IMPLEMENTING FIELD ORIENTED CONTROL OF BRUSHLESS DC MOTOR

Project Report submitted

In Partial Fulfilment of the Requirements for the award of the degree

of

Bachelor of Technology

In

Electrical and Electronics Engineering

by

Dunaka Bharghavi	**(Roll No: 178W5A0203)**
Koganti Harsha Vardhan	**(Roll No: 168W1A0224)**
Pamarthi Yamini	**(Roll No: 168W1A0239)**
Yagati Pavani	**(Roll No: 168W1A0258)**

Under the guidance of

R MADHUSUDHANA RAO, MTECH

Assistant Professor

DEPARTMENT OF

ELECTRICAL AND ELECTRONICS ENGINEERING

VELAGAPUDI RAMAKRISHNA

SIDDHARTHA ENGINEERING COLLEGE

(AUTONOMOUS)

(Affiliated to JNTUK, Approved by UGC, Accredited By NBA)

April 2020

CERTIFICATE

This is to certify that, the project report entitled ***"IMPLEMENTING FIELD ORIENTED CONTROL OF BRUSHLESS DC MOTOR"*** *is the bonafide record of work carried out by* ***Dunaka Bharghavi (178W5A0203), Koganti Harsha Vardhan (168W1A0224), Pamarthi Yamini (168W1A0239), Yagati Pavani (168W1A0258)*****,*** *under my guidance and submitted on partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Electrical and Electronics Engineering to Jawaharlal Nehru Technological University Kakinada during the academic year 2019- 2020.*

R Madhusudhana Rao, MTECH
Assistant Professor,
Department of E.E.E.

Dr.P.V.R.L.Narasimham
Professor & Head,
Department of E.E.E.

ACKNOWLEDGMENT

We are highly indebted to our guide **Mr. R Madhusudhana Rao** for his guidance and continuous support in completing the project. We sincerely thank **Dr. P V R L NARASIMHAM**, Professor & Head, Department of EEE for his encouragement and suggestion during the project. We express our profound gratitude to Principal **Dr. A V RATNAPRASAD** and our management for providing the facilities during the survey. We would like to thank staff members of our department and friends for extending their cooperation for the successful completion of the project.

PROJECT ASSOCIATES

Dunaka Bharghavi (178W5A0203)

Koganti Harsha Vardhan (168W1A0224)

Pamarthi Yamini (168W1A0239)

Yagati Pavani (168W1A0258)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This project focuses on various aspects of space vector modulation-based field-oriented control of trapezoidal permanent magnet synchronous motors. The purpose is to introduce concisely the fundamental theory, main results and practical applications of field-oriented control for BLDC motors. The primary focus is on controlling the motor in the d-q axis using Field Oriented Control techniques along with the generation of space vector modulated pulses for the inverter driving the BLDC. An STM32F407VGT6 microcontroller which is based on ARM CORTEX M4 architecture is utilized to control the motor drivers of the BLDC motor. The vector control method improves the system performance with low torque ripple, and high efficiency is also introduced thus making it suitable for immense applications such as Air conditioners, Automotive electrical power steering, Machining tools, etc.

Chapter 1:

INTRODUCTION

Brushless DC motors (BLDC) are variable frequency permanent magnet synchronous motors. They are constructed with a permanent magnet rotor and wire-wound stator poles. Electrical energy is converted to mechanical energy by the magnetic attractive forces between the permanent magnet rotor and a rotating magnetic field induced in the wound stator poles. Most BLDC motors have a three-phase winding topology with star connection. A motor with this topology is driven by energizing two phases at a time.

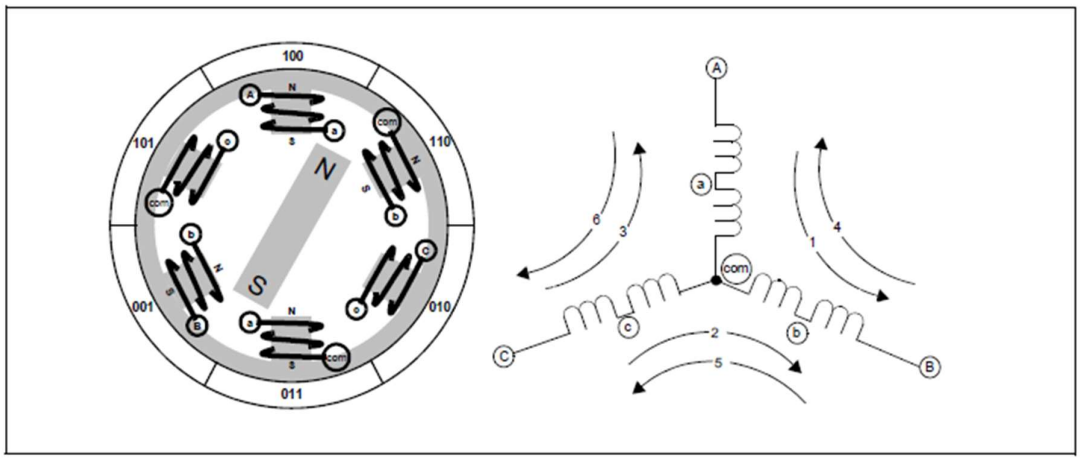


Fig. 1.1 BLDC Cross-Section and Winding Topology

Since it is brushless it requires an electronic circuit for commutation. For BLDC commutation we generally use a three-phase inverter circuit. The three-phase inverter consists of six switching elements arranged into pairs, known as inverter legs. The output voltage of the inverter leg depends on the DC bus voltage and the switching status. One of the most common methods used for inverter switching is Pulse width modulation (PWM) Techniques. In this technique, we control the output voltage by varying the on-off time of the switching elements in the inverter. The most popular PWM techniques used to control BLDC are Trapezoidal PWM, Sinusoidal PWM (SPWM), and Space Vector PWM (SVPWM). With the increase in the capabilities of the microcontrollers for motor control,

SVPWM has become one of the prominent PWM methods. In SVPWM we compute the on-off time for each switch course.

To run a motor for intended applications different kinds of speed control methods are implemented. The most common motor speed control methods are Voltage control, Frequency control, V/f control, FOC (Field Oriented Control), etc. The field-oriented control (FOC) is one of the most suitable and popular speed control technique for the applications where load torque changes are dynamic. In the field-oriented control, a d-q coordinates reference frame locked to the rotor flux vector is used to achieve decoupling between the motor flux and torque. They can be thus separately controlled by stator direct-axis current and quadrature-axis current respectively, like in a dc motor.

Today the use of BLDC motor has increased and it is competing with induction motor and DC motors. The Brushless DC motor (BLDC) shows many advantages over the induction motor. The BLDC motor has trapezoidal back-emf characteristics and requires constant stator current at the middle to phase voltage waveform to produce a constant torque. The torque-speed characteristics of the BLDC motor are similar to that of a DC motor. Permanent magnet synchronous motor (PMSM) exhibits sinusoidal back-emf and to produce constant torque it's required a sinusoidal shaped current. PMSM is similar to the synchronous machine with a permanent magnet rotor instead of the field winding.

1.1 Motivation:

Since the end of the 19th century, Brushed DC motors are used for different applications and it was the most used motor for commercial operations. With the development of brushless DC motors, it gives competition to the use of Brushed DC motor. BLDC motor was supplied by the three-phase inverter. PWM technique is used to achieve the desired voltage and frequency from the inverter. Different PWM techniques are used to control the inverter output, the SPWM is the most commonly used techniques. In the mid-'80s SVPWM technique is derived and its use also increases. SVPWM is easy to implement using microcontrollers and also

reduces harmonics in the output, reduces the switching frequency of the inverter, and utilizes dc-link voltage better.

1.2 Problem Definition:

The Trapezoidal PWM control of BLDC motor incur more transient losses, torque ripples, slow to respond to dynamic torque changes and it causes heavy environmental noises under high-speed operation. The transient losses reduce the efficiency of the motor. Under loaded conditions, these losses cause the motor winding to heat up rapidly which can melt the insulation of the winding leading to a short circuit. The torque ripples can cause damage to the motor ball bearings around the shaft. So, the overall life span of the motor is reduced.

1.3 Objective

The main objective of this project is to suppress the transient losses incurred in the motor, eliminate torque ripples and operate the BLDC motor at variable speed under dynamic load changes by applying a space vector modulation-based field-oriented control method.

Chapter 2:

BLDC MOTOR OPERATION AND FIELD ORIENTED CONTROL

2.1 Working Principle of BLDC motor:

BLDC motor works on the principle similar to that of a conventional DC motor, i.e., the Lorentz force law which states that whenever a current-carrying conductor placed in a magnetic field it experiences a force. As a consequence of the reaction force, the magnet will experience an equal and opposite force. In case BLDC motor, the current-carrying conductor is stationary while the permanent magnet moves.

When the stator coils are electrically switched by a supply source, it becomes an electromagnet and starts producing the uniform field in the air gap. Though the source of supply is DC, switching makes to generate an AC voltage waveform with trapezoidal shape. Due to the force of interaction between electromagnet stator and permanent magnet rotor, the rotor continues to rotate.

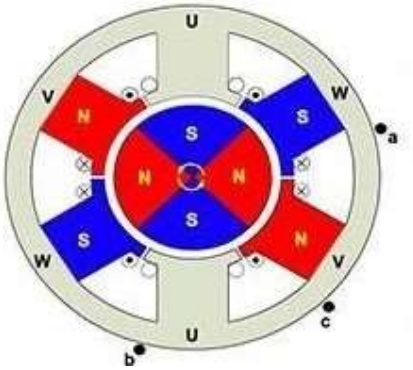


Fig. 2.1

Consider the figure above in which the motor stator is excited based on different switching states. With the switching of windings as High and Low signals, corresponding winding energized as North and South poles. The permanent magnet rotor with North and South poles align with stator poles causing the motor to rotate. Observe that motor produces torque because of the development of attraction forces (when North-South or

South-North alignment) and repulsion forces (when North-North or South-South alignment). In this way motor moves in a clockwise direction.

2.2 Powering of BLDC motor:

By far the most common configuration for sequentially applying current to a three-phase BLDC motor is to use three pairs of power MOSFETs arranged in a bridge structure. Each pair governs the switching of one phase of the motor. In a typical arrangement, the high-side MOSFETs are controlled using pulse-width modulation (PWM) which converts the input DC voltage into a modulated driving voltage. The use of PWM allows the start-up current to be limited and offers precise control over speed and torque. The PWM frequency is a trade-off between the switching losses that occur at high frequencies and the ripple currents that occur at low frequencies, and which in extreme cases, can damage the motor. Typically, designers use a PWM frequency of at least an order of magnitude higher than the maximum motor rotation speed.

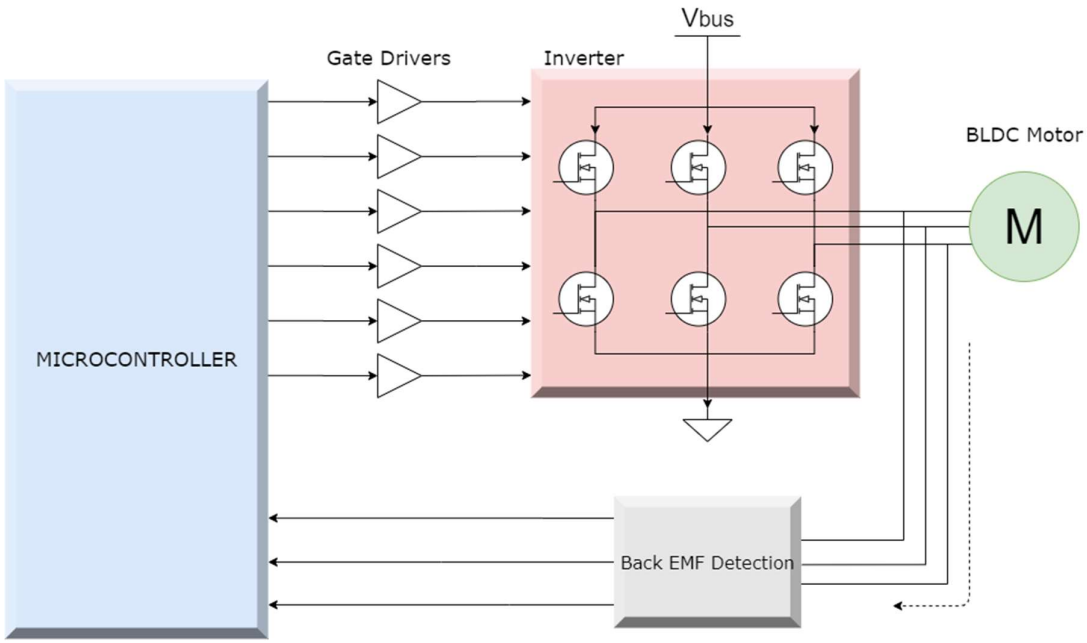


Fig. 2.2 BLDC Motor Control Block Diagram

2.3 Commutation Sequence of BLDC Motor:

There are six commutation states in each electrical revolution of the BLDC motor. Each state drives one motor phase high and one motor phase low. The third phase is undriven. Each sequence lasts a 60° electrical angle. Ideal phase currents last 120° for each half of the electrical revolution and is displaced from each other phases by 120° electrical angle.

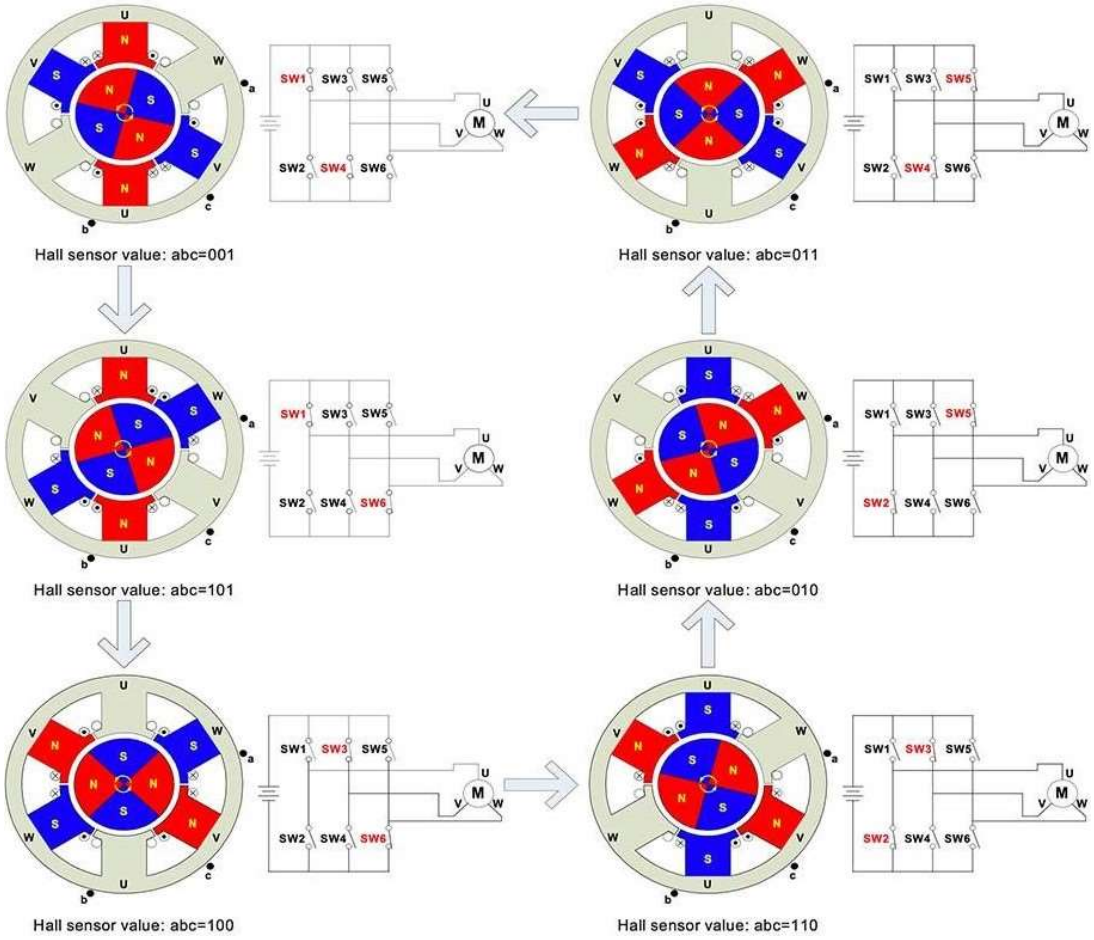


Fig. 2.3 BLDC Motor Switching Sequence

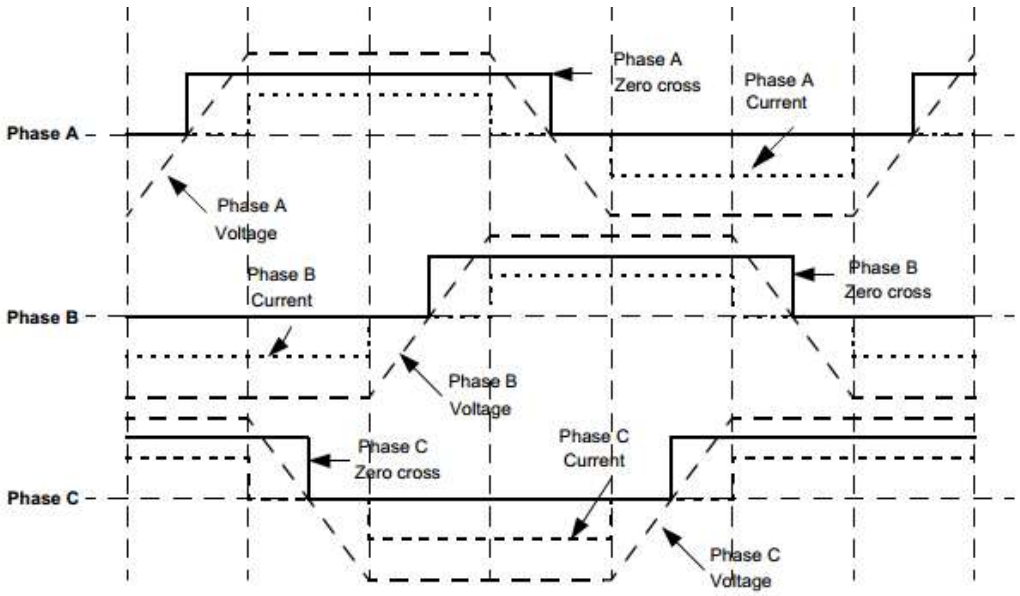


Fig. 2.4 Back EMF and Current Waveforms of BLDC Motor

The above figures show the commutation sequence, voltage, and current waveforms of an ideal BLDC motor. The clockwise or the anticlockwise rotation of the motor depends on the direction of the switching sequence of the three-phase inverter.

2.4 Positional Feedback of BLDC motor:

Because there is no mechanical or electrical contact between the stator and rotor of the BLDC motor, alternative arrangements are required to indicate the relative positions of the rotor to facilitate motor control. BLDC motors use one of two methods to achieve this, either employing Hall sensors or measuring back EMF. Sensors work well, but add cost, increase complexity (due to the additional wiring), and reduce reliability. Measuring back EMF addresses these drawbacks. It is possible to determine when to commutate the motor drive voltages by sensing the back EMF voltage on an undriven motor terminal during one of the drive phases.

There are several techniques for measuring the back EMF. The simplest is to compare the back EMF to half the DC bus voltage using a comparator. The main drawback of this simple comparator method is that the three windings may not have identical characteristics resulting in a positive or negative phase shift from the actual zero-crossing point. The

motor likely still run, but may draw excessive current. The solution is to generate a virtual neutral point by using three resistor networks connected in parallel with the motor windings. The back EMF is then compared with the virtual neutral point.

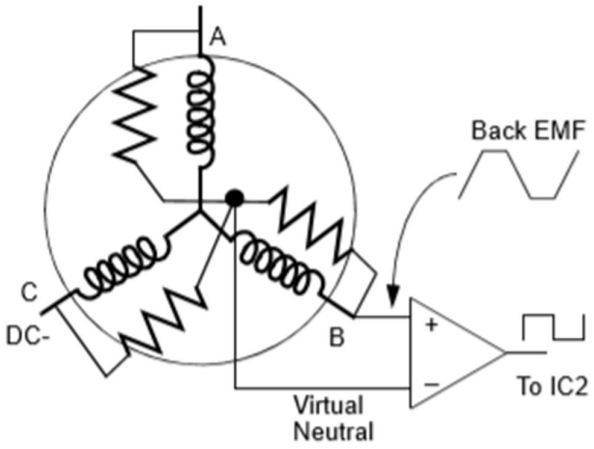


Fig. 2.5 BEMF Detection by Generating Virtual Neutral Point

There are several disadvantages to sensorless control:

- The motor must be moving at a minimum rate to generate sufficient back EMF to be sensed. Abrupt changes to the motor load can cause the BEMF drive loop to go out of the lock.
- The BEMF voltage can be measured only when the motor speed is within a limited range of the ideal commutation rate for the applied voltage.
- Commutation at rates faster than the ideal rate will result in a discontinuous motor response

If low cost is a primary concern and low-speed motor operation is not a requirement and the motor load is not expected to change rapidly then sensorless control may be the better choice.

2.5 Simplified Circuit of BLDC motor:

Let us consider what happens when a voltage is applied to a DC motor.

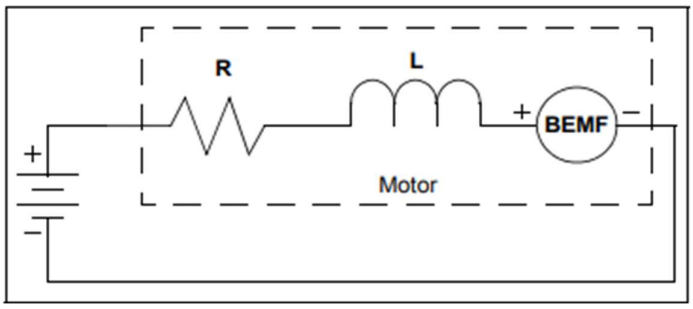


Fig. 2.6 Simplified Electrical Model of DC Motor

When the rotor is stationary, the only resistance to current flow is the impedance of the electromagnetic coils. The impedance is comprised of the parasitic resistance of the copper in the windings and the parasitic inductance of the windings themselves. The resistance and inductance are very small by design, so start-up currents would be very large, if not limited. When the motor is spinning, the permanent magnet rotor moving past the stator coils induces an electrical potential in the coils called Back Electromotive Force, or BEMF. BEMF is directly proportional to the motor speed and is determined from the motor voltage constant K_V.

$$RPM = K_V \times Volts \tag{2.1}$$

$$BEMF = RPM / K_V \tag{2.2}$$

In an ideal motor, R and L are zero, and the motor will spin at a rate such that the BEMF exactly equals the applied voltage. The current that a motor draws is directly proportional to the torque load on the motor shaft. Motor current is determined from the motor torque constant K_T.

$$Torque = K_T \times Amps \tag{2.3}$$

An interesting fact about K_T and K_V is that their product is the same for all motors i.e.

$$K_V \times K_T = 1 \tag{2.4}$$

2.5.1 Determining the BEMF:

The BEMF, relative to the coil common connection point, generated by each of the motor coils, can be expressed as

$$B_{BEMF} = \sin (\alpha) \tag{2.5}$$

$$C_{BEMF} = \sin (\alpha - 2\pi/ 3) \quad (2.6)$$

$$A_{BEMF} = \sin (\alpha - 4\pi/ 3) \quad (2.7)$$

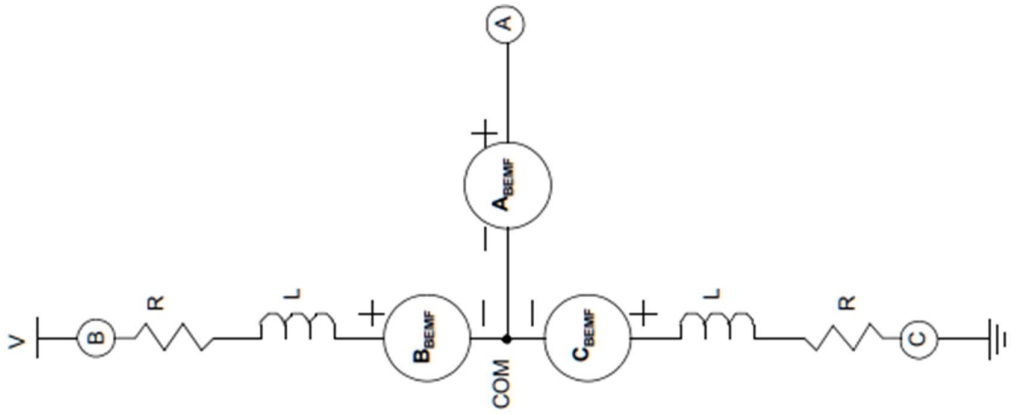


Fig. 2.7

The above figure shows the equivalent circuit of the motor with coils B and C are driven while coil A is undriven and available for BEMF measurement. At the commutation frequency, the L's are negligible. The R's are assumed to be equal. The L and R components are not shown in the A branch since no significant current flows in this part of the circuit so those components can be ignored. The BEMF generated by the B and C coils in tandem can be expressed as

$$BEMF_{BC} = B_{BEMF} - C_{BEMF} \quad (2.8)$$

The sign reversal of C_{BEMF} is due to moving the reference point from the common connection to the ground.

At full speed, the applied DC voltage is equivalent to the RMS BEMF voltage in that 60-degree range. In terms of the peak BEMF generated by anyone winding, the RMS BEMF voltage across two of the windings can be expressed as

$$BEMF_{RMS} = \sqrt{\frac{3}{\pi}\int_{\frac{\pi}{2}}^{\frac{\pi}{6}}(\sin\alpha - \sin(\alpha - \frac{2\pi}{3}))^2\, d\alpha} = 1.6554 \quad (2.9)$$

Let us consider the expected BEMF at the undriven motor terminal. Since the applied voltage is pulse-width modulated, the drive alternates

between on and off throughout the phase time. The BEMF, relative to the ground, seen at the A terminal when the drive is on, can be expressed as

$$BEMF_A = \frac{[V - (B_{BEMF} - C_{BEMF})]R}{2R} - C_{BEMF} + A_{BEMF} = \frac{[V - (B_{BEMF} - C_{BEMF})]}{2} - C_{BEMF} + A_{BEMF} \quad (2.10)$$

Notice that the winding resistance cancels out, so resistive voltage drop, due to motor torque load, is not a factor when measuring BEMF.

The BEMF, relative to the ground, seen at the A terminal when the drive is off can be expressed as shown

$$BEMF_A = A_{BEMF} - C_{BEMF} \quad (2.11)$$

2.6 Field Oriented Control of BLDC motor:

Field Oriented Control is one of the methods used in variable frequency drives or variable speed drives to control the torque (and thus the speed) of three-phase AC electric motors by controlling the current. With FOC, the torque and the flux can be controlled independently. FOC provides a faster dynamic response than is required for applications like automotive electric power steering. There is no torque ripple and smoother, accurate motor control can be achieved at low and high speeds using FOC.

The torque of a BLDC motor is at a maximum when the stator and the rotor magnetic fields are orthogonal to each other. In FOC, the stator currents are measured and adjusted so that the angle between the rotor and stator flux is 90 degrees to achieve the maximum torque (as shown in the following figures).

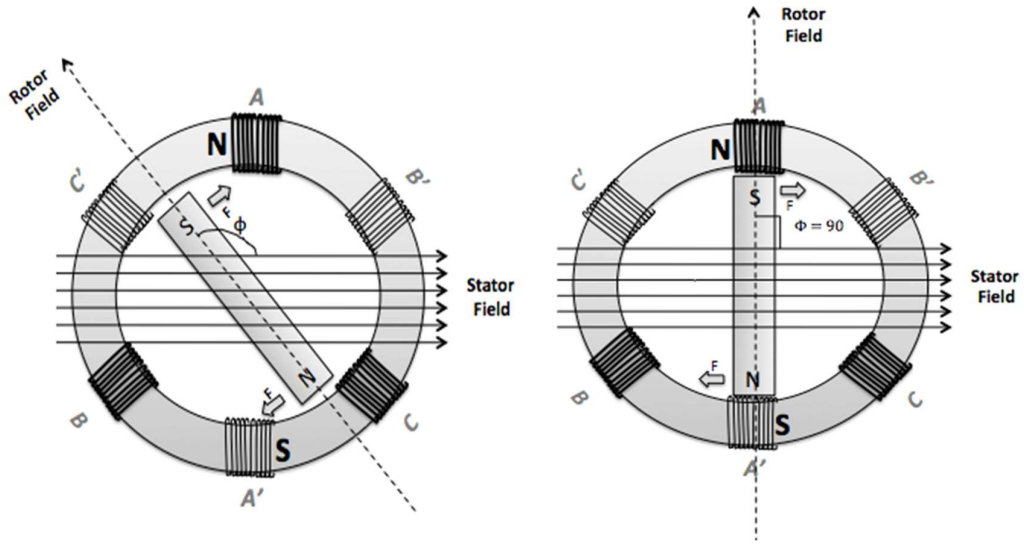


Fig. 2.8

FOC operates on the resultant vector of the three-phase currents rather than controlling each phase independently. The control variables of an AC induction motor are made stationary (DC) using mathematical transformations. In a way, FOC tries to control an induction motor by imitating DC motor operation as it deals with stationary parameters.

There are two methods of FOC, Direct FOC, and Indirect FOC. In Direct FOC, the rotor flux angle is directly computed from flux estimations or measurements. In Indirect FOC, the rotor flux angle is indirectly computed from available speed and slip computations.

2.6.1 Steps Involved in FOC:

Field oriented control for BLDC motors has six basic steps as described in:

1. Phase currents are measured.
2. Measured phase currents are converted into a two-axis system by using Clarke's transformation.
3. The two-axis coordinate system which was generated in the previous step is transformed to align with the rotor magnetic flux by using Park's transformation.
4. Calculated d and q axis currents are used as feedback for PI controllers, where the direct axis current reference value is zero for permanent magnet motor and direct axis current reference is

generated from the torque command generated by the speed control loop.

5. The direct and quadrature axis voltages from PI controllers are translated back to the stationary two-axis reference frame using inverse Park's transformation.
6. The alpha-beta reference frame values are converted back to the 3-phase quantities and are inputs to the pulse width modulation.

2.6.2 Implementation of FOC:

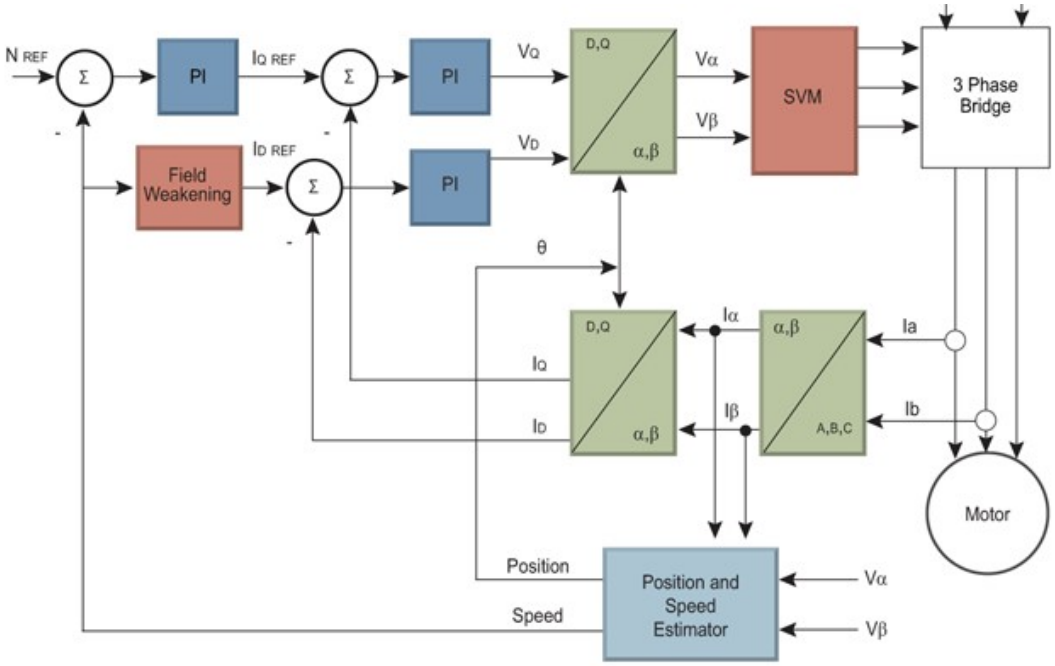


Fig. 2.9 FOC Block Diagram

- *Current Reconstruction Module:* A two-shunt method is used to reconstruct the currents. With this method, the current from two legs is measured and the third current is reconstructed using Kirchhoff's current law. The PWMs are designed to be center-aligned, and the two current samples are captured at the center of the PWM period, every FOC cycle. Once the samples are taken, the FOC begins the ADC conversion of these samples, and the currents are reconstructed.

- *Forward Clarke and Park Transformations:*

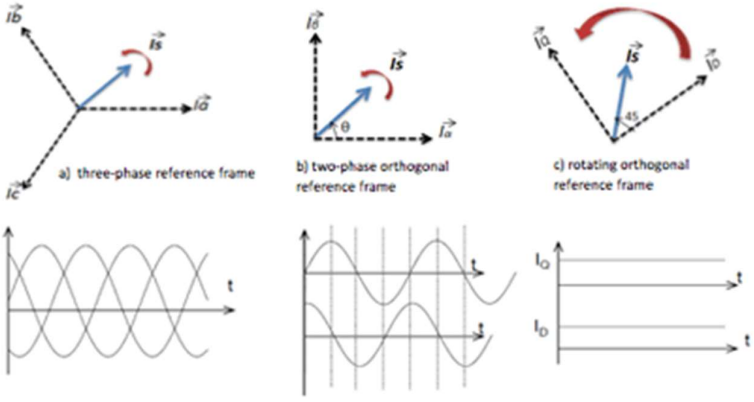


Fig. 2.10

The reconstructed currents are then transformed to 2-phase stator reference and then to 2-phase rotor references using Clarke and Park transforms respectively. Once the transformations are completed, the current in the rotor reference is ready to be regulated for the desired speed and torque. Clarke transform translates the three-phase system into two orthogonal components. The transform is given with the given equation

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \frac{2}{3}\begin{bmatrix} 1 & \cos\frac{2\pi}{3} & \cos\frac{4\pi}{3} \\ 0 & \sin\frac{2\pi}{3} & \sin\frac{4\pi}{3} \end{bmatrix}\begin{bmatrix} i_A \\ i_B \\ i_C \end{bmatrix} \tag{2.12}$$

Park transformation translates the two-axis orthogonal system from Clarke's transformation into a rotating orthogonal system. Park transformation is given by the equation

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos\vartheta & \sin\vartheta \\ -\sin\vartheta & \cos\vartheta \end{bmatrix}\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} \tag{2.13}$$

- *Inverse Clark and Park Transformation:* The inverse Park transformation, as the name suggests, is a translation from the rotating orthogonal system back to the stationary orthogonal system. It is given by

$$\begin{bmatrix} V_\alpha \\ V_\beta \end{bmatrix} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta \\ \sin\vartheta & \cos\vartheta \end{bmatrix}\begin{bmatrix} \mathrm{V_d} \\ \mathrm{V_q} \end{bmatrix} \tag{2.14}$$

Inverse Clarke transformation transforms the voltage vectors in the stationary orthogonal two-axis system back to three-phase quantities.

$$\begin{bmatrix} V_A \\ V_B \\ V_C \end{bmatrix} = \begin{bmatrix} V_\alpha \\ \frac{-1}{2}V_\alpha + \frac{\sqrt{3}}{2}V_\beta \\ \frac{-1}{2}V_\alpha + \frac{\sqrt{3}}{2}V_\beta \end{bmatrix} \quad (2.15)$$

- *SVM (Space Vector Modulation):* The space vector modulation technique is used to generate a sine wave to be fed to the stator coils. Based on the 3-phase reference generated by the inverse Clarke transform, the SVM generates the PWM compare values, which are phase-shifted 120 degrees.

$$t_c = \frac{PWM_{Period} - 2 * (T_1 + T_2)}{2} \quad (2.16)$$

$$t_b = t_c + 2 * T_1 \quad (2.17)$$

$$t_a = t_b + 2 * T_1 \quad (2.18)$$

Sector	T_1	T_2
1	$-V_b$	$-V_c$
2	$-V_c$	$-V_a$
3	V_b	V_a
4	$-V_a$	$-V_b$
5	V_a	V_c
6	V_c	V_b

Table 2.1

- *Speed and Position sensing:* The speed is measured by measuring the period between two rising or falling edges of one of the comparator inputs. The period measured is nothing but one electrical period, which is the inverse of the electrical frequency or speed.

Chapter 3:

HARDWARE AND SCHEMATICS

SNo	Name of the Component	Ratings
1	STM32F407VGT6 DISCOVERY BOARD	
2	IRS2304 HALF-BRIDGE DRIVER	10 – 20 V
3	IRLR7843 N CHANNEL MOSFET	30 V, 3.3 mΩ
4	INA199 CURRENT SHUNT AMPLIFIER	50 V/ V
5	LMV331 COMPARATOR	2.7 – 5.5 V
6	BLDC MOTOR	12 V, 1400 K_v
7	RESISTORS	68kΩ, 12kΩ, 5.1kΩ, 1kΩ, 51Ω, 0.001Ω
8	CAPACITORS	680μF, 2.2μF, 1μF, 100nF, 15nF
9	1N4148 SWITCHING DIODE	

Table 3.1

3.1 STM32F407 Microcontroller:

- Core: ARM® 32-bit Cortex®-M4 CPU with FPU, frequency up to 168 MHz, and DSP instructions.
- Up to 1 Mbyte of Flash memory.
- Up to 192+4 Kbytes of SRAM including 64- Kbyte of CCM (core coupled memory) data RAM.

- 3×12-bit, 2.4 MSPS A/D converters: up to 24 channels and 7.2 MSPS in triple interleaved mode.
- 2×12-bit D/A converters • General-purpose DMA: 16-stream DMA controller with FIFOs and burst support.
- Up to 17 timers: up to twelve 16-bit and two 32- bit timers up to 168 MHz, each with up to 4 IC/OC/PWM or pulse counter and quadrature (incremental) encoder input.
- Up to 140 I/O ports with interrupt capability.

3.2 IRS2304 Half-Bridge Driver:

The IRS2304 is a high voltage, high-speed power MOSFET and IGBT driver with independent high-side and low-side referenced output channels. The logic input is compatible with standard CMOS or LSTTL output, down to 3.3 V logic. The output driver features a high pulse current buffer stage designed for minimum driver cross-conduction. The floating channel can be used to drive an N-channel power MOSFET or IGBT in the high-side configuration.

- V_{OFFSET} - 600 V max
- IO+/- (min) - 60 mA/130 mA
- V_{OUT} - 10 V - 20 V
- Delay Matching - 50 ns
- Internal deadtime - 100 ns
- ton/off (typ.) - 150 ns/150 ns

3.3 IRLR7843 N Channel MOSFET:

- V_{DSS} = 30 V
- R_{DS}(on) = 3.3 m
- max Q = 34 nC
- V_{GS} = ±20 V

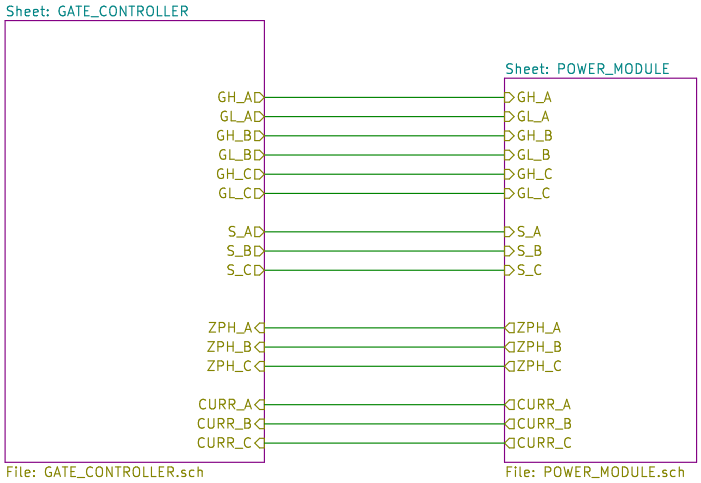
3.4 INA199 Current Shunt Amplifier:

The INA199 series of voltage-output, current-shunt monitors (also called current-sense amplifiers) are commonly used for overcurrent protection, a precision current measurement for system optimization, or in closed-loop feedback circuits. This series of devices can sense drops across shunt resistors at common-mode voltages from –0.3 V to 26 V, independent of the supply voltage. Three fixed gains are available: 50 V/V, 100 V/V, and 200 V/V. The low offset of the zero-drift architecture enables current sensing with maximum drops across the shunt as low as 10-mV full-scale. These devices operate from a single 2.7-V to 26-V power supply, drawing a maximum of 100 μA of supply current.

3.5 LMV331 Comparator:

The LMV331 series are low-voltage, (2.7V to 5.5V) single comparators, which are designed to effectively reduce cost and space at low-voltage levels.

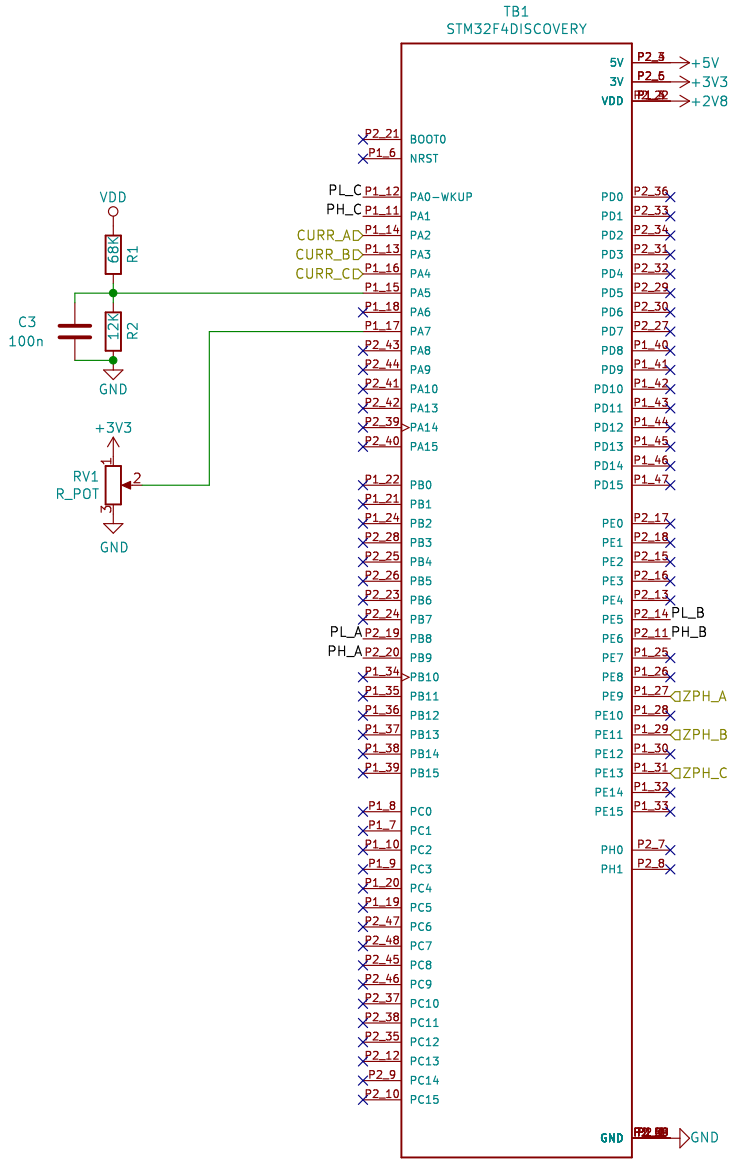
3.6 Schematic Diagram:



Sheet: /
File: ESC SENSORLESS.sch

Title: ESC SENSORLESS

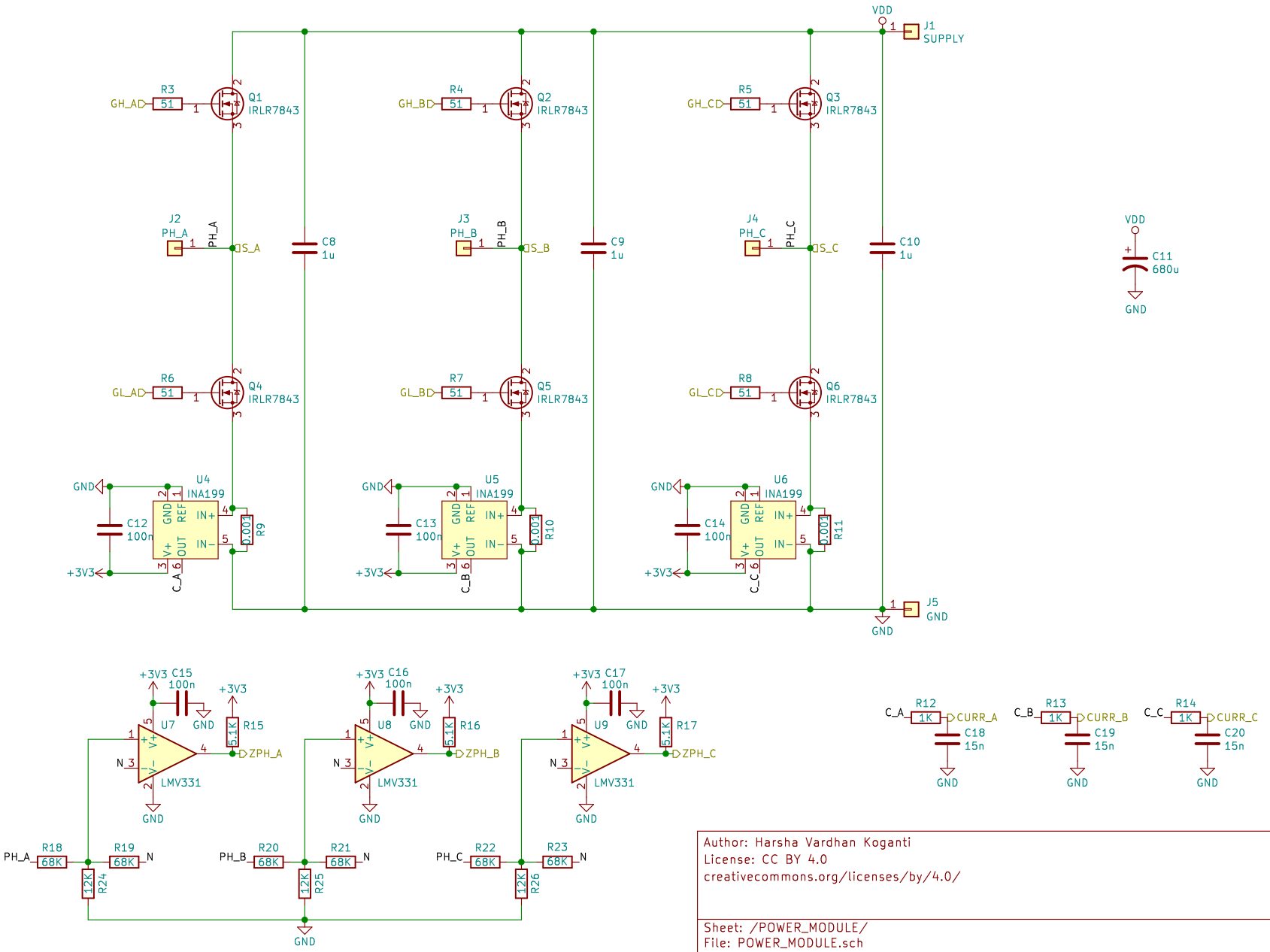
Size: A4	Date: 2019-12-07	**Rev: v03**
KiCad E.D.A. kicad (5.1.0)-1		Id: 1/3



Sheet: /GATE_CONTROLLER/
File: GATE_CONTROLLER.sch

Title: ESC SENSORLESS

Size: A4	Date: 2019-12-07	**Rev: v03**
KiCad E.D.A. kicad (5.1.0)-1		Id: 2/3



Sheet: /POWER_MODULE/
File: POWER_MODULE.sch

Title: ESC SENSORLESS

Size: A4	Date: 2019-12-07	**Rev: v03**
KiCad E.D.A. kicad (5.1.0)-1		Id: 3/3

Chapter 4:

BLDC MOTOR CONTROL ALGORITHM

When the motor is running the motor control code has two major functions: status monitoring and motor control interrupts. The overall view of the motor control code is shown in the following Figure. The main System Service loop, in addition to status monitoring, controls the start-up sequence. Two interrupt types are enabled to control motor operation: Timer1 and comparator. Timer1 interrupts are always enabled and invoke each motor commutation. Comparator interrupts are only enabled every second commutation period to capture the zero-crossing event.

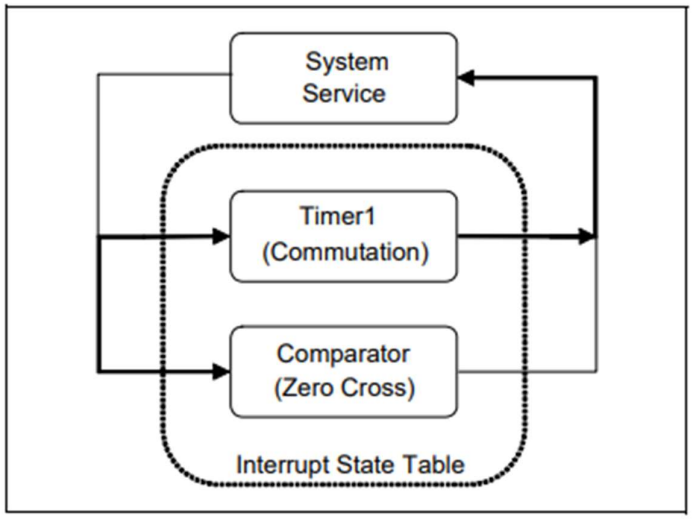


Fig. 4.1 Overall view of Motor Control Code

```
/* USER CODE BEGIN Private defines */
/*
#define INIT_COMPLETE_FLAG                0x01U
#define WARMUP_COMPLETE_FLAG              0x02U
#define SLOWSTART_COMPLETE_FLAG           0x04U
#define STARTUP_IN_PROGRESS               0x08U
#define STARTUP_COMPLETE_FLAG             0x10U
#define RUN_FLAG                          0x20U
#define STOP_FLAG                         0x40U
#define RISING_BEMF_FLAG                  0x80U
#define TIMER_WARMUP_FLAG                 0x01U
#define TIMER_SLOWSTART_FLAG              0x02U
#define TIMER_STARTUP_FLAG                0x04U
#define TIMER_STALL_FLAG                  0x08U
#define TIMER_DUTY_FLAG                   0x10U
#define TIMER_SERIAL_FLAG                 0x20U
#define WARMUP_TIME                       40U
#define SLOWSTART_TIME                    20U
#define STARTUP_TIME                      200U
#define DUTY_TIME                         1U
#define STALL_TIME                        100U
#define STALL_CHECK_TIME                  50U
#define SERIAL_TIME                       100U
#define SLOW_STEPS                        1U
#define PWM_PERIOD                        1023U
#define START_PWM_DUTY                    PWM_PERIOD * 13U / 100U
#define TIMER_AUTO_RELOAD                 65535U
#define START_COMMUTATION_TIME            1200U
#define TIMER_START_COUNT                 TIMER_AUTO_RELOAD –
                                          START_COMMUTATION_TIME + 1U
#define BEMF_DETECTION                    101U
#define COMMUTATE                         102U
#define MIN_COMMUTATION_TIME              90U
#define MIN_COMMUTATION_COUNT             65535U - MIN_COMMUTATION_TIME
#define BLANK_PERIOD                      10U
#define ERROR_FACTOR                      4U
#define ADC_MAX_RESOLUTION                255U
#define OFF_ADC_VAL                       ADC_MAX_RESOLUTION * 15U / 100U
#define ON_ADC_VAL                        ADC_MAX_RESOLUTION * 25U / 100U
#define US_PER_SEC                        1000000U
#define SEC_PER_MIN                       60U
#define NUM_POLES                         14U
#define NUM_PHASES                        3U
/* USER CODE END Private defines */
```

- Declaring all variables used.

```
/* USER CODE BEGIN PV */
uint8_t StateFlags;
uint8_t TimerFlags;
uint8_t SlowStartEvents;
uint8_t TMR_WarmUp_Timer;
uint8_t TMR_SlowStart_Timer;
uint8_t TMR_StartUp_Timer;
uint8_t TMR_Duty_Timer;
uint8_t TMR_Stall_Timer;
uint8_t TMR_StallCheck_Timer;
uint8_t TMR_Serial_Timer;
uint16_t TMR_Comm_Time;
uint16_t Comm_After_ZC;
uint16_t Expected_ZC;
uint16_t ZC;
int ZCError;
uint8_t PwmDuty;
uint8_t CommState;
uint8_t IsrState;
uint16_t Bit16;
uint32_t Bit32;
uint16_t Temp_Comm_Time;
int Temp;
char buffer[50];
const int CCR_Values[256]={
  0,  4,  8,  12, 16, 20, 24, 27, 31, 35, 39, 43, 47, 51, 55, 59,
  63, 67, 71, 75, 78, 82, 86, 90, 94, 98, 102,106,110,114,118,122,
  125,129,133,137,141,145,149,153,157,161,165,169,173,176,180,184,
  188,192,196,200,204,208,212,216,220,224,227,231,235,239,243,247,
  251,255,259,263,267,271,275,278,282,286,290,294,298,302,306,310,
  314,318,322,325,329,333,337,341,345,349,353,357,361,365,369,373,
  376,380,384,388,392,396,400,404,408,412,416,420,424,427,431,435,
  439,443,447,451,455,459,463,467,471,475,478,482,486,490,494,498,
  502,506,510,514,518,522,525,529,533,537,541,545,549,553,557,561,
  565,569,573,576,580,584,588,592,596,600,604,608,612,616,620,624,
  627,631,635,639,643,647,651,655,659,663,667,671,675,678,682,686,
  690,694,698,702,706,710,714,718,722,725,729,733,737,741,745,749,
  753,757,761,765,769,773,776,780,784,788,792,796,800,804,808,812,
  816,820,824,827,831,835,839,843,847,851,855,859,863,867,871,875,
  878,882,886,890,894,898,902,906,910,914,918,922,925,929,933,937,
  941,945,949,953,957,961,965,969,973,976,980,984,988,992,996,1000
};
/* USER CODE END PV */
```

- Infinite loop inside the main function

```
/* USER CODE BEGIN 2 */
 StateFlags |= STOP_FLAG;
 /* USER CODE END 2 */

 /* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
 {
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
        if(StateFlags & STOP_FLAG) InitSystem();
        TimeBaseManager();
        WarmUpControl();
        ControlSlowStart();
        ControlStartUp();
        StallControl();
        SpeedManager();
        SerialMonitoring();
 }
 /* USER CODE END 3 */
```

4.1 System Services:

System services control the start-up sequence and monitor system status while the motor is running. start-up consists of system initialization, warm-up, slow start, and Phase Lock search. Status monitoring includes stall monitoring and speed control. The frequency of each start-up event and status check is determined by the time base manager.

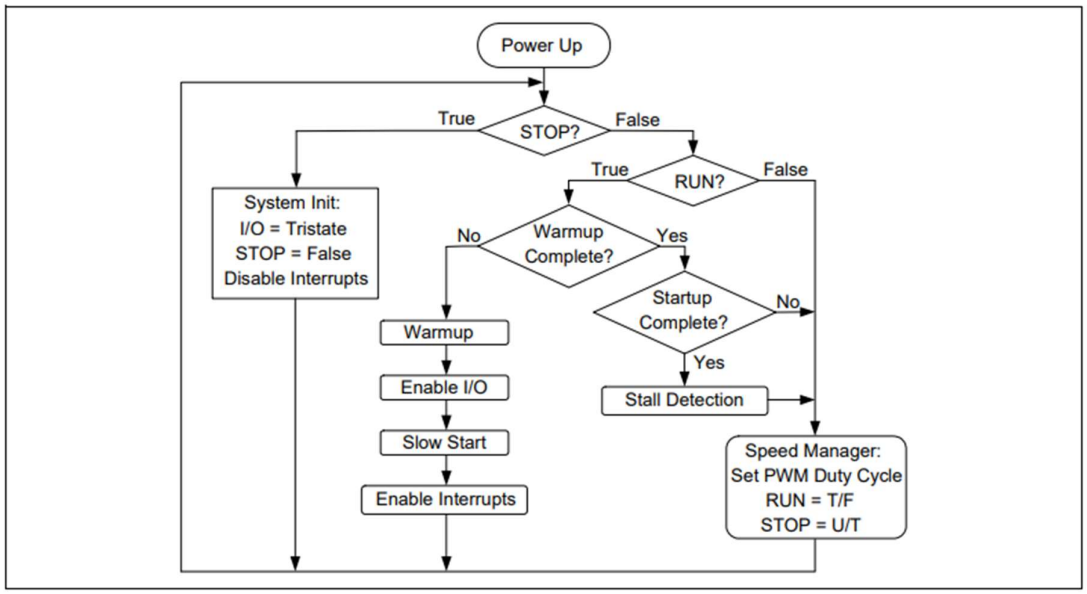


Fig. 4.2 Flow Chart of System Services

4.1.1 Time Base Manager:

The time base manager is a 10-millisecond delay. Each status monitor subroutine keeps track of the time base manager time-ticks through a flag. Every 10 milliseconds, the time base manager sets a series of flags, one for each start-up control and status monitor. The flag is an indication of the respective control or status monitor to update its internal counter. The control or status function is serviced when the corresponding counter reaches zero. The counter is then reset to the time count for that monitor or control and the service is performed.

```
void TimeBaseManager(void)
{
        HAL_Delay(9);
        TimerFlags |= (TIMER_WARMUP_FLAG | TIMER_SLOWSTART_FLAG);
        TimerFlags |= (TIMER_STARTUP_FLAG | TIMER_DUTY_FLAG);
        TimerFlags |= TIMER_SERIAL_FLAG;
}
```

4.1.2 System Initialization:

System Initialization sets all ports to their initial off state. Initialization occurs at power-up and whenever the motor is stopped. The motor may be stopped as a result of a Fault or because the speed request is below the lowest run speed. When initialization is active, all of the status functions, other than speed request, are disabled.

```
void InitSystem(void)
{
	HAL_TIM_Base_Stop_IT(&htim2);
	HAL_TIM_IC_Stop_IT(&htim2, TIM_CHANNEL_1);
	HAL_TIM_IC_Stop_IT(&htim2, TIM_CHANNEL_2);
	HAL_TIM_IC_Stop_IT(&htim2, TIM_CHANNEL_3);

	HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_1);
	HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_2);
	HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_3);
	HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
	HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
	HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);

	TMR_WarmUp_Timer = WARMUP_TIME;
	TMR_SlowStart_Timer = SLOWSTART_TIME;
	TMR_Duty_Timer = DUTY_TIME;
	TMR_Stall_Timer = STALL_TIME;
	TMR_StallCheck_Timer = STALL_CHECK_TIME;
	TMR_Serial_Timer = SERIAL_TIME;

	StateFlags = 0x00U;
	TimerFlags = 0x00U;
	SlowStartEvents = SLOW_STEPS;

	Comm_After_ZC = 0;
	Expected_ZC = 0;
	ZC = 0;
	ZCError = 0;
}
```

4.1.3 Warm-Up

Warm-up allows the system to settle after initialization. Warm-up commences immediately after initialization and lasts for the warm-up time specified by the TMR_WarmUp_Timer. When warm-up is complete, the motor drivers are enabled and slow start commences.

```
void WarmUpControl(void)
{
	if (StateFlags & WARMUP_COMPLETE_FLAG) return;

	if ((~(TimerFlags)) & TIMER_WARMUP_FLAG) return;
	TimerFlags &= ~((uint8_t)TIMER_WARMUP_FLAG);

	if (TMR_WarmUp_Timer)
	{
		TMR_WarmUp_Timer--;
	}
	else
	{
		if (StateFlags & RUN_FLAG)
		{
			StateFlags |= WARMUP_COMPLETE_FLAG;
			InitDriver();
		}
	}
}

void InitDriver(void)
{
	PwmDuty = FindTableIndex(START_PWM_DUTY);
	GetPulseVal(PwmDuty);

	TMR_Comm_Time = TIMER_START_COUNT;
	Expected_ZC = (TMR_Comm_Time>>1) | 0x8000;

	CommState = 1;
	Commutate();

	sprintf(buffer, "BLDC started\n");
	HAL_UART_Transmit(&huart2, (uint8_t *)buffer, strlen(buffer), 10);

	StateFlags |= (INIT_COMPLETE_FLAG | STARTUP_IN_PROGRESS);
}

uint8_t FindTableIndex(uint16_t Find_duty)
{
	uint8_t Index;
	for (Index = 0; CCR_Values[Index] < Find_duty; Index++);
	return Index;
}
```

4.1.4 Control Slow Start:

Control slow start prepositions the motor to a known commutation state so that subsequent commutations can start to accelerate the motor up to speed. Recall that the drive voltage is applied to two of the three motor windings. This causes the permanent magnet rotor to align with the created magnetic field which is between the active windings. Some motors have a strong tendency to align the rotor magnets with one of the motor windings when power is not applied. There are 60 electrical degrees between windings so that applying power to such a motor will cause the rotor to move at least 30 electrical degrees, or more if the rotor was aligned to the winding not being powered. Other motors have a weak alignment tendency, so that the rotor may be aligned 180 degrees out of phase with the generated magnetic field when power is applied. In this case, the rotor will not move at all and will be in an improper position to accelerate properly when commutation begins. For this reason, motors that have a weak alignment tendency must receive two slow start positioning drive periods of sequential commutation drive states to ensure that the rotor is not stuck 180 degrees out of phase when the actual commutation starts. The amount of time to dwell in each slow start state depends on the inertia of the motor and the drive voltage applied. High inertia motors need more dwell time than low inertia motors. This allows the rotor to settle to a known position which makes the acceleration response to the first commutation drive more consistent and predictable. The dwell time for each slow start step is specified by the SlowStartEvents. The start-up commutation period is set in Timer1 and interrupts are enabled after the slow start. At this point, Timer1 interrupts have complete control of the motor commutation and comparator interrupts determine how to adjust Timer1 to achieve Phase Lock with the BEMF signal. CONTROL START-UP commences immediately after a slow start and looks only slightly different than the normal run condition. The only difference between normal run mode and start-up is the startup_complete flag. The startup_complete flag is cleared during warm-up and remains clear until the zero-crossing event is detected within +/- 12% of the commutation mid-point. The startup_complete flag is necessary to prevent start-up from being detected

as a stall condition. The motor accelerates during start-up and the commutation period shortens to catch up because zero-crossing is detected almost immediately after commutation. See the Section "Zero-Crossing During Start-up" for more detail on how this works. Control start-up routine does nothing more than check that Phase Lock to the BEMF signal is achieved in a reasonable amount of time. It does this by setting a timer and checking that the startup_complete flag is set when the start-up time is complete. If zero-crossing fails to set the startup_complete flag within the allowed start-up time, then the motor is stopped and the start-up steps repeat from system initialization.

```
void ControlSlowStart(void)
{
	if ((~(StateFlags)) & INIT_COMPLETE_FLAG) return;

	if (StateFlags & SLOWSTART_COMPLETE_FLAG) return;

	if ((~(TimerFlags)) & TIMER_SLOWSTART_FLAG) return;
	TimerFlags &= ~((uint8_t)TIMER_SLOWSTART_FLAG);

	if (--TMR_SlowStart_Timer == 0)
	{
		if (--SlowStartEvents == 0)
		{
			StateFlags |= SLOWSTART_COMPLETE_FLAG;
			TMR_StartUp_Timer = STARTUP_TIME;
			HAL_TIM_Base_Stop(&htim2);
			__HAL_TIM_SET_COUNTER(&htim2, TMR_Comm_Time);
			HAL_TIM_Base_Start(&htim2);
			IsrState = COMMUTATE;
			HAL_TIM_Base_Start_IT(&htim2);
		}
		else
		{
			TMR_SlowStart_Timer = SLOWSTART_TIME;
			Commutate();
		}
	}
}
```

4.1.5 Control Start-Up:

Start-up commences immediately after the slow start and looks only slightly different than the normal run condition. The only difference between normal run mode and start-up is the startup_complete flag. The

startup_complete flag is cleared during warm-up and remains clear until the zero-crossing event is detected within +/- 12% of the commutation mid-point. The startup_complete flag is necessary to prevent start-up from being detected as a stall condition. The motor accelerates during start-up and the commutation period shortens to catch up because zero-crossing is detected almost immediately after commutation. See the Section "Zero-Crossing During Start-up" for more detail on how this works. Control start-up routine does nothing more than check that Phase Lock to the BEMF signal is achieved in a reasonable amount of time. It does this by setting a timer and checking that the startup_complete flag is set when the start-up time is complete. If zero-crossing fails to set the startup_complete flag within the allowed start-up time, then the motor is stopped and the start-up steps repeat from system initialization.

```
void ControlStartUp(void)
{
        if ((~(StateFlags)) & SLOWSTART_COMPLETE_FLAG) return;

        if ((~(StateFlags)) & STARTUP_IN_PROGRESS) return;

        if ((~(TimerFlags)) & TIMER_STARTUP_FLAG) return;
        TimerFlags &= ~((uint8_t)TIMER_STARTUP_FLAG);

        if (--TMR_StartUp_Timer == 0)
        {
                StateFlags &= ~((uint8_t)STARTUP_IN_PROGRESS);

                if ((~(StateFlags)) & STARTUP_COMPLETE_FLAG)
                {
                        StateFlags |= STOP_FLAG;
                }
        }
}
```

4.1.6 Stall Monitoring:

Stall monitoring checks to make sure that the motor responded properly to the start-up conditions and is rotating. If a stall is detected then the motor is stopped and a restart is attempted. There are various reasons why the motor may not be rotating. For example, the rotor could be held in position by some blockage, or the rotor did not settle fully during

a slow start-up. In both cases, the commutation will go through the acceleration process without the motor following. As we will see later, a stalled motor will produce a zero-crossing event almost immediately after commutation. We allow for early zero-crossing for a short while during start-up, because a starting motor is by definition stalled. If the motor does not sense the zero-crossing event in the middle of the commutation period in a reasonable amount of time, then the motor is assumed to be stalled. What sometimes happens though, is that the control algorithm continues to shorten the commutation period as a result of acceleration until the commutation period just happens to be twice the time to the zero-crossing. The time from commutation to zero-crossing did not change, only the commutation period did. The commutation period is compared to the speed control speed request and, if the commutation period is much shorter than expected, then a stall condition is assumed.

```
void StallControl(void)
{
	if ((~(StateFlags)) & INIT_COMPLETE_FLAG) return;

	if ((~(StateFlags)) & STARTUP_COMPLETE_FLAG) return;

	if ((~(TimerFlags)) & TIMER_STALL_FLAG) return;
	TimerFlags &= ~((uint8_t)TIMER_STALL_FLAG);

	if(--TMR_Stall_Timer == 0)
	{
		StateFlags |= STOP_FLAG;
	}

	if (--TMR_StallCheck_Timer == 0)
	{
		TMR_StallCheck_Timer = STALL_CHECK_TIME;

		if(TMR_Comm_Time > MIN_COMMUTATION_COUNT)
		{
			StateFlags |= STOP_FLAG;
		}
	}
}
```

4.1.7 Speed Manager:

The speed manager varies the voltage applied to the motor. This is accomplished by pulse-width modulating the motor driver switches. Even in systems where the motor always runs at full speed, speed control is needed to soft start the motor. Without a soft start, the start-up currents will be excessive and starting torque could cause system damage. The speed control manager always starts the motor at the same voltage. The speed control manager starts increasing or decreasing the applied voltage up to the desired level when the zero-crossing detection senses that commutation is synchronized with the motor. Voltage is varied by varying the on-period.

```
void SpeedManager(void)
{
	uint16_t AdcVal;

	if((~(TimerFlags)) & TIMER_DUTY_FLAG) return;
	TimerFlags &= ~((uint8_t)TIMER_DUTY_FLAG);

	if(--TMR_Duty_Timer) return;

	HAL_ADC_Start(&hadc1);
	if(HAL_ADC_PollForConversion(&hadc1, 5) == HAL_OK)
	{
		AdcVal = HAL_ADC_GetValue(&hadc1);
	}
	HAL_ADC_Stop(&hadc1);

	if (AdcVal < OFF_ADC_VAL)
	{
		if (StateFlags & RUN_FLAG) StateFlags |= STOP_FLAG;
		StateFlags &= ~((uint8_t)RUN_FLAG);
		return;
	}

	if (AdcVal > ON_ADC_VAL) StateFlags |= RUN_FLAG;

	if ((~(StateFlags)) & STARTUP_COMPLETE_FLAG) return;

	if (AdcVal > PwmDuty) PwmDuty++;

	if (AdcVal < PwmDuty) PwmDuty--;

	GetPulseVal(PwmDuty);
	return;
}

void GetPulseVal(uint8_t PulseVal)
{
	htim1.Instance->CCR1 = CCR_Values[PulseVal];
	htim1.Instance->CCR2 = CCR_Values[PulseVal];
	htim1.Instance->CCR3 = CCR_Values[PulseVal];
}
```

4.1.8 Commutation States:

```
void Commutate(void)//Low Side Modulation
{
	switch (CommState)
	{
	case 1:
		HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);
		HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
		HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
		StateFlags &= ~((uint8_t)RISING_BEMF_FLAG);
		Bit32 = GPIOB_BASE;
		Bit16 = GPIO_PIN_10;
		break;
	case 2:
		HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_2);
		HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);
		StateFlags |= RISING_BEMF_FLAG;
		Bit32 = GPIOA_BASE;
		Bit16 = GPIO_PIN_1;
		break;
	case 3:
		HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
		HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
		StateFlags &= ~((uint8_t)RISING_BEMF_FLAG);
		Bit32 = GPIOA_BASE;
		Bit16 = GPIO_PIN_0;
		break;
	case 4:
		HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_3);
		HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
		StateFlags |= RISING_BEMF_FLAG;
		Bit32 = GPIOB_BASE;
		Bit16 = GPIO_PIN_10;
		break;
	case 5:
		HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
		HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_SET);
		StateFlags &= ~((uint8_t)RISING_BEMF_FLAG);
		Bit32 = GPIOA_BASE;
		Bit16 = GPIO_PIN_1;
		break;
	case 6:
		HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_1);
		HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
		StateFlags |= RISING_BEMF_FLAG;
```

```
                Bit32 = GPIOA_BASE;
                Bit16 = GPIO_PIN_0;
                CommState = 0;
                break;
        default:
                HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_1);
                HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_2);
                HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_3);
                HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);
                CommState = 1;
                break;
        }
        CommState++;
}
```

4.1.8 Serial Monitoring:

```
void SerialMonitoring(void)
{
        if ((~(StateFlags)) & INIT_COMPLETE_FLAG) return;

        if ((~(StateFlags)) & STARTUP_COMPLETE_FLAG) return;

        if ((~(TimerFlags)) & TIMER_SERIAL_FLAG) return;
        TimerFlags &= ~((uint8_t)TIMER_SERIAL_FLAG);

        if(--TMR_Serial_Timer == 0)
        {
                TMR_Serial_Timer = SERIAL_TIME;

                Temp = CCR_Values[PwmDuty] * 100 / PWM_PERIOD;
                sprintf(buffer, "PWM Duty Cycle = %d %%\n", Temp);
                HAL_UART_Transmit(&huart2, (uint8_t *)buffer, strlen(buffer), 10);

                Temp = ((uint16_t)-TMR_Comm_Time) * NUM_PHASES * NUM_POLES;
                Temp = US_PER_SEC * SEC_PER_MIN / Temp;
                sprintf(buffer, "Speed = %d RPM\n", Temp);
                HAL_UART_Transmit(&huart2, (uint8_t *)buffer, strlen(buffer), 10);
        }
}
```

4.2 Interrupts:

Two types of interrupts are used to control the motor: Timer1 interrupts set the commutation period and comparator interrupts capture the time of the zero-crossing event.

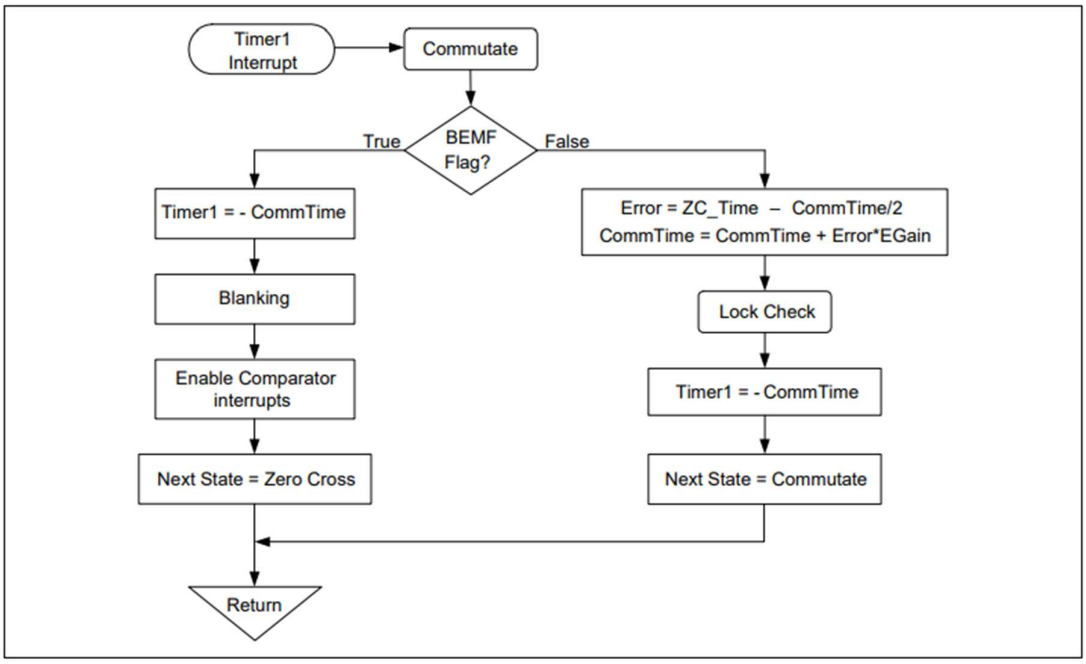


Fig. 4.3 Flow Chart of Interrupts

4.2.1 Commutation Interrupt:

At the beginning of each commutation interrupt the commutation subroutine is called in which the motor drivers and BEMF sense lines are switched for the upcoming commutation period. Commutation time is controlled by Timer2. Timer2 is preset so that overflow will occur when the commutation time has elapsed. The interrupt occurs when Timer2 overflows one commutation period later and the process is repeated for the next commutation phase. Timer2, in effect, always contains a negative number representing the time remaining until the next commutation event.

At each commutation event, the commutation subroutine is called to switch the motor drivers. The commutation subroutine also sets a flag to let the Interrupt Service Routine (ISR) know whether to set up for a zero-crossing event or make error correction calculations. The commutation interrupt has two purposes other than switching the drivers: Zero-crossing

setup and commutation time calculation. In the first instance, if the zero-crossing event will be captured in the upcoming period, then the comparator interrupt must set up for that occurrence. In the other instance, the zero-crossing event cannot be captured so there is more time available to make calculations to correct the commutation period. In both cases, Timer2 is preset with the negative of the last calculated commutation time.

4.2.2 Commutation Period Math:

Since Timer2 is a 16-bit timer, the Timer2 count registers can contain a number from 0x0000 to 0xFFFF. Timer2 values from 0x8000 to 0xFFFF are treated as negative signed integers. However, Timer2 values from 0x0000 to 0x7FFF are treated as positive signed integers. Normally, this wouldn't matter because Timer2 could be treated as an unsigned integer, but that causes problems when performing math on partial commutation periods such as the mid-commutation point, otherwise known as the expected zero-crossing point. Signed integers are necessary to accommodate positive and negative error calculations. To avoid the need to do all commutation period calculations with long signed integers, one simple trick can keep the math to the size of a signed integer and at the same time realize the full 16-bit count capability of Timer2. Consider that the commutation period as written to Timer2 is a 16-bit negative integer. If we assume that the sign bit is the 17th unimplemented bit, then that bit will always be '1'. The only time the commutation period is measured is at the zero-crossing event. By definition, the zero-crossing time is half the commutation period. The expected zero-crossing value can be calculated from the signed 16-bit commutation period value by shifting the commutation value right by '1' and always setting the Most Significant bit to '1' after the shift. Therefore, values that have no sign bit to extend will assume the negative sign of the unimplemented 17th bit which is always '1'. Negative integer values will have their sign bit extended during the shift, so setting the Most Significant bit makes no difference. Once we have the expected zero-crossing value in a signed 16-bit integer format, all other calculations, such as the zero-crossing error and the next commutation period are calculated with a simple 16-bit signed integer math functions.

However, remember that the commutation time is stored and used as a signed 16-bit number which is always negative, even when the sign bit says otherwise because the implied 17th bit makes it so. Since the value in Timer2 is always negative, the Timer2 value captured at zero-crossing will also be negative. The error between the expected zero-crossing and the actual is computed, scaled, and accumulated in the stored commutation time as follows:

$$ZCE = ZCT - CT/2 \quad (4.1)$$

$$CT = CT + ZCE * EGain \quad (4.2)$$

Where,

ZCE = Zero-Crossing error

CT = Commutation time

CT/ 2 = Expected zero-crossing

ZCT = Zero-Crossing time

EGain = Feedback Gain Factor

When zero-crossing occurs early, then ZCT will be a larger negative number than CT/2, and the error will reduce the commutation time. The opposite will be true when zero-crossing occurs late. The scaling factor determines how quickly the system responds to commutation errors. A large scaling factor will provide a slower response. However, a large scaling factor will also introduce larger truncation errors since we are performing all calculations in 16-bit integer math. Small, low-inertia systems respond best to a small scaling factor whereas high-inertia systems work better with a large scaling factor.

4.2.3 Zero-Crossing Interrupt:

The zero-crossing interrupt occurs when the BEMF voltage reaches the BEMF reference voltage. Two things happen in the zero-crossing interrupt service: First, the value of Timer2 is read, and then Timer2 is preset with half the previously computed commutation time. During steady-state operation, the value read from Timer2 is essentially the same as the value written to Timer2. Regardless of when zero-crossing occurs, early or late, the value written to Timer2 will always be half the previously

computed commutation time. During acceleration, zero-crossing will occur early and the value read from Timer2 will be a larger negative number than expected. The difference between the read number and the expected number is the zero-crossing error which will be used to correct the commutation time when the new commutation time is computed as part of the next commutation interrupt.

4.2.4 Zero-Crossing During Start-Up:

When the motor is not rotating, it is not generating any voltage. The undriven motor terminal then will be approximately half the motor supply voltage because the high-side and low-side drivers form a low-impedance voltage divider. This is the voltage that will be presented to the zero-crossing comparator input. The inter-winding capacitance of the motor coils will cause a small overshoot in the PWM drive signal so the zero-crossing comparator will sense every PWM pulse from the beginning of each commutation period. This means that when the motor is starting the zero-crossing interrupt will occur almost immediately after the blanking period. At the same time, the motor will start to accelerate and the calculated period for the next commutation will be shorter because of the early zero-crossing event. Until the motor comes up to speed, it will be operating in a step response to the applied power. As the motor approaches the ideal commutation rate, and the motor generated voltage grows, the zero-crossing events will move toward the center of the commutation period. When the period is twice the time to zero-crossing, the control will be synchronized with the motor.

```
void TIM2_IRQHandler(void)
{
  /* USER CODE BEGIN TIM2_IRQn 1 */
        switch (IsrState)
        {
        case BEMF_DETECTION:
                __HAL_TIM_DISABLE_IT(&htim2, TIM_IT_CC1);
                __HAL_TIM_DISABLE_IT(&htim2, TIM_IT_CC2);
                __HAL_TIM_DISABLE_IT(&htim2, TIM_IT_CC3);

                if (__HAL_TIM_GET_FLAG(&htim2, TIM_FLAG_CC1) ||
                __HAL_TIM_GET_FLAG(&htim2, TIM_FLAG_CC2) ||
                __HAL_TIM_GET_FLAG(&htim2, TIM_FLAG_CC3))
                {
                        ZC = __HAL_TIM_GET_COUNTER(&htim2);
                        if (StateFlags & STARTUP_COMPLETE_FLAG)
                        {
                                HAL_TIM_Base_Stop(&htim2);
                                __HAL_TIM_SET_COUNTER(&htim2, Comm_After_ZC);
                                HAL_TIM_Base_Start(&htim2);
                        }

                        IsrState = COMMUTATE;
                        break;
                }

                if (StateFlags & STARTUP_COMPLETE_FLAG)
                {
                        TMR_Comm_Time += (Expected_ZC>>2);
                }

        case COMMUTATE:
                if(__HAL_TIM_GET_FLAG(&htim2, TIM_FLAG_UPDATE))
                {
                        htim2.Instance->SR &= ~TIM_IT_UPDATE;// Clear Interrupt
                        Commutate();

                        if ((~(StateFlags)) & RISING_BEMF_FLAG)
                        {
                                Temp_Comm_Time = TMR_Comm_Time + BLANK_PERIOD;

                                while(__HAL_TIM_GET_COUNTER(&htim2) < BLANK_PERIOD);

                                HAL_TIM_Base_Stop(&htim2);
```

```
                        __HAL_TIM_SET_COUNTER(&htim2, Temp_Comm_Time+1);
                        HAL_TIM_Base_Start(&htim2);
                        while (HAL_GPIO_ReadPin((GPIO_TypeDef *)Bit32, Bit16) !=
                        GPIO_PIN_RESET)
                        {
                                if(__HAL_TIM_GET_FLAG(&htim2,
                                TIM_FLAG_UPDATE)) break;
                        }
                        if (!(__HAL_TIM_GET_FLAG(&htim2, TIM_FLAG_UPDATE)))
                        {
                                htim2.Instance->SR &= ~(TIM_IT_CC1 | TIM_IT_CC2 |
                                TIM_IT_CC3);// Clear Interrupt
                                NextZC();
                                //HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 1);
                                IsrState = BEMF_DETECTION;
                        }
                }
                else
                {
                        Expected_ZC = (TMR_Comm_Time>>1) | 0x8000;
                        ZCError = ZC - Expected_ZC;
                        Temp = ZCError;
                        if (Temp & 0x8000) Temp = ~Temp + 1;// Absolute Value
                        if (Temp < ((uint16_t)-Expected_ZC>>1))
                        {
                                StateFlags |= STARTUP_COMPLETE_FLAG;
                                TMR_Stall_Timer = STALL_TIME;
                        }

                        TMR_Comm_Time -= (ZCError>>ERROR_FACTOR);
                        Temp_Comm_Time = TMR_Comm_Time;
                        HAL_TIM_Base_Stop(&htim2);
                        __HAL_TIM_SET_COUNTER(&htim2, Temp_Comm_Time+1);
                        HAL_TIM_Base_Start(&htim2);
                        IsrState = COMMUTATE;
                        Comm_After_ZC = Expected_ZC;
                }
        }
        break;
    default:
        StateFlags |= STOP_FLAG;
        break;
    }
  /* USER CODE END TIM2_IRQn 1 */
}
```

```
void NextZC(void)
{
	switch (CommState)
	{
	case 1:
		HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_1);
		break;
	case 2:
		HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_3);
		break;
	case 3:
		HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_2);
		break;
	case 4:
		HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_1);
		break;
	case 5:
		HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_3);
		break;
	case 6:
		HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_2);
		break;
	default:
		HAL_TIM_IC_Stop_IT(&htim2, TIM_CHANNEL_1);
		HAL_TIM_IC_Stop_IT(&htim2, TIM_CHANNEL_2);
		HAL_TIM_IC_Stop_IT(&htim2, TIM_CHANNEL_3);
		break;
	}
}
```

Chapter 5:

HARDWARE RESULTS

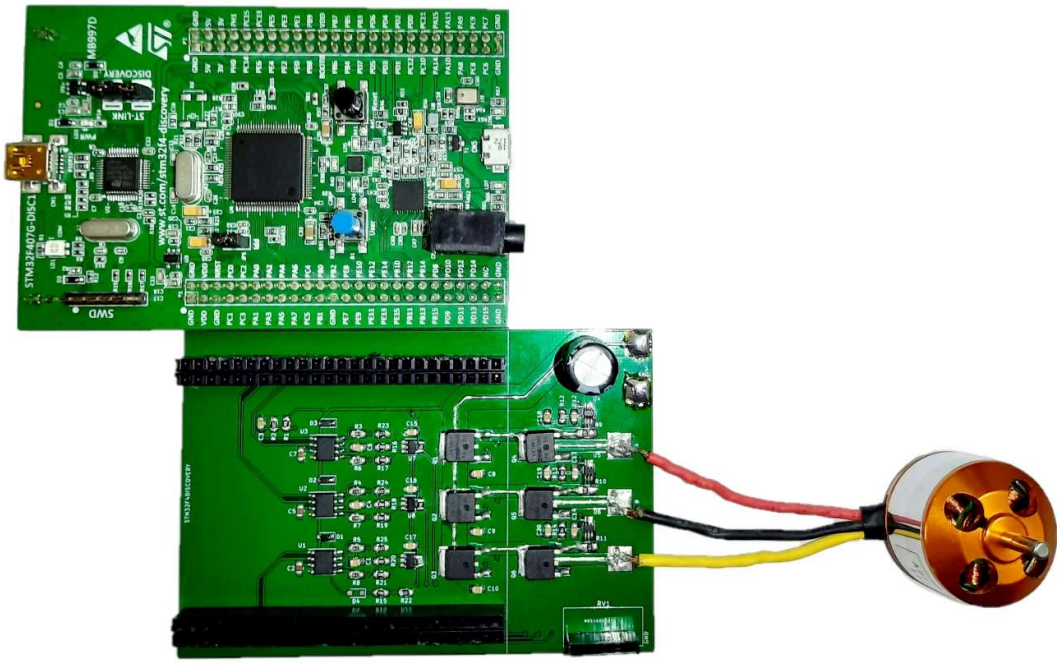


Fig. 5.1 Hardware Assembly

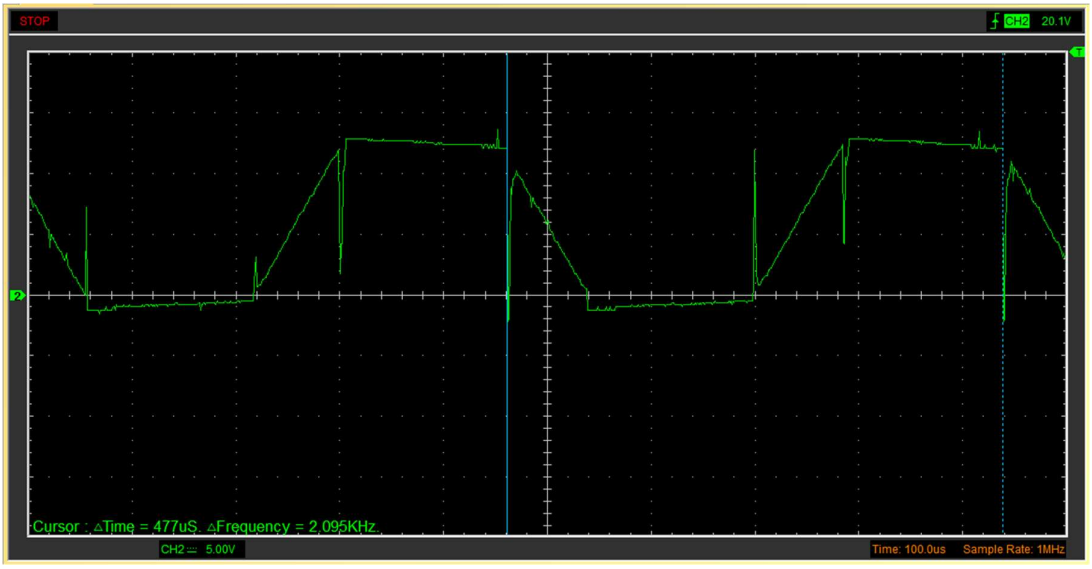


Fig. 5.2 Waveform *[Commercial Controller]*

Waveform obtained at following conditions: V_{IN} = 12 V, Duty Ratio = 100 % Speed Obtained = 17969 RPM

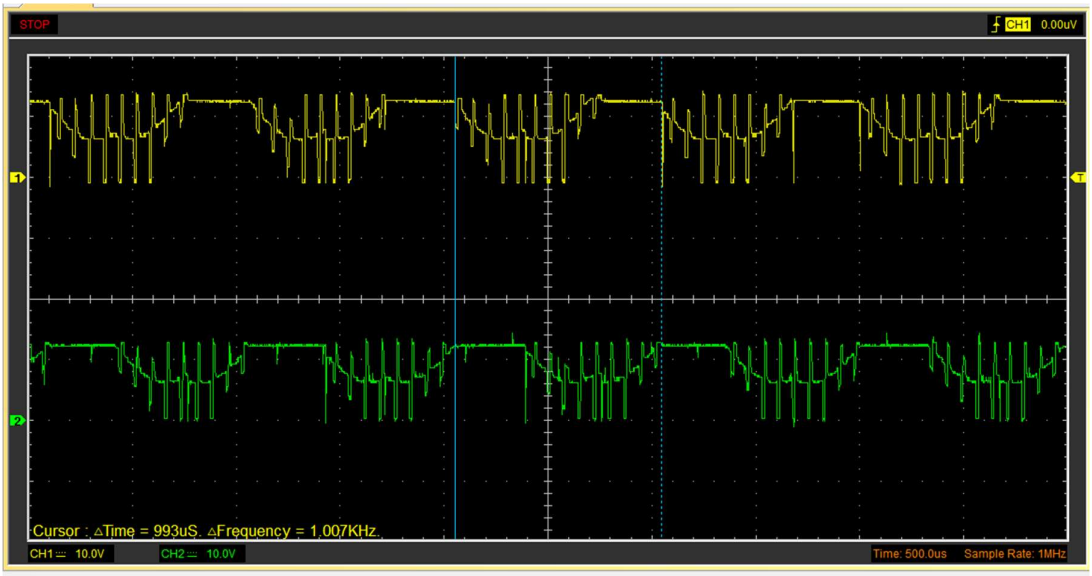


Fig. 5.3 Waveforms *[Our Controller]*

Waveforms obtained at following conditions: V_{IN} = 12 V, Duty Ratio = 21 %, Speed Obtained = 9523 RPM

Chapter 6:

CONCLUSION

This project presents a three-phase sensorless field-oriented control for BLDC motor drives. The proposed closed-loop field-oriented control not only controls the wide speed ranges it also obtains the maximum torque per ampere. This circuit maintains stable operation under very low speed ranges it is the most challenging task in sensorless control. Its robustness to machine parameter variation and achieving better control performance without using any speed feedbacks. Robustness and advantages of the proposed closed-loop sensorless control circuit are verified through hardware results. The results have been presented and analyzed for different speed conditions.

REFERENCES

[1] “Permanent Magnet Synchronous and Brushless DC Motor Drives” – Krishnan. R, 2010.

[2] “Fundamentals of Electrical Drives” – Gopal K. Dubey, 2 Edition.

[3] “Power Electronics” – Dr. P. S. Bimbhra, 6 Edition.

[4] “Control Systems Engineering” - I.J. Nagrath, M. Gopal, 5 Edition.

[5] J. P. John, S. S. Kumar and B. Jaya, "Space Vector Modulation based Field Oriented Control scheme for Brushless DC motors," 2011 International Conference on Emerging Trends in Electrical and Computer Technology, Nagercoil, 2011, pp. 346-351.

[6] M. Lazor and M. Štulrajter, "Modified field oriented control for smooth torque operation of a BLDC motor," 2014 ELEKTRO, Rajecke Teplice, 2014, pp. 180-185.

[7] H. Rasmussen, P. Vadstrup and H. Borsting, "Sensorless field oriented control of a PM motor including zero speed," IEEE International Electric Machines and Drives Conference, 2003. IEMDC'03., Madison, WI, USA, 2003, pp. 1224-1228 vol.2.

[8] Cham, Chin-Long & Samad, Zahurin. (2014). Brushless DC Motor Electromagnetic Torque Estimation with Single-Phase Current Sensing. Journal of Electrical Engineering and Technology. 9. 10.5370/JEET.2014.9.3.866.

[9] Mandal, S. (2014) “Speed control of SVPWM inverter fed BLDC motor drive”.

[10] Ramesh Babu, P., Ramprasath, S., & Paranthagan, B. (2013). Modeling and Dynamic Simulation of Permanent Magnet Brushless DC Motor (PMBLDCM) Drives. Communications in Computer and Information Science, 556–564.

[11] https://vesc-project.com/sites/default/files/Benjamin%20Posts/VESC_6_plus.pdf

[12] https://www.st.com/resource/en/user_manual/dm00384353-electronic-speed-controller-for-bldc-and-pmsm-three-phase-brushless-motor-stmicroelectronics.pdf

[13] https://www.microchip.com/mymicrochip/filehandler.aspx?ddocname=en012037

[14] https://www.microchip.com/mymicrochip/filehandler.aspx?ddocname=en546013

[15] https://www.microchip.com/mymicrochip/filehandler.aspx?ddocname=en025522

[16] https://www.microchip.com/mymicrochip/filehandler.aspx?ddocname=en530042

[17] https://www.microchip.com/mymicrochip/filehandler.aspx?ddocname=en544825

APPENDIX

[1] https://www.st.com/resource/en/datasheet/stm32f405rg.pdf

[2] https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405415-stm32f407417-stm32f427437-and-stm32f429439-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[3] https://www.st.com/resource/en/user_manual/dm00105879-description-of-stm32f4-hal-and-ll-drivers-stmicroelectronics.pdf

[4] https://www.infineon.com/dgdl/Infineon-IRS2304-DataSheet-v01_00-EN.pdf?fileId=5546d462533600a40153567a8fe72802

[5] https://www.infineon.com/dgdl/irlr7843pbf.pdf?fileId=5546d462533600a40153566de53526d8

[6] http://www.ti.com/lit/gpn/ina199

[7] https://www.diodes.com/assets/Datasheets/LMV331_393.pdf

PROJECT RESOURCES

[1] https://github.com/kogantiharsha/BLDC_MOTOR_CONTROLLER