Politecnico di Torino

POLITECNICO DI TORINO

Master's degree in Aerospace Engineering

Master's Thesis

Development of control system for electro-mechanical actuator

Supervisors
Prof. Paolo Maggiore
Ing. Matteo D. L. Dalla Vedova
Ing. Roberto Grassi
Ing. Leonardo Baldo

Candidate
Andrea Cagnasso, 280377

October 2022

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Paolo Maggiore e l'Ing. Matteo Dalla Vedova, due presenze costanti durante il mio percorso universitario, sin dal primo anno.
Un ringraziamento particolare va all'Ing. Roberto Grassi e l'Ing. Leonardo Baldo, con i quali si è creato un legame ben più profondo di un semplice rapporto tra colleghi di lavoro.
In ultimo, ma non per importanza, desidero ringraziare la mia famiglia per avermi sempre aiutato e la mia fidanzata, Federica, per essermi stata accanto in tutti questi anni.

Abstract

The issue of environmental sustainability is becoming of primary importance also in aviation field. In this perspective, development of more electric aircraft concept could represent a valid solution, even if new technologies must be developed in future.

The work presented in the context of this master's thesis concerns the development of the control software for an electro-mechanical actuator and its subsequently validation on a test bench. The purpose of this actuator is to replace hydraulic piston actually mounted on braking system of general aviation aircraft.

The actuation group is constituted by a linear mechanical actuator driven by Faulhaber's brushless DC (BLDC) electric motor. The microprocessor unit is C2000 Delfino, while the driver is DRV8305, both from Texas Instruments. The software has been developed using Matlab/Simulink software environment, thanks to an external compiler compatible with C2000 processor.

Here, all mathematical computation is done with integer arithmetic, allowing to reduce microprocessor's workload.

This software must be capable of both low-level management of BLDC motor (controlling each driver's MOSFETs) and high-level management. Because there are no brushes in a BLDC motor, commutation must be reproduced electronically. In order to correctly select which phase must be turned on, the shaft position is sensed thanks to three different Hall digital sensors. An internal function generator allows to generate the reference force signal that feed the controller. Controller includes both an open loop and a closed loop branch. The open loop term allows to both bypass the need for external sensors in case of failure and to guarantee a non-zero phase current when the reference signal has been met, while the closed loop gives a contribution proportional to the error between the reference and the actual force signal. The force feedback is provided by an external load cell.

Analog-to-Digital Converters are used to measure different physical values, such as force and phase currents.

The program has been validated on a test bench specifically designed to make up for temporary lack of real mechanical actuator (still under construction). To safely perform the tests, different safety measures have been implemented. These measures also apply to the real brake architecture and include a limitation on the maximum reference force, a virtual end of stroke and a limitation on the maximum real load. This last limit is performed without the use of external load cell but directly sensing the shaft position.

All tests variable, such as function generator settings or controller parameters, can be set by user via serial communication. In the same way, it is possible to log some data useful to characterize the behaviour of the electro-mechanical actuator, allowing further analysis.

Results obtained at the end of this work are satisfactory and allow complete control of a single actuation group. Even if this code can be easily duplicated in order to control four different groups (the final configuration of the braking system), it can be suggested to directly write that code using a lower-level language, improving software efficiency.

Contents

Chapter 1

Introduction

1.1 Scope

While looking at aircraft, almost everyone see only wing, fuselage and tail. However, inside of those big aluminum sheets, it is plenty of other little parts like hydraulic pump, pipes, servovalve, control unit and so many others: the aircraft systems.

Aircraft System Architecture is a very complex subject: it requires to put together different technologies (hydraulic, pneumatic, electric ...), choosing, for every single task, which is the best in term of costs, mass, power consumption, maintainability and safety.

In order to reduce aircraft weight (or increase the payload mass) and the systems complexity, in the past few years a lot of research has been conducted. The future trend is to develop the so called *more electric aircraft*, in which there is no other system but electric one (and some few others). Despite of the problems that, with no doubt, will arise with this trend, the main advantages are noticeable: weight and complexity reduction, less different spare parts, easier power transportation across the aircraft, ...

This work is not a dissertation about the advantages of more electric aircraft compared to traditional aircraft architecture, but deals with the development of software to control mechanical actuator driven by a brushless DC motor (BLDC). The electro-mechanical actuator aim to replace hydraulic brake in general aviation aircraft.

One of the main advantage of this solution is to remove the need of pressurized hydraulic line from the aircraft body to the main landing gear, reducing hydraulic power needs and, hopefully, increasing safety.

In chapter 2 there is the general description of the brake architecture and the software environment. The whole software architecture is described in chapter 3, while chapter 4 and 5 contain the physical validation of this work.

1.2 Research program

This work only cover a little part of the whole design of an electro-mechanical actuator. Design of the general architecture and the actuator has been conducted under the European

Regional Development Fund (ERDF)[1] and PiTeF program of Regione Piemonte [2]

In this particular case, three mechanical actuators (see section 2.1.1) has been developed: one solution comes from Mecaer Aviation Group (MAG, the head of the project), the second one from Meccanica BPR and the third from Politecnico di Torino.

The next step after the software development is to test each solutions, in order to select the best one.

1.3 Requirements

General architecture of brake system, described in chapter 2, is complex and must meet a lot of safety and operational requirements. For now, it is enough to say that there are four actuator for each brake.

Even because of the limitation of the software environment, deeply described in chapters 3 and 6, the software developed for this master's thesis is able to control just one single motor-actuator group. However, this work is complete in itself and can be easily reproduced for all the four actuators groups.

Hereafter there is a list of high level requirements that has to be met to control the behaviour of each electro-mechanical actuator:

- The software shall be safe to use in aeronautical environment.
- The software shall guarantee a brake force, even after a failure of the load sensor.
- The control law shall be stable under external disturbances.
- The software shall work properly after brake wear out.
- The software shall be fault tolerant.
- The software shall collect useful data from each mechanical actuator.

[1] https://ec.europa.eu/info/funding-tenders/find-funding/eu-funding-programmes/european-regional-development-fund-erdf_en

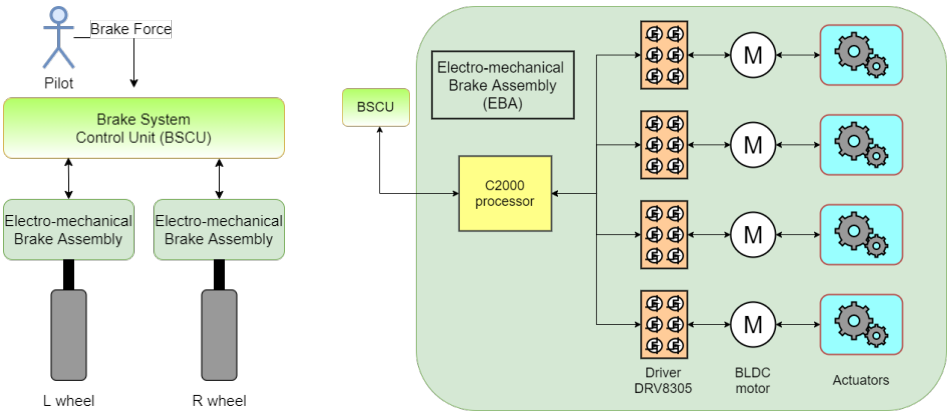
[2] https://www.regione.piemonte.it/web/temi/fondi-progetti-europei/fondo-europeo-sviluppo-regionale-fesr/ricerca-sviluppo-tecnologico-innovazione/piattaforma-tecnologica-filiera-pitef-0

Chapter 2

Hardware and software environment

The brake architecture, as proposed by Mecaer Aviation Group, is reported in figure 2.1. It is possible to see Brake System Control Unit (BSCU, figure 2.1a), which aim is to acquire the brake force imposed by pilot (through pedals) and equalize this input force among each actuator, both in left and right wheels. This BSCU must be capable of fault diagnostic, in order to evaluate when failures occur.

In each Electro-mechanical Brake Assembly (EBA, figure 2.1b) there are one microprocessor (Texas Instruments C2000™) and four mechanical actuators. Each actuator is driven by a Brushless DC (BLDC) motor, described in subsection 2.1.2, and a DRV8305 driver, described in subsection 2.1.5.



(a) *Brake System Architecture*

(b) *Electro-mechanical Brake Assembly (EBA)*

Figure 2.1: Brake system global architecture.

2.1 Hardware

2.1.1 Actuators

As mentioned before in section 1.2, three different mechanical actuators has been developed. None of those solutions will be presented here for different reasons. First of all, actuators development is not part of this master's thesis; furthermore, any superficial description brings no added value.

What is remarkable to write down here (see table 2.1) are the requirements that each actuator must satisfy, the gear ratio and the correspondence between mechanical actuator and BLDC motor.

Each actuator is equipped with a load cell in order to measure the real braking force. This signal will constitute the feedback for the control law (see section 3.6 on page 36).

Requirements	Solution A (BPR)	Solution B (MAG)	Solution C (PoliTo)
Total force (each actuator) [N]	2645	2645	2645
Bandwidth [Hz]	> 10	> 10	> 10
Clearance [µm]	125	125	125
Gear ratio	450	33.75	39.86
Selected motor	4221	3216	3216

Table 2.1: Mechanical actuators characteristics.

2.1.2 Electric motors

Selected motors are Faulhaber Brushless DC 3216W024BXTR and 4221G024BXTR. Both of these are 7 pole-pairs brushless DC motor with nominal voltage of 24 V, equipped with digital Hall sensors. Other characteristics can be found in table 2.2 and at Faulhaber's website[1].

Software described in chapter 3 applies to all actuators without modification (actually, some few variables can be modified, according to which motor will be used, like the ratio converting force into duty-cycle).

Electric motor is a device capable of converting electrical energy into mechanical energy (rotational motion) thanks to the interaction of magnetic forces repulsing and attracting each others. In the simples form, it is composed by two different parts: a rotating part, called *armature* or *rotor*, and a stationary part, the *field* or *stator*.

While current flows in a conductor, a magnetic field is created around it. If this magnetic field form a certain angle with another external magnetic field, a force arises.

If that conductor is not a simple wire piece, but has a particular shape, it is possible to create a net torque, as shown in figure 2.2. In this figure, current flows from point M to

[1] 3216 motor: https://www.faulhaber.com/it/prodotti/serie/3216bxtr/
4221 motor: https://www.faulhaber.com/it/prodotti/serie/4221bxtr/

BLDC motor	3216	4221
Rated torque [mN m]	41	134
Rated current [A] (thermal limit)	1.17	1.66
Rated speed [rpm]	4150	4390
Efficiency	0.82	0.88

Table 2.2: Faulhaber BLDC motors.

point N.

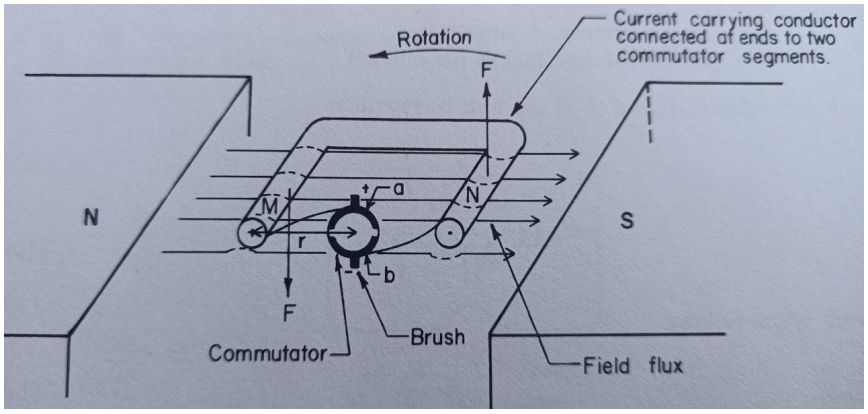


Figure 2.2: Motor action of coil rotating in magnetic field. Courtesy of [1].

Of course, this is not enough to create an electric motor: it is possible to demonstrate that this configuration of forces leads to a stable configuration with a minimum energy. To obtain a continuous rotational motion, a commutator must be added: the scope of this device is to change the current flow direction every 180° (or less in complex configuration with more than one pole pair). Usually, commutator is constituted by a pair of carbon brushes. Rotor shaft is composed by a certain number of section: brushes allows current to flow only in one section at time. After a certain angle, brushes touch another section, so current can flow in a different path (or in the same path, but in reverse direction).

Brushes are critical parts: they require replacement, they wear out, and they impose severe speed limitations. Moreover, arcing cannot be permitted in certain hazardous location (like inside fuel tanks).

One possible solution[2] is to move the coil from the rotor to the stator part, while putting the permanent magnet in the rotary part. This solution allows to completely remove the need of brushes, provided the availability of some electronic switching devices capable of replace the old commutator: here is a brushless DC motor.

The function of electronic switching devices is to switch the right currents in the right stator coils at the right time and in the right sequence. In order to do this, some position sensors, like encoder, resolver or digital Hall sensors, are needed.

[2]This is not the only one, but it is the one useful here to explain how a BLDC works.

A real example, the one used for this project, of how this type of circuit works is reported in subsection 2.1.5.

Although this type of motor can have an almost arbitrary number of phases, the most common configuration is the one with three phases.

The number of pole-pairs is selected taking in to account both costs (more pole-pair implies more acquisition cost, because of complexity) and cogging torque. In three-phase BLDC motor, but this concept applies to all electric motor types, output torque is not perfectly constant, both because of the time required to perform each commutation and the geometry of the rotor and the stator. If this could not be a great problem while the rotor is spinning at high speed, it can cause some problems during start-up or low-speed operation, because of equilibrium position and internal friction. One possible way to reduce this ripple is to choose a motor with an higher number of pole-pairs.

2.1.3 Texas Instruments' Piccolo Launchpad™

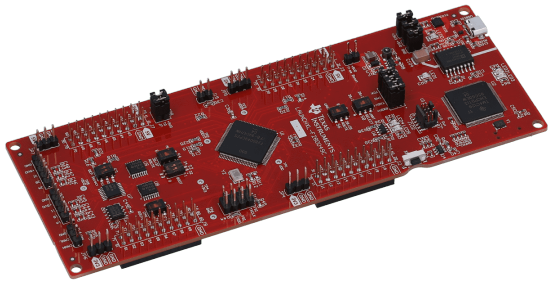


Figure 2.3: F280049C Piccolo Launchpad™.

C2000™ LAUNCHXL-F280049C (figure 2.3[3]) is a development board for Texas Instruments C2000™ Piccolo F28004x series of microcontrollers. This microcontroller is[4] a 32-bit floating-point type, with 256 KiB Flash memory, 100 KiB RAM and operates at 100 MHz (maximum value). There are up to 40 General-Purpose Input/Output (GPIO) pins and 21 analog input pins. These analog pins are connected to three 12 bit Analog-to-Digital Converters (ADCs) modules. In addition, there are also up to 16 ePWM channels and two Enhanced Quadrature Encoder Pulse (eQEP) modules.

This package provides two independent standard headers, allowing connection of up to two BoosterPack XL (subsection 2.1.5).

The USB communication area should be optically isolated from the Micro Controller Unit (MCU) area while using a BoosterPack XL. This can be achieved removing three jumper (JP1, JP2, JP3), as reported in datasheet.

Piccolo Launchpad™ is compatible with InstaSPIN-FOC™.

[3] https://www.ti.com/tool/LAUNCHXL-F280049C

[4] TMS320F28004x Microcontrollers datasheet (Rev. F)

2.1.4 Texas Instruments' Delfino Launchpad™

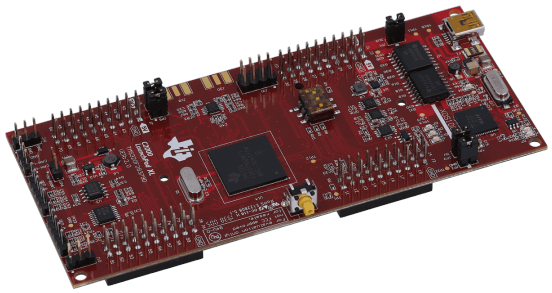


Figure 2.4: Delfino Launchpad™ F28379D.

C2000™LAUNCHXL-F28379D LaunchPad™ (figure 2.4[5]) is a development board for Texas Instruments Delfino™ F2837xD microcontrollers. The TMS320F2837xD is a dual core microcontroller, with 512 KiB Flash memory, 172 KiB RAM and operates at 200 MHz. It is equipped with a 32 bit single-precision Floating-Point Unit (FPU), a Trigonometric Math Unit (TMU) and Viterbi/Complex Math Unit (VCU-II). There are up to 169 GPIO and 4 Analog-to-Digital Converters (ADCs) with 12 bit resolution (that can be increased up to 16 bit), 24 Pulse Width Modulator (PWM) channels and 3 Enhanced Quadrature Encoder Pulse (eQEP) modules.

It is important to notice that not all channels are accessible at the same time. Probably because of space needs, some of them are muxed together, while few others do not have an external connection on this Launchpad™ (for example, it is possible to find only two of three eQEP channels on the board).

In the same way as Piccolo Launchpad™, Delfino Launchpad™ has been designed to be fully compatible with two BoosterpackXL module. So, even this Launchpad™ can be powered up from USB port or from Boostpack (to avoid short circuit, it is possible to optically separate MCU area from USB area).

In figure 2.5 there is the pinout of Delfino Launchpad™. It is possibile to switch from among different output thanks to an internal multiplexer. More informations can be found in the TMS320F2837xD Dual-Core Delfino™ Microcontrollers Data Manual at http://www.ti.com/lit/pdf/SPRS880.

2.1.5 Texas Instruments' DRV8305 driver

BoostXL DRV8305EVM BoosterPack (figure 2.6[6]) is a complete 3-phase driver stage allowing evaluation of motor application, thanks to DRV8305 motor gate driver. It support 4,4 V to 45 V power supply and up to 15 A RMS drive current. It is equipped with DC bus and motor phase voltage sense and each half bridge has a low side current shunt sense.

[5] https://www.ti.com/tool/LAUNCHXL-F28379D

[6] https://www.ti.com/tool/BOOSTXL-DRV8305EVM

Mux Value				J1 Pin	J3 Pin	Mux Value			
X	2	1	0			0	Alt Function	2	X
			3.3V	1	21	5V			
			GPIO32	2	22	GND			
	SCIRXDB		GPIO19	3	23	ADCIN14	CMPIN4P		
	SCITXDB		GPIO18	4	24	ADCINC3	CMPIN6N		
			GPIO67	5	25	ADCINB3	CMPIN3N		
			GPIO111	6	26	ADCINA3	CMPIN1N		
SPICLKA(1)			GPIO60	7	27	ADCINC2	CMPIN6P		
			GPIO22	8	28	ADCINB2	CMPIN3P		
		SCLA	GPIO105(2)	9	29	ADCINA2	CMPIN1P		
		SDAA	GPIO104(2)	10	30	ADCINA0	DACOUTA		

Mux Value				J4 Pin	J2 Pin	Mux Value			
X	2	1	0			0	1	2	X
		EPWM1A	GPIO0	40	20	GND			
		EPWM1B	GPIO1	39	19	GPIO61			
		EPWM2A	GPIO2	38	18	GPIO123			SD1_C1(1)
		EPWM2B	GPIO3	37	17	GPIO122			SD1_D1(1)
		EPWM3A	GPIO4	36	16	RST			
		EPWM3B	GPIO5	35	15	GPIO58			SPISIMOA(1)
		OUTPUTXBAR1	GPIO24	34	14	GPIO59			SPISOMIA(1)
OUTPUTXBAR7(1)			GPIO16	33	13	GPIO124			SD1_D2(1)
			DAC1	32	12	GPIO125			SD1_C2(1)
			DAC2	31	11	GPIO29(2)			OUTPUTXBAR6(1)

Figure 2.5: F28379D Delfino LaunchPad pin out and mux options.

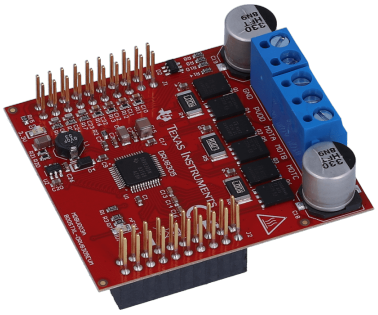


Figure 2.6: BOOSTXL-DRV8305EVM.

Its design is fully compatible with both Launchpad™ Piccolo and Delfino, in order to create a complete motor control platform.

The DRV8305 is a gate driver IC for three-phase motor driver application. It contain three half-bridge drivers, charge pumps, three current shunt amplifiers and a variety of protection circuit (like overcurrent, overtemperature, overvoltage, and undervoltage protection).

A general scheme for a simple drive control of a BLDC electric motor is reported in figure 2.7. It is possible to see six MOSFET (named $Q_1 \ldots Q_6$). The MOSFET Q_1, Q_2 and Q_3 constitute the high side of the bridge, the side directly connected to V_{DCbus}, while the remaining form the bridge low side. Each half bride (constituted by one high side and one low side MOSFET) controls the corresponding phase motor (named A, B and C): e.g.

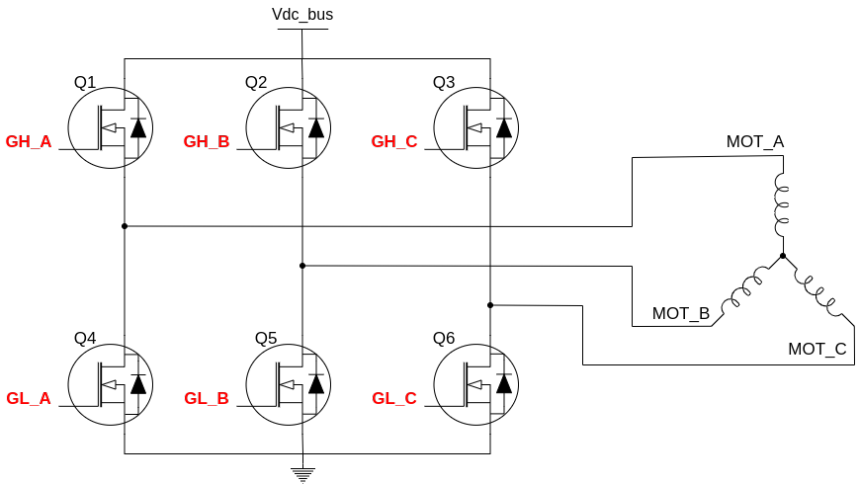


Figure 2.7: Three half bridge circuits for three phase driver.

if Q_1 and Q_5 are on conduction state and the other MOSFET are on interdiction state, phase A is connected to V_{DCbus}, while phase B is connected to GND, so current flows from MOT A to MOT B. Giving the correct switching sequence to GHx and GLx MOSFET gate allows the motor to spin.

The real half bridge scheme, the one mounted on BoostXL DRV8305EVM BoosterPack, is reported in figure 2.8. Here, it is possible to see more electronic parts: R_4, R_5 and R_6 are the shunt resistors for current measure, while R_{11}, R_{12} and R_{13} are three 4,99 kΩ resistors for back-EMF sense. For explanation of how back-EMF and current measures work, see section 3.3 on page 28.

Because of the high angular speed typical of an electric motor and the need of a punctual control of the motor, usually the signals that drive the gates are Pulse-Width-Modulation (PWM) signals.

PWM is a method of reducing the average power delivered by an electrical signal when there are only two possible states of power: ON and OFF. The output is a square wave with a certain frequency and a certain duty cycle. The duty cycle is defined as the ratio between the ON time and the period of the wave. This signal is created comparing a carrier wave (the blue one in figure 2.9), which give also the frequency of the signal (usually from above 5 kHz up to 20 kHz for motor control operations), with a certain threshold (red line), that give the duty cycle (or its complementary).

Varying the duty cycle, it is possible to change the mean value of the PWM signal, that can range from 0 to the maximum voltage available. Thank to its high frequency (compared with any mechanical constant), while the MOSFET can follow this fast switching, the electric motor see only the mean value. This allow to modify almost continuously the mean phase voltage value, even without a resistive divider, which will cause a lot of energy dissipation.

The major drawback (but it could also be seen as a strength) of using this technique is that PWM generation is not based on software, but it rely on electronic components. For this reason, the number of different PWM signal available in a single board is often limited to few units.

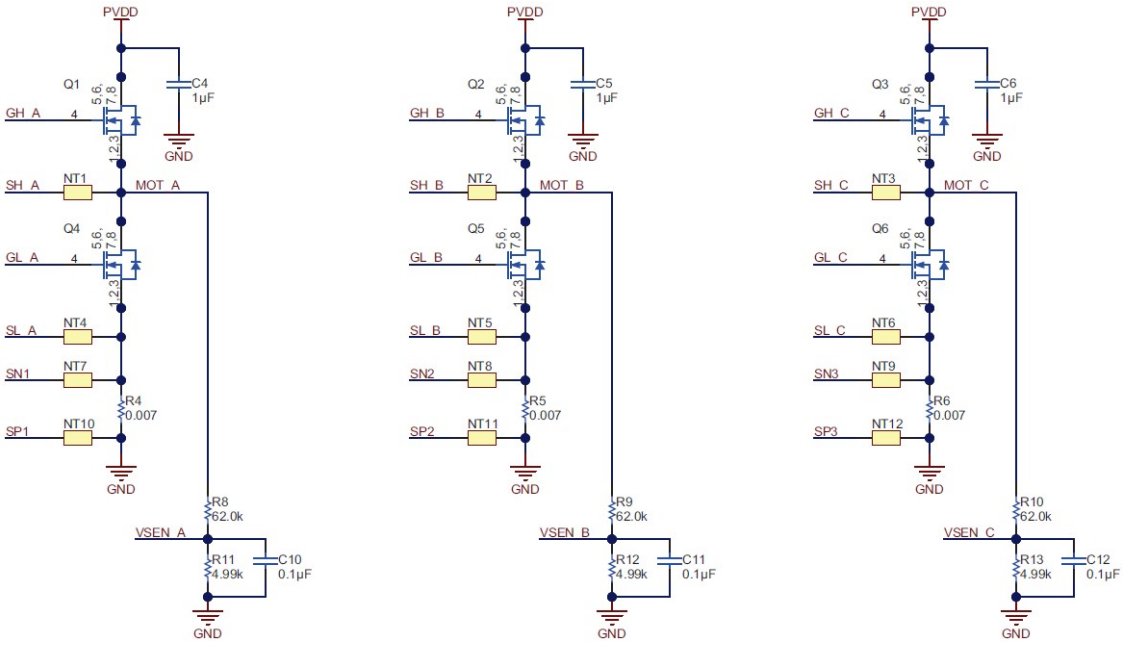


Figure 2.8: Real half bridge scheme, mounted on BoostXL DRV8305EVM BoosterPack.

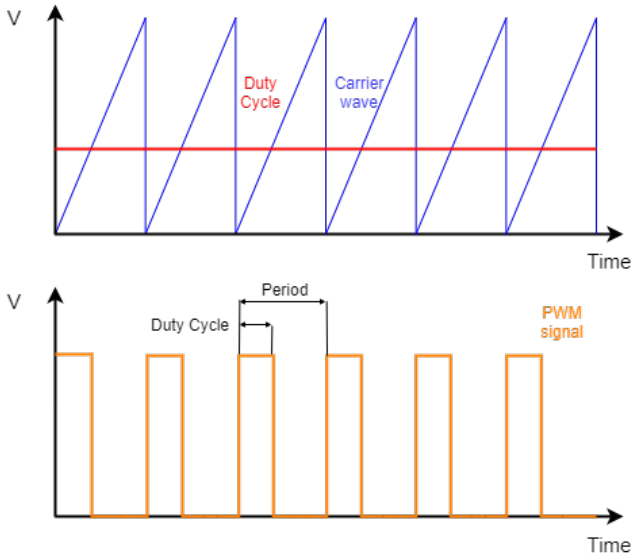


Figure 2.9: PWM generation.

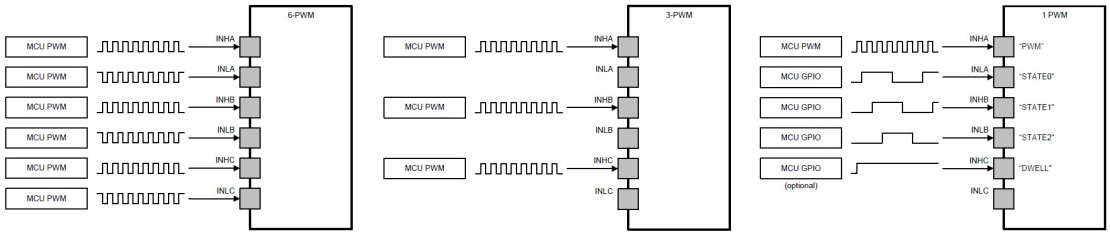
The architecture proposed by MAG (figure 2.1) includes a single microcontroller that control 4 different motor, requiring at least 24 different PWM signals. This is not compatible with Delfino LaunchPad™ so a new solution has to be found.

DRV8305 can supply to this drawback: it accepts three different input modes, in order to support various commutation schemes: 6-PWM mode, 3-PWM mode and 1-PWM mode

(figure 2.10). The input mode can be selected writing the correct value inside the SPI register of the driver, sending:

- 0x3A16 for 6-PWM mode (decimal 14870);
- 0x3A96 for 3-PWM mode (decimal 14998);
- 0x3B16 for 1-PWM mode (decimal 15126).

SPI communication also allows to investigate error causes and set the proper current shunt amplifiers gain.



(a) *6-PWM mode* (b) *3-PWM mode* (c) *1-PWM mode*

Figure 2.10: Input modes for DRV8305 IC.

The 6-PWM mode (figure 2.10a) is the default settings and allows, for each half bridge, to be placed in one of three states, either High, Low or Hi-Z, based on the input. This allow a direct control on each single MOSFET.

The 3-PWM mode (figure 2.10b) allows for each half bridge to be placed in one of two states: either High or Low. Only the three high-sides input (INHx) are used and the complementary signal is internally generated. Any activity on low-sides input pin (INLx) is ignored.

The 1-PWM mode (figure 2.10c), the one used for this project, allow to fully control three half bridge through a single PWM source and a 6-step internally stored block commutation table. According to the logical value on pins INLA, INHB and INLB, the driver redirect the PWM signal, pin INHA input, to the corresponding half bridge. The INHC can be used to facilitate the insertion of dwell states, or phase current overlap states between the commutation steps. For the complete switching sequence, including dwell states too, see the driver datasheet.

The truth tables are reported in table 2.3 if active freewheling is used, and in table 2.4 in case of diode freewheeling. Diode freewheeling is when the current is carried by the diode mounted anti-parallel to the MOSFET (see figure 2.7) while the MOSFET is reverse biased. In active freewheeling, when the power MOSFET is reverse biased, that MOSFET is enable: this allows the system to increase efficiency thanks to the lower impedance of the MOSFET conduction channel compared to the body diode. Which freewheeling technique to use can be set through SPI register. Here, active freewheeling is being used.

State	Input code	GHA	GLA	GHB	GLB	GHC	GLC
AB	0110	PWM	!PWM	LOW	HIGH	LOW	LOW
CB	0100	LOW	LOW	LOW	HIGH	PWM	!PWM
CA	1100	LOW	HIGH	LOW	LOW	PWM	!PWM
BA	1000	LOW	HIGH	PWM	!PWM	LOW	LOW
BC	1010	LOW	LOW	PWM	!PWM	LOW	HIGH
AC	0010	PWM	!PWM	LOW	LOW	LOW	HIGH
Align	1110	PWM	!PWM	LOW	HIGH	LOW	HIGH
Stop	0000	LOW	LOW	LOW	LOW	LOW	LOW

Table 2.3: Truth table for 1-PWM, active freewheeling.

State	Input code	GHA	GLA	GHB	GLB	GHC	GLC
AB	0110	PWM	LOW	LOW	HIGH	LOW	LOW
CB	0100	LOW	LOW	LOW	HIGH	PWM	LOW
CA	1100	LOW	HIGH	LOW	LOW	PWM	LOW
BA	1000	LOW	HIGH	PWM	LOW	LOW	LOW
BC	1010	LOW	LOW	PWM	LOW	LOW	HIGH
AC	0010	PWM	LOW	LOW	LOW	LOW	HIGH
Align	1110	PWM	LOW	LOW	HIGH	LOW	HIGH
Stop	0000	LOW	LOW	LOW	LOW	LOW	LOW

Table 2.4: Truth table for 1-PWM, diode freewheeling.

2.2 Software environment

2.2.1 Code Composer Studio™

Code Composer (CC) Studio™[7] is an Integrated Development Environment (IDE) that support all Texas Instruments' microcontrollers. This is the TI master software because allows the development of many different code for many different hardware in one single place, thanks to the integrated plug-ins. In fact, there is a large number of example that allows to spin a BLDC motor without almost any effort (except reading the documentation) with Piccolo LaunchPad™ and DRV8305 BoosterPack.

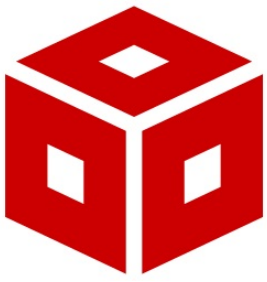


Figure 2.11: Code Composer Studio.

In figure 2.12 is depicted the framework structure provided by Texas Instruments. This framework, at least the part related to motor control, has been modified in 2018.

Before that year, inside of Code Composer Studio™ there was some different libraries and drivers like ControlSuite and

[7] https://www.ti.com/tool/CCSTUDIO

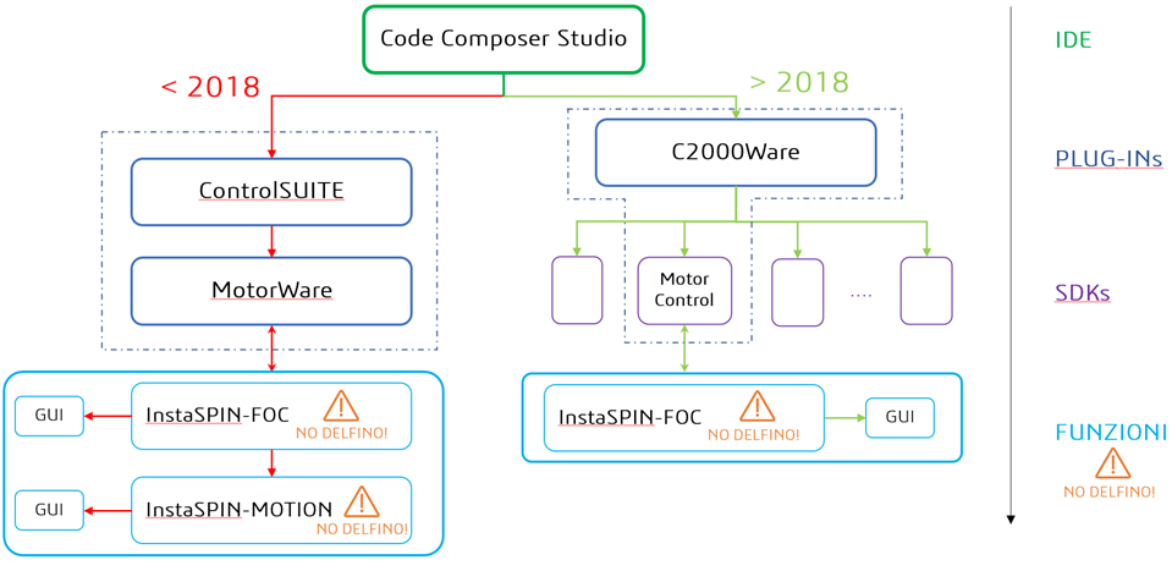


Figure 2.12: CC Studio™ software framework, before and after 2018. Thanks to Ing. Baldo.

MotorWare. Together with this solutions, there was InstaSpin FOC™ and InstaSPIN-MOTION™.

InstaSPIN-FOC™ is a complete sensorless FOC (Field-Oriented Control) solution provided by TI, allow efficient motor control without the use of any mechanical rotor sensors.

InstaSPIN-MOTION™ is a comprehensive sensorless or sensored FOC solution for motor-, motion-, speed- and position-control.

Actually, ControlSuite and MotorWare have been replaced by C2000ware[8], a cohesive set of software and documentation that includes device-specific drivers, libraries, peripheral examples, hardware design schematics and documentations. Inside of C2000ware environment, it is possible to install different Software Development Kit (SDK). One of this SDK is MotorControl, including InstaSpin FOC software.

The new version of Piccolo LaunchPad™ is compatible with the newest software (i.e. C2000ware), while the oldest version (not described here), supported the pre-2018 software.

Unfortunately, there are no ready-to-use examples compatible with Delfino LaunchPad™.

2.2.2 Simulink

Simulink[9] is a block diagram software environment created by Mathworks. It is a useful tool to model, simulate and analyze different type of systems, providing an user-friendly interface.

With its large amount of packages supported, it can be helpful to numerically solve differential equations (both stiff and non stiff problems),

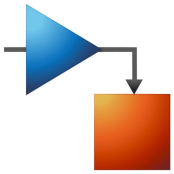


Figure 2.13: Simulink.

[8] https://www.ti.com/tool/C2000WARE

[9] https://www.mathworks.com/products/simulink.html

simulate control loop and autonomous systems, wireless communications and even develop Artificial Intelligence solutions.

Simulink is also capable of generating C/C++ code to be deployed in microcontroller. Thanks to `Embedded Coder Support Package for Texas Instruments C2000 Processor` and to `SoC Blockset Support Package for Texas Instruments C2000 Processors`, it is possible to convert a Simulink block diagram into C code, build and deploy it on C2000 Processors (in any case, Code Composer Studio™ must be previously installed).

These packages include also two different running modes:

Monitor&Tune This mode allows to run the software and to monitor and modify in real time some variables through serial communication. This mode is quite similar to standard simulation in Simulink. It is also called *external mode*.

Build, Deploy & Start This mode generate the C code and build it to the target, generating a stand-alone program. This mode replicates the behavior of Code Composer Studio™.

Both running modes have their advantages and drawbacks. Monitor&Tune allows to easy modify the code[10] without regenerating the whole code (this is an awesome fact, considering that the code generation process and the deployment on the board can last for up to 4 min, with both Monitor&Tune and Build, Deploy & Start). On the contrary, this mode impose a severe slow down of code execution, because of the large amount of data transmitted through serial port, causing difficulties on real time software monitoring. In addition, only for Matlab version previous than R2021, the length of each data must be greater than 8 bit. This means that if you want to reduce memory usage by using a boolean value to set to High a digital output pin, you will get an error saying that it is not possible to start the external mode. This bug has been resolved in Matlab R2021 release.

On the other side, Build, Deploy and Start makes the code run faster, but data logging is not possible (unless a serial communication is set, see section 3.4 on page 31).

For the development of the software described in chapter 3 both run mode has been used. However, in that chapter only the version compatible with Build, Deploy & Start is deeply described because that is the most complete version and the one with better performances.

The Matlab version used here is R2020b.

2.3 Trade off

Because of stringent time requirement imposed by MAG and the lack of examples, directly development of C language code for Delfino LaunchPad™ and BoosterPack DRV8305 was not a viable solution (the presumed development time was more than 6 months, while the available time was less than 2 months). For this reason, a number of alternative solutions, summarized in table 2.5 has been proposed to start the software development and meet the

[10] Only minor changes are allowed, like modification of constant values. No change in block diagram are permitted.

time requirement. Given that the global hardware architecture cannot be change, pursuing a solution similar to the definitive one is probably the best choice to reduce development time.

	Solution A	Solution B	Solution C
LaunchPad™	Piccolo F280049C	N/A	Delfino F28379D
Driver	DRV8320	Fahulaber's driver	DRV8305
InstaSPIN™ enabled	Yes	N/A	No
Available examples	Yes	Yes	No
Presumed time required to:			
- spin one motor	1 month	2 weeks	$>$ 6 months
- control one motor	+1/+2 months	+1/+1.5 month	+1/+2 months

Table 2.5: Different solutions proposed.

One possible solution is to migrate to Piccolo LaunchPad™ F280049C and BoosterPack DRV8320. This is a slight different architecture with little less performance, but with a lot of community support and working examples, with a presumed development time of less than one month to make a motor to spin.

Some tests have been made with this architecture and, following the available examples for InstaSPIN™, in about one month it has been achieved:

- Motor parameter identification;
- Speed control loop;
- Torque control loop.

Even if these results do not totally meet the target software requirements, they are encouraging.

Another solution is to use Faulhaber proprietary driver: even if this is the quickest solution, it is the most expensive one. Furthermore, this way is totally different from what has to be done at the end.

In conclusion, MAG decides that even minor architecture changes are not possible at this point of the project. In order to reduce development time, they suggested to use Simulink instead of Code Composer to write code.

Chapter 3

Software development

This chapter describes the development and the global structure of the last version of the software needed to control one single electro-mechanical actuator.

The target hardware is Delfino F28379D Launchpad™ (subsection 2.1.4), with BoostXL DRV8305EVM BoosterPack (subsection 2.1.5). This software works with both type of motor, after changing few parameters in the `init.m` file.

Three files are needed to run the software: `init.m`, the Matlab code that contain some variables definition, `Host_Log_Dati.slx`, a Simulink mode that can be used to control the motor and log some useful data via serial communication, and `Target200.slx`, the core of the software.

The `init.m` file allow to apply the correct settings to the serial communication (only host side, see section 3.4), like baud rate and communication timing, to set frequency and other time specification on target device and to select the motor that will be used.

Some of this variables, for example, `baud` at line 7, are not used, but are here just as a reminder. This is because some blocks of `Embedded Coder Support Package for Texas Instruments C2000 Processor` and `SoC Blockset Support Package for Texas Instruments C2000 Processors` and the Hardware Implementation section of the Configuration Parameters menu in Simulink do not allow the use of variable inside numerical fields (in opposition to classical Simulink behaviour). Not knowing this particular annoying aspect is a loss of time because no errors rise while compiling. Simply, after deploy, the software works only partially, apparently without any logical reason.

For this and other reasons that will be described later in this chapter, Simulink is not the best environment to develop this type of code. So, here only the software to control just one single motor has been developed. Further details can be found in chapter 6.

During this work, a number of different versions has been developed: some of them are only for test purpose, while other implemented some different features, like a slider to directly control the duty cycle, without the need of serial communication,s or a simplified model to make the first test on real actuator. These versions are not described here because there are only minimal differences among them.

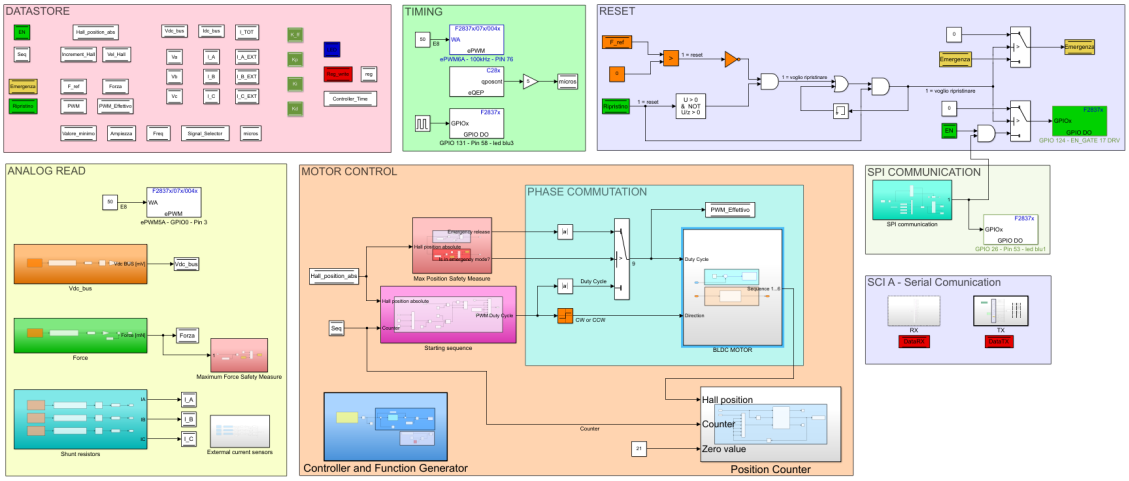


Figure 3.1: `Target200.slx` main view

3.1 Target file

The `Target200.slx` file is shown in figure 3.1.

The main parts, whose description is the main topic of this chapter, are listed below:

Data Store A place to put all Data Store Memory block.

Timing Manually computation of elapsed time (section 3.5).

Reset Recovery procedure after emergency stop (subsection 3.8.2).

Analog read Analog data acquisition (section 3.3).

Motor control The core of the software; includes

- Phase commutation (section 3.2)
- Controller and function generator (section 3.6 and section 3.7)
- Safety measures (section 3.8)
- Initial reset routine, or starting sequence (section 3.9)
- Position counter (subsection 3.2.1)

SPI communication BoostXL DRV8305's settings are sent via SPI communication (section 2.1.5 and subsection 3.2.2).

SCI A - Serial communication It contain all what is needed to perform serial communication with an host computer (section 3.10 for the host file and section 3.4 for target file).

Even if the Delfino LaunchPad™ is equipped with floating point unit (FPU), it has been decided to use only `int` data type. If using of real data type simplify the code (e.g. see sinus wave computation in section 3.7), the required time for computation

is not deterministic and will increase the microprocessor workload. The workload is a fundamental aspect because Hall sensors must be read synchronously with the program execution. Usually, when an electronic board should drive a BLDC motor, the Hall sensors are read using Interrupt Service Routine (ISR). ISR allows to stop the sequential execution of the program and do something only when a change is detected (i.e. change in hall state causes phase current to switch from one phase to another). This guarantees a correct execution of some critical time-related tasks. Because of Delfino can natively support up to two different motors but MAG architecture require four of them, there are not enough interrupt available. This lack of interrupt connection can be override by polling reading. For further details, see section 3.2.

In this software, there are many different blocks, some of them defined as Atomic blocks. An Atomic block is a code portion which execution time is different from that of the main code. Examples of these type of blocks are the Controller and the block sending and receiving data via serial communication (SCI A – Serial communication, RX and TX blocks).

To handle different sample time, usually Simulink's Rate Transition block work well, but some problems arises while using this type of block with Delfino LaunchPad™. For this reason and also because of coding needs (e.g. creating a counter), Memory Data Store block are used instead. These blocks act just like a declaration and initialization of a variable in C language: they allocate some memory slots for a variable with a certain name and a certain initial value. It is possible to read from this memory location or to overwrite what was in memory using two other different blocks. Thanks to its asynchronously execution (they can be read or written at any point of the code), they are not affected by sample time problems.

3.2 Switching sequence

In figure 3.5 on page 26 there are different zoom level of the blocks required to perform a low level control of a BLDC motor.

Looking at figure 3.5a, it is possible to see that the duty cycle value is split in its absolute value and its sign. This allow to control not only the magnitude of the duty cycle, but also the spin direction.

Figure 3.5b shows the PWM setting and the Phase Commutation block. In order to set a proper duty cycle value, the Simulink block for PWM's duty cycle settings requires its complement to 100. Obviously, the duty cycle value must be positive and limited in range from 0 to 100. The PWM signal selected inside this block is ePWM1A; the output is GPIO 0, on pin 3, the one corresponding to DRV8305's INHA pin.

The phase commutation block (figure 3.5c) is in charge to continuously decide in which phase current must flow, depending on motor alignment. The input of this block is duty cycle's sign. In order to make the motor to spin in a good way, an offset of three position in the switching sequence is required. The blocks shown in figure 3.2 allow to correctly handle a sign input (i.e. ± 1) and transform in it to $+3$ and 0, respectively for forward and reverse spin direction. Forward means that mechanical actuator connected to the motor press brake disks.

Shaft position is acquired inside Hall position block (figure 3.3). The values read from

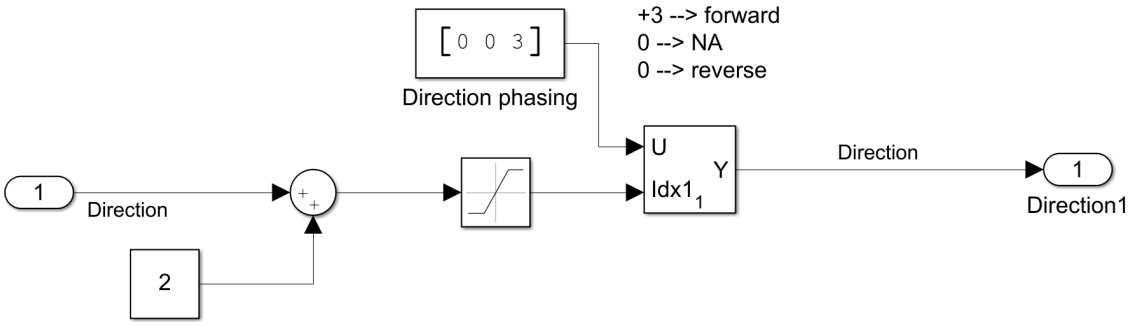


Figure 3.2: Direction management.

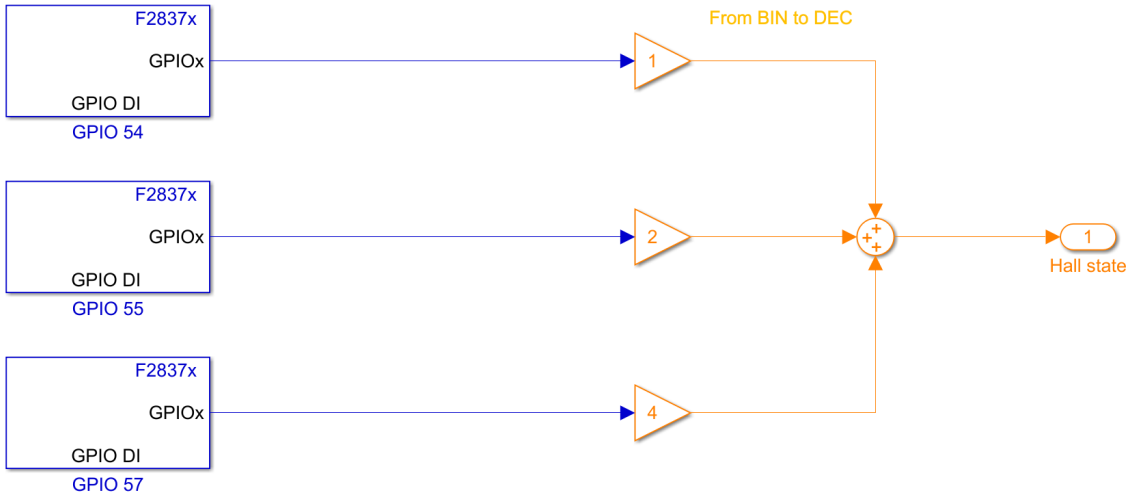


Figure 3.3: Hall sensor reading.

GPIO 54, GPIO 55 and GPIO 57 can be codified into three bit signal. This allow to obtain a numerical sequence that can be directly related to the shaft position. Because the selected motors are of 7 pole-pairs type, this sequence (figure 3.4) will repeat 7 times per revolution.

The output sequence that appear in figure 3.4, after binary-to-decimal conversion, is

$$6 \quad 4 \quad 5 \quad 1 \quad 3 \quad 2$$

and it is store in the red Hall sequence block on the left of figure 3.5c

This sequence is directly related with wire connection of the Hall sensors on the Delfino Launchpad:

- Hall A, green cable, on GPIO 54;
- Hall B, blu cable, on GPIO 55;
- Hall C, grey cable, on GPIO 57.

It is possible to change which GPIO to use by simply modifying it inside of the blue block of figure 3.3.

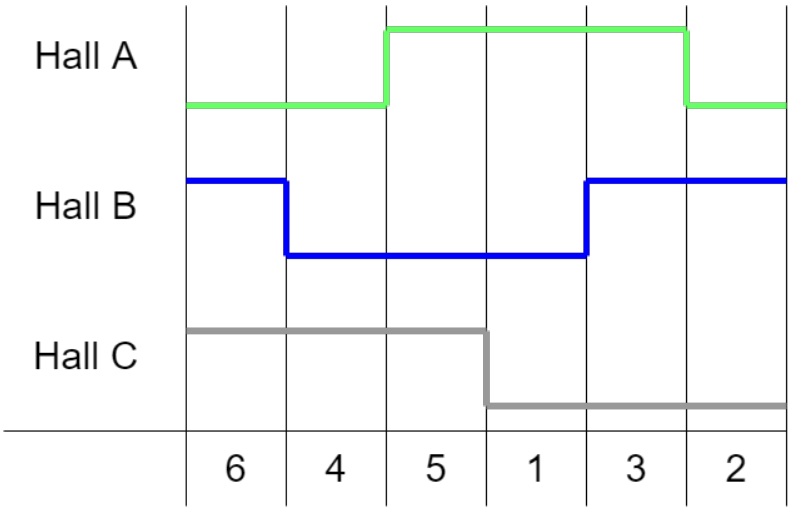


Figure 3.4: Hall sensors signals.

The Hall sequence pass through a look-up table, which allows to generate a sequence from 1 to 6 or from 6 to 1, depending on the spin direction. This is also the output of BLDC MOTOR block. The use of a look-up table allows to change the motor connection (for example, connecting another BLDC motor from another manufacturer) without modify the code inside command block.

Command block accept a sequence from 1 to 6, the spin direction phasing and, eventually, a manual phasing and output a number related to which phase has to be activate. The subsequent look-up table converts this number in the code that has to be sent to DRV8305, according to table 2.3 on page 16.

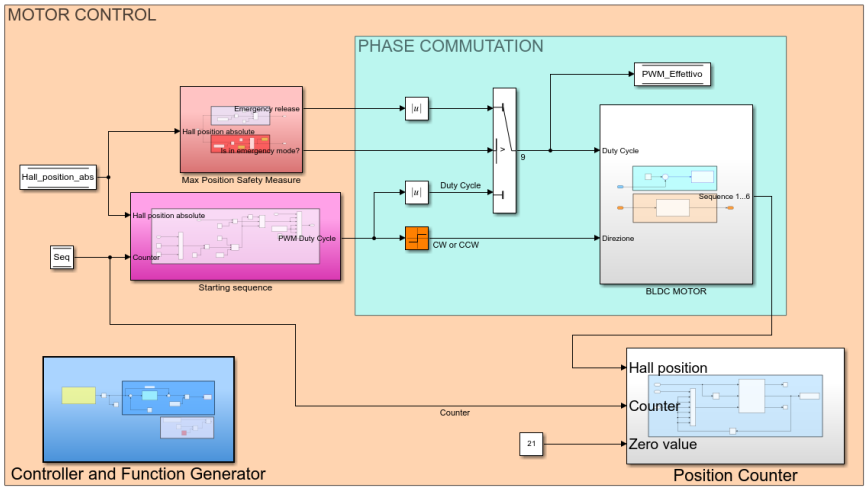
If the three input are `hall_state` for Hall state (a number from 1 to 6), `dir` for the spin direction (0 or +3) and `ph` for manual phasing (an integer value defined in the `init.m` file and, for now, set to 0), the formula implemented in this block is

$$output = (hall_{state} - 1 + dir + ph + 6) \bmod 6 + 1 \tag{3.1}$$

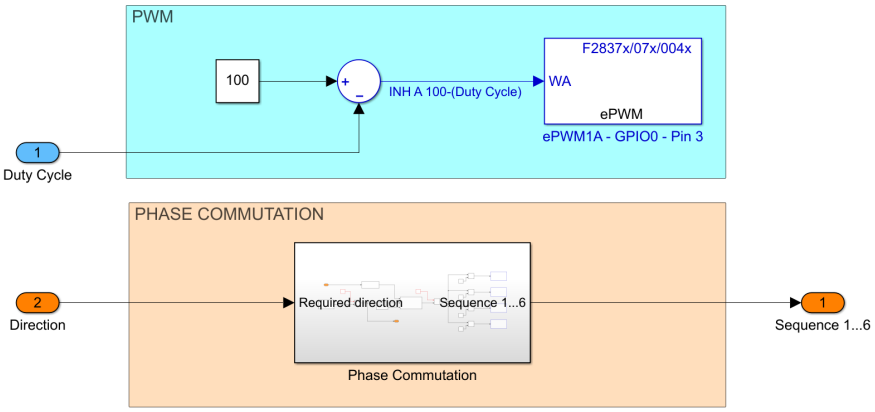
After the last look-up table (which output is a decimal number), there are four Bitwise AND block that split this number in order to obtain four single bit line. These values are then written on GPIO 1, GPIO 2, GPIO 3 and GPIO 4.

3.2.1 Position counter

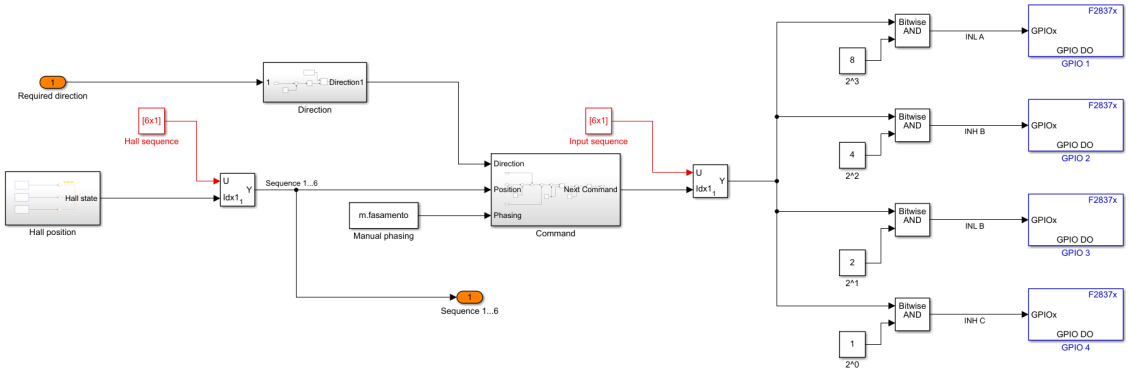
On figure 3.5a, it is possible to notice that the BLDC MOTOR block output is one of the input of the Position Counter block. The other input are the starting sequence counter (see section 3.9 and the zero value. This block computes the number of changes occurring in the hall signal. From this, it is possible to compute the angular position of the shaft multiplying it by $360°/(7 * 6)$, where 7 is the number of pole-pair and 6 are the possible different Hall state (the total step per revolution are 42). The output of this block is the number of step and not the corresponding angular position because an angular value (both degrees or radian) will not be an integer, requiring the use of floating-point unit.



(a) *Motor control blocks*



(b) *BLDC Motor block*



(c) *Phase commutation block*

Figure 3.5: BLDC control blocks.

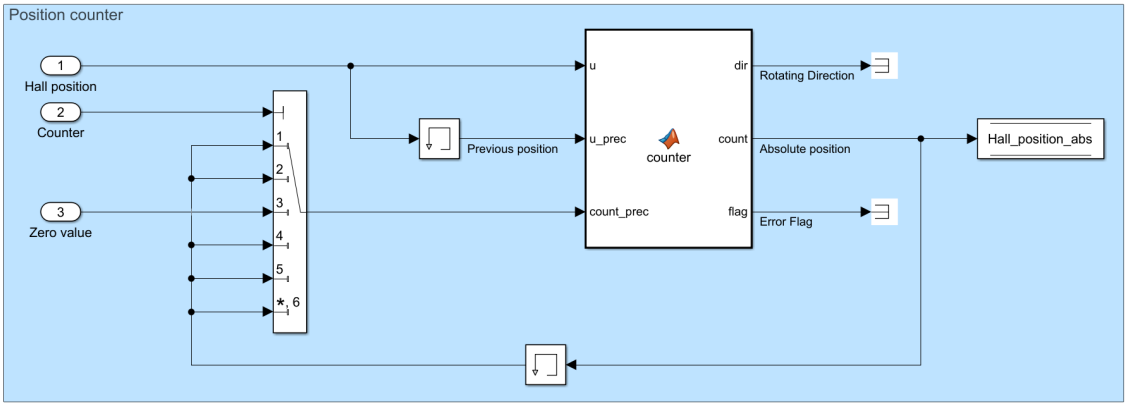


Figure 3.6: Position counter block.

The Matlab function inside of this block add $+1$, 0 or -1 to the previous step number depending on the spin direction (if spin occurs). Flow chart in figure 3.7 explains this process. The counter is needed to properly set the zero value (set to half revolution, i.e. 21 points).

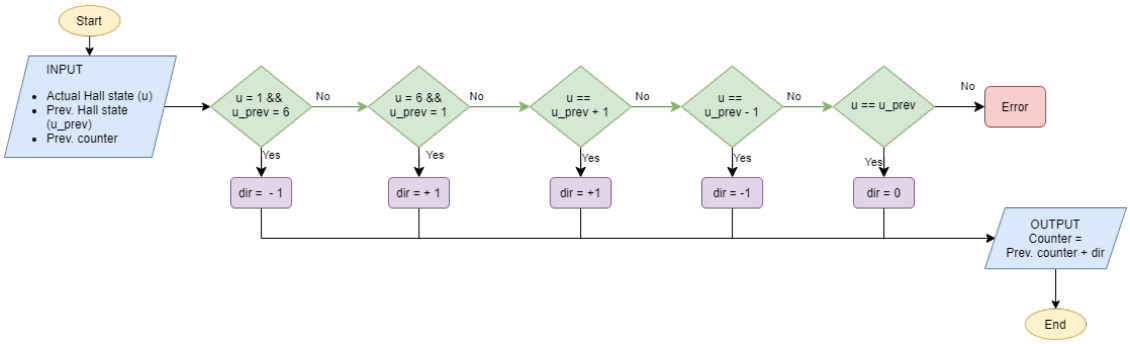


Figure 3.7: Flow chart for position counter block.

3.2.2 SPI communication

The SPI communication is required to properly set the DRV8305 motor driver. The block SPI Master Transfer, included in Simulink plug-in for Delfino Launchpad™, accept an `uint16` input and output the value read or write to the corresponding register. This very simple working mode hides a major drawback: probably because of Simulink nature, this block runs during *all* simulation time. It means for each time step, an SPI reading/writing operation is performed. This cause an overload of the LaunchPad™ and it is one of the cause of the timing problems explained in section 3.5.

To fit this problem, a more complex structure must be implemented (figure 3.8). At the start of program execution, this block diagram allows to write up to three different values on SPI bus (but it is easy to increase that number), thanks to a counter that increases its value each second. This counter feeds two switch blocks: when counter is less than 4, the

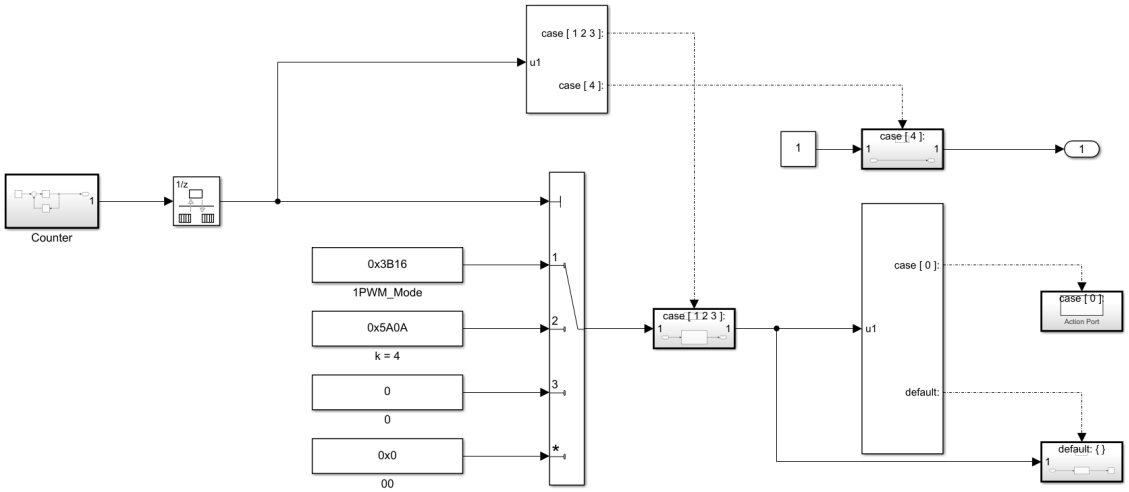


Figure 3.8: SPI communication management.

SPI block is activate and the correct value is written. In all other cases, when counter becomes equal or greater than 4, SPI block is deactivated and the global output is set to 1[1].

One of the code that must be writte on SPI register is 0x3B16, the value that set 1-PWM mode on DRV8305.

The output of this block is in AND combination with the EN signal (enable signal, sent from Host via USB communication). This stops the driver until the correct way of work has been set.

Because of the logical simplicity of the task that must be accomplished, not justifying the use of a Data Store Memory block, here and only here a Rate Transition block has been used.

3.3 Analog read

One of the main characteristics of an analog signal is that it can vary continuously in time. It is opposed to discrete signal, which can have just two different value, high and low. One of the most common way to interface a microprocessor (a digital device) with the real world is via analog sensors. This type of sensors exploit the modification of some electrical characteristics (like resistance), caused by external agent, to generate an electric signal (voltage level). Analog-to-Digital Converters (ADCs) are devices that convert an analog input into a digital signal. Their resolution is expressed in bit. Here, the acquisition has been linked to a PWM signal (ePWM5) with a frequency of 10 kHz.

Delfino LaunchPad™ is equipped with 12 bit ADCs, so an analog signal is codified as a numerical value between 0 and $2^{12} - 1$.

[1]This value should be a boolean one, but because of Monitor&Tune bug it has been set to be an `int16` type.

The maximum voltage supported by ADCs is 3 V. Unfortunately, Delfino LaunchPad provide only 3,3 V or 5 V. For this reason, when a sensor is powered by the LaunchPad, some precautions has to be taken: one possible solution is to increase the sensor range, so that the real maximum value fall below 3 V; another one is to connect a resistor in series with the sensor (as in case of using a potentiometer), obtaining the proper voltage drop.

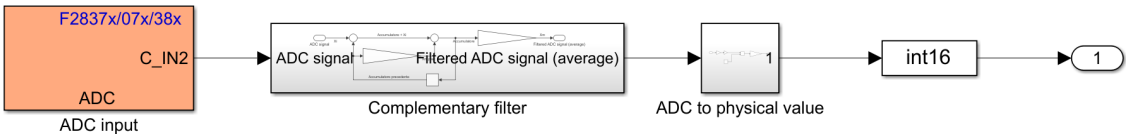


Figure 3.9: Blocks performing a generic analog acquisition.

In figure 3.9 is shown a general process for the acquisition of an analog signal. The output of the ADC block (`int32`) pass through a complementary filter.

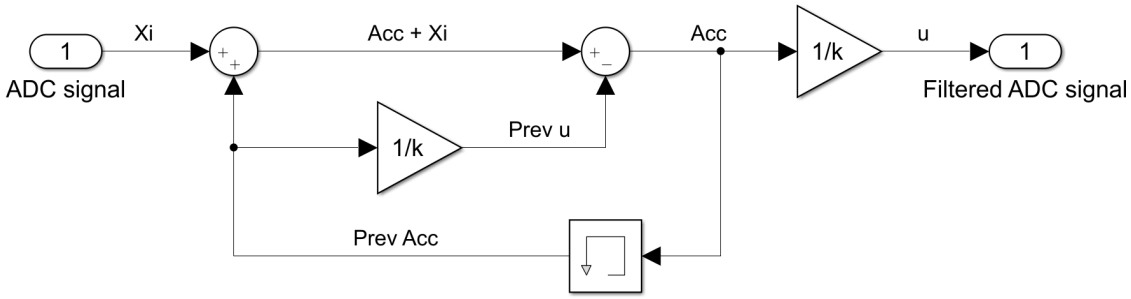


Figure 3.10: Complementary filter's block diagram.

Complementary filter (which block diagram is shown in figure 3.10) allows to filter raw data coming from ADCs. If u is the mean value and x_i the actual value read by ADC, it can be defined as

$$u_i = (1 - n)\, u_{i-1} + n\, x_i, \text{ with } 0 < n < 1 \tag{3.2}$$

It compute a weighted average between the previous mean value and the actual read value.

In order to implement this filter on Delfino LaunchPad™, its formulation must be rearranged. Introducing the variable $k = 1/n$ and an accumulator defined as

$$Acc = \frac{u}{n} = k\, u \tag{3.3}$$

it is possible to write

$$u_i = \frac{1}{k}\left(x_i + Acc_{i-1} - \frac{1}{k}\, Acc_{i-1}\right) \tag{3.4}$$

that is the formulation reported on figure 3.10.

Even if the ADC resolution is only 12 bit[2], ADC output is an `int32`. This is because accumulator can reach very high value, causing overflow if `int16` data type is used. After

[2]This is a good resolution, the *low* adjective refers to data length.

complementary filter, the mean value is reduced to `int16` data type because all other parts work with this type.

In the next block, the mean value (the filter's output, an integer number from 0 to $2^{12} - 1$) is converted into a physical value. Because of integer numbers, the commutative property of sum and multiplication is not yet valid, so the two formulas

$$V_{out} = \left(\frac{ADC_{in}}{2^{12} - 1} \right) p \, V_{supply} 1000 = 0 \tag{3.5}$$

$$V_{out} = \left(ADC_{in} \, p \, V_{supply} 1000 \right) \frac{1}{2^{12} - 1} \neq 0 \tag{3.6}$$

are not the same (i.e. the first always return 0). The p factor allows to pass from ADC point to a physical value (expressed in N, A or V) while the 1000 factor add milli- prefix, so the truncating error is not exaggerate.

In order to let the system works in a correct manner, the only fundamental sensor is a load cell to measure the brake force, that is the feedback for the controller. However, for diagnostic and prognostic purpose, other sensors can be implemented. The Boosterpack DRV8305 BoostXL natively include a voltage supply sensor, three phase voltage sensors and three phase current shunt sensors. Here, even if the voltage supply and phase current are measured, their values are used only for data analysis purpose.

3.3.1 Voltage supply measurement

Voltage supply, like phase voltage, are sensed thanks to an high-value resistor (whose order of magnitude is units of [kΩ]) connected in parallel to the voltage drop that must be measured: for the Ohm's law, an higher resistor value require less current to produce the same voltage drop. As an example, the phase voltage sense's resistors are R11, R12 and R13 of figure 2.8 on page 14. Capacitors work as low-pass filter.

3.3.2 Phase current measurement

Current measure is more complex: it require a low-value resistor because this resistor must be connected in series with the load: the low value (here is 7 mΩ) will not affect too much load's impedance, even with very high current. However, an high efficiency measure will produce a virtually null voltage drop, so it requires an operational amplifier to amplify this signal and send it to the ADC. These shunt resistor are R4, R5 and R6 in figure 2.8, each one of this is R_{SENSE} in figure 3.11

The amplifier scheme is reported in figure 3.11, as reported in BoosterPack DRV8305EVM datasheet. Through SPI communication, it is possible to change OpAmp gain, modifying the value of resistor Rx and Ry. The shunt resistor is connected on the low side of the MOSFET.

In order to compare and verify values read by the shunt resistors, some external current sensors has been connected in series with each phase. That sensors use Hall effect to generate a potential difference that can be detected by the ADCs.

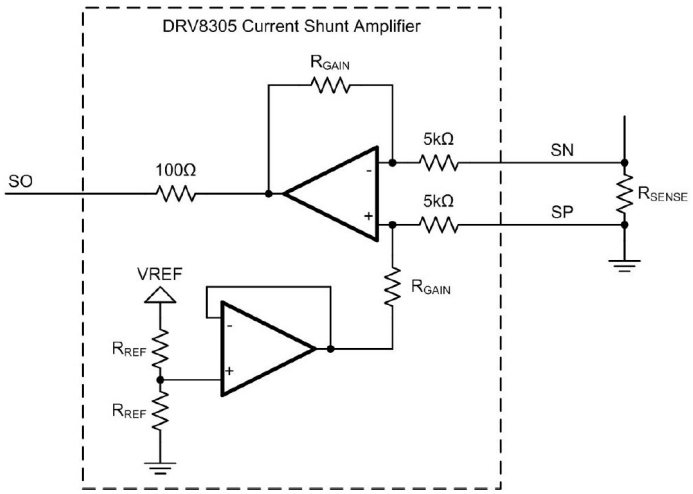


Figure 3.11: Low side current shunt sense.

3.3.3 Force measurement

The load cell used for the test bench is a general purpose one, with DY510 amplifier. Range is from 0 kg to 2 kg. Both load cell and amplifier are shown in figure 4.1 on page 50.

The amplifier allows to set the zero value of the load cell and can output both voltage or current signal. In order to use ADC without the need of shunt resistor and operational amplifier, voltage output has been selected. The amplifier is powered by an external 5 V source.

This sensor is not the one that will be mounted on the real electro-mechanical actuator, both because of the range and resolution. However, the working principle and the output are of the same type, so it will be pretty easy to switch between the two load cells types.

3.4 Serial communication

While mounted on airplane, the main board will receive pilot input from the main computer through ARINC 429 data bus. At this moment, one solution to replace the pilot input is to use an external potentiometer connected to an analog input of Delfino microcontroller. Potentiometer can directly control the duty cycle applied to each phase or a force reference. In spite of its simplicity, this solution does not give enough precision to conduct the required validation tests described in chapter 5. Moreover, during controller tuning, it could be useful to make real-time change to controller parameter, without waiting some minutes for any code rebuild[3].

For these reasons, it has been decided to implement a serial communication between an host software (a Simulink program that run on a computer) and the Delfino microcontroller, the target. Host file is described in section 3.10. A serial communication, together with safety measures described in section 3.8, allows external control over microcontroller: for

[3]Rebuild and deployment of Simulink code can require up to 3 min.

example, human intervention is mandatory to enable the driver or to complete recovery mode.

The module used on Delfino LaunchPad™ is SCI A (Serial Communication Interface™ module A). The baud rate of serial communication has been set to 230 400 bps. Even if this value is stored inside `baud` variable in `init.m` file, it must be numerically written in General Settings/Hardware Implementation/Hardware Details/SCI A/Baud Rate field of Simulink target file.

All data sent via serial communication are `int16` type. This will reduce the need for data type conversion blocks, since this data type is widely used inside the software. In early version of this program, it has been found that the presence of these blocks, even if unnecessary, will contribute to the general slow down of the execution time, causing the timing problem described in section 3.5. For the same reason, data sending and data receiving on target have now tow different architectures:

- Receiving block follows an index-value structure;
- Transmitting block simply muxs all data in a single vector.

Initially, both of them perform just mux and demux operations: the RX block read the serial port, waiting for a very long vector composed by `int16` data values, while the TX block packs all the `int16` data in a single vector and send it to the host. It has been noticed that sometimes target will not reply properly to host input and other time the reply is given with a very high delay (up to 30 s or 1 min). This behavior was caused by two different aspects:

- target read serial buffer continuously; after any reading the buffer was cleared; this happen even if the host did not complete data sending, causing loss of data;
- demux operations, as will be explained in section 3.5 require high workload, causing a slowing down of the execution. Because of this time stretching, both host and target sent 230 400 bps, but with two different definition of time unit.

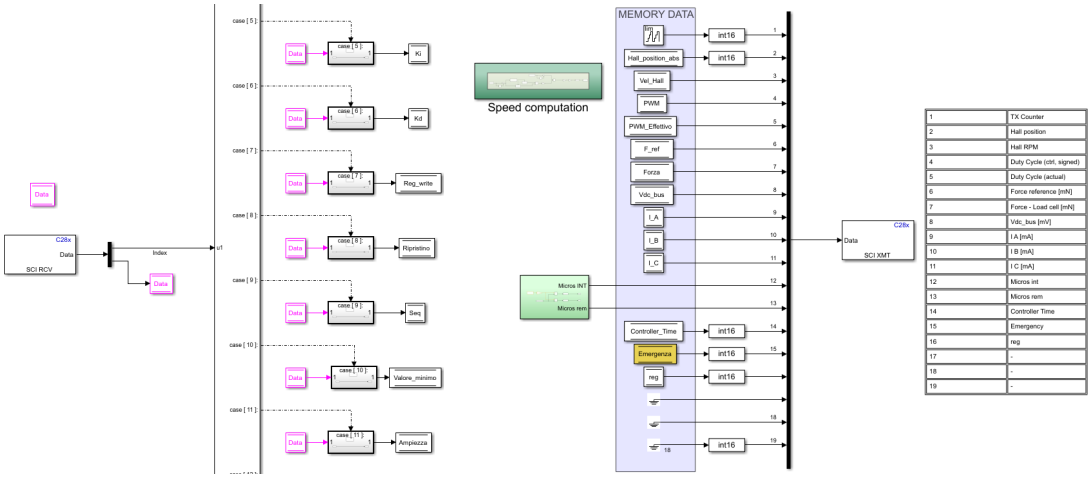
At this moment, both problems have been resolved: the first with a local solution, while the latter with a major intervention.

3.4.1 Data receive

RX is an Atomic Block with sample time of 0,002 s, a frequency 50-times greater than the host's transmit sample time (see section 3.10).

As mentioned before, the RX follow index-value architecture: the host send an index and its corresponding value. In this way, if data loss occurs, only one single value is not updated, not affecting the others. Thanks to memory block, a not written value implies that the previous value remain active, avoiding problems caused by not valid input.

For graphical reason, figure 3.12a only shows the central portion of this block. A complete list of index-value pair can be found in table 3.5 on page 48.



(a) *RX block* (b) *TX block*

Figure 3.12: SCI A Target communication.

3.4.2 Data transmit

Even TX is an Atomic Block with a sample time of 0,001 s. Unlike RX, it simply gathers all data in a single vector and send it to the host. This is because implementing the same architecture of the RX block (index-value pair) leads to the same time management problems that arose with the old RX structure.

A list of all variables sent from target to host can be found in table 3.4 on page 46, even with a different order. The only difference is `micros` variable: because it is 32 bit, it must be split and sent through two different `int16` channels: the first sends the integer result of $\texttt{micros}/2^{15}$, while the last give the remainder of the division. In this way, it is possible to recreate the original value in the host file.

Figure 3.12b shows the TX block. Inside of this block, it is possible to find the speed computation block too. Even if these are two very different tasks, they are put together in order to exploit the same atomic block.

3.4.3 Speed computation

By definition, angular speed is the time derivative of angular position. In Simulink there is a block who numerically computes derivative, using system time. Furthermore, only discrete block can be used here, because microcontroller compiler does not support continuous function.

Speed computation is performed inside an Atomic block and not in the main program. This allows to increase the counted step, with an overall reduction of relative errors caused by both machine numerical precision and timing.

Because of timing problem and the use of integer type, it has been decided to manually recreate the derivative computation using a simple backward incremental ratio

$$\frac{\mathrm{d}u}{\mathrm{d}t}_i \simeq \frac{u_i - u_{i-1}}{t_i - t_{i-1}} \tag{3.7}$$

The number of occurred steps are computed in Position Counter block (subsection 3.2.1). Elapsed time is store in `micros` variable; it is measured in [µs]. In order to avoid overflow, `micros` is 32 bit length, thus allowing to compute up to $2^{31} - 1 = 2\,147\,483\,647\,\mu s \approx 35\,min$, just like position counter.

Even if TX block is of atomic type, i.e. its execution frequency should be fully deterministic, in order to reduce approximation errors, elapsed time is manually computed at each execution.

Because of `micros` is a signed variable, when overflow occurs, it is interpreted as a negative value, so the time difference will be greater than the maximum allowable value or less than zero (depending at which instant overflow occurs). Handling this behavior requires an IF block: when anomaly is detected, the time difference is automatically set to theoretical value of 10 µs, TX block's sample time. Even if this will cause a momentary loss of precision, it is better than output a completely wrong value.

After this, elapsed time is multiplied by 0.001, in order to obtain ms, so a lower numerical value is obtained, reducing errors caused by integer math operations. The incremental ratio is then converted to a rpm `int16` value.

For a detailed description of time computation, see section 3.5.

3.5 Timing management

A critical aspect of this software is the timing management. While performing some tests, many different problems arose:

- speed computation return a set of random values;
- speed computation return a plausible value k-times greater than the correct one, where k is a constant value that change if code is modified;
- serial connection is not stable, even if settings are correct;
- when serial communication works, sometimes there is a huge delay between sent and reply;
- function generator fails to reproduce correctly signal's frequency;
- when generating any sequence, time between two different values is greater respect to the nominal value;

All that problems occurs at different time and with different software version. Because of their nature, it was difficult to understand that there was a single common cause: time. Unfortunately, this is not the start of a theoretical physical discovery about the nature of time. It is just the understanding that Simulink compiler for Delfino microcontroller is not capable of time computation management.

It has been possible to find the common cause behind those problems during a whole day of debug session. In the morning, it has been noticed that angular speed followed the correct behaviour, but the read value were constantly 2.57 times greater than the correct one (measured with an external stroboscope). On the same day, while implementing the very first function generator (a simple square wave, with a period of 10 s), a delay was

found. Measuring it, the wave period result between 2 and 3 times greater than nominal one.

Investigating how this offset changes after software modification, it has been found that some major causes of this problem are demux operation in serial communication RX block, SPI communication and Data Type Conversion block. These operations increase drammatically microcontroller workload because of Simulink's compiler inefficiency. In order to understand why this happen, a better analysis of this compiler is required. Probably, at each execution step, demux block allocates and frees new memory slots, the SPI block writes out its input (even if it is not changed).

One possible way to solve this problem is to increase software efficiency:

- for serial communication, index-value architecture has been adopted (see section 3.4), when demux block has been replaced by a switch case;
- a new procedure has been developed to manage SPI communication, with SPI Master Transfer block confined in a Switch Action block (see section 3.2.2);
- by default, Matlab and Simulink work with `double` data type; to reduce the amount of data type conversions needed, all blocks have been properly set to directly output the desired data type. Generally, all source blocks (constants, ADC blocks, data store memory blocks ...) have a defined data type (i.e. `int16` or, in some case, `int32`), while all other blocks (gain, math operation, switch output, ...) have been set to output the same type of input (otherwise, if a data type is specified, the block automatically perform cast operation, even if it is not necessary).

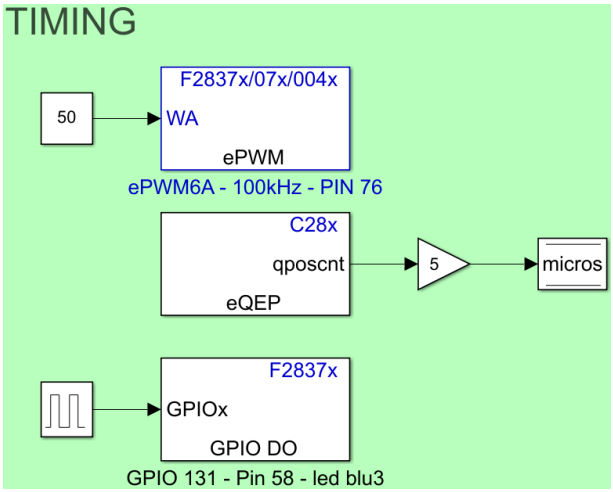


Figure 3.13: Timing management.

However, software efficiency improvement is not a robust solution: any little code modification could lead to this problem again. In order to constantly control how Delfino LaunchPad™ compute elapsed time, 1 kHz square wave is generated on GPIO 131, pin 58. This pin is directly connected to an oscilloscope: comparing the real frequency on the oscilloscope's screen with theoretical value allows to find when problems occur.

In order to obtain a reliable time measure (figure 3.13), a PWM signal with 200 kHz and eQEP module are used. PWM signal is externally routed to eQEP A module, pin I; then, with the proper block, it is possible to automatically computed the number of pulse. Knowing the absolute number of pulse and the PWM frequency, it is possible to evaluate the real elapsed time (expressed in [µs]). and to save it in `micros` variable.

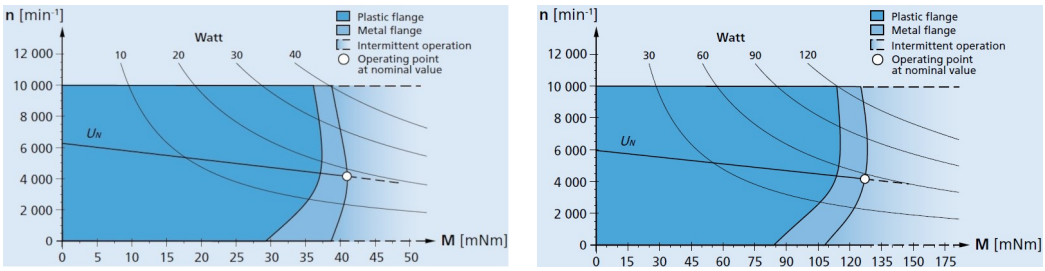
3.6 Controller

At this point, the software is capable to spin a BLDC motor by setting duty cycle value. This is a very simple open loop control, where the controlled variable is the total power, the product of torque and angular speed (and the efficiency).

Given the value of duty cycle, the motor automatically sets its torque and speed depending on external load, according to what is depicted in the proper torque-speed diagram (see figure 3.14).

Usually, this is not an acceptable behaviour because this system will be susceptible to any external disturbance, that can cause the motor to stall or to accelerate, overcoming the safe working area. Furthermore, even if boundaries are not exceeded, the final equilibrium point could not be the desired one.

For these and other reasons, it is necessary to implement a controller, which aim is to guarantee desired braking force on each actuator.



(a) *BLDC 3216 motor operation areas @24 V* (b) *BLDC 4221 motor operation areas @24 V*

Figure 3.14: Recommended operation areas for Faulhaber's 3216 and 4221 BLDC motors.

3.6.1 Controller architecture

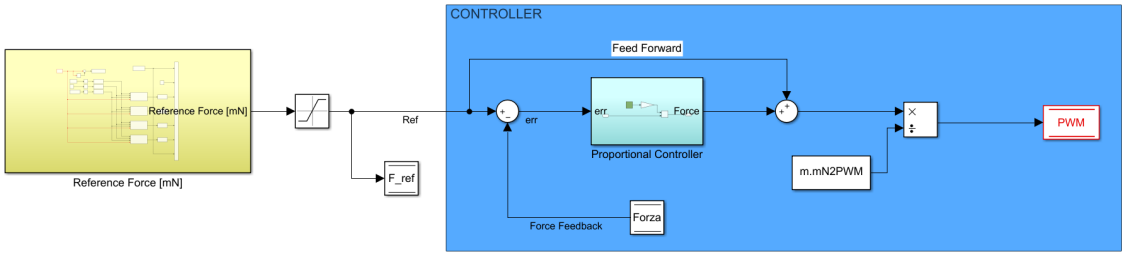


Figure 3.15: Controller architecture.

In figure 3.15, it is possible to see the controller architecture. On the left side there is the function generator, described in section 3.7, that creates the reference signal, while on the right side there is the controller.

The force feedback is provided by a load cell mounted on the electro-mechanical actuator.

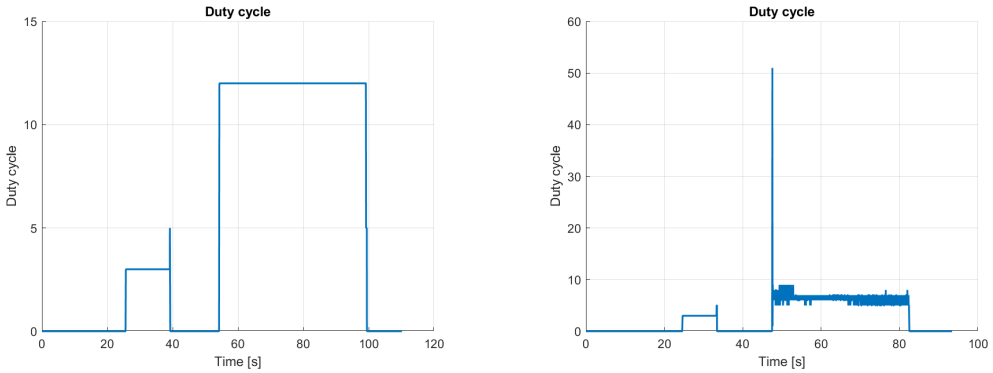
Here, the controlled variable is the force, expressed in [mN]. It has been decided to directly monitor the output force and not other variables (such as current) because in MAG's brake system architecture, only one shunt resistor (collecting all phase current) will be used. This became necessary because of the limited amount of analog input on Delfino LaunchPad™. Moreover, the load cell output a force value (dimensionally [mN]), not the desired phase current.

A simple PID closed loop scheme is not suitable here because, when the target force has been met, the error will be zero, so will the controller output. On the other way, feedback is mandatory in order to control the real braking force.

Here, both open and closed loop architecture has been used here. Figure 3.16 shows the controller output (the duty cycle) for open loop only response (3.16a) and the closed loop response, without feedforward (3.16b) to a force step of 2000 mN.

On the left of both graphs, it is possible to see the initial reset routine (section 3.9).

The duty cycle of the feedforward branch (figure 3.16a) is just a constant value, not affected by load cell output. On the other way, the duty cycle of feedback controller (figure 3.16b) reaches the very high value of 50 % because of the initial error. This last control architecture does not allow to face any load cell failure.



(a) *Open loop step response* (b) *Closed loop step response*

Figure 3.16: Differences in step response between feedforward open loop only response and proportional feedback only closed loop response.

The closed loop is a simple proportional controller. Proportional constant can be set via serial communication, This allows real time tuning of the controller.

Integrative and derivative terms are not implemented here for two different reasons: the first is that, using the test bench described in chapter 4, time response is satisfactory and do not require any other term; the latter is that these terms require a reliable time computation (see section 3.5). Those two contributions will be developed in future only if they will indispensable using real mechanical actuators.

The output of feedforward branch and the one of proportional branch are then added together and multiplied by a constant value. This value has been derived empirically and represents the converting factor between the required force [mN] and the corresponding PWM duty cycle.

3.6.2 Current limitations

In electric motor, the output torque can be directly related to the current phase, thanks to the so-called torque constant K_m. Theoretically, a current loop or a torque/force loop[4] are equally (i.e. they control the same variable).

However, implementig a current loop or a force loop requires different sensors, different architecture and, in fact, each loop controls different characteristics.

Phase current is related not only to the output torque, but also to winding temperature and bearing temperature. In the same way, torque is directly related to the maximum allowable load on mechanical actuator.

Maximum load for the selected mechanical actuator can be easily set inside the Saturation block of figure 3.15 on the reference signal line.

Thermal limits are more complex to evaluate. According to Faulhaber's analysis, thermal limits for 3216 and 4221 motor are resumed in table 3.1. Given the absence of a temperature sensor inside motor case, phase current mean value must be used to mathematically derive winding temperature. At this moment, no thermal and current limitations have been implemented yet.

	Winding temperature	Bearing temperature
3216	115 °C	90 °C
4221	110 °C	85 °C

Table 3.1: Thermal limits for Faulhaber's BLDC motors.

3.7 Waveform generator

Reference Force (figure 3.17), the yellow rectangle of figure 3.15, is a block which aim is to output different waveforms. Signal selector variable (defined in `Host_Log_Dati.slx` file, section 3.10) allows the user to select which waveform to use, according to table 3.2.

Each signal is characterized by its amplitude (A, [mN]), its frequency (f, [mHz]) and its minimum value (x_0, [mN]). All these values can be set via serial communication.

Time signal t comes from `micros` variable (see section 3.5).

As an example, the mathematical formulation for the sawtooth function is reported below (equation 3.8). It has been tried to reproduce the correct execution order of mathematical

[4]Electric motor's output is torque, but mechanical actuator convert this torque in linear force.

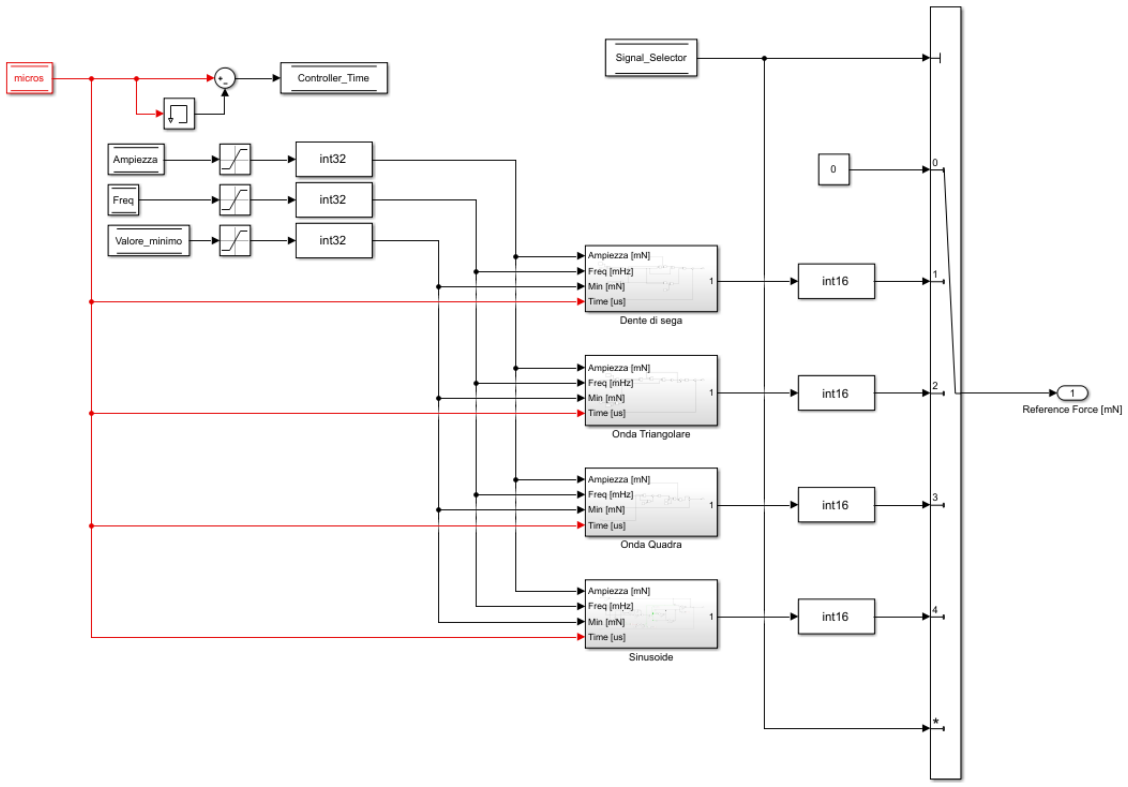


Figure 3.17: Function generator.

Signal selector value	Signal description
0	Constant 0 mN value
1	Sawtooth wave
2	Triangular wave
3	Square wave
4	Sinusoidal wave
≥ 5	Output the corresponding value

Table 3.2: Signal selector possible values.

operation. For the correct math operation execution order and other waveforms, please refers to Simulink model `Target200.slx`.

$$\begin{aligned} T &= \frac{10^9}{1000f} \\ t_* &= \frac{t}{1000} \\ F &= x_0 + \left[(t_* \bmod T)A\frac{1}{T}\right]\left[\frac{t_*}{T} \bmod 2\right] \end{aligned} \tag{3.8}$$

3.8 Safety

Safety is an important matter in all engineering subjects and must be taken in account during all design phases.

Furthermore, the aim of this software is to perform different tests with different actuators, so it is important to implement some safety measures.

The software developed during this master's thesis work is not conceived to undergo the certification process, but the logical structure behind it will constitute the starting point for a future certifiable code.

Figure 3.18 shows the block Max Position Safety Measure (red block), capable of evaluating and handling emergency. If the second output of this block is true, the BLDC motor control is switched from Starting Sequence block to this safety block. This remain true until recovery procedure is performed, so the electro-mechanical actuators is disabled. This drastic solution is useful during test phase because it give enough time to think about what caused the emergency.

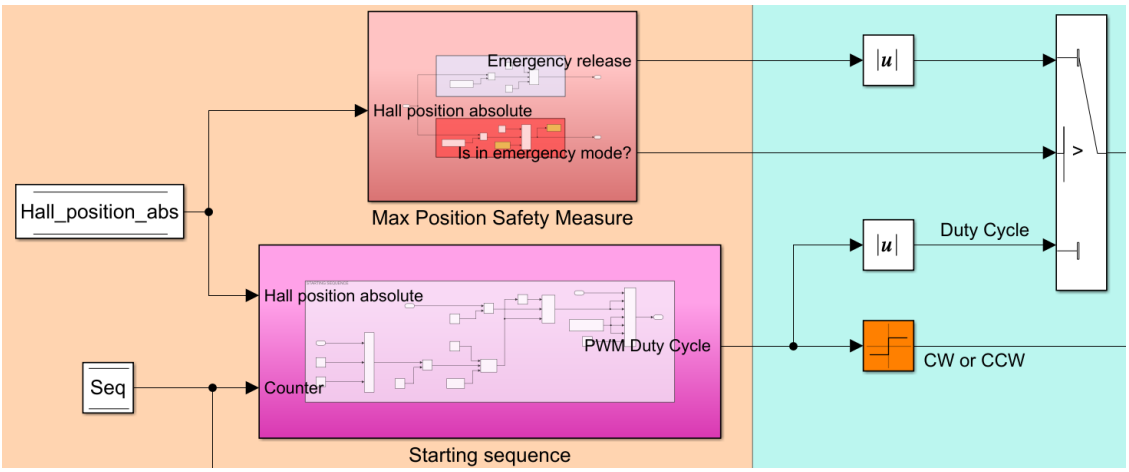


Figure 3.18: Maximum Position Safety Measure and Starting sequence blocks.

3.8.1 Safety measures

Safety measures implemented in this work are:

- Maximum force limitation;
- Maximum run limitation;
- Reverse spin protection.

Maximum force limitation

The limitation over the maximum applicable force is the simplest one. It consists of a saturation block on the output line of Function Generator block. This saturation will not rise the emergency flag, but will just limited the reference force to a certain value, based

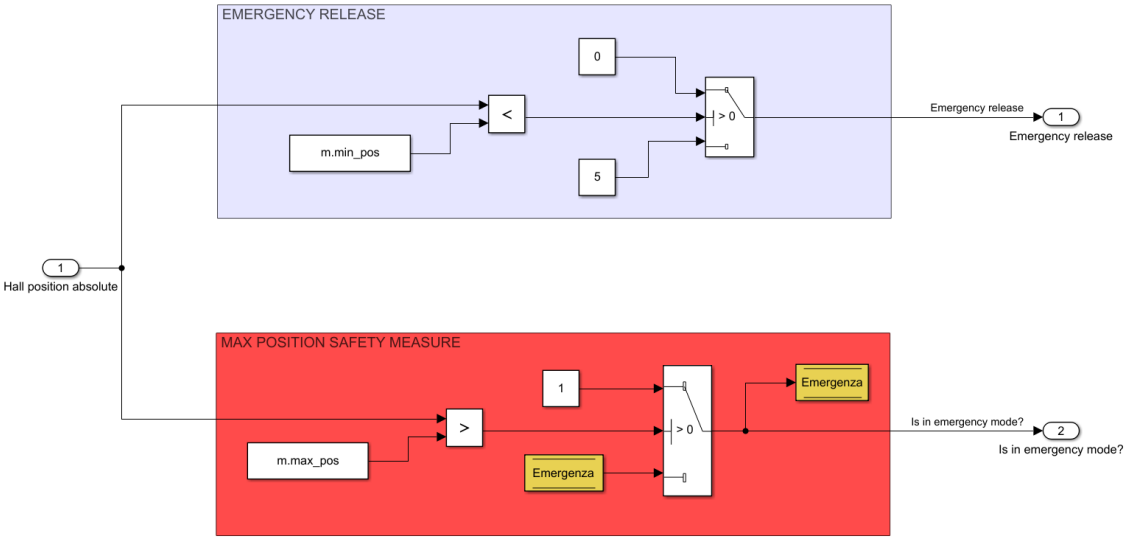


Figure 3.19: Safety measures.

on the maximum load each actuator can support. So, in this case, no recovery procedure is needed.

This limit is required because the user can directly set a reference force value through serial communication. If this value is greater than the maximum allowable load, actuator can be damaged.

Maximum run limitation

Maximum run limitation performance is similar in scope to force limitation, but with different implementation. When the motor spins in the right direction, the actuator will move forward, causing the brake disks to press each others. Erroneous input or wrong computation (caused by run-time errors) could eventually cause an uncontrolled and/or undesired spin. Thanks to position counter block (subsection 3.2.1), when position exceeds a certain threshold, an emergency occurs (red square on figure 3.19). In the same time, duty cycle control automatically switch from controller block to the violet square of figure 3.19, where duty cycle is set to 5 % until a minimum position, close to the initial unloaded one, is reach and then duty cycle is set to null value.

This portion takes into account elastic response that occurs in test bench (where the main load is a rubber band). The need of this last portion will be empirically reviewed taking into account the real stiffness of the system, when this software will be validated with real actuators.

Implementing this safety measure, different possible versions have been studied.

An easier version of this safety measure will simply read load cell measurement and directly implement an almost real time (compared to theoretical limit described above) force limitation. However, this solution's major drawback is the load cell itself. If a failure on the load cell occurs, measured force can be a random value, so the actuator behaviour will be unpredictable (e.g. it can enter in emergency mode even if it is not necessary). Here, position is measured thanks to Hall effect sensors: they are more reliable because

of their simple package not directly subjected to the brake force. Moreover, Hall sensors' failure will cause an almost complete loss of control of BLDC motor.

Although the limitation over the maximum run comes from mechanical considerations, mostly related to test bench design, it can be useful to prevent electronic damage.

Indeed, if the duty cycle is instantaneously set to 0 when emergency occurs, the stretched rubber band will cause the motor to forcibly spin with a maximum speed proportional to elastic band's elongation. This behaviour will cause MOSFETs to be reverse biased and high reverse current flowing on the body diode of MOSFETs.

Reverse spin protection

Because of test bench design (chapter 4), it cannot transmit compression stress, but only tensile stress. This is one of the major different respect to the real system.

If the load cell output a real force greater than the reference value, the controller cause the motor to spin in the opposite direction. This is a normal behaviour if force has a positive value; however, it can happens that reverse spin causes the rope to roll up in reverse around the spool. According to controller scheme, a reverse spin will cause a force decreasing and negative values are allowed. In the test bench, negative force values are not allowed and the real force follows a trend similar to the absolute value function.

For this reason, a minimum run limit must be set. The actual implementation can be seen on the red square of figure 3.21. Here, the minimum value has been set to 0: when the position counter falls below this limit, any duty cycle sent to motor will be a positive value, interdict reverse rotation.

This will not causing the rise of emergency flag, so recovery procedure is not needed here.

Even if this protection is directly related to the test bench developed here, it is not excluded that it will be implemented on the real system too: changing the minimum position value, it is possible to reproduce via software a physical end of stroke.

3.8.2 Recovery procedure

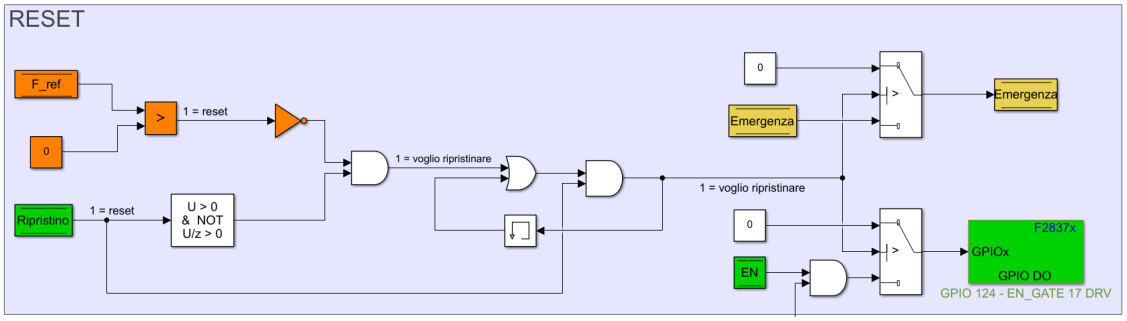


Figure 3.20: Recovery procedure.

Recovery procedure is needed only when the emergency flag becomes true, i.e. when the maximum run limit is reached. Figure 3.20 shows the logical scheme required to reset the software. `F_ref` and `Ripristino` variables can be set through serial communication.

First of all, the reference force must be set to 0; then `Ripristino` variable should be set equal to 1. It is recommended to set to LOW value also `EN` variable: this will prevent, at the end of the reset, an abrupt start of the motor.

If these conditions are met, the DRV8305's enable pin is forcibly set down, thus resetting all fault indicators saved on driver's register. On the same time, the emergency flag is set to zero, so the led on `Host_Log_Dati.slx` (section 3.10) can turn green again and the motor control switch from safety blocks to Starting sequence block.

Subsequently, the enable pin is automatically set to the value sent from host file, so it is possible to enable the driver and restart as usual.

As stated before, emergency mode will take complete control of the motor, ignoring any user input (except for those required to restart the software). This behaviour will not be considered safety compliant in a real aircraft architecture, because it will preclude to pilot the possibility to take corrective action upon system failure. Here, on the contrary, this forced stop and the articulated process needed to restart the motor impose to perform a detailed analysis of what caused the emergency mode to activate.

3.9 Initial reset routine

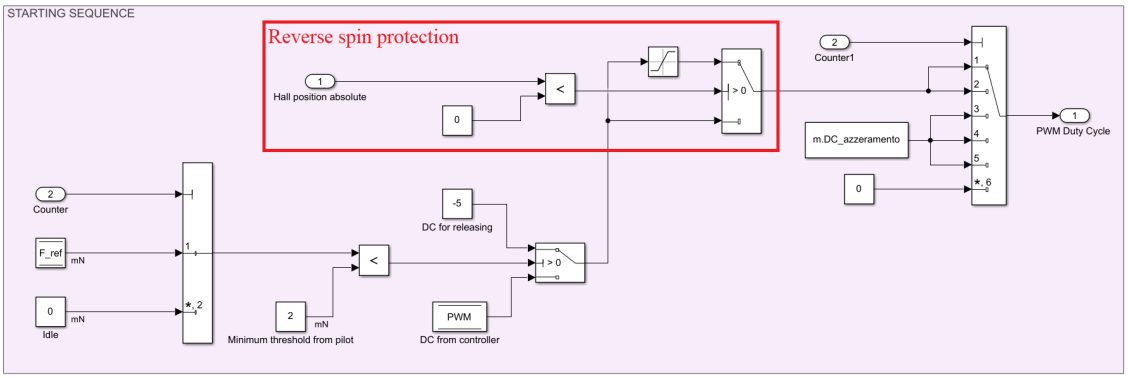


Figure 3.21: Initial reset routine.

A BLDC motor not equipped with high resolution position sensors (like an encoder) should not be used where high precision servo performances are required. Depending on the reduction ratio, the resolution of only 8,57° (subsection 3.2.1) could be not enough to guarantee an accurate position control.

Brake pads are subjected to wear out: if this is not properly taken into account during software development, it will cause a performance degradation of the global system. Wear out will cause a progressive increase in time required to obtain a certain brake force and this is not acceptable.

For this reason, a starting sequence has been developed: when performed, this procedure allows to modify the zero point reference taking into account pads wear.

Moreover, if an external reference is given, with this starting sequence it is possible to quantitatively monitor and measure brake wear out, allowing the development of a more efficient and cost effectiveness maintenance plan without adding external sensors.

Figure 3.21 shows the block implementation of the initial reset routine.

This process is constituted by different steps (the switch block on the right), governed by a counter.

At this moment, the counter (that is `Seq` variable) is manually set via serial communication, but the software has been developed taking into account the automation of this procedure (that is why some `Seq` values seems to be unnecessary, to let pass enough time between two different instruction).

In order to perform the starting sequence, `Seq` variable must be sequentially set to a value from 6 to 1, according to table 3.3.

`Seq` value	Description
6	Duty cycle is set to 0 %
5	Duty cycle is set to `m.DC_azzeramento` value, equal to 3 %
4	Wait the forces to reach equilibrium point
3	Set actual position to zero value (21 points, half turn, see subsection 3.2.1)
2	Duty cycle is set to −5 %
1	Duty cycle value is decided by controller block

Table 3.3: Initial reset routine sequence.

During the last two phases, the reverse spin protection safety measure (section 3.8.1) applies, so the motor cannot reach position below than the zero value set during this procedure.

Figure 3.22 shows the shaft position (expressed as hall steps) during initial reset routine.

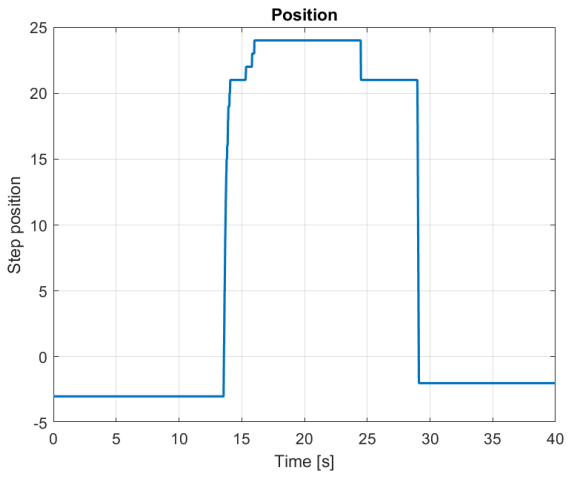


Figure 3.22: Position during initial reset routine.

3.10 Host file

The `Host_Log_Dati.slx` file (figure 3.23) is the one used to send command to the Delfino LaunchPad™ and to log data.

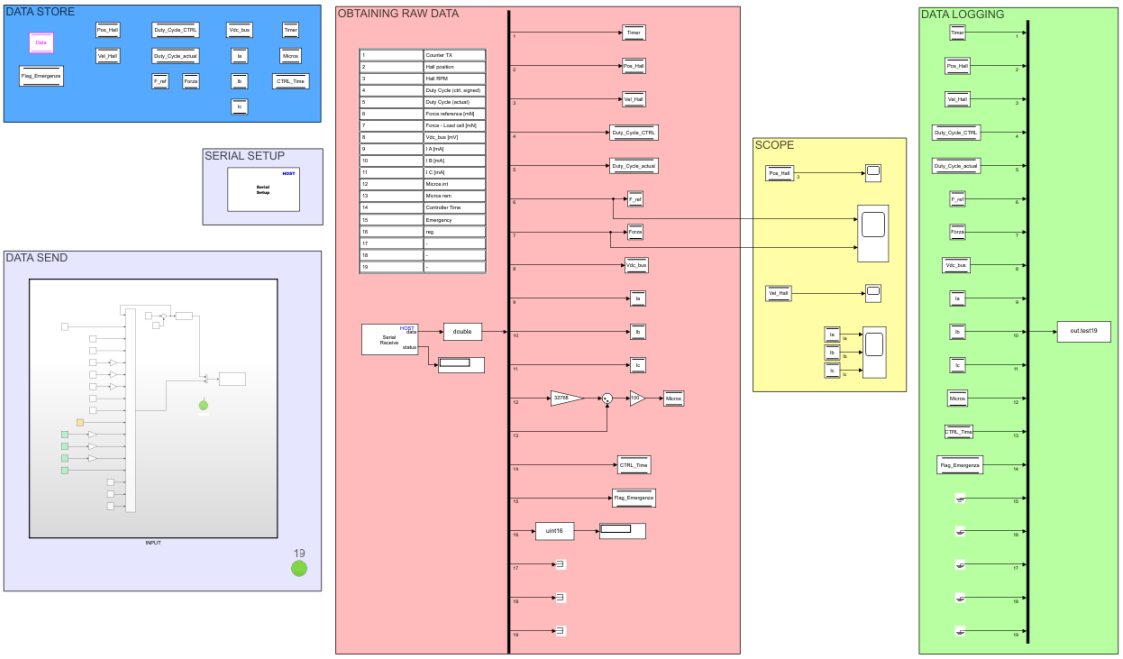


Figure 3.23: `Host_Log_Dati.slx`, main view.

It is a very simple program:

- the blue part contain all Data Store Memory Block;
- the violet part sends command;
- the red area reads data from Delfino LaunchPad™;
- the yellow area allows to visualize some data in almost real time;
- the green part saves all data received in a `.mat` file as a Structure With Time format.

The data saved in the `.mat` file are listed in table 3.4, with a brief description. Some channels are intentionally left void, allowing to save more data without large modification of block diagram.

This program runs with a time step of 0,001 s in real time mode. To allows real time mode to be on, it is important to not overload this program with a lot of scope or other way to graphically represents data.

3.10.1 Data sending via USB

In figure 3.24 is depicted the block diagram inside the violet area of figure 3.23. Here, the main structure of this communication is highlighted.

Number	Signal	Description
1	TX counter	Transmission counter
2	Hall Pos ABS	Total Hall steps
3	Vel Hall	Angular speed [rpm]
4	PWM	Duty cycle, controller output
5	PWM Effettivo	Actual duty cycle (safety measures, see section 3.8)
6	F_ref	Force, controller output [mN]
7	Force	Force read by load cell [mN]
8	Vdc_bus	Power supply voltage [mV]
9	I_A	Phase A current [mA]
10	I_B	Phase B current [mA]
11	I_C	Phase C current [mA]
12	micros	Elapsed time [µs] (see section 3.5)
13	Controller_Time	Elapsed time between two controller call [µs]
14	Emergency flag	Flag, true if emergency occurred (see section 3.8)
15	-	Void
16	-	Void
17	-	Void
18	-	Void
19	-	Void

Table 3.4: Data to be saved.

This is an atomic block with a sample time of 0,1 s, defined inside variable `h.step_time`.

Because of what has already explained in section 3.4, for each input, two data are sent: the first one is an index, the latter is the real value. Index is computed using a repeating sequence block (count from 0 to 15), adding 1 (because switch block require input starting from 1) and is converted to `int16` type. The sample time is inherited, so it is the one defined in `h.step_time`.

The data sent are reported in table 3.5. The detailed meaning of all of this variables was explained in previous sections.

Because of it has been decide to do not use Delfino's floating point unit and to reduce the need of Data Type Conversion blocks, all data here are of `int16` type. The length is set inside each constant block. In order to allow a better understanding of data meanings, units of measure are written. Decimal data can be sent multiplying the numerical, decimal value by a proper power of ten. For example, the waveform generator's parameter input ([N] and [Hz]) are multiplied by 10^3, obtaining [mN] and [mHz].

The green circle is a virtual led. Its color is green while the motor is working normally and turns to red if a problem occurs. After the completion of recovery procedure (see subsection 3.8.2), it turns green again.

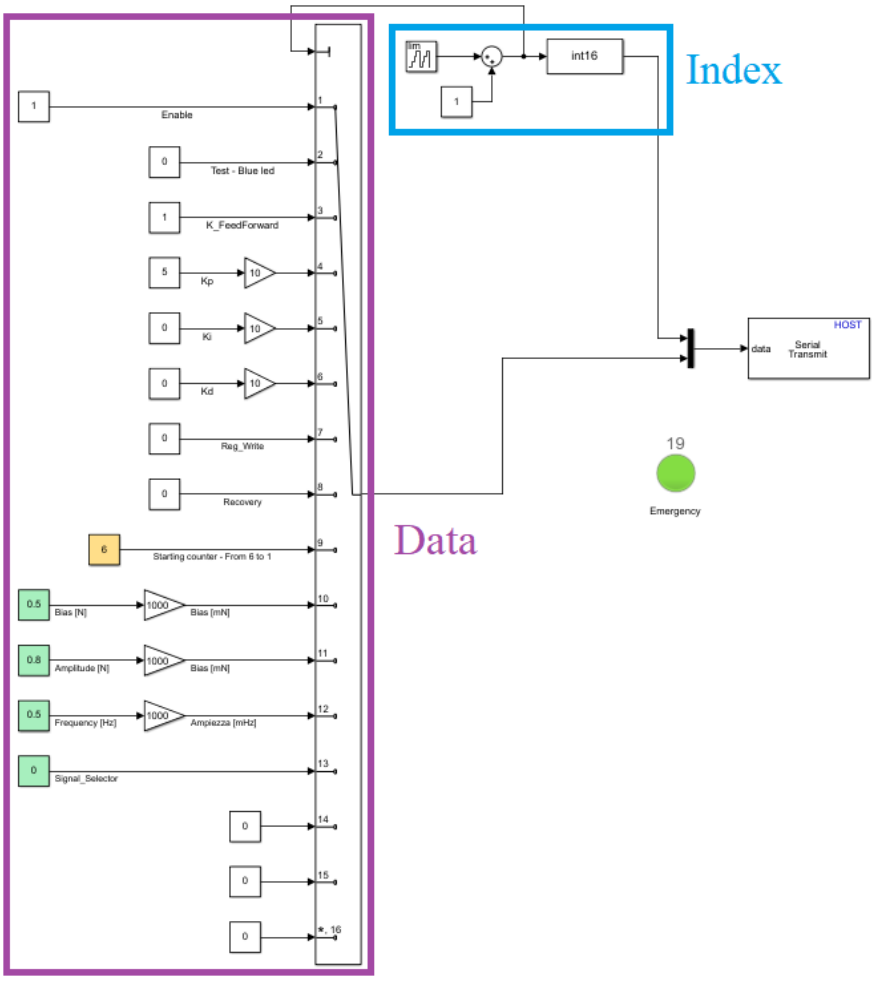


Figure 3.24: Block diagram to send data to Delfino Launchpad™.

Index	Signal	Description
1	Enable	Driver Enable
2	Test - Blue led	Blue led to perform communication test
3	K_FeedForward	Controller's Feed Forward constant (see section 3.6)
4	Kp	Controller's proportional constant (see section 3.6)
5	Ki	Controller's integral constant (see section 3.6)
6	Kd	Controller's derivative constant (see section 3.6)
7	Reg_write	Value to be read from/written on driver's SPI register
8	Recovery	Recovery variable (see subsection 3.8.2)
9	Starting counter	Counter for the initial reset routine
10	Bias	Waveform generator's bias [N] (see section 3.7)
11	Amplitude	Waveform generator's amplitude [N] (see section 3.7)
12	Frequency	Waveform generator's frequency [Hz] (see section 3.7)
13	Signal selector	Waveform generator's signal selector (see section 3.7)
14	-	Void
15	-	Void
16	-	Void

Table 3.5: Data to be sent from Host to Target devices.

Chapter 4

Test bench

This chapter describes the design of the test bench used to validate the software described in chapter 3.

All tests described in chapter 5 have been conducted on this bench.

This is a simplified version of the bigger test bench that will be used to test real actuators. Here, a rope connected to a rubber band replace the actuator. The reasons behind this choice are different: at this moment, actuators have not been built yet; moreover the pressing force required for each actuator is a very large value, thus requiring some structural analysis in order to avoid damages on test bench structure. As final consideration, a new software requires huge validation before tests can be safely conducted on real hardware.

4.1 Needs and requirements

This test bench has been designed to support different type of tests:

- System time response
- Tuning of controller parameter
- Starting sequence performance
- Safety measures effectiveness and recovery procedure management
- Data logging

At this moment, what is important is to test software behaviour in different scenarios. Because no real actuator is used here, the test bench have to reproduce in a satisfactory way the real behaviour of the system. Particular attention was paid to the torque seen by the BLDC motor and to the number of step required to reach the nominal torque.

To virtually reproduce the compression free run (250 µm downstream the actuator) and system's elasticity, a rubber band has been used. The other variable that must be met is the maximum braking force (2645 N).

Serial communication allows to set the desired parameters' values (such as reference force, or starting sequence counter) and to log data. Both these functions must work

seamless. For safety reason, an almost real time data sending is required, while data logging latency can have a greater value. Since some time passes between sending and receiving data on host file, the function generator block (section 3.7) has been implemented on Delfino LaunchPad™: sending back both reference signal and actual signal allows to analyze time response without an undesired offset.

The load cell (figure 4.1) output is similar to the output of the sensor that will be mounted on the real electro-mechanical actuator.

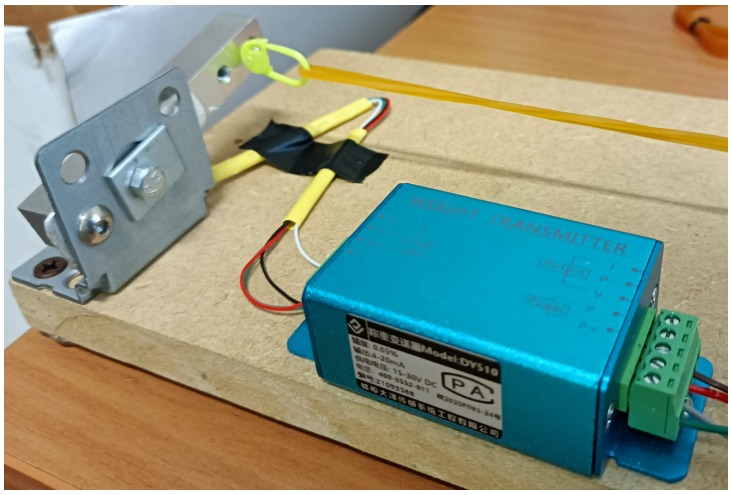


Figure 4.1: Load cell and amplifier used in test bench.

In this test bench, both shunt resistors (subsection 3.3.2) and hall effect external current sensors have been used to measure phase currents and I_{DCbus} current. Even if hall effect sensors will not be implemented on the real system, they are here for calibration and validation purpose only.

4.2 Test bench design

The design process is related to the mechanical actuator developed by MAG and PoliTo (section 2.1.1 on page 8). In order to reproduce the behaviour of the third actuator, a different load cell (with an increased range) and a different rubber band are required. However, mathematical description and software are exactly the same.

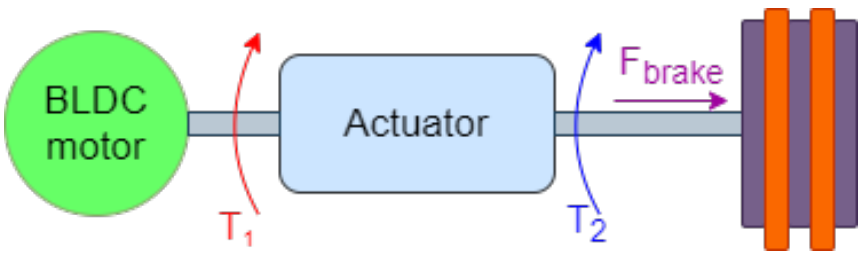


Figure 4.2: BLDC motor and actuator scheme.

First of all, it must be noted that all design parameters mentioned in the previous section are defined after the mechanical actuator. According to the simplified scheme of figure 4.2, if the gear ratio is

$$\tau = \frac{\dot{\theta}_1}{\dot{\theta}_2} = \frac{\theta_1}{\theta_2} = \frac{T_2}{T_1} \tag{4.1}$$

and p is the pitch of the ball screw (where applicable), the torque seen by the motor can be computed from

$$T_1 = \frac{p}{2\pi\,\tau}\frac{F_{brake}}{\eta} \tag{4.2}$$

where $F_{brake} = 2645\,\mathrm{N}$ is the required braking force and η the mechanical efficiency of the system.

The motor shaft diameter is 6 mm. The BLDC motor can mount an external joint on its shaft, so the final diameter is increased up to 19 mm. This leads to a force transmitted to the load cell of

$$F_{load\,cell} = \frac{T_1}{r},\ \text{where } r = 9{,}5\,\mathrm{mm} \tag{4.3}$$

If the efficiency is $\eta = 0.8$, a conservative value, the maximum torque and force seen by the motor are resumed on table 4.1.

Solution	Torque	Force
Sol. B (MAG)	31,2 mN m	3,28 N
Sol. C (PoliTo)	26,4 mN m	2,78 N

Table 4.1: Force and torque seen by the electric motor.

The other parameter that must be met is the free run before the total compression of the disks. This clearance has been designed to be 250 µm. Because of the actuator, the required turn of the motor to obtain the translation Δx is

$$\Delta\theta_1 = \frac{\tau}{p}\Delta x \tag{4.4}$$

This lead to a value of about 2.5 turns for both actuators (with an error less than half turn). Here, what is important is not to exactly reproduce the effective run, but to reproduce its order of magnitude (1, 10, 100 ... turns).

At this point, it is possible to compute the length of the rope that will wind around the joint mounted on motor shaft

$$\Delta l = \pi\, d\, \Delta\theta_1 = 150\,\mathrm{mm} \tag{4.5}$$

Figure 4.3 shows the test bench. It is possible to see the load cell and the amplifier, the rubber band, the BLDC motor and Delfino LaunchPad™.

The motor holder (figure 4.4) has been made with Fusion Deposition Modeling (FDM) method of additive manufacturing. The material used here is polylactic acid (PLA). Despite the poor thermal and mechanical characteristics of this material, it is cheap and easy to manufacture with a commercial 3D printer. Moreover the mechanical stress is very low in this test bench, justifying the use of PLA instead of other materials.

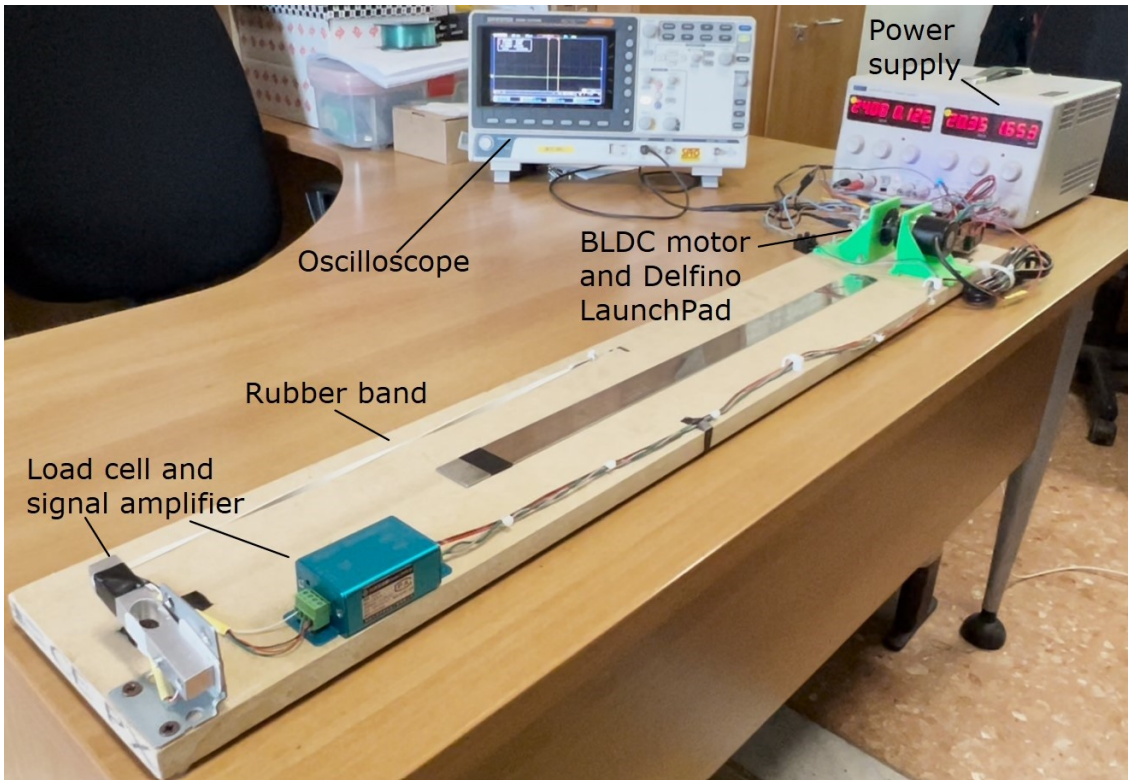


Figure 4.3: Test bench.

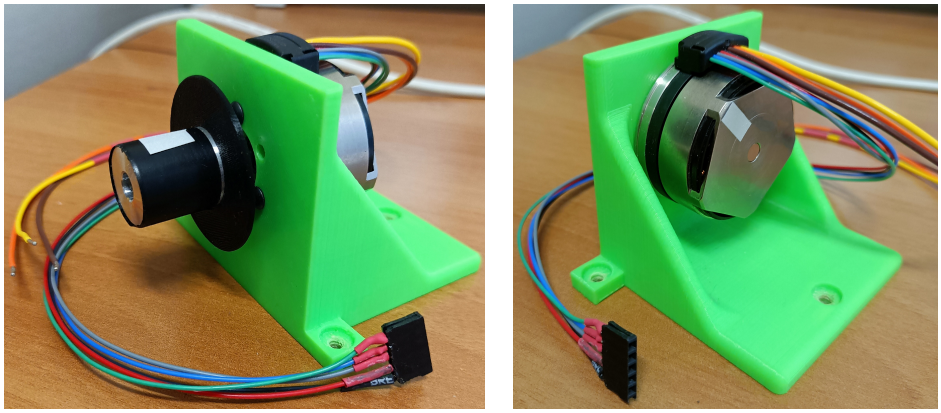


Figure 4.4: Motor holder.

4.3 Field of use

Although actuator's behaviour cannot be exactly reproduced (the design process described in the previous section constitute a good starting point, but it is nothing more than this), this test bench was found to be very useful. Indeed, the main purpose here is to debug and

test all different software aspects, such as the phase commutation scheme, the controller architecture or data logging feature. The test bench allowed to perform different software improvement and to highlight some latent problems (like the time management problem describe in section 3.5).

One of the major difference between this bench and the real system is the load transmission: here, a rubber band and a rope are used. Leaving aside the stiffness given by the rubber band, these links are capable of tensile load transmission only. This aspect will cause some troubles while introducing external disturbances when force feedback is enabled. That is why the need of implementing a reverse spin protection (see section 3.8 on page 40).

One aspect that must be taken into account while designing the real system is the capability of the driver to be enabled. To make the driver become operative, a certain pin must be put on high state. This is not a problem itself, on the contrary it allows a powerful control on the electric motor: in case of emergency, setting the enable pin to low value disconnect the driver's charge pumps, so MOSFETs work has open circuit on high power line. What must be noted is that values sensed by analog sensors are different if the driver is enabled or if it is not. This aspect affect not only the sensors directly mounted on the driver (like phase current and voltage sense shunt resistors), but also external sensor, such as the load cell. Knowing this before testing the real mechanical actuator make possible to avoid some potential troubles on both sensor's tuning and global software management.

Chapter 5

Experimental analysis

This chapter concerns about the analysis of different validation tests performed thanks to the test bench described in the previous chapter. For each reference signal (step response, triangular wave, sinusoidal wave, ...), different controller's proportional constant have been tested.

Not all logged data are reported here, but only the most significant ones to prove software capabilities.

Before any test, starting sequence has always been performed, even if it is not depicted in some figures for graphical reasons.

On the left and on the right sides of some of the graphs reported here, it is possible to notice some peaks (especially in phase current and duty cycle graphs). The reason behind those peaks is the driver's enable pin: when the driver is disabled (i.e. at the start and at the end of acquisition), all analog sensors behaviour changes in an unpredictable way. Therefore, those data can be deleted.

5.1 Free run

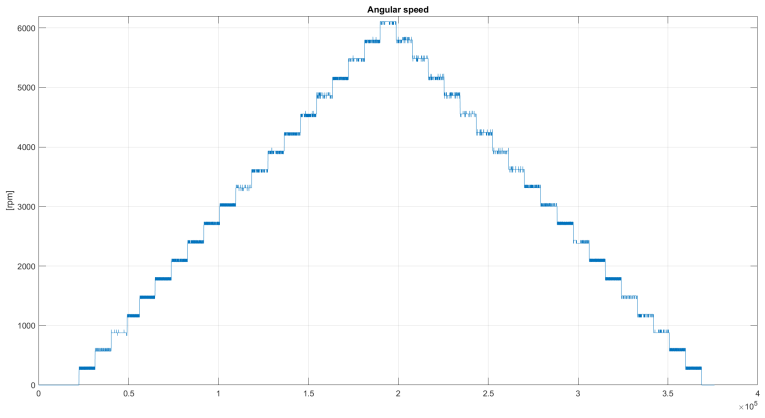


Figure 5.1: Free run test of Faulhaber 3216 BLDC motor.

The aim of this test is to measure the maximum angular speed achievable by the Faulhaber 3216 motor. Therefore, the motor was not connected to the load cell and no load has been applied. Input is directly the duty cycle value (from 0 % to 100 % and back, with 5 % steps), bypassing the controller block.

As shown in figure 5.1, the maximum, achievable speed is 6100 rpm, a value really close to the nominal one declared on the motor datasheet (6200 rpm).

5.2 Step response

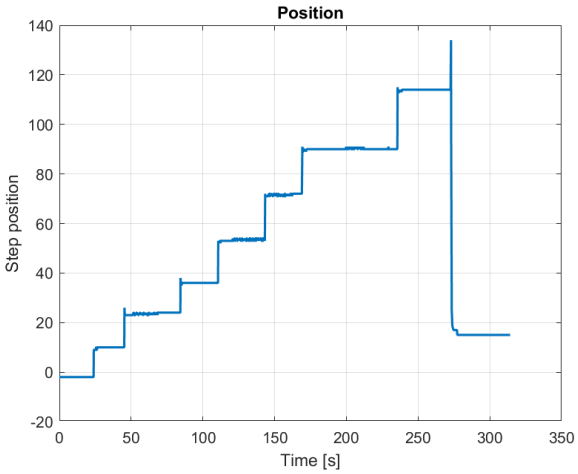
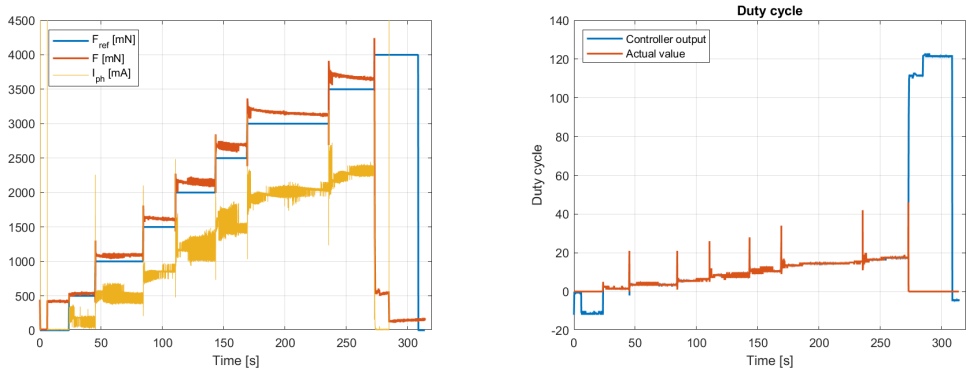


Figure 5.2: Step response, position.

On this test, different constant force values has been set thanks to serial communication. The controller's proportional constant is equal to 5.

The reference signal is a constant value from 0 mN up to 4000 mN, with 500 mN step.



(a) *Force and phase current* (b) *Duty cycle*

Figure 5.3: Step response.

On figure 5.2 it is possible to see the number of steps, while in figure 5.3 there are the force signal (both reference and actual), the mean phase current and the duty cycle.

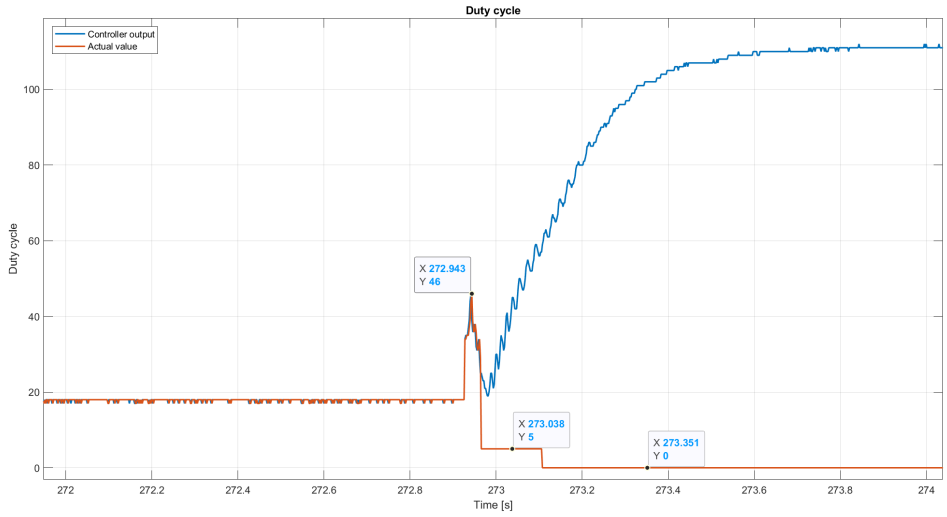


Figure 5.4: Step response, duty cycle, zoom on the final phase.

This test allows to understand how the maximum run safety measure works. When the reference force is set to 4000 mN, the number of step exceeds the declared limit, so the emergency flag is raised and the motor is smoothly deloaded, without negative overshoot, nor a rubber band *sling effect*. This behaviour can be clearly seen on figure 5.4, showing a zoom of the final portion of figure 5.3b.

5.3 Square wave

Figure 5.5 shows the force behaviour when a square wave with a frequency of 0,5 Hz. Minimum and maximum force values are 500 mN and 1500 mN.

Here, different proportional constants have been tested.

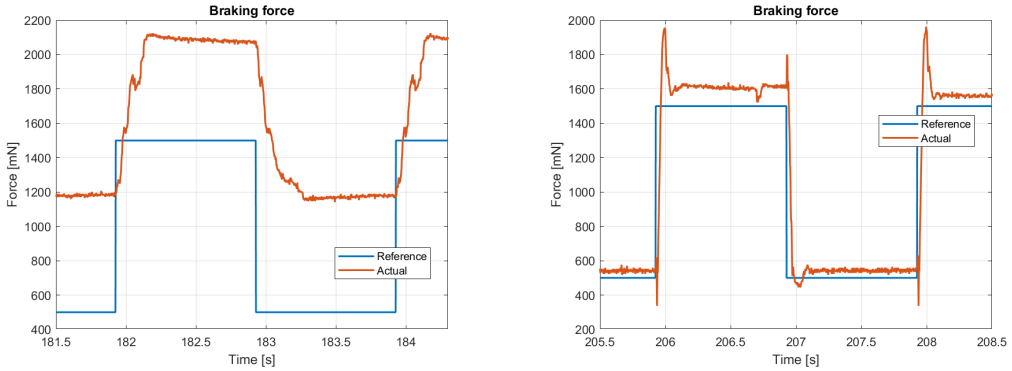
It is possible to notice that deleting the proportional feedback (figure 5.5a) does not give a satisfactory time response, especially because of the offset. However, some braking force is always possible, even if with degraded performance.

A proportional value equal to 5 (figure 5.5b) leads to an acceptable tuning. The overshoot is 30 % of the maximum value, the time to peak is about 20 ms and the steady state error is 7 %.

If K_p is equal to 10 (figure 5.5c), the closed loop response is still stable, but highly underdamped and become marginally stable if a value of 15 is set (figure 5.5d).

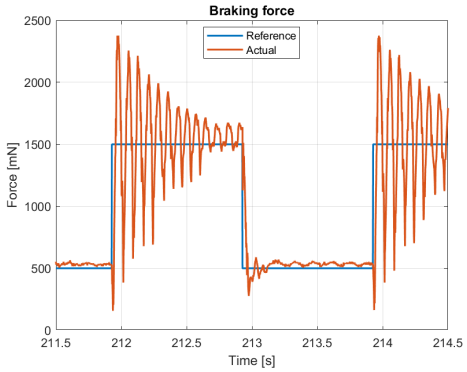
Figures 5.6a and 5.6c on page 59 shows the phase current for two different K_p values, 0 and 5. Here, driver's shunt resistors has been used to sense the phase current.

Those graphs shown the effect of the presence of a force feedback on the phase current. While on figure 5.6a current grows almost linearly, adding a proportional feedback cause

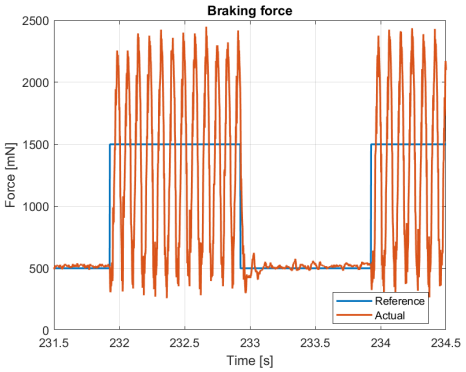


(a) $K_p = 0$

(b) $K_p = 5$



(c) $K_p = 10$



(d) $K_p = 15$

Figure 5.5: Square wave, force.

an increased peak value and a reduction of the steady state mean value. This behaviour is coherent with what is depicted on figure 5.5.

5.4 Triangular wave

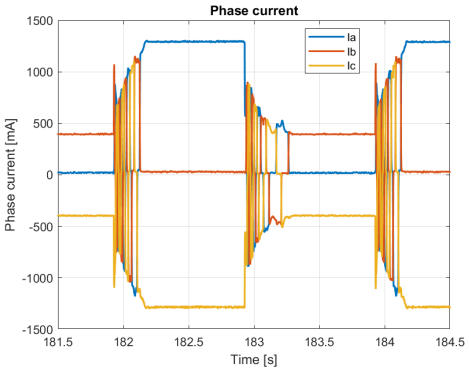
Figure 5.7 report the force and mean phase current behaviour when the reference signal is a triangular wave, at first from 500 mN to 1500 mN and then from 500 mN to 2000 mN.

Here, the K_p has a value of 5 (set at about 65 s).

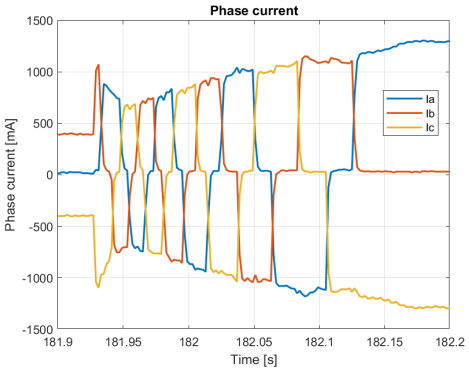
On the left side of both figures it is possible to see the effect of the driver's enable and then the starting sequence.

The undershoot peak when the force is set to the minimum value at the end of the triangular wave is caused by the rubber band *sling effect*. In those points, the force is instantly set to a value lower than the previous, causing the rubber band return force to accelerate the motor instead of braking it (that is what happen when the required force is increased). This is also the cause of the very high current peaks of figure 5.7b.

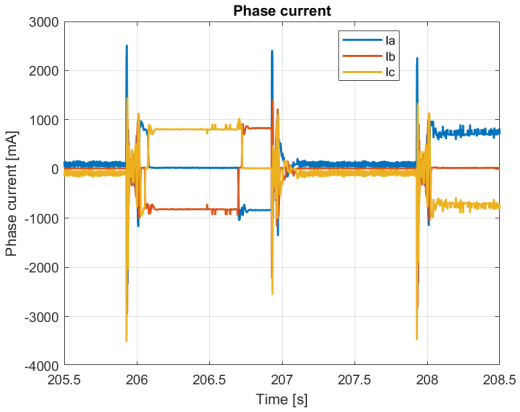
One possible solution to reduce those peaks is decreasing the rate of change of the reference signal, according to handling qualities, including a low pass filter. However,



(a) $K_p = 0$

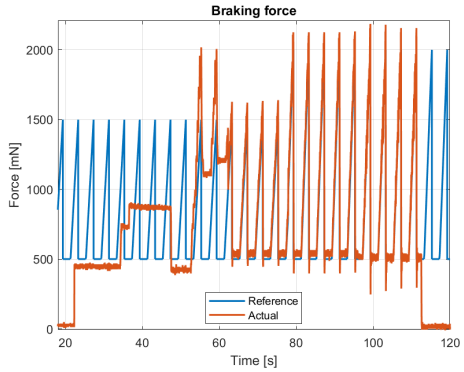


(b) $K_p = 0$, *zoom on initial phase*

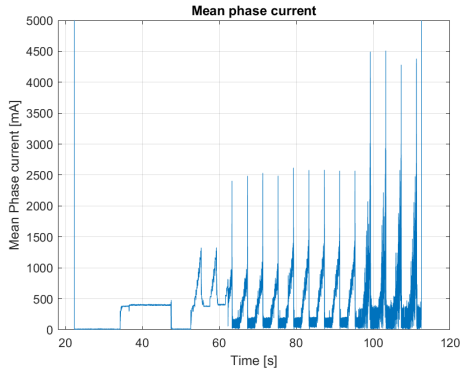


(c) $K_p = 5$

Figure 5.6: Square wave, phase current.



(a) *Force*



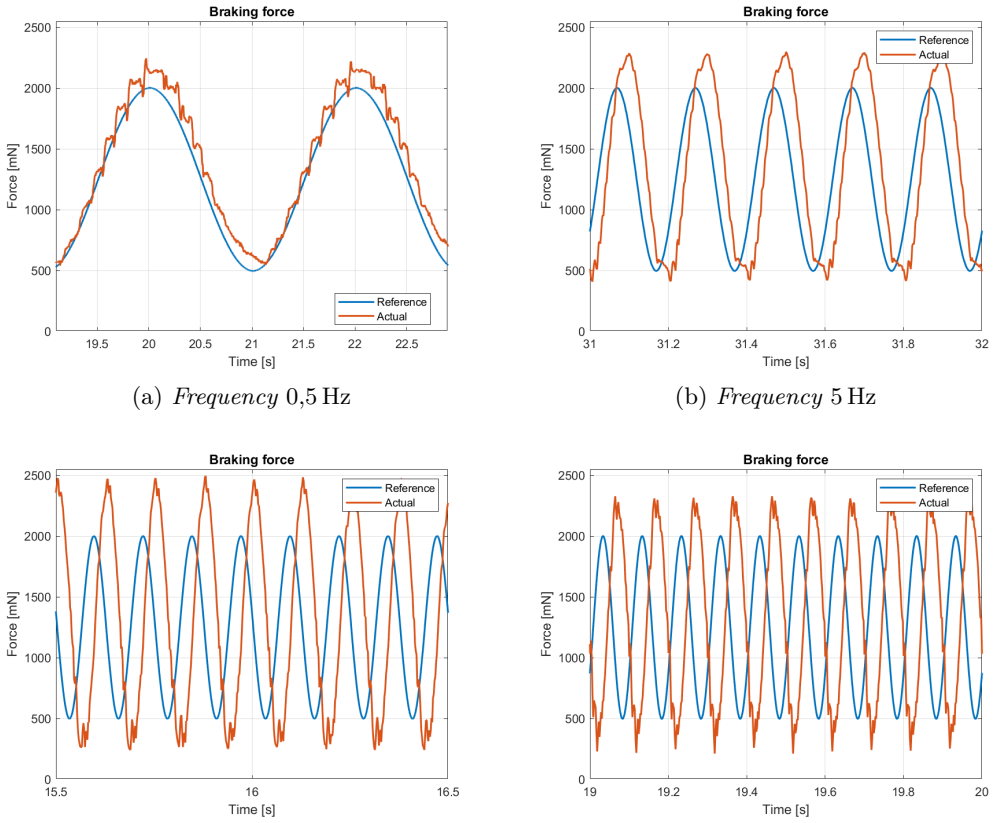
(b) *Mean phase current*

Figure 5.7: Triangular wave.

this feature must be evaluated once the real mechanical actuator can be tested and in accordance with acceptable time and frequency response for braking system.

5.5 Sinus wave

5.5.1 Time response



(a) *Frequency* 0,5 Hz

(b) *Frequency* 5 Hz

(c) *Frequency* 8 Hz

(d) *Frequency* 10 Hz

Figure 5.8: Time response, sinus wave.

Figure 5.8 shown the force behaviour after a sinusoidal reference signal with various frequency, from 0,5 Hz (figure 5.8a) up to 10 Hz (figure 5.8d). For all those tests, the sinus amplitude is 1,5 N, the offset is 1,25 N and the proportional feedback constant K_p is 5.

Increasing the frequency from 0,5 Hz up to 8 Hz will cause an increase in both amplitude and phase offset. When the frequency reaches 10 Hz, it is possible to notice a little decrease in amplitude, while the phase is almost $-180°$. This behaviour hallows to locate the natural frequency of the system between 8 Hz and 10 Hz.

The last graph (figure 5.8d) is of great importance here because the antiskid will work exactly at 10 Hz frequency. Here it has been proved that this software can generate at least a 10 Hz sinus wave and log it without particular effort.

5.5.2 Frequency response

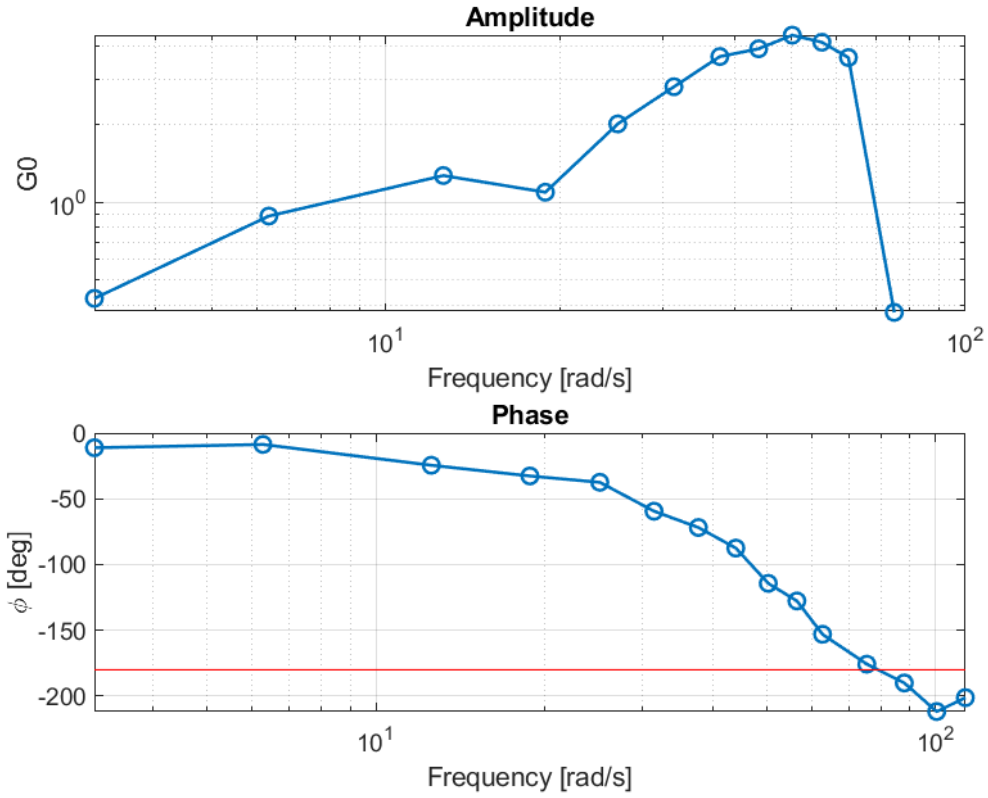


Figure 5.9: Bode diagram.

Figure 5.9 shows the Bode diagram of the system. The red line on phase graph highlight the $-180°$ limit.

These data covered a frequency from 0,5 Hz up to 18 Hz. The maximum amplitude is reached at a frequency of 8 Hz, or 50,27 rad/s.

Reducing the frequency step it became possible to characterize the real mechanical actuator in both time and frequency domain.

It must be declared that these data cannot be directly related to previous test's data because here a different rubber band has been used. The replacement has become necessary because of wear out.

Chapter 6

Conclusions and future development

The main goal of this master's thesis is the development of control system for an electromechanical actuator that will replace classical hydraulic actuator on brake system of a general aviation aircraft.

This software covers all different aspect of a BLDC motor control, from the MOSFETs activation up to high level safety measures.

The core of the software is the MOSFETs management, that has been performed thanks to a 6-step commutation scheme, based on the shaft position derived from three different Hall effect sensors mounted inside of the motor. Despite its easy implementation, if compared with other advanced control logic, this technique guarantees to minimize commutation losses and to obtain the maximum torque ([2]).

After the completion of phase commutation management, the controller has been implemented. The aim of this part of the software is to ensure that the desired reference braking force is achieved. This must be true even in case of load cell failure: for this reason, feedforward and proportional feedback are used together here.

The software has been validated on a dedicated test bench. In order to increase safety during those tests, different safety measures have been implemented.

The test campaign demonstrates that the software is capable of controlling a BLDC motor in a satisfying way, producing the required braking force. The data logging feature allows to gather useful data to characterize the real mechanical actuators. All safety measures developed for those tests remain valid also when the the actuator will be integrated in the test bench.

The BLDC motor's control software has been developed using Simulink environment. During this work, that choice shows all its advantages and disadvantages. There is no doubt that Simulink allows to noticeably decrease the development time thanks to its user-friendly approach and to its dedicated libraries. Unfortunately, this is true only for simple projects. Three are the main reasons behind this statement.

First of all, Simulink was originally developed to simplify the process of finding a solution to numerical problems; this has been done hiding all low level settings (such as the numerical scheme, the step size or the checks needed to find out when divergence occurs) and the result is pretty good. Those settings are still accessible by the user, but the *Auto* feature of Simulink allows to obtain a satisfying solution for simple problems even without almost any deep knowledge of numerical methods. Automatic management of low level

setting is what allows to develop a simple code for Delfino LaunchPad™ reducing the needed time. However, when the software becomes a little more complex, this feature shows all its limitations. Moreover, the library documentation cannot be considered exhaustive. Therefore, simple actions (such as time computation or set a working serial communication), something not so difficult to implement in almost all procedural languages, here are very struggling aspect, requiring a lot of time to be developed.

The other aspect that put some limitations on code development is the use of blocks instead of lines of code. This is not a bad feature at all, but it is good only for high level software development. As an example, the implementation of safety measures logic on Simulink block scheme has not been difficult at all. The most difficult part of it has been creating a flag variable and maintain its state until recovery procedure completion. Another blocks scheme's drawback is that blocks' execution order cannot be controlled. This is because, virtually, all blocks perform their action together, without a predefined order. If it is not a problem while trying to numerically solve a differential equation, this aspect become of great importance when a microcontroller must be programmed.

The last reason why Simulink is not suitable for developing complex code is the lack of advanced logic control blocks. As an example, while all bitwise operations can be natively performed, the if block does not allow an else-if condition. This is, the reason why the position counter block described in subsection 3.2.1 has been included in a Matlab function block instead of writing it using basic Simulink blocks. This aspect is not as serious as the previous ones, it just causes a general slow down of software development.

At this moment, the next step will be to test the software developed in this work using the real mechanical actuator (when at least one of them will be available). This step will allow to properly tune the parameter related to the controller and safety measures, to justify control architecture and safety measures and eventually to highlight the need of software modification.

With the software produced here, it is possible to control a single BLDC motor. However, the brake system architecture requires to contemporary manage four different electro-mechanical actuators with a single microcontroller. Because of the problems encountered during the software development, this is a task that cannot be performed using Simulink environment. Delfino LaunchPad™ was natively designed
to support up to two different BLDC motors. Trying to connect four of them require the use of almost all GPIOs and ADCs, even those which are not directly available on Delfino LaunchPad™ (a dedicated board will be developed). This will lead to an increased workload that cannot be managed by Simulink dedicated compiler.

Now that there is a software capable of control a BLDC motor and to test the real mechanical actuator, it is possible to allocate some time to develop the same software using a lower level language, such as C language. While doing this, some improvement should be made. The first change will be related to time management, an aspect that must be supervised from the beginning of software development. In order to control four different electric motor, a general routine should be introduced. This routine must not only control the motors, but must be capable of failures and data management. This requires that all analog sensors (first of all, the load cells) output are consistent among them. This aspect could require a modification of starting sequence, including an additional phase in order to align each sensor's initial offset.

Finally, this work has been developed during a period of global crisis of the entire

electronic supply chain. The lack of microcontrollers and integrated circuit make any actual project susceptible of modification.

Bibliography

[1] Sokira, Thomas J. and Wolfgang Jaffe (1990), *Brushless DC motors: electronic commutation and controls*, TAB BOOKS.

[2] Matt Hein (2020), *Demystifying BLDC motor commutation: Trap, Sine, & FOC*, Texas Instruments.

[3] Carolus Andrews, Manny Soltero, Mekre Mesganaw (2019) *Brushless DC Motor Commutation Using Hall-Effect Sensors*, Texas Instruments.

[4] https://www.faulhaber.com

[5] https://www.ti.com/tool/LAUNCHXL-F28379D

[6] https://www.ti.com/tool/BOOSTXL-DRV8305EVM

[7] https://www.ti.com/tool/CCSTUDIO

[8] https://www.ti.com/tool/C2000WARE

[9] https://www.mathworks.com/products/simulink.html