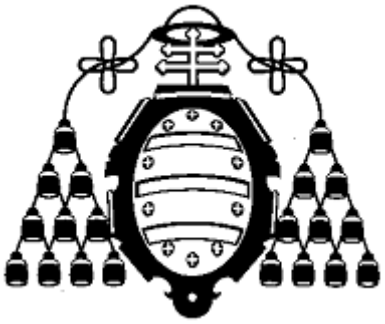


UNIVERSIDAD DE OVIEDO

CENTRO INTERNACIONAL DE POSTGRADO

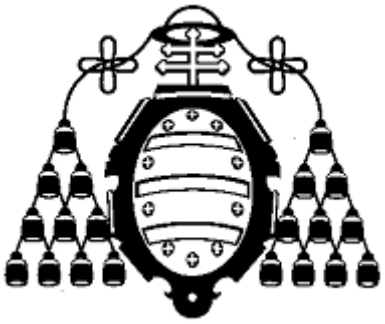
MASTER EN INGENIERÍA MECATRÓNICA

TRABAJO FIN DE MÁSTER

CONTROL OF BLDC MOTORS FOR A TERRESTRIAL LUNAR ROVER PROTOTYPE

JULIO 2014

AUTOR: CRISTINA SERRANO GONZÁLEZ
TUTOR: JUAN CARLOS ÁLVAREZ ÁLVAREZ
TUTOR: ARMIN WEDLER



UNIVERSIDAD DE OVIEDO

CENTRO INTERNACIONAL DE POSTGRADO

MASTER EN INGENIERÍA MECATRÓNICA

TRABAJO FIN DE MÁSTER

CONTROL OF BLDC MOTORS FOR A TERRESTRIAL LUNAR ROVER PROTOTYPE

JULIO 2014

AUTOR:
CRISTINA
SERRANO GONZÁLEZ
[FIRMA]

TUTOR:
JUAN CARLOS
ÁLVAREZ ÁLVAREZ
[FIRMA]

TUTOR:
ARMIN
WEDLER
[FIRMA]

Deutsches Zentrum
für Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft
Institut für
Robotik und Mechatronik
Oberpfaffenhofen
Postfach 11 16
D-82230 Weßling

Resumen

Los 'rovers lunares' han sido un elemento importante en la investigación espacial desde que los soviéticos lanzaron la sonda 'Lunokhod I' en 1970. Para llevar a cabo con éxito su tarea, un rover debe ser capaz de moverse alrededor de la superficie de la Luna, sin importar las condiciones del terreno. Por lo tanto, está claro que motores y otros actuadores son una parte fundamental de cada rover.

A lo largo de los años, se han utilizado varios tipos de motores eléctricos. Hoy en día, los motores BLDC son cada vez más importantes en las aplicaciones industriales, en la investigación, y exploración espacial. El objetivo de este Proyecto Fin de Máster es el desarrollo de un control de motores BLDC en un microprocesador ARM Cortex-A8, que se encuentra dentro de la plataforma de desarrollo BeagleBone Black, y el uso de Matlab/Simulink para crear un regulador Proporcional-Integral que se utilizará para controlar la velocidad.

El desarrollo del controlador de motores se llevó a cabo en las oficinas del Centro Aeroespacial Alemán (DLR) en Oberpfaffenhofen y fue terminado en julio de 2014. En primer lugar, se desarrolló un control de lazo abierto para probar el funcionamiento del motor. Posteriormente, se añadió un control de lazo cerrado para mantener la velocidad. Debido al uso de dos plataformas de desarrollo diferentes, se diseñó un circuito intermedio para interconectarlas. También se instaló un sistema operativo en tiempo real, no sólo para proporcionar una plataforma en la que cargar el código, sino también para ayudar a la integración de Matlab/Simulink dentro del proyecto.

Los resultados, aun siendo aceptables en términos de control e integración, muestran que el uso de un RTOS no tiene apenas ningún impacto importante en la programación del microprocesador. Esta tesis pretende servir como punto de partida para el futuro desarrollo de un control para la placa Phytec diseñada en el DLR.

Palabras clave:
Motores sin escobillas, ARM Cortex-A8, BeagleBone Black, Matlab/Simulink, Teoría de Control

Abstract

Lunar Rovers have been an important element in space research since the soviet's 'Lunokhod I' was launched in 1970. In order to successfully perform its task, a Lunar Rover should be able to move around the surface of the Moon, no matter how bad the condition of the terrain is. Therefore, it is clear that actuators and motors are a fundamental part of every rover.

Over the years, several types of electric motors have been used. Today, BLDC motors are becoming more important in industry, research, and space exploration. The aim of this Master Thesis is to develop a BLDC motor controller for an ARM-Cortex A8 microprocessor, using the BeagleBone Black platform, and Matlab/Simulink to create a Proportional-Integral controller.

The development of the motor controller was conducted at the German Aerospace Center (DLR) in Oberpfaffenhofen and was finished in July 2014. First, an open-loop control was developed to test the operation of the BLDC. Later, the closed-loop speed control was added. Due to the use of two different boards, additional circuitry that will allow to interconnect both boards has to be designed and mounted. The BeagleBone Black board is embedded with a Real-Time Operating System not only to provide a platform in which code is loaded, but also to help the integration of Matlab/Simulink inside the board.

The results, being acceptable in terms on control and integration, show that the use of an embedded RTOS has barely any important impact in programming the microprocessor. This thesis hopes to serve as a starting point to the future development of a BLDC motor control for the Phytec board designed at the DLR.

Key words:
BLDC Motors, ARM Cortex-A8, BeagleBone Black, Matlab/Simulink, Control Theory

Contents

List of Figures

List of Tables

Acronyms

AC	Alternate Current
A/D	Analog to Digital
AQ	Action Qualifier
BBB	BeagleBone Board
BLDC	Brushless DC
DC	Direct Current
DRV	Driver Board
EC	Electronically Commutated
FOC	Field-oriented Control
GPIO	General Purpose Input-Output
IC	Integrated Circuit
IDE	Integrated Development Environment
IGBT	Insulated Gate Bipolar Transistor
ISR	Interrupt Service Routine
MOSFET	Metal-oxide-semiconductor Field-effect transistor
MPU	Microprocessor Unit
PMSM	Permanent Magnet Synchronous Motor
PI	Proportional-Integral
RTOS	Real Time Operating System
SVM	Space Vector Modulation

Part I.

RESUMEN

1. Introducción

La humanidad tiene el afán de descubrir el mundo a su alrededor desde el principio de los tiempos y, durante los últimos cincuenta años, también explorar lo que hay en el espacio exterior. El primer objeto hecho por el hombre en alcanzar el espacio fue el satélite de la soviética 'Sputnik 1' , en 1957, seguido por el cosmonauta Yuri Gagarin, quien en 1961 se convirtió en el primer ser humano en completar una órbita alrededor de la Tierra en el espacio exterior. Tras el éxito de Gagarin ocho años antes, en 1969 la misión americana 'Apollo 11' aterrizó en la Luna.

En 1970, sólo un año después de que el primer ser humano pisara la luna, la URSS lanzó la nave espacial 'Luna 17' que contenía al primer robot no tripulado, el 'Lunokhod I' (Figura 6.1). Ese objeto en concreto, el primero controlado a distancia desde la Tierra a través de una superficie astronómica, marcó el inicio de la exploración espacial autónoma.

Hasta 2014, sólo tres países han desarrollo un rover lunar: la Unión Soviética (Proyecto Lunokhod), los Estados Unidos de América (Apollo Lunar Roving Vehicle) y China (Yutu). Actualmente se encuentran en curso varios proyectos se encuentran en curso como son el 'Luna-Glob ruso' [2], el chino 'Chang'e' [3] y el 'Chandrayaan' indio [4].

1.1. Motivación

Debido a la creciente importancia de los motores eléctricos sin escobillas en aplicaciones industriales, es necesario conocer a fondo la construcción y sus principios con el fin de desarrollar un software capaz de mantenerlos operativos.

Los motores eléctricos sin escobillas, a diferencia de los tipos tradicionales de motores, carece de las escobillas utilizadas para la conmutación de la polaridad del motor. Esto proporciona varias ventajas en el uso de estos motores, pero también añade la necesidad de una conmutación externa, lo cual añade dificultad en el proceso de control. La

conmutación se realiza mediante un programa creado para realizar una secuencia de conmutación, programa que está basado en las lecturas de un sensor o sin sensores.

Debido a las razones anteriormente expuestas, el proyecto contiene nuevos retos que pueden ser resueltos a través de los principios y herramientas de la ingeniería.

1.2. Objetivos

Los objetivos principales de este proyecto son los siguientes:

1. realizar un control para un motor eléctrico sin escobillas utilizando un microprocesador basado en ARM al cual se le instala el RTOS VxWorks. Para una solución temporal simple y barata, se utilizará una plataforma BeagleBone para la programación principal.

2. realizar un control de velocidad con un regulador PI utilizando Simulink. Simulink es una potente herramienta para la creación de diagramas de bloques gráficos que pueden ser integrados en el microprocesador.

2. Motores eléctricos sin escobillas

Hoy en día los motores eléctricos sin escobillas están ganando popularidad en diversas industrias como son la industria de electrodomésticos, automotriz, aeroespacial o la automatización industrial, sobre todo por su mayor eficiencia, par motor y durabilidad; desplazando a los motores de corriente alterna y con escobillas tradicionales. Además, el coste inherente de un motor sin escobillas es menor que el coste de los motores tradicionales, aunque al ser su tasa de fabricación más baja y la necesidad de añadir electrónica adicional puede hacer que el precio sea mayor [5].

Algunas de las ventajas relacionadas con el uso de motores sin escobillas frente a los motores con escobillas, son: [6]

- No hay desgaste de las escobillas, importante para las aplicaciones espaciales
- Mejor velocidad frente al par
- Alta respuesta dinámica
- Alta eficiencia
- Aumenta la vida útil
- Funcionamiento silencioso
- Rangos de velocidad más altos
- Temperaturas de operación más bajas [7]

Un motor eléctrico sin escobillas, o electrónicamente conmutado, es un tipo de motor síncrono alimentado por una fuente de corriente continua, que tiene la singularidad de que carece de escobillas físicas para la conmutación. Esta singularidad aumenta la vida del motor, ya que es una de las piezas que requieren el mayor nivel de mantenimiento, pero

suma la necesidad de una conmutación externa en lugar de la conmutación por escobillas tradicional. A diferencia de los motores con escobillas, que tienen una conmutación mecánica interna para invertir la corriente y el sentido de rotación, la falta de escobillas hace necesario el uso de electrónica y software con el fin de realizar la misma tarea de conmutación.

Estos motores poseen una relación lineal entre la corriente y el par motor, el voltaje y la velocidad. El estator, la parte estacionaria, se compone de láminas de acero, con devanados de cobre enrollados alrededor de las ranuras, que generan un campo electromagnético controlable en magnitud y dirección. El rotor, la parte giratoria, es un imán permanente a base de aleaciones de tierras raras como neodimio (Nd), samario-cobalto (SmCo) o una aleación de neodimio, ferrita y boro (NdFeB), que genera un campo magnético de magnitud constante. Un rotor puede variar desde dos a un número ilimitado de pares de polos, cada uno con sus polos norte (N) y sur (S), que influyen en la relación entre una revolución eléctrica y una revolución mecánica.

Los motores pueden tener dos configuraciones diferentes según la conexión del bobinado. En la conexión triángulo se conectan todas las bobinas entre sí (circuito serie), en la conexión estrella sólo se conecta un extremo de la bobina, mientras se da tensión al otro extremo (circuito paralelo).

A pesar de que la mayoría de los motores BLDC han incorporado sensores de efecto Hall, es posible desarrollar una secuencia de conmutación sin sensores, sobre la base del Back-EMF. Aunque la conmutación sin sensores es menos complejo en términos de hardware y más fiable que conmutación con sensor, para el propósito de este proyecto se elige una conmutación con sensores basado en la posición con el apoyo de los sensores Hall digitales incorporadas.

Hay tres métodos de conmutación con sensores ampliamente utilizados:

- Conmutación sinusoidal.
- Conmutación trapezoidal.
- Control vectorial.

La conmutación sin sensores se basa en el efecto de Back-EMF. Los devanados generan un campo magnético que se opone al campo magnético generado por la tensión inductora, según la ley de Lenz.

La conmutación en bloque, también llamada conmutación trapezoidal debido a la forma de la señal, es la forma más extendida para determinar la secuencia de conmutación de los motores gracias a su simplicidad. Este método se conoce con el nombre de 'Six-step' porque hay seis estados discretos diferentes que sirven para energizar el inversor. Cada uno de los estados es determinado por la lectura de los tres sensores Hall integrados en el motor.

Al contrario que la conmutación en bloque, no demasiado adecuada para aplicaciones de baja velocidad, la conmutación sinusoidal es considerada una buena solución para aplicaciones que requieran velocidades tanto bajas como altas. Puede ser utilizado en aplicaciones que requieren control de velocidad y par o en sistemas de lazo abierto.

Por último, el control vectorial es un método basado en el hecho de que sólo la corriente del estator perpendicular al rotor ayuda en la generación de par, por lo que se hace necesario controlar el vector de corriente manera que el vector de corriente del estator sea perpendicular al rotor de posición en todo momento.

3. Plataformas de desarrollo

BeagleBone Black es una plataforma de desarrollo de hardware de código abierto y bajo coste producida por la Fundación BeagleBoard.org y Texas Instruments. La placa proporciona una forma barata y fácil de programación para desarrolladores de sistemas embebidos, así como la posibilidad de ser utilizado como un ordenador de una placa gracias a su conexión HDMI.

La plataforma BeagleBone Black está siendo cada vez más famosa para el desarrollo de pequeños y grandes proyectos entre los desarrolladores profesionales o aficionados, sobre todo gracias a su comunidad en línea, que contiene los recursos, proyectos y soluciones de problemas que permiten hacerlo accesible para todo tipo de personas.

Algunas de sus características son:

– Procesador: AM335x 1GHz ARM® Cortex-A8
– Connectividad
  - Cliente USB para alimentación y comunicaciones
  - USB adicional
  - Ethernet
  - HDMI
  - 2x 46 pines
– Compatibilidad con software
  - Ångström Linux
  - Android

- Ubuntu
- Cloud9 IDE

La segunda placa utilizada, DRV8312-C2-KIT, es un kit de evaluación para el control de motores desarrollado por Texas Instruments, que permite controlar motores trifásicos sin escobillas y motores síncronos de imanes permanente (PMSM).

El kit incluye, entre otros, un inversor trifásico integrado en la placa base, DRV8312, que soporta hasta 50V y 6.5A, una controlCARD con código preinstalado para operar los motores usando una interfaz gráfica, la XDS100 GUI para emulación, y conexiones UART, SPI y CAN.

- Motor NEMA17 BLDC/PMSM 55W
- Enchufe de corriente de 24V con adaptadores
- DRV8312 baseboard with controlCARD slot
- Piccolo F28035 controlCARD
- Cable USB
- USB Stick with GUI, CCStudie IDE, Quick Start Guide, y links para controlSUITE y documentación

4. Desarrollo

El principal problema que se tuvo que resolver fue la interconexión de las dos plataformas, debido a diferencias de tensión entre ellas. La placa DRV funciona a 5V, algunos componentes necesitan una tensión de 24V, mientras que la plataforma BeagleBone tiene una tensión de 3.3V para los pines E/S. El uso de dos diferentes fuentes de alimentación es algo que debe tenerse en cuenta, ya que fácilmente podría conducir a un mal funcionamiento de las placas. La solución propuesta es el uso de un circuito integrado inversor, como el 74LVX14, de baja tensión en cuya entrada existe un disparador de Schmitt. Este circuito integrado será utilizado para la conversión de voltaje de 5V a 3V.

Se recomienda tener dos fuentes de alimentación distintas, aunque sería óptimo utilizar sólo una, una para alimentar la placa DRV (24V) y la otra la BeagleBone(5V). Aunque el uso de dos diferentes fuentes de alimentación no es el enfoque óptimo, por ser una conexión en paralelo, usar GND común y el circuito integrado LVX para aislarlas entre sí es una solución aceptable, ya que reduce la probabilidad de destruir la placa.

En este proyecto, el primer paso será la creación de un circuito de control abierto es decir, sin realimentación. El control en lazo abierto es un tipo de control que calcula su entrada en un sistema que utiliza sólo el estado actual y un modelo del sistema. Este sistema de control no observa la salida del sistema, por lo que es incapaz de corregir los posibles errores que pudieran aparecer durante el funcionamiento. Se prefiere este control sobre el control de bucle cerrado cuando se necesita simplicidad, bajo coste y la realimentación no es importante para conseguir que el sistema funcione correctamente.

En el control de bucle abierto la entrada se da al modelo del sistema cuya salida, normalmente, se llevará a la entrada del actuador. Sin embargo, este método, existe un paso intermedio entre el sistema modelo y el actuador (el motor): el circuito integrado DRV8312. El bucle será el siguiente: la entrada (velocidad) será una variable en el sistema (el código); la salida del código será el duty de la señal de PWM que será la entrada en el chip DRV. El conductor "transforma" la señal PWM en la tensión necesaria para encender el motor. El motor BLDC tiene una velocidad y un par que, en este caso, no se mide.

El controlador se hará a partir de cero, y se corresponde con el código escrito para el microprocesador ARM. La señal PWM, que será la salida, se generará a través de un algoritmo, en este caso se utilizará la modulación por vector (SVM). La señal PWM también lanzará una función en un momento determinado (cada 25 μs); función necesaria para calcular los nuevos ciclos de PWM.

A diferencia del control de lazo abierto, control en lazo cerrado tiene un bucle de control de realimentación que permite leer la posición real del eje del rotor y corregir cualquier error que el sistema pueda tener durante su funcionamiento. El control en lazo cerrado se utiliza cuando hay una fuerte necesidad de controlar la salida del sistema a fin de evitar errores en el proceso. También se puede utilizar en los procesos de 'machine learning', aunque este no es el caso.

El esquema de control en lazo cerrado es prácticamente igual al control en lazo abierto, añadiendole una rama de 'realimentación' a la salida del sistema. El motor cuenta con tres sensores Hall integrados, lo que proporciona información acerca de la posición del motor y, mediante cálculos, hallar su velocidad. Este último dato se utilizará para calcular el error entre la velocidad deseada y la velocidad real, que se introduce en el controlador PI. Sensor Hall también ofrece información acerca de la posición del eje, lo que tendrá un impacto directo en la secuencia de activación del PWM.

Existen algunos cambios en el uso de PWM y la configuración GPIO, sobre todo en este último con la introducción de las interrupciones de GPIO. No habrá una interrupción cada 25 μs, que será sustituida por las interrupciones de cambio de estado de los sensores Hall.

La integración con Matlab será el elemento final del control en lazo cerrado. El entorno 'Simulink 'se utilizará para crear un proyecto con bloques en el que se calcula la velocidad real del motor y, a continuación, el error entre ese valor y la velocidad deseada será la entrada del controlador PI. El entorno 'Simulink' también conectará las variables de su proyecto con los que están dentro del núcleo a través de las 'S-function'.

'S-functions' es la descripción de un bloque de Simulink que puede ser escrita en varios lenguajes de programación como MATLAB, C / C + +, FORTRAN, etc y compilado como un archivo MEX. Estas funciones pueden alojar tanto sistemas continuos como discretos e híbridos. También es posible implementar un algoritmo en una función y utilizar el bloque S-Function para agregarlo a un modelo de Simulink.

5. Conclusiones

En resumen, esta tesis sirve como una forma de programar un motor BLDC con una plataforma pequeña y de bajo coste como la Beaglebone Black además de la conexión de la placa a una interfaz externa, un PC y cargar un proyecto Matlab dentro del kernel, lo cual es un nuevo enfoque en la forma de creación de los controladores.

La programación de motores eléctricos sin escobillas es algo bien estudiado, sin embargo, el desarrollo de nuevas placas de prototipos, más pequeñas y potentes, también añade complejidad al proceso. Para este proyecto se han hecho dos enfoques, un control de bucle abierto y un control de bucle cerrado. El control de bucle abierto utiliza el algoritmo SVM para crear un vector de tensión rotatorio con el fin de ejecutar la secuencia PWM. Más tarde, el control de bucle cerrado utiliza un controlador para garantizar que la velocidad deseada y la velocidad real del motor de BLDC es la misma en todo momento. Para conectar la acreditación con el entorno Matlab proporciona una poderosa herramienta para crear controladores y realizar cálculos con el entorno Simulink.

Sin embargo, la utilidad de un sistema operativo dentro del procesador, en lugar de programar directamente en él, no se puede asegurar en este momento. Es posible que, en un futuro, la adición de nuevas características podría ser un buen punto a favor del uso de un sistema operativo, en comparación con no usarlo.

Otras líneas de investigación serían el desarrollo de un control de bucle cerrado utilizando otros métodos de conmutación, como 'FOC', ya que el sistema usado no es el mejor enfoque para su uso en aplicaciones de baja velocidad, debido a la simplicidad del mismo. Además, 'FOC' también controla la corriente del motor, lo que podría ser una ventaja para su control. El DLR ha desarrollado también en una nueva plataforma, Phytec, para esta aplicación, por lo que la incorporación del código desarrollado en la nueva placa también puede ser una continuación del proyecto.

Part II.

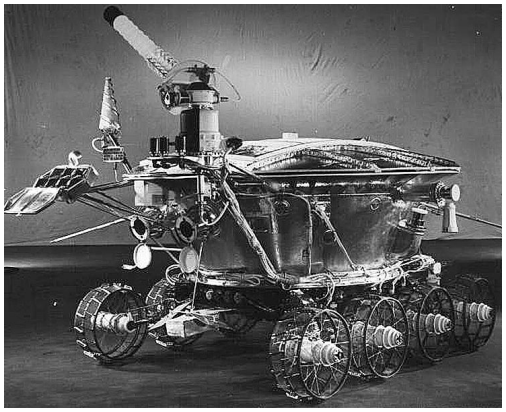
INTRODUCTION

6. Introduction

Humankind has had a desire for discovering the world around him since the dawn of times and, for the last fifty years, also for knowing what there is in outer space. The first human-made object to reach space was the soviet's satellite "Sputnik 1", in 1957, followed by the cosmonaut Yuri Gagarin, who in 1961 became the first human being to complete an orbit around the Earth in the outer space. After Gagarin's success eight years earlier, in 1969 the American Mission "Apollo 11" landed on the Moon.

In 1970, only one year after the first human landing on the Moon, the USSR launched the spacecraft "Luna 17" containing the first unmanned robot, the "Lunokhod I" (Figure 6.1). That particular object, the first to be remotely-controlled from Earth across an astronomical surface, marked the beginning of autonomous space exploration.

Figure 6.1.: Lunokhod I [1]



As of 2014 only three countries have launched a lunar rover: the Soviet Union (Lunokhod Project), the United States of America (Apollo Lunar Roving Vehicle) and China (Yutu). Several projects are currently on-going, such as the Russian "Luna-Glob" [2], the Chinese "Chang'e" [3] and the Indian "Chandrayaan" [4].

6.1. Motivation

Due to the growing importance of BLDC motors in several industrial applications, it is necessary to have an understanding of the construction and working principles of them in order to develop a software capable of operating them.

BLDC motors, unlike the traditional types of motors, lacks the brushes used for physical commutation. This provides several advantages in the use of BLDC motors, but also adds the need for an external commutation that adds difficulty to the process of motor control. The commutation is done using a program created to perform the commutation sequence, based on the readings from sensor or performing a sensorless commutation.

Due to the reasons above explained, the project contains new challenges that can be resolved through engineering principles and tools.

6.2. Objectives

The main goal of this thesis is:

1. to run a BLDC motor using an ARM-based microprocessor embedded with the RTOS VxWorks. For a simple and inexpensive temporary solution, a BeagleBone Board is going to be used as the main programming platform. Since a PWM Motor Driver is needed, the processor board is going to be connected to a driver board (DRV8312EVM) which contains the necessary microchips and circuitry to convert the PWM signals into an AC supply.

2. to speed control the BLDC though a PI-controller using Simulink. Simulink is a strong tool for graphic block diagramming that could be interfaced with the ARM-based microprocessor.

Part III.

STATE-OF-THE-ART

7. Brushless DC Motors

Brushless DC motors are a rather new type of motor, the first one was developed in 1962 by T.G. Wilson and P.H. Trickey. In that moment, even if they were a good choice for its lack of brushes for commutation, they could not drive as much power as traditional DC motors could. That changed with the appearance of permanent magnet materials in the 1980s.

Nowadays BLDC Motors are gaining popularity in diverse industries such as Appliances, Automotive, Aerospace, and Industrial Automation, mainly for its higher efficiency, torque and durability, displacing stepper motors, AC motors and traditional brushed DC motors. Also, the inherent cost of a BLDC is lower than that related to brushed DC motors, although its lower manufacturing rate and the need to add drive electronics may cause the price to rise [5]. Some of the advantages related to the use of BLDC motors, in contrast to brushed motors, are: [6]

- No brush wear, important for space applications
- Better speed versus torque characteristics
- High dynamic response
- High efficiency
- Long operating life
- Noiseless operation
- Higher speed ranges
- Lower operating temperatures [7]

In addition, the ratio of torque delivered to the size and weight of the motor is higher, making it useful in applications where space and weight are critical factors [6].

This section is dedicated to explain the basics about BLDC motors: physical construction, operating principles and commutation.

7.1. Construction and operating principle

A BLDC or electronically-commutated DC motor (EC Motor), is a type of synchronous motor powered by a DC source, that have the singularity of lacking physical brushes for commutation. This singularity improves the operating life of the motor, as it is one of the parts that requires the highest level of maintenance, but adds the necessity of an external commutation instead of the traditional brush commutation. Unlike brushed DC motors, which have a internal mechanical commutation to reverse motor windings' current in synchronism with rotation, the lack of brushes require the BLDC motors to have electronics in order to perform the same task.

BLDC Motors can be referred as a reversed brushed DC motor and, like them, have a linear relationship between current and torque, voltage and speed. The stator, the stationary part, of a BLDC is made up of stacked steel laminations with windings (copper wire) placed around the slots that generate an electromagnetic field of controllable magnitude and direction, whereas the rotor, the rotating part, is a permanent magnet, usually rare earth alloy magnets such as Neodymium (Nd), Samarium Cobalt (SmCo) and the alloy of Neodymium, Ferrite and Boron (NdFeB), that generates a magnetic field of constant magnitude. A rotor can vary from two to an unlimited number of pole pairs, each with its North (N) and South (S) poles, which have an influence on the relationship between an electrical revolution and a mechanical revolution. Figure 7.1 shows the physical model of BLDC motors.

BLDC motors can have two different winding configuration. Delta configuration connects all windings to each other following a series circuit pattern, whereas Wye configuration (Y or Star configuration) only connects together one end of the winding, while applying voltage to the other end (parallel circuit). Figure 7.2 shows these configurations. Another difference is, while delta configuration gives low torque at low speed and low voltage; Wye configuration gives high torque at low speed, but requires more voltage. The difference between Wye/Delta configuration is always a factor of 1.73 because of the way that the windings of an induction motor are put together [8].

Figure 7.1.: Construction of a BLDC motor

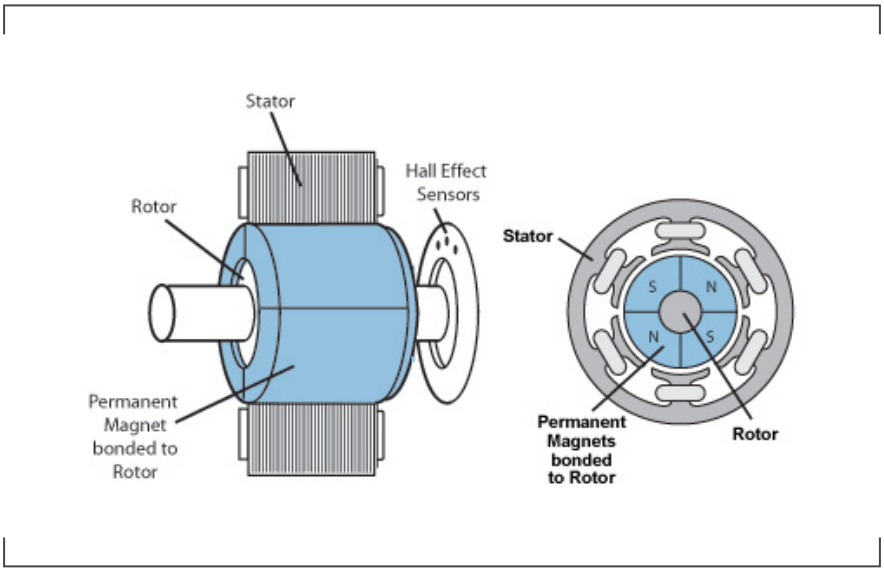
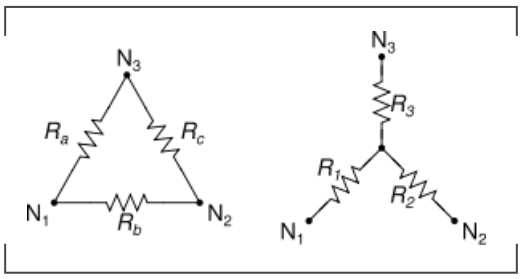


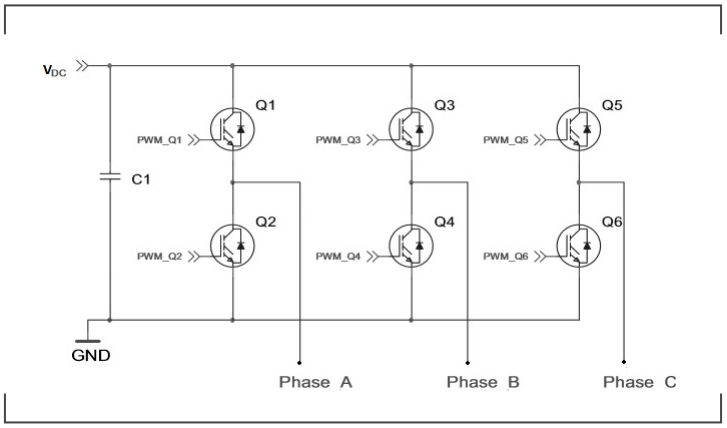
Figure 7.2.: Winding configurations



The 3-phase supply is controlled by the switching of a 3-phase bridge with six transistors (IGBT or MOSFET), converting the PWM signals from the microprocessor into an AC supply (figure 7.3). Each bridge of the inverter is connected to a motor phase; in the figure, Q1 and Q2 to Phase A; Q3 and Q4 to Phase B, and Q5 and Q6 to Phase C. Simultaneously, Q1, Q3 and Q5, which are connected to the voltage, form the 'High side' of the inverter, and Q2, Q4 and Q6, connected to ground, the 'Low Side'.

The transistors switch synchronously with the rotor position and only two transistors could be switched on at the same time in order to energise, positively or negatively, two of the three phases (the third one should be switched off or floating). For example, when energizing the windings as step (1) in figure 7.4, current entering Phase A and leaving through Phase C, the transistors switched on should be Q1 (on the high side) and Q6 (on the low side), which close the circuit. For each of the six steps, one step equals

Figure 7.3.: 3-phase BLDC Inverter



60° electrical , different transistors are switched on and off, changing its state only one of them, but remaining active only two at a time. Table 7.1 summarises the switching sequence for all the steps.

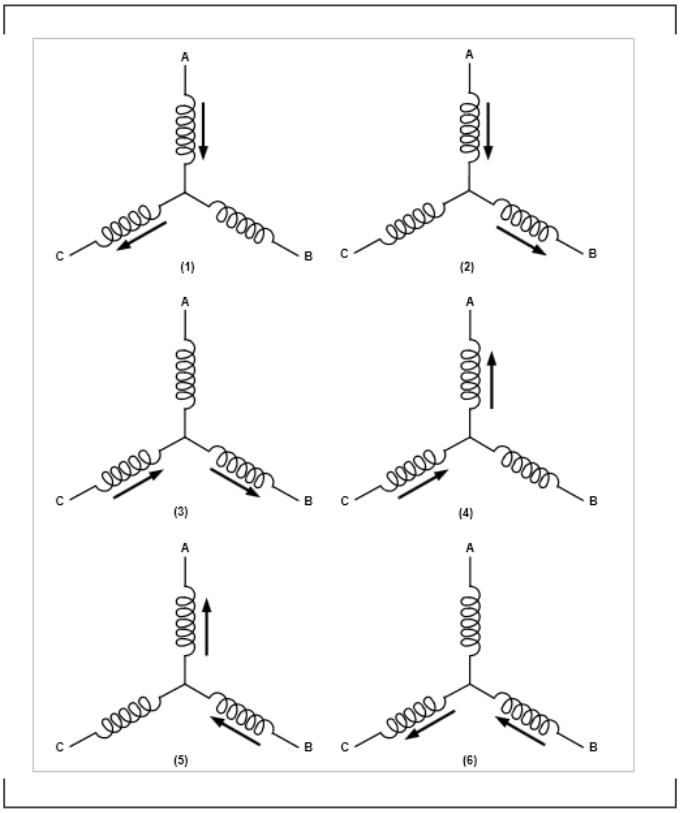
Table 7.1.: Switching sequence

Step	Phase A	Phase B	Phase C	Transistors ON
1	V_{DC} +	V_{DC} -	0	Q1 - Q4
2	V_{DC} +	0	V_{DC} -	Q1 - Q6
3	0	V_{DC} +	V_{DC} -	Q3 - Q6
4	V_{DC} -	V_{DC} +	0	Q3 - Q2
5	V_{DC} -	0	V_{DC} +	Q5 - Q2
6	0	V_{DC} -	V_{DC} +	Q5 - Q4

As previously said, BLDC motors are electronically commutated, meaning that the stator windings should be energised in a sequence, so that the position of the rotor should be known. Hall Sensors embedded into the non-driving end of the stator usually fulfil these requirements, but there are others such as encoders, analogue Hall sensors, magneto-resistive sensors, used when the resolution of the digital Hall sensor is not enough; or sensorless, by reading the BEMF. The existing methods for BLDC commutation, both sensored and sensorless, which will be summarised and analysed in the next section.

Motors rotate due to the torque produced by two interacting magnetic fields forming an angle (ϕ), as shown in the following equation. Then, for optimal torque, the stator's magnetic field (β_f) should be located 90° in front of the rotor field (β_f) [7]. The previous statement should be taken into account when deciding the feedback system for the control loop, as it has a impact on the type of sensors that are going to be used (Hall, encoder or

Figure 7.4.: Winding energizing sequence [6]



other type of magnetoresistive sensors), though for this project only digital Hall sensors are going to be used.

$$T = K\beta_f\beta_s sin\phi \tag{7.1}$$

The motor used for this project is a NEMA17 BLDC Motor included in the DRV 8312 - EVM Kit by Texas Instruments and its characteristics can be found in Anaheim Automation's BLY17 Series datasheet [9]. Next table 7.2, summarises some of the specifications for model BLY172S-24V-4000 in Imperial Units, as they were given by the manufacturer.

Table 7.2.: Characteristics of the motor

Phases	Rated Voltage (V)	Rated Speed (RPM)	Rated Power (W)	Rated Current (A)	Peak Torque (oz-in)	Rotor Inertia (oz-in- sec2)	Pair Poles
3	24	4000	53	3.5	54	0.00068	4

7.2. Commutation Methods

The commutation circuit can be implemented with discrete components or dedicated control Integrated Circuits (IC), the latter requiring less or no additional circuitry. Usually, the design with discrete components requires a lot of effort from the engineers, is more time-consuming in terms of hardware design and troubleshooting [10].

Despite most of the BLDC motors have embedded Hall sensors, it is possible to develop a sensorless commutation sequence, based on the 'Back Electromotive Force' (Back-EMF). Though back-EMF commutation is less complex in hardware than sensored commutation and more reliable than sensored commutation [11], for the purpose of this project a position-based sensor commutation with the support of the in-built digital Hall sensors is chosen.

There are three widely used sensored commutation methods:

- Sinusoidal commutation.
- Six-step (Trapezoidal or Block) commutation.
- Field-oriented control or vector commutation.

7.2.1. Sensorless commutation

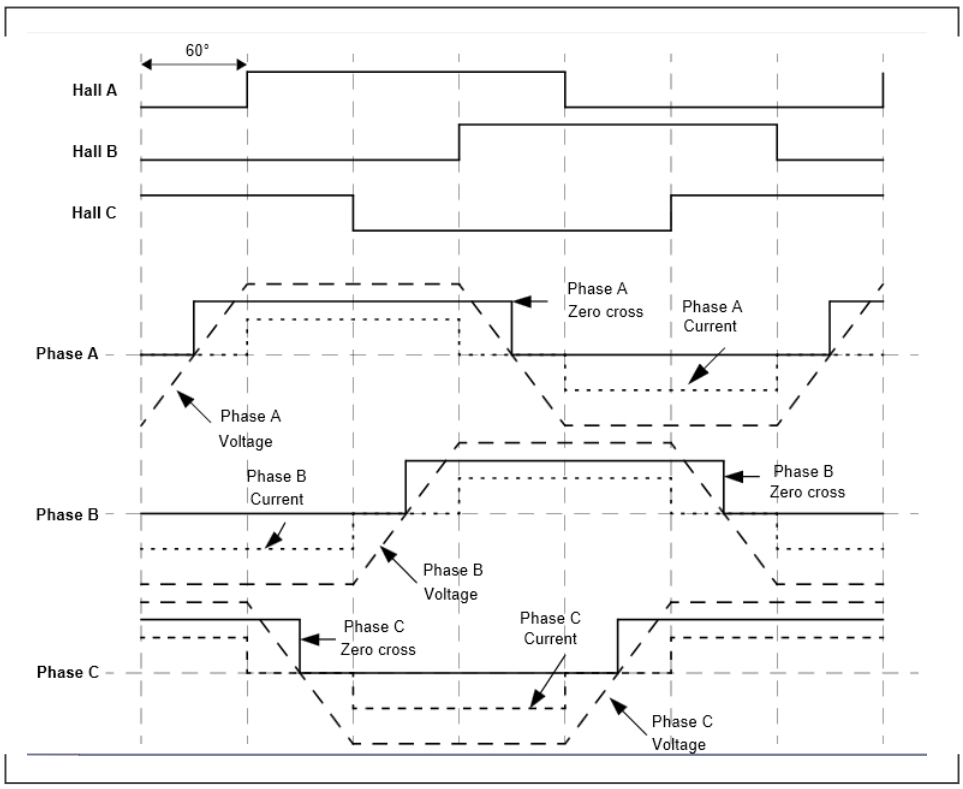
Sensorless commutation of BLDCs is based on the effect of Back-EMF. The windings generate a magnetic field, which is opposed to the magnetic field generated by the energizing voltage, as Lenz's Law formulates: "An induced electromotive force (EMF) always gives rise to a current whose magnetic field opposes the original change in magnetic flux.". Mathematically, specially for electrical motors, is represented as [12]:

$$\varepsilon = NIr\beta\omega \tag{7.2}$$

Once the motor is designed, the number of winding turns (N), the rotor's lenght (l) and radius (r), and the magnetic field (β) remain constant, leaving the angular speed of the rotor(ω) as the only variable that governs the EMF.

As seen with the Hall sensors, the BEMF is also 120° out of phase to each other but both signals aren not synchronous (in figure 7.5 the difference is 30°). At each sequence, two winding phases are connected to power supply, while the third one is turned off. The result is a trapezoidal waveform (dashed-line in figure 7.5) that crosses the "zero-line" each 60°. The combination of the zero crossings determine the commutation sequence for the motor.

Figure 7.5.: Waveform of Hall Sensors vs BEMF [12]

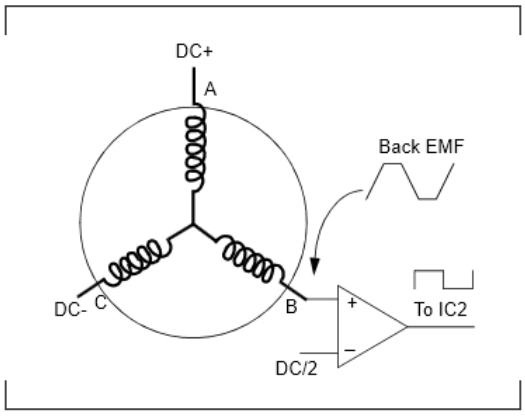


BEMF zero crossings can be detected by comparing the BEMF to half of the DC bus voltage by using comparators as shown in figure 7.6. When there is BEMF at Phase B, it increases and decreases as DC+ and DC- are connected or disconnected to the winding terminals. Each phase of the motor needs a circuit like figure 7.6 to determine the operating sequence. However, this method has the disadvantage of drawing excessive current due to phase shifting if the windings don't have identical characteristics.

Another method of detecting BEMF is the use of A/D converters to measure the voltage. A signal conditioning circuit should lower the signal until a value that the A/D converter can read, then the signal is sampled and compared to a value corresponding to the zero value. Once both values match, the commutation sequence is updated. This method is

more flexible than the comparator method, as the microcontroller has absolute control over the zero crossing value [12] .

Figure 7.6.: BEMF detecting with comparator [12]



Since BEMF is proportional to speed, it is possible that at lower speed the system is not able to detect the zero crossings, making it impossible to start the motor from standstill. This drawback is overcome by starting the motor in open loop and then changing the mode to BEMF sensing [12].

7.2.2. Block commutation

Block commutation, also called six-step commutation or trapezoidal commutation (due to the shape of the signal), is the most widespread way of determining the commutating sequence of BLDC motors thanks to its simplicity and results. The method is called 'Six-Step' because there are six different discrete states to drive the inverter bridge. Each state can be determined by reading the status of the three Hall sensors embedded in the motor.

Each six commutation steps, the rotor turns one electrical revolution. The number of electrical revolutions required for a mechanical revolution depends on the number of pole pairs: one pole pair equals one electrical revolution. The motor used in this project has four pole pairs, thus requiring four electrical revolutions to complete one mechanical revolution.

Whenever the rotor poles pass near the Hall sensors, they give a signal (high or low) that indicates the moment in which the pole is passing near the sensor. Most BLDC motors have three Hall sensors situated 120 electrical degrees apart. Knowing the combination

of the Hall sensors, it is possible to roughly determine the position of the rotor. Digital Hall sensors behave the same way as the transistors do, with only one sensor changing its value at each time.

As stated in section 2.1, for maximum torque efficiency the stator field and rotor field should be 90° apart, however, the Hall sensor's resolution is 60°, therefore it is only able to detect with certainty an angle between 60° and 120°. This leads to an undetectable error of maximum 30° that causes a small torque decay.

In table 7.3, table 7.1 is updated to add three columns for the Hall sensors represent each possible combination of the sensor's signals. The order of the sequence's values of table 7.1 will follow the one shown in Texas Instruments' DRV 8312 datasheet, which will be included in the Annex, as it is the driver chip that is going to be used in this project.

Table 7.3.: Switching sequence including hall sensors

Step	Hall A	Hall B	Hall C	Phase A	Phase B	Phase C	Transistors ON
1	1	0	1	V_{DC} +	V_{DC} -	0	Q1 - Q4
2	1	0	0	V_{DC} +	0	V_{DC} -	Q1 - Q6
3	1	1	0	0	V_{DC} +	V_{DC} -	Q3 - Q6
4	0	1	0	V_{DC} -	V_{DC} +	0	Q3 - Q2
5	0	1	1	V_{DC} -	0	V_{DC} +	Q5 - Q2
6	0	0	1	0	V_{DC} -	V_{DC} +	Q5 - Q4

In this method, there is a torque ripple with a magnitude up to 13% of the maximum torque, being more noticeable at low speed. For this reason, this approach is more suitable for high speed applications where torque ripple has low or no importance [7].

7.2.3. Sinusoidal commutation

In contrast to block commutation, which was not suitable for low speed applications, sinusoidal commutation is regarded as a good solution for both low and high speeds, as sinusoidal eliminates the torque ripple. It can be operated as an open-loop or closed-loop configuration in applications requiring speed and torque control [10]

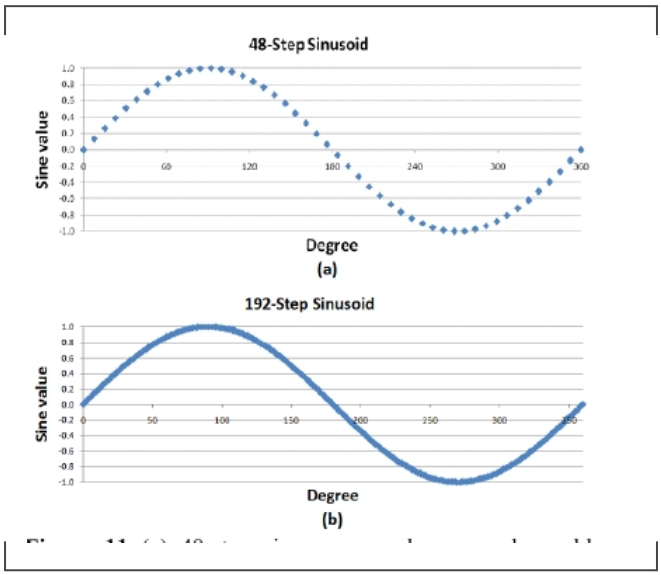
In sinusoidal commutation, the windings are energized by three sinusoidal shaped signals shifted 120° apart. The sinusoidal waveform is generated by PWM signals, varying gradually instead of each 60° as in six-step control.

Torque at sinusoidal commutation can be expressed as the product of a motor constant (K) by the average current that flows into the three phases (I). Torque does not depend on the rotor position anymore, and a correctly executed control implies that the torque remains constant and the rotor rotates smoothly [10].

$$T = 1.5KI \tag{7.3}$$

The fact that torque is not dependant on the position of the rotor doesn't imply that knowing the position is not required, in fact, it is needed to determine the commutation and to maintain the required 90^{o} angle between the magnetic fields in order to have the optimal torque. Hall sensors don't have enough resolution for a smooth behaviour, therefore the use of optical encoders, resolvers or other high resolution sensors is strongly recommended.

Figure 7.7.: Sinusoidal commutation [7]



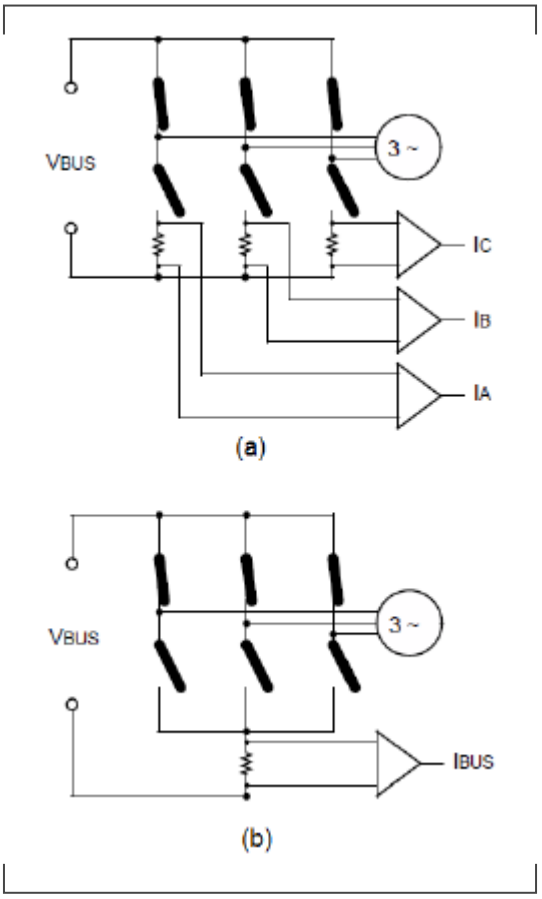
In summary, sinusoidal commutation provides optimal (constant) torque control but imposes certain position feedback requirements. Also, the use of encoders or resolvers for position control can lead to a more expensive system and the complexity of the control increases.

7.2.4. Field-oriented control

Field-oriented control is a method based on the fact that only the stator current that is perpendicular to the rotor helps to generate torque, then it is practical to control the current vector in a way that the stator's current vector is perpendicular to the rotor's position at all time.

The current in each phase is measured through the shunt resistor in a three-shunt bridge (a in Figure 7.8) or reconstructing the current using information from a single shunt resistor (b in Figure 7.8). In order to keep the stator's current vector perpendicular to the rotor's position, the part of the current parallel to the rotor, direct current (I_d), should be zero, and the one perpendicular to the rotor, quadrature current (I_q), depends on the desired motor speed [7].

Figure 7.8.: Current sensing [7]



The BLDC motor is a three-phase sinusoidal system, adding difficulty to the calculations needed in this method. The Clarke Transformation (figure 7.9 converts the three-phase system (A, B, C) into a two-phase time variant system (α, β) and a two-phase invariant system (d, q) is obtained applying the Park Transformation (figure 7.10). This last system uses the previously said direct and quadrature currents.

Figure 7.9.: Clarke Transformation [7]

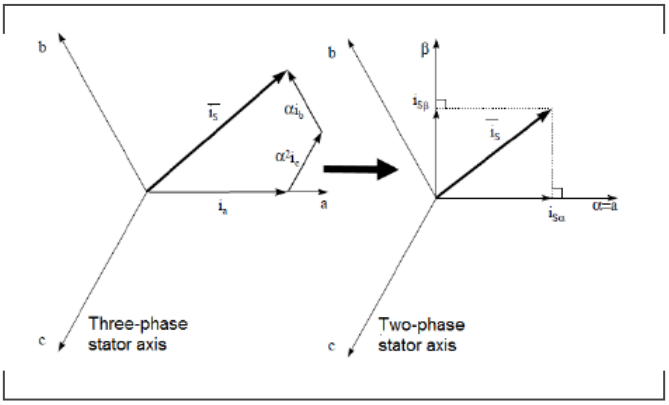
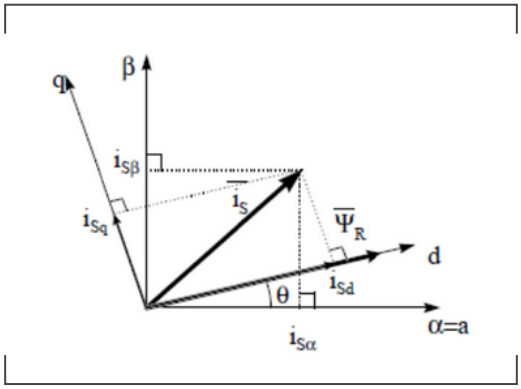


Figure 7.10.: Park Transformation [7]



The direct and quadrature currents are fed into a PI controller, whose output is a voltage in two-phase axis. The values are converted back into a three-phase system applying inverse Park and Clarke transforms and, finally, applied to the half-bridges of the BLDC though Space Vector Modulation (SVM).

SVM generates a voltage vector with certain direction and magnitude, which causes current to flow through the coil. These voltages can be applied on the stator in six different directions for a certain time, generating an equivalent voltage (V_{ref}) that controls the current vector. Manipulating the voltage vector has an impact in the stator current vector.

FOC is the most complex algorithm for BLDC commutation, and depends heavily in the accuracy of the current measured, the accuracy of the angle between the rotor and the stator axes and the processing time between current measurements in order to be successfully implemented [7].

7.3. Summary of Commutation Methods

Each of the commutation methods described above are used when different requirements should be fulfilled, whether it is speed, torque or the need to have an algorithm as less complex as possible. Table 7.4 summarizes the three sensored control methods and shows their most important features.

Table 7.4.: Comparison of sensored commutation methods [10]

Commutation Methods	Speed Control	Torque Control		Required feedback devices	Algorithm complexity
		Low Speed	High Speed		
Trapezoidal	Excellent	Torque Ripple	Efficient	Hall	Low
Sinusoidal	Excellent	Excellent	Inefficient	Encoder, resolver	Medium
FOC	Excellent	Excellent	Inefficient	Current sensing, encoder	High

Part IV.

PROTOTYPING PLATFORMS

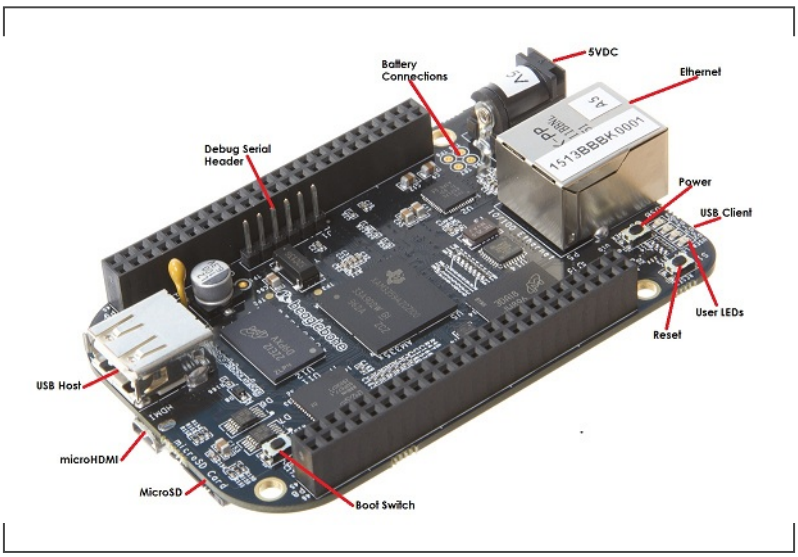
8. Prototyping Platforms

8.1. BeagleBone Black

BeagleBone Black is a low-cost ($45), low-power, open-source hardware development platform produced by the BeagleBoard.org Foundation and Texas Instruments. The board provides a cheap and easy way of programming for embedded developers, as well as the possibility of being used as a single-board computer thanks to its HDMI connection.

The BeagleBone platform is getting more attention for both big and small projects and professional or amateur developers, thanks to its online community, which contain resources, projects and troubleshooting, making it accessible for all kinds of people.

Figure 8.1.: BeagleBone Black [16]



Several add-ons (capes) have been developed for the BeableBone, including Touchscreens, prototyping boards or serial communication boards.

The characteristics of BeagleBone Black are:

– Processor: AM335x 1GHz ARM® Cortex-A8
– Connectivity

  - USB client for power & communications
  - USB host
  - Ethernet
  - HDMI
  - 2x 46 pin headers

– Software Compatibility

  - Ångström Linux
  - Android
  - Ubuntu
  - Cloud9 IDE

Though it is not stated on the list, the software installed on the BeagleBone used for this project will be the proprietary RTOS VxWorks, which will be booted from an SD-card.

8.1.1. AM335x 1GHz ARM® Cortex-A8

The MPU integrated in the BeagleBone is an AM335x 1GHz ARM® Cortex-A8 from Texas Instruments.

  - 512MB DDR3 RAM
  - 2GB 8-bit eMMC on-board flash storage
  - NEON floating-point accelerator

- 2x PRU 32-bit microcontrollers
- Up to Four Banks of General-Purpose IO (GPIO); 32 GPIOs per Bank (Multiplexed with Other Functional Pins). BeagleBone Black has 66 operational GPIO pins out of the total.
- GPIOs Can be Used as Interrupt Inputs (Up to Two Interrupt Inputs per Bank)
- Six UARTs
- Up to Three 32-Bit Enhanced Capture Modules (eCAP)
- Up to Three Enhanced High-Resolution PWM Modules (eHRPWM)
- Boot modes

The BeagleBone Black will be used as the main processor board, driving out the PWM signals required for the commutation and reading the Hall sensor signals required for the control loop.

8.2. Three Phase BLDC Motor Kit

The DRV8312-C2-KIT is a motor control evaluation kit developed by Texas Instruments, for spinning three-phase brushless DC (BLDC) and permanent magnet synchronous (PMSM) motors. The kit includes sub-50-V and 7-A brushless motors for driving medical pumps, gates, lifts and small pumps, as well as industrial and consumer robotics and automation applications.

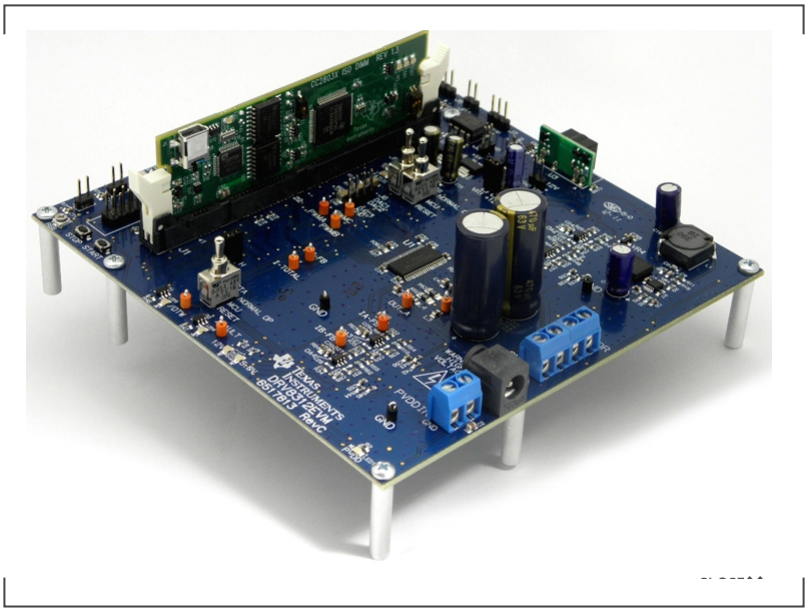
The kit includes, among others, a DRV8312 three phase inverter integrated power module base board supporting up to 50V and 6.5A with controlCARD interface -C2000 Piccolo F28035 controlCARD (pre-flashed with code to spin all motors using GUI)- GUI- Isolated XDS100 Emulation, UART, SPI and CAN Connectivity.

Each kit includes:

- 1 NEMA17 BLDC/PMSM 55W Motor
- 24V wall power supply (with adapters)

- DRV8312 baseboard with controlCARD slot
- Piccolo Isolated F28035 controlCARD
- USB Cable
- USB Stick with GUI, CCStudie IDE, Quick Start Guide, and link to controlSUITE for all documentation

Figure 8.2.: DRV8312EVM board [16]



8.2.1. Three Phase Brushless DC Motor Driver - DRV8312

Typical microcontrollers or microprocessors drive out signals up to 5V or less, for example the BleagleBone Black board GPIO output is 3.3V and a current of 8mA. Motors typically require voltages or currents that exceed what can be provided by the analogue or digital signal processing circuitry that controls them. For this, there should be a circuit, or IC, whose purpose will be that of 'amplifying' the microcontoller's signals. The motor driver provides the interface between the signal processing circuitry and the motor itself.

The DRV8312/32 included in the kit are high performance, integrated three phase motor drivers with an advanced protection system. Some of their characteristics are [13]

- Two power supplies, one at 12V for GVDD and VDD, and another up to 50V for PVDD
- Up to 500kHz PWM switching frequency
- Protection system against fault conditions:short-circuit, overcurrent, undervoltage, and thermal.
- Current-limiting circuit that prevents device shutdown during load transients such as motor start-up

Figure 8.3.: DRV8312 IC [16]

9. VxWorks RTOS

VxWorks is a real-time operating system (RTOS) developed as proprietary software by Wind River. First released in 1987, VxWorks is designed for use in embedded systems, and currently is used mainly in robotics, spacecraft and transportation systems. It has been ported to a number of platforms, including the x86 family, MIPS, PowerPC (and BAE RAD), Freescale ColdFire, Intel i960, SPARC, Fujitsu FR-V, SH-4 and the closely related family of ARM, StrongARM and xScale CPUs.From 2011 is also available for 64-bit systems [14].

Among the features of the RTOS it can be found:

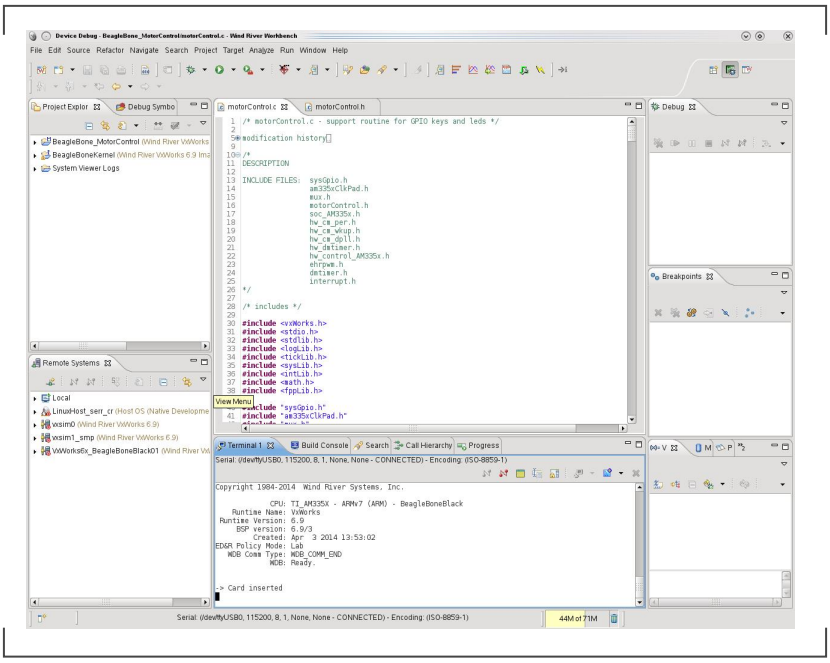
- Multitasking kernel with preemptive and round-robin scheduling and fast interrupt response.
- Native 64-bit operating system (only one 64-bit architecture supported: x86-64)
- User-mode applications ("Real-Time Processes", or RTP)
- Error handling framework
- Binary, counting, and mutual exclusion semaphores with priority inheritance
- Local and distributed message queues
- POSIX PSE52 certified conformity in user-mode execution environment
- File systems: High Reliability File System (HRFS), FAT-based (DOSFS), Network File System (NFS)

Cross-compiling (a compiler capable of creating executable code for a platform other than the one on which the compiler is running) is available in VxWorks. Development is done on a "host" system, for example Windows or GNU/Linux, where an integrated

development environment (IDE), including the editor, compiler toolchain, debugger, and emulator can be used. Software is then compiled to run on the "target" system, in this case the BeagleBoneBlack platform.

Until VxWorks 5.x, the Tornado IDE was used as a development environment, changing to an Eclipse-based IDE (figure 9.1) from version 6.x onwards, WindRiver Workbench.

Figure 9.1.: A view of VxWorks Workbench environment



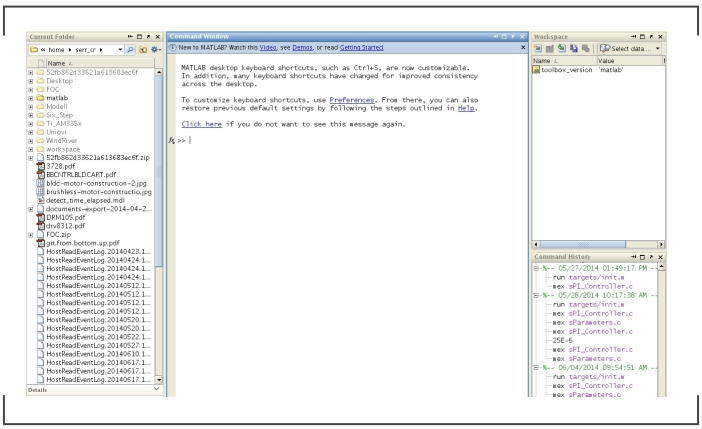
10. Matlab/Simulink

MATLAB is a multi-paradigm (capable of supporting different programming paradigms as OOP, logic, symbolic...) numerical computing environment developed as proprietary software by MathWorks. MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, Java, and Fortran. An additional package, Simulink, adds graphical multi-domain simulation and Model-Based Design for dynamic and embedded systems.

Simulink, developed by MathWorks, is a data flow graphical programming language tool for modeling, simulating and analysing multidomain dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. Simulink can automatically generate C source code for real-time implementation of systems. As the efficiency and flexibility of the code improves, this is becoming more widely adopted for production systems,in addition to being a popular tool for embedded system design work because of its flexibility and capacity for quick iteration. Embedded Coder creates code efficient enough for use in embedded systems.

A view of Matlab's environment is found in figure 10.1.

Figure 10.1.: A view of Matlab's environment



Part V.

DEVELOPMENT

11. Hardware and software frameworks

11.1. Board interconnection

As seen in the previous section two different prototyping boards are going to be used in order to perform a basic motor control. Initially, the Texas Instruments IDK was going to be used instead of the BBB as the main processor board, but it was extremely complex to interface with the driver board and was, consequently, rejected as the processor board. Both the IDK and the BBB shared the type of microprocessor, an ARM-Cortex A8 AM3359, therefore this change didn't lead to software incompatibilities that could increase the complexity of the code.

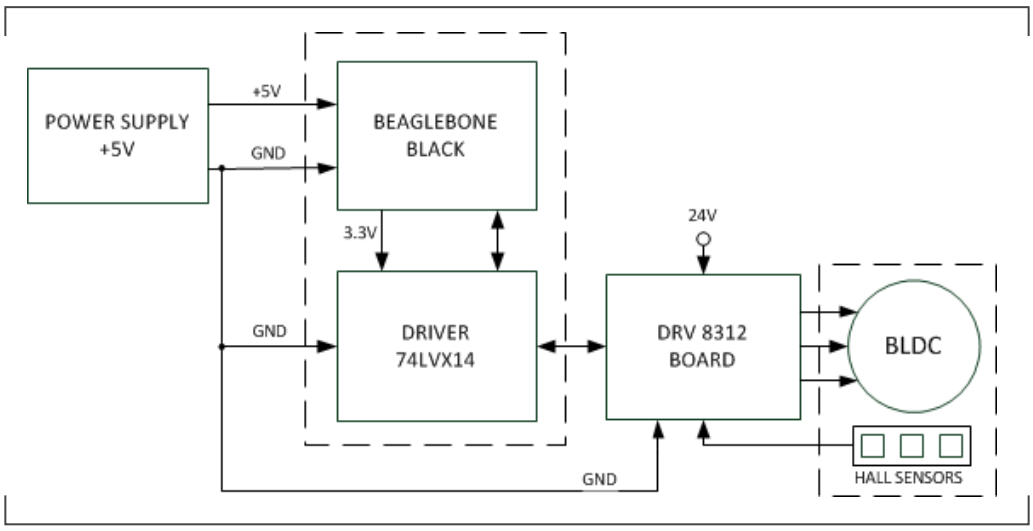
The most troublesome issue to resolve was the interconnection of the two boards, due to voltage differences. The DRV board runs at mainly 5V, with some components at 24V, while the BBB is a 3.3V board. The use of two different power supplies was also something to be taken into account, as it could easily lead to the malfunction of the boards. The proposed solution is the use of a inverter, such as the 74LVX14, a Low-Voltage inverter with Schmitt Trigger input used for 5V to 3V voltage conversions. The LVX also has to convert the 5V input signals into 3.3V signals that the BBB admits.

Figure 11.1 shows a connection diagram including all the necessary elements for the motor control: microprocessor, driver, circuitry... It is recommended to have two different power supplies, though it would be optimal to have only one, for powering the DRV board (24V) and the BBB (5V). Although the use of two different power supplies is not the optimal approach, parallel supply, putting the two boards in common GND and using the LVX IC for isolation is a rather good solution, as it reduces the probability of destroying the board.

The DRV board is powered by a 24V connector included in the kit. The BBB also includes a power connection, but it is not going to be used in this project. Instead, a power supply is connected to the BBB's GND (P9- Pins 1 and 2)and VDD_ 5V (P9-Pins 5 and 6), that provide a safer way to power the system. The BBB should be powering the

LVX through its 3.3V outputs and all the GND points are connected to one of the GND points in the BBB or the line from the power source. There is no difference whether the general line or the BBB pins should be used, as all the BBB's GND pins are internally connected, but at least one source has to be connected to the DRV board's GND (J5-Pins 19, 20, 27, 30, 39, 40), due to the reasons explained on the previous paragraph.

Figure 11.1.: Block diagram for physical connection



Other signals that should go between the two boards are the PWM signals, Hall Sensor lines and Reset lines. They should go though the LVX IC as previously said, taking care of connecting inputs and outputs adequately. The 74LVX14 has six inputs and corresponding outputs, whereas there are three PWM signals, three Hall Sensor signals and three Reset lines; a total of nine different signals. For that reason, two different IC are going to be used, one for Hall Sensor and Reset lines and the other one for PWM signals. PWM signals and reset lines should be connected from the BBB to the LVX's inputs and from there to the DRV board, as it is the processor who is in charge of driving out the signals. On the other hand, the sensor's signals should be connected from the DRV to the LVX's inputs, and the output to the BBB; because the DRV board is the board generating the signals, while the BBB receives and processes them.

The microprocessor AM3359 Cortex-A8 has a PWM subsystem with six PWM outputs, PWM0-2 with two PWM outputs each (A and B), three of which will be used to generate the voltage needed by the motor. The chosen PWM signals are: PWM0-A, PWM2A and PWM2B, each of which has a corresponding pin output in one of the BBB output headers, P8 and P9. The BBB output pins are then connected to the input of the LVX IC, the output of the LVX to a 40-pin header that is connected to the DRV board J5 header. The final connections for PWM signals are as following:

- PWM2-A BBB: P8-Pin 19 (EHRPWM2A)→ 74LVX14/1: Pin 1 (I_0) - Pin 2 (O_0) → DRV8312 Board: J5-Pin 26 (PWMA)

- PWM0-A BBB: P9-Pin 22 (UART2_RXD)→ 74LVX14/1: Pin 3 (I_1) - Pin 4 (O_1) → DRV8312 Board: J5-Pin 25 (PWMB)

- PWM0-B BBB: P9-Pin 21 (UART2_TXD)→ 74LVX14/1: Pin 5 (I_2) - Pin 6 (O_2) → DRV8312 Board: J5-Pin 28 (PWMC)

It was previously stated that Hall sensors are a vital part of BLDC motors, therefore they should be implemented into the physical system. Hall sensors are usually a digital type of sensors with two states, active or inactive. Connecting the Hall sensors to a GPIO pin is the easiest way to read the pin's status in the BBB and process the results. Out the 66 GPIO pins, three of them are going to be used as the input for the sensors. The signal flow is opposite to the PWM, from the DRV board to the BBB, as the BLDC is connected to the DRV board and the BBB receives the signals and processes them. It is mandatory to convert these signals from 5V to 3.3V, what the LVX does. Below it is shown how the signals are connected.

- HALLA DRV8312 Board: J5 -Pin 16 (CAP1)→ 74LVX14/2: Pin 9 (I_3) - Pin 8 (O_3) → BBB: P8-Pin 36 (UART3_CTSN) - GPIO2-16

- HALLB DRV8312 Board: J5-Pin 11 (CAP2)→ 74LVX14/2: Pin 11 (I_4) - Pin 10 (O_4) → BBB: P8-Pin 35 (UART4_CTSN) - GPIO0-8

- HALLC DRV8312 Board: J5-Pin 14 (CAP3)→ 74LVX14/2: Pin 13 (I_5) - Pin 12 (O_5) → BBB: P8-Pin 34 (RART3_CTSN) - GPIO2-17

Finally, the DRV8312 has an input for Reset action. The $\overline{RESETx}$ lines are activated or not depending on the operation mode that is selected. The signals are driven out from the BBB to the DRV board, just like the PWM signals, but due their mode of operation, only using two states: on and off, GPIO pins are used. Once more, three GPIO pins are selected from the 63 remaining outputs, and connected as following.

- $\overline{RESETA}$ BBB: P8-Pin 22(GPIO1_5)→ 74LVX14/2: Pin 1 (I_0) - Pin 2 (O_0) → DRV8312 Board: J5-Pin 6 (RESET_A)

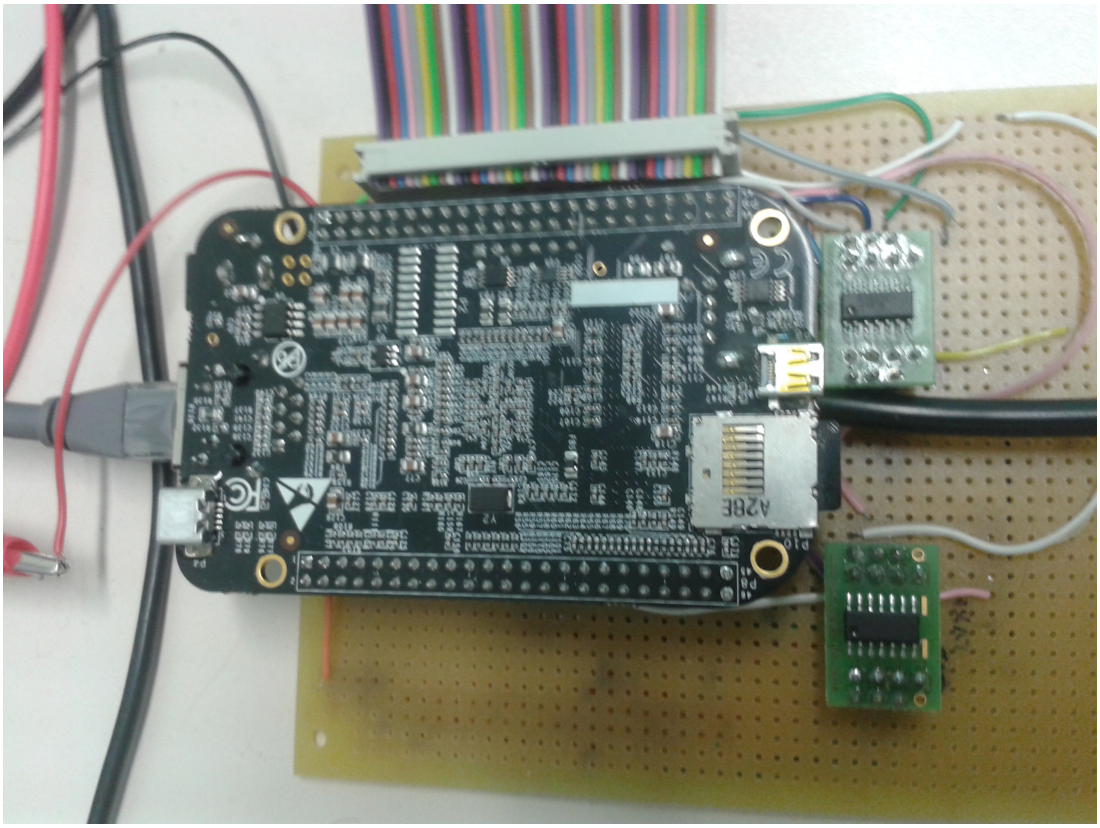
- $\overline{RESETB}$ BBB: P8-Pin 23 (GPIO1_4)→ 74LVX14/2: Pin 3 (I_1) - Pin 4 (O_1) → DRV8312 Board: J5-Pin 9 (RESET_B)

- $\overline{RESETC}$ BBB: P8-Pin 24 (GPIO1_1)→ 74LVX14/2: Pin 5 (I_1) - Pin 6 (O_2) → DRV8312 Board: J5-Pin 12 (RESET_C)

Notice that the LVX IC has inverting gates, which should be taken into account when processing the Hall sensor's signals and the reset action.

Figure 11.2 shows the finished version of the circuit. All elements are mounted on a perforated prototyping board, being a temporary solution rather than a permanent one. The connection between the two boards is made though a 40-pin header and flat wire. The 5V power supply is connected through wire, in the picture red for 5V and black for GND. Other connections are the UART and Ethernet, which will have more impact in next sections.

Figure 11.2.: Final result of circuit and connection



11.2. VxWorks IDE

The WindRiver Workbench is going to be used as the main environment for the code development for the motor control, therefore it should be connected to the processor board. There will be two connections between the PC and the BBB, one being a serial communication that access the BBB through the Terminal in the VxWorks Workbench and the Ethernet connection that allows the board to be programmed.

The serial connection is using a USB to TTL serial cable like the one in figure 11.3. The BBB's voltage level for TX and RX is 3.3V, therefore, when selecting a serial cable, the voltage level should be taken into account so that the BBB won't break down. For this purpose, the chosen cable is the TTL-232R-3V3 by FTDI Chip connected to the BBB's serial header J1 and an USB port.

Figure 11.3.: USB to TTL serial cable [15]

In Linux systems, the workbench can be accessed through the Terminal window by writing the file's path in the command line.

```
/opt/vxworks/vxworks-6.9.3.3_0/startWorkbench.sh
```

The 'Settings' button is located at the right side of the the 'Terminal' tab in the Workbench. In the 'Settings' window, parameters like 'Port' or 'Baud Rate', that serial connections need, can be introduced. USB ports in Linux systems are usually located inside '/dev' and the exact channel can be found using the command 'dmesg'. This command prints out all the hardware connected to the PC so it would be better to use a filter command like 'grep' to reduce the list. The output of the command:

```
dmesg | grep FTDI
```

gives this list:

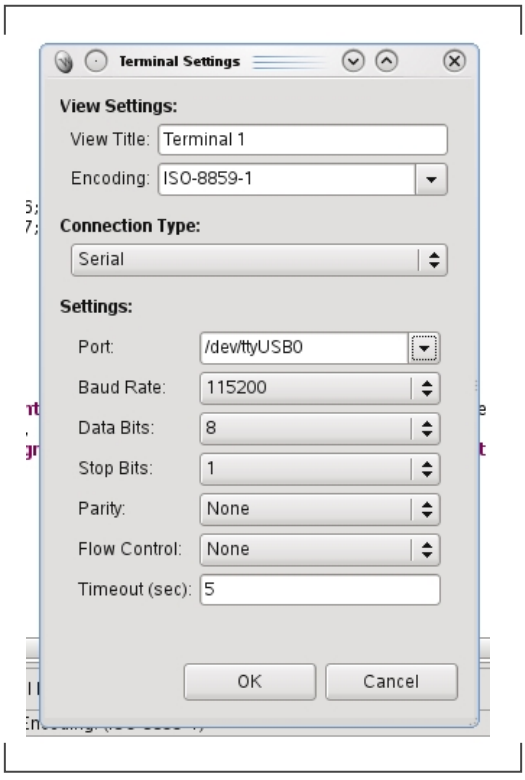
```
[4.264791] usb 4-1: Manufacturer: FTDI
[8.514807] USB Serial support registered for FTDI USB Serial
        Device
[8.514945] ftdi_sio 4-1:1.0: FTDI USB Serial Device converter
        detected
[8.517601] usb 4-1: FTDI USB Serial Device converter now
        attached to ttyUSB0
[8.517797] ftdi_sio: v1.6.0:USB FTDI Serial Converters Driver
```

The fourth line indicates the port in which the FTDI cable is connected. The USB port for this connection will then be:

```
/dev/ttyUSB0
```

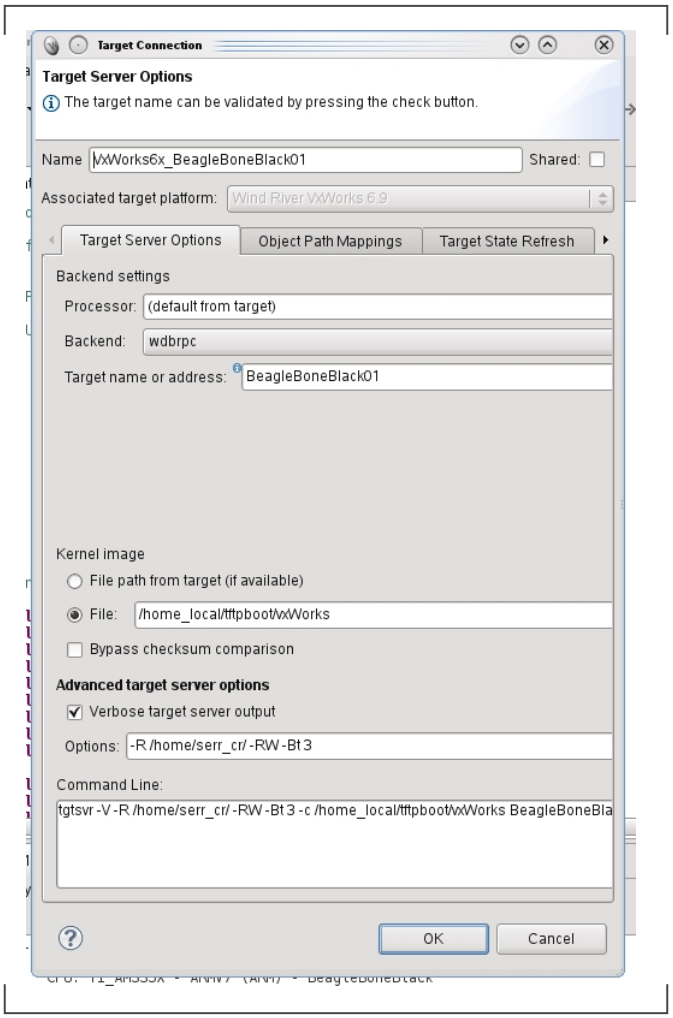
Other paramenters are shown in the following figure (figure 11.4)

Figure 11.4.: Open-loop system overview



The ‘target’ connection is established in the ‘Remote Systems’ tab, usually located at the bottom left of the Workbench. Clicking the leftmost button in this tab, creates a new connection to a remote system. The first ‘pop-up’ window will show the ‘Remote System type’, in this case ‘Wind River VxWorks 6.x Target Server Connection’. Clicking ‘Next’ leads to the ‘Target Server options’ window, where the connection parameters will be written in (figure 11.5).

Figure 11.5.: Target Server options



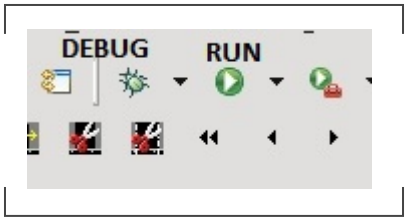
Once all the parameters have been written, clicking on the green ‘Connect’ buttons in the ‘Remote Systems’ and in the ‘Terminal’ tab will automatically connect the PC to the board. Once the board is connected, it can be disconnected from the ‘Terminal’ by pressing the red "Disconnect" buttons on the ‘Terminal’ tab or the ‘Remote Systems’.

When accesing the BBB though the 'Terminal', clicking 'Ctrl+x' with the window active restarts the BBB.

New projects are created by going to 'File → New', or right-clicking in the 'Project Explorer' window 'New', then selecting 'VxWorks Downloadable Kernel Module' Project. VxWorks kernel applications execute in the same mode and memory space as the kernel itself and can either be interactively downloaded and run on a VxWorks target system, or linked with the operating system image. Creating a new project is really straightforward until the 'Build Specs' window, where the only option selected is the 'ARMARCH7gnu' box.

The 'Debug' and 'Run' buttons (figure 11.6) will load the code into the microprocessor. The 'Debug' button does this automatically after selecting an 'Entry point', while to 'Run' it, it must be downloaded first into the kernel by right-click on the project's name in the 'Project Explorer' window and selecting 'Download Kernel Module'.

Figure 11.6.: Debug and Run buttons



Matlab/Simulink projects can be loaded and unloaded into the kernel, but this will be explained when talking about the 'Closed-loop control'.

11.3. Matlab

The speed control loop for the BLDC is going to be made using the Matlab/Simulink environment, loading the blocks into the kernel. As with the VxWorks Workbench, the program is initialised, in Linux systems, by writing its path into the 'Terminal's' command line. The path could be different to the one here written.

```
/opt/vxworks/vxworks-6.9.3.matlab/2010b/bin/matlab_acad
```

Interconnecting Matlab/Simulink and VxWorks is not straightforward and it calls for the writing of new code in order to allow Matlab to run in other environments. The process

of adapting software so that an executable program can be created for a computing environment that is different from the one for which it was originally designed is called porting.

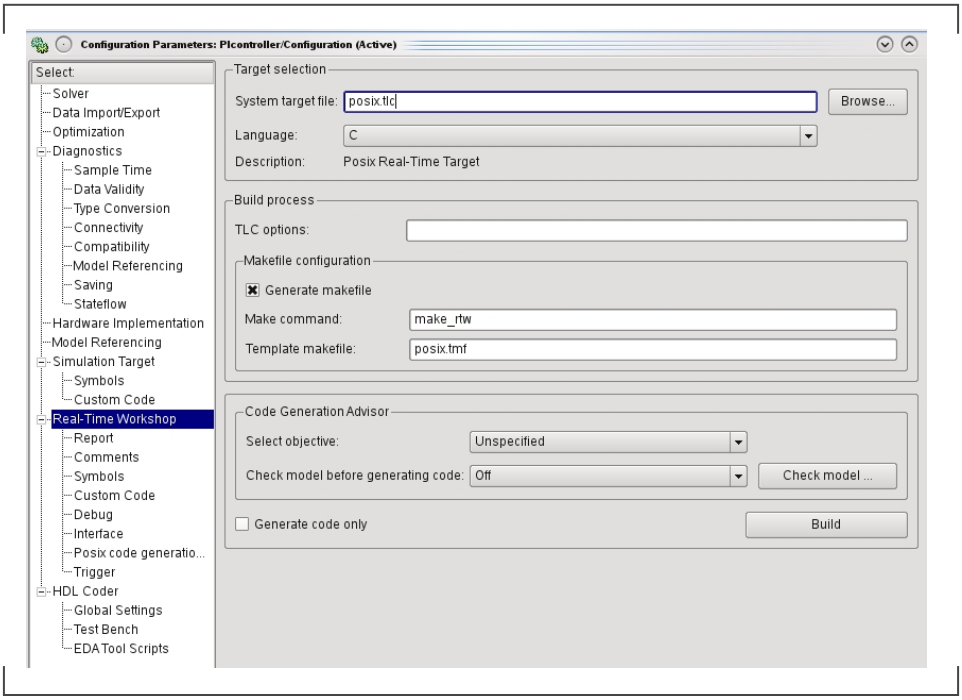
The port used in this project was developed by colleagues at the 'German Aerospace Center' and given to use as a tool. These instructions here stated are only valid for this port and there is no guarantee that they will work in other cases.

First of all, the parameters or the POSIX targets should be loaded into the Matlab Workspace by writing the following command in the Matlab console. If they are not loaded, the port won't work properly.

```
run targets/init.m
```

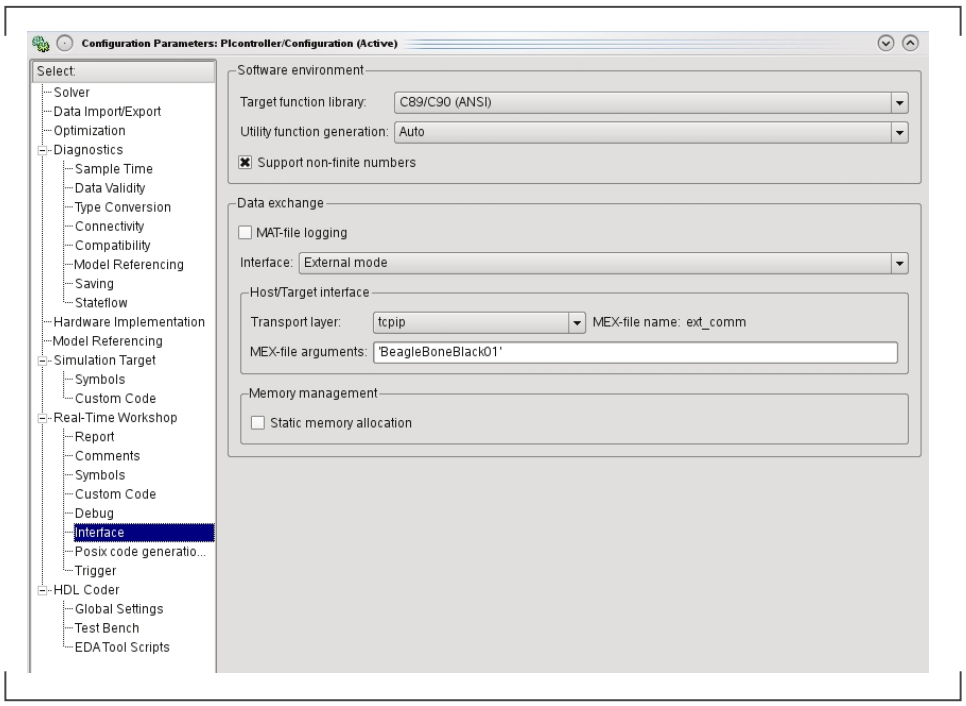
The following steps are done in the Simulink environment. In the project created, the 'Real Time' configuration is done 'Simulation → Configuration Parameters'. On the left side of the 'Configuration Parameters' window there are several options for the simulation configuration. First, in the 'Real Time Workshop' panel (figure 11.7), it is mandatory to select the 'System Target File', in this case 'posix.tlc'

Figure 11.7.: Matlab Simulation Configuration



Next step is the configuration of the 'Interface' parameters for the 'External Mode', which will allow the connection between the Matlab environment and the VxWorks and BBB (figure 11.8). The 'Interface' box must be switched to 'External', the 'Transport layer' is the type of connection 'TCP/IP' (tcpip), and the 'Mex-file arguments' correspond to the name of the target, 'BeagleBoneBlack01'

Figure 11.8.: Matlab Simulation 'Interface' Configuration



In the 'Posix code generation' menu there is a 'drop-down' menu where the 'Target toolchain' is selected. Depending on the installed toolchains there are several different possibilities: the toolchain used in this project is called 'vxworks6.9-armv7-gcc4.x-kernel' and was developed at the DLR.

Finally, building the project is done by going to 'Tools → Real Time Workshop → Build'. The resulting file with the name of the project will be loaded into the BBB via the VxWorks Workbench.

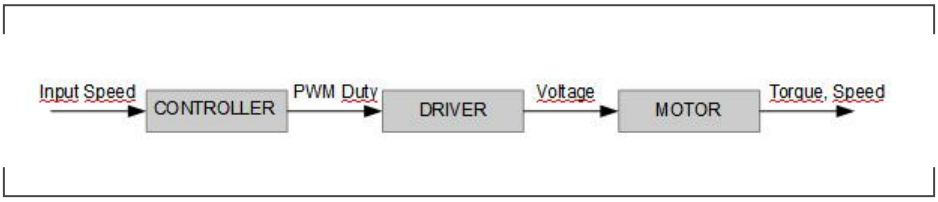
12. Motor Control

12.1. Open-loop Control

The first attempt to develop a motor control will be by creating an open-loop, or non-feedback, control for the BLDC. Open-loop control is a type of control that computes its input into a system using only the current state and its model of the system. It does not observe the output of the system, making it incapable of correcting any errors that could appear during operation. This control is preferred over closed-loop control when simplicity and low-cost are recommended and feedback is not important.

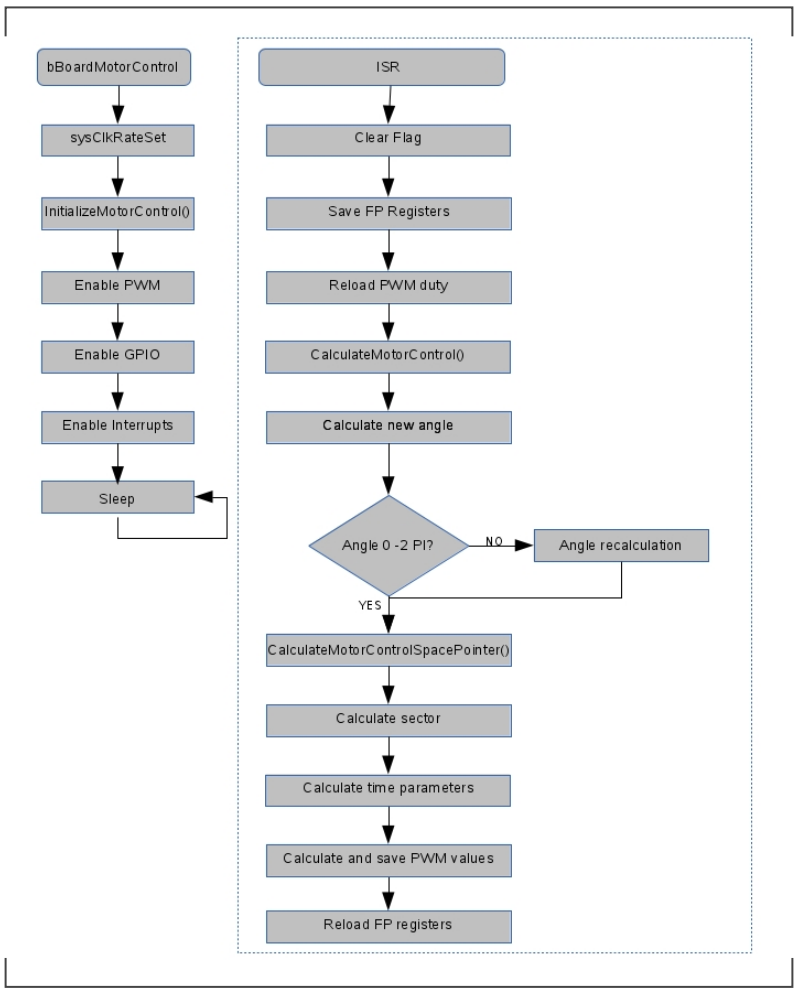
In open-loop control the input is given to the model if the system, whose output is driven into the actuator. In this method, there exists an intermediate step between the model system and the actuator (the BLDC motor): the driver mentioned in the previous chapter of this document (DRV8312). The loop will be as follows (figure 12.1) the input(speed) will be a variable in the system (the code); the output of the code will be the PWM signal which will be driven into the DRV chip. The driver "transforms" the PWM signal into the voltage needed to power up the motor. The BLDC motor has a speed and a torque which, in this case, won't be measured.

Figure 12.1.: Open-loop system overview



The "Controller" is the only part of the loop that will be done from scratch, and corresponds to the code written for the ARM microprocessor. The PWM signal, which will be the output, is going to be generated through an algorithm. PWM will also trigger an ISR at a specified time; necessary to calculate the new PWM duties.

Figure 12.2.: Open-loop flow diagram



The chosen algorithm is Space Vector Modulation, which is mainly used for creating AC waveforms. SVM calls for the implementation of a rotating voltage vector, whose angle value is incremented at each time an ISR is triggered.

Since the Beaglebone Black's pin outputs are highly multiplexed and more than one feature can be used for each pin, it is mandatory to chose the operation mode from the list included in the BBB System Reference Manual (SRM) [17] to correctly determine the outputs. The pinmuxing table is taken out from the 'Sitara AM335x ARM Cortex-A8 Microprocessors (MPUs)' [18] datasheet and pins selected in the SRM should correspond to one number (in the ZCZ package schematic) in the MPU datasheet. Another useful document to configure will be the 'AM335x ARM Cortex-A8 Microprocessors (MPUs)

Technical Reference Manual' (TRM) [19], which contains all the main information about the microprocessor, like modules and registers, as well as configuration examples.

12.1.1. Function Main

The flow diagram in figure 12.2, presents the steps that the software should complete to create a correct PWM waveform. The left branch contains the functions present on the main routine, called "bBoardMotorControl", and are called only one time at the beginning of the running process. These functions are the configuration functions for PWM (PWMConfiguration()) and GPIO (openGPIOConfig()), and the initial parameters for the SVM and the motor control in general (InitializeMotorControl()).

The configuration and initialisation function's content will be explained in the next sections. However, in the main function, a message will be printed in the Terminal screen indicating whether the configuration of the PWM and GPIO was successful or not. One of the main causes of the configuration functions failing is that the BBB was not shut down correctly and the input parameters are not written correctly. To ensure the configuration, the BBB should be shut down every time before running the code.

```
//PWM Configuration
if (PWMConfiguration() == ERROR){
        printf("PWM Configuration: Failed. \n");
}
else{
        printf("PWM Configuration: OK. \n");
}

//GPIO Configuration
if (openGPIOConfig() == ERROR){
        printf("GPIO Configuration: Failed. \n");
}
else{
        printf("GPIO Configuration: OK. \n");
}
```

At the end of the "bBoardMotorControl" function, a infinite while loop prevents the program from shutting down and allows running for an undefined amount of time. For the time the processor is doing nothing for example, waiting for a new ISR, the "sleep" function is used so that the program is not consuming resources.

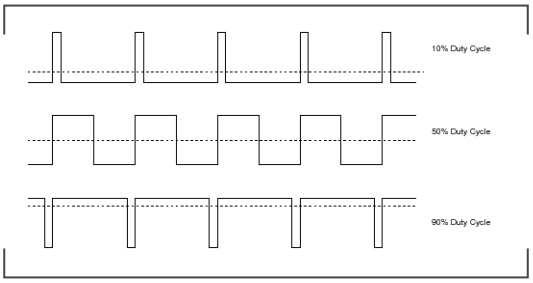
```
while(1){
        sleep(1);
}
```

12.1.2. Pulse-Width Modulation (PWM)

Pulse-Width Modulation is a modulation technique in which the duration of each pulse (a square wave) is set by equation (7.1). The duty cycle (D) is the percentage between the active time of the signal (T) and the period of the signal (P) (figure 12.3). The main advantage for PWM signals is that power loss in the switching devices is very low as, when a switch is off, there is practically no current, and when it is on, there is almost no voltage drop across the switch.

$$D = \frac{T}{P} \tag{12.1}$$

Figure 12.3.: PWM signal with different duty cycles

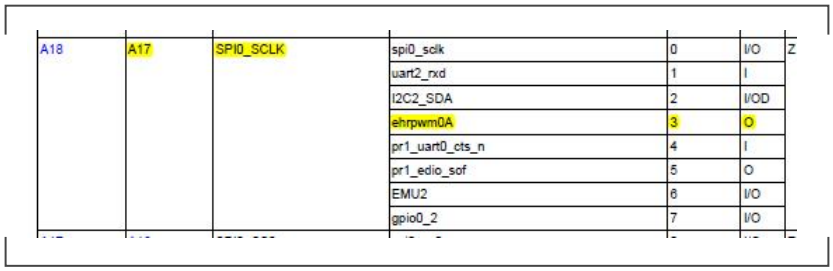


The AM3359 microprocessor has three ePWM Modules, each of which has up to two separate outputs, all of them connected to the output pins. The first step will be to configure the pin multiplexing for the ePWM operation mode. The ePWM outputs chosen for the board interconnection in 'Section 6.1' will be used in the code, and their corresponding ZCZ number in the datasheet are:

- eHRPWM0A (UART2_RXD) → A17 (SPI0_SCLK)
- eHRPWM0B (UART2_TXD) → B17 (SPI0_D0)
- eHRPWM2A (EHRPWM2A) → U10 (GPMC_AD8)

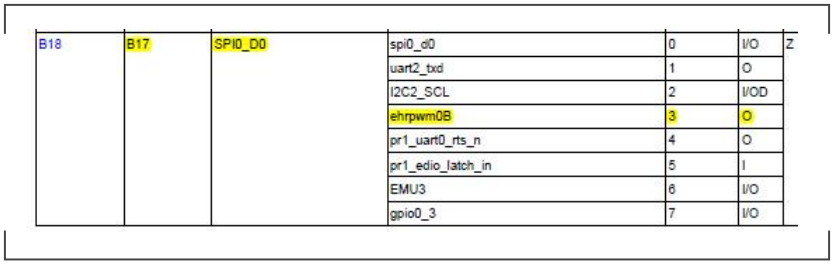
The following figures (12.5, 12.5 and 12.6) indicate the operation mode for each PWM output. The table shows the 'Pin number' for ZCE and ZCZ packages (currently using the ZCZ package), the signal name (which may or may not be the same as the operation mode that is being used), the number for each mode and the type of pin: Input (I), Output (O) or Input/Output (I/O). There is several other information, but it is not relevant for this purpose. After the operation mode for each ePWM output has been found, code should be written to indicate the processor which output to use.

Figure 12.4.: ePWM0A Mode [18]



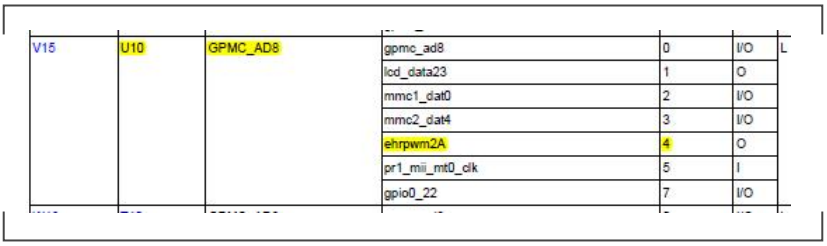
A18	A17	SPI0_SCLK	spi0_sclk	0	I/O	Z
			uart2_rxd	1	I	
			I2C2_SDA	2	I/OD	
			ehrpwm0A	3	O	
			pr1_uart0_cts_n	4	I	
			pr1_edio_sof	5	O	
			EMU2	6	I/O	
			gpio0_2	7	I/O	

Figure 12.5.: ePWM0B Mode [18]



B18	B17	SPI0_D0	spi0_d0	0	I/O	Z
			uart2_txd	1	O	
			I2C2_SCL	2	I/OD	
			ehrpwm0B	3	O	
			pr1_uart0_rts_n	4	O	
			pr1_edio_latch_in	5	I	
			EMU3	6	I/O	
			gpio0_3	7	I/O	

Figure 12.6.: ePWM2A Mode [18]



V15	U10	GPMC_AD8	gpmc_ad8	0	I/O	L
			lcd_data23	1	O	
			mmc1_dat0	2	I/O	
			mmc2_dat4	3	I/O	
			ehrpwm2A	4	O	
			pr1_mii_mt0_clk	5	I	
			gpio0_22	7	I/O	

The code snippet below shows a way of resolving this using a struct in C. The I/O multiplexing is interfaced by the 'Control' module (CTRL) (base address 0x44E1 0000) and is configured by adding the 'offset' given on the TRM's register specifications at the end of the CTRL base address. The first variable indicates the register's offset address for each output pin, the second one interfaces the operation mode from '0' to '7' and other features. ePWM pins will be used only as an output, therefore the 'Input enable

value'(Receiver) should be disabled (IDIS is a define containing 0). If the pins are used as input, or input/output, the Receiver should be enabled by writing '1' in the register.

```
struct am335xPadConf pwm_pad [] = {
        {CONTROL_PADCONF_GPMC_AD8, (IDIS | MODE4 )},/* PWM A */
        {CONTROL_PADCONF_SPI0_SCLK, (IDIS | MODE3 )},/* PWM B */
        {CONTROL_PADCONF_SPI0_D0, (IDIS | MODE3 )}, /* PWM C */
        {AM335X_PAD_END, AM335X_PAD_END }};
```

After the pin multiplexing is done, the proper ePWM configuration can be started. It will be done in a function called 'PWMConfiguration()' and called from the main function, as seen previously. This function returns an 'int' parameter with the status of the configuration (OK if the configuration was successful or ERROR if not). The ePWM modules used will be 'ePWM0' (EPWMSS0) and 'ePWM2' (EPWMSS2), as the selected pins were ePWM0A, ePWM0B and ePWM2A. The whole module should be configured even though only one output is used for ePWM2. It is important to notice that each PWM subsystem has its own register address where the configuration parameters are written in. The exact address can be checked in the TRM. For the AM335x, ePWM0 has its base address at (0x4830 0200) and ePWM2 at (0x4830 4200).

The ARM microprocessor needs to configure the ePWM Clock in order to run the ePWM. The 'Clock Management' protocol for the microprocessor is stated in the TRM , on the 'Power, Reset, and Clock Management (PRCM)'.

At the beginning of the function, the parameter that will be returned is initially declared as 'OK' and the pin multiplexing configuration is done by calling a configuring function using the previous struct as an argument.

The elements that will be configured on the first step are 'Module Mode' and 'Idle'. 'Module mode' (MODULEMODE) option controls whether the interface clock is enabled, while 'Idle'(IDLEST) controls if the module is in 'Idle' mode or performing another action. The MODULEMODE parameter will be set to 'Enable' and the IDLEST to 'Func', both being enabled and operational. A function written specially for ARM microprocessors will write the options into to registers and return the status of the action (OK or ERROR). The 'Shift' parameter will displace the option to match the register's bit corresponding to IDLEST.

```
/* Writing to MODULEMODE field of CLKCTRL register. */
SetAndChkRegister(SOC_CM_PER_REGS+CM_PER_EPWMSS0_CLKCTRL,
                  CM_PER_EPWMSS0_CLKCTRL_MODULEMODE,
                  CM_PER_EPWMSS0_CLKCTRL_MODULEMODE_ENABLE );
```

```
SetAndChkRegister(SOC_CM_PER_REGS+CM_PER_EPWMSS2_CLKCTRL,
                  CM_PER_EPWMSS2_CLKCTRL_MODULEMODE,
                  CM_PER_EPWMSS2_CLKCTRL_MODULEMODE_ENABLE );


/* Check writing to IDLEST field in CLKCTRL register. */
ChkRegister( SOC_CM_PER_REGS + CM_PER_EPWMSS0_CLKCTRL,
                  CM_PER_EPWMSS0_CLKCTRL_IDLEST,
                  CM_PER_EPWMSS0_CLKCTRL_IDLEST_FUNC <<
                  CM_PER_EPWMSS0_CLKCTRL_IDLEST_SHIFT );
ChkRegister( SOC_CM_PER_REGS + CM_PER_EPWMSS2_CLKCTRL,
                  CM_PER_EPWMSS2_CLKCTRL_IDLEST,
                  CM_PER_EPWMSS2_CLKCTRL_IDLEST_FUNC <<
                  CM_PER_EPWMSS2_CLKCTRL_IDLEST_SHIFT );
```

Next, the clock and the 'Time Base Module Clock' should be enable in the 'Control Module' register. The control module includes status and control logic not addressed within the peripherals or the rest of the device infrastructure. The ePWM clock is enabled in the ePWM Sub-System's registers.

```
/* Enable Clock for EHRPWM in PWM sub system */
EHRPWMClockEnable(SOC_PWMSS0_REGS);
EHRPWMClockEnable(SOC_PWMSS2_REGS);

/* Enable Timer Base Module Clock in control module */
status |= SetAndChkRegister(SOC_CONTROL_REGS+CONTROL_PWMSS_CTRL,
                  CONTROL_PWMSS_CTRL_PWMSS0_TBCLKEN,
                  CONTROL_PWMSS_CTRL_PWMSS0_TBCLKEN);
status |= SetAndChkRegister(SOC_CONTROL_REGS+CONTROL_PWMSS_CTRL,
                  CONTROL_PWMSS_CTRL_PWMSS2_TBCLKEN,
                  CONTROL_PWMSS_CTRL_PWMSS2_TBCLKEN);
```

After configuring the clocks, the proper configuration of the PWM module starts. The ePWM module has seven different submodules: Time-base (TB), Counter-Compate (CC), Action-Qualifier (AQ), Dead-band (DB), ePWM-chopper (PC), Event-trigger (ET) and Trip-zone (TZ). The most important submodules are the first three: TB, CC and AQ; but the ET will also be configured to create a ePWM interrupt.

The 'Time Base Clock' submodule is in charge of the major features of every ePWM

signal: time base, period, operation mode (up, down or up-down). The first step will be configurating the 'Time-Base Clock, the minimum time step value, for each ePWM module, at a frequency of 100MHz (10 ηs), with a prescaler(CLKDIV) value of 1 ($PWMClockFrequency = 100000000, CLOCK\_DIV\_VAL = 1$) in the ePWM registers. The registers associated with the 'Time base' period value are 'Time-Base Clock' (TBCLK) for the period, and 'Time-Base Control'(TBCTL) to fix prescaler value. Texas Instruments provides several pre-made functions that automatically write the desired values into the correct registers that make the configuration process easier.

```
EHRPWMTimebaseClkConfig( SOC_EPWM_0_REGS,
                         PWMClockFrequency/CLOCK_DIV_VAL,
                         PWMClockFrequency  );

EHRPWMTimebaseClkConfig( SOC_EPWM_2_REGS,
                         PWMClockFrequency/CLOCK_DIV_VAL,
                         PWMClockFrequency  );
```

According to the DRV8312 datasheet, the PWM period signal should not have a frequency lower than 10KHz or higher than 500KHz to ensure a correct functioning. A frequency of 40KHz (25 μs) will be selected as the ePWM cycle frequency ($PWMCycleFrequency = 40000$) and written inside the register 'Time-Base Period' (TBPRD). The period value in the TBPRD register is not directly set, but rather it is written as 'n' times the Time-Base Clock. The value written inside the register is, therefore:

$$TBPRD = \frac{PWMClockFrequency}{PWMCycleFrequency} = \frac{10x10^6}{40x10^3} = 2500 \tag{12.2}$$

The time-base counter has three modes of operation selected by the time-base control register (TBCTL) (figure 12.7):

- Up-Down-Count Mode: In up-down-count mode, the time-base counter starts from zero and increments until the period (TBPRD) value is reached. When the period value is reached, the timebase counter then decrements until it reaches zero.

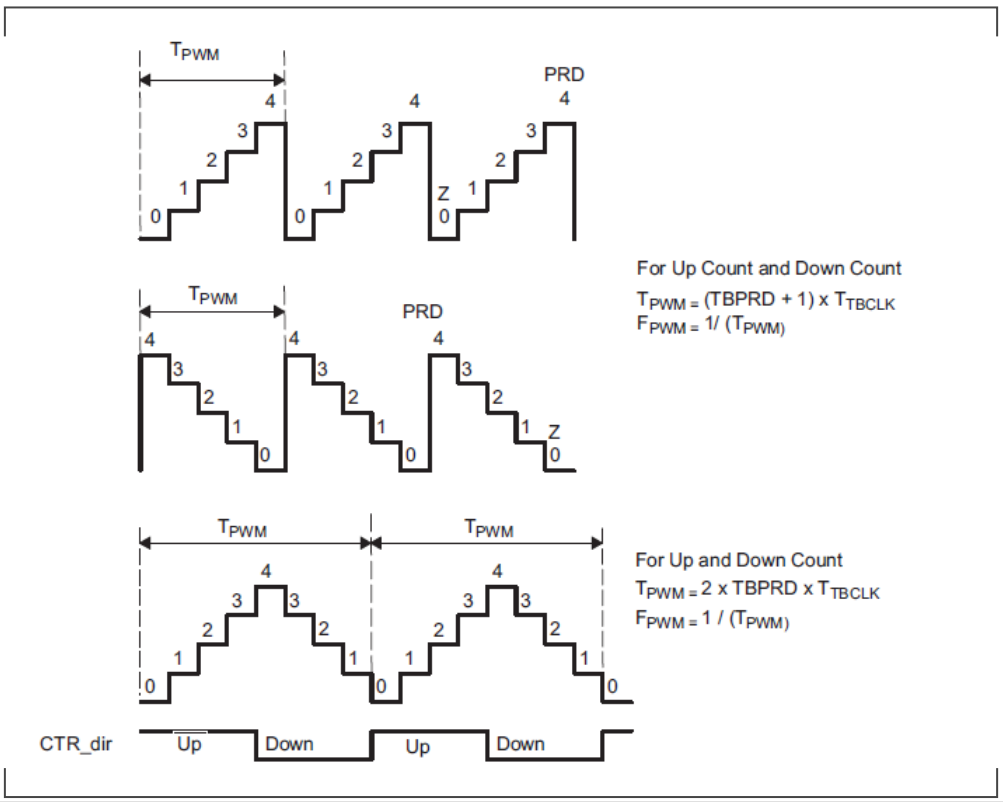
- Up-Count Mode: In this mode, time-base counter starts from zero and increments until it reaches the value in the period register (TBPRD).

- Down-Count Mode: In down-count mode, the time-base counter starts from the period (TBPRD) value and decrements until it reaches zero.

The chosen solution for SVM will be creating a symmetrical PWM waveform, making 'Up-Down Count Mode' the best operation mode among the existing three. The PWM values that are going to be calculated represent the CC value, which will be explained afterwards. To enable 'Up-Down Count Mode' the field inside the 'Counter Mode' (CTRMODE) in the TBCTL registers should be loaded with the value '2h'.

The time-base period register (TBPRD) has a shadow register that allows the register update to be synchronized with the hardware. This feature won't be used in this project, the TBPRD register will be loaded without a shadow register, therefore it should be disabled by writing the value '1h' into 'Active Period Register Load From Shadow Register Select'(PRDLD) in the TBCTL register.

Figure 12.7.: PWM Periods based on Count Mode [19]




```
EHRPWMPWMOpFreqSet( SOC_EPWM_0_REGS,
                    PWMClockFrequency/CLOCK_DIV_VAL,
                    PWMCycleFrequency,
                    EHRPWM_COUNT_UP_DOWN,
                    EHRPWM_SHADOW_WRITE_DISABLE );
```

```
EHRPWMPWMOpFreqSet( SOC_EPWM_2_REGS,
                    PWMClockFrequency/CLOCK_DIV_VAL,
                    PWMCycleFrequency,
                    EHRPWM_COUNT_UP_DOWN,
                    EHRPWM_SHADOW_WRITE_DISABLE );
```

Counter-compare submodule takes as input the time-base counter value. The time-base value is continuously compared to the counter-compare A (CMPA) and counter-compare B (CMPB) registers. When the timebase counter is equal to one of the compare registers, the counter-compare unit generates an appropriate event programmed by the AQ module. When appropriately configured in the AQ module, the CC controls the PWM duty cycle.

The expected CC value should be loaded into the 'Counter-Compare A Register'(CMPA) or 'Counter-Compare B Register'(CMPB) as a value 'n' times the TBCLK. The value can be reloaded as many times as needed, and so it will be done at each PWM cycle. In the configuration function, the value loaded will be '0' to ensure that the duty cycle is 0. The CC configuration and value loading should be done for all the ePWM outputs involved, though in the code below only ePWM0A and ePWM0B are stated.

```
/* Load Compare A value and B value */
EHRPWMLoadCMPA( SOC_EPWM_0_REGS,
                0,
                EHRPWM_SHADOW_WRITE_DISABLE,
                EHRPWM_COMPA_NO_LOAD,
                EHRPWM_CMPCTL_OVERWR_SH_FL );
EHRPWMLoadCMPB( SOC_EPWM_0_REGS,
                0,
                EHRPWM_SHADOW_WRITE_DISABLE,
                EHRPWM_COMPB_NO_LOAD,
                EHRPWM_CMPCTL_OVERWR_SH_FL );
```

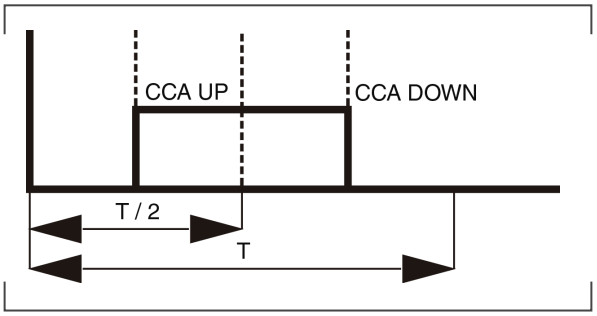
The action-qualifier submodule is responsible for qualifying and generating actions (set, clear, toggle) based on the following events:

- CTR = PRD: Time-base counter equal to the period
- CTR = 0: Time-base counter equal to zero
- CTR = CMPA: Time-base counter equal to the counter-compare A register

- CTR = CMPB: Time-base counter equal to the counter-compare B register

A symmetric PWM waveform can be obtained loading a CC value and configuring the AQ module as shown in figure 12.8. However, due to the inverting effect of the LVX IC, the actual configuration written into the register 'Action-Qualifier Control Register for Output A Section (EPWMxA)' (AQCTLA) or its 'B' counterpart, will be the exact opposite of said figure. When a new cycle begins (CTR = 0), the signal is high until it reaches CTR = CMPAU (Up), when it is turned low. The signal continues to be low until CTR = CMPAD (Down), when is turned high again until the end of the cycle, CTR = PRD. When configuring the AQ for the output 'B' it is good to use the register CMPB and configure the switchings for CMPBU and CMPBD.

Figure 12.8.: PWM Counter Compare AQ configuration example




```
/* Configure Action Qualifier */
EHRPWMConfigureAQActionOnA( SOC_EPWM_0_REGS,
                        EHRPWM_AQCTLA_ZRO_EPWMXAHIGH,
                        EHRPWM_AQCTLA_PRD_EPWMXALOW,
                        EHRPWM_AQCTLA_CAU_EPWMXALOW,
                        EHRPWM_AQCTLA_CAD_EPWMXAHIGH,
                        EHRPWM_AQCTLA_CBU_DONOTHING,
                        EHRPWM_AQCTLB_CBD_DONOTHING,
                        EHRPWM_AQSFRC_ACTSFB_DONOTHING  );

EHRPWMConfigureAQActionOnB( SOC_EPWM_0_REGS,
                        EHRPWM_AQCTLB_ZRO_EPWMXBHIGH,
                        EHRPWM_AQCTLB_PRD_EPWMXBLOW,
                        EHRPWM_AQCTLB_CAU_DONOTHING,
                        EHRPWM_AQCTLB_CAD_DONOTHING,
                        EHRPWM_AQCTLB_CBU_EPWMXBLOW,
                        EHRPWM_AQCTLB_CBD_EPWMXBHIGH  ,
                        EHRPWM_AQSFRC_ACTSFB_DONOTHING  );
```

The Event-Trigger sub-module main features are:

- Receives event inputs generated by the time-base and counter-compare submodules
- Uses the time-base direction information for up/down event qualification
- Uses prescaling logic to issue interrupt requests at:

  Every event

  Every second event

  Every third event

The Event Trigger will generate an interrupt every cycle (25 μs), in which the new PWM duty cycles will be calculated and their value reloaded into the CMPx registers. The interrupt will only be generated in one on the ePWM channels (ePWM0) at the beginning of every event ($TBCNT = 0$). The field 'ePWM Interrupt (EPWMx_INT) Period Select'(INTPRD) in the 'Event-Trigger Pre-Scale Register' (ETPS) register, determines how many events need to occur until an event is generated ('1h' to generate interrupts in the first event). The 'Event-Trigger Selection Register' (ETSEL) determines when the interrupt is generated by writing a value in the 'ePWM Interrupt (EPWMx_INT) Selection Options'(INTSEL) field ('1h' to generate an interrupt at $TBCNT = 0$).

```
/* Generate interrupt every occurrence of the event */
EHRPWMETIntPrescale( SOC_EPWM_0_REGS,
                     EHRPWM_ETPS_INTPRD_FIRSTEVENT);

/* Generate event when TBCTR = 0 */
EHRPWMETIntSourceSelect( SOC_EPWM_0_REGS,
                     EHRPWM_ETSEL_INTSEL_TBCTREQUZERO );
```

The remaining modules Dead-band (DB), ePWM-chopper (PC), Trip-zone (TZ) and the High-Resolution PWM capabilities (HRPWM) will not be configured.

The last step is connect the interrupt event to the function (ISR) where the PWM duty cycles will be calculated. VxWorks has its own functions that allow interrupt connection. First, the IRQ number (AM335X_ePWM0INT = 86) should be known, so that the event triggering the ISR is the correct one. The function 'intConnect' connects the ISR to the IRQ number, while the function 'intEnable' allows the interrupt generation. An optional

priority value, that determines whether the function will be executed before or after another function, can be selected using the 'IntPrioritySet' function.

```
/* Connect interrupt to ISR and set priority */
intConnect(AM335X_ePWM0INT,(VOIDFUNCPTR)InterruptEHRPWM25usec,0)
intEnable(AM335X_ePWM0INT)
IntPrioritySet(AM335X_ePWM0INT, 16, AINTC_HOSTINT_ROUTE_IRQ)
```

If the configuration was successful, the function returns the variable 'status' as 'OK' and returns 'ERROR' if there was any problem on the configuration. A message indicating whether or not the configuration was successful will be printed in the 'Terminal' window of the Workbench.

Each 25 μs, the interrupt is generated, entering into the ISR (InterruptEHRPWM25usec). After clearing the interrupt flag, the ISR will reload the ePWM with the values calculated in the previous iteration. The value is loaded into the register by writing the CMPx register for each PWM address.

```
/* Reload PWM duty via Counter-Compare */
HWREGH( SOC_EPWM_0_REGS + EHRPWM_CMPA ) =
        AllPWMChannels.PWMA[1] & EHRPWM_CMPA_CMPA;
HWREGH( SOC_EPWM_0_REGS + EHRPWM_CMPB ) =
        AllPWMChannels.PWMB[1] & EHRPWM_CMPB_CMPB;
HWREGH( SOC_EPWM_2_REGS + EHRPWM_CMPA ) =
        AllPWMChannels.PWMA[2] & EHRPWM_CMPA_CMPA;
```

After the duty is reloaded, there is a call to the function 'CalculateMotorControl(& AllPWMChannels)', which will calculate the PWM new duty cycle. Once the duty cycle is calculated, the program exits the ISR and waits for a new interrupt.

12.1.3. General Purpose Input/Output (GPIO)

The 'General Purpose Input/Ouput' (GPIO) are generic pins on an integrated circuit whose behaviour, including whether it is an input or output pin, can be controlled by the user. GPIO pins can be used for the following applications.

- Data input (capture)/output (drive)
- Keyboard interface with a debounce cell

- Interrupt generation in active mode upon the detection of external events.
- Wake-up request generation (in Idle mode) upon the detection of signal transition(s)

GPIO pins will be used to drive the '$\overline{RESETx}$' status, active or not, to the half-bridges in the DRV8312 IC, allowing an independent control of each half-bridge. In the open-loop control they will be maintained on 'high' at all times (not resetting). Due to its inverting behaviour, driving out 'low' level signals will reset the half-bridge by forcing it into a high-impedance state.

There exist four GPIO modules (GPIO0-3) with 32 dedicated input/output pins each making a total of 128 input/output pins, though multiplexed with other options. As explained in the PWM section, the high level of pin multiplexing calls for the necessity of finding the operation mode in the BBB's and the microprocessor's datasheets. The GPIO module used for the three reset lines will be GPIO1 and the configuration process below explained will be for this module's operation.

- $\overline{RESETA}$ (GPIO1_5) → V8
- $\overline{RESETB}$ (GPIO1_4)→ U8
- $\overline{RESETC}$ (GPIO1_1)→ V7

Figure 12.9.: GPIO1 5 Mode [18]

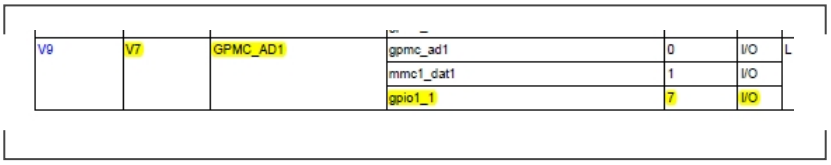
W14	V8	GPMC_AD5	gpmc_ad5	0	I/O	L
			mmc1_dat5	1	I/O	
			gpio1_5	7	I/O	

Figure 12.10.: GPIO1 4 Mode [18]

V13	U8	GPMC_AD4	gpmc_ad4	0	I/O	L
			mmc1_dat4	1	I/O	
			gpio1_4	7	I/O	

The reset lines for the DRV8312 will be configured as data outputs in the GPIO pins. This lines will drive The pinmuxing configuration is done exactly as it was done with the ePWM module, changing the register's addresses and the operation mode number to fit the tables, taking into account that GPIO mode is usually MODE 7.

Figure 12.11.: GPIO1 1 Mode [18]



V9	V7	GPMC_AD1	gpmc_ad1	0	I/O	L
			mmc1_dat1	1	I/O	
			gpio1_1	7	I/O	

```
struct am335xPadConf reset_line_pads [] = {
        /* GPIO1[5] RESET A */
    {CONTROL_PADCONF_GPMC_AD5, (IDIS | MODE7)},
    /* GPIO1[4] RESET B */
    {CONTROL_PADCONF_GPMC_AD4, (IDIS | MODE7)},
    /* GPIO1[1] RESET C */
    {CONTROL_PADCONF_GPMC_AD1, (IDIS | MODE7)},

{ AM335X_PAD_END, AM335X_PAD_END }};
```

The configuration of GPIO pins is different from the ePWM's. TI offers a set of functions for this purpose, that makes the process easier as it involves less register configuration and is a more straightforward method than the previous one.

The GPIO pins should be allocated for an specific function in order to be able to run as input or output. The function for allocation has two input parameters, one to select the output pin and a parameter that allows to write the usage of the pin inside a 'string'.

The GPIO pin number should be submitted not by indicating the pin number only but also the module the pin is in, as there is not a way of distinguish the GPIO module. Each GPIO has 32 pins, so GPIO-0 pins will be identified with a number between '0' and '31; GPIO-1, a number between '32' and '63, and so on. In this project, GPIO pins will be expressed this way:

- am335xResetA = 32 + 5 /* gpio 1 pin 5 */
- am335xResetB = 32 + 4 /* gpio 1 pin 4 */
- am335xResetC = 32 + 1 /* gpio 1 pin 1 */

Ultimately, GPIO allocation will have this structure.

```
// Allocate pins for GPIO
status |= sysGpioAlloc (am335xResetA, "RESET A");
status |= sysGpioAlloc (am335xResetB, "RESET B");
status |= sysGpioAlloc (am335xResetC, "RESET C");
```

The output operation mode should be enabled by writing a '0' inside the 'Output Enable' (GPIO_OE) register for each one of the output pins.

```
// Set as output
status |= sysGpioSelectOutput (am335xResetA);
status |= sysGpioSelectOutput (am335xResetB);
status |= sysGpioSelectOutput (am335xResetC);
```

Even though the input of '$\overline{RESETx}$' is inverted, the action of the LVX IC at the output of the pins, driving out a 'low level' signal will put a 'high level' input in the DRV8312. The output value is selected by setting or clearing the bit number corresponding to the output pin in the register 'GPIO Data Output' (GPIO_DATAOUT) for each output pin.

```
// Set low value
sysGpioSetValue(am335xResetA,0);
sysGpioSetValue(am335xResetB,0);
sysGpioSetValue(am335xResetC,0);
```

It is a good practice to release the pin allocation before starting a new configuration to avoid errors in the allocation of new pins. The already-made function by TI allows to do it by inserting the pin number as parameter.

```
// Releases pin from specified purpose
status |= sysGpioFree (am335xResetA);
status |= sysGpioFree (am335xResetB);
status |= sysGpioFree (am335xResetC);
```

GPIO pins and their features will have a more prominent role in the 'closed-loop' control algorithm.

12.1.4. Space Vector Modulation (SVM)

Space vector modulation is the most common algorithm for controlling PWM duty for DC motors [20]. This method has six possible 'active' states in which the vector can be included, figure 12.12, and two extra states ($\{0,0,0\} and \{1,1,1\}$) called 'zero vectors' that have no effect in the voltage vector. The voltage vector can be generated using the 'Clarke Transformation' and the FOC explained, but this project will use a 'virtual' voltage vector instead, whose amplitude is a percentage (uConst) of the voltage value in the capacitors of the voltage supply (UZK = 24 V).

$$Uabs = uConst * \frac{1}{\sqrt{3}} * UZK; \tag{12.3}$$

Figure 12.12.: SVM possible sectors

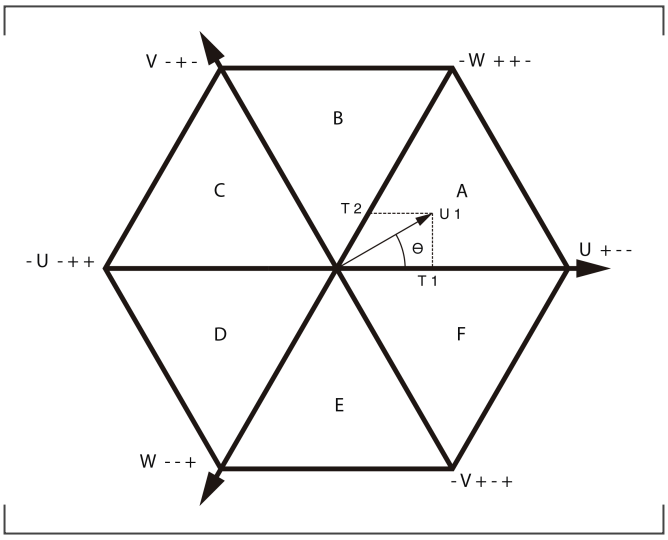


Table 12.1.: Switching sequence and output voltage for SVM

U	V	W	Phase A	Phase B	Phase C	Sector
1	0	0	V_{DC} +	0	V_{DC} -	1
1	1	0	0	V_{DC} +	V_{DC} -	2
0	1	0	V_{DC} -	V_{DC} +	0	3
0	1	1	V_{DC} -	0	V_{DC} +	4
0	0	1	0	V_{DC} -	V_{DC} +	5
1	0	1	V_{DC} +	V_{DC} -	0	6

Each 25 μs, ePWM0 generates an interrupt, whose ISR will perform several calculations to determine the PWM duty cycle that is going to be reloaded to the PWM in the next iteration. The calculation is done within two functions: 'CalculateMotorControl' which calculates the angle value, given as parameter to the space vector function, and saves the PWM result, and 'CalculateMotorControlSpacePointer', which calculated the space vector given the angle.

Since the open-loop control lacks any form of feedback, it is impossible to determine the actual position of the rotor shaft. However, this can be avoided by creating a "virtual" rotating voltage vector dependent on the input speed. At every interrupt, the angle of the voltage vector will be incremented a certain angle, whose amplitude is based on the desired speed; the higher the speed, the higher the amplitude for a fixed amount of time. Below there is a general equation for calculating angles in radians. The calculated angle should maintain its value between 0 and 2π.

$$Angle = Angle + [(2\pi)x\frac{revolutionsPerMinute}{60}x(25x10^{-6})] \tag{12.4}$$

It can be noticed here,when talking about Electrical Revolution / Mechanical Revolution conversion, thatthe previous equation is a 1:1 equation (One electrical revolution equals one mechanical revolution). This is true only when the BLDC motor has only one pole-pair; however, the motor in this project has four pole-pairs. Having four pole-pairs instead of one implies that, at the time one electrical revolution is completed, only a quarter of the mechanical revolution has been completed. A quick way of resolving this issue, since the program has no feedback from the motor, is to multiply the speed variable, 'revolutionsPerMinute', by the number of pole-pairs of the motor, for example, four electrical revolutions per mechanical revolution.

As seen in figure 12.12, the voltage vector's position could be in one of the six possible vectors depending on its angle value. The amplitude of each sector is 60º, therefore the angle should be converted from radians to degrees using the equation:

$$Angle(degrees) = Angle(radians)x\frac{360}{2\pi} \tag{12.5}$$

Dividing the resulting angle in degrees by 60 will result on a number whose integer part will be a number between '0' and '6' (decimals are not important), with 360 as the sole value that gives '6' as a result. The results, table 12.2, will be saved into a variable, for it will be used later in the code.

Table 12.2.: Sectors according to angle value

Angle	Sector
0 - 60	0
60 - 120	1
120 - 180	2
180 - 240	3
240 - 300	4
300 - 360	5

Figure 12.13 show the sequence that will lead to the equations that calculate the pwm duty based on the SVM's space vector as shown in figure 12.12. Each sector has four 'states', the first and the last are 'zero vectors' and the two in the middle are 'active vectors' and the equations are obtained by the sum of each sector's U, V, and W variables and their active vectors.

In figure 12.13 't1' and 't2' correspond to the same names in figure 12.12. 't2' is the sine of the actual angle, while 't2' is the sine of $\frac{\pi}{3} - \theta$, not its cosine. These values should be multiplied by a constant number, which will be a value depending on the time period (Timep = 25 μs), the voltage vector (Uabs) and the voltage at the capacitors (UZK). 't0' is the result of substracting 't1' and 't2' from the time period. [20]

$$constValue = Timepx\sqrt{3}x\frac{Uabs}{Uzk} \tag{12.6}$$

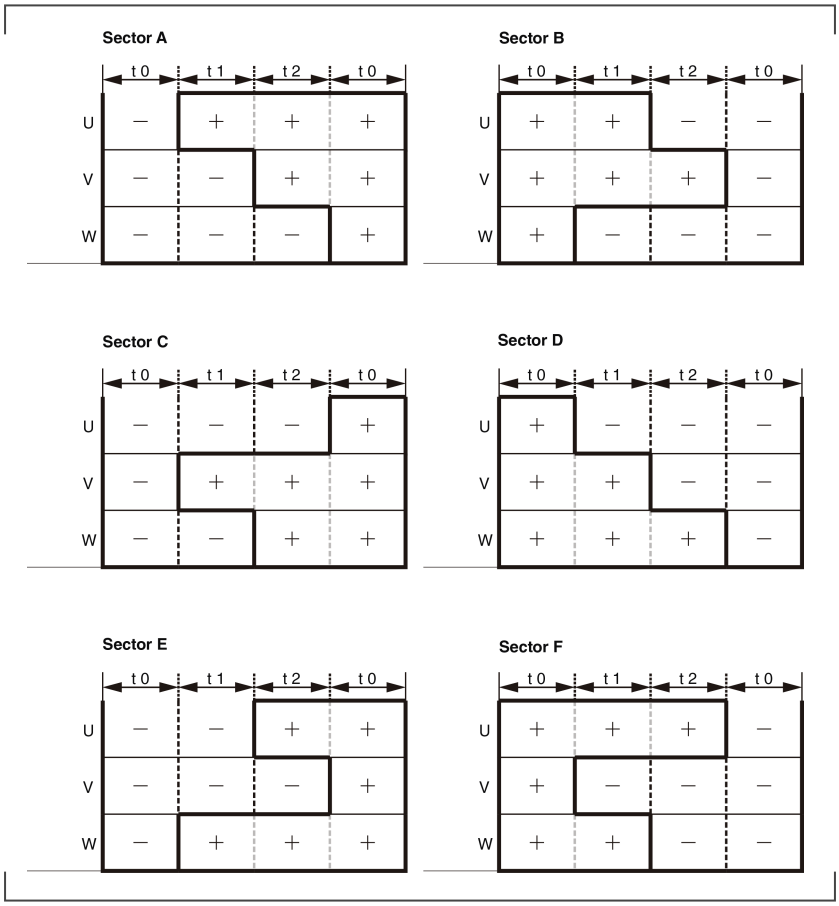
$$t1 = sin(\frac{\pi}{3} - \theta) \tag{12.7}$$

$$t2 = sin(\theta) \tag{12.8}$$

$$t0 = Timep - t1 - t2 \tag{12.9}$$

The equations for each sector are obtained from figure 12.13 by looking at each 'state'. In the first 'state', t0, the status should be a 'zero vector' ($\{0,0,0\} or \{1,1,1\}$), afterwards, state t1, is the status at the beginning of the sector as seen in figure 12.13; for example, $\{0,0,1\}$ at the beginning of sector A. The next part, t2, is the status at the end of the sector, $\{0,1,1\}$ for sector A, and the final t0 is another 'zero vector'. It should be noticed that it only changes one state in the 'active vectors' because the switching losses

Figure 12.13.: Sectors



are lower. By doing the same procedure for every sector, a set of equations is obtained, with which makes possible to calculate the pwm duty for each phase.

```
Sector A:
        u = t0/2 + t1 + t2
        v = t0/2 + t2
        w = t0/2

Sector B:
        u = t0/2 + t1
        v = t0/2 + t1 + t2
        w = t0/2

Sector C:
```

```
        u = t0/2
        v = t0/2 + t1 + t2
        w = t0/2 + t2

Sector D:
        u = t0/2
        v = t0/2 + t1
        w = t0/2 + t1 + t2

Sector E:
        u = t0/2 + t2
        v = t0/2
        w = t0/2 + t1 + t2

Sector F:
        u = t0/2 + t1 + t2
        v = t0/2
        w = t0/2 + t1
```

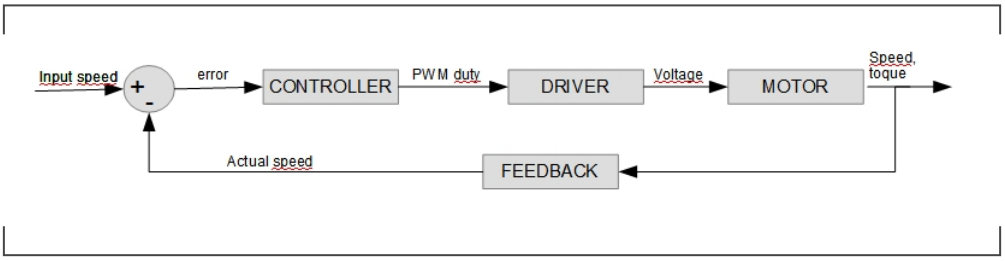
The results are finally saved into variables, a struct was created in order to do so, and their value will be loaded into the PWM on the next ISR iteration.

12.2. Closed-Loop Control

Unlike open-loop control, closed-loop control has a 'feedback' control loop that allows to read the actual position of the rotor's shaft and correct any error that the system can have during its operation. Closed-loop control is used when there is a strong need for controlling the system's output in order to avoid errors in the process. It can also be used in 'Machine Learning' processes, though this is not the case.

The closed-loop control diagram is essentially equal to the open-loop's, only adding a 'feedback' branch reflected in figure 12.14, after the system's output. The BLDC motor has three 'Hall sensors' embedded within, which provides information about the motor's position and, eventually, its speed. The latter will be used to calculate the error between the desired speed and the actual speed, which will be fed into the PI controller. Hall sensor also gives information about the position of the shaft, which will have a direct impact on the PWM energizing sequence.

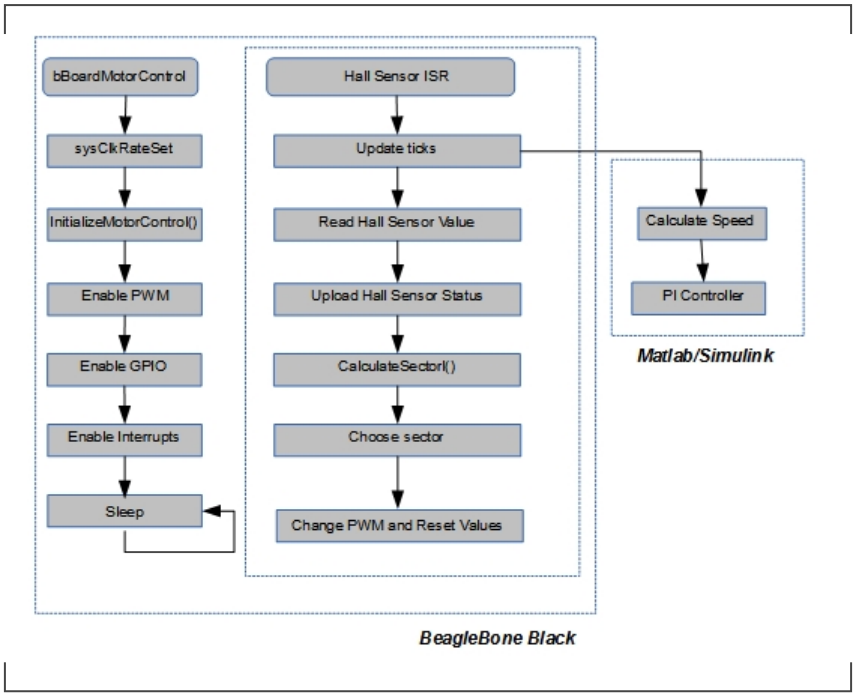
Figure 12.14.: Closed-loop diagram



There will be a few changes in the use of PWM and GPIO configuration, mostly in the latter with the introduction of GPIO interrupts. There will not be a 25 μs PWM interrupt anymore, substituted by interrupts that will be activated each time a Hall sensor changes its state.

The only change inside function 'bBoardMotorControl()' will be adding a call to the interrupt configuration function (openIntConfig()). The rest of the function will remain exactly the same as it was in the open-loop control. The flow diagram in figure 12.15 shows the structure of the program.

Figure 12.15.: Closed-loop flow diagram



12.2.1. Pulse-Width Modulation (PWM)

Most of the ePWM configuration process has not changed from the one used in the open-loop control. However, there are some minor changes that will affect the ePWM behaviour that are worth explaining. The major change will be changing the count mode from 'Up-Down' no 'Up' count mode, and the AQ, since there is no need for a symmetrical PWM signal anymore.

The count mode is changed when the frequency of the ePWM signal is set. The AQ should be then configure to reflect a regular PWM signal, as figure 12.3 shows, instead of the symmetric signal created in the open-loop control. The process' signal will start in 'high level' and switch to 'low level' once the counter has reached the 'CAU' value. The inverting effect of the LVX should be noticed when writing the AQ configuration, so the code will be 'low' the beginning and the switch to 'high' in order to obtain the desired signal.

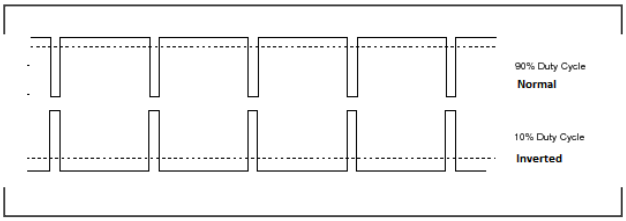
```
EHRPWMPWMOpFreqSet( SOC_EPWM_0_REGS,
                    PWMClockFrequency/CLOCK_DIV_VAL,
                    PWMCycleFrequency,
                    EHRPWM_COUNT_UP,
                    EHRPWM_SHADOW_WRITE_DISABLE );

EHRPWMConfigureAQActionOnA( SOC_EPWM_0_REGS,
                    EHRPWM_AQCTLA_ZRO_EPWMXALOW,
                    EHRPWM_AQCTLA_PRD_DONOTHING,
                    EHRPWM_AQCTLA_CAU_EPWMXAHIGH,
                    EHRPWM_AQCTLA_CAD_DONOTHING,
                    EHRPWM_AQCTLA_CBU_DONOTHING,
                    EHRPWM_AQCTLB_CBD_DONOTHING,
                    EHRPWM_AQSFRC_ACTSFB_DONOTHING);
```

In addition to the changes above, the commutation method for the closed-loop control needs to have two different PWM signals, one 'normal' signal and one 'inverted' signal (figure 12.16) The reason behind this need will be explained when talking about the commutation method for the closed-loop, which will be 'block commutation'.

An 'inverted' PWM signal is, basically, a mirrored 'normal' PWM signal. It can also be explained as the signal whose PWM duty is the remaining duty of the 'normal' signal; a 'normal' signal with a duty cycle of 90% will have a 10% 'inverted' signal. A function containing the configuration for the inverted PWM signals will be made, and the input

Figure 12.16.: PWM normal duty and inverted duty



parameter 'numPWM' will contain the address to the module register of the PWM signal, PWM0 or PWM2.

```
EHRPWMConfigureAQActionOnA( numPWM,
                            EHRPWM_AQCTLA_ZRO_EPWMXAHIGH,
                            EHRPWM_AQCTLA_PRD_DONOTHING,
                            EHRPWM_AQCTLA_CAU_EPWMXALOW,
                            EHRPWM_AQCTLA_CAD_DONOTHING,
                            EHRPWM_AQCTLA_CBU_DONOTHING,
                            EHRPWM_AQCTLB_CBD_DONOTHING,
                            EHRPWM_AQSFRC_ACTSFB_DONOTHING );
```

12.2.2. General Purpose Input/Output (GPIO) and interrupt

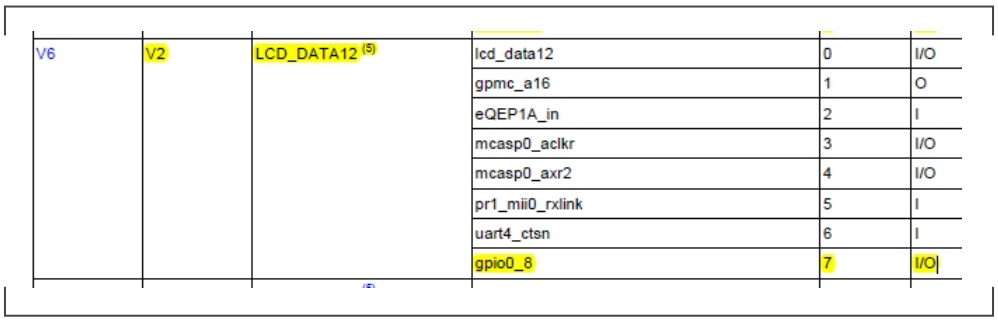
The feedback loop is done by reading the input of the Hall sensors embedded in the BLDC motor. The sensors are connected to three GPIO pins in the BBB, which will be configured as an input to read its state. They will also promt an interrupt each time one sensor changes its state: from 'high' to 'low' or vice-versa. The GPIO pins used for this purpose are:

- $HALL1$ (UART3_CTSN) $\rightarrow$ U4
- $HALL2$ (UART4_CTSN)$\rightarrow$ V2
- $HALL3$ (UART3_CTSN)$\rightarrow$ U3

The general configuration follows the same pattern as with the '$\overline{RESETx}$' outputs in the open-loop control. Figures 12.17, 12.18 and 12.19 show the configuration modes for the selected GPIO pins. The code snippet below is fairly similar to the one in the open-loop

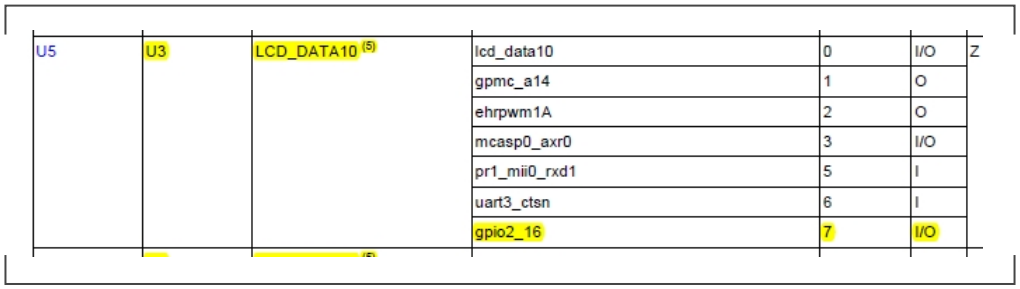
control, except for the direction of the pin, writing the value '1'(IEN) into the register to select the input mode.

Figure 12.17.: Hall 1 Mode [18]



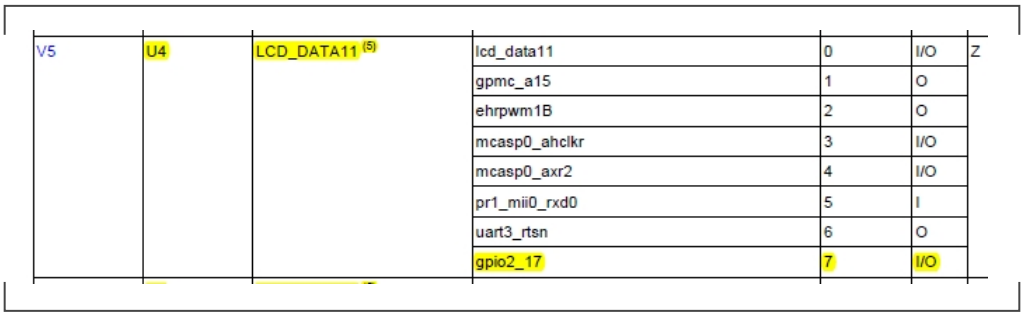
V6	V2	LCD_DATA12 (5)	lcd_data12	0	I/O
			gpmc_a16	1	O
			eQEP1A_in	2	I
			mcasp0_aclkr	3	I/O
			mcasp0_axr2	4	I/O
			pr1_mii0_rxlink	5	I
			uart4_ctsn	6	I
			gpio0_8	7	I/O

Figure 12.18.: Hall 2 Mode [18]



U5	U3	LCD_DATA10 (5)	lcd_data10	0	I/O	Z
			gpmc_a14	1	O	
			ehrpwm1A	2	O	
			mcasp0_axr0	3	I/O	
			pr1_mii0_rxd1	5	I	
			uart3_ctsn	6	I	
			gpio2_16	7	I/O	

Figure 12.19.: Hall 3 Mode [18]



V5	U4	LCD_DATA11 (5)	lcd_data11	0	I/O	Z
			gpmc_a15	1	O	
			ehrpwm1B	2	O	
			mcasp0_ahclkr	3	I/O	
			mcasp0_axr2	4	I/O	
			pr1_mii0_rxd0	5	I	
			uart3_rtsn	6	O	
			gpio2_17	7	I/O	

```
struct am335xPadConf hall_sensor_pad [] = {
        /* LCD_DATA12 HALL-1 */
        {CONTROL_PADCONF_LCD_DATA12, (IEN | MODE7 ) },
        /* LCD_DATA10 HALL-2 */
        {CONTROL_PADCONF_LCD_DATA10, (IEN | MODE7 ) },
         /* LCD_DATA11 HALL-3*/
        {CONTROL_PADCONF_LCD_DATA11, (IEN | MODE7 ) },
        { AM335X_PAD_END, AM335X_PAD_END }};
```

The pin allocation follows the same structure as before, taking into account that the pins are located in different modules (GPIO0 and GPIO2) and, as such, its pin number differ. They should also be configured as input by selecting the right function.

- am335xGPIO0_8 = 8 /* gpio 0 pin 8 */
- am335xGPIO2_16 = 2 * 32 + 16 /* gpio 2 pin 16 */
- am335xGPIO2_17 = 2 * 32 + 17 /* gpio 2 pin 17 */

```
sysGpioAlloc(am335xGPIO0_8,"HALL 1");
sysGpioSelectInput(am335xGPIO0_8);

sysGpioAlloc(am335xGPIO2_16, "HALL 2");
sysGpioSelectInput(am335xGPIO2_16);

sysGpioAlloc(am335xGPIO2_17, "HALL 3");
sysGpioSelectInput(am335xGPIO2_17);
```

These lines will have to be configured to perform an interrupt every time a hall sensor changes its state in order to read the current sector and enable the possibility of BLDC speed calculation, basic for the configuration of the PI controller afterwards. Interrupts should be activated whenever a rising or falling edge is read. The function is activated by writing the variable (GPIO_IRQ_BOTH_EDGE_SENSITIVE), and, by doing so, there will be a better control the status of the motor. The registers in charge of enabling interrupts for one or both edges are (GPIO_RISINGDETECT) for rising edge, and (GPIO_FALLINGDETECT) for falling edge.

The three interrupts are connected to the same ISR, as the task they will perform is the same. This ISR will have the interrupt pin as input parameter, allowing to know which pin prompted the interrupt. The GPIO has inner glitch issues, if no debounce handling, the GPIO value may not be right. A debounce time, measured in steps of 31μs ($AM335X\_KEY\_DEBOUNCE = 7936$), will take care of the issue. The debounce is enabled by writing a '1' in the pin corresponding to the GPIO inside the register (GPIO_DEBOUNCENABLE), and the time value in steps is loaded into (GPIO_DEBOUNCINGTIME)

```
sysGpioSetDebounceTime(am335xGPIO0_8,
        AM335X_USDBOOT_KEY_DEBOUNCE);
sysGpioEnableDebounce(am335xGPIO0_8);
sysGpioIntConnect(am335xGPIO0_8,
```

```
            GPIO_IRQ_BOTH_EDGE_SENSITIVE,(FUNCPTR)hallSensorIsr,
            (void *)am335xGPIO0_8);

sysGpioSetDebounceTime(am335xGPIO2_16,
          AM335X_USDBOOT_KEY_DEBOUNCE);
sysGpioEnableDebounce(am335xGPIO2_16);
sysGpioIntConnect(am335xGPIO2_16,
            GPIO_IRQ_BOTH_EDGE_SENSITIVE,(FUNCPTR)hallSensorIsr,
            (void *)am335xGPIO2_16);

sysGpioSetDebounceTime(am335xGPIO2_17,
          AM335X_USDBOOT_KEY_DEBOUNCE);
sysGpioEnableDebounce(am335xGPIO2_17);
sysGpioIntConnect(am335xGPIO2_17,
            GPIO_IRQ_BOTH_EDGE_SENSITIVE,(FUNCPTR)hallSensorIsr,
            (void *)am335xGPIO2_17);
```

Finally, the interrupt should be enabled by writing '1' in (GPIO_IRQSTATUS_SET_x). It is important to note that the ISR should be connected first and then enabled to avoid possible errors when running the program.

```
sysGpioIntEnable(am335xGPIO0_8);
sysGpioIntEnable(am335xGPIO2_16);
sysGpioIntEnable(am335xGPIO2_17);
```

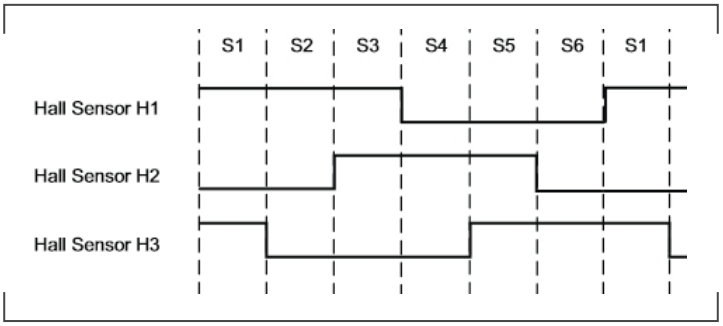
Each time a rising or falling egde is detected, the ISR will be called and give the number of the pin that performed the interrupt. Knowing which pin raised the interrupt is necessary for the implementation of the commutating sequence, as it depends heavily on the data from the Hall sensors. More specifically, it provides information about the sector in which the motor is located. Also, the number of ticks performed until the interrupt will be read, as it will be used as a way of knowing the actual speed of the motor.

```
TickCounter.countTicks = tickGet();
tmp = sysGpioGetValue (gpio);
```

The reading of the pin os performed by reading bit inside the register (GPIO_DATAIN) associated to the pin that entered the interrupt. Its value is then switched, if it was 'low' is changed to 'high', due to the LVX inverting effect, and then loaded into the variable that corresponds to the hall sensor that activated the interrupt.

One of the hall sensors, the one connected to GPIO0-8 in this case, will be used to know the actual speed of the motor. Reading only one channel instead of all of them,is the simplest and quickest method as it reduces the amount of code and its complexity. Whenever GPIO0-8 enters the ISR, the actual tick count for speed is calculated by subtracting the actual value of the count minus the previous value. The previous value is the previous rising edge counter value. The time between two rising edges of a hall sensor is equal to one electrical revolution (S1 to S1 for Hall A in figure 12.20), and it can be used to calculate the rotation speed using the equivalent electrical/mechanical revolutions conversion.

Figure 12.20.: DRV8312 Hall sensor sequence [13]



Since the interrupt occurs at rising and falling edges, and only rising edges are required, discriminating the falling edge is done by reading the actual state of the interrupt pin. If the actual state is 'high', the triggered was a rising one.

```
if(am335xGPIO0_8 == 1){
actualTicks = TickCounter.countTicks - TickCounter.pastTicks;
TickCounter.pastTicks = TickCounter.countTicks; }
```

The result of the operation, actualTicks, will be fed into the PI controller on the Matlab environment. Also, the final step before leaving the ISR, and waiting for the next ISR to occur, will be calling the function (CalculateSector(&sensor)) where the actual state of the motor will be calculated and later used to perform the commutation sequence needed for the BLDC.

12.2.3. Commutation sequence

The commutation sequence for the closed-loop control is simpler than the open-loop control, as most of the information about the configuration is contained within the

DRV datasheet. Figure 12.21 shows the configured signals status for each of the six commutation steps: 'Hall Sensor', 'PWM' and '$\overline{RESET}$'. The 'Phase Current' line is left out in this project. As said in the previous section, the commutation sequence is called from the ISR via (CalculateSector(&sensor).

The sector's actual status is obtained with the results given from the reading of the 'Hall sensor' input lines in the ISR. The PWM signal can trigger two possibilities, any duty over 50% has V_{DC} + effect, and below 50%, V_{DC} - effect. A summary of the signal's status is shown in table 12.3, looking very similar to the one in section 2.2.2 'Block commutation', table 7.3.

Table 12.3.: Switching sequence and output voltage for Block Commutation

H1	H2	H3	PWM A	PWM B	PWM C	$\overline{RESETA}$	$\overline{RESETB}$	$\overline{RESETC}$
1	0	1	V_{DC} +	V_{DC} -	0	1	1	0
1	0	0	V_{DC} +	0	V_{DC} -	1	0	1
1	1	0	0	V_{DC} +	V_{DC} -	0	1	1
0	1	0	V_{DC} -	V_{DC} +	0	1	1	0
0	1	1	V_{DC} -	0	V_{DC} +	1	0	1
0	0	1	0	V_{DC} -	V_{DC} +	0	1	1

A series of nested 'if-else' loops will provide the interface for driving out the 'PWM' and '$\overline{RESETx}$' signals depending on whichever sector the motor is in. Afterwards, the program exits the ISR and waits for a new interrupt.

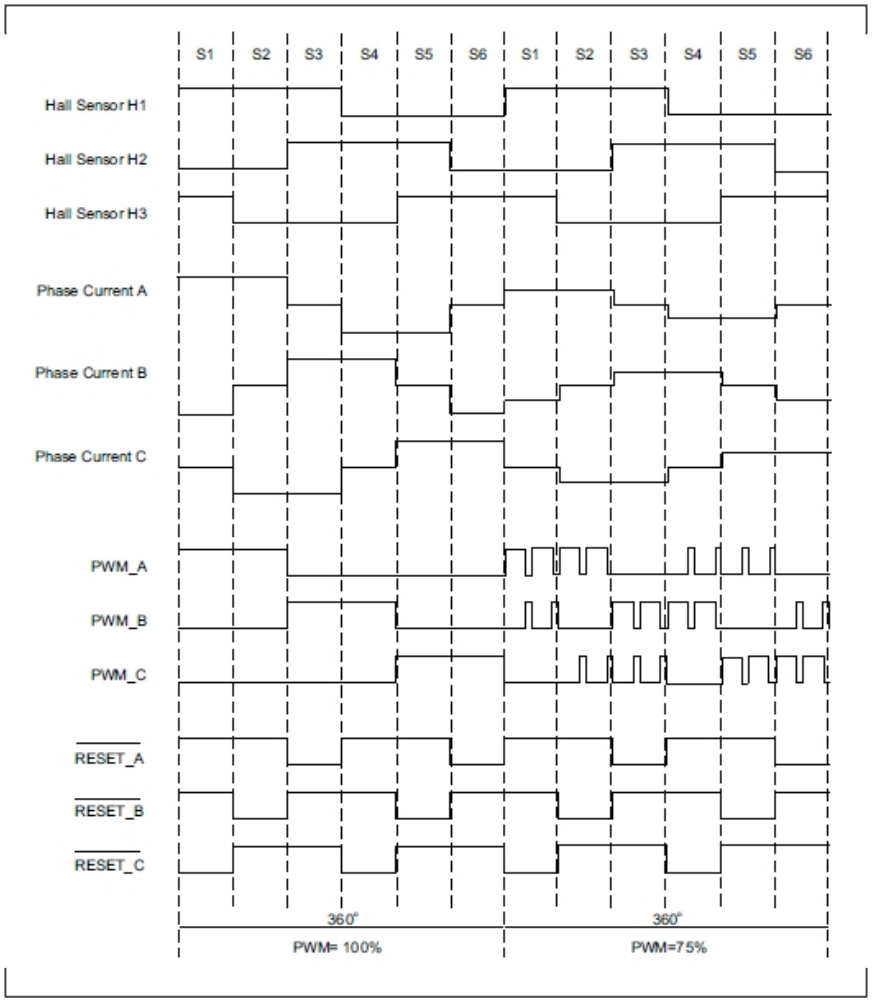
The speed is controlled directly by the duty cycle, as it controls the voltage flow into the motor. Any value above 50% duty cycle will have a rotation speed in one direction, below 50% will have a rotation speed in the opposite direction. The equation relating speed and duty depends on the characteristics of the BLDC.

12.2.4. Matlab Integration and PI Controller

Matlab integration will be the final element in the closed-loop control. The environment 'Simulink' will be used to crete a project in which the actual speed of the motor is calculated and then, its value fed into a PI controller. The 'Simulink' environment will also connect the variables of its project with those inside the BBB kernel through 'S-functions'.

S-functions are a computer language description of a Simulink block written in several

Figure 12.21.: Hall Sensor Control with 6 Steps [13]

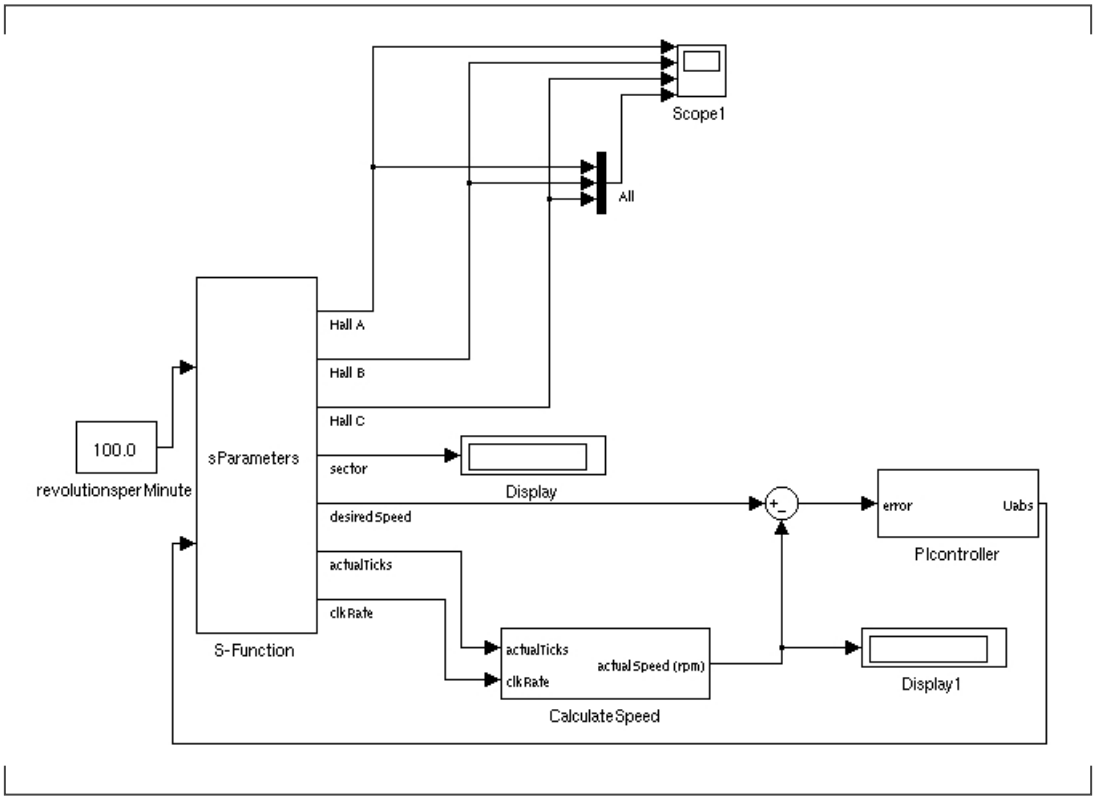


programming languages as MATLAB, C/C++, FORTRAN, etc. and compiled as a MEX-file. S functions can accommodate continuous, discrete, and hybrid systems. It is also possible to implement an algorithm in an S-function and use the S-Function block to add it to a Simulink model.

To connect the BBB and Simulink variables, they should be declared as 'extern' inside the S function and with the same name in both platforms. As an example, below there is a code snippet showing how the variables should be declared.

```
extern unsigned char hall1Status, hall2Status, hall3Status;
extern int sector;
```

Figure 12.22.: Block diagram in Simulink showing S functions




```
extern float revolutionsPerMinute;
extern unsigned int actualTicks;
extern unsigned int clkRate;
```

Several functions must be written for the S function to work properly. I/O allocation is done in the function 'mdlInitializeSizes'. Each input or output has its own number, which will be addressed later to read in and drive out the variables for the system. The I/O variables are initialized with the same type that they originally had in the BBB's code.

```
/*  set number of block inputs */
if (!ssSetNumInputPorts(S, 2)) return;
ADD_DOUBLE_INPUT(S, 0, 1)        // revolutions per minute
ADD_DOUBLE_INPUT(S, 1, 1)        // PI controller output

/* set number of block outputs */
```

```
if (!ssSetNumOutputPorts(S, 7)) return;
ADD_DOUBLE_OUTPUT(S, 0, 1)  // Hall sensor 1
ADD_DOUBLE_OUTPUT(S, 1, 1)  // Hall sensor 2
ADD_DOUBLE_OUTPUT(S, 2, 1)  // Hall sensor 3
ADD_UINT8_OUTPUT(S, 3, 1)  // Sector
ADD_DOUBLE_OUTPUT(S, 4, 1) // desired speed
ADD_UINT32_OUTPUT(S, 5, 1) // actual ticks
ADD_UINT32_OUTPUT(S, 6, 1) // CLKrate
```

The variable inputs are written into the declared variables, using the same number as they were declared as inputs with the function 'get_input'.

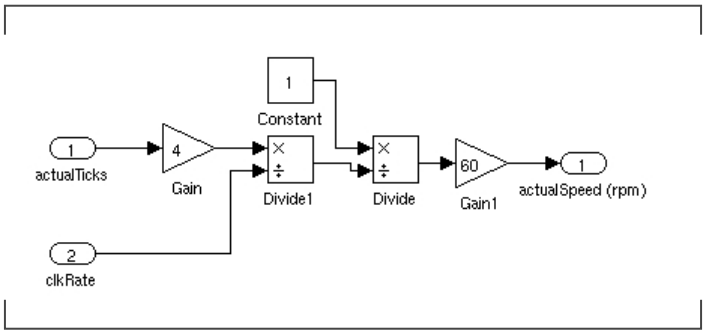
```
revolutionsPerMinute = get_input(0, double, 0);
```

As well as outputs, which are driven out to the system with 'set_output'. Variable type and number are also needed in the function's parameters.

```
set_output(4, double, 0, revolutionsPerMinute);
set_output(5, uint32_T, 0, errorTicks);
set_output(6, uint32_T, 0, clkRate);
```

The actual speed is calculated using the 'ticks' that were read in the BBB's hall sensor ISR. Borh variables, in the BBB and Simulink, are interconnected so that their values are equal at all times. Using the electrical/mechanical revolution conversion and the 'Clock rate', in Hz, used in the BBB, it is possible to calculate the speed of the BLDC. Figure 12.23 show the speed calculation for a 4 pole-pair motor.

Figure 12.23.: Speed calculation



Finally, the second 'S function' is going to be used to write the equations that will form the PI control loop. A PID controller calculates an error value as the difference between

a measured process variable (actual speed) and a desired setpoint (desired speed), and attempts to minimize the error by adjusting the process through use of a manipulated variable. The PI controller is a form of controller in which the derivative (D) term is not used, leaving the proportional (P) and integral (I) terms as the only affecting the error.

The proportional term produces an output that is proportional to the current error value, thus it is only needed to multiply the error (e(t)) by the proportional gain (K_p). If the proportional gain is too high, the system may become unstable, if it is too low, the output is small and can create a less responsive or less sensitive controller.

$$P_{Term} = K_p e(t); \tag{12.10}$$

The integral term considers both the magnitude of the error and the duration of the error; the sum of instantaneous error over time. The accumulated error (integral) is multiplied by the integral gain (K_p) and added to the proportional term to form the output of the PI controller.

$$I_{Term} = K_i \int_0^t e(t)\,dt \tag{12.11}$$

$$PI = K_p e(t) + K_i \int_0^t e(t)\,dt \tag{12.12}$$

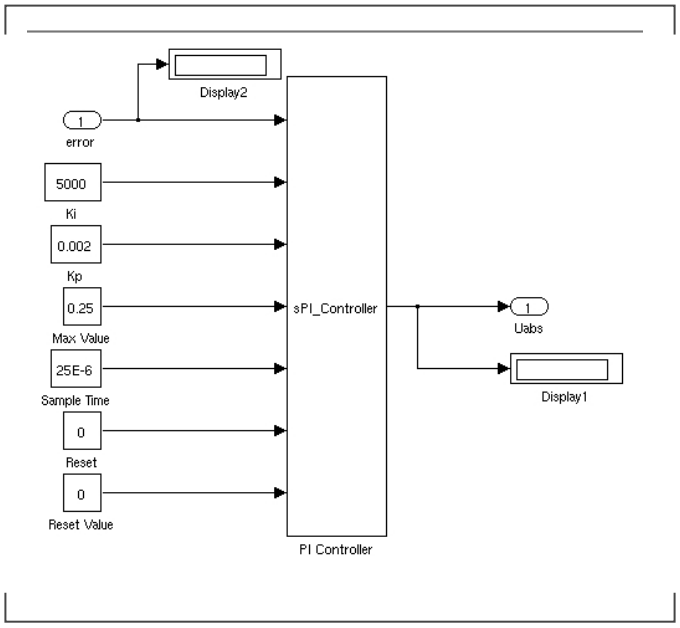
However, since the PI controller is going to be a part of a microcontroller, it would be a good practice to write the controller as a discrete controller. The proportional term is left as it was before, but the integral term should be discretized.

```
/* Proportional Term */
PTerm = Kp * Error;
/* Integral Term*/
Iterm = PTerm / Ki;
/* Integration */
Ioutput = past_Ioutput + Error * (sampleTime / 2);
/* Output */
PIOutput = PTerm + Ioutput;
```

The tuning of the PI controller can be done though algorithms, like the Ziegler–Nichols method or the Cohen–Coon method. However, the parameters will be adjusted manually

in this case, as is the method with less mathematical requirements. Figure 12.24 shows the input variables to the PI controller and its output.

Figure 12.24.: PI controller parameters



After all the parts have been set, the model has to be built and loaded into the BBB. The 'S Functions' are compiled through the 'Matlab' command line. The MEX-file is compiled by writing 'mex' and the name of the 'S function'. In the 'Simulink' environment, clicking 'Tools → Real Time Workshop → Build', will build the model and create the necessary files.

In the 'VxWorks Workbench', the terminal window is used to get the files from the computer and load them into the BBB's kernel. The 'cmd' line will enter the console and, from there, the file path is selected by writing it in the 'cd' command. To load the file, the command below should be written and, afterwards, enter the command 'main' to enter the file and run it.

```
ld 1,0,"PIcontroller"
```

Back in the 'Simulink' environment, once all the steps are done and the BLDC is running with the PI controlled loaded, the changes in real time can be seen by connecting 'Simulink' to the BBB clicking on 'Simulation → Connect to target'.

Part VI.

CONCLUSIONS

In summary, this thesis serves as a way of programming a BLDC motor with a small, low-cost platform as the BeagleBone Black is, and the integration with a driver board, diferent from the processor board. Also, connecting the BeagleBone Black board to an external interface, a PC, and load a Matlab project within the kernel is a new approach in the way of creating controllers.

The programming of BLDC motors is well known, however the development of new prototyping boards, smaller and more powerful, also adds complexity to the process. Two approaches have been made, an open-loop control and a closed-loop control The open-loop control uses a SVM method to create a rotating voltage vector in order to execute the PWM sequence. Later, the closed-loop control uses a 'Block commutation' approach to connect the data from the 'Hall' sensors with the PWM signals that are used to run the motor. A PI controller is used to guarantee that the desired speed and the actual speed of the BLDC motor is the same at all times. Interfacing the BBB with the Matlab environment provides a powerful tool to create controllers and perform calculations with the 'Simulink' environment.

However, the usefulness of an Operating System inside the processor, instead of directly programming it, can not be assured at this point. It is possible that, in the future, adding new features might be a good start point in favour of the OS, compared to not using an OS.

Further lines of investigation would be the development of a closed-loop control using other commutation methods: like 'FOC', as 'Block commutation', due to its simplicity, is not the best approach to use in 'low speed' applications. Furthermore, 'FOC' controls the current of the BLDC, which might be an advantage in the motor control. The DLR has also developed a new board, Phytec board, for this application, so embedding the developed code into the new board can also be a continuation of the project.

Part VII.

REFERENCES

Bibliography

[1] Lunokhod I.Department of Lunar and Planetary Research. Moscow University. Russia. Retrieved from: [RUSSIAN]
http://selena.sai.msu.ru/Home/Spacecrafts/Lunokhod1/lunokhod1e.htm

[2] Luna-Glob probe to launch in 2014. Solar System Exploration Research Virtual Institute (SERVI) of NASA. USA. 2011. Retrieved from:
http://sservi.nasa.gov/articles/luna-glob-probe-launch-2014/

[3] Official Website of the "China National Space Administration" (CNSA).China National Space Administration. China.
http://www.cnsa.gov.cn/n360696/index.html

[4] Official Website of "Chandrayaan" Project. Indian Space Research Organisation (ISRO). India.
http://www.isro.org/chandrayaan/htmls/home.htm

[5] Larry K. Baxter. Capacitive Sensors: Design and Applications. Edn. Wiley-IEEE Press. New York, USA, 1996.

[6] P. Yedamale. AN885:Brushless DC Motor Fundamentals. Microchip Technology Incorporated. USA. 2003.

[7] P. Supinya, S. Athanasios, Dr. M. Lambrechts, Prof. Ir. L. Bientsman. Design of a high power controller for BLDC-motors on FPGA and dsPIC. Leuven Engineering College. Leuven, Belgium. 2009-2010.

[8] K. Zierhut. Wye/Delta In Perspective. Modern Machine Shop. 1999. Article source:
http://www.mmsonline.com/articles/wyedelta-in-perspective

[9] K. Zierhut. BLY17 Series - Brushless DC Motors. Anaheim Automation. USA. Webpage:
http://www.anaheimautomation.com

[10] S. Lee, T.Lemley. A comparison study of the commutation methods for the three-phase permanent magnet brushless DC motor.

[11] S. Keeping. Controlling Sensorless, BLDC Motors via Back EMF. Digi-Key. 2013. Retrieved from:
http://www.digikey.com/en/articles/techzone/2013/jun/controlling-sensorless-bldc-motors-via-back-emf

[12] P. Yedamale. AN: 970. Using the PIC18F2431 for Sensorless BLDC Motor Control. Microchip Technology Incorporated. USA. 2000.

[13] DRV8312 Three Phase PWM Motor Driver. Texas Instruments USA. Revised: January 2014.
http://www.ti.com

[14] Official website of WindRiver. Wind River Releases 64-Bit VxWorks RTOS. WindRiver. USA. 2011.
http://www.windriver.com/news/press/pr.html?ID=8881

[15] Website of FTDI chip
http://www.ftdichip.com

[16] Website of Texas Instruments
http://www.ti.com

[17] G.Coley. BeagleBone Black System Reference Manual. BeagleBoard.org. USA. 2013.
http://circuitco.com/support/index.php

[18] AM335x ARM® Cortex-A8 Microprocessors (MPUs). Texas Intruments. USA. 2013.

[19] AM335x ARM® Cortex-A8 Microprocessors (MPUs) Technical Reference Manual. Texas Intruments. USA. 2013.

[20] ATMEL Corporation. AVR32710: Space Vector Modulation using AVR32 UC3 Microcontroller. Atmel Corporation. California, USA. 2009.

Part VIII.

ANNEXES

MÁSTER DE INGENIERÍA MECATRÓNICA			Control of BLDC motors for a terrestrial Lunar Rover prototype		Pag. Mediciones y Presupuesto	
Nº ORDEN	CONCEPTOS	Nº UNIDADES	FABRICANTE	PRECIO UNITARIO (€)	PRECIO TOTAL (€)	TOTAL (€)
1	BeagleBone Black	1	BeagleBone.org	46,10	46,10	
2	Three Phase BLDC Motor Kit with DRV8312	1	Texas Instruments	220,00	220,00	
3	74LVX14M, CMOS Schmitt-Trigger Inverter	2	Fairchild Semiconductor	0,35	0,70	
4	Cable Cinta Plano Speedbloc, 50 vías	0,3	Speedblock	113 (30m)	1,13	
5	Placa de prototipado con orificios RE010	1	Roth Elektronik	9,17	9,17	
6	Placa para SMD RE931-01	2	Roth Elektronik	7,74	15,48	
7	Cable TTL-232R-3V3	1	FTDI Chip	20	20,00	
						312,58

BLY17 Series - Brushless DC Motors

FEATURES

- ***42mm Square Body***
- ***Compact Size and Power Density***
- ***Cost-Effective Replacement for Brush DC Motors***
- ***Long Life and Highly Reliable***
- ***Can Be Customized for:***
  - ***Maximum Speed***
  - ***Winding Current***
  - ***Shaft Options***
  - ***Cables and Connectors***
- ***CE Certified and RoHS Compliant***

DESCRIPTION

The BLY17 Series Brushless DC Motors come in a compact package with high power density. These motors are cost-effective solutions to many velocity control applications. They come in four different stack lengths to provide you with just the right torque for your application. A number of windings are available off-the-shelf and all the motors can be customized to fit your machine requirements. The motors come in a standard 8-lead configuration with three wires for the phases and five wires for the hall sensors. We can also customize the windings to perfectly match your voltage, current, and maximum operating speed. Special shaft modifications, cables and connectors are also available upon request.

SPECIFICATIONS

Model #	Rated Voltage (V)	Rated Speed (RPM)	Rated Power (W)	Peak Torque (oz-in)	Rated Current (A)	Line to Line Resistance (ohms)	Line to Line Inductance (mH)	Torque Constant (oz-in/A)	Back EMF Voltage (VkRPM)	Rotor Inertia (oz-in-sec^2)	Weight (lbs)	"L" Length (in)
BLY171S-15V-8000	15	8000	26	14	2.2	0.35	0.35	1.98	1.14	0.00034	0.66	1.59
BLY171S-17V-8000	17	8000	42	21	3.6	0.20	0.26	1.98	1.14	0.00034	0.66	1.59
BLY171S-24V-4000	24	4000	26	27	1.8	1.50	2.10	4.96	2.45	0.00034	0.66	1.59
BLY172S-17V-9500	17	9500	70	30	6.4	0.09	0.09	1.56	0.90	0.00068	0.99	2.37
BLY172S-24V-2000	24	2000	41	63.72	3.6	1.60	2.70	8.78	5.00	0.00068	0.99	2.37
BLY172S-24V-4000	24	4000	53	54	3.5	0.80	1.20	5.81	3.35	0.00068	0.99	2.37
BLY173S-24V-4000	24	4000	77	79	4.9	0.46	0.70	5.38	3.10	0.00102	1.43	3.19
BLY174S-24V-4000	24	4000	104	106	7.0	0.30	0.50	5.32	3.10	0.001359	1.76	3.95
BLY174S-24V-12000	24	2000	41	38.2	3.6	0.07	0.08	1.97	1.02	0.00068	0.99	2.37
BLY171D-17V-8000	17	8000	42	21	3.6	0.20	0.26	1.98	1.14	0.00034	0.66	1.59
BLY171D-24V-1400	24	1400	9.8	27	0.9	8.00	10.50	10.62	8.42	0.00034	0.66	1.59
BLY171D-24V-2800	24	2800	25	31	2.0	2.78	3.36	6.09	4.57	0.00034	0.66	1.59
BLY171D-24V-4000	24	4000	26	27	1.8	1.50	2.10	4.81	2.7	0.00034	0.66	1.59
BLY171D-24V-6000	24	6000	25	27	1.4	1.15	1.47	3.97	4.00	0.00034	0.66	1.59
BLY172D-24V-2000	24	2000	41	54	3.6	1.40	2.25	7.79	5.66	0.00068	0.99	2.37
BLY172D-24V-4000	24	4000	53	54	3.5	0.80	1.20	5.03	3.1	0.00068	0.99	2.37
BLY173D-24V-4000	24	4000	77	79	4.9	0.46	0.70	5.38	4.14	0.00102	1.43	3.19
BLY173D-160V-4000	160	4000	77	79	0.7	26.67	20.00	36.96	18.83	0.00102	1.43	3.19
BLY174D-24V-4000	24	4000	104	106	7.0	0.30	0.50	5.32	3.1	0.001359	1.76	3.95
BLY174D-24V-12000	24	12000	113	38	5.4	0.07	0.76	1.97	1.02	0.001359	1.76	3.95
BLY174S-24V-12000	24	12000	113	38	5.4	0.07	0.09	2.35	1.55	0.001359	1.76	3.95

Notes:

- Custom leadwires, cables, connectors, and windings are available upon request.

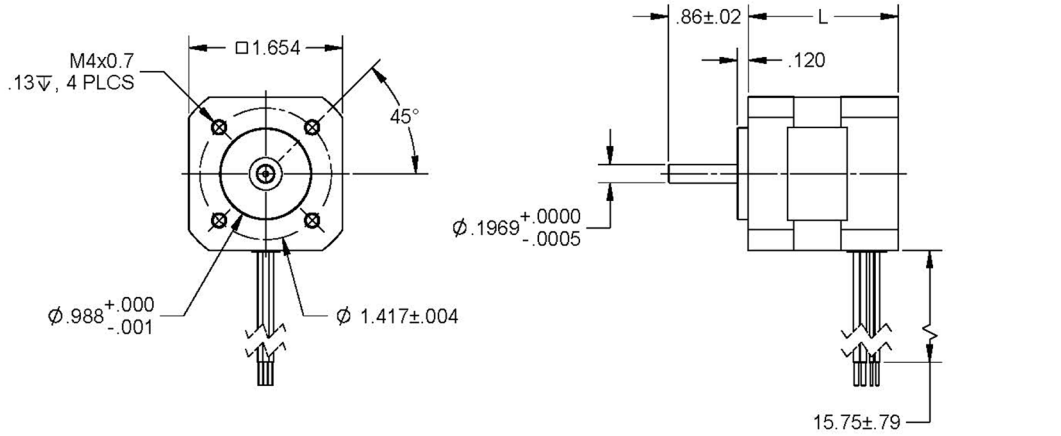
- The 7th character "S" denotes a single shaft, use "D" for double shaft.

- Dual Shaft motors have different mounting dimensions; please see drawings on next page.

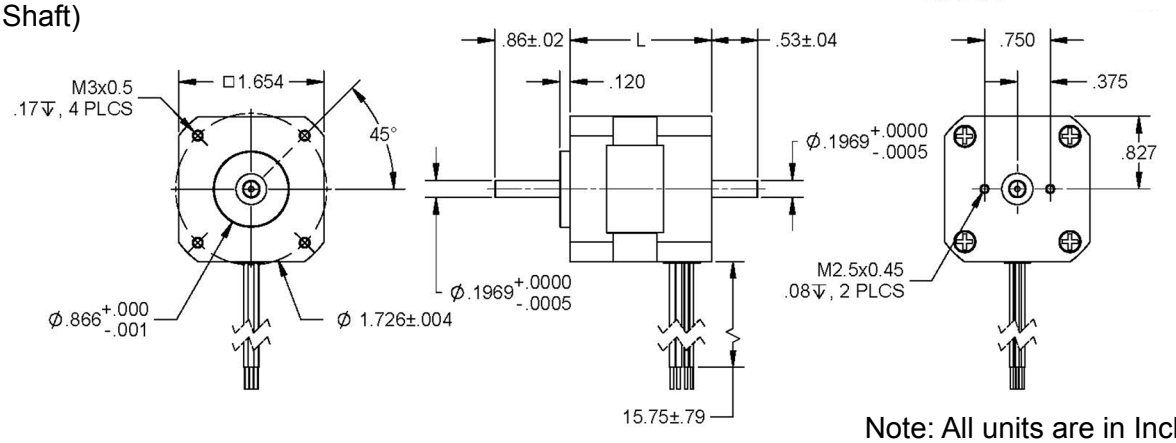
910 East Orangefair Ln. Anaheim, CA 92801 **Tel. (714) 992-6990** Fax. (714) 992-0471 **www.anaheimautomation.com**

DIMENSIONS

(Single Shaft)



(Double Shaft)

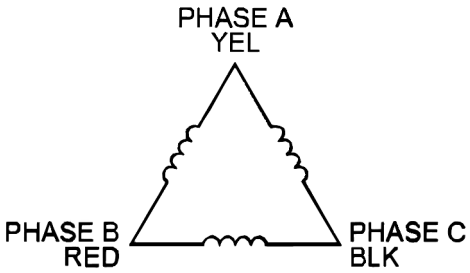


Note: All units are in Inches

Notes: Dual Shaft motors have different mounting dimensions.

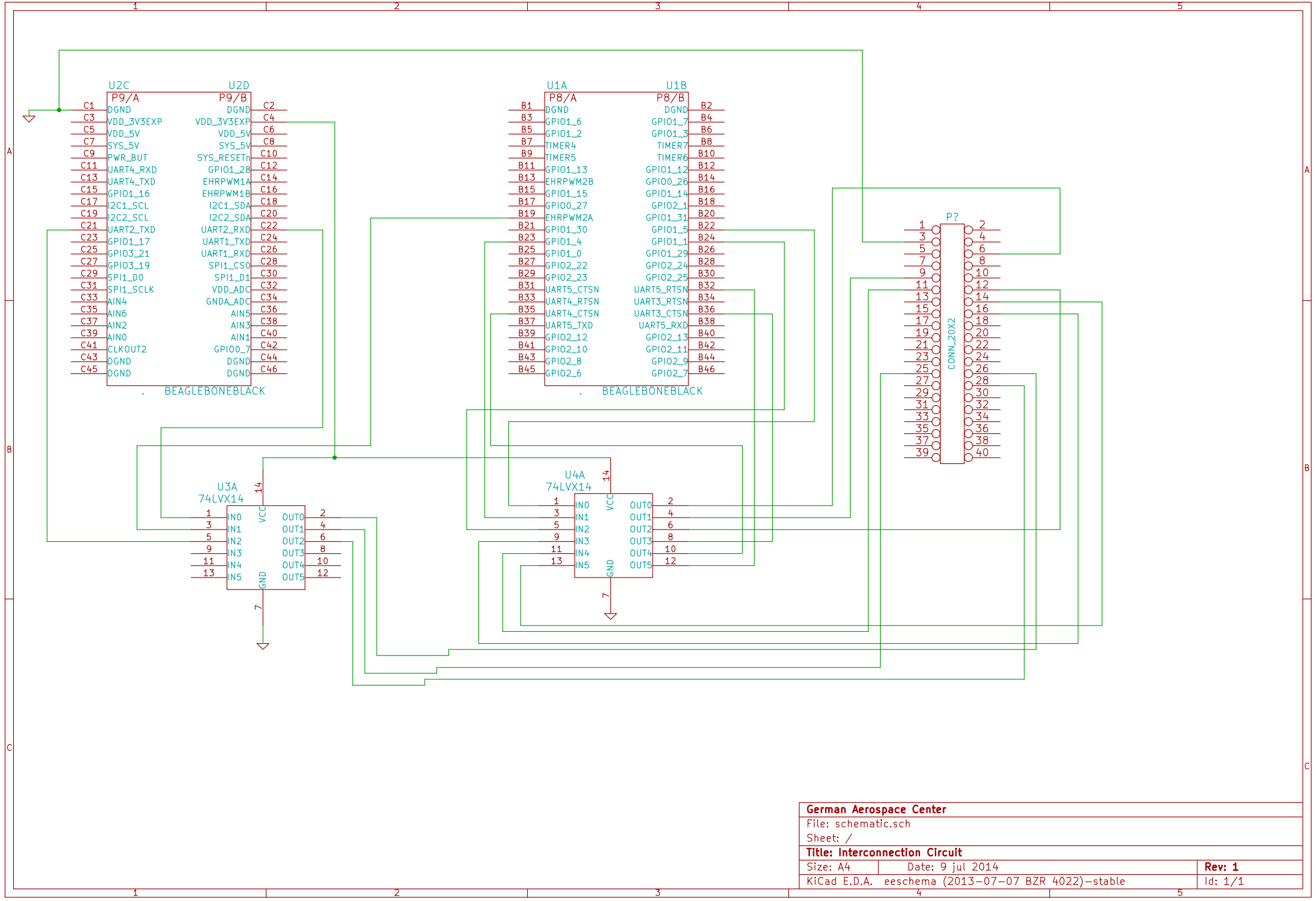
WIRING INFORMATIONS

Wire Color	Description
Red	Hall Supply
Blue	Hall A
Green	Hall B
White	Hall C
Black	Hall Ground
Yellow	Phase A
Red	Phase B
Black	Phase C



SPECIFICATIONS

Winding Type:	Delta, 8 Poles	Max. Radial Force:	28N @ 20mm from the Flange
Hall Effect Angle:	120 Degree Electrical Angle	Max. Axial Force:	10N
Shaft Run Out:	0.025mm	Insulation Class:	Class B
Radial Play:	0.02mm@450g	Dielectric Strength:	500VDC for one Minute
End Play:	0.08mm@450g	Insulation Resistance:	100MOhm, 500VDC



German Aerospace Center		
File: schematic.sch		
Sheet: /		
Title: Interconnection Circuit		
Size: A4	Date: 9 jul 2014	**Rev: 1**
KiCad E.D.A. eeschema (2013-07-07 BZR 4022)-stable		Id: 1/1