++)
        {
            tmp += RPM[i];
        }

        RPM_avg = (tmp/10);
    }
}

//Comparator isr for overload protection
void COMP_E1_IRQHandler()
{
```

```
    if(COMP_E1->IV & COMP_E_INT_IFG)
    {
        motor_off();
    }
}

//refrence timer isr to generate flags at 10ms to calculate control and speed values
void TA1_N_IRQHandler(void)
{
    TIMER_A1->CCTL[1] &= ~TIMER_A_CCTLN_CCIFG;
    TIMER_A1->CCR[1] += 30000;
    pid_flag = 1;
}

//pwm generation logic on GPIOs where required as per current hall state
void TA0_N_IRQHandler(void)
{
    if(TIMER_A0->CTL & TIMER_A_CTL_IFG)              //set all R G B bits on overflow
    {
        TIMER_A0->CTL &= ~TIMER_A_CTL_IFG;
        P3->OUT |= pwm_drive_pin;
    }
    if(TIMER_A0->CCTL[1] & TIMER_A_CCTLN_CCIFG)      //clear R bit on compare 1
    {
        TIMER_A0->CCTL[1] &= ~TIMER_A_CCTLN_CCIFG;
        P3->OUT &= ~pwm_drive_pin;
    }
}

//hall sensor GPIO ISR. Alternate ISR are skipped for time difference calculation as a firmware fix for not accessible GPIO with glitch filter
void PORT5_IRQHandler(void)
{
    uint8_t hall;
```

```
uint16_t tmp;

if((interrupt_bypass %2 )==0)
{
   tmp = TA1R;
}

//read the hall state and control drive
if(P5->IFG & (BIT0 | BIT1 | BIT2))
{
   P5->IFG = 0X00;
   hall = (P5->IN & 0x07);
   mosfet_drive(Hall_Lookup[hall].Mosfet_Value);
   pwm_drive_pin = Hall_Lookup[hall].Mosfet_H_Value;
   P5->IES = Hall_Lookup[hall].interrupt_dir;
}

//alternate time and error measurement
if((interrupt_bypass %2 )==0)
{
   if(tmp >= current_time)
   {
    time_diff = tmp-current_time;
   }
   else
   {
    time_diff = 65535 - (current_time - tmp);
   }
   update_error();
   current_time=tmp;
}

interrupt_bypass++;
```

```
}

//ADC conversion complete isr to calculate current value
void ADC14_IRQHandler(void)     //read adc value from memory and set the local flag
{
   uint16_t tmp;
   tmp= ADC14->MEM[0];
   current_sensing= ((tmp* 1.2) / (16384 * 0.6));
}

//drive required GPIO as per current hall state
void mosfet_drive(uint8_t mv)
{
   P3->OUT=mv;
}

//update values whenever set RPM is changed from GUI
void update_desired_rpm()
{
   expected_time = (3 * 1000000 * 60)/(RPM_user * 24); //expected time between hall state changes
   duty_cycle = (((float)(75*RPM_user))/5500); //jump duty cycle as per observation in control table for updated set speed
   TIMER_A0->CCR[1] =  (uint32_t)(((float)(PWM_Period*duty_cycle))/100);   //update compare value on duty cycle
}

//update the error
void update_error()
{
   error = expected_time - time_diff;
}
```

```
/*****************************************************************************
* @motor.h
* @This file contains declaration of functions and variables used in main.c
* RPM_avg and current_sensing variables are global for GUI to track it.
*
* @author Vatsal Sheth & Puneet Bansal
*****************************************************************************/

#ifndef MOTOR_H_
#define MOTOR_H_

#include "msp.h"

#define H_U 0x01
#define L_U 0x04
#define H_V 0x08
#define L_V 0x20
#define H_W 0x40
#define L_W 0x80

#define Rising_edge 0x00
#define Falling_edge 0x07

#define PWM_Period 375  //8Khz
#define startup_duty_cycle 10      //starting duty cycle 10%

uint16_t count1,count2;
uint16_t arr1[100],arr2[100];



#define CW

typedef struct
```

```
{
   uint8_t interrupt_dir;
   uint8_t Mosfet_Value;
   uint8_t Mosfet_H_Value;
}Hall_commutaion;

volatile uint8_t pid_flag, rpm_count;
volatile uint8_t pwm_drive_pin,interrupt_bypass;
volatile uint16_t time_diff, current_time, RPM_avg, RPM[10], RPM_user, expected_time;
volatile int32_t error;
volatile float current_sensing, duty_cycle;

void motor_init();
void pwm_init();
void mosfet_drive(uint8_t mv);
void comp_init();
void motor_on();
void motor_off();
void calc_param();
void pid_calc();
void speed(uint16_t diff);
void time_diff_init();
void adc_init();
void update_desired_rpm();
void update_error();
#endif /* MOTOR_H_ */
```