



**Politecnico
di Torino**

POLITECNICO DI TORINO

Master of Science in Electrical Engineering

Master Thesis Dissertation

Development of Firmware for a Back-to-Back Motor Test Bench with CANopen Communication Protocol

Supervisors

Prof. Gianmario Pellegrino
Dr. Fausto Stella
Dr. Paolo Pescetto

Candidate

Paolo Ubico

Academic Year 2024 - 2025

Acknowledgements

Nonostante la tesi sia redatta in lingua inglese credo sia doveroso in questo frangente utilizzare l'italiano per meglio esprimere la mia gratitudine per tutti coloro che mi hanno accompagnato in questo percorso universitario.

I primi che voglio ringraziare sono i miei relatori: il professor Pellegrino, il dottor Stella e il dottor Pescetto. Il loro aiuto è stato prezioso sin dai primi giorni in laboratorio fino ad arrivare alla stesura definitiva dell'elaborato.

Il ringraziamento più grande va alla mia famiglia: a mamma e papà i miei primi e più convinti sostenitori, senza dimenticare mia sorella Francesca, zia Luisa, gli zii Dario e Danilo e le nonne Angela e Maria Grazia.

Un sentito ed affettuoso ricordo lo voglio dedicare a chi non ha potuto assistere alla chiusura di questo percorso anche se sono sicuro ne sarebbero stati orgogliosi, un saluto a zia Rita e nonno Baldo.

Tutti, chi in un modo chi nell'altro, ha contribuito alla mia formazione umana e culturale iniziata da bambino accompagnato nei miei primi passi fino all'età della maturità, che posso dire di aver raggiunto concludendo questo percorso universitario. Purtroppo non credo smetterò mai di avere bisogno del vostro aiuto, perciò spero possiate ancora sostenermi e sopportarmi a lungo.

In ultimo mi rivolgo i miei amici, la loro vicinanza, i consigli e la leggerezza che hanno saputo trasmettere sono stati preziosi nel rendere più mite questo viaggio. Alcuni di voi mi conoscono per loro sfortuna da lungo tempo, altri sono arrivati in tempi più recenti, quello che mi auguro è che possiamo continuare uniti verso i nostri futuri obiettivi.

Summary

The objective of this thesis is to implement firmware within the STM32CubeIDE environment for the STM32F303RE microcontroller by STMicroelectronics. This microcontroller is responsible for managing communication with a second device, the Micro Digital One by Microphase, via the CANopen communication protocol, in order to control two SPM motors mechanically coupled in opposition and powered through a common DC link.

The CANopen protocol is implemented at the hardware level using the STM32F303RE's integrated CAN peripherals, and at the software level using the open-source CANopenNode library. In this setup, the STM32F303RE operates as the master, issuing commands to the Micro Digital One, which is configured to receive and execute these commands accordingly.

The two motors are controlled using different strategies: the first operates in speed control mode, following the logic defined by the Micro Digital One, while the second is driven using a Field-Oriented Control (FOC) algorithm implemented directly in STM32CubeIDE and executed by the STM32F303RE.

Contents

List of Tables	IX
List of Figures	x
1 Hardware overview and assembly	1
1.1 System Components	1
1.1.1 STM32Nucleo-64-UM1724 Nucleo Board	1
1.1.2 Inverter Board X-NUCLEO-IHM08M1	3
1.1.3 Microphase Mirco digital One inverter	7
1.1.4 Induction Motors	14
1.1.5 Other required components	15
1.2 Test bench set up configuration	16
2 Microcontroller Configuration and Firmware Implementation	19
2.1 Project Generation	19
2.1.1 Clock Configuration	21
2.1.2 TIM1 Configuration	22
2.1.3 ADC1 Configuration	23
2.1.4 TIM2 Configuration	24
2.1.5 TIM7 Configuration	26
2.1.6 NVIC Configuration	26
2.1.7 IWDG Configuration	28
2.1.8 USART Interface	29
2.2 Project File Organization	30
2.3 Control Code	32
2.3.1 ERROR	37
2.3.2 WAKE-UP	38
2.3.3 COMMISSIONING	39
2.3.4 READY	40
2.3.5 START	41

3	CANopen and Control strategy implementations	43
3.1	Introduction to CANopen	43
3.1.1	CANopen - higher layer protocol	46
3.2	Six core CANopen concepts	47
3.3	CANopen communication basics	50
3.3.1	CANopen communication models	50
3.3.2	The CANopen frame	51
3.3.3	CANopen communication protocols/services	53
3.4	CANopen Object Dictionary	55
3.4.1	OD standardized sections	55
3.5	SDO - configuring the CANopen network	56
3.6	PDO - operating the CANopen network	57
3.6.1	How does the CANopen PDO service work	58
3.7	CANopen network set up in this specific application	58
3.7.1	CANopen in STM32F303RE	58
3.7.2	CANopen in Micro digital One	65
3.7.3	Micro digitl One set up via STM32F303RE	67
4	Control code implementation	71
4.1	Control strategy for the Nucleo Board	71
4.1.1	Field-Oriented Control (FOC)	71
4.1.2	FOC implemented in the Nucelo Board	77
4.2	Control configuration for Micro digital One	78
5	Motors control results	79
5.1	Data retrived from speed control	79
5.2	Data acquired by speed and FOC control over the test bench	80
5.2.1	Analisis of bench test speed	81
5.3	Fault Test	82
6	Additional software	85
6.1	Drive Watcher	85
6.1.1	Functions description and usage	87
6.2	CANopenEditor - EDS editor	98
6.3	PCAN-View	104
6.3.1	Receive	106
6.3.2	Transmit	107
7	Conclusions	109
	Bibliography	111

List of Tables

1.1	Micro digital One models	7
1.2	Micro digital One sizes	7
1.3	Micro digital One paramters	8
1.4	Micro digital One ports	9
1.5	Micro digital One CN2 connector	10
1.6	Micro digital One CN3 connector	11
1.7	Micro digital One CN5 and CN6 connector	12
1.8	Micro digital One CN1 connector	12
1.9	Micro digital One CN4 connector	12
1.10	Micro digital One M1 connector	13
1.11	Induction motors S1601B303 Brushless Servomotor specifics	14
1.12	Induction motors S1402B353 Brushless Servomotor specifics	15

List of Figures

1.1	NUCLEO-F303RE Board	2
1.2	Pinout	2
1.3	Expansion Board X-NUCLEO-IHM08M1	3
1.4	Power MOSFET and Gate Drivers	4
1.5	Shunt Resistors	5
1.6	Conditioning Circuitry and Hardware Protection	6
1.7	Micro digital One I/O reference circuits	9
1.8	Micro digital One ports	10
1.9	Micro digital One top connectors schematic	11
1.10	Micro digital One side connectors schematic	13
1.11	Test bench set up	16
1.12	Scheme for the test bench	17
2.1	STM32CubeMX Overview	19
2.2	MCU and Nucleo Board Configuration	20
2.3	Block Diagram of the Embedded Controller	21
2.4	System Clock Configuration	21
2.5	TIM1 Configuration	22
2.6	ADC1 Configuration	24
2.7	TIM2 Configuration	25
2.8	TIM7 Configuration	26
2.9	NVIC Configuration	27
2.10	IWDG	28
2.11	IWDG Configuration	28
2.12	Pandora Scope	29
2.13	File <i>main.c</i>	32
2.14	Interrupt Handlers and Simulink Integration	33
2.15	Angle & Speed Computation, Feedback Acquisition	35
2.16	PWM Task Organization	36
2.17	ERROR State	37
2.18	WAKE_UP State	38
2.19	COMMISIONING	39
2.20	READY State	40

2.21	START State	41
3.1	7-layer OSI model	46
3.2	Communication Models	47
3.3	Communication Protocols	47
3.4	Device States	47
3.5	Object Dictionary	48
3.6	Electronic Data Sheet	48
3.7	Device Profile Standards	48
3.8	CANopen concepts link together	49
3.9	Master/Slave	50
3.10	Client/Server	51
3.11	Consumer/Producer	51
3.12	CANopen frame	52
3.13	CANopen COB-ID	52
3.14	CANopen OD human-readable form	55
3.15	CANopen SDO communication	57
3.16	SDO and PDO message comparison	57
3.17	CANopen PDO message	58
3.18	canopen_config.c source file	59
3.19	canopen_config.h source file	59
3.20	CO_SDOclient_utils.c source file	61
3.21	CO_SDOclient_utils.h source file	62
3.22	while(1) from main.c	63
3.23	HAL_TIM_PeriodElapsedCallback function	64
3.24	Drive Watcher software	65
3.25	Parameters and Registers configuration sections	66
3.26	canopen_config.c	67
3.27	handle_off_state	68
3.28	handle_pause_state	68
3.29	handle_on_state	69
4.1	Induction Motor Diagrams	71
4.2	Torque Generation	72
4.3	Block Diagram of FOC	73
4.4	Block Diagram of the IFOC	74
4.5	Block Diagram of the DFOC	74
4.6	Block Diagram of the Indirect FOC	75
4.7	FOC Code, Ctrl_type 3	77
5.1	Measurements for speed control and FOC control	80
5.2	Test bench speed confrontation	81
5.3	Test bench speed confrontation	82
6.1	Drive Watcher	85
6.2	Drive Watcher	87

6.3	Sections	87
6.4	Open configuration file	89
6.5	Save configuration file	89
6.6	Read from device	90
6.7	Write from device	90
6.8	Save configuration	90
6.9	Load default configuration	90
6.10	Start communication with device	91
6.11	Stop communication with device	91
6.12	Register configuration panel	92
6.13	Load default configuration	92
6.14	Forward button	93
6.15	Continuously reading	93
6.16	Data format	93
6.17	Driver Easy set-up panel	94
6.18	USB configuration	95
6.19	Communication settings	96
6.20	Com-test	96
6.21	Communication settings and test section	97
6.22	OD editor first look	98
6.23	OD editor opening file .xpd	99
6.24	OD editor with file .xpd open	99
6.25	ODE - Device Info	100
6.26	ODE - Object Dictionary	101
6.27	ODE - TX PDO Mapping	102
6.28	ODE - RX PDO Mapping	103
6.29	PCAN-View main page	105
6.30	PCAN-View Receive section	106
6.31	PCAN-View Transmit section	107
6.32	Example of a creation of a message in PCAN-View	107

Chapter 1

Hardware overview and assembly

The main goal of this first chapter is to provide an overview of the hardware components involved in the experimental set up. The system is composed of an STM32 Nucleo Board with a paired inverter, a Microphase Micro Digital One inverter a transceiver and two coupled induction motors. These componets assembled create the system for the control in back to back configuration.

1.1 System Components

The bench test consists of several components, each contributing to the realisation of the tests. These include a microcpntroller unit (MCU) mounted on the STM32 Nucleo Board, the two invertes responsible for power delivery to the two motors and the transceiver connecting the micro via CANopen. In the following sections detailed explanation of each component will be provided, focusing on their roles and how they interact within the system.

1.1.1 STM32Nucleo-64-UM1724 Nucleo Board

The STM32 Nucleo-64 boards represent cost-effective and versatile development platforms, specifically designed to support users in evaluating and initiating development with STM32 microcontrollers in 64-pin LQFP packages. These microcontrollers are built on widely adopted ARM Cortex cores and incorporate an on-board ST-LINK debugger/programmer, thereby removing the need for external debugging hardware. Additionally, the boards support Arduino Uno V3 connectivity as well as ST morpho headers, allowing for seamless integration with a broad range of expansion shields. The specific model utilized in this work is the 64-pin NUCLEO-F303RE board, which is based on the STM32F303RE microcontroller. This MCU offers the following key features:

- ARM Cortex-M4 core (CPU) running at an internal clock speed of 72 MHz ;

- 512 Kbytes of Flash memory;
- Four ADCs with selectable resolution ranging from 6 to 12 bits;
- 14 timers, including PWM channels, pulse counting and encoder signals;
- Communication interfaces such as a CAN interface, up to 5 USARTs and up to 4 SPI channels.

Figure 1.1 depicts the Nucleo board used in this application.

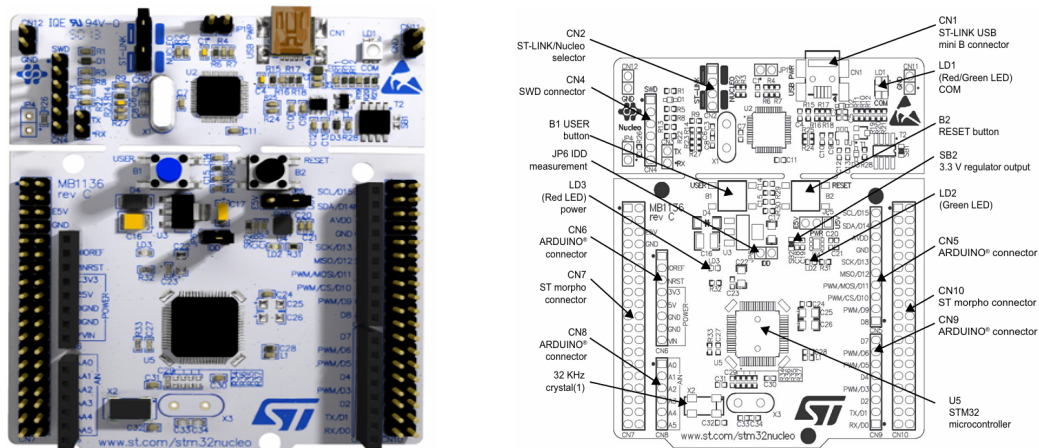
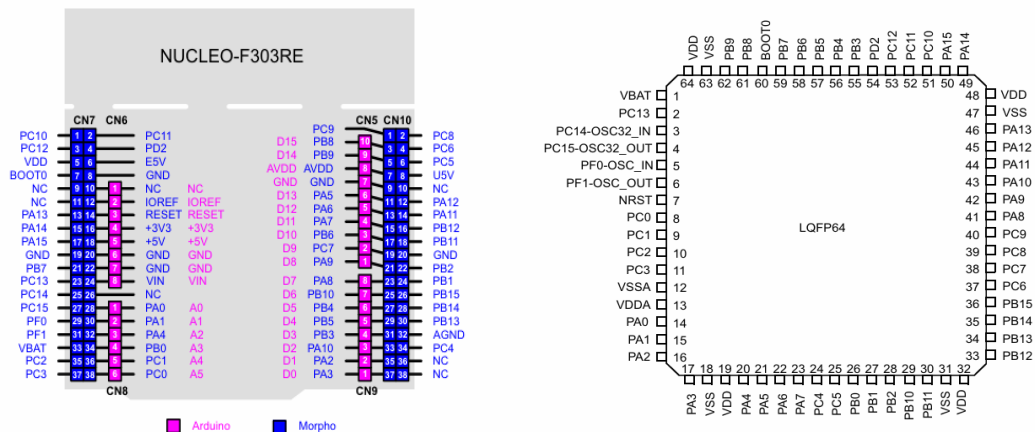


Figure 1.1: NUCLEO-F303RE Board

The pins of the MCU are accessible through the side connectors of the board. Figure 1.2 displays the pinout configurations of both the Nucleo board and the LQFP64 package.



(a) NUCLEO-F303RE Pinout

(b) LQFP64 Package Pinout

Figure 1.2: Pinout

1.1.2 Inverter Board X-NUCLEO-IHM08M1

The X-NUCLEO-IHM08M1 is a three-phase inverter expansion board developed for use with STM32 Nucleo platforms. It integrates the STL220N6F7 STripFET™ F7 Power MOSFET and the L6398 half-bridge gate driver, delivering an efficient and compact solution for motor control applications. The board supports both sensorless and sensor-based control strategies and includes configurable jumpers to enable either single-shunt or three-shunt current sensing. Full compatibility with STM32 Nucleo boards is ensured, and the presence of ST morpho connectors allows straightforward integration and expansion. An overview of its connection and configuration is provided in Figure 1.3.

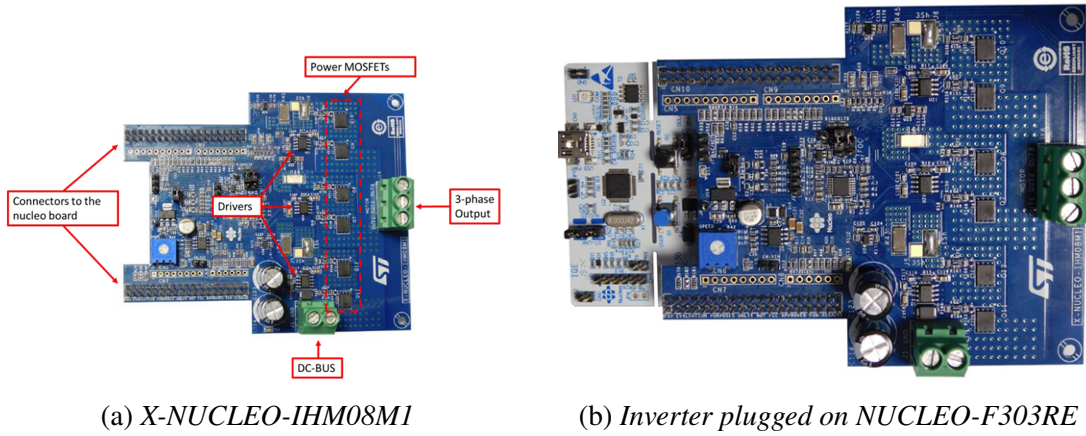


Figure 1.3: Expansion Board X-NUCLEO-IHM08M1

The schematic of the inverter's power section is shown in Figure 1.4. The DC-link requires a supply voltage between 8 V and 48 V, and the MOSFET bridge is realized using 60 V STL220N6F7 discrete devices. Each leg of the three-phase inverter is driven by a dedicated gate driver IC, such as U22 for phase 1. The labels TIM1_Cxx indicate the gate driver commands generated by the MCU of the Nucleo board, such as TIM1_CH1 and TIM1_CH1N for phase one.

Gate Driver and Bootstrap

The L6398 integrated circuit employed on this STM32 Nucleo expansion board is a high-voltage gate driver developed by STMicroelectronics. It is capable of driving N-channel power MOSFETs in half-bridge configurations and supports supply voltages up to 600 V. Each gate driver stage utilizes a $1\ \mu\text{s}$ external bootstrap capacitor to supply the floating high-side drive circuitry. In this context, floating refers to a node not referenced to ground.

As illustrated for phase one in Figure 1.4, when the low-side MOSFET Q12 is conducting, the bootstrap diode D15 becomes forward-biased, allowing the bootstrap capacitor C119 to charge to approximately +15 V. When Q12 is turned OFF and the high-side MOSFET Q11 is activated, the output at node OUT1 rises to the DC-link voltage (exceeding 15 V), reverse-biasing diode D15. At this point, capacitor C119, now disconnected from ground, serves as a 15 V floating supply referenced to OUT1, enabling proper high-side gate drive.

While the bootstrap technique is a simple and cost-efficient method for supplying the high-side driver, it does come with several inherent limitations:

- Periodically, the low-side switch must turn ON to recharge the bootstrap capacitor;
- Lower switching frequencies require a larger bootstrap capacitor to ensure stable operation.

As discussed later, it is crucial to limit the duty cycle within a narrower range to ensure proper recharging of the bootstrap capacitor during each PWM cycle, preventing issues with the high-side gate drive performance. In the presented control system, duty cycles are saturated between 0.05 and 0.95.

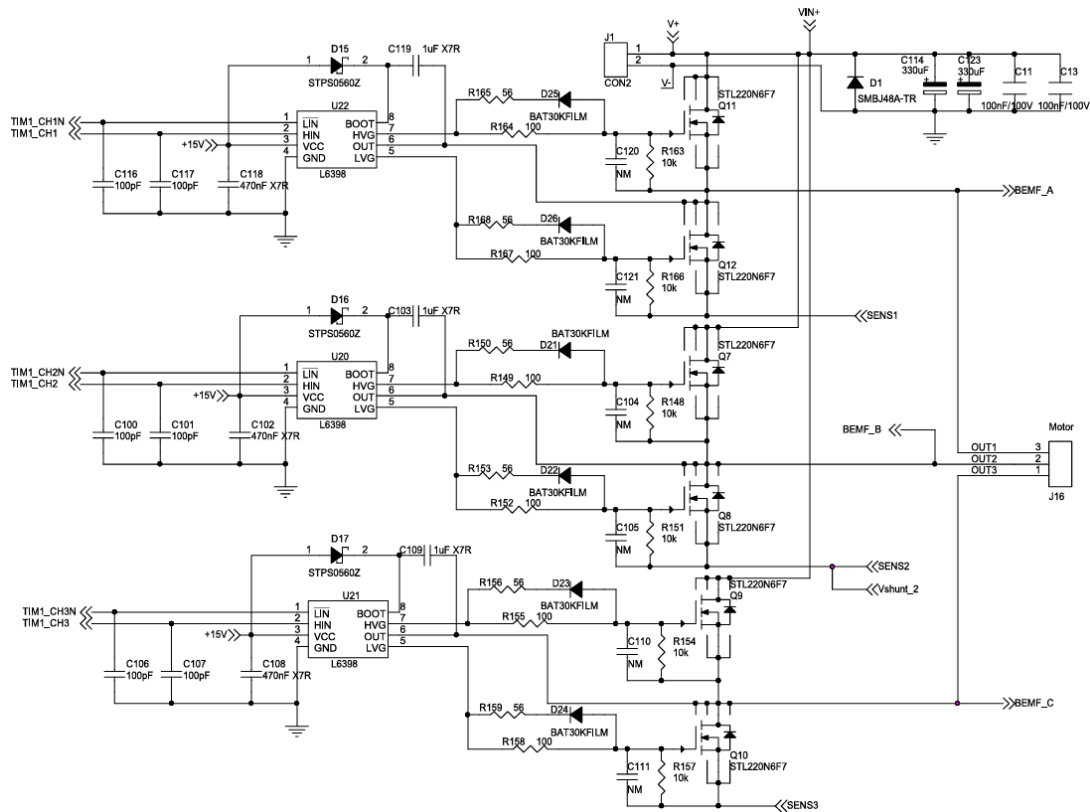


Figure 1.4: Power MOSFET and Gate Drivers

Shunt Resistors and Overcurrent Protection

Figure 1.5 shows the location of three shunt resistors positioned at the bottom of each inverter leg. These components are responsible for detecting the current flowing through the respective low-side MOSFETs. When a low-side MOSFET is active, the current passing through its corresponding shunt resistor effectively represents the motor phase current, albeit with a negative sign. As previously outlined, reliable current sampling within each PWM cycle requires appropriate saturation of the duty cycles. This condition ensures that the low-side switches remain active for a sufficient duration during every switching period, thereby enabling the acquisition of valid current measurements.

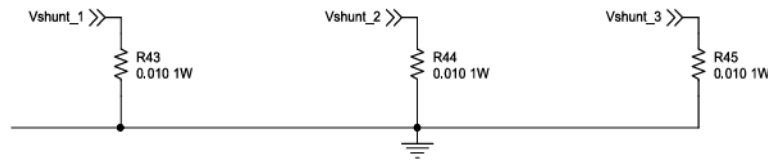


Figure 1.5: Shunt Resistors

The analog conditioning circuitry depicted in Figure 1.6 serves to buffer the voltage drop across the shunt resistors and rescale it to fit within the 0–3.3 V input range of the microcontroller’s ADC channels. This rescaling is achieved through a combination of gain and offset, defined by the resistor network within the circuit. However, due to the relatively high tolerance of these resistors, precise current measurements cannot be ensured without recalibrating the offset digitally at each motor startup.

Once conditioned, the analog signal within the 0–3.3 V range is sampled by a 12-bit ADC and converted into a digital value, which is then read in real time directly from the ADC register.

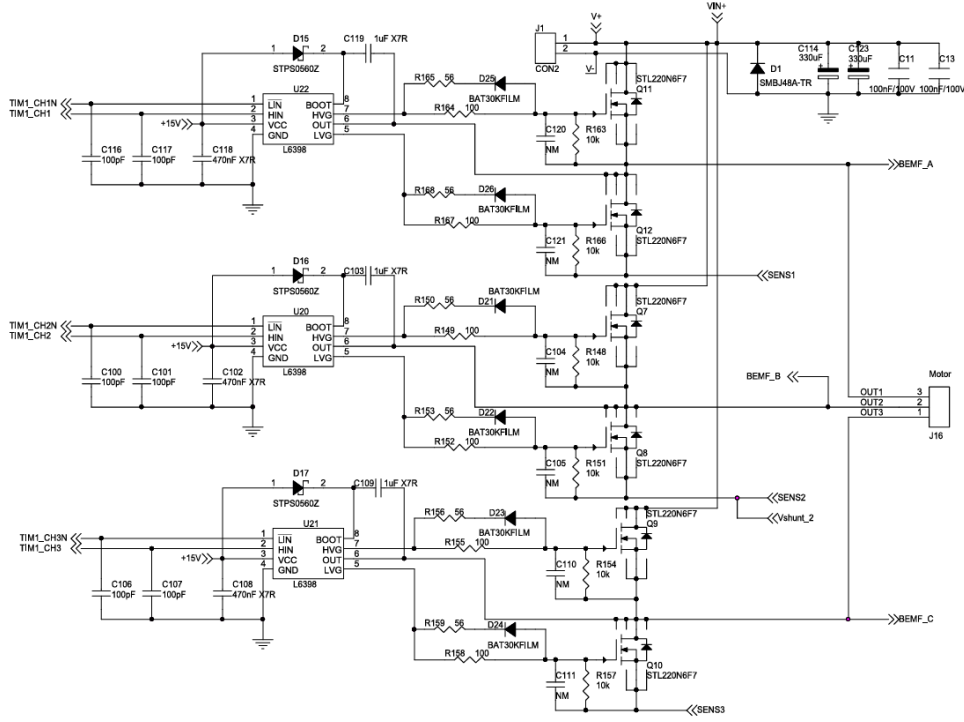


Figure 1.6: Conditioning Circuitry and Hardware Protection

The X-NUCLEO-IHM08M1 is rated for 15 A ($22 A_{pk}$) output current. Figure 1.6 also highlights the hardware overcurrent protection circuit, which is configured with a threshold of $30 A_{pk}$. When the current through a shunt resistor exceeds the reference voltage V_{ref} , the output of the comparator (BKIN) is pulled low (to GND). This BKIN signal is connected to the $TIM1_{BKIN}$ input of the microcontroller, which allows the timer TIM1 to immediately and asynchronously disable the PWM modulation signals by triggering its Break function.

1.1.3 Microphase Mirco digital One inverter

The Micro digital One is a very compact full digital regenerative servo drive for permanent Brushless and Brush DC servomotors. The Micro Digital One drive features a high-integration RISC microprocessor, enabling a compact design, high dynamics, and exceptional resolution in speed and positioning control.

It can operate in torque, speed, and positioning control modes and supports fieldbus communication protocols such as S-NET, S-CAN, Modbus RTU, and CANopen.

Models

There are available two different models due to different range for voltage operation. They take the name of *Modello 65* and *Modello 100*.

Table 1.1: Micro digital One models

Models	Voltage range	Nominal voltage
<i>Modello 65</i>	20-84 V_{DC}	65 V_{DC}
<i>Modello 100</i>	30-130 V_{DC}	100 V_{DC}

It is also possible choose from different current sizes for the two models.

Table 1.2: Micro digital One sizes

Sizes	Nominal current	Peak voltage
2/4	2 A	4 A
4/8	4 A	8 A
7/14	7 A	14 A
10/20	10 A	20 A

For this specific application is used the *Modello 65* size 10/20.

Programming

Before using Micro Digital One, it is necessary to configure certain parameters based on the motor and the selected application.

To modify these parameters, the dedicated Drive Watcher software (version 4.03 or later) must be used on a Windows-based PC, connecting to the drive via an RS422 serial interface through one of the CN5 or CN6 connectors.

A comprehensive set of parameters is available to optimize the system according to specific requirements. Additionally, dedicated software tools are provided for more complex mechatronic functions.

The Drive Watcher program enables an in-depth analysis not only of the drive's operational variables but also of the entire dynamic system, including the motor and load.

Using the program's utility, it is possible to graphically monitor and store key variables such as current, speed, and voltage, facilitating a precise assessment of the torque demand. This, in turn, helps optimize motor sizing.

The generated graphs can be either printed or saved as files for further analysis.

Onboard Diagnostics

The onboard diagnostic system enables real-time monitoring of the drive's status and verification of its proper operation.

The drive is equipped with protective measures against power stage short circuits, ground faults, overcurrent, overvoltage, and encoder cable disconnection.

These faults are logged in the system memory. Multiple alarms are stored and can be retrieved as long as the drive's service power supply (+24 V_{DC}) remains active

Serial Communication

The RS422 serial port is available as a standard feature, allowing the Micro Digital One drive to be connected to a PC for programming and debugging via a standard USB/RS422 interface. Additionally, the serial interface supports Modbus communication, enabling remote control and integration into industrial automation systems.

Technical Specifications

In the following table are reported the most relevant parameters of the Micro one Digital.

Table 1.3: Micro digital One paramters

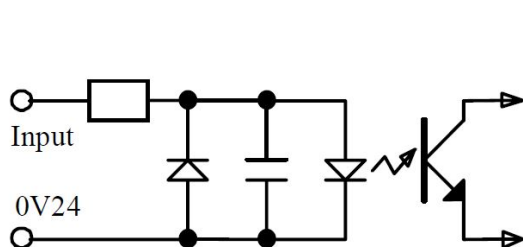
Signal	Description
V_{DC}	Power Supply 20-84 V
$V_{DCI/O}$	Control Section Power Supply and Digital Inputs 24 V \pm 15 , 2 A
A	Nominal output current 10 A_{rms}
f_{sw}	Swithcing frequency 20 kHz PWM

Interface Description

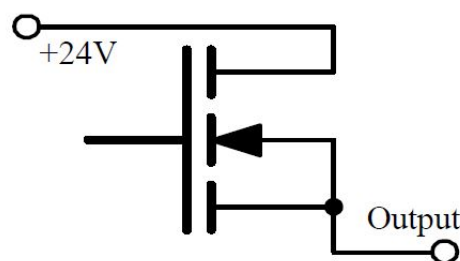
In this paragraph are listed and briefly discussed all the port present on the Micro digital One which give its ability to connect with other devices.

Table 1.4: Micro digital One ports

Port	Description
Motor Hall Sensor Inputs	The inputs are single-line type operating at 0-5 V and can be connected to 5 V line driver encoders, using either the direct signal or a push-pull configuration
Motor Encoder Inputs	The inputs are single-line type, operating at 0-5 V, and can be connected to 5 V line driver encoders, using either the direct signal or a push-pull configuration.
Serial Port	Dual RS422 (opto-isolated)
CAN port	Dual and opto-isolated
Isolated Digital Inputs	The system includes 8 isolated digital inputs operating at 24 V with an impedance of 2.2 k Ω . - Logic level 1: Input signal at +24 V _{DC} - Logic level 0: Input signal at 0 V or disconnected
Isolated Digital Outputs	1 relay output: 24 V, 100 mA, used for drive OK signaling. 4 configurable outputs: 24 V, 0.5 A each.



(a) Digital input reference circuit



(b) Digital output reference circuit

Figure 1.7: Micro digital One I/O reference circuits

Connectors

The variuos ports prenent on the Micro digital One are placed along the front side and the upper side of the cover like shown in figure 1.8.

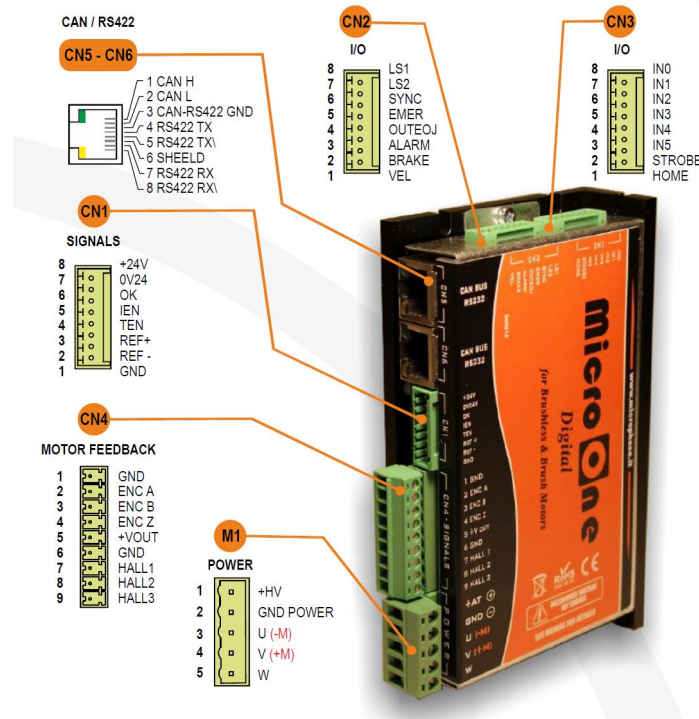


Figure 1.8: Micro digital One ports

In the upper side are presents two connectors **CN2** and **CN3**. In table 1.5 and 1.6 are briefly described the connectors pinout whose labels are also visible in figure 1.8.

Table 1.5: Micro digital One **CN2** connector

N° pin	Signal	Signal description
1	LS1	Input 24 V limit switch 1
2	LS2	Input 24 V limit switch 2
3	SYNC	Input 24 V SYNC
4	EMER	Input 24 V EMER
5	OUTEOJ	Output OUTEOJ 24 V, 0.5 A
6	ALARM	Output ALARM 24 V, 0.5 A
7	BRAKE	Output BRAKE 24 V, 0.5 A
8	VEL	Output VEL 24 V, 0.5 A

Table 1.6: Micro digital One **CN3** connector

N° pin	Signal	Signal description
1	IN0	Configurable input 24 V
2	IN1	Configurable input 24 V
3	IN2	Configurable input 24 V
4	IN3	Configurable input 24 V
5	IN4	Configurable input 24 V
6	IN5	Configurable input 24 V
7	STROBE	Input STROBE 24 V
8	HOME	Input HOME 24 V

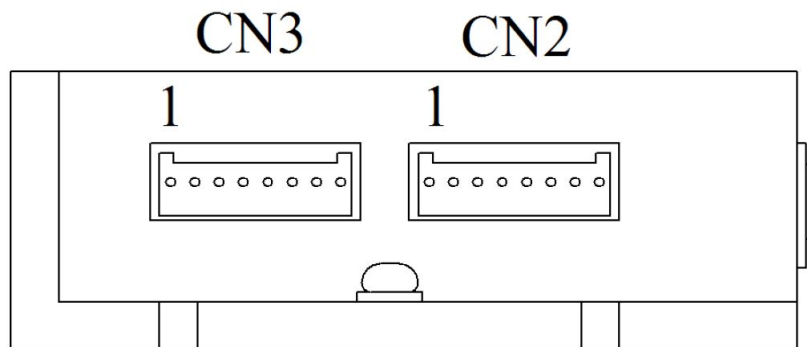


Figure 1.9: Micro digital One top connectors schematic

The majority of the connectors are placed on the side of the Micro digital One and in this application are the most used, connecting the Micro to PC with **CN5** or **CN6** (table 1.7), supplying the logic with **CN1** (table 1.8), monitoring the motor with **CN4** (table 1.9) and supplying the motor with **M1** (table 1.10).

Table 1.7: Micro digital One **CN5** and **CN6** connector

N° pin	Signal	Signal description
1	CAN H	Signal CAN H
2	CAN L	Signal CAN L
3	CAN GND	CAN refernece / RS422
4	TX	RS422 TX
5	TX(-)	RS422 TX(-)
6	SH	Shield
7	RX	RS422 RX
8	RX(-)	RS422 RX(-)

Table 1.8: Micro digital One **CN1** connector

N° pin	Signal	Signal description
1	24 V	Input 24 V
2	0V24	Signal reference for 24 V iput
3	OK	Output 24 V
4	IEN	Input 24 V movement enablement
5	TEN	Input 24 V coupling enabling
6	REF	Positive input 10 V
7	REF/	Negative input -10 V
8	SH	Analog signal reference

Table 1.9: Micro digital One **CN4** connector

N° pin	Signal	Signal description
1	GND	Ground for encoder
2	ENC A	Input encoder A 0-5 V
3	ENC B	Input encoder B 0-5 V
4	ENC Z	Input encoder Z 0-5 V
5	+V (OUT)	Onput 5 V (150 mA)
6	GND	Ground for hall
7	Hall 1	Input hall 1 0-5 V
8	Hall 2	Input hall 2 0-5 V
9	Hall 3	Input hall 3 0-5 V

Table 1.10: Micro digital One **M1** connector

N° pin	Signal	Signal description
1	+HV	Power supply for motor
2	GND	Power supply reference for motor
3	U (-M)	Phase U (positive DC motor)
4	V (+M)	Phase V (negative DC motor)
5	W	Phase W

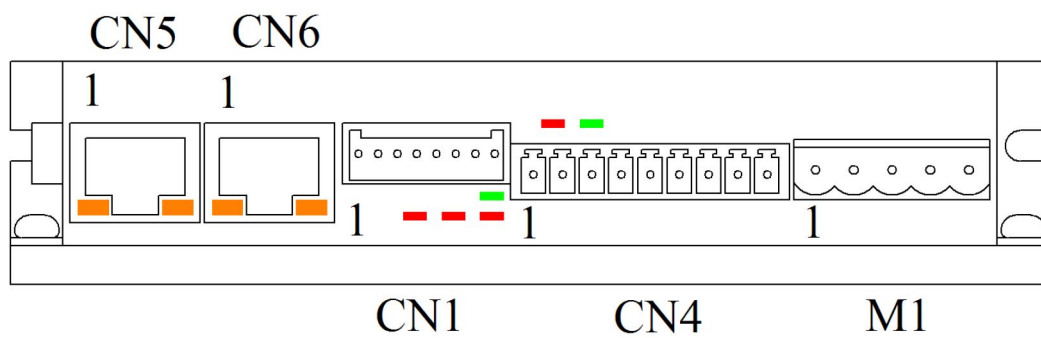


Figure 1.10: Micro digital One side connectors schematic

1.1.4 Induction Motors

For this application two induction motors are coupled together. One motor is the **S1402B353 Brushless Servomotor** and **S1601B303 Brushless Servomotor** both provided by Microphase.

They have some common aspects:

- Sinusoidal B.E.M.F.;
- Rare earth magnets (NdFeB);
- Special falanges and shaft;
- Facoder 2048PPR 5V LD or absolut encoder;
- Flying screw connectors;
- Low rotor inertia.

In the following tables are listed the specifics of the two induction motors.

Table 1.11: Induction motors **S1601B303 Brushless Servomotor** specifics

Parameter	Symbol	Value
Stall torque	M_0 (Nm)	0.35
Peak torque	M_{pk} (Nm)	2.60
Rated torque	M_N (Nm)	5.30
Torque constant	K_T	0.159
Voltage constant	K_E ($V_{rms} / krpm$)	9.36
Stall current	I_{T0} (A_{rms4})	5.45
Rated current	I_N (A)	5.30
Peak current	I_{MAX} (A_{rms})	16.4
Rated power	P_N (W)	250
Rated speed	N_N (rpm)	3000
Max speed	N_{MAX} (rpm)	4000
Rotor inertia	J_R ($kg * cm^2$)	0.19
Winding resistance	R_{U-V} (ohm)	0.85
Winding inductance	L_{U-V} (mH)	2.60
Weight	M (kg)	1.2
Max radial load	(N)	250
Max axial load	(N)	80
Time rating	-	Continuos
Level of protection	-	IP55
Insulation class	-	FClass

Table 1.12: Induction motors **S1402B353 Brushless Servomotor** specifics

Parameter	Symbol	Value
Stall torque	M_0 (Nm)	0.35
Peak torque	M_{pk} (Nm)	0.96
Rated torque	M_N (Nm)	0.32
Torque constant	K_T	0.05
Voltage constant	K_E (V_{rms} / $krpm$)	3.15
Stall current	I_{T0} (A_{rms4})	6.7
Rated current	I_N (A)	6.6
Peak current	I_{MAX} (A_{rms})	18
Rated power	P_N (W)	100
Rated speed	N_N (rpm)	3000
Max speed	N_{MAX} (rpm)	5000
Rotor inertia	J_R ($kg * cm^2$)	0.06
Winding resistance	R_{U-V} (ohm)	0.5
Winding inductance	L_{U-V} (mH)	0.53
Weight	M (kg)	0.65
Max radial load	(N)	120
Max axial load	(N)	80
Time rating	-	Continuos
Level of protection	-	IP55
Insulation class	-	FClass

1.1.5 Other required components

In order to connect via CANopen the Nucleoboard with the inverter and the Micro digital One a **SN65HVD230** CAN transceiver (3.3 V) is used. A PCAN probe can be used paire with the PCAN-View software with the purpose of monitornig the CANopen net and with the possibility of sendind message to the nodes of the net directly from PCAN-View.

1.2 Test bench set up configuration

In the following Figure 1.11 a look to the test bech is provided.

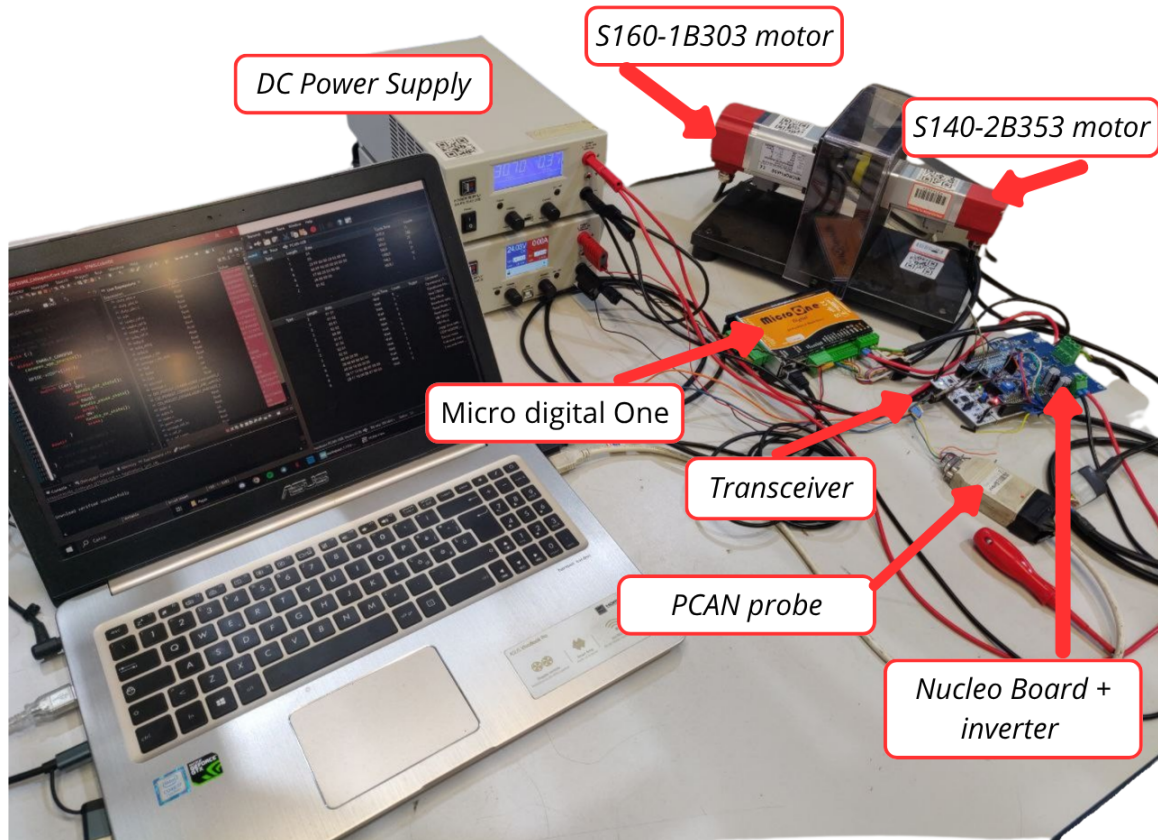


Figure 1.11: Test bench set up

As can be seen two DC power supply are needed for the Micro digital One, the first supply the logic and the second supply the power part. The STM32F303RE use the second DC power supply for the inverter portion while the nucleo is supply by the USB port. The set up is completed with the trwo motors in back-to-back configuration, the transceiver for CANopen communication and the PCAN probe for reading and sending messages into the network.

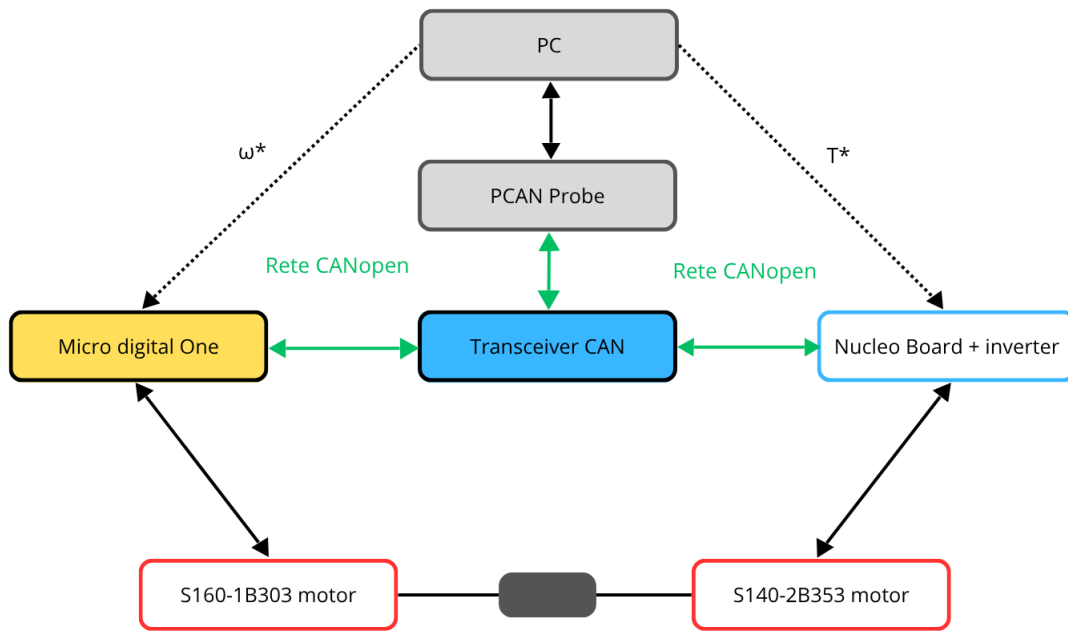


Figure 1.12: Scheme for the test bench

The scheme in Figure 1.12 can be clarify the interaction of the CANopen network between the two microcontroller driving the motors.

Chapter 2

Microcontroller Configuration and Firmware Implementation

This chapter presents a comprehensive description of the development of the embedded firmware and the configuration of the microcontroller peripherals employed in the project. It details the procedures involved in automatic code generation, the configuration of hardware timers, the initialization of key peripheral modules, and the implementation of the CANopen communication protocol. These activities are fundamental to enabling reliable data exchange between the STM32F303RE-based Nucleo Board and the Micro Digital One system.

2.1 Project Generation

The real-time control project is developed using **STM32CubeMX**, a graphical configuration tool provided by STMicroelectronics. It simplifies the initialization of microcontroller peripherals, including pin assignment and peripheral setup.

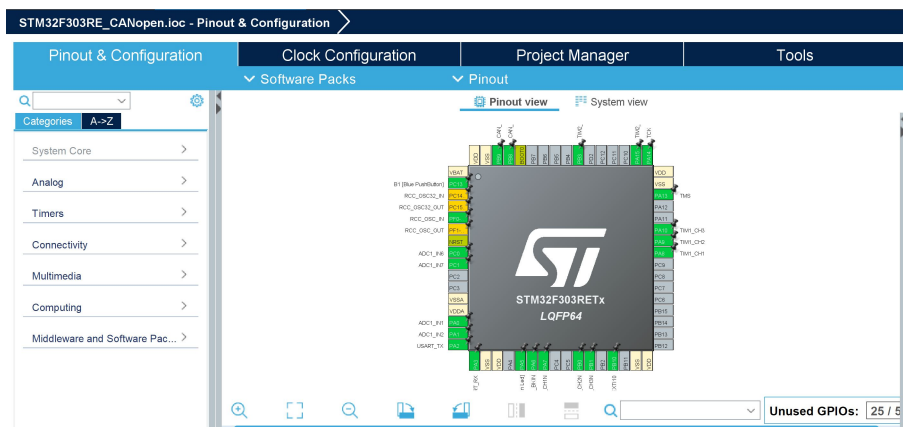


Figure 2.1: STM32CubeMX Overview

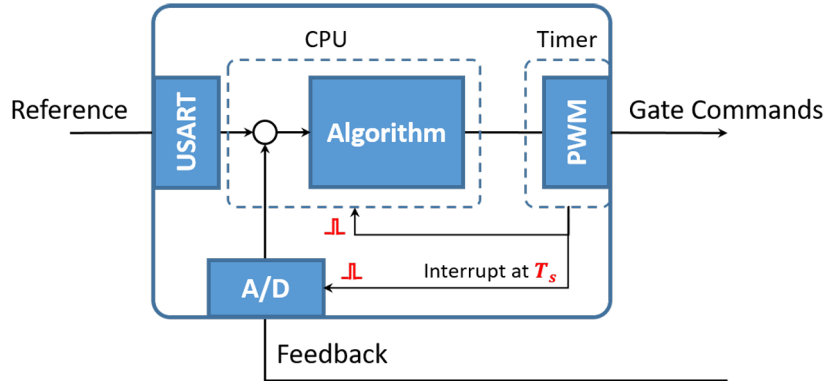


Figure 2.3: Block Diagram of the Embedded Controller

2.1.1 Clock Configuration

The System Clock is responsible for managing the internal timing of the MCU and distributing clock signals to all peripherals. Figure 2.4 illustrates the adopted configuration, in which each peripheral receives an input clock frequency of 72 MHz .

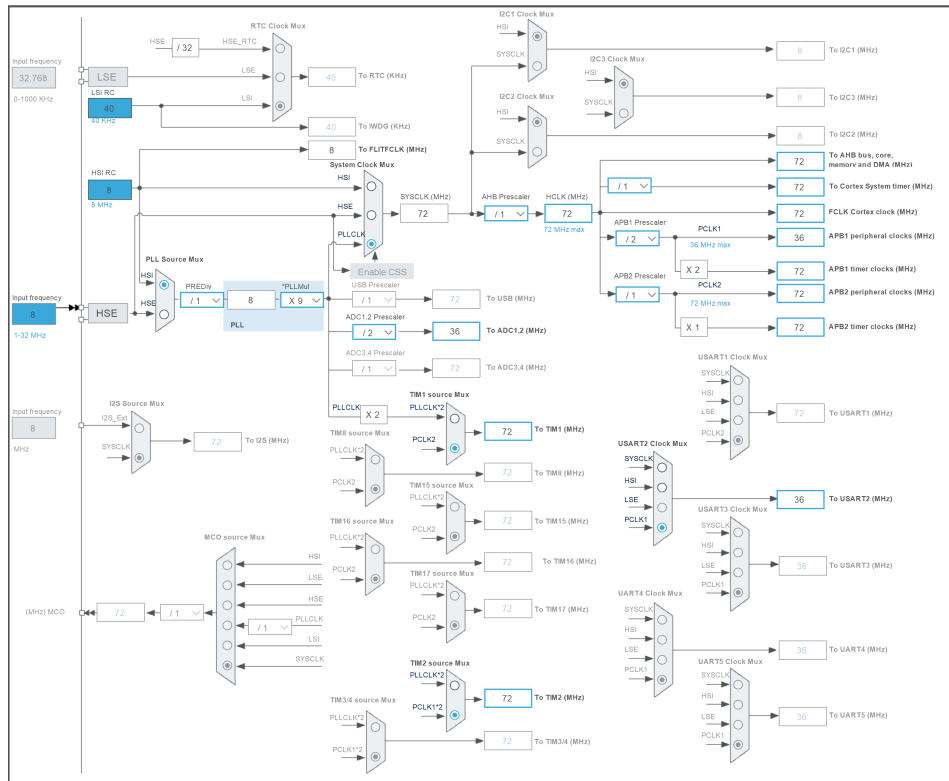


Figure 2.4: System Clock Configuration

2.1.2 TIM1 Configuration

Figure 2.5 shows the configuration of Timer TIM1. Channels CH1, CH2, and CH3 are set to generate PWM signals that control the three legs of the inverter. Each channel provides two complementary outputs: CHx for the high-side MOSFET and CHxN for the low-side MOSFET. Additionally, CH4 is configured to generate a trigger signal for ADC1, enabling synchronized sampling of the three-phase currents and the DC-link voltage.

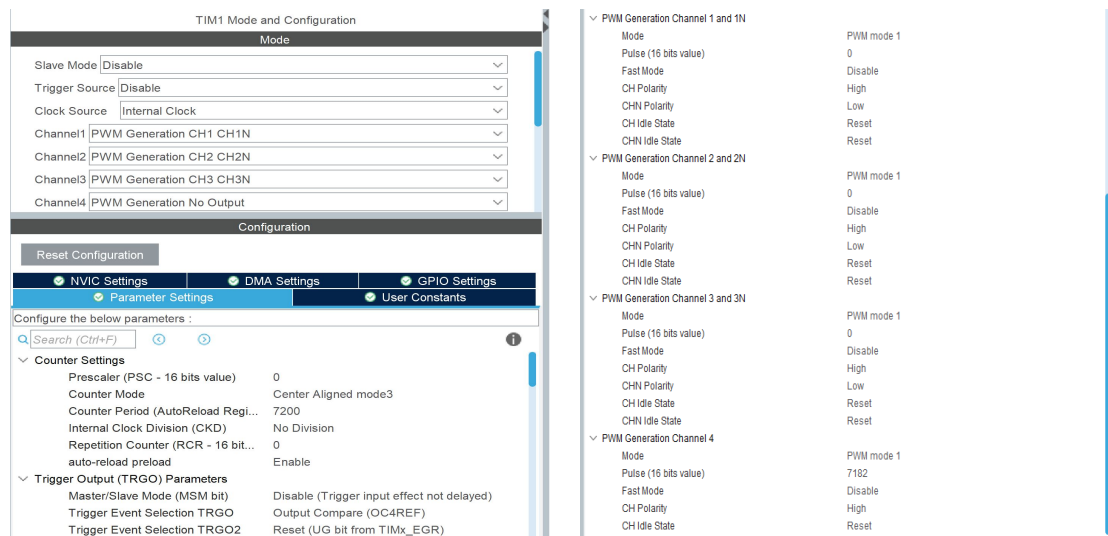


Figure 2.5: TIM1 Configuration

These settings configure TIM1 in an up/down counting mode to emulate a symmetric triangular PWM carrier. The counter increments up to the value in the Auto-Reload Register (ARR), set to 7200, then decrements back to zero. With a 72 MHz input clock, each counter step occurs every 13.8 ns . This results in a PWM carrier frequency of 5 kHz , as the full up-down cycle takes $200\text{ }\mu\text{s}$.

TIM1 also generates update events to synchronize key system operations. These events trigger both the Interrupt Service Routine (ISR) for control code execution and the ADC1 conversion process. Operating in Center-Aligned Mode 3, TIM1 produces update events at both the rising and falling edges of the triangle waveform—i.e., at the counter's zero and ARR values—resulting in two update events per PWM period.

To limit ISR execution to the positive peak (ARR value), the Repetition Counter Register is used. Initially set to 0, it is later configured to 1 in the user code, allowing an ISR call on every second update event. This setup is critical because the phase currents, sensed via low-side shunt resistors, can only be accurately sampled when the low-side MOSFETs are conducting.

Channel 4 of TIM1 is configured in PWM mode, but instead of generating a PWM output, it produces a TRG0 trigger output when the counter reaches 7182. This occurs

0.25 μs before the ISR is invoked (when the counter reaches 7200). This timing ensures that ADC conversions begin just prior to ISR execution, aligning current and voltage sampling with the ideal point in the PWM cycle.

2.1.3 ADC1 Configuration

Due to physical pin mapping constraints between the inverter and the Nucleo board, it is not feasible to utilize all four ADC units available on the microcontroller. Consequently, only ADC1 is employed to sample the three-phase currents and the DC-link voltage.

ADC1 measures:

- Phase current i_a (channel 1),
- Phase current i_b (channel 7),
- Phase current i_c (channel 6),
- DC-link voltage v_{dc} (channel 2).

ADC1 operates in injected conversion mode, which allows a sequence of conversions to be triggered by an external event—specifically, the TRG0 signal generated by TIM1 Channel 4. Once triggered, ADC1 performs the four conversions in sequence, storing each result in a dedicated JDATAx register.

Ideally, all three-phase currents should be sampled at the positive peak of the triangular PWM carrier, where they most accurately reflect their average values. However, since only one ADC unit is available, the currents cannot be sampled simultaneously.

Each 12-bit ADC conversion takes approximately 0.25 μs , so sampling the three currents requires 0.75 μs in total. To align the samples as closely as possible with the PWM peak, TRG0 is configured to occur 0.25 μs before the ISR is triggered at the triangle's maximum. This scheduling ensures that the first current sample (i_a) is taken at the optimal point, while i_b and i_c follow with minimal delay, still remaining close to the peak.

The final conversion in the sequence is the DC-link voltage, v_{dc} , which is not as time-sensitive as the phase current measurements. Therefore, its placement at the end of the sequence does not impact control performance.

Figure 2.6 illustrates the ADC1 configuration and the detailed timing of the conversion sequence.

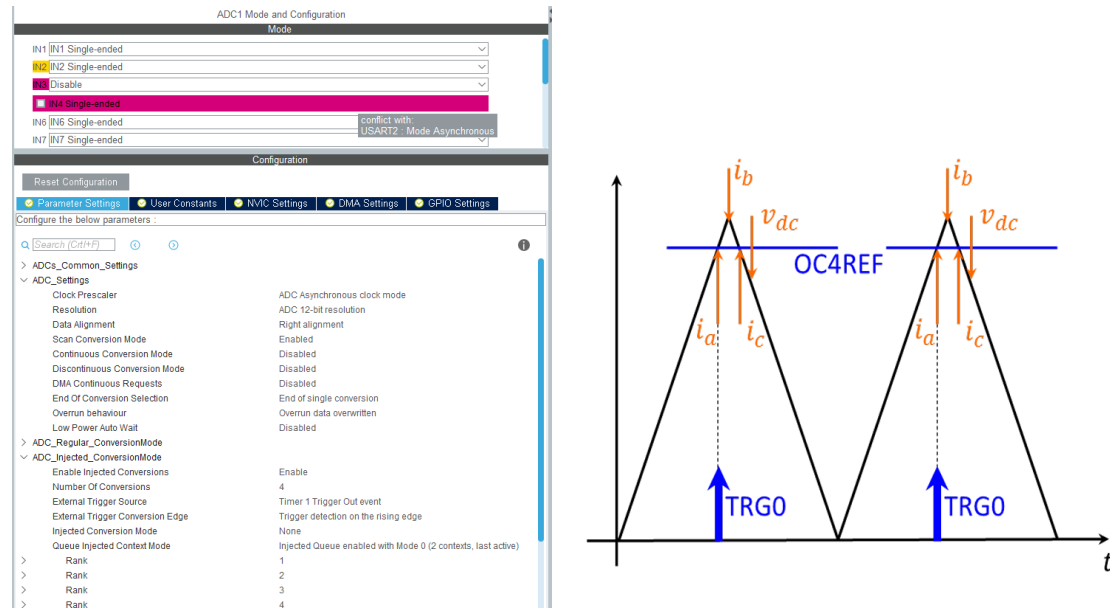


Figure 2.6: ADC1 Configuration

2.1.4 TIM2 Configuration

Timer TIM2 is a general-purpose timer that, unlike TIM1, does not produce complementary PWM signals to control the inverter legs. This type of timer is typically used to count the position pulses from an incremental encoder.

Figure 2.7 shows the configuration of TIM2.

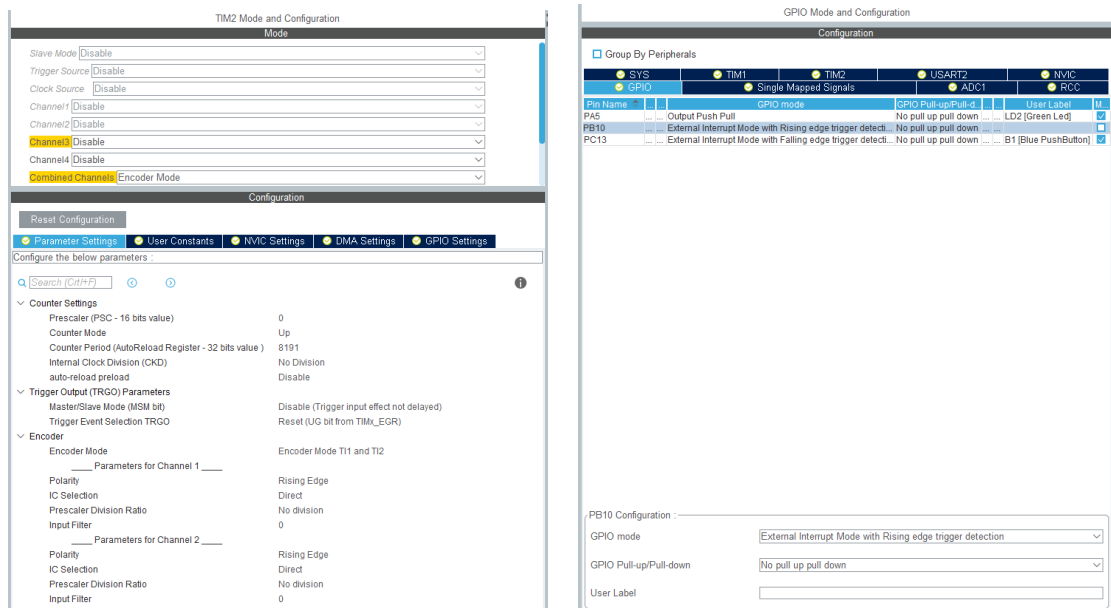


Figure 2.7: TIM2 Configuration

When operating in **Encoder Mode**, channels 1 and 2 receive the quadrature signals A and B from the encoder. The encoder features a 13-bit resolution, providing 2048 mechanical divisions per revolution. Due to the 90° phase shift between signals A and B, this results in 8192 pulses per revolution. Both channels are configured to trigger on rising edges, allowing the counter to increment on each rising transition of signals A and B. The encoder's index signal (Ri), which marks the mechanical zero position, is connected to pin PB10. This pin is set up in EXTI (external interrupt) mode to generate an interrupt on a rising edge. When the interrupt is triggered, the TIM2 counter is reset to zero.

2.1.5 TIM7 Configuration

Timer TIM7 is like timer TIM2 a general purpose timer that is used to count the pulse position in order to giving a precise time count for the CANopen network. TIM7 is related to bus APB1 and it has a 72 MHz clock.

TIM7 Mode and Configuration

Mode

☒ Activated

☐ One Pulse Mode

Configuration

Reset Configuration

☒ Parameter Settings
 ☒ User Constants
 ☒ NVIC Settings
 ☒ DMA Settings

Configure the below parameters :

Search (Ctrl+F)

Counter Settings

Prescaler (PSC - 16 bits value)	71
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value)	1000
auto-reload preload	Enable

Trigger Output (TRGO) Parameters

Trigger Event Selection	Reset (UG bit from TIMx_EGR)
-------------------------	------------------------------

Figure 2.8: TIM7 Configuration

2.1.6 NVIC Configuration

The system handles multiple interrupts, including the TIM1 update event, TIM1 break event, external interrupt (EXTI), CAN_{TX} interrupt, CAN_{RX} interrupt, and the TIM7 interrupt. The Nested Vectored Interrupt Controller (NVIC) is configured to manage these interrupts based on their predefined priority levels, as shown in Figure 2.9. In the corresponding section of the CubeMX-generated code, it is possible to configure interrupt behavior by adjusting both priority and subpriority values.

NVIC			
Code generation			
Memory management fault	<input checked="" type="checkbox"/>	0	0
Pre-fetch fault, memory access fault	<input checked="" type="checkbox"/>	0	0
Undefined instruction or illegal state	<input checked="" type="checkbox"/>	0	0
System service call via SWI instruction	<input checked="" type="checkbox"/>	0	0
Debug monitor	<input checked="" type="checkbox"/>	0	0
Pendable request for system service	<input checked="" type="checkbox"/>	0	0
Time base: System tick timer	<input checked="" type="checkbox"/>	15	0
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0	0
Flash global interrupt	<input type="checkbox"/>	0	0
RCC global interrupt	<input type="checkbox"/>	0	0
DMA1 channel6 global interrupt	<input checked="" type="checkbox"/>	0	0
DMA1 channel7 global interrupt	<input checked="" type="checkbox"/>	0	0
ADC1 and ADC2 interrupts	<input type="checkbox"/>	0	0
USB high priority or CAN_TX interrupts	<input checked="" type="checkbox"/>	0	0
USB low priority or CAN_RX0 interrupts	<input checked="" type="checkbox"/>	0	0
CAN_RX1 interrupt	<input type="checkbox"/>	0	0
CAN_SCE interrupt	<input type="checkbox"/>	0	0
TIM1 break and TIM15 interrupts	<input checked="" type="checkbox"/>	0	0
TIM1 update and TIM16 interrupts	<input checked="" type="checkbox"/>	1	0
TIM1 trigger, commutation and TIM17 interrupts	<input type="checkbox"/>	0	0
TIM1 capture compare interrupt	<input type="checkbox"/>	0	0
TIM2 global interrupt	<input type="checkbox"/>	0	0
USART2 global interrupt / USART2 wake-up interrupt through EXTI line 26	<input type="checkbox"/>	0	0
EXTI line[15:10] interrupts	<input checked="" type="checkbox"/>	0	0
TIM7 global interrupt	<input checked="" type="checkbox"/>	2	0
Floating point unit interrupt	<input type="checkbox"/>	0	0

☐ Enabled
 Preemption Priority
 Sub Priority

Figure 2.9: NVIC Configuration

The external interrupt is linked to the encoder's Ri channel, ensuring that the TIM2 counter is accurately reset to zero when the encoder reaches its mechanical zero position. As a result, this interrupt is assigned a higher priority than the TIM1 update event interrupt. In the event that the external interrupt is triggered while the main interrupt service routine (ISR) is being executed, the main ISR is preempted, and the external ISR takes precedence. The break event interrupt, which is triggered upon overcurrent detection, is assigned the highest priority among all interrupts. This configuration guarantees that, in the event of an overcurrent, the converter is immediately shut down to prevent hardware damage.

2.1.7 IWDG Configuration

The Independent Watchdog Timer (IWDG) is a down-counter that automatically resets the microcontroller if the execution time exceeds a predefined threshold, determined by the IWDG reload value. Under normal operating conditions, the watchdog is periodically refreshed by the software at the beginning of each interrupt service routine (ISR), as illustrated in Figure 2.10.

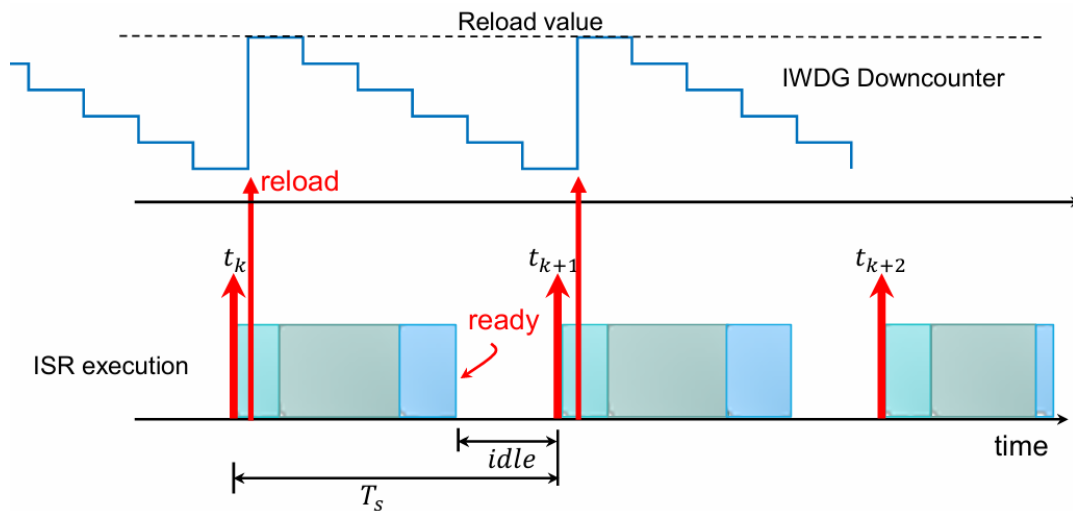


Figure 2.10: IWDG

For safety purposes, the IWDG clock is internally generated and operates independently of the MCU's main system clock. As illustrated in Figure 2.4, it runs at a fixed frequency of 40 kHz . As shown in Figure 2.11, the prescaler is configured to 4, and the reload value of the down-counter is set to 50. This configuration results in a timeout period of $300\text{ }\mu\text{s}$, after which the microcontroller is automatically reset if the watchdog is not refreshed in time.

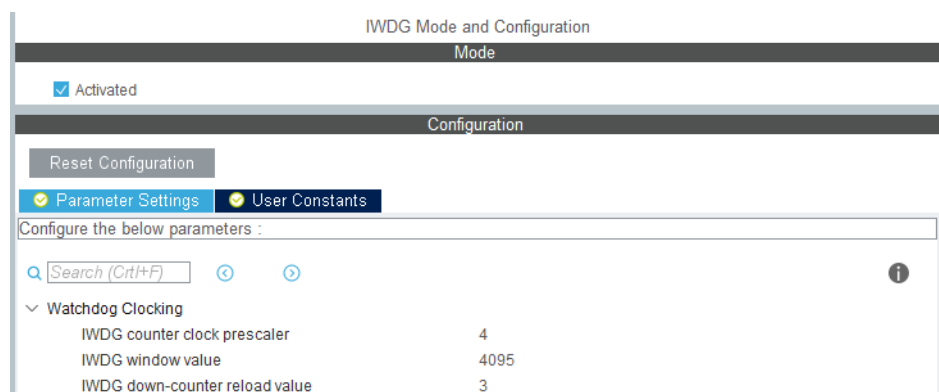


Figure 2.11: IWDG Configuration

2.1.8 USART Interface

To monitor global variables within the MCU in a manner similar to using an oscilloscope, a MATLAB-based application called Pandora Scope was employed. This tool was designed and developed by Fausto Stella and Enrico Vico, researchers at the Politecnico di Torino. Figure 2.12 displays the application interface along with a typical real-time data acquisition session performed during motor testing. On the left side of the interface, a list of relevant variables from the control code is presented, enabling the user to select which variables to visualize and how many to plot concurrently. Furthermore, the application supports the creation of multiple figures or the overlay of different data sets within a single figure for comparative analysis.

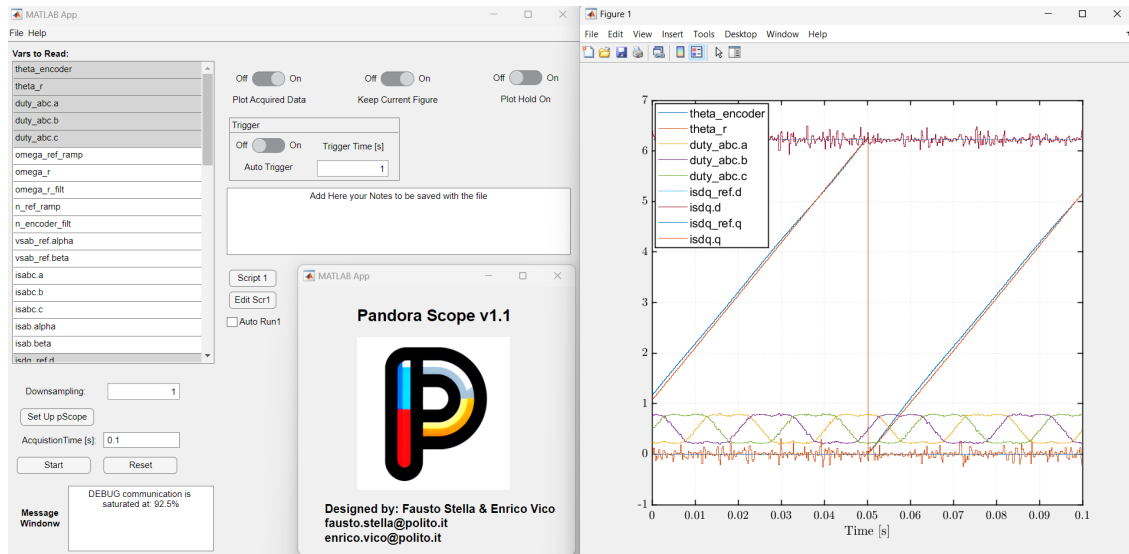


Figure 2.12: Pandora Scope

During each TIM1 update interrupt service routine (ISR), data is transmitted via the USART interface. The microcontroller sends the signals to the ST-LINK/V2 debugger over the USART communication line, which then relays the information to the host computer through a USB interface emulating a COM port.

The application's downsampling functionality enables the transmission of a large number of variables, which was crucial for generating the motor control plots presented in the subsequent chapters.

Moreover, the application supports both the recording of several seconds of signal data for offline post-processing in MATLAB and real-time visualization, functioning as a virtual oscilloscope with an integrated trigger mechanism.

2.2 Project File Organization

The project is composed of multiple source files, some of which are entirely user-defined, including:

- Constants file, where motor parameters and key constant values are defined for use within the control code. This strategy enhances computational efficiency by avoiding runtime operations such as divisions;
- Variables file, which contains the definition and initialization of all global variables required for system operation;
- Data types file, which defines custom data structures used to represent three-phase quantities and other non-standard variable types;
- Control functions file, where all functions required for implementing the control algorithm are defined. This modular approach improves code readability, maintainability, and execution efficiency;
- Additional optimization files, which include user-defined functions and callbacks for CANopen communication. While the core CANopen functionality is implemented using the CANopenNode library, these files support the integration and customization of its features.

The STM32CubeMX-generated code is further developed and refined within STM32CubeIDE, an integrated development environment that provides a complete workflow for writing, compiling, and debugging firmware. Once the user code is finalized, it can be compiled (via the Build action) and programmed into the MCU memory using the built-in programmer. STM32CubeIDE also offers an advanced debugging interface, enabling real-time monitoring of register states, variable values, and code execution flow through breakpoints, watch windows, and live expressions—facilitating efficient debugging and in-depth system analysis.

The two primary files generated by STM32CubeMX that require user modifications are:

- *main.c*: This file contains the initialization and configuration code for the MCU peripherals, along with the infinite *while(1)* loop that continuously executes while waiting for interrupts. In this project, the main loop is responsible for initializing, managing, and terminating the CANopen network communication between the STM32F303RE and the Mirco Digital One device. After each interrupt is serviced, control returns to this loop, which then awaits the next event;
- *stm32f3xx_it.c*: This file handles the interrupt service routines (ISRs) and includes the user-defined control logic. To meet real-time constraints, each ISR must complete its execution within 200 μs , ensuring the system is ready to handle subsequent interrupts without delay.

2.3 Control Code

Figure 2.13 shows the user code integrated within the *main.c* file. In this file, the CANopen network is initialized (a more detailed discussion on this topic is provided in the following chapter), while the peripherals configured through STM32CubeMX are explicitly initialized and enabled by directly manipulating specific bits in their corresponding registers using C bitwise operations. This low-level register access is necessary because certain peripheral configurations cannot be fully accomplished within the STM32CubeMX graphical interface.

```
#ifndef ENABLE_CANOPEN
    CANopen_Node_Setup(MX_CAN_Init);
    //init_sync(canopenNodeSTM32);
#endif
pScope_init(); // initialization of Pandora scope

//////////ADC1//////////
ADC1->CR|=((1<<0)); // enable ADC1 (ADEN) (ADEN=0)
ADC1->CR|=(1<<3); // AD conversion star (JADSTART) (ADEN=0)

//////////TIMER1//////////
//CHANNEL 1
TIM1->CCER|=1; //capture/compare output1 enable CC1E=1
TIM1->CCER|=(1<<2); //capture/compare complementary output1 enable CC1NE=1

//CHANNEL 2
TIM1->CCER|=(1<<4); //capture/compare output2 enable CC2E=1
TIM1->CCER|=(1<<6); //capture/compare complementary output2 enable CC2NE=1

//CHANNEL 3
TIM1->CCER|=(1<<8); //capture/compare output3 enable CC3E=1
TIM1->CCER|=(1<<10); //capture/compare complementary output3 enable CC3NE=1

//BREAK AND DEAD TIME REGISTER
TIM1->BDTR|=(1<<15); //main output enable MOE=1
TIM1->SR=0; //clean STATUS register

TIM1->CR1|=1; // enable TIM1
TIM1->DIER|=1<<7; // break interrupt enable

TIM1->RCR=1; // repetition counter
TIM1->DIER|=1; // update interrupt enable after timer start

//////////TIMER2//////////
TIM2->CR1|=1; // enable TIM2 (ENCODER)

//////////TIMER3//////////
TIM6->CR1|=1; //counter enable enable CEN = 1
TIM6->DIER|=1; //update interrupt enable UIE =1

//////////Independent Watchdog//////////
IWDG->KR = 0xAAAA; // reload watchdog TIMER (IDWG RLR)
```

Figure 2.13: File *main.c*

The *stm32f3xx_it.c* file contains the *TIM1_UP_TIM16_IRQHandler()* function, which serves as the interrupt service routine (ISR) triggered by TIM1 update events occurring once per PWM period. The user-defined code within this function implements the motor control algorithm, performing all necessary real-time computations to regulate the motor operation. This file is organized to support both the embedded control system and the Simulink motor control model. It includes conditional code sections that can be enabled or disabled depending on whether the control algorithm is executed in real-time on the motor or is being tested within the Simulink environment.

```
void TIM1_BRK_TIM15_IRQHandler(void)
{
    /* USER CODE BEGIN TIM1_BRK_TIM15_IRQn 0 */
    State=0;
    pwm_stop=1;
    TIM1->SR&=~(1<<7); //clean break interrupt flag
    /* USER CODE END TIM1_BRK_TIM15_IRQn 0 */
    HAL_TIM_IRQHandler(&htim1);
    /* USER CODE BEGIN TIM1_BRK_TIM15_IRQn 1 */

    /* USER CODE END TIM1_BRK_TIM15_IRQn 1 */
}

/**
 * @brief This function handles TIM1 update and TIM16 interrupts.
 */
void TIM1_UP_TIM16_IRQHandler(void)
{
    /* USER CODE BEGIN TIM1_UP_TIM16_IRQn 0 */
#ifdef SIM
    /*
        isabc.a      =U(0);
        isabc.b      =U(1);
        isabc.c      =U(2);
        vdc          =U(3);

        theta_r      = U(4);          // encoder
        n_ref_in     = U(5);          // rpm
        // T_ref      = U(5);          // Torque reference (direct)

        Reset = U(7);                // Black button
        Go     = U(8);                // Blue button

        // Position and SinCos from encoder [elt rad]
        theta_enc_compute(&theta_r, PP, &SinCos_r, &SinCos_r_elt);

        // rotor speed [elt rad/s]
        speed_compute_sc(SinCos_r, &SinCos_r_old, &omega_r);

        // Filter speed signal
        omega_r_filt = Filter(omega_r, omega_r_filt, TWOPI*50*Ts); // 50 Hz
    */
#else

```

Figure 2.14: Interrupt Handlers and Simulink Integration

Figure 2.14 illustrates the function *TIM1_{BRK}TIM15_{IRQ}Handler()*, which manages break events triggered by overcurrent detection. Upon detecting an overcurrent condition, this function immediately transitions the system into an ERROR state and disables the PWM modulation, thereby protecting both the power electronics and the motor from potential damage. Additionally, the figure highlights a dedicated code section for Simulink-based control, where input signals—such as measured phase currents, DC-link voltage, reference speed, and the selected control technique—are received directly from the Simulink model.

Figure 2.15 illustrates the initial operations executed when the ISR is called.

```

/* USER CODE BEGIN TIM1_UP_TIM16_IRQn 0 */
GPIOC->ODR^=(1<<3);

pScope_task();

IWDG->KR = 0xAAAA; // reload watchdog TIMER (IWDG RLR)
thetaEnc = ((float) TIM2 ->CNT)*ENC_FAC;
theta_r = thetaEnc - enc_offset;

// Encoder mechanical angle saturation
if(theta_r<0)          theta_r+=TWOPI;
if(theta_r>=TWOPI)     theta_r-=TWOPI;

// SinCos from encoder
SinCos_r.sin = sinf(theta_r); // mech
SinCos_r.cos = cosf(theta_r);

SinCos_r_elt.sin = sinf(PP*theta_r); // elct
SinCos_r_elt.cos = cosf(PP*theta_r);

// rotor speed [rad/s]
speed_compute_sc(SinCos_r, &SinCos_r_old, &omega_r);

//Filter for the speed - low pass filter
Filter(omega_r, omega_r_filt, (TWOPI*100.0f*Ts));

omega_r_rpm_filt=omega_r_filt*(1.0f/TWOPI*60);

while(((ADC1->ISR)&(1<<6))==0){} //wait end of ADC conversions
ADC1->ISR|=1<<6; //clear flag JEOP (the flag JEOP is cleared by setting the correspondent bit to 1)

// feedback
input.ch0=(float)ADC1->JDR1;
input.ch1=(float)ADC1->JDR2;
input.ch2=(float)ADC1->JDR3;

isabc.a=-(input.ch0-offset_current_a)*scala_current; // current phase a
isabc.b=-(input.ch1-offset_current_b)*scala_current; // current phase b
isabc.c=-(input.ch2-offset_current_c)*scala_current; // current phase c
vdc = ((int)ADC1->JDR4)*scala_voltage; // vdc-link voltage

// Read Button State
if(((GPIOC->IDR)&(1<<13))==0 && Go_flag==1){
    Go = 1.0f; // Go Button
    Go_flag=0;
}
else Go = 0.0f;
if ((GPIOC->IDR)&(1<<13)) Go_flag=1;
#endif

//Over current protection
CurrentProtection(isabc,&State,&pwm_stop);

switch(State){

```

Figure 2.15: Angle & Speed Computation, Feedback Acquisition

First, the watchdog timer is reloaded to prevent the microcontroller from resetting due to a timeout condition. Subsequently, the rotor position is calculated in radians based on the pulse count obtained from the TIM2 counter. This angle is then corrected as described in Section 2.3.3. The rotor speed is derived from the position data and filtered to minimize measurement noise. Next, the system waits for the completion of the feedback signal acquisition, which started $0.25 \mu s$ prior to the ISR trigger. This ensures that the ADC conversions of the phase currents and DC-link voltage are finalized before these values are utilized within the control algorithm, enabling precise closed-loop control and accurate motor operation.

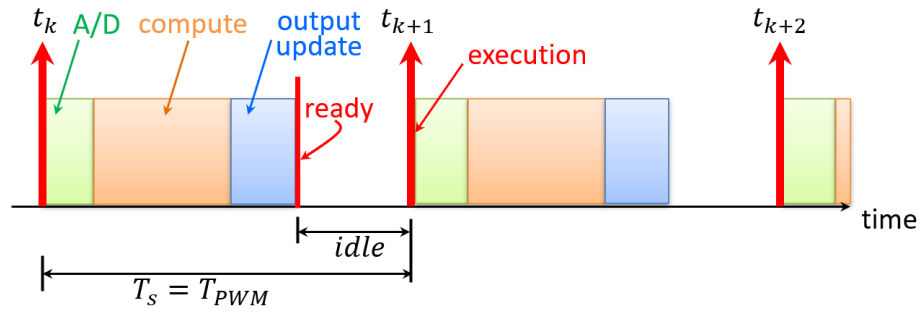


Figure 2.16: PWM Task Organization

The ISR is then organized into five operating states: ERROR, WAKE-UP, COMMISSIONING, READY and START.

2.3.1 ERROR

The ERROR state is the default condition when the MCU is turned ON or upon Reset. In this state, PWM modulation is disabled to ensure a safe operating environment. It also serves as the initialization stage for system variables, as depicted in Figure 2.17. Furthermore, the system transitions to the ERROR state if a hardware fault is detected, such as an overcurrent condition.

```
switch(State){
case ERROR:

    // Variables Init
    pwm_stop = 1;
    counter = 0;

    offset_current_a=2120;
    offset_current_b=2120;
    offset_current_c=2120;

    offset_in.ch0=0.0f;
    offset_in.ch1=0.0f;
    offset_in.ch2=0.0f;

    offset_in.ch3=0.0f;

    duty_abc.a=0.0f;
    duty_abc.b=0.0f;
    duty_abc.c=0.0f;

    n_ref_in = 0.0f;
    omega_ref_in=0.0f;
    omega_ref_ramp=0.0f;
    SinCos_ref.sin=0.0f;
    SinCos_ref.cos=1.0f;

    // Control parameters
    Ctrl_type=3; // 0 -> I/Hz, 1 -> Saliency evaluation, 2 -> FOC (speed), 3 -> FOC (torque)

    n_ref_in=800; //rpm
    accel = 1000; // rpm/s

    pos_kp= TWOPI*2.0f; // 2 Hz position Loop

    sp_par.kp= TWOPI*40.0f*T; //40 Hz
    sp_par.ki= (sp_par.kp*TWOPI*40.0f/20.0f)*Ts; // zero 1/20 wb

    isdq_ref.d=6.6f; // for 1-Hz control

    id_par.kp=0.3330f; // 200 Hz bandwidth
    id_par.ki=314*Ts;

    iq_par.kp=0.3330f;
    iq_par.ki=314*Ts;

    if(Go) State = WAKE_UP; // Wait for Go state
    break;
}
```

Figure 2.17: ERROR State

When the blue user button on the Nucleo board is pressed, the system transitions to the temporary WAKE-UP state.

2.3.2 WAKE-UP

As shown in Figure 2.18, the WAKE-UP state is dedicated to preliminary operations:

- PWM activation: PWM modulation is enabled, and all three inverter legs are set to a duty cycle of 0.2;
- Bootstrap capacitors pre-load: with all duty cycles set to 0.2, the low-side MOSFETs are active for 80% of each PWM period. This configuration ensures that the gate drivers' bootstrap capacitors fully charge over 500 cycles (approximately 0.1 seconds), as they charge during the low-side conduction period and discharge during the high-side conduction period;
- ADC offset computation: ADC1 channels capture phase current readings, which should be near zero under these conditions, to compute the offsets for each channel.

After 500 cycles, the preliminary operations are complete, and the system automatically transitions to the READY state.

```
case WAKE_UP:

    duty_abc.a=0.5f;
    duty_abc.b=0.5f;
    duty_abc.c=0.5f;

    pwm_stop = 0;

    // Current offset accumulation
    if (counter>100){
        offset_in.ch0+=input.ch0;
        offset_in.ch1+=input.ch1;
        offset_in.ch2+=input.ch2;
    }

    // Offset computation and bootstrap
    if (counter==(100+200)) {
        offset_current_a=(float)(offset_in.ch0/200.0f);
        offset_current_b=(float)(offset_in.ch1/200.0f);
        offset_current_c=(float)(offset_in.ch2/200.0f);
    }

    // Switch to READY state
    if (counter > 1000) {
        counter=0;
        State=COMMISSIONING;
    }

    counter++;

    break;
```

Figure 2.18: WAKE_UP State

2.3.3 COMMISSIONING

In general, the COMMISSIONING state is typically used in Field-Oriented Control (FOC) of synchronous motors to align the control axes (dq -axes) before starting the control loop.

For asynchronous motors controlled using FOC techniques, this state is not strictly required because the dq -axes are determined by the rotor flux.

```
case COMMISSIONING:

    ramp(TWOPI*4, 10*Ts, &theta_ref );

    if(theta_ref==TWOPI*4)
        counter++;

    if (counter==5000){
        enc_offset=theta_r;
    }

    if (counter==6000){
        State=READY;
        counter=0;
    }

    SinCos_ref.sin=sinf(theta_ref);
    SinCos_ref.cos=cosf(theta_ref);

    //Clarke transformation (a,b,c)--> (alpha,beta)
    _clarke(isabc, isab);

    isdq_ref.q = 0.;
    _rot(isab, SinCos_ref, isdq);

    //d-axis current control loop
    id_var.ref=isdq_ref.d;
    id_var.fbk=isdq.d;
    id_par.lim=SQRT10VER3*vdc;
    PIReg(&id_par, &id_var);
    vsdq_ref.d=id_var.out;

    //q-axis control loop
    iq_var.ref=isdq_ref.q;
    iq_var.fbk=isdq.q;
    iq_par.lim=sqrtf(id_par.lim*id_par.lim-id_var.out*id_var.out);
    PIReg(&iq_par, &iq_var);
    vsdq_ref.q=iq_var.out;

    _invrot(vsdq_ref, SinCos_ref, vsab_ref);
    break;
```

Figure 2.19: COMMISSIONING

2.3.4 READY

When in the READY state, the duty cycles for all three legs are set to 0.5. Once the blue user button on the Nucleo board is pressed, the system transitions to the START state.

```
case READY:
    duty_abc.a=0.5f;
    duty_abc.b=0.5f;
    duty_abc.c=0.5f;
    counter = 0;
    if (Go) State=START;

    break;
```

Figure 2.20: READY State

2.3.5 START

After the correct execution of the previous State in the START the motor is getting in motion by a specif control tht can be chose by the variable Ctrl_type. In the START are present 4 diffrent type of motor control but the only one relevant for this application is the fourth.

```

case 3: // FOC -> torque

    if (counter < 10000){
        Tref = 0.4f;
        counter++;
    }
    else
        Tref = 0.6f;

    // Id, Iq reference
    isdq_ref.d = 0.0f;
    isdq_ref.q = Tref * INV_KT; // da Nm a Ampere

    // current loop
    _rot(isab, SinCos_r_elt, isdq);

    // feed-forward
    omega_r_filt_elet = PP * omega_r_filt;
    vsdq_ref_ffw.d = RS * isdq_ref.d - omega_r_filt_elet * LS * isdq_ref.q;
    vsdq_ref_ffw.q = RS * isdq_ref.q + omega_r_filt_elet * (LS * isdq_ref.d + LAMBDA_M);

    //d-axis current control loop
    id_var.ref = isdq_ref.d;
    id_var.fbk = isdq.d;
    id_par.lim = SQRT10VER3 * vdc - vsdq_ref_ffw.d;
    PIReg(&id_par, &id_var);
    vsdq_ref.d = id_var.out;

    //q-axis control loop
    iq_var.ref = isdq_ref.q;
    iq_var.fbk = isdq.q;
    iq_par.lim = sqrtf(id_par.lim * id_par.lim - id_var.out * id_var.out) - vsdq_ref_ffw.q;
    PIReg(&iq_par, &iq_var);
    vsdq_ref.q = iq_var.out;

    // feedforward sum
    vsdq_ref.d += vsdq_ref_ffw.d;
    vsdq_ref.q += vsdq_ref_ffw.q;

    // invert transformation
    _invrot(vsdq_ref, SinCos_r_elt, vsab_ref);

    break;
}

```

Figure 2.21: START State

As reported in Figure 2.21 this FOC control impose a constant torque that has a step variation after the counter variable reach 10000. Then the FOC control is implemeted, the theory behind the implementation will be discussed in the next Chapter.

Chapter 3

CANopen and Control strategy implementations

3.1 Introduction to CANopen

CANopen is a robust communication protocol and device profile specification primarily designed for embedded systems used in automation environments. Situated within the OSI model, CANopen spans layers from the network layer upward, providing structured communication, network management, and device interaction.

This protocol is based on the Controller Area Network (CAN) standard, although implementations over alternative transport layers, such as Ethernet Powerlink and EtherCAT, are also supported. The protocol architecture encompasses various components, including addressing schemes, lightweight communication protocols, and application-level specifications defined by device profiles.

The foundation of the CANopen standard lies in the CiA 301 specification, maintained by CAN in Automation (CiA), which defines core device and communication profiles. Specialized device functionalities are addressed through additional specifications such as CiA 401 for I/O modules and CiA 402 for motion control systems.

A key feature of each CANopen-compliant device is the implementation of a standardized communication interface. Devices operate within a well-defined state machine that includes Initialization, Pre-operational, Operational, and Stopped states, transitioning based on Network Management (NMT) commands. At the heart of the device's configuration and data structure lies the Object Dictionary, a hierarchical collection of indexed variables used for parameter configuration and real-time data exchange.

CANopen facilitates various communication paradigms including master/slave, client/server, and producer/consumer models, each tailored to different operational contexts. Communication protocols such as SDO (Service Data Object), PDO (Process Data Object), and EMCY (Emergency Messages) ensure flexible, timely, and prioritized data exchanges. Advanced synchronization is achieved through SYNC and TIME protocols, allowing deterministic behavior in time-critical automation applications.

The protocol's adaptability and comprehensive support for network diagnostics, configuration, and data transport make CANopen a widely adopted standard in industrial automation systems, particularly where reliability, interoperability, and real-time performance are crucial.

CANopen is a robust communication protocol and device profile specification primarily designed for embedded systems used in automation environments. Situated within the OSI model, CANopen spans layers from the network layer upward, providing structured communication, network management, and device interaction.

This protocol is based on the Controller Area Network (CAN) standard, although implementations over alternative transport layers, such as Ethernet Powerlink and EtherCAT, are also supported. The protocol architecture encompasses various components, including addressing schemes, lightweight communication protocols, and application-level specifications defined by device profiles.

The foundation of the CANopen standard lies in the CiA 301 specification, maintained by CAN in Automation (CiA), which defines core device and communication profiles. Specialized device functionalities are addressed through additional specifications such as CiA 401 for I/O modules and CiA 402 for motion control systems.

A key feature of each CANopen-compliant device is the implementation of a standardized communication interface. Devices operate within a well-defined state machine that includes Initialization, Pre-operational, Operational, and Stopped states, transitioning based on Network Management (NMT) commands. At the heart of the device's configuration and data structure lies the Object Dictionary, a hierarchical collection of indexed variables used for parameter configuration and real-time data exchange.

CANopen facilitates various communication paradigms including master/slave, client/server, and producer/consumer models, each tailored to different operational contexts. Communication protocols such as SDO (Service Data Object), PDO (Process Data Object), and EMCY (Emergency Messages) ensure flexible, timely, and prioritized data exchanges. Advanced synchronization is achieved through SYNC and TIME protocols, allowing deterministic behavior in time-critical automation applications.

The protocol's adaptability and comprehensive support for network diagnostics, configuration, and data transport make CANopen a widely adopted standard in industrial automation systems, particularly where reliability, interoperability, and real-time performance are crucial. CANopen is a robust communication protocol and device profile specification primarily designed for embedded systems used in automation environments. Situated within the OSI model, CANopen spans layers from the network layer upward, providing structured communication, network management, and device interaction.

This protocol is based on the Controller Area Network (CAN) standard, although implementations over alternative transport layers, such as Ethernet Powerlink and EtherCAT, are also supported. The protocol architecture encompasses various components, including addressing schemes, lightweight communication protocols, and application-level specifications defined by device profiles.

The foundation of the CANopen standard lies in the CiA 301 specification, maintained

by CAN in Automation (CiA), which defines core device and communication profiles. Specialized device functionalities are addressed through additional specifications such as CiA 401 for I/O modules and CiA 402 for motion control systems.

A key feature of each CANopen-compliant device is the implementation of a standardized communication interface. Devices operate within a well-defined state machine that includes Initialization, Pre-operational, Operational, and Stopped states, transitioning based on Network Management (NMT) commands. At the heart of the device's configuration and data structure lies the Object Dictionary, a hierarchical collection of indexed variables used for parameter configuration and real-time data exchange.

CANopen facilitates various communication paradigms including master/slave, client/server, and producer/consumer models, each tailored to different operational contexts. Communication protocols such as SDO (Service Data Object), PDO (Process Data Object), and EMCY (Emergency Messages) ensure flexible, timely, and prioritized data exchanges. Advanced synchronization is achieved through SYNC and TIME protocols, allowing deterministic behavior in time-critical automation applications.

The protocol's adaptability and comprehensive support for network diagnostics, configuration, and data transport make CANopen a widely adopted standard in industrial automation systems, particularly where reliability, interoperability, and real-time performance are crucial.

3.1.1 CANopen - higher layer protocol

It is essential to understand that CANopen functions as a higher-layer protocol built on top of the CAN bus standard. In this context, the CAN bus (as defined by ISO 11898) acts as the transport medium while CANopen messages are carried through the network. CANopen can also be interpreted within the framework of the 7-layer OSI model, occupying the upper layers of the stack.

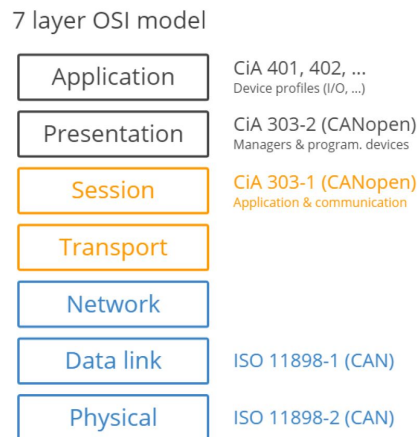


Figure 3.1: 7-layer OSI model

CANopen in OSI model context

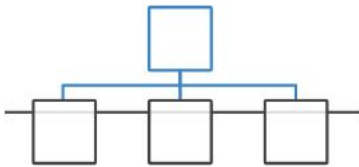
The OSI model is a conceptual framework that standardizes communication functions across heterogeneous communication systems. The lower layers are responsible for fundamental transmission tasks, such as the handling of raw bit streams, while the upper layers manage more abstract functions, including message segmentation and communication services like initiation, indication, response, and confirmation.

Within this framework, the CAN bus corresponds to the two lowest OSI layers—Layer 1 (Physical) and Layer 2 (Data Link). In practical terms, this means that CAN is responsible for the transmission of data frames containing an 11-bit CAN identifier, a Remote Transmission Request (RTR) bit, and up to 64 data bits. These fields are essential for the operation of higher-layer protocols. Accordingly, CAN plays the same foundational role in CANopen as it does in other protocols such as J1939.

As shown above, CANopen operates at the Application layer (Layer 7) of the OSI model and is defined through a set of standardized specifications. Within this layer, CANopen introduces a range of higher-level functionalities and abstractions, which will be discussed in the following sections. It is also worth mentioning that, although CANopen is typically associated with the CAN data link layer, it can be adapted to work over other communication protocols such as EtherCAT, Modbus, or Powerlink.

3.2 Six core CANopen concepts

Although familiarity with CAN bus and protocols like J1939 may provide a useful foundation, CANopen incorporates a number of important higher-layer concepts that distinguish it from other CAN-based protocols.



Communication Models: There are 3 models for device/node communication: Master/slave, client/server and producer/consumer.

Figure 3.2: Communication Models



Communication Protocols: Protocols are used in communication, e.g. configuring nodes (SDOs) or transmitting real-time data (PDOs).

Figure 3.3: Communication Protocols



Device States: A device supports different states. A 'master' node can change the state of a 'slave' node – e.g. resetting it.

Figure 3.4: Device States



Object Dictionary: Each device has an OD with entries that specify, e.g., the device configuration. It can be accessed via SDOs.

Figure 3.5: Object Dictionary



Electronic Data Sheet: The EDS is a standard file format for OD entries – allowing, e.g., service tools to update devices.

Figure 3.6: Electronic Data Sheet



Device Profile Standards: Describe modules (CiA 401) and motion control (CiA 402) for vendor independence.

Figure 3.7: Device Profile Standards

The below illustration shows how the CANopen concepts link together - and we will detail each below:

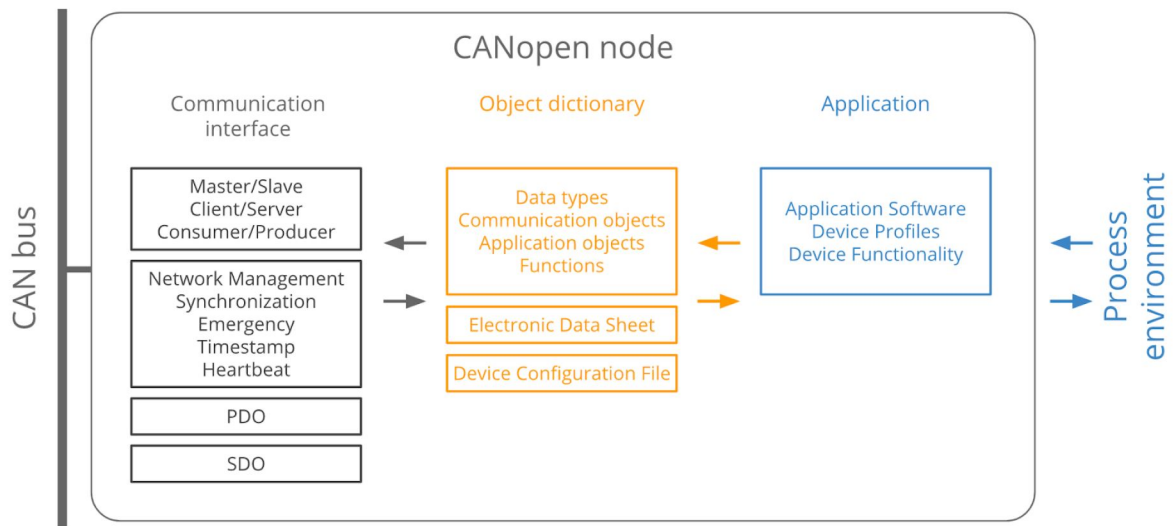


Figure 3.8: CANopen concepts link together

3.3 CANopen communication basics

Effective communication among multiple devices in a CANopen network is enabled through three distinct communication models, each designed to address specific types of data exchange and interaction patterns.

3.3.1 CANopen communication models

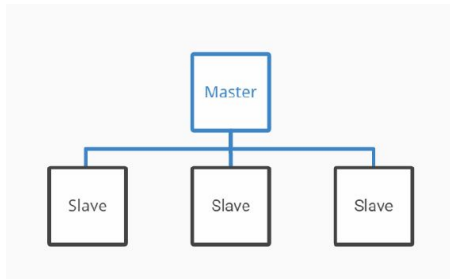


Figure 3.9: Master/Slave

Master/Slave In a typical CANopen network, one node—such as a control interface—acts as the application master or host controller. It communicates with multiple slave nodes, such as servo motors, by sending commands or requesting data. This model is commonly used for tasks such as diagnostics or device state management. In this context, a producer-consumer paradigm is employed: the producer node transmits data across the network, while the consumer node receives and processes it. The data can be transmitted either upon explicit request from the consumer (pull model) or autonomously by the producer (push model).

Standard CANopen networks can support up to 127 slave nodes. It is also important to note that multiple host controllers can coexist on the same network, sharing the same data link layer.

An example of a service operating under this model is the Network Management (NMT) protocol.

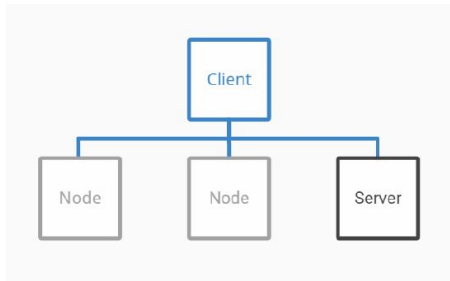


Figure 3.10: Client/Server

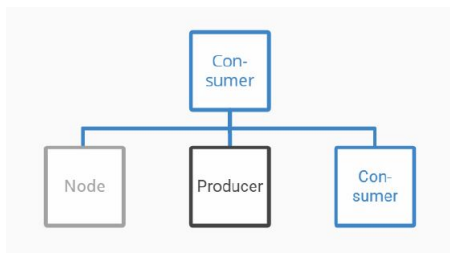


Figure 3.11: Consumer/Producer

Client/Server In this communication model, a client node initiates a request for data from a server node, which then responds with the requested information. This mechanism is typically used when an application master needs to access data from the Object Dictionary (OD) of a slave device.

From the perspective of the server, reading data is referred to as an upload, while writing data to the server is termed a download. This terminology reflects the server-centric viewpoint adopted in the CANopen specification.

An example of a service implementing this model is the Service Data Object (SDO) protocol.

Consumer/Producer In this model, a producer node periodically broadcasts data to the entire network, where it is received and interpreted by one or more consumer nodes. The data transmission can follow either a pull model, where the data is sent in response to a request, or a push model, where the producer transmits the data autonomously without an explicit request.

A typical example of a service utilizing this model is the Heartbeat protocol, which is used for node monitoring and fault detection in CANopen networks.

As illustrated, the underlying communication mechanisms in these models are largely similar. However, they are distinguished primarily for the sake of terminology consistency within the CANopen specification.

3.3.2 The CANopen frame

To understand CANopen communication, it is necessary to break down the CANopen CAN frame:

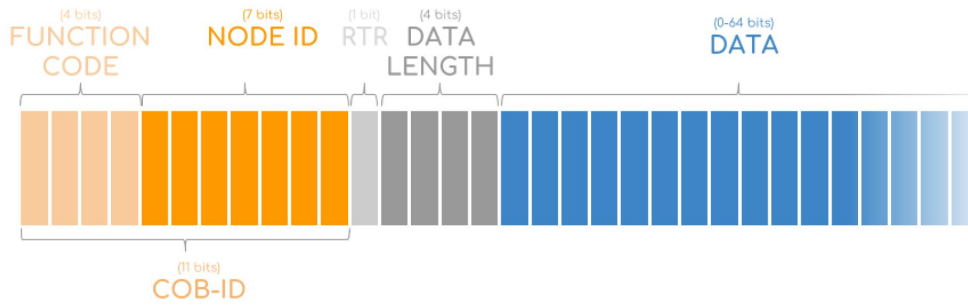


Figure 3.12: CANopen frame

The 11-bit CAN identifier used in CANopen is known as the Communication Object Identifier (COB-ID). By default, it is divided into two parts: the first 4 bits represent the function code, while the remaining 7 bits specify the node ID.

To understand how the COB-ID is structured and utilized, we refer to the pre-defined identifier allocation scheme commonly used in simple CANopen networks (see Figure 3.13). Note that COB-IDs and node IDs will be referenced in hexadecimal format in the following discussion.

As shown, specific COB-IDs (e.g., 0x381, 0x581, ...) are associated with particular communication services, such as Transmit PDO 3 or Transmit SDO. This means that each COB-ID encodes not only the type of service being used, but also the identity of the node involved in the data transmission or reception.

COMMUNICATION OBJECT	FUNCTION CODE (4 bit, bin)	NODE IDs (7 bit, bin)	COB-IDs (hex)	COB-IDs (dec)	#
1 NMT	0000	0000000	0	0	1
2 SYNC	0001	0000000	80	128	1
3 EMCY	0001	0000001-1111111	81 - FF	129 - 255	127
4 TIME	0010	0000000	100	256	1
5 Transmit PDO 1	0011	0000001-1111111	181 - 1FF	385 - 511	127
Receive PDO 1	0100	0000001-1111111	201 - 27F	513 - 639	127
Transmit PDO 2	0101	0000001-1111111	281 - 2FF	641 - 767	127
Receive PDO 2	0110	0000001-1111111	301 - 37F	769 - 895	127
Transmit PDO 3	0111	0000001-1111111	381 - 3FF	897 - 1023	127
Receive PDO 3	1000	0000001-1111111	401 - 47F	1025 - 1151	127
Transmit PDO 4	1001	0000001-1111111	481 - 4FF	1153 - 1279	127
Receive PDO 4	1010	0000001-1111111	501 - 57F	1281 - 1407	127
6 Transmit SDO	1011	0000001-1111111	581 - 5FF	1409 - 1535	127
Receive SDO	1100	0000001-1111111	601 - 67F	1537 - 1693	127
7 HEARTBEAT	1110	0000001-1111111	701 - 77F	1793 - 1919	127

Figure 3.13: CANopen COB-ID

3.3.3 CANopen communication protocols/services

Below, is provided a brief overview of the seven CANopen service types referenced earlier, including how each one utilizes the 8 data bytes available in a standard CAN frame.

- **Network Management (NMT)** The Network Management (NMT) service is responsible for controlling the operational state of CANopen devices, such as transitioning between pre-operational, operational, and stopped states, through specific NMT commands (e.g., start, stop, reset).

To initiate a state change, the NMT master transmits a 2-byte message using a CAN ID of 0 (corresponding to function code 0 and node ID 0). This message is processed by all slave nodes on the network. The first data byte specifies the requested state, while the second data byte identifies the target node ID. A node ID value of 0 indicates that the command is broadcast to all nodes.

Common NMT commands include transition to operational mode (state 0x01), stopped mode (state 0x02), pre-operational mode (state 0x80), as well as reset application (state 0x81) and reset communication (state 0x82).

- **Synchronization (SYNC)** The SYNC message is primarily used to synchronize the input sampling and output actuation of multiple CANopen devices, typically coordinated by the application master.

The application master transmits the SYNC message with a COB-ID of 0x080 to the CANopen network, optionally including a SYNC counter. Multiple slave nodes can be configured to respond to this SYNC message by either transmitting input data captured simultaneously or by applying outputs at the same precise moment, enabling coordinated synchronous operation across devices. The SYNC counter allows the configuration of multiple groups of devices operating synchronously within the network.

- **Emergency (EMCY)** The Emergency (EMCY) service is employed when a device encounters a critical error, such as a sensor failure, enabling it to notify the entire network of the fault condition.

The affected node transmits a single EMCY message—using a COB-ID corresponding to the node (for example, 0x085 for node 5)—with high priority on the CANopen network. The data bytes within the message convey detailed information about the nature of the error, which can be referenced for diagnostics and troubleshooting.

- **Timestamp (TIME) [PDO]** The TIME service enables the distribution of a global network time across CANopen devices. This service transmits date and time information using a 6-byte data field.

An application master periodically broadcasts the TIME message with a CAN ID of 0x100. The first four data bytes represent the time in milliseconds elapsed since midnight, while the final two bytes indicate the number of days passed since January 1, 1984.

- **Process Data Object [PDO]** The Process Data Object (PDO) service is used for the real-time transmission of data between devices. This includes measured values such as position feedback, as well as command data like torque requests.
- **Service Data Object [SDO]** Service Data Object (SDO) services enable access to and modification of entries within a CANopen device's Object Dictionary. This is typically employed when an application master needs to configure or update specific parameters of a CANopen device.
- **Node monitoring (Heartbeat) [SDO]** The Heartbeat service serves two primary purposes: to signal that a node is operational ("alive") and to acknowledge receipt of NMT commands.

An NMT slave device periodically transmits the Heartbeat message—typically every 100 ms—with a CAN ID corresponding to the node (for example, 0x2C1 for node 5). The first data byte of this message contains the current state of the node. The recipient(s) of the Heartbeat message, such as the NMT master and optionally other devices, monitor its reception and trigger an alert if no message is received within a predefined timeout period.

3.4 CANopen Object Dictionary

All CANopen nodes are required to maintain an Object Dictionary (OD), which is a standardized data structure encompassing all parameters that define the behavior and configuration of the node. Entries within the OD are accessed using a 16-bit index and an 8-bit subindex. For instance, the entry at index 0x1008 (subindex 0) typically holds the device name of a CANopen-compliant node.

Each Object Dictionary entry is characterized by a set of attributes, specifically defined as follows:

- Index: 16-bit base address of the object;
- Object name: Manufacturer device name;
- Object code: Array, variable, or record;
- Data type: E.g. `VISIBLE_STRING`, `UNSIGNED32` or Record Name;
- Access: `rw` (read/write), `ro` (read-only), `wo` (write-only);
- Category: Indicates if this parameter is mandatory/optional (M/O);

3.4.1 OD standardized sections

The Object Dictionary is organized into standardized sections, with certain entries defined as mandatory and others left fully customizable by the device manufacturer. Crucially, the entries within a device's Object Dictionary can be accessed and modified by other devices over the CAN network using services such as Service Data Objects (SDOs). For example, this allows an application master to configure parameters on a slave node, such as enabling or disabling data logging from a specific sensor input or adjusting the frequency at which the slave transmits heartbeat messages.

Link to Electronic Data Sheet and Device Configuration File

To better comprehend the structure and content of the Object Dictionary, it is useful to consider its 'human-readable' representations: the Electronic Data Sheet (EDS) and the Device Configuration File (DCF).



Figure 3.14: CANopen OD human-readable form

The Electronic Data Sheet (EDS)

In practice, the configuration and management of complex CANopen networks are performed using specialized software tools. To facilitate this process, the CiA 306 standard defines a human-readable and machine-friendly INI file format that serves as a "template" for a device's Object Dictionary. This Electronic Data Sheet (EDS), typically provided by the device vendor, contains detailed information about all device objects, excluding their runtime values.

Device Configuration File (DCF)

Consider a scenario where a factory acquires a new component for integration into their conveyor belt system. The operator modifies the device's Electronic Data Sheet (EDS) by adding specific parameter values and, if necessary, renaming objects described within the EDS. Through this process, the operator generates what is known as a Device Configuration File (DCF). With the DCF completed, the new device is fully prepared for seamless integration into the particular CANopen network deployed on-site.

3.5 SDO - configuring the CANopen network

The Service Data Object (SDO) service enables a CANopen node to read from or write to the Object Dictionary of another node over the CAN network. As previously described in the communication models section, SDO communication follows a client-server paradigm. Specifically, an SDO client initiates communication with a designated SDO server. This interaction can serve two main purposes: updating an Object Dictionary entry, known as an SDO download, or retrieving an entry, referred to as an SDO upload. In typical master-slave configurations, the node performing the Network Management (NMT) master role acts as the client, accessing the Object Dictionaries of all NMT slave nodes for reading or writing operations.

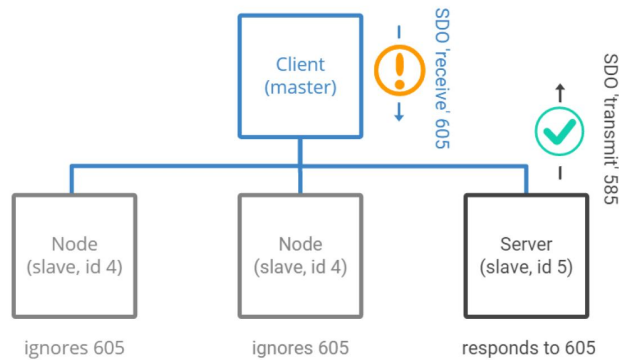


Figure 3.15: CANopen SDO communication

3.6 PDO - operating the CANopen network

The CANopen Process Data Object (PDO) service is designed for efficient real-time exchange of operational data between CANopen nodes. For example, a PDO might carry pressure readings from a pressure transducer or temperature measurements from a temperature sensor.

While the Service Data Object (SDO) service could theoretically be used for similar data transfers, it has limitations. Due to protocol overhead—such as the command byte and Object Dictionary addressing—each SDO response can only carry up to 4 data bytes. For instance, if a master node needs two parameter values from Node 5, retrieving this data via SDO would require four complete CAN frames: two request frames and two response frames.

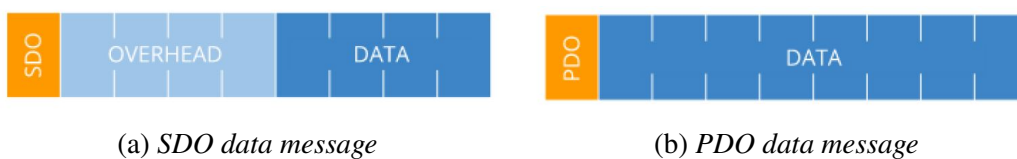


Figure 3.16: SDO and PDO message comparison

In contrast, a PDO message can carry up to 8 bytes of data within a single frame and may include multiple object parameter values simultaneously. Consequently, a data transfer that would require at least four frames using the SDO service can often be accomplished with just one PDO frame. Due to its efficiency and role in transmitting the majority of real-time data, the PDO service is widely regarded as the most essential protocol within the CANopen stack.

3.6.1 How does the CANopen PDO service work

In the context of PDO communication, the terms producer and consumer are employed. A producer node generates data and transmits it to a consumer node (typically the master) using a Transmit PDO (TPDO). Conversely, the producer can receive data from the consumer via a Receive PDO (RPDO). Producer nodes are often configured to respond to a SYNC trigger broadcast by the consumer at fixed intervals, such as every 100 ms.

For example, Node 5 may transmit the following TPDO with a COB-ID of 0x185:

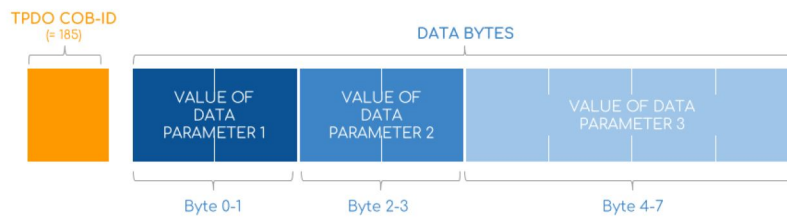


Figure 3.17: CANopen PDO message

Note that the data bytes are densely packed with three parameter values, which represent real-time data corresponding to specific Object Dictionary entries of Node 5. The nodes consuming this information must be configured to correctly interpret the contents of the PDO data bytes.

3.7 CANopen network set up in this specific application

In this specific project the CANopen network is the basic for the communication between STM32F303RE and Micro digital One for the monitoring and the set up of the test bench.

3.7.1 CANopen in STM32F303RE

Specifically the CANopen network and all the required configuration take place in the main.c file. A monolithic approach is avoided in order to have a more favorable debug session and an easier approach to coding.

CANopen initialization

The CANopen initialization is placed in a separate file called `canopen_config.c` and its header file `canopen_config.h`, they are allocated in `STM32F303RE_CANopen/Core/Src/canopen_config.c` and `STM32F303RE_CANopen/Core/Inc/canopen_config.h` respectively.

In Figure 3.18 is presented `canopen_config.c` and in Figure 3.19 the `canopen_config.h` file.

```
#include "main.h"
#include "canopen_config.h"
#include "CO_app_STM32.h"

extern CAN_HandleTypeDef hcan;
extern TIM_HandleTypeDef htim7;

void CANopen_Node_Setup(void) {
    static CANopenNodeSTM32 node;

    node.CANHandle = &hcan;
    node.HWInitFunction = MX_CAN_Init;
    node.timerHandle = &htim7;
    node.desiredNodeID = 1;
    node.baudrate = 500;

    canopenNodeSTM32 = &node; // setting global pointer
    canopen_app_init(&node);
}
```

Figure 3.18: `canopen_config.c` source file

```
#ifndef CANOPEN_CONFIG_H
#define CANOPEN_CONFIG_H

#include "CANopen.h" // where Canopen is defined CANopenNodeSTM32
#include "main.h" // hcan, htim7
#include "CO_app_STM32.h" // typedef CANopenNodeSTM32

void CANopen_Node_Setup(void);

#endif // CANOPEN_CONFIG_H
```

Figure 3.19: `canopen_config.h` source file

In the source file the function: `CANopen_Node_Setup()` is created, the aim is to recall it in the `main.c` file instead of having the initial set up directly in the `main.c` file. For the STM32F303RE the CANopen network operate on the CAN buses already present on the board, as already mentioned in Chapter 2 section 2.1.5 a specific timer TIM7 is dedicated to the correct execution of the CANopen network, moreover is necessary to specify with which node number the board will be visible on the network and set an appropriate baudrate (500 *kBaud/s* work just fine for CANopen) that must be the same for every node present on the network.

The task of the header file is to make visible to all the part of the code the source file.

CANopen SDO messages in STM32F303RE

The source file `CO_SDOclient_utils.c` and its header `CO_SDOclient_utils.h` are not part of the original CANopenNode library but are specifically created for this bench test application in order to simply manage the SDO messages, which cover a very relevant role in the initial set up of the CANopen configuration parameters for the Micro digital One and the control of the associated motor.

```

/*
 * CO_SDOclient_utils.c
 *
 * Created on: Apr 30, 2025
 * Author: Paolo Ubico
 */

#include "CO_SDOclient_utils.h"
#include "../Core/Inc/Utils.h"
#include <unistd.h> // per sleep_us o equivalente

CO_SDO_abortCode_t read_SDO(CO_SDOclient_t* SDO_C, uint8_t nodeId,
                             uint16_t index, uint8_t subIndex,
                             uint8_t* buf, size_t bufSize, size_t* readSize) {
    CO_SDO_return_t SDO_ret;

    SDO_ret = CO_SDOclient_setup(SDO_C, CO_CAN_ID_SDO_CLI + nodeId,
                                  CO_CAN_ID_SDO_SRV + nodeId, nodeId);
    if (SDO_ret != CO_SDO_RT_ok_communicationEnd) {
        return CO_SDO_AB_GENERAL;
    }

    SDO_ret = CO_SDOclientUploadInitiate(SDO_C, index, subIndex, 1000, false);
    if (SDO_ret != CO_SDO_RT_ok_communicationEnd) {
        return CO_SDO_AB_GENERAL;
    }

    do {
        uint32_t timeDifference_us = 500; //10000
        CO_SDO_abortCode_t abortCode = CO_SDO_AB_NONE;

        //canopen_app_process(); //forcing CANopen

        SDO_ret = CO_SDOclientUpload(SDO_C, timeDifference_us, false,
                                      &abortCode, NULL, NULL, NULL);
        if (SDO_ret < 0) {
            return abortCode;
        }

        HAL_Delay(timeDifference_us);
    } while (SDO_ret > 0);

    *readSize = CO_SDOclientUploadBufRead(SDO_C, buf, bufSize);
    return CO_SDO_AB_NONE;
}

CO_SDO_abortCode_t write_SDO(CO_SDOclient_t* SDO_C, uint8_t nodeId,
                              uint16_t index, uint8_t subIndex,
                              uint8_t* data, size_t dataSize) {
    CO_SDO_return_t SDO_ret;
    bool_t bufferPartial = false;

    SDO_ret = CO_SDOclient_setup(SDO_C, CO_CAN_ID_SDO_CLI + nodeId,
                                  CO_CAN_ID_SDO_SRV + nodeId, nodeId);
    if (SDO_ret != CO_SDO_RT_ok_communicationEnd) {
        return CO_SDO_AB_GENERAL;
    }

    SDO_ret = CO_SDOclientDownloadInitiate(SDO_C, index, subIndex,
                                             dataSize, 1000, false);
    if (SDO_ret != CO_SDO_RT_ok_communicationEnd) {
        return CO_SDO_AB_GENERAL;
    }

    size_t nWritten = CO_SDOclientDownloadBufWrite(SDO_C, data, dataSize);
    if (nWritten < dataSize) {
        bufferPartial = true;
    }

    do {
        uint32_t timeDifference_us = 500; // 10000
        CO_SDO_abortCode_t abortCode = CO_SDO_AB_NONE;

        //canopen_app_process(); //forcing CANopen

        SDO_ret = CO_SDOclientDownload(SDO_C, timeDifference_us, false,
                                        bufferPartial, &abortCode, NULL, NULL);
        if (SDO_ret < 0) {
            return abortCode;
        }

        HAL_Delay(timeDifference_us);
    } while (SDO_ret > 0);

    return CO_SDO_AB_NONE;
}

CO_SDO_return_t safe_write_SDO(CO_SDOclient_t* client, uint8_t nodeId,
                                uint16_t index, uint8_t subIndex,
                                uint8_t* data, uint8_t dataLen, uint8_t retries) {
    CO_SDO_return_t ret;
    uint8_t attempt = 0;

    do {
        ret = write_SDO(client, nodeId, index, subIndex, data, dataLen);
        if (ret == CO_SDO_RT_ok_communicationEnd) break;
        HAL_Delay(5); //breve attesa prima di riprovare
    } while (++attempt < retries);

    return ret;
}

```

Figure 3.20: CO_SDOclient_utils.c source file

In this source file are reported the two main SDO function, `read_SDO` and `write_SDO` already provided by `CANopenNode` library and another function is implemented here: `safe_write_SDO`. The only difference from a classic `write_SDO` is that if the the communication fails the function is not stopped but retries to send the message.

```
#ifndef CO_301_SDOCLIENT_UTILS_H_
#define CO_301_SDOCLIENT_UTILS_H_

#include "301/CO_SDOclient.h"

CO_SDO_abortCode_t read_SDO(CO_SDOclient_t* SDO_C, uint8_t nodeId,
                             uint16_t index, uint8_t subIndex,
                             uint8_t* buf, size_t bufSize, size_t* readSize);

CO_SDO_abortCode_t write_SDO(CO_SDOclient_t* SDO_C, uint8_t nodeId,
                              uint16_t index, uint8_t subIndex,
                              uint8_t* data, size_t dataSize);

CO_SDO_return_t safe_write_SDO(CO_SDOclient_t* client, uint8_t nodeId,
                                uint16_t index, uint8_t subIndex,
                                uint8_t* data, uint8_t dataLen, uint8_t retries);
#endif /* 301_CO_SDOCLIENT_UTILS_H_ */
```

Figure 3.21: `CO_SDOclient_utils.h` source file

As before the header file associated to the its source file contains the function declaration in order to be used in every other file.

CANopen network process

The CANopen network must always be processed in order to ensure the correct communication between every node. The easier solution is to place the core of the process in the `while(1)` of the nucleo board that a part from the ISR is always processed.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    #ifndef ENABLE_CANOPEN
        canopen_app_process(); // CANopen in esecuzione

        //GPIOC->ODR^=(1<<3);
        //IWDG->KR = 0xAAAA; // reload watchdog TIMER (IDWG RLR)

        /* Gestione messaggi CANopen */
        switch (Can) {
            case OFF:
                handle_off_state();
                break;
            case PAUSE:
                handle_pause_state();
                break;
            case ON:
                handle_on_state();
                break;
        }
    #endif
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
}
```

Figure 3.22: `while(1)` from `main.c`

In the `while(1)` must be present without any interruption the function `canopen_app_process()`, present in the `CANopenNode` library ensure the processing of the CANopen network. The `switch` is used to handle the different state of the motor drive by Micro digital One.

CANopen network interrupt

In order to visualize the messages in the network the `canopen_app_process()`, must be stopped when the TIM7 timer runs out with a specific function: `HAL_TIM_PeriodElapsedCallback()`.

```
#ifndef ENABLE_CANOPEN
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {

    // SYNC producer -> dedicated timer
    if (htim->Instance == TIM3) {
        HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);
        canopenNodeSTM32->canOpenStack->CANmodule[0].firstCANtxMessage = true;
        CO_SYNCsend(canopenNodeSTM32->canOpenStack->SYNC);
    }

    // System tick
    if (htim->Instance == TIM7) {
        HAL_IncTick();
    }

    // Timer CANopenNode -> cyclical interrupt
    if (htim == canopenNodeSTM32->timerHandle) {
        canopen_app_interrupt();
    }
}
#endif
```

Figure 3.23: HAL_TIM_PeriodElapsedCallback function

In this function is present the code for generate a SYNC message for the node of the network with its dedicate timer TIM3 and the function: `canopen_app_interrupt()` necessary to stop the flow of the `canopen_app_process()` in order to monitoring the network via **PCAN-View** software.

3.7.2 CANopen in Micro digital One

As already reported in the Chapter 1, the Micro digital One is capable of communicate via CANopen (as one the possible connction). The goal is to correctly sut up the Micro digital One in order to drive the **S1601B303 Brushless Servomotor** tahnks to a velocity control mode. This control must be enabled by the STM32F303RE.

Micro digital One CANopen set up

All the drive paramters present in the in the "Parameters Configuration" ofthe software Drive Watcher as well as the "Register Configuration" can be set via CANopen form another node as long as the node of the Micro digital One is a slave node. In this test bench configuration all the necessary parameters are imposed via CANopen at the start of the code. The most important parameters for Micro digital One in CANopen configuration are:

- c7: node number;
- c9: set to 5 for CANopen mode;
- d8: set the motor;
- n1: CANopen bit rate (6 for 500 *kBit/s*);

This parameters can be set via STM32F303RE or as in this case via specific software **Drive Watcher** give by Microphase to program the Micro digital One as a stand alone product.

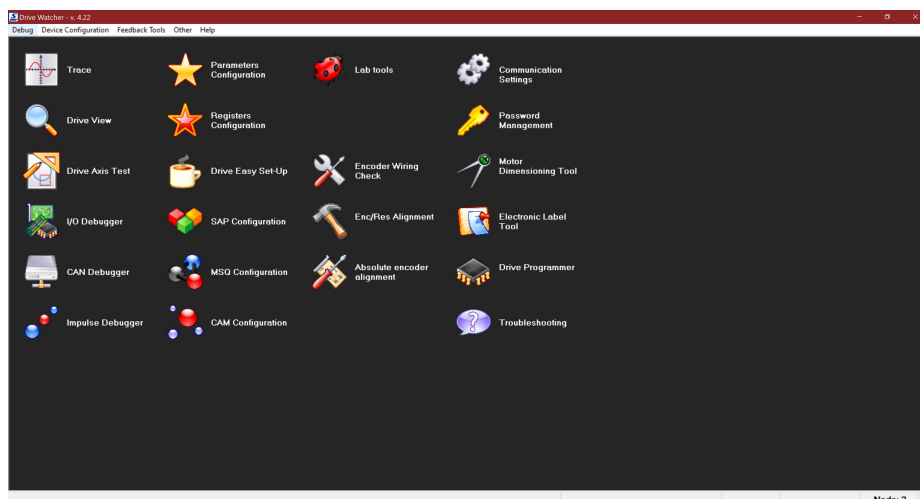
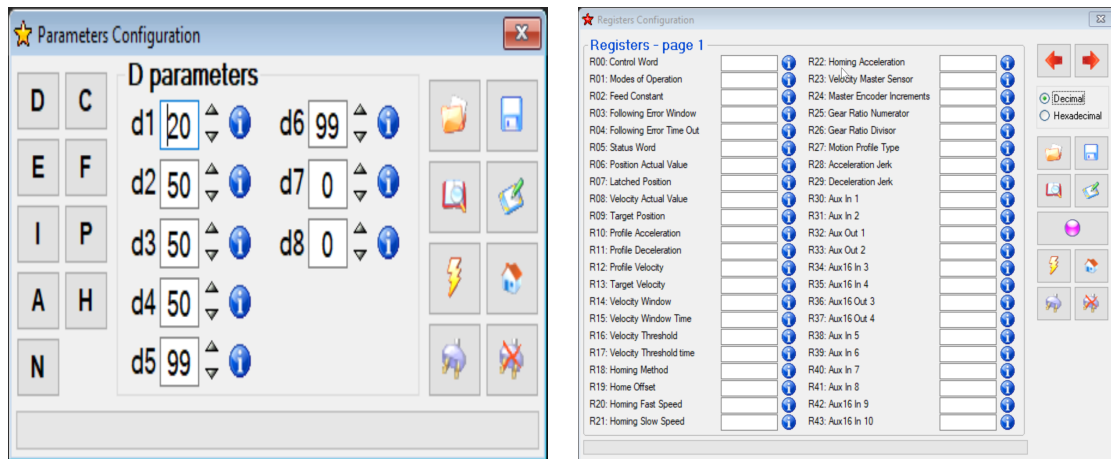


Figure 3.24: Drive Watcher software

The interface is simple as can be seen in Figure 3.24 but from the section **Parameters Configuration** and **Register Configuration** is possible to monitoring in real time every state of the Micro digital One moreover the setting of all the parameters presented could be rewrite and set before the start up. Many other functions are present but not useful for this application.



(a) Parameters configuration

(b) Registers configuration

Figure 3.25: Parameters and Registers configuration sections

As can be seen in Figure 3.25, there are many parameters and even more registers that can be modified due to specific request upon the Micro digital One. In the user manual are present many tables where every specific parameters and registers are accurately described.

All the other parameters needed to set up the motor and drive it in a speed control loop are set via Nucleo board by CANopenNode command found in the library.

3.7.3 Micro digitl One set up via STM32F303RE

In 3.7.1 was presented the code in the main.c specifically the while(1) section where three different function were called. A more in depth analysis is required.

```
#include "../CANopenNode/301/CO_NMT_Heartbeat.h"
#include "../CANopenNode/301/CO_SDOclient.h"
#include "OD.h" // Object Dictionary CANopen

#include "../User_functions/Inc/User_Constants.h" // User defines, da mettere dopo le librerie di sistema e CANopen
#include "../User_functions/Inc/User_Variables.h" // Variabili utente

#include "canopen_states.h" // Header del modulo corrente
#include "canopen_config.h" // Config CANopen

void handle_off_state(void) {
    if (!nmt_stop) {
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6040, 0, (uint8_t*)&disableTorque, sizeof(disableTorque), 3);
        CO_NMT_sendInternalCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_ENTER_PRE_OPERATIONAL);
        CO_NMT_sendCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_RESET_NODE, 0x02);
        nmt_stop = 1;
    }

    if (State == READY) {
        Can = PAUSE;
    }
}

void handle_pause_state(void) {
    if (!nmt_po) {
        CO_NMT_sendInternalCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_ENTER_OPERATIONAL);
        CO_NMT_sendCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_ENTER_PRE_OPERATIONAL, 0x02);

        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6092, 1, (uint8_t*)&feedConst, sizeof(feedConst), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6098, 0, (uint8_t*)&homingMethod, sizeof(homingMethod), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x607C, 0, (uint8_t*)&homeOffset, sizeof(homeOffset), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6099, 2, (uint8_t*)&markerSpeed, sizeof(markerSpeed), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6060, 0, (uint8_t*)&homingMode, sizeof(homingMode), 3);

        for (int i = 0; i < sizeof(ctrlSeq)/sizeof(ctrlSeq[0]); i++) {
            safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6040, 0, (uint8_t*)&ctrlSeq[i], sizeof(ctrlSeq[i]), 3);
            HAL_Delay(5);
        }

        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6040, 0, (uint8_t*)&startHoming, sizeof(startHoming), 3);
        nmt_po = 1;
    }

    Can = ON;
}

void handle_on_state(void) {
    if (!nmt_op) {
        CO_NMT_sendCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_ENTER_OPERATIONAL, 0x02);
        nmt_op = 1;

        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x1803, 5, (uint8_t*)&time, sizeof(time), 3);

        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x1017, 0, (uint8_t*)&heartbeatTime, sizeof(heartbeatTime), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6092, 1, (uint8_t*)&feedConstant, sizeof(feedConstant), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6083, 0, (uint8_t*)&acceleration, sizeof(acceleration), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6084, 0, (uint8_t*)&deceleration, sizeof(deceleration), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6060, 0, (uint8_t*)&speedControlMode, sizeof(speedControlMode), 3);

        for (int i = 0; i < sizeof(controlWordSeq)/sizeof(controlWordSeq[0]); i++) {
            safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6040, 0, (uint8_t*)&controlWordSeq[i], sizeof(controlWordSeq[i]), 3);
            HAL_Delay(5);
        }

        write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x60FF, 0, (uint8_t*)&targetVelocity, sizeof(targetVelocity));

        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);

        if (HAL_GetTick() - lastTPDotime >= 1500) {
            OD_PERSIST_COMM.x6000_counter++;
            CO_TPDO_sendRequest(&canopenNodeSTM32->canOpenStack->TPDO[0]);
            lastTPDotime = HAL_GetTick();
        }
    }
}
```

Figure 3.26: canopen_config.c

handle_off_state

This function is recalled in the `main.c` but is defined in `canopen_config.c` source file, paired with `canopen_state.h` header file.

```
void handle_off_state(void) {
    if (!nmt_stop) {
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6040, 0, (uint8_t*)&disableTorque, sizeof(disableTorque), 3);
        CO_NMT_sendInternalCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_ENTER_PRE_OPERATIONAL);
        CO_NMT_sendCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_RESET_NODE, 0x02);
        nmt_stop = 1;
    }

    if (State == READY) {
        Can = PAUSE;
    }
}
```

Figure 3.27: handle_off_state

All the line present in the if must be executed only once after the Black Button is pressed namely the Nucleo Board reset so the motor drive by the Micro digital One first the motor must be stopped via SDO then the node 1 (STM32F303RE) must enter in pre-operational state while node 2(Micro digital One) is stopped. The state of the node are not managed via SDO with specific command in the CANopenNode library. Once the code is executed the State must be updated to the next.

handle_pause_state

The pause state is a transition state, its duty is to manages the homing operations of the Micro digital One, setting the encoder and resetting eventual alarms. Moreover the node 1 is set operational and the node 2 preoperational. Once these setting (via SDOs) are completed this state is concluded and the next one is started.

```
void handle_pause_state(void) {
    if (!nmt_po) {
        CO_NMT_sendInternalCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_ENTER_OPERATIONAL);
        CO_NMT_sendCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_ENTER_PRE_OPERATIONAL, 0x02);

        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6092, 1, (uint8_t*)&feedConst, sizeof(feedConst), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6098, 0, (uint8_t*)&homingMethod, sizeof(homingMethod), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x607C, 0, (uint8_t*)&homeOffset, sizeof(homeOffset), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6099, 2, (uint8_t*)&markerSpeed, sizeof(markerSpeed), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6060, 0, (uint8_t*)&homingMode, sizeof(homingMode), 3);

        for (int i = 0; i < sizeof(ctrlSeq)/sizeof(ctrlSeq[0]); i++) {
            safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6040, 0, (uint8_t*)&ctrlSeq[i], sizeof(ctrlSeq[i]), 3);
            HAL_Delay(5);
        }

        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6040, 0, (uint8_t*)&startHoming, sizeof(startHoming), 3);
        nmt_po = 1;
    }

    Can = ON;
}
```

Figure 3.28: handle_pause_state

As for the `handle_off_state` also the `handle_pause_state` must be executed once then after the set up is not needed anymore.

handle_on_state

The last state is the `handle_on_state` where node 2 is set operational, the first SDO is used to give a time reference to TPDO 4 of the Micro digital One that manage to transmit data about the actual velocity of the shaft. Then with the other block of SDOs all the parameters needed for a proper control are set feed constant, acceleration, deceleration and mode of operation (Velocity Control Mode) while in 'for' eventual alarm are reset. Once these operations are concluded the target velocity is set and therefore the motor motion is started. For debugging puporse the LED2 (LD2) is on during this state meanwhile a TPDO used as a counter continuously transmit its information in the network.

```
void handle_on_state(void) {
    if (!nmt_op) {
        CO_NMT_sendCommand(canopenNodeSTM32->canOpenStack->NMT, CO_NMT_ENTER_OPERATIONAL, 0x02);
        nmt_op = 1;

        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x1803, 5, (uint8_t*)&time, sizeof(time), 3);

        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x1017, 0, (uint8_t*)&heartbeatTime, sizeof(heartbeatTime), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6092, 1, (uint8_t*)&feedConstant, sizeof(feedConstant), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6083, 0, (uint8_t*)&acceleration, sizeof(acceleration), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6084, 0, (uint8_t*)&deceleration, sizeof(deceleration), 3);
        safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6060, 0, (uint8_t*)&speedControlMode, sizeof(speedControlMode), 3);

        for (int i = 0; i < sizeof(controlWordSeq)/sizeof(controlWordSeq[0]); i++) {
            safe_write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x6040, 0, (uint8_t*)&controlWordSeq[i], sizeof(controlWordSeq[i]), 3);
            HAL_Delay(5);
        }

        write_SDO(canopenNodeSTM32->canOpenStack->SDOclient, 0x02, 0x60FF, 0, (uint8_t*)&targetVelocity, sizeof(targetVelocity));
    }

    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);

    if (HAL_GetTick() - lastTPDotime >= 1500) {
        OD_PERSIST_COMM.x6000_counter++;
        CO_TPDOsendRequest(&canopenNodeSTM32->canOpenStack->TPDO[0]);
        lastTPDotime = HAL_GetTick();
    }
}
```

Figure 3.29: `handle_on_state`

Chapter 4

Control code implementation

4.1 Control strategy for the Nucleo Board

In the code section triggered by the TIM1 ISR, two different control algorithms for the S1402B353 Brushless Servomotor are implemented, as described in Chapter 2: the I-Hz control and Field-Oriented Control (FOC). For this specific application, the primary focus is on FOC, while the I-Hz control is implemented but not actively used—it remains available for debugging purposes.

More specifically, two variations of FOC are present in the code, identified by Ctrl_type values 2 and 3. Only Ctrl_type 3 is effectively utilized in the application, whereas Ctrl_type 2 is reserved for debugging.

4.1.1 Field-Oriented Control (FOC)

A briefly theoretical introduction to FOC is needed.

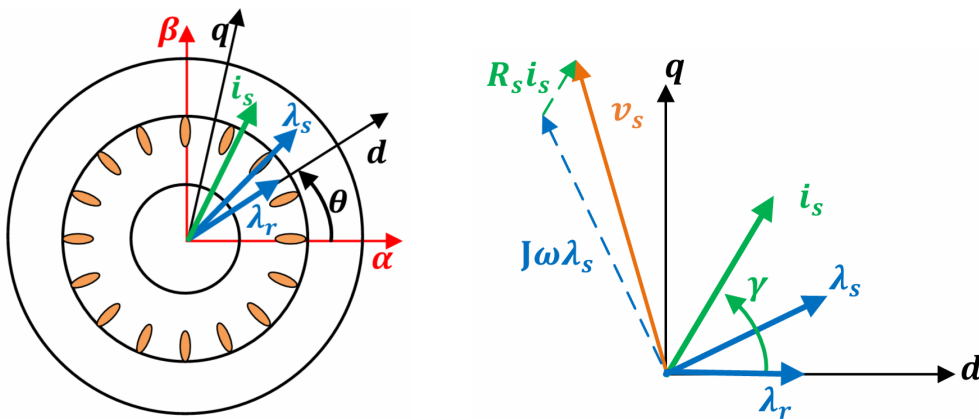


Figure 4.1: Induction Motor Diagrams

A common characteristic of all vector control schemes is the regulation of current in the rotating dq reference frame, which is established by the orientation angle θ , representing the phase of the rotor flux vector.

Therefore, by analyzing the stator and rotor models within the dq frame, the voltage equations can be expressed as follows:

$$\begin{cases} v_{s,dq} = R_s i_{s,dq} + \frac{d\lambda_{s,dq}}{dt} + \mathbf{J}\omega \lambda_{s,dq} \\ v_{r,dq} = 0 = R_r i_{r,dq} + \frac{d\lambda_{r,dq}}{dt} + \mathbf{J}(\omega - p\omega_r) \lambda_{r,dq} \end{cases} \quad (4.1)$$

While, the flux linkage equations are:

$$\begin{cases} \lambda_{s,d} = k_r \lambda_{r,d} + \sigma L_s i_{s,d} \\ \lambda_{s,q} = \sigma L_s i_{s,q} \end{cases} \quad (4.2)$$

The electromagnetic torque is defined in the dq FOC frame as follows:

$$T = \frac{3}{2} p \cdot k_r \lambda_r \cdot i_{s,q} \quad (4.3)$$

Specifically, the d -axis current is used to regulate the rotor excitation level, while the torque is controlled through the $i_{s,q}$ current component. This relationship is illustrated in Figure 4.2 and is defined by the steady-state correlation between rotor flux and currents, as expressed in equation (4.4).

$$\lambda_{r,\alpha\beta} = \frac{L_m i_{s,\alpha\beta}}{1 + j(\omega - p\omega_r) \tau_r} \quad (4.4)$$

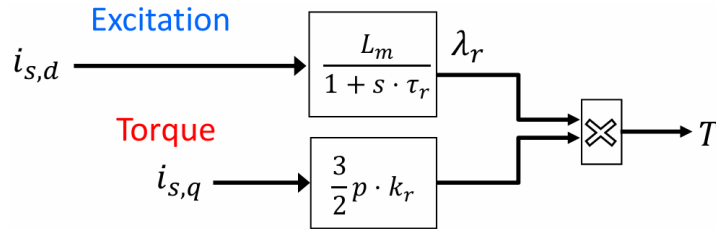
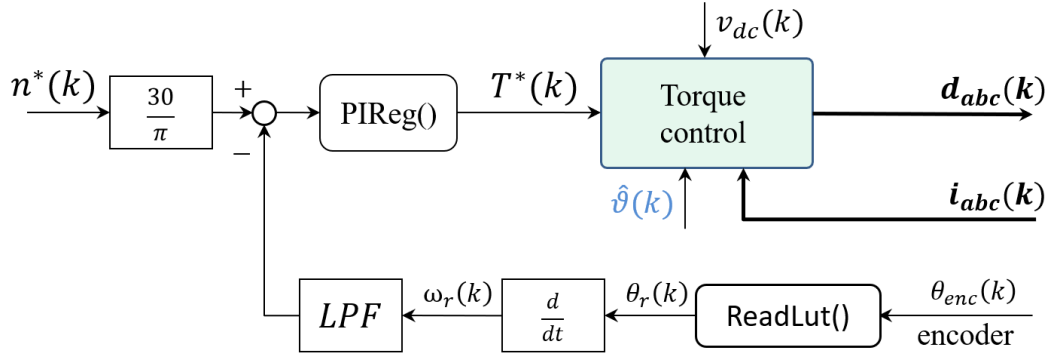
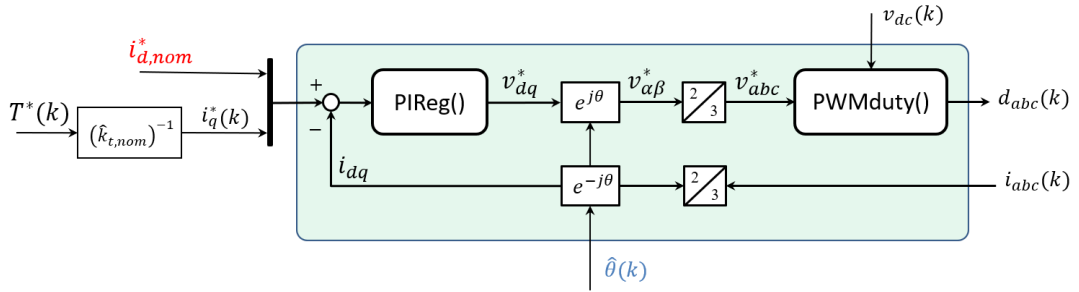


Figure 4.2: Torque Generation

The implemented FOC schemes operate as closed-loop speed controllers that utilize encoder feedback to measure rotor speed. The measured speed is filtered and compared to the reference speed to generate a torque reference signal within the speed control loop, as illustrated in Figur 4.3. This torque reference is then converted into a reference current along the q -axis, enabling rapid dynamic response and accurate tracking of torque changes. In contrast, the current reference on the d -axis is maintained constant at its nominal value to ensure proper excitation of the machine, as variations on this axis would not contribute effectively to dynamic torque control.



(a) Speed Control



(b) Torque Control

Figure 4.3: Block Diagram of FOC

The reference angle $\hat{\theta}$, which defines the orientation of the rotating dq reference frame, is determined differently depending on whether the control strategy follows Indirect FOC or Direct FOC principles.

Figure 4.4 presents the block diagram of the Indirect FOC scheme, which does not require flux estimation. In this approach, the orientation angle is computed in a feedforward manner using encoder feedback and is derived from the reference current components $i_{s,dq}^*$.

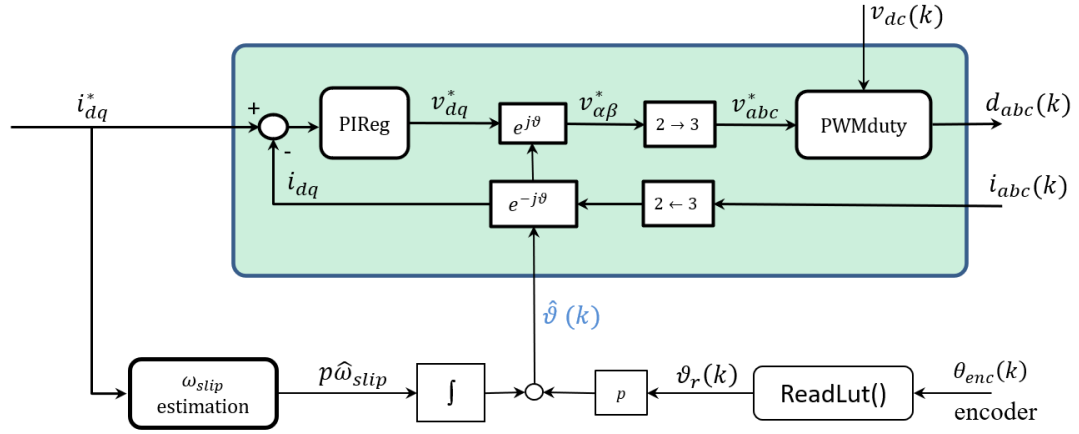


Figure 4.4: Block Diagram of the IFOC

Conversely, as shown in Figure 4.5, the orientation angle $\hat{\theta}$ for the dq reference frame in Direct FOC corresponds to the phase angle of the rotor flux linkage vector. As a result, this control strategy requires the implementation of a reliable and accurate flux estimator or observer to determine the rotor flux position in real time.

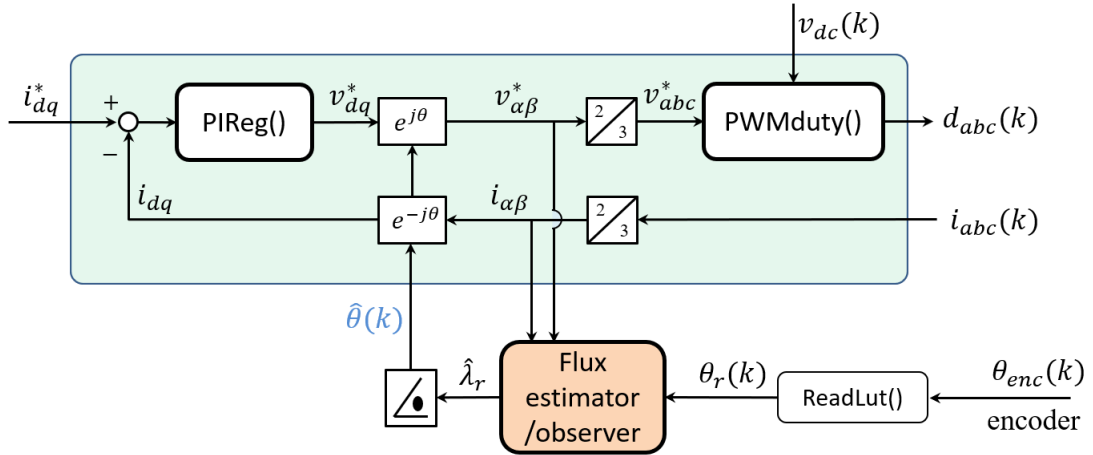


Figure 4.5: Block Diagram of the DFOC

The reference angle $\hat{\theta}$ is then computed from the estimated flux components in the $\alpha\beta$ stationary reference frame using the following expression:

$$\begin{aligned} \hat{\lambda}_{r,\alpha\beta} &= \hat{\lambda}_r \cdot (\cos \hat{\theta} + j \sin \hat{\theta}) \\ \cos \hat{\theta} &= \frac{\hat{\lambda}_{r,\alpha}}{\hat{\lambda}_r} \quad , \quad \sin \hat{\theta} = \frac{\hat{\lambda}_{r,\beta}}{\hat{\lambda}_r} \end{aligned} \quad (4.5)$$

Indirect Field-Oriented Control

Figure 4.6 illustrates the block diagram scheme of the simplest FOC implementation: the Indirect Field-Oriented Control (IFOC).

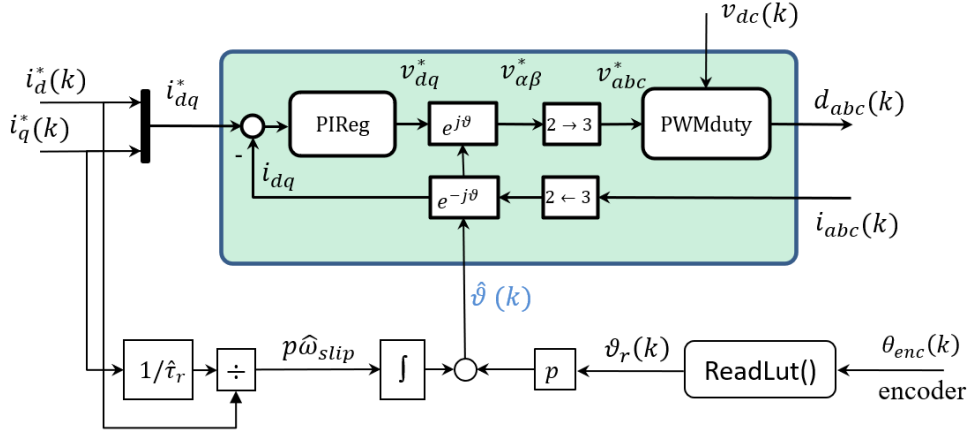


Figure 4.6: Block Diagram of the Indirect FOC

To define the dq -axes, a forward estimation of the rotor flux phase is performed using the steady-state rotor equation:

$$\lambda_r = \frac{L_m \cdot i_{s,q}}{p\hat{\omega}_{slip} \cdot \tau_r} \quad (4.6)$$

From (4.6), the phase angle of the rotor flux is indirectly estimated using the encoder readings (first corrected via software $\theta_{enc} \rightarrow \theta_r$) and the estimated rotor slip phase angle, as expressed by (4.7):

$$\begin{aligned} \hat{\theta} &= p\theta_r + \int (p\hat{\omega}_{slip}) dt \\ &= p\theta_r + \int \left(\frac{i_{s,q}^*}{\tau_r \cdot i_{s,d}^*} \right) dt \end{aligned} \quad (4.7)$$

The rotor slip frequency is estimated using the reference dq stator currents instead of the measured ones, following a forward model approach.

This estimation relies on a single motor parameter, the rotor time constant $\tau_r = \frac{L_r}{R_r}$, which must be accurately identified.

Despite this method is simple to implement, its accuracy is affected by variations in τ_r due to temperature changes, as rotor resistance R_r increases with heating.

Additionally, the rotor inductance L_r depends on the magnetizing inductance L_m , which varies with saturation, making the estimation sensitive to the operating point.

In this approach, the speed controller compares the reference mechanical speed with the filtered speed obtained from the encoder readings. The error between these two

signals is then used to generate a torque reference through a PI regulator.

This torque reference is then converted into a current reference along the q -axis, as defined by the (4.9).

Since $i_{s,d}$ is used to keep the rotor excited at the nominal flux λ_r^{nom} , torque is regulated solely by varying the current along the q -axis.

$$T = \frac{3}{2}p \cdot k_r \lambda_r \cdot i_{s,q} \quad (4.8)$$

From which:

$$i_{s,q}^* = \frac{T^*}{k_T} \quad \text{with} \quad k_T = \frac{3}{2}p \cdot k_r \lambda_r^{nom} \quad (4.9)$$

The orientation angle defining the dq reference frame for Field-Oriented Control (FOC) is computed as previously described and constrained within the range $0, 2\pi$ to avoid numerical overflow during long-term execution. Once this angle is available, the measured phase currents are transformed into the dq frame and processed by the current control loop, which calculates the corresponding voltage references. These voltage references are then used to determine the appropriate duty cycles for the inverter leg modulation.

4.1.2 FOC implemented in the Nucelo Board

```

case 3: // FOC -> torque

    if (counter < 10000){
        Tref = 0.4f;
        counter++;
    }
    else
        Tref = 0.6f;

    // Id, Iq reference
    isdq_ref.d = 0.0f;
    isdq_ref.q = Tref * INV_KT; // da Nm a Ampere

    // current loop
    _rot(isab, SinCos_r_elt, isdq);

    // feed-forward
    omega_r_filt_elet = PP * omega_r_filt;
    vsdq_ref_ffw.d = RS * isdq_ref.d - omega_r_filt_elet * LS * isdq_ref.q;
    vsdq_ref_ffw.q = RS * isdq_ref.q + omega_r_filt_elet * (LS * isdq_ref.d + LAMBDA_M);

    //d-axis current control loop
    id_var.ref = isdq_ref.d;
    id_var.fbk = isdq.d;
    id_par.lim = SQRT10OVER3 * vdc - vsdq_ref_ffw.d;
    PIReg(&id_par, &id_var);
    vsdq_ref.d = id_var.out;

    //q-axis control loop
    iq_var.ref = isdq_ref.q;
    iq_var.fbk = isdq.q;
    iq_par.lim = sqrtf(id_par.lim * id_par.lim - id_var.out * id_var.out) - vsdq_ref_ffw.q;
    PIReg(&iq_par, &iq_var);
    vsdq_ref.q = iq_var.out;

    // feedforward sum
    vsdq_ref.d += vsdq_ref_ffw.d;
    vsdq_ref.q += vsdq_ref_ffw.q;

    // invert transformation
    _invrot(vsdq_ref, SinCos_r_elt, vsab_ref);

    break;
}

```

Figure 4.7: FOC Code, Ctrl_type 3

The previous Figure 4.7 represents the specific code used in this application.

The torque provided to the control is constant after an initial step variation controlled by the counter variables. The reference current for d and q axis are set immediately after, `isdq_ref.d` is equal to 0, `isdq_ref.q` is calculated from the reference torque and the constant `INV_KT`.

The currents in alpha-beta reference coordinates are switched to dq thanks to a custom macro, after the feed-forward voltages are computed (dq axis) with the `omega_r_filt_elet` obtained by previous calculation from `omega_r_filt`. Then the d-axis current loop is computed followed by the q-axis. In the end of the script the voltages are computed from dq-axis to the alpha-beta in order to correct the error in the loop.

4.2 Control configuration for Micro digital One

As already mention the Micro digital One is set with a velocity control, unfortunately from the refence manual there are no detailed infomation about the code implementation from Microphase. Only few paramters can be changed from the software Drive Watcher or via SDOs, like accelration and deceleration ramp and the feed constant has mentioned in Chapter 3 (3.7.3)

Chapter 5

Motors control results

This chapter discusses the results of several tests conducted on the developed test bench. The primary focus is on evaluating the performance of the Field-Oriented Control (FOC) algorithm implemented using STM32CubeIDE. This emphasis is justified by the fact that the speed control provided by the Micro Digital One allows for only limited parameter adjustments, thereby constraining the scope of the tests that can be performed with that system. It is important to note that achieving a perfectly tuned and optimized FOC was not the primary objective of this thesis. Instead, the main goal was the successful implementation of the CANopen communication network between the two microcontrollers. That said, the FOC must still meet certain requirements to ensure proper motor control. These include the ability to accurately track the reference speed, maintain system stability, and respond effectively to dynamic changes in operating conditions.

5.1 Data retrived from speed control

This section focuses on the speed control system managed by the Micro Digital One. The **S1601B303 Brushless Servomotor** is actively controlled using a speed control strategy, while the **S1402B353 Brushless Servomotor** operates as a passive mechanical load. Unfortunately, no direct data were acquired from the Micro Digital One; only indirect measurements, obtained via the STM32F303RE, are available. As a result, no data are presented in this section due to their limited relevance. Nonetheless, the available waveforms confirm the correct operation of the motor control algorithm implemented on the Micro Digital One. The controller reaches and maintains steady-state conditions, with minimal current ripple and no observable instability. For a more comprehensive performance evaluation, it would be advisable to include both the actual motor speed and its reference signal, in order to directly assess the dynamic behavior of the speed control loop.

5.2 Data acquired by speed and FOC control over the test bench

After a brief analysis of the speed control system in isolation, it is now appropriate to evaluate the behavior of the complete test bench. The following results are obtained from the combined operation of both control strategies: the speed control handled by the Micro Digital One and the Field-Oriented Control (FOC) executed by the STM32F303RE.

This integrated assessment provides a more comprehensive understanding of the system's overall performance and its ability to manage coordinated motor control via the CANopen communication network.

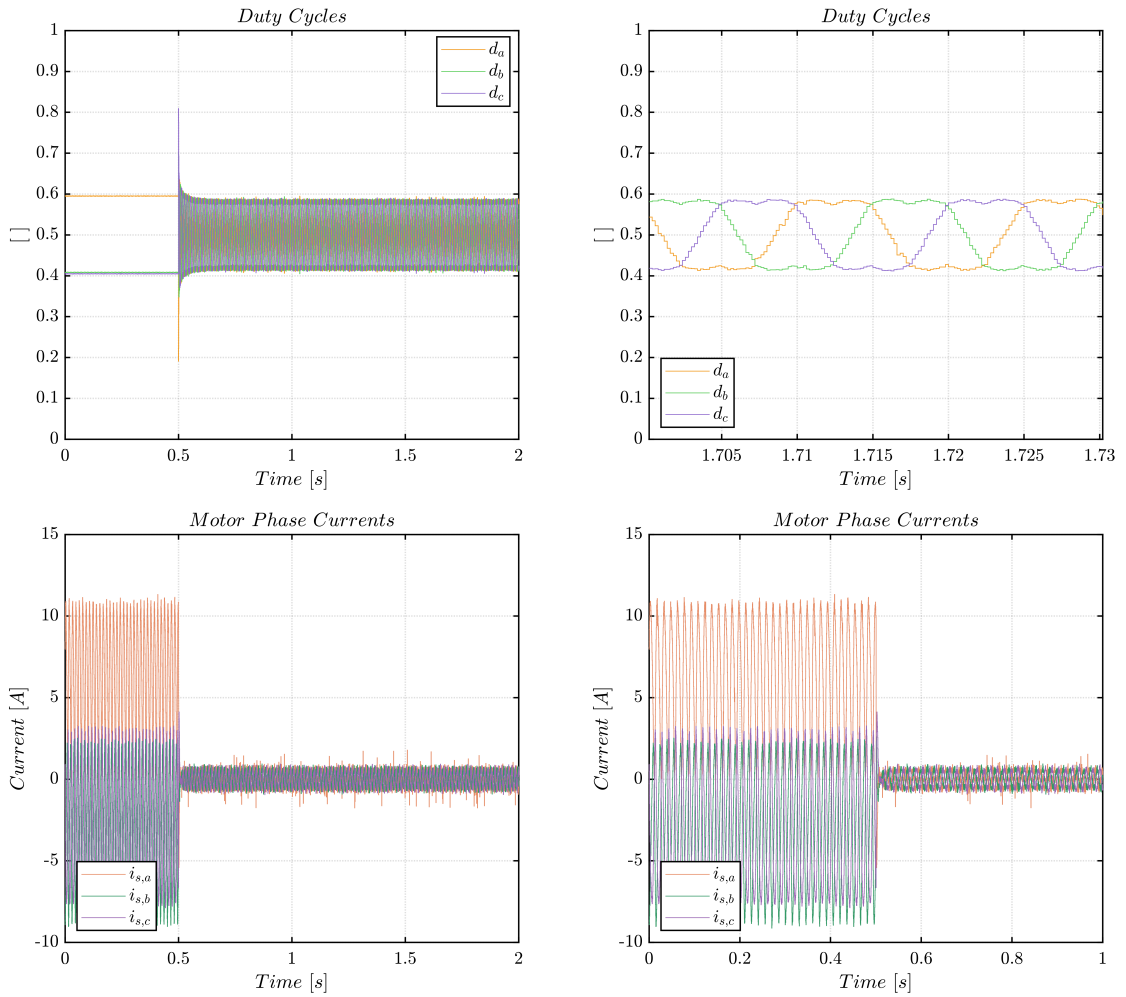


Figure 5.1: Measurements for speed control and FOC control

The figure displays two measurements related to the speed control of two motors connected in a back-to-back configuration, using Field-Oriented Control (FOC). Here are

represented the duty cycles and a consequent zoom over 2 periods and the motors phase currents with particular attention towards the start of the FOC control. The control strategy aims to maintain a desired speed and current profile under dynamic conditions. The plots can be interpreted as follows:

- **Duty Cycles** The duty cycles d_a , d_b and d_c exhibit a transient response at around $t = 0.5$ s, followed by stabilization. This transition corresponds to the engagement of the second motor in the back-to-back scheme. Specifically the control switch from the speed control alone to the speed and FOC combined. The stable PWM modulation after the transient suggests that the inverter is maintaining steady-state switching signals effectively.
- **Motor Phase Currents** The currents i_a , i_b and i_c show a clear transition around $t = 0.5$ s, followed by balanced three-phase sinusoidal waveforms. This indicates that after an initial adjustment period, the FOC controller successfully regulates the motor currents, achieving phase balance and current ripple control. The initial high amplitude may reflect motor start-up or torque synchronization.

5.2.1 Analysis of bench test speed

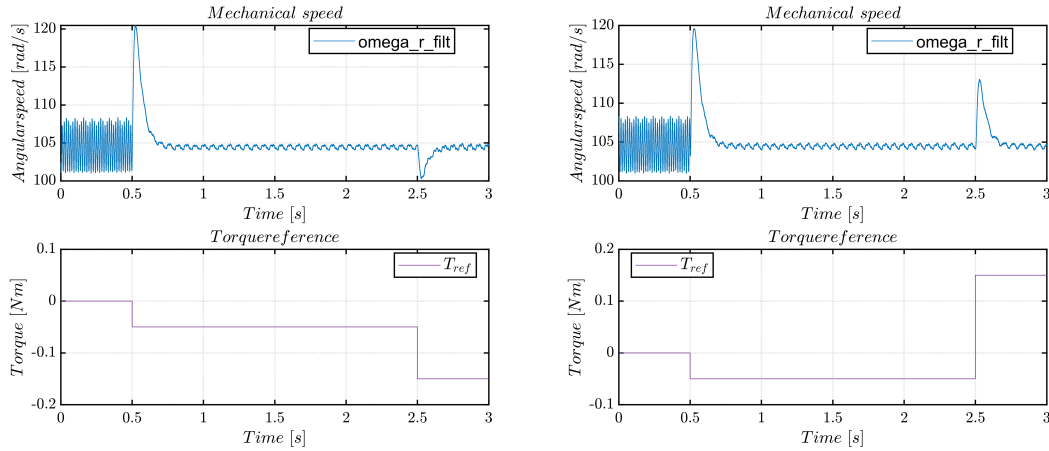


Figure 5.2: Test bench speed confrontation

The Figure 5.2 illustrates the angular velocity profiles of the test bench paired with the torque reference of the FOC control. A couple observations can be done:

- At the beginning of the plot, a transient peak is observed, corresponding to the activation of FOC control and therefore of the torque reference. Following this, ω_{r_filt} stabilizes quickly at a nearly constant value, indicating effective dynamic performance and stability of the speed control loop.

- The spikes in the ω_r filter represent the loading from the step variation of the reference torque of the FOC control. A positive spike is bond with a positive step in torque while a negative spike is bond with a nrgtive step in torque. In both the scenario the gain of the speed loop act quickly in order to maintain the reference speed.

5.3 Fault Test

A critical test is performed on the test bench to assess its behavior under fault conditions. The worst-case fault scenario occurs when Field-Oriented Control (FOC) is not active, resulting in a sudden drop of the shaft speed to zero.

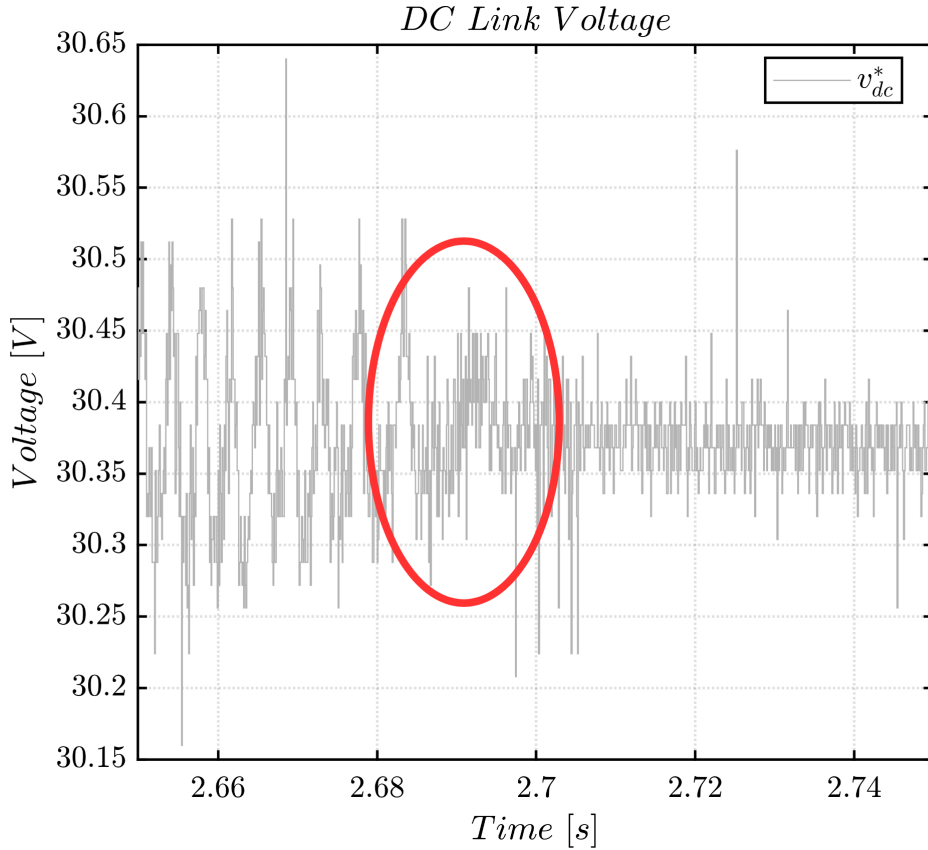


Figure 5.3: Test bench speed confrontation

Figure 5.3 shows the DC link voltage of the generator supplying the test bench. This parameter is a crucial aspect when evaluating the safety of the bench. Under normal operating conditions, the DC link voltage remains approximately constant. However, during this type of fault, the kinetic energy released by the abrupt stop of the shaft is

converted into reverse current flowing back into the generator. Since the generator is not capable of energy regeneration, its only response to this backflow is a rise in the DC link voltage (V_{DC}). The V_{DC} value must remain within a specific range, with any voltage spike not exceeding 15 V. The test demonstrates a very good resilience of the system against this type of fault. Cause as soon as the V_{DC} starts rising drops back to a steady value.

Chapter 6

Additional software

In this chapter will be briefly discussed the auxiliary software to the main tools. They are two, the **PCAN-View** and the **Drive Watcher** already mentioned before for some aspects.

6.1 Drive Watcher

Drive Watcher is a software developed by Salema for Microphase.

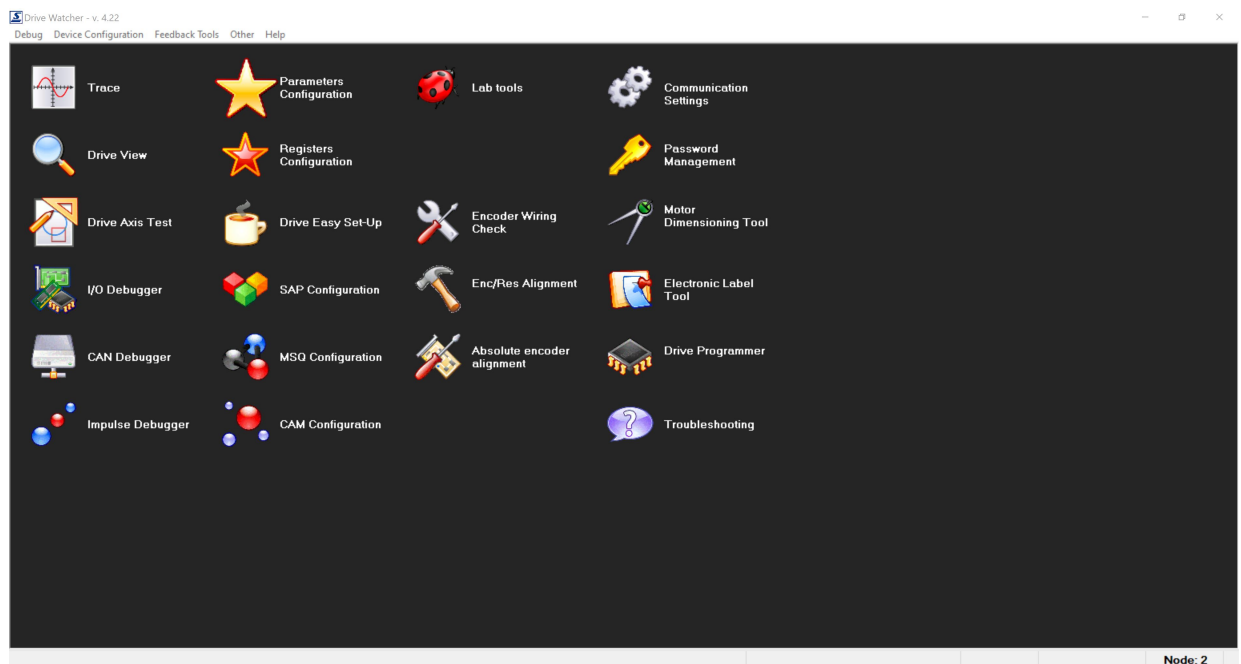


Figure 6.1: Drive Watcher

In Figure 6.1 is visible the software in its integrity. There are many functions in foreground, but not all of these are completely accessible without specific keys.

The following functions are not accessible without a specific password/ID:

- I/O deubber;
- Lab tools;
- Password Management;
- Electroic label tool;
- Drive programmer.

All the other functions can be used but only few of them were useful for this specific application.

The most relevant functions are:

- Parameters Configuration;
- Registers Configuration;
- Drive Easy Set-Up;
- Communication Settings.

6.1.1 Functions description and usage

In the following part all the relevant functions will be discussed and explained with a focus on their role in the project and possible use.

Parameters configuration

An example as this section of the program as it shows, was previously reveals in Figure 3.25.

The crucial function of this part of the program was already mentioned in Chapter 3. Now the goal is to give a more specific description of all the parameters and how can be set.

The portion Parmaters Configuration appears as in Figure 6.2

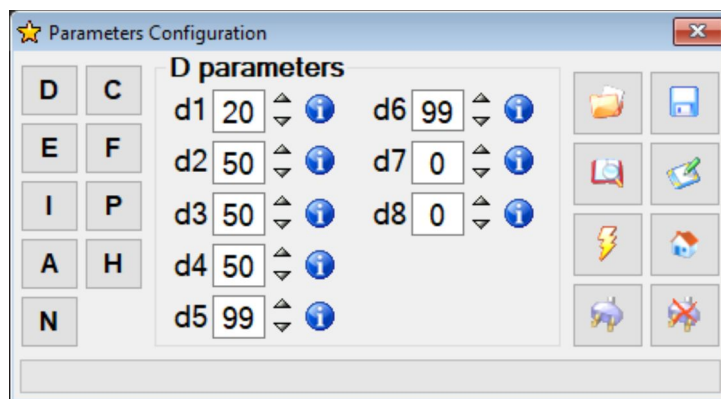


Figure 6.2: Drive Watcher

It's possible to identify three sections in this view.

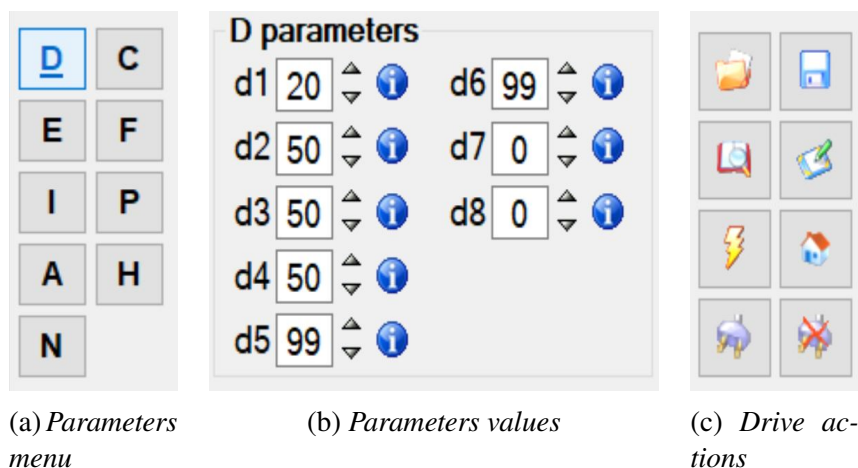


Figure 6.3: Sections

In the **Parameters menu** every letter regroup a certain numbers of paramters that manange a specific portion of the Micro digital One.

- **D paramters:**In this menu, it is possible to view and modify: the motor selection, certain current-related parameters, and parameters related to speed and its control loops. It is important to note that when the motor is selected, the Drive automatically sets the correct scaling factors and all the data related to the current loops. Therefore, the parameters to be set are exclusively application-dependent (e.g., moment of inertia, etc.). **WARNING!** Parameters d5, d6, d7, and d8 must not be modified while TEN is active and the axis is in regulation mode (either locked or in motion);
- **C paramters:**This menu allows modification of the drive control parameters. This menu is extremely important and must be used with great caution and only after fully understanding the details of each parameter. Here, you select the operating modes (analog setpoint, digital setpoint, or integrated mechatronic functions, and finally, the implemented fieldbus networks). Furthermore, a very important parameter to be set is the motor rotation direction, which is, of course, application-dependent;
- **E paramters:**This menu allows modification of the following three specific drive parameters. Two of them (E1 and E2) affect the speed setpoint, and the third affects the operational behavior of certain Drive conditions;
- **F paramters:**This menu enables the setting of the ramp parameters for the speed setpoint;
- **I paramters:**In this menu, it is possible to view and modify the main parameters of the drive's positioning system section (position loop). This menu is available in all options involving the space loop and allows only limited parameter modification of the Drive's Positioning section. For maximum flexibility, it is advisable to use the Drive Watcher software or operate through the FIELDBUS by accessing the REGISTERS section as well;
- **P paramters:** all these paramters contain the values for Pulse position (not used in this specific application);
- **A paramters:** Axis paramters define all the aspects about the axis motion and error for mesurements (default value not changed for this application);
- **H paramters:**This section describes the various methods by which the drive searches for the axis zero position. You can use two limit switches placed at the stroke ends or a home switch placed at any point along the path. Some methods, after finding the switch, search for the encoder's index pulse (marker) to achieve higher precision. The user can specify both the homing speeds and acceleration,

bearing in mind that the method uses a higher speed to search for the switch and a lower speed for the potential index pulse search;

- **N paramters**: only one parameter to set CANbus Baud rate.

This is a brief description of the group of paramters and what aspect they cover, mre details are present in the manual of the **Micro digital One**.

Parameters value section is pretty much self explanatory, here are listed all the paramters of a certain group were they value can be changed and more detailed information about the single paramters are given clicking on the info buttuno (the blue one near the paramters' value).

The **Driver actions** section is not proper of the **Parameters Configuration** view. It appears in many other view because its role is to give the possiblity to user to interact with the the **Micro digital One** directly.



Figure 6.4:
Open con-
figuration
file

Open configuration file: Once pressed, if present you can open a configuration file for all your parmeters and instantly apply it.



Figure 6.5:
Save con-
figuration
file

Save configuration file: Once pressed, permits to create a configuration file that can be later upload.



Figure 6.6:
Read from
device

Read from device: Once pressed, the parameters that are previously uploaded to the Micro digital One are written to the section Parameters value.



Figure 6.7:
Write from
device

Write from device: Once pressed, the parameters that are written in the section Parameters value are uploaded to the Micro digital One.



Figure 6.8:
Save configuration

Save configuration: Once pressed, the parameters that are written in the section Parameters value are saved to the Micro digital One; in contrast to uploaded that can be erased after the Micro is switched off the saved parameters are permanent.



Figure 6.9:
Load default
configuration

Load default configuration: Once pressed, the parameters are restored to the default value.

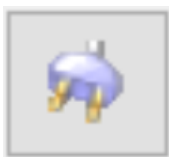


Figure 6.10:
Start com-
munication
with device

Start communication with device: Once pressed, the communication with the device is started.

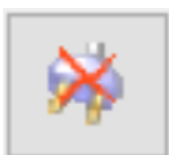


Figure 6.11:
Stop com-
munication
with device

Stop communication with device: Once pressed, the communication with the device is stopped.

Registers configuration

The Register configuration panel is conceptually similar to the Parameters configuration one, as is shown in Figure 6.12.

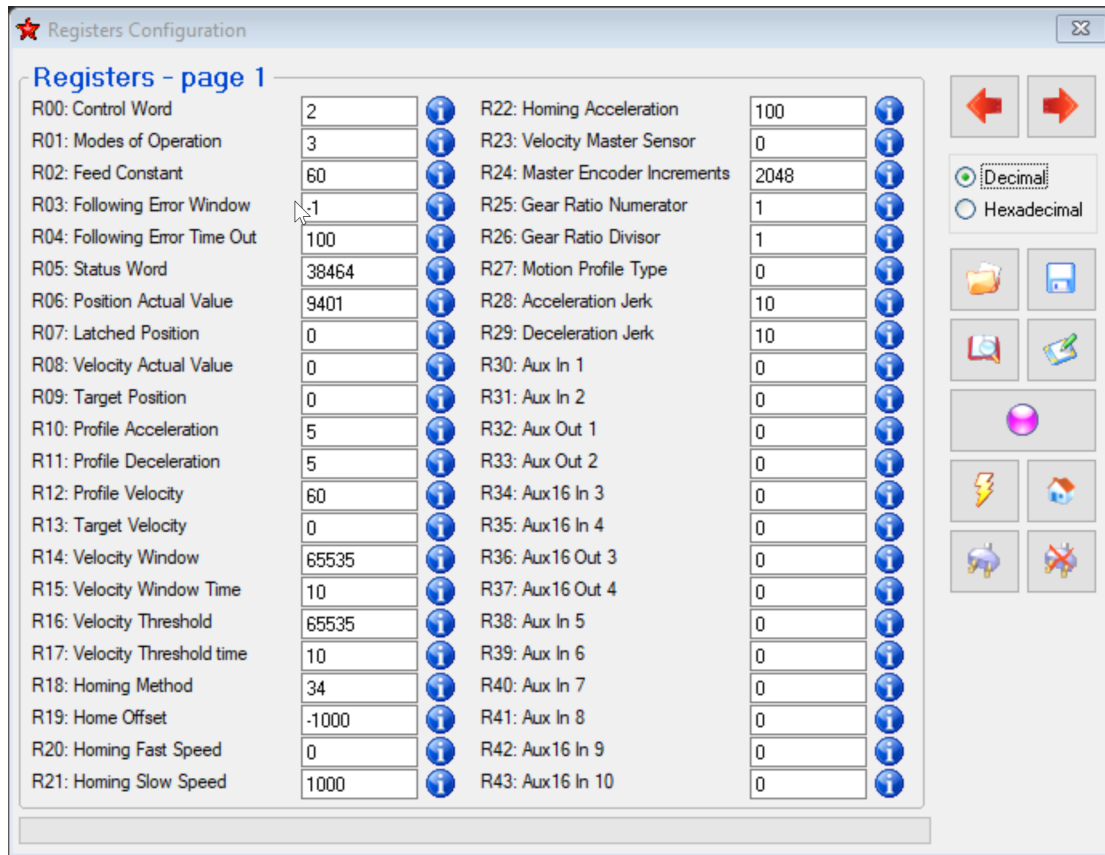


Figure 6.12: Register configuration panel

The section of the panel are only two this time. One where the value of the registers can be setted (with a proper 'info' button) and a second where are present all the button descibed before and three new.



Figure 6.13:
Load default
configura-
tion

Back button: Once pressed, the panel goes back to the previous page .



Figure 6.14:
Forward button

Start communication with device: Once pressed, the panel goes to the next page.

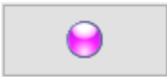


Figure 6.15:
Continuously
reading

Stop communication with device: Once pressed, the registers of the device are continuously reading.

Directly under the back and forward button another small section is present.

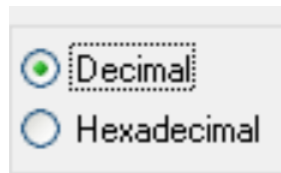


Figure 6.16: Data format

Here can be set the data format for the registers between decimal or hexadecimal.

Drive easy set-up

In this section of the program the **Micro digital One** can be set up from the start without compiling the Parameters and Register configuration portions. It is presented as in Figure 6.17.

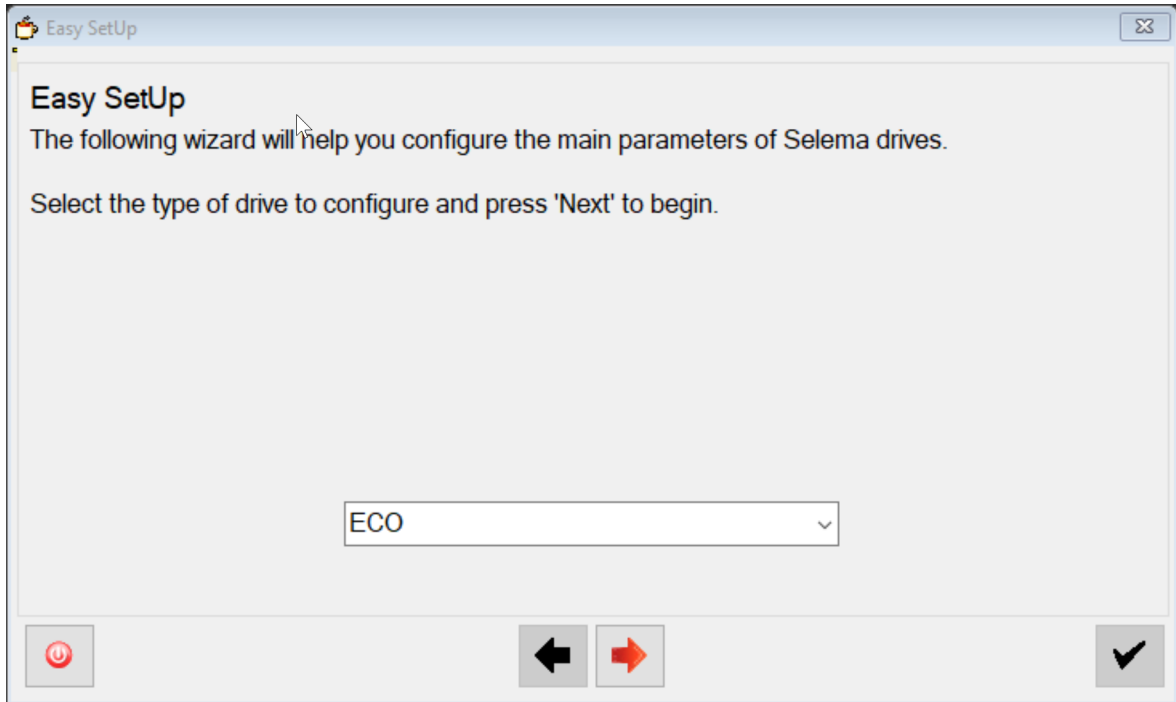


Figure 6.17: Driver Easy set-up panel

A series of direct question are asked to the user, who can navigate throu the question with the back and forward button, in order to set the essential paramters of the driver. Once the configuration is over with the 'tick' button in the bottom right corner the changes are saved and the drive is ready to operate.

Communication Settings

The Communication Settings can be seen as the most important portion of the program; beacuse if are not correctly set there is no communication between the **Micro digital One** and the PC. As already mentioned the **Micro digital One** has no USB port, so it's only way to communicate to a PC is to utilized the **CN5** or **CN6** port. An RJ45 to DB9 cable is required with a specific DB9 to USB interface provided by Moxa, the Uport 1130I. To ensure a correct connection in the Device Manager of Windows under serial Multiport, Port Settings, Port Number and Intrface must be set properly as reported in Figure 6.18

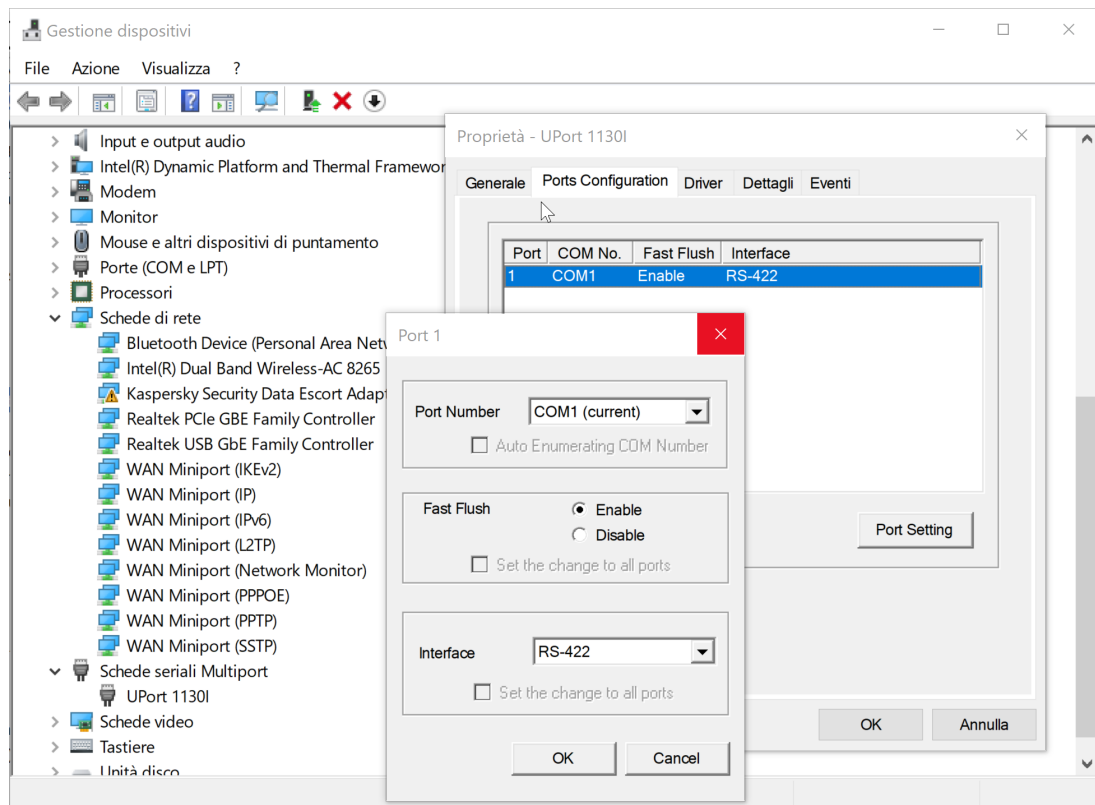


Figure 6.18: USB configuration

Port Number is the current PC port where the DB9 to USB interface is connected and Interface is which type of communication protocol is desired. Once this preliminary steps are completed the same setting must be reported in Communication Settings in Drive Watcher.

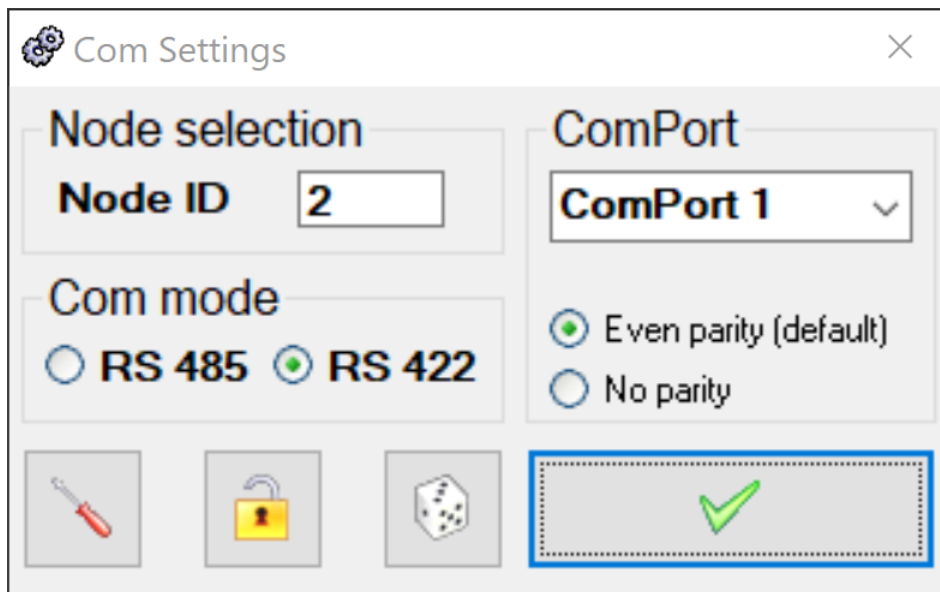
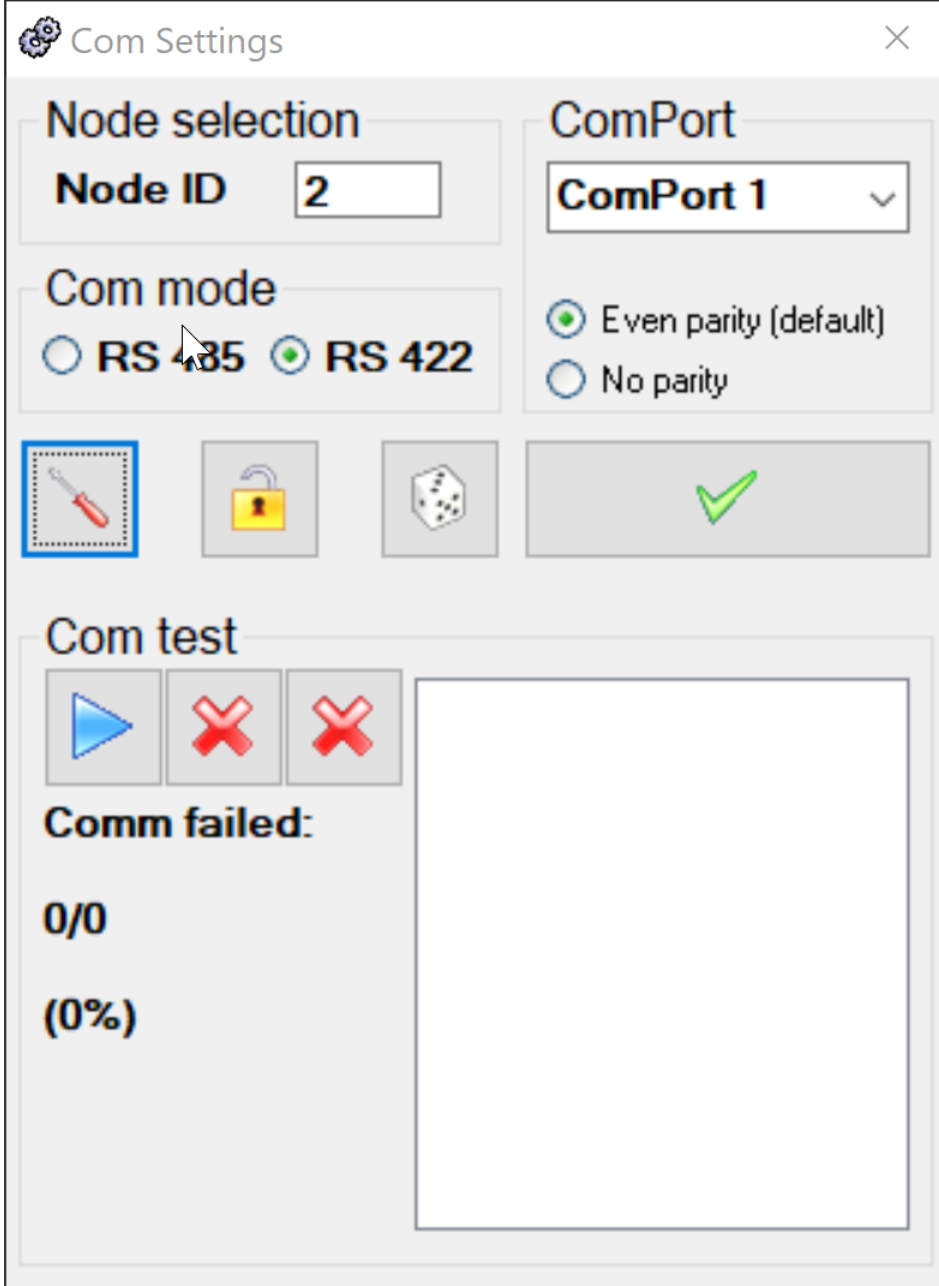


Figure 6.19: Communication settings



Figure 6.20:
Com-test

ComPort communication test: Once pressed, is possible to test the PC and driver connection with three simple button: start, reset and stop.



The image shows a 'Com Settings' dialog box with a close button (X) in the top right corner. It is divided into several sections:

- Node selection:** Contains a 'Node ID' text box with the value '2'.
- ComPort:** A dropdown menu showing 'ComPort 1'.
- Com mode:** Two radio buttons: 'RS 485' (unselected) and 'RS 422' (selected). A mouse cursor is pointing at the 'RS 485' button.
- Parity:** Two radio buttons: 'Even parity (default)' (selected) and 'No parity' (unselected).
- Icons:** A row of four icons: a screwdriver (highlighted with a blue dashed border), an open padlock, a die, and a green checkmark.
- Com test:** A section containing three buttons (a blue play button and two red X buttons) and a large empty rectangular area for test results.
- Comm failed:** Text indicating '0/0' and '(0%)'.

Figure 6.21: Communication settings and test section

6.2 CANopenEditor - EDS editor

This specific software can be found and download paired with the CANopenNode open source library. Object Dictionary Editor is an external graphical tool used to configure and edit the CANopen Object Dictionary of a custom device. It automatically generates the corresponding C source code, the Electronic Data Sheet (EDS), and the associated documentation required for the device's integration within a CANopen network.

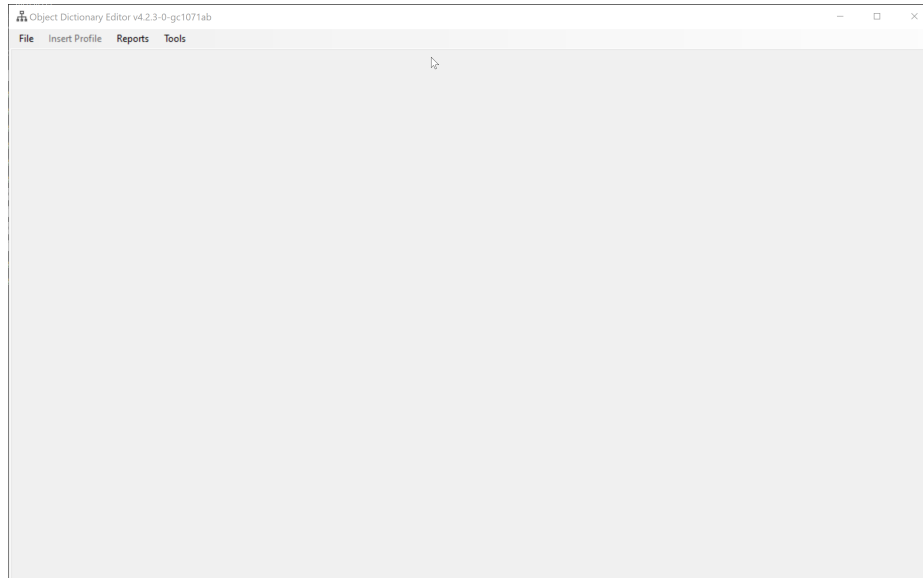


Figure 6.22: OD editor first look

At first sight the software is empty, only a toolbar in the top is present. In order to have access to modify the EDS file of the STM32F303RE is necessary to press File on toolbar then Open and select the .xpd file present in the CANopenNode_STM32 folder.

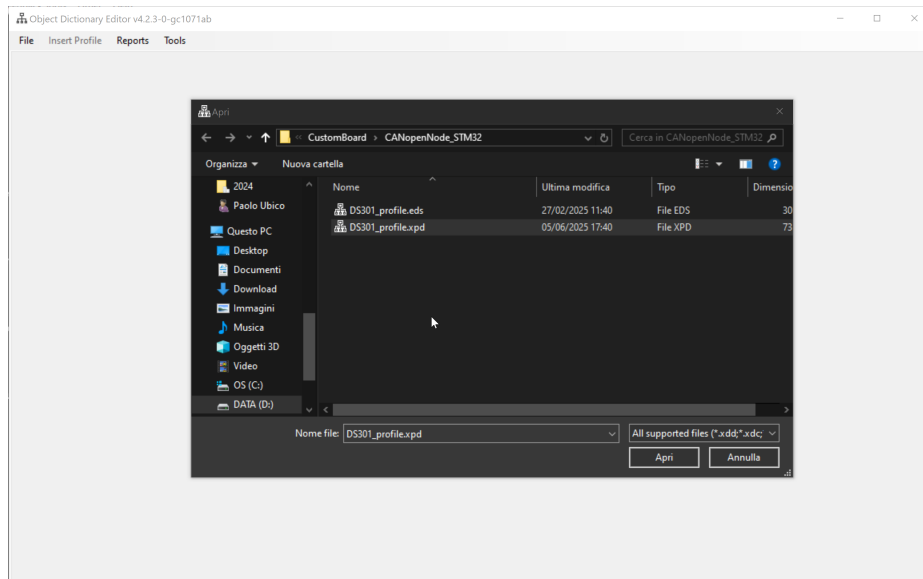


Figure 6.23: OD editor opening file .xpd

The software will appear as shown in Figure 6.24.

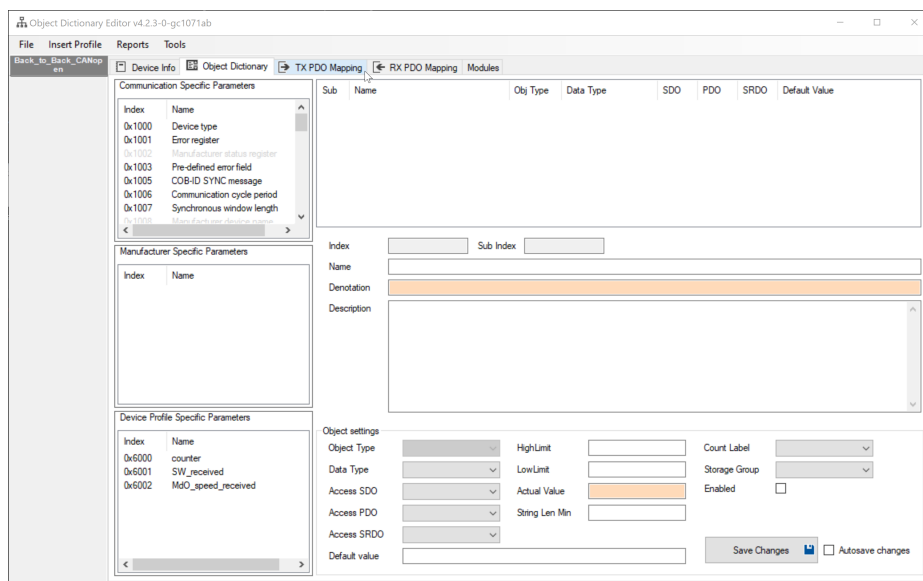


Figure 6.24: OD editor with file .xpd open

A new set of tabs are available now, each of them cover a specific aspect of the .xpd file.

Device Info

Inn this tab all the relevant information about the EDS file are reported organized in specific subsections. The only field that is modified is the project name.

Figure 6.25: ODE - Device Info

Object Dictionary

This tab is divided in four parts, three on the left side and one bigger on the right. On the left are reported all the parameters of a specific device:

- Communication Specific Paramters: only communication ones are present here;
- Manufactuer Specifi Pramters: here are set all the paramters created by manufacturer;
- Device Profile SSpecific Paramters: here are placed alle the prameters defined by user and the ones editable by the user.

On the right side are presented a series of information bonded with the paramter selected. The most relevant in this application are the ones under the Object settings.

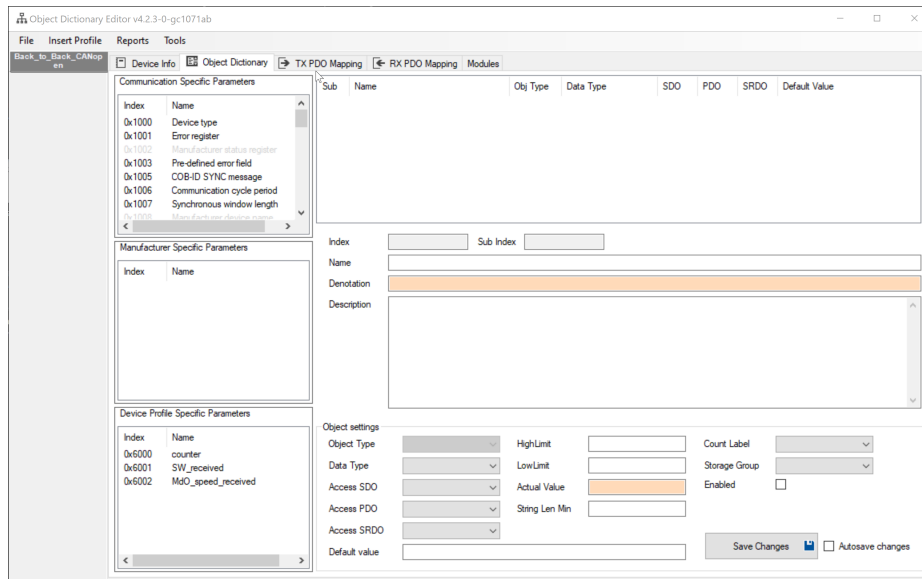


Figure 6.26: ODE - Object Dictionary

Once created a new Device Profile Specific Parameters (by right click of the mouse in the specific session), all the values, beside Object Type, can be set by the user. The following list contains the must set parameters, meanwhile the other can be left empty.

- Data type: define the type of data;
- Access SDO: define how this parameter can interact via SDOs (no, ro, wo, rw -> no, read only, write only, read and write);
- Access PDO: define how the parameter can interact via PDOs (no, ro, wo, rw -> no, read only, write only, read and write);
- Storage Group: indicate where the parameter is stored;
- Default value: here the default value defined by user is set.

TX PDO Mapping

In this tab the TPDO are created.

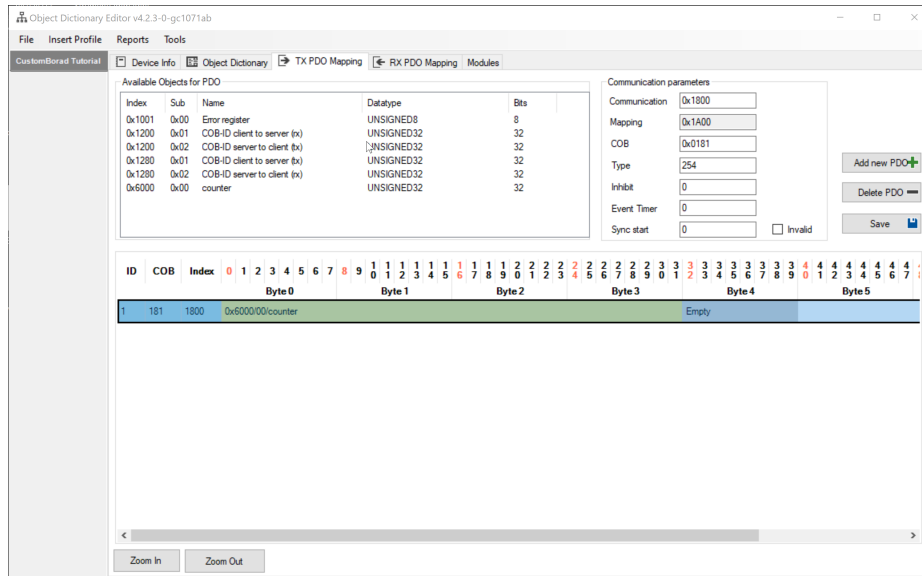


Figure 6.27: ODE - TX PDO Mapping

Three main sections are present in this tab:

- Available Objects for PDO: list of all the objects for PDO;
- Communications parameters: editable parameters of the TPDO;
- TPDO message: graphical representation of the TPDO message COB-ID, INDEX, and data bytes.

By clicking the button "Add new PDO" a new TPDO is created in the specific section, all the communication parameters can be set accordingly and the objects to be inserted are available in the list. By clicking the "Save" button the TPDO is saved.

RX PDO Mapping

This tab works in the exact same way as the **TX PDO Mapping** described in the previous section.

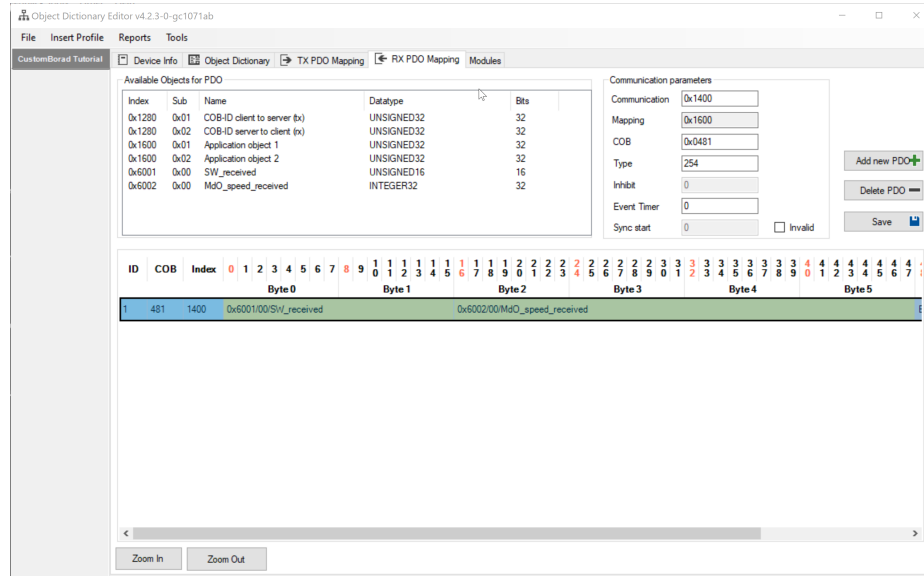


Figure 6.28: ODE - RX PDO Mapping

Modules

For this application this part of the code is not involved and therefore there is no point in describing it.

6.3 PCAN-View

PCAN-View is a Windows-based diagnostic and monitoring software developed by PEAK-System Technik GmbH. It serves as a lightweight, standalone application designed to interface with Controller Area Network (CAN) and CAN FD buses through PEAK-System hardware. This tool is widely used in both research and industry for the real-time inspection, transmission, and logging of CAN messages.

PCAN-View allows users to observe the traffic on a CAN network by displaying received and transmitted messages in a structured format. The software supports standard CAN (11-bit identifiers) as well as extended CAN (29-bit identifiers), and it also provides compatibility with CAN FD (Flexible Data-Rate) frames.

Key features include:

- Real-time CAN message monitoring: Messages are timestamped and shown in a continuous stream, allowing for dynamic analysis of network activity.
- Message transmission: Users can manually construct and send CAN messages with custom identifiers, data lengths, and payloads.
- Filter configuration: Acceptance filters can be applied to limit the scope of received messages, enhancing readability and focus.
- Logging capabilities: Sessions can be saved to log files in various formats for offline analysis.
- Bus load measurement: Provides statistics on message rate and bus utilization.

Internally, PCAN-View communicates with the CAN controller via PEAK's device drivers, enabling low-latency data exchange with the hardware interface. It operates in user space without requiring programming, making it suitable for both engineering validation and troubleshooting tasks.

The image shows the PCAN-View software interface. It has a menu bar (File, CAN, Edit, Transmit, View, Trace, Window, Help) and a toolbar. Below the toolbar are several icons for functions like Receive/Transmit, Trace, PCAN-USB FD, Bus Load, and Error Generator. The main area is divided into two sections: 'Receive' and 'Transmit'.

Receive Section:

CAN-ID	Type	Length	Data	Cycle Time	Count
702h		1	05	3973,0	12
701h		1	05	248,8	195
602h		8	23 FF 60 00 58 02 00 00	504,6	25
502h		8	60 FF 60 00 00 00 00 00	504,6	25
402h		6	37 06 81 02 00 00	1408,6	33
181h		4	38 00 00 00	750,1	60
000h		2	01 02	6030,9	2

Transmit Section:

CAN-ID	Type	Length	Data	Cycle Time	Count	Trigger	Comment
000h		2	01 01	Wait	0		Operational STM32
000h		2	01 02	Wait	0		Operational Micro
000h		2	02 01	Wait	0		Stop STM32
000h		2	02 02	Wait	0		Stop Micro
000h		2	80 00	Wait	0		Broadcast stato HB pre-operational
000h		2	81 01	Wait	0		Reset Nodo 1
000h		2	81 02	Wait	0		Reset Nodo 2
000h		2	82 01	Wait	0		HB STM32
000h		2	82 02	Wait	0		HB Micro
602h		4	40 00 20 00	Wait	0		Leggi drive settings
602h		6	28 40 60 00 02 00	Wait	0		STOP MOTORE (disabilitazione asse)
602h		6	40 08 10 00 00 00	Wait	0		Device name
602h		8	2B 17 10 00 A0 0F 00 00	Wait	0		4 secondi intervallo heartbeat
602h		8	2B 17 10 00 D0 07 00 00	Wait	0		2 secondi intervallo heartbeat

At the bottom, a status bar shows: Connected to hardware PCAN-USB FD, Device ID 0h, Bit rate: 500 kbit/s, Status: OK, Overruns: 0, QxmtFull: 0.

Figure 6.29: PCAN-View main page

As shown on Figure 6.29 the PCAN in its main view is basically spilt in two main sections: **Receive** and **Transmit**.

6.3.1 Receive

In this section can be seen all the messages transmitted by every node in the CANopen network.

702h	1	05	3975,0	8
701h	1	05	250,0	123
602h	8	23 FF 60 00 58 02 00 00	504,6	25
582h	8	60 FF 60 00 00 00 00 00	504,6	25
482h	6	37 06 42 02 00 00	1499,1	21
181h	4	23 00 00 00	750,1	36
000h	2	01 02	6030,9	2

Figure 6.30: PCAN-View Receive section

Here are present the messages transmitted and received by the two nodes of this application specifically node 1 and node 2. The different type of messages SDOs , TPDOs, ecc. are discriminated by their COB-ID (called CAN-ID).

6.3.2 Transmit

000h	2	01 01	Wait	0	Operational STM32
000h	2	01 02	Wait	0	Operational Micro
000h	2	02 01	Wait	0	Stop STM32
000h	2	02 02	Wait	0	Stop Micro
000h	2	80 00	Wait	0	Broadcast stato HB pre-operational
000h	2	81 01	Wait	0	Reset Nodo 1
000h	2	81 02	Wait	0	Reset Nodo 2
000h	2	82 01	Wait	0	HB STM32
000h	2	82 02	Wait	0	HB Micro
602h	4	40 00 20 00	Wait	0	Leggi drive settings
602h	6	28 40 00 00 02 00	Wait	0	STOP MOTORE (disabilitazione asse)
602h	6	40 08 10 00 00 00	Wait	0	Device name
602h	8	28 17 10 00 40 0F 00 00	Wait	0	4 secondi intervallo heartbeat
602h	8	28 17 10 00 00 07 00 00	Wait	0	2 secondi intervallo heartbeat

Figure 6.31: PCAN-View Transmit section

Here are present all the messages not directly transmitted in the CANopen network but these messages are created by the user and then transmitted in the network by clicking on them.

Figure 6.32: Example of a creation of a message in PCAN-View

This function is very useful specifically for debugging puporse and controllo over the network. Beacuse these messages can be sent anytime during network usage.

In conclusion PCAN-View is a very powerful software not only for scan the network but also to intervein on it in a very simple way.

Chapter 7

Conclusions

The primary objective of this thesis was to implement a back-to-back configuration for two motors, each controlled by its own microcontroller, providing proper control of speed and Field-Oriented Control (FOC) torque respectively. The selection of both the microcontrollers and motors was constrained by the availability of equipment in the Politecnico di Torino's laboratory.

Despite these constraints, the positive results obtained from the test bench validate its effectiveness and potential for further applications. In fact, the developed setup can now serve as a case study for additional experiments, such as generating a magnetic flux map for the motors or integrating the system into educational programs.

The proposed test bench offers several strengths. All the necessary instrumentation is already available in the laboratory, minimizing the need for additional hardware. The STM32F303RE microcontroller is widely used in course curricula, while the introduction of the Micro Digital One requires only minor integration, some of which has already been addressed in Chapters 1, 2, 3, and 6 of this thesis.

Moreover, the use of commercially available, industry-relevant components bridges the gap between academic study and real-world engineering practice. The inclusion of the CANopen communication protocol, a widely adopted standard in industrial automation, further enhances the educational value of the setup. Its correct integration enables flexible communication between the two microcontrollers, making the system not only useful for teaching purposes but also adaptable for future enhancements.

In conclusion, the developed back-to-back motor configuration offers a versatile, accessible, and pedagogically valuable platform that can serve both academic and research needs.

Bibliography

- [1] STMicroelectronics. *STM32F303RE Datasheet*. 2020. URL: <https://www.st.com/resource/en/datasheet/stm32f303re.pdf>.
- [2] STMicroelectronics. *L6398 High Voltage Half-Bridge Gate Driver Datasheet*. 2018. URL: <https://www.st.com/resource/en/datasheet/l6398.pdf>.
- [3] STMicroelectronics. *X-NUCLEO-IHM08M1 Data Brief*. 2020. URL: https://www.st.com/resource/en/data_brief/x-nucleo-ihm08m1.pdf.
- [4] G. Pellegrino. *Power Converters and Electrical Drives*. Course notes. 2018.
- [5] CSS Electronics. *CAN bus the ultimate guide*. 2021. URL: <https://www.csselectronics.com/pages/can-bus-ultimate-guide>.
- [6] Microphase. *Micro digital One Servodrive for Brushless and Brush motors Manuale*. 2017. URL: <https://www.microphase.eu/en/product/micro-digital-one/>.
- [7] Microphase. *Field-Buses-Information*s. 2017. URL: <https://www.microphase.eu/en/product/micro-digital-one/>.
- [8] Microphase. *Micro digital One interface*. 2017. URL: <https://www.microphase.eu/en/product/micro-digital-one/>.
- [9] Microphase. *datasheet s140*. 2017. URL: <https://www.microphase.eu/en/product/s-series-brushless-servomotors/>.
- [10] Microphase. *datasheet s160*. 2017. URL: <https://www.microphase.eu/en/product/s-series-brushless-servomotors/>.