POLITECNICO DI TORINO

Master's Degree in Electronic and Embedded Systems Engineering

MASTER THESIS

Development of an Automated Multifunctional Motor Test Bench

Supervisors:
Prof. Luca Ferraris
Ing. Emir Poskovic

Laboratory Technician:
Ing. Fausto Franchini

Candidate:
Farshad Hassanalizadeh

Academic Year 2025/2026

Acknowledgment

The author would like to express sincere gratitude to Prof. Luca Ferraris and Ing. Emir Poskovic for their supervision, technical guidance, and continuous support during the development of this thesis.

Special thanks are extended to Ing. Fausto Franchini, laboratory technician, for his invaluable support during the laboratory activities, experimental setup, debugging of the motor control platform, and practical assistance in the use of the test bench equipment. His help was fundamental in moving the work from a software-only study to an operational experimental activity.

The author also thanks the members of the laboratory for their availability and for the useful discussions regarding motor control, STM32 Motor Control SDK configuration, and experimental validation.

Abstract

This thesis presents the development, implementation, and experimental validation of an automated multifunctional motor test bench for the control and analysis of a permanent magnet synchronous motor (PMSM). The work was carried out in the context of embedded motor-control experimentation at Politecnico di Torino, with the objective of creating a practical and repeatable platform able to start and stop the motor, acknowledge faults, command speed references, tune control parameters, acquire experimental data, and support future laboratory automation.

The platform is based on an STM32F302 control board, an ST three-phase inverter board, the STM32 Motor Control Software Development Kit (MCSDK), ST Motor Control Workbench, ST Motor Profiler, PuTTY serial communication, and a LabVIEW-based measurement environment. Following the final organization requested during the review, the thesis first introduces motor types, drives, PMSM modelling, and motor-control methods. It then describes the experimental hardware, the motor-brake test bench, the inverter, current sensing, pin assignment, project-generation workflow, laboratory tools, and the reasons for selecting field oriented control (FOC). FOC was selected because it enables decoupled flux and torque regulation, smooth torque production, and direct compatibility with the STM32 MCSDK workflow.

The main embedded contribution is the customization of the MCSDK-generated firmware and the implementation of a UART command interface through USART2. The interface was tested with PuTTY and supports fault acknowledgement, motor start and stop, numerical speed-reference commands, speed storage before motor start, motor-state/fault-code reporting, and PID-related debugging. The firmware structure uses interrupt-based command reception while executing the motor-control actions in the main loop, improving robustness and avoiding complex operations inside the interrupt callback.

A central result of the work is the solution of the speed-synchronization problem between the value entered by the user in the serial terminal and the real mechanical speed of the motor. Direct terminal values did not initially correspond to mechanical rpm because the custom firmware call required MCSDK internal speed-unit scaling. The motor behavior was identified experimentally as approximately $\omega_{real} \approx 6\,\omega_{input}$. The implemented compensation uses MC command = rpm/6, together with a 3000 rpm software limit. Final speed-validation measurements from 500 rpm to 3000 rpm showed stable operation without

faults and errors between 1.00% and 0.13%.

The thesis also reports PI/PID tuning experiments performed with the brake attached to the motor. The characterized motor-brake system has significant inertia and friction, which explains the overshoot, oscillation, and long settling times observed during aggressive tuning. The final investigated configuration used longer speed ramps and modified speed-loop and current-loop gains in order to improve stability. Excessive integral action was shown to increase oscillation and, in some cases, to produce speed-feedback fault code 0x0020.

As a final experimental validation activity, an efficiency map of the tested motor-drive system was generated using the LabVIEW acquisition environment. The map was obtained from 45 experimental operating points measured at nominal speeds of 500, 1000, 1500, 2000, 2500, and 3000 rpm while progressively changing the braking load. For each point, the efficiency was calculated as $\eta = P_{out}/P_{in}$ from the measured mechanical output power and electrical input power. The highest measured efficiency was approximately 0.603, corresponding to 60.3%, near 2000 rpm and 27.6 mNm. The resulting map confirms that the developed test bench can provide not only speed-control validation and PID-tuning support, but also a meaningful experimental performance characterization of the complete PMSM drive. The low-speed/high-torque part of the map was limited to stable and repeatable points because the brake produced difficult stabilization conditions in that region.

The final outcome is a documented and experimentally validated motor-control test bench integrating hardware setup, PMSM theory, FOC implementation, UART communication, fault management, speed synchronization, PID tuning, LabVIEW observation, and efficiency-map generation. The platform provides a solid foundation for future automated testing, calibrated data logging, comparison of motor topologies, and advanced PMSM-drive characterization.

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

V_d, V_q	Stator voltage components in the rotating dq reference frame
i_d, i_q	Stator current components in the rotating dq reference frame
R_s	Stator phase resistance
L_d, L_q	Direct-axis and quadrature-axis inductances
λ_m	Permanent magnet flux linkage
λ_d, λ_q	Flux linkages in dq reference frame
ω_e	Electrical angular speed
ω_m	Mechanical angular speed
p	Number of pole pairs
T_e	Electromagnetic torque
T_L	Load torque
J	Equivalent moment of inertia
B	Viscous friction coefficient
PMSM	Permanent Magnet Synchronous Motor
SM-PMSM	Surface Mounted Permanent Magnet Synchronous Motor
IPMSM	Interior Permanent Magnet Synchronous Motor
FOC	Field Oriented Control
SVPWM	Space Vector Pulse Width Modulation
PWM	Pulse Width Modulation
MCSDK	Motor Control Software Development Kit
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
UART	Universal Asynchronous Receiver Transmitter
PLL	Phase Locked Loop
STO	State Observer
BEMF	Back Electromotive Force
PI/PID	Proportional-Integral / Proportional-Integral-Derivative controller

GUI	Graphical User Interface

Chapter 1

Introduction

1.1 Motivation

Electric motors are a central component of modern energy conversion systems. They are used in industrial automation, robotics, automotive systems, aerospace actuators, household appliances, ventilation systems, pumps, and electric propulsion. In many of these applications, the motor is not used alone: it is part of a complete electromechanical drive composed of a power converter, a control unit, sensing elements, communication interfaces, protection functions, and software layers. For this reason, the development of a motor test bench is not only a mechanical or electrical task, but also an embedded systems task.

Permanent magnet synchronous motors are particularly attractive because they provide high efficiency, high power density, compact construction, and good dynamic behavior. These advantages are obtained thanks to the permanent magnets mounted in or on the rotor, which eliminate the need for rotor excitation currents. The absence of rotor copper losses makes PMSMs suitable for high-performance drives, but it also makes their control more complex than that of simpler brushed DC machines. High-performance PMSM drives require the synchronization of inverter currents with the rotor magnetic field; therefore, accurate control algorithms and reliable speed or position feedback are needed.

The objective of this thesis is to develop an automated multifunctional motor test bench using an STM32 microcontroller and the ST Motor Control Software Development Kit. The final system is intended to start and stop the motor, acknowledge faults, command speed ramps, tune control parameters, and collect experimental results. The work is also intended as a foundation for future integration with LabVIEW, as suggested during the development activity.

1.2 Thesis Context

The work was carried out in the context of motor control and experimental testing activities at Politecnico di Torino. The present thesis follows a hardware-first engineering workflow: it starts from the physical system, introduces the motor-control theory required to understand the drive, explains the selected control method, and finally presents the firmware implementation and experimental validation.

Unlike a thesis focused on the electromagnetic design of a new motor, the present work focuses on the test bench and embedded motor-control implementation. Therefore, the central contributions are firmware integration, serial command interpretation, real-time speed control, PID tuning, fault handling, and practical validation of the STM32 MCSDK environment.

1.3 Objectives

The main objective is the development of an automated motor test bench capable of controlling a PMSM using STM32 MCSDK and field oriented control. The specific objectives are:

- describe the hardware configuration of the motor, inverter, STM32 control board, and laboratory instruments;
- study the motor control methods available for brushless and PMSM drives;
- justify the choice of FOC for the implemented system;
- configure and use the STM32 Motor Control SDK and ST Motor Control Workbench;
- implement a UART command interface for Start, Stop, Ack, speed reference, and PID tuning;
- solve the speed-reference scaling mismatch between user command and MCSDK internal speed units;
- analyze the influence of PID gains on speed response, overshoot, oscillation, and settling time;
- document the experimental procedure and provide a structured test bench description suitable for future laboratory use.

1.4 Thesis Flow

Following the recommendation received during the work, the thesis is organized according to the following flow:

1. hardware description and motor specification;
2. mathematical theory and physical principles;
3. review of motor control methods;
4. justification of the selected method;
5. implementation on STM32 MCSDK;
6. experiments and results.

This order allows the reader to first understand the physical system and then progressively move toward the control theory, software implementation, and experimental validation.

Chapter 2

Motor Theory, Modeling, and Control Methods

2.1 Permanent Magnet Motor Theory

2.1.1 Electric Machine Families

Electrical machines can be classified into DC machines, induction machines, synchronous machines, switched reluctance machines, and permanent magnet brushless machines. In classical brushed DC motors, commutation is mechanical, while in brushless motors the commutation is performed electronically through an inverter. Modern drive systems have progressively moved toward brushless permanent magnet machines because of their higher efficiency, reduced maintenance, and compatibility with digital control.

2.1.2 Permanent Magnet Synchronous Motors

A PMSM is a synchronous motor in which the rotor magnetic field is produced by permanent magnets rather than by a rotor winding. The rotor rotates synchronously with the stator rotating magnetic field. The mechanical speed ω_m and electrical speed ω_e are related by the number of pole pairs p:

$$\omega_e = p\omega_m. \tag{2.1}$$

This relationship is important in embedded control because the control algorithm often works with electrical angle and electrical speed, while the user usually commands mechanical speed in revolutions per minute. A wrong interpretation of the speed unit can therefore create a large mismatch between the user command and the real motor behavior.

2.1.3 Surface and Interior Permanent Magnet Motors

Two common PMSM rotor structures are surface-mounted PMSM and interior PMSM. In an SM-PMSM, magnets are mounted on the rotor surface. This configuration is often close to magnetic isotropy, meaning that L_d and L_q are approximately equal. In an IPMSM, magnets are embedded inside the rotor. This structure introduces saliency and can provide reluctance torque in addition to magnet torque.

The experimental configuration in this work was set as SM-PMSM in the Motor Profiler. For an SM-PMSM, a common control objective is to keep i_d near zero and regulate torque primarily through i_q. This simplifies the control law and is consistent with standard FOC operation for surface magnet machines.

2.1.4 Axial-Flux and Radial-Flux Motors

The motor shown in the laboratory has a geometry visually similar to compact axial-flux machines investigated in previous Politecnico activities. In the literature, axial-flux permanent magnet machines are considered attractive for compact applications because they can provide high torque density and efficient use of volume. The review by Shao et al. states that axial-flux permanent magnet machines are attractive for propulsion applications due to high power density, high efficiency, and compact structure. Cavagnino et al. also show that when axial length is very short and pole number is high, axial-flux motors can be an attractive alternative to radial-flux solutions.

For the purpose of this thesis, the exact electromagnetic design of the motor is not redesigned. However, the theoretical distinction is useful because the mechanical inertia, geometry, and magnetic structure influence test-bench behavior, speed estimation, and controller tuning.

2.1.5 Magnetic Fundamentals

The basic magnetic relationship in vacuum is:

$$\mathbf{B} = \mu_0 \mathbf{H} \tag{2.2}$$

where $\mathbf{B}$ is magnetic flux density, $\mathbf{H}$ is magnetic field intensity, and μ_0 is the magnetic permeability of vacuum. In magnetic materials, magnetization must be considered:

$$\mathbf{B} = \mu_0(\mathbf{H} + \mathbf{M}). \tag{2.3}$$

The performance of permanent magnet machines depends strongly on the magnet properties, air-gap flux density, stator material, winding arrangement, and thermal behavior. Previous Politecnico theses on axial-flux machines include detailed chapters on magnetic

materials, SMC, plastomagnets, and experimental characterization. In the present thesis these aspects are not the main contribution, but they provide the background for understanding why PMSMs require accurate control and test procedures.

2.2 Mathematical Modeling of PMSM

2.2.1 Three-Phase Stator Model

The stator of a three-phase PMSM is supplied by three phase voltages v_a, v_b, and v_c. The corresponding currents are i_a, i_b, and i_c. For a balanced three-phase system without neutral connection:

$$i_a + i_b + i_c = 0. \tag{2.4}$$

In the phase reference frame, the equations are coupled and depend on rotor position. This makes direct control difficult. Field oriented control solves this problem by transforming the currents and voltages into a rotating coordinate system aligned with the rotor flux.

2.2.2 Clarke Transformation

The Clarke transformation converts the three-phase quantities into the stationary $\alpha\beta$ reference frame:

$$i_\alpha = i_a \tag{2.5}$$

$$i_\beta = \frac{1}{\sqrt{3}}(i_a + 2i_b). \tag{2.6}$$

The transformation reduces the three-phase system to two orthogonal components in a stationary frame. The same transformation can be applied to voltages.

2.2.3 Park Transformation

The Park transformation rotates the stationary $\alpha\beta$ frame into the rotor-oriented dq frame:

$$i_d = i_\alpha \cos\theta_e + i_\beta \sin\theta_e \tag{2.7}$$

$$i_q = -i_\alpha \sin\theta_e + i_\beta \cos\theta_e. \tag{2.8}$$

The d axis is aligned with the rotor magnetic field, while the q axis is orthogonal to it.

In this frame, DC quantities can be controlled using PI regulators instead of controlling sinusoidal phase currents directly.

2.2.4 Voltage Equations in the dq Frame

The PMSM voltage equations in the synchronous rotating frame are:

$$V_d = R_s i_d + L_d \frac{di_d}{dt} - \omega_e L_q i_q \tag{2.9}$$

$$V_q = R_s i_q + L_q \frac{di_q}{dt} + \omega_e L_d i_d + \omega_e \lambda_m. \tag{2.10}$$

These equations show that the d and q axes are coupled by speed-dependent terms. Current control must compensate or tolerate this coupling. For an SM-PMSM, $L_d \approx L_q$, and the torque is mainly proportional to i_q.

2.2.5 Torque Equation

The electromagnetic torque of a PMSM is:

$$T_e = \frac{3}{2} p[\lambda_m i_q + (L_d - L_q) i_d i_q]. \tag{2.11}$$

For a surface-mounted PMSM, if $L_d \approx L_q$, the reluctance torque term becomes negligible, and the equation is simplified to:

$$T_e = \frac{3}{2} p \lambda_m i_q. \tag{2.12}$$

This relation explains why the i_q current is called the torque-producing current.

2.2.6 Mechanical Equation

The mechanical dynamics of the motor and load are described by:

$$J \frac{d\omega_m}{dt} = T_e - T_L - B\omega_m \tag{2.13}$$

where J is the total inertia, T_L is the load torque, and B is the viscous friction coefficient. The PID tuning experiment showed that the test bench behaves like a high-inertia system; therefore, the mechanical equation is essential for interpreting the long settling time and overshoot observed during tests.

2.2.7 Electrical and Mechanical Power

The mechanical output power is:

$$P_m = T_e \omega_m. \tag{2.14}$$

The electrical input power can be approximated from the DC bus by:

$$P_{in} = V_{dc} I_{dc}. \tag{2.15}$$

The efficiency can be estimated as:

$$\eta = \frac{P_m}{P_{in}} \times 100. \tag{2.16}$$

In this thesis, the available monitoring data were sufficient for speed, torque-reference, and controller-response analysis. A calibrated shaft-torque measurement was not part of the final campaign, therefore efficiency is discussed theoretically rather than used as a final validation metric.

2.3 Motor Control Methods

2.3.1 Need for Motor Control

A PMSM cannot be connected directly to a DC source. It requires a power inverter and a control algorithm able to generate the appropriate stator voltage vectors. The control method determines the quality of torque production, dynamic response, efficiency, acoustic noise, current ripple, and robustness to load changes.

2.3.2 Open-Loop Voltage-Frequency Control

Voltage-frequency control is one of the simplest control methods. It applies a voltage magnitude proportional to frequency and is often used in induction motor drives. Its main advantage is simplicity. However, it does not directly control torque and is not suitable for high-performance PMSM operation, especially during fast transients or at low speed.

2.3.3 Six-Step or Trapezoidal Control

Six-step control is commonly used in brushless DC motors. The inverter commutates every 60 electrical degrees, and typically two phases conduct at the same time. This method is simple and requires less computational effort than FOC. However, it generates higher torque ripple and is not ideal for applications requiring smooth torque and accurate speed regulation.

2.3.4 Sinusoidal Control

Sinusoidal control applies sinusoidal phase currents synchronized with rotor position. Compared with six-step control, it reduces torque ripple and acoustic noise. However, without current transformation and decoupled control, it does not achieve the full dynamic performance of FOC.

2.3.5 Field Oriented Control

Field oriented control transforms the motor currents into the rotating *dq* reference frame. The torque and flux components are then controlled independently. This method is computationally more demanding but provides high dynamic performance, smooth torque, and efficient operation. The STM32 MCSDK implements FOC for PMSM drives and provides current sampling, speed estimation, SVPWM, PID regulation, and communication interfaces.

2.3.6 Direct Torque Control

Direct torque control directly controls torque and flux through voltage vector selection. It can provide fast torque response but may lead to higher torque ripple and requires careful implementation. For this thesis, DTC was not selected because the MCSDK environment and the available ST boards are already optimized for FOC.

2.3.7 Sensored and Sensorless Control

A PMSM drive can use sensors such as encoders or Hall sensors to measure rotor position. Alternatively, sensorless algorithms estimate rotor position from electrical measurements. The MCSDK supports sensorless operation based on BEMF reconstruction, State Observer, PLL, CORDIC, and High Frequency Injection depending on the microcontroller and motor type. The implemented setup uses sensorless control with observer/PLL, which reduces hardware complexity but is sensitive to low-speed operation and parameter mismatch.

2.3.8 Justification of the Selected Method

FOC was selected because it is the standard high-performance control method supported by the STM32 MCSDK. It enables independent torque and flux regulation, supports speed and torque modes, allows integration with the Motor Control Workbench, and provides real-time tuning of PID regulators. The availability of the MCSDK API also made it possible to implement user commands through UART without developing the complete motor-control algorithm from zero.

2.4 Field Oriented Control and SVPWM

2.4.1 FOC Control Structure

The implemented FOC system uses a cascade structure. The outer loop controls speed and generates a torque current reference i_q^*. The inner current loops control i_q and i_d by generating voltage references v_q^* and v_d^*. The voltage references are transformed back to the stationary frame and applied to the inverter using SVPWM.

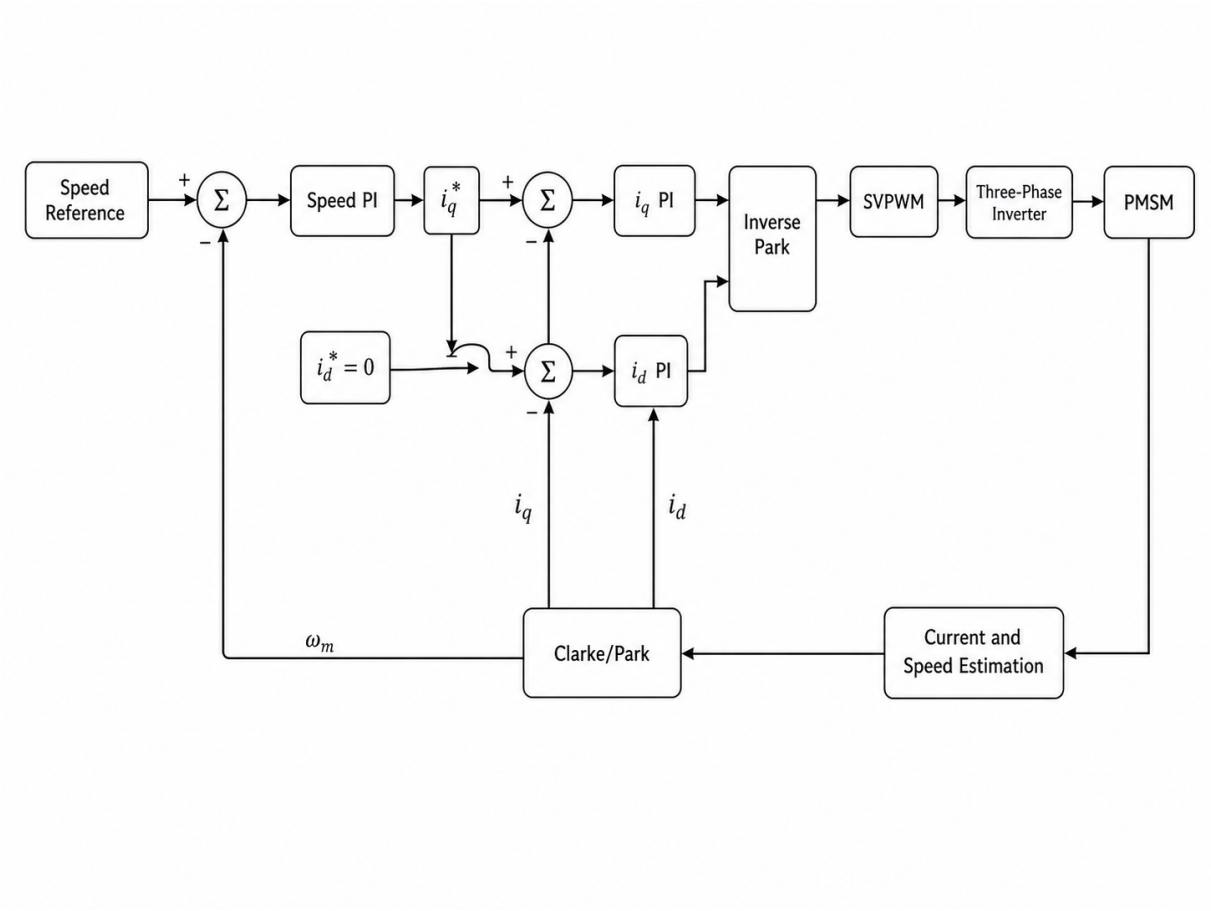


Figure 2.1: FOC cascade control structure used in the motor test bench. The updated layout avoids overlap in the lower feedback paths and improves readability for the speed, current, transformation, and estimation blocks.

2.4.2 Current References

For an SM-PMSM, the default operating strategy is:

$$i_d^* = 0. \tag{2.17}$$

This means that no additional flux-producing current is intentionally applied. The torque is controlled through i_q:

$$i_q^* = f(\omega_m^* - \omega_m). \tag{2.18}$$

The function f is implemented by the speed PI regulator.

2.4.3 PI Regulators

The digital PI controller used in the current and speed loops can be described as:

$$u(k) = K_p e(k) + K_i \sum_{n=0}^{k} e(n) T_s \tag{2.19}$$

where $e(k)$ is the control error and T_s is the sampling period. If K_i is too high, the accumulated error can cause integral wind-up. This was observed in the PID tuning experiment, where the motor continued accelerating after reaching the target speed and produced large overshoot.

2.4.4 Space Vector PWM

SVPWM generates the inverter switching signals by selecting voltage vectors in the complex plane. Compared with sinusoidal PWM, SVPWM provides better DC bus utilization and is widely used in FOC drives. In the MCSDK, SVPWM is integrated in the PWM generation module and works together with the current-sampling strategy. The user manual lists SVPWM as one of the core motor-control library features, with adjustable PWM frequency and centered PWM pattern.

2.4.5 Circle Limitation

The requested voltage vector must remain inside the inverter voltage capability. The MCSDK implements a circle limitation component to prevent the requested v_d and v_q values from exceeding the maximum available voltage module. This is especially important during transients, high speed, or aggressive PID tuning.

Chapter 3

Hardware Description and Test Bench

3.1 Overview of the Test Bench

The test bench is composed of an embedded control unit, a three-phase inverter, a PMSM motor, a power supply, serial communication tools, and monitoring software. The main objective of the hardware system is to provide a safe and repeatable environment in which control algorithms can be tested without modifying the power electronics hardware at every iteration.

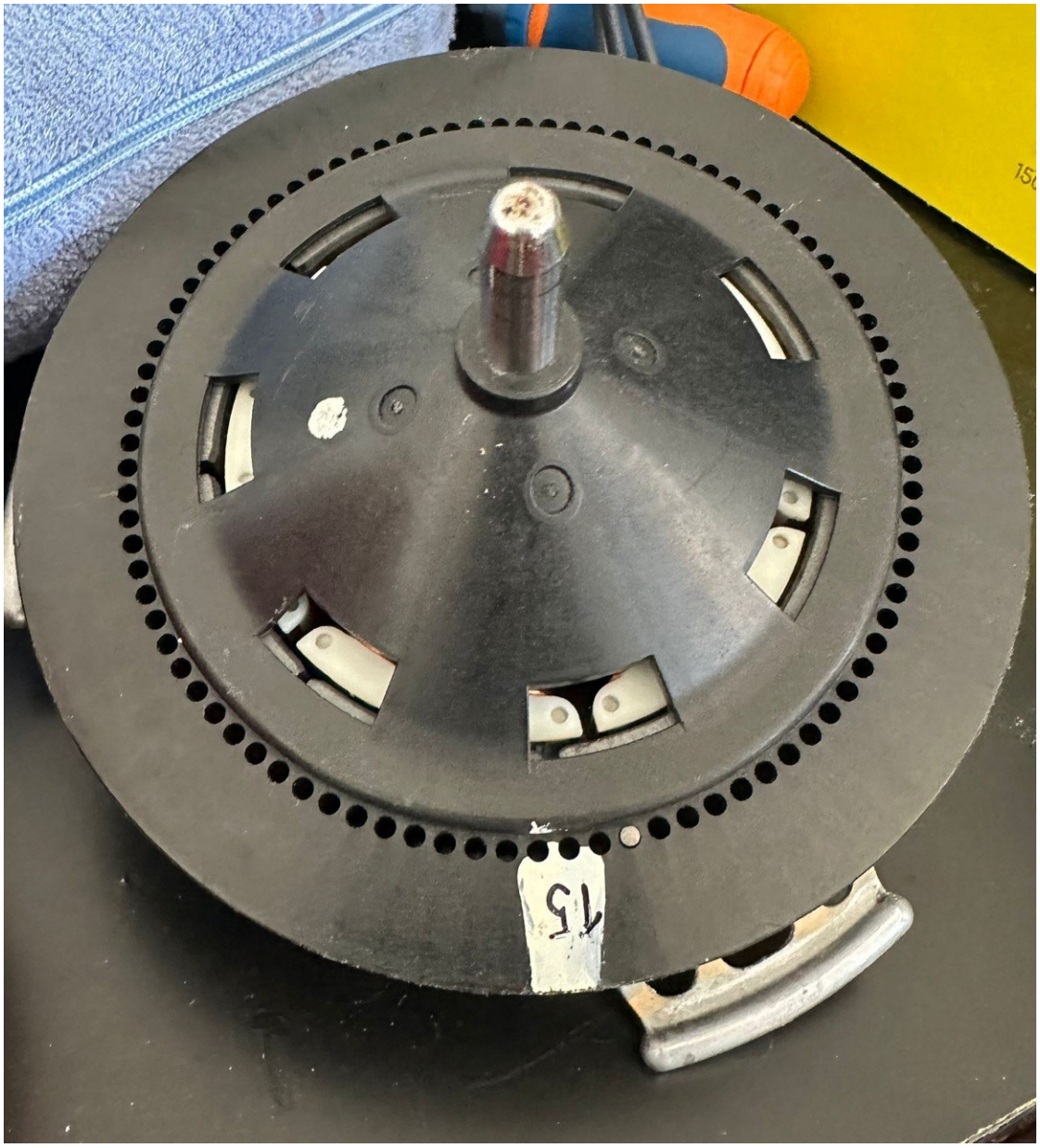


Figure 3.1: Motor under test used during the experimental activity. The image shows the mechanical structure and the rotating shaft used for the test bench.

The motor shown in Fig. 3.1 was connected to the ST development platform. The motor was controlled in sensorless mode using the configuration generated by the Motor Control Workbench. The mechanical system has significant inertia, especially when coupled to the brake or measurement apparatus; this feature strongly influenced the PID tuning results.

3.2 Laboratory Measurement Chain

The complete laboratory setup included the STM32 control board, the ST inverter, the motor under test, a mechanical brake/load unit, a speed/torque measurement chain, a DC power supply, a PC running ST tools, PuTTY, and LabVIEW. The brake and

measurement device were used to observe the mechanical behavior of the motor while the embedded controller regulated the electrical drive. The speed could be observed both through the digital acquisition chain connected to the PC and through a manual measurement instrument. This combination was useful because it allowed the firmware behavior, the mechanical output, and the LabVIEW curves to be compared during the PID tuning campaign.

Figure 3.2: Complete laboratory test bench including PC interface, ST control hardware, motor/load assembly, power supply, and measurement instruments.

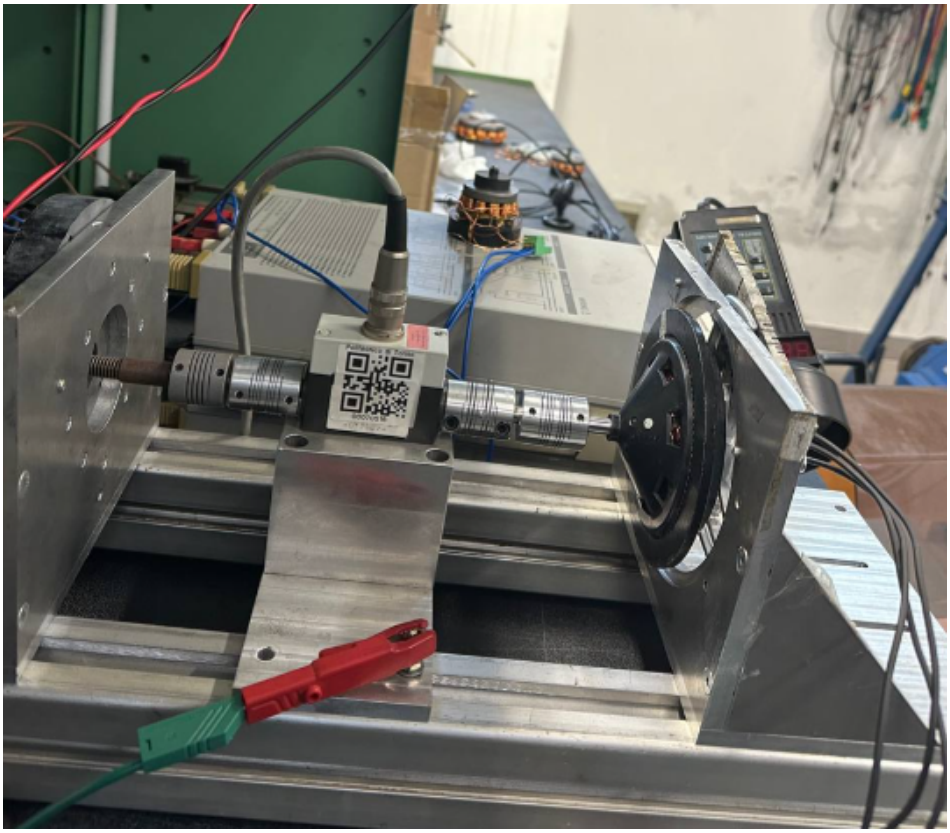


Figure 3.3: Mechanical coupling between the motor, brake/load device, and speed-measurement chain used during the experimental tests.

Figure 3.4: Manual speed-measurement instrument used as an additional reference during the laboratory activity.

3.3 Laboratory Motor Families

During the laboratory activity, different electric motor structures and disassembled machines were inspected. This was useful to connect the theoretical classification of machines with the real physical construction of stators, rotors, windings, shafts, laminations, and permanent-magnet parts. The observation confirmed that motor-control behavior depends not only on the controller but also on the electromagnetic structure, mechanical inertia, winding distribution, and type of rotor.

Figure 3.5: Different motor parts and motor families observed in the laboratory. The photo was used as practical support for comparing machine structures and understanding the difference between radial and axial configurations.

3.4 Extended Literature Review on PMSM Drives

3.4.1 Evolution of Electric Drives

Electric drives have evolved from simple electromechanical systems into integrated power-electronic and embedded-control platforms. The replacement of brushed DC drives by induction and permanent-magnet brushless drives is motivated by efficiency, reliability, reduced maintenance, and compatibility with variable-speed operation. Modern drives combine the motor, inverter, sensing chain, embedded microcontroller, and communication interface. In this thesis the motor test bench therefore is analyzed as an integrated system rather than as a motor alone.

The literature on electric machines emphasizes that machine performance is increasingly linked to progress in permanent magnet materials, power electronics, embedded processors, and control strategies. Rare-earth permanent magnets, fast switching devices, and digital controllers have made compact high-performance drives practical. For this reason, a thesis on an automated motor test bench naturally lies at the intersection of electrical machines, power electronics, embedded software, and laboratory instrumentation.

3.4.2 Permanent-Magnet Motors in Modern Applications

Permanent-magnet motors are widely used in servo drives, robotics, electric vehicles, pumps, compressors, drones, and auxiliary drives. Their advantages include high efficiency, high torque density, and good dynamic response. These advantages are especially important when the available volume is limited or when rapid acceleration is required. A PMSM normally requires electronic commutation through an inverter; therefore, the drive performance depends strongly on the control algorithm.

From a test-bench perspective, permanent-magnet machines are attractive because they allow repeatable experimental studies of speed regulation, current control, sensorless estimation, fault management, and parameter tuning. At the same time, they introduce practical challenges. A PMSM can generate back-electromotive force when rotating, requires a correctly phased inverter supply, and may become unstable if the speed estimator or current feedback is not correctly configured.

3.4.3 Surface-Mounted and Interior Permanent-Magnet Machines

PMSMs can be divided into surface-mounted and interior permanent-magnet machines. A surface-mounted PMSM has magnets fixed on the rotor surface. Its direct-axis and quadrature-axis inductances are often close to each other, so the torque is mainly produced by the magnet flux and the quadrature current. An interior PMSM has magnets embedded in the rotor. This produces saliency, meaning that L_d and L_q are different, and part of the torque can be produced by reluctance effects.

The Motor Profiler configuration used in this thesis selected SM-PMSM. This choice is consistent with the control strategy used later: $i_d^* = 0$ in the base-speed range and i_q is used as the torque-producing current. For a future extension of the test bench, an IPMSM could be used to study maximum torque per ampere (MTPA), field weakening, and saliency-based sensorless control.

3.4.4 Axial-Flux and Radial-Flux Machine Background

The broader thesis material shared for this work includes several documents on axial-flux permanent-magnet machines and comparisons between axial-flux and radial-flux structures. Axial-flux permanent-magnet machines are often associated with high torque density, high efficiency, and compact axial length, which make them attractive for electric propulsion and direct-drive applications. Review literature highlights that AFPM machines are especially interesting when space and weight are critical, including automotive, aerospace, marine, and industrial applications [5].

The comparison by Cavagnino, Lazzari, Profumo, and Tenconi shows that axial-flux motors can be attractive for short axial length and high pole number. This is important for the present test bench because compact motors with high inertia or special rotor structure may behave differently from conventional small radial-flux motors during start-up and speed regulation [4].

3.4.5 Relevance to the Present Test Bench

Although the present thesis does not redesign the motor electromagnetically, the background on motor topologies is still relevant. The test bench must be able to handle motors with different inertia, speed limits, current limits, and sensorless estimation requirements. This is why the thesis starts from hardware and motor specification before moving to control. The controller cannot be treated independently of the machine: the mechanical inertia influenced the PID tuning, the sensorless estimator influenced start-up and fault behavior, and the current-sensing topology influenced the validity of FOC operation.

3.5 Main Hardware Components

The hardware configuration used in the thesis includes:

- **Control board:** ST NUCLEO-F302R8, based on an STM32F302 microcontroller.
- **Power stage:** X-NUCLEO-IHM08M1 3Sh, a three-shunt motor control expansion board based on ST motor-driver technology.
- **Motor:** three-phase PMSM configured as an SM-PMSM in the Motor Profiler.
- **Communication interface:** USART2 configured at 115200 baud.
- **User terminal:** PuTTY connected on COM22 during the first communication tests.
- **Software tools:** ST Motor Control Workbench, ST Motor Profiler, Atollic TrueSTUDIO for STM32, and later LabVIEW as planned future interface.

3.6 Motor Profiler Configuration

The ST Motor Profiler was used to configure the control board, power board, motor type, operating limits, and the first motor model. The selected hardware was the NUCLEO-F302R8 control board together with the X-NUCLEO-IHM08M1 3Sh power board. The selected motor type was SM-PMSM, with 4 pole pairs, a 12 V DC bus configuration, 6 A peak current limit, and an initial maximum speed setting of 4000 rpm.

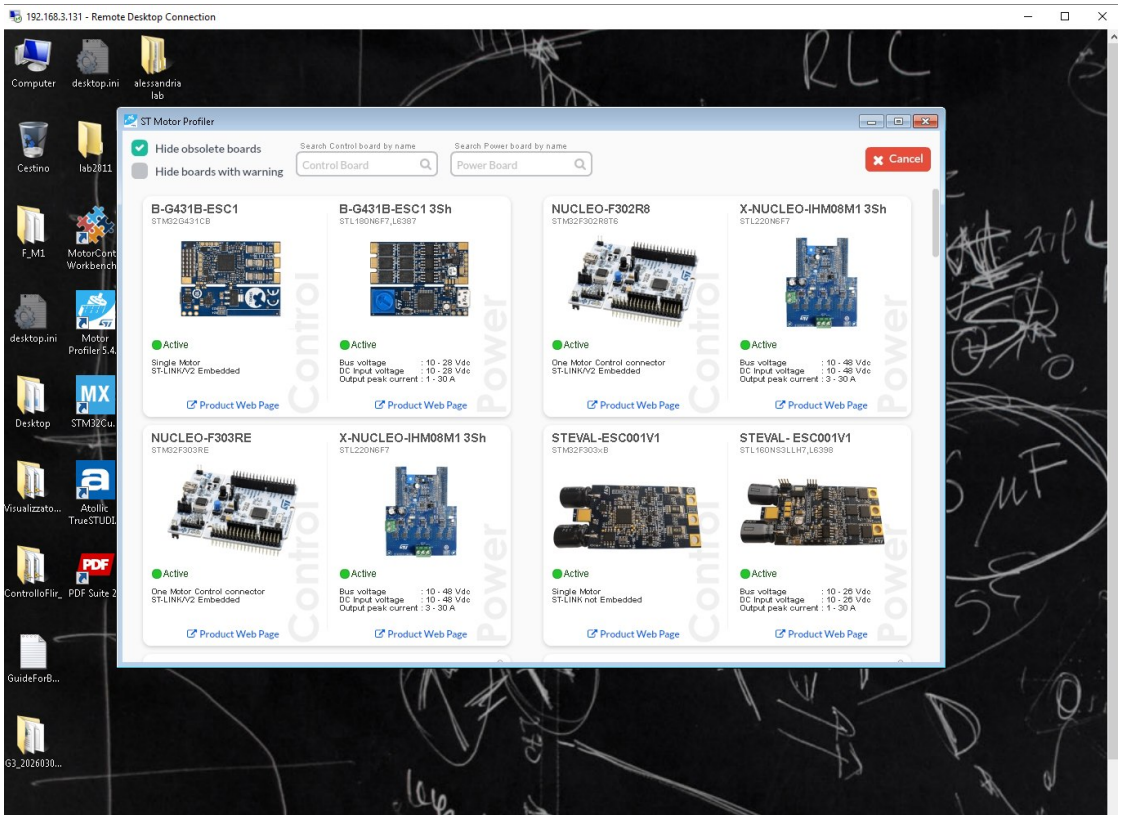


Figure 3.6: ST Motor Profiler board-selection window showing the NUCLEO-F302R8 control board and the X-NUCLEO-IHM08M1 3Sh power board used for the test bench.

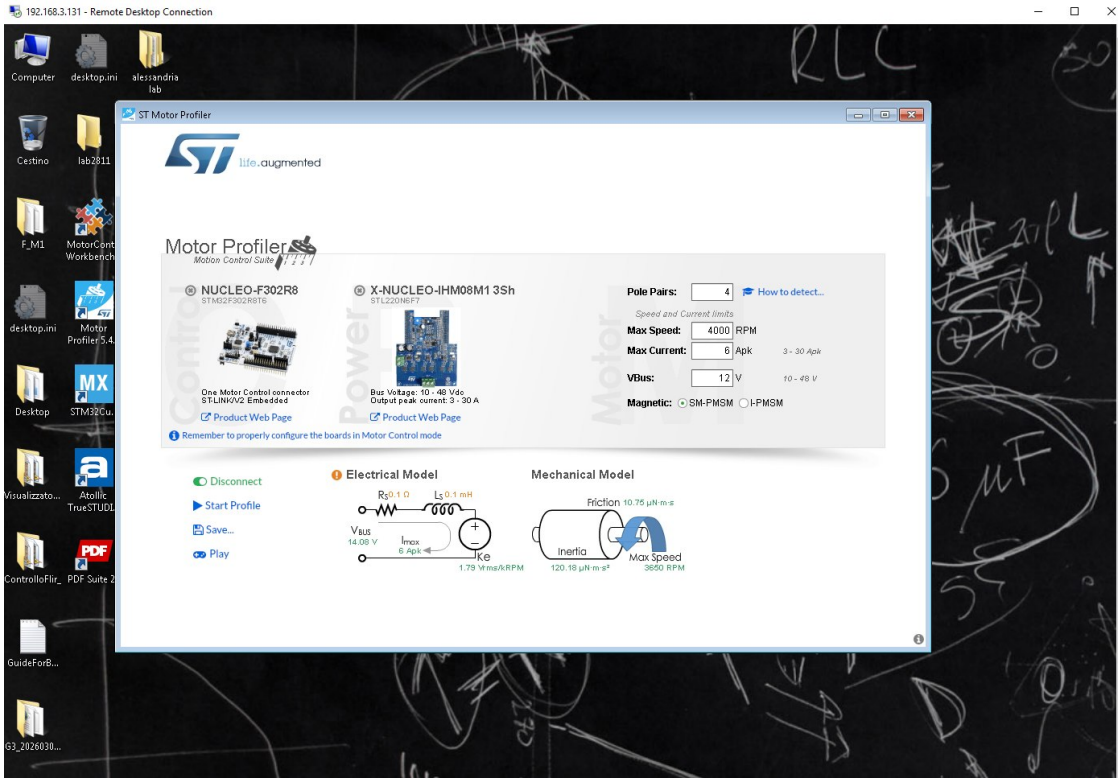


Figure 3.7: Motor Profiler screen after characterization. The identified electrical model shows approximately $R_s = 0.1\ \Omega$, $L_s = 0.1$ mH, $K_e = 1.79\ \mathrm{V_{rms}/kRPM}$, while the mechanical model with brake attached shows $J = 120.18\ \mu\mathrm{Nms}^2$ and $B = 10.75\ \mu\mathrm{Nms}$.

3.7 Hardware Parameter Table

Table 3.1 summarizes the hardware and profiler parameters used in the final project configuration. The values reported here come from the Motor Profiler and Workbench screens used during the laboratory activity.

Parameter	Final observed value	Engineering meaning
Control board	NUCLEO-F302R8	STM32F302-based ST development board used for the MCSDK project.
Power board	X-NUCLEO-IHM08M1 3Sh	Three-shunt current-sensing inverter expansion board.
Motor type	SM-PMSM / PMSM	Surface-mounted permanent magnet synchronous motor selected in Motor Profiler.
Pole pairs	4	Used for conversion between mechanical and electrical speed.
Configured DC bus	12 V	Value entered during initial profiler setup.
Observed DC bus during profile	approximately 14.08 V	Value shown in the Motor Profiler electrical model screen.
Nominal voltage imported	14.1 V	Value imported by Workbench after profiling.
Maximum current	6 A peak	Current limit shown in the profiler and imported project.
Maximum speed	approximately 3650 rpm	Profiler mechanical model showed 3650 rpm; Workbench imported 3648 rpm.
Stator resistance R_s	0.1 Ω	Electrical model parameter used for current-control behavior.
Stator inductance L_s	0.1 mH	Electrical model parameter; firmware comparison showed approximately 0.097 mH.
BEMF constant K_e	1.79 V_{rms}/kRPM	Voltage constant identified by Motor Profiler.
Inertia J	120.18 μNms^2	Mechanical model of the motor plus brake/load system.
Friction B	10.75 μNms	Viscous friction model of the motor plus brake/load system.
PWM frequency	30000 Hz	Workbench project-generation information.
FOC rate	1 PWM period	Workbench imported configuration value.
UART port	COM22	Serial port used for PuTTY tests.
UART baud rate	115200 bit/s	Terminal and firmware communication speed.

Table 3.1: Final hardware and profiler parameters used in the motor test-bench development.

3.8 Workbench Project Generation and Pin Assignment

After the motor profile was obtained, the values were imported into an ST Motor Control Workbench project. The imported profile included 6 A peak nominal current, 14.1

V nominal voltage, 30000 Hz PWM frequency, one-PWM-period FOC rate, and 6000 rad/s cut-off frequency. The generated Workbench project was then configured for the NUCLEO-F302R8 control board, the X-NUCLEO-IHM08M1 3Sh power board, and the surface-mounted PMSM profile.

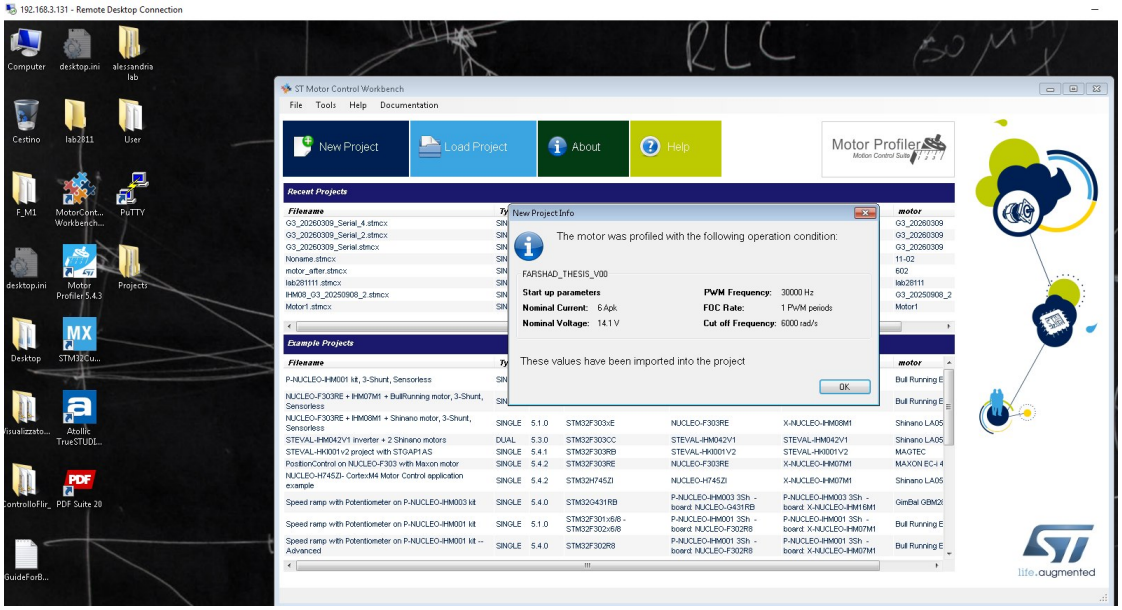


Figure 3.8: Workbench import window showing the operating conditions obtained from the motor profile: 6 A peak nominal current, 14.1 V nominal voltage, 30000 Hz PWM frequency, FOC rate of one PWM period, and 6000 rad/s cut-off frequency.

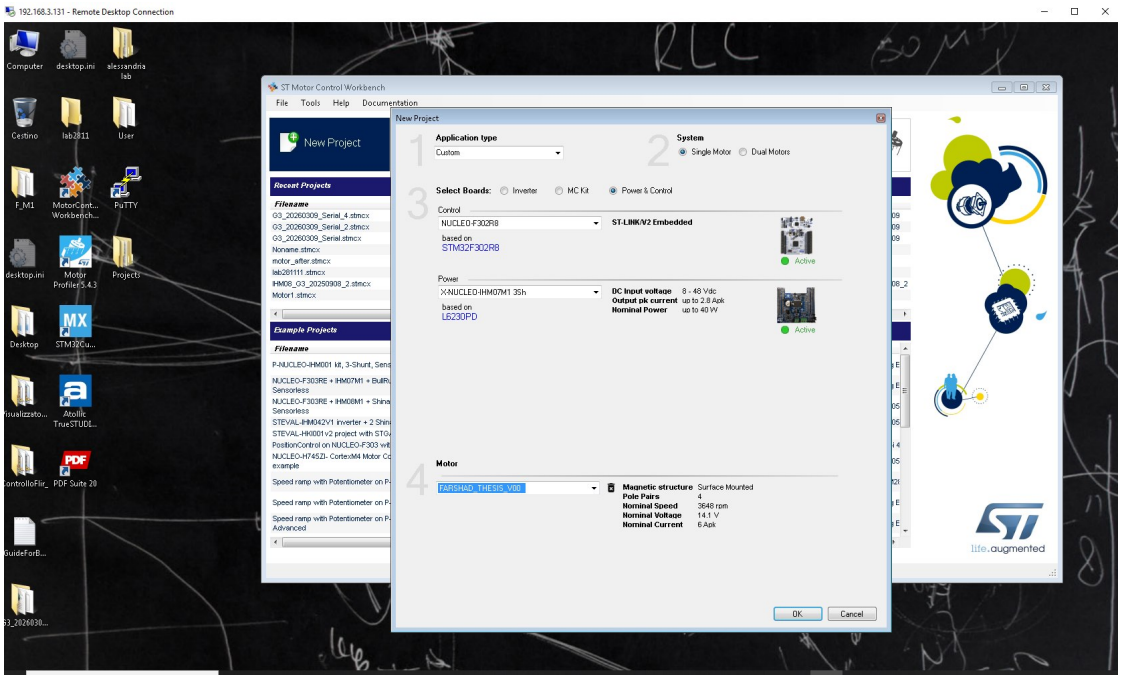


Figure 3.9: Workbench project configuration with NUCLEO-F302R8, X-NUCLEO-IHM08M1 3Sh, and the profiled PMSM motor. The imported motor data show surface-mounted structure, 4 pole pairs, nominal speed close to 3648 rpm, 14.1 V nominal voltage, and 6 A peak nominal current.

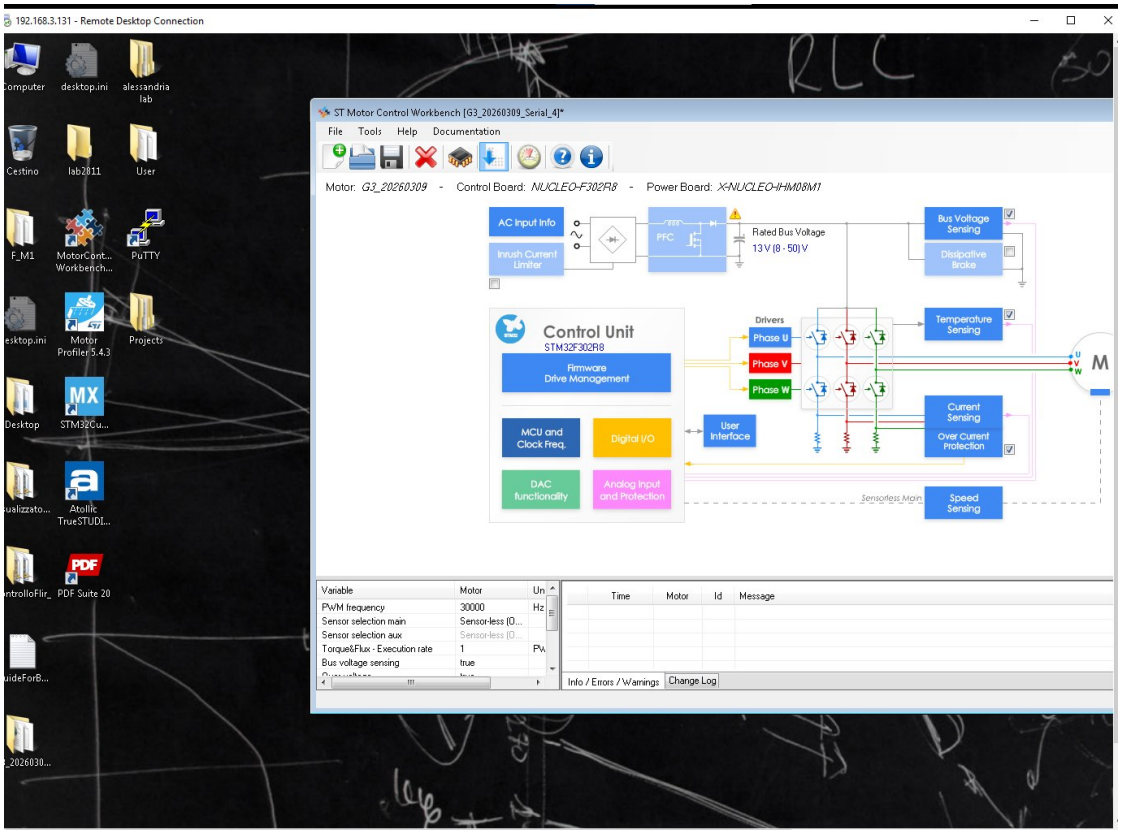


Figure 3.10: Workbench system view showing the relationship between the control unit, three-phase inverter, bus-voltage sensing, temperature sensing, current sensing, overcurrent protection, speed sensing, and motor terminals.

The Workbench-generated project was exported through STM32CubeMX using the HAL driver layer and the ST SW4 STM32 target toolchain. The generated pin assignment confirms the hardware resources used by the test bench: TIM1 for PWM, USART2 for PuTTY communication, ADC1 injected channels for phase-current feedback, ADC1 channel for bus voltage, ADC1 temperature input, and a DAC debug output.

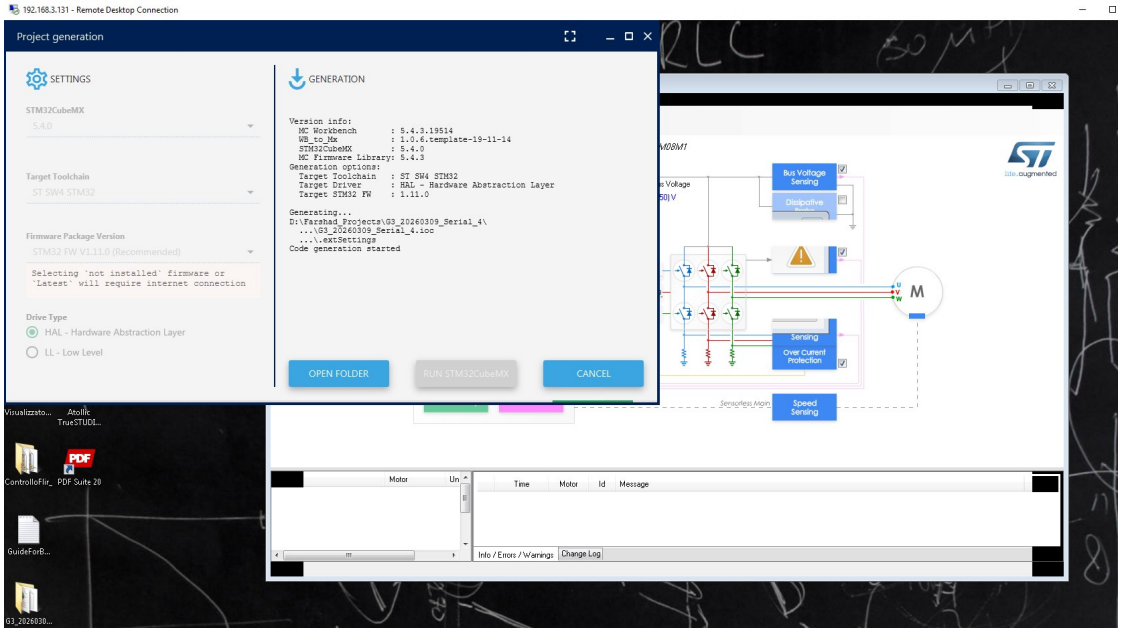


Figure 3.11: Project-generation settings used by Workbench before producing the STM32CubeMX configuration and application code.

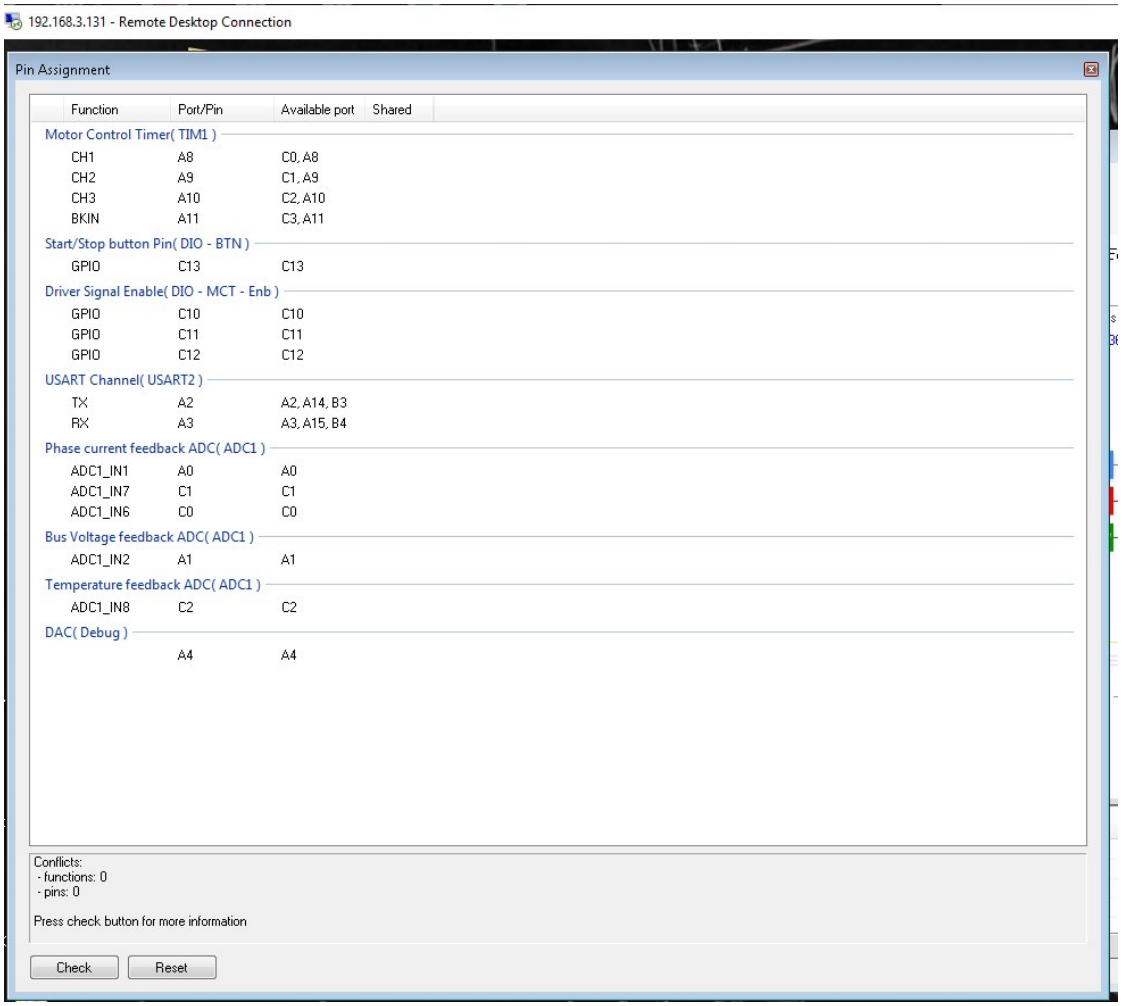


Figure 3.12: Pin-assignment table: TIM1 channels on PA8/PA9/PA10, BKIN on PA11, USART2 TX/RX on PA2/PA3, phase-current feedback on ADC1 channels IN1/IN7/IN6, bus-voltage feedback on ADC1 IN2, temperature feedback on ADC1 IN8, and DAC debug output on PA4.

3.9 Three-Phase Inverter

The inverter converts the DC bus voltage into a three-phase voltage system applied to the motor terminals. In the implemented control system, the inverter is driven by PWM signals generated by the advanced timer of the STM32 microcontroller. The power stage is responsible for switching the DC bus at high frequency and synthesizing the voltage vector required by the FOC algorithm.

A three-phase inverter for PMSM control typically consists of six semiconductor switches arranged in three half-bridges. Each motor phase is connected to the midpoint of one half-bridge. The controller generates complementary PWM signals with dead time to avoid shoot-through events. In the MCSDK project, TIM1 is used as the main PWM timer; it is configured in center-aligned mode, which reduces harmonic distortion and improves current-sampling synchronization.

3.10 Current Sensing

Current sensing is essential for FOC because the control algorithm needs the stator currents to reconstruct i_d and i_q. In the hardware configuration used for this thesis, the power board is a three-shunt inverter. This means that shunt resistors are placed on the low side of the inverter legs, allowing phase currents to be reconstructed through ADC measurements synchronized with the PWM cycle.

The STM32 MCSDK supports different current-sensing topologies, including three-shunt sensing, single-shunt sensing, and isolated current sensors. The user manual reports current-sampling sections for three-shunt topologies with one or two ADC converters, single-shunt topologies, and isolated current sensors. The selected setup uses the three-shunt configuration because it is directly compatible with the X-NUCLEO-IHM08M1 board and provides good observability of the motor phase currents.

3.11 Configuration Derived from `mc_parameters.c`

The generated file `mc_parameters.c` contains hardware-specific parameters used by the MCSDK current-feedback and PWM modules. In this project the relevant structure is `R3_1_ParamsM1`, which indicates a three-shunt, one-ADC current-sensing topology for Motor 1. The configuration uses ADC1 injected conversions synchronized with TIM1 trigger output. The ADC channels visible in the generated configuration are channels 1, 6, and 7, while TIM1 is used as the PWM generation timer.

```
const R3_1_Params_t R3_1_ParamsM1 =
{
    .ADCx = ADC1,
    .RepetitionCounter = REP_COUNTER,
    .TIMx = TIM1,
    .LowSideOutputs = (LowSideOutputsFunction_t)LOW_SIDE_SIGNALS_ENABLING,
    .BKIN2Mode = EXT_MODE,
    .DAC_OCP_Threshold = 23830,
    .DAC_OVP_Threshold = 23830,
};
```

Listing 3.1: Extract of current-sensing and PWM hardware parameters from mc-parameters.c.

This configuration links the theoretical current-control loop to the real hardware implementation. The FOC algorithm computes voltage commands in the rotating reference frame, but the controller can only regulate current correctly if the ADC samples are synchronized with the PWM cycle. The use of TIM1 trigger output for injected ADC conversions ensures that the phase-current measurement is performed at deterministic

instants of the PWM period. The presence of break input and protection thresholds also shows that the system is a complete embedded motor drive with safety functions, not only a control algorithm.

3.12 Protection and Fault Handling

Motor-control applications require protection against electrical and control faults. The MCSDK includes fault management for overcurrent, overvoltage, overtemperature, speed-feedback reliability error, and FOC execution overrun. In this thesis, a fault acknowledgement command was implemented through UART because after a stop or a fault event the motor may not restart unless the fault state is acknowledged.

During the first laboratory commissioning phase, the serial-control firmware was organized around three basic commands:

- **Start**: starts the motor;
- **Stop**: stops the motor; after stop some fault condition can be generated or remain active;
- **Ack**: acknowledges the fault, allowing a new start command to be accepted.

This command structure became the basis for the later UART command interpreter developed in the thesis.

Chapter 4

STM32 MCSDK Architecture and Firmware Implementation

4.1 STM32 MCSDK Architecture

4.1.1 MCSDK Overview

The STM32 PMSM FOC SDK is designed to implement FOC drives for three-phase PMSMs using STM32 microcontrollers. It includes motor-control algorithms, current sensing, speed estimation, speed and current regulators, PWM generation, communication interfaces, and a graphical configuration environment. The user manual describes support for SM-PMSM and IPMSM motors, current sampling methods, sensorless speed feedback, programmable speed ramps, programmable torque ramps, real-time tuning, and fault management.

4.1.2 Software Layers

The MCSDK architecture can be understood as a set of layers:

- low-level drivers and CMSIS/peripheral libraries;
- motor-control library classes;
- motor-control application layer;
- user interface and communication layer;
- user application code.

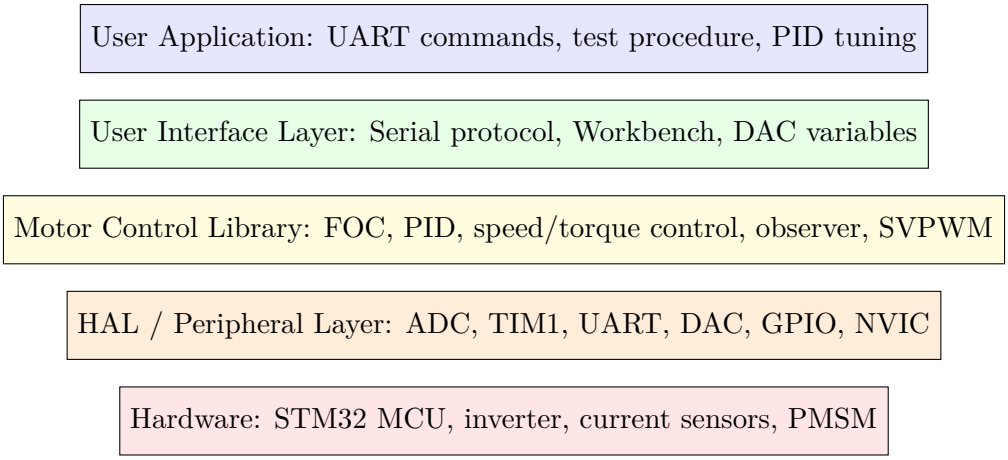


Figure 4.1: Simplified software and hardware layers of the implemented MCSDK-based system.

4.1.3 Motor Profiler and Workbench

The Motor Profiler was used to identify or configure motor parameters before firmware generation. During this activity, a speed-feedback fault appeared during mechanical model analysis, as shown in Fig. 4.2. The error message indicates that this can happen when motor load changes too quickly or when the maximum speed specification is too low. This observation is important because it shows the sensitivity of sensorless profiling to load, inertia, and correct configuration.

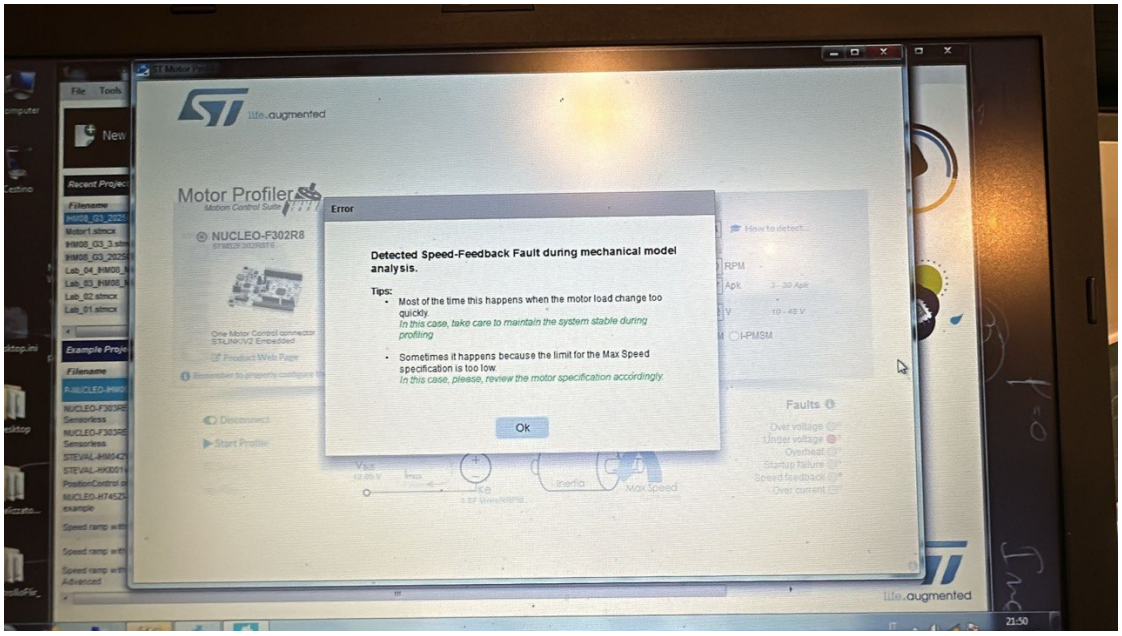


Figure 4.2: Speed-feedback fault detected during the Motor Profiler mechanical model analysis.

4.1.4 Speed and Measurement Units

The MCSDK API does not always use physical units directly. Some functions use internal units or scaled digital quantities. This aspect became a central point in the thesis because

the user speed command entered in rpm through UART did not initially correspond to the real motor speed. In the official serial protocol, ramp data are documented as rpm values; however, in the direct user firmware call used here, the practical behavior required a conversion factor between the user terminal input and the generated speed unit.

4.1.5 Fault Management

The MCSDK includes fault management for overcurrent, overvoltage, overtemperature, speed-feedback reliability error, and FOC duration errors. In the developed test bench, fault acknowledgement was exposed to the user through the Ack command. This decision was practical because after a stop or a fault event the system may remain in a state that prevents the next start command. The command sequence therefore became:

`Start` → motor runs, `Stop` → motor stops, `Ack` → fault cleared.

4.1.6 ST Motor Control Workbench Monitoring

The ST Motor Control Workbench was used for monitoring and tuning. Figure 4.3 shows the advanced monitoring screen used during testing. The screenshot shows speed control mode, bus voltage, heatsink temperature, current references, measured currents, speed ramp reference, observer parameters, and PID gains.

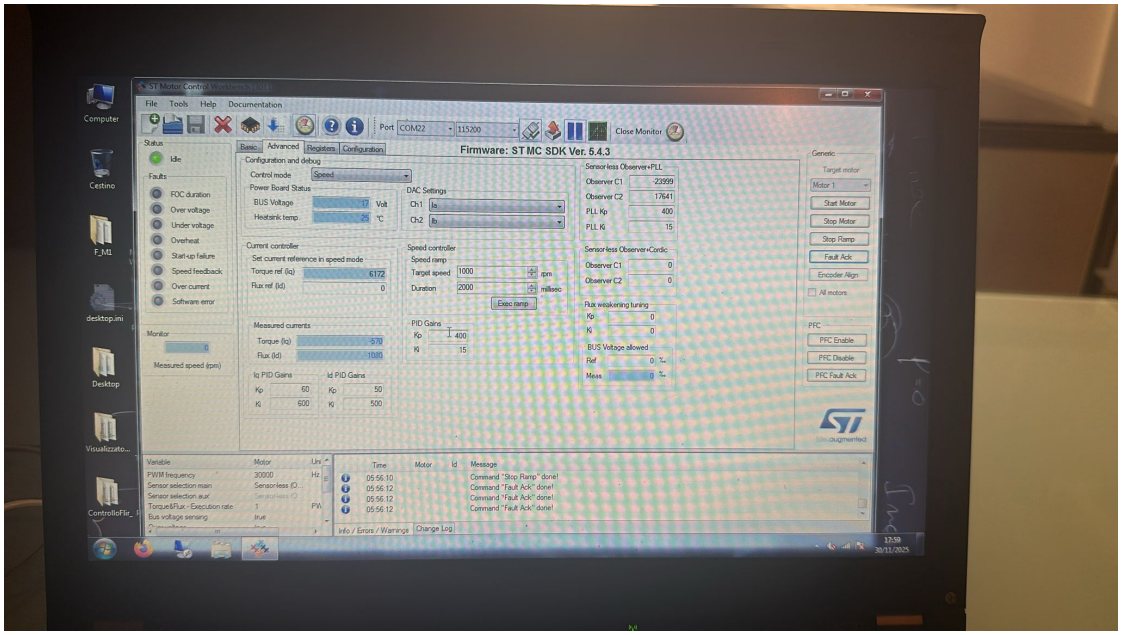


Figure 4.3: ST Motor Control Workbench advanced monitoring screen used during PID and speed tests.

4.2 Firmware Implementation

4.2.1 Development Environment

The firmware was developed and debugged using Atollic TrueSTUDIO for STM32. The project was generated from the ST Motor Control Workbench and then modified in the user-code sections. The most important modified files were:

- `main.c`: UART command interface, command parsing, speed command management, start/stop/ack logic, state/fault printing, and runtime PID-related debugging;
- `usart.c` and `usart.h`: manual USART2 integration because the original Workbench project did not generate a CubeMX USART peripheral layer;
- `stm32f3xx_it.c`: USART2 interrupt handler for interrupt-based reception;
- `mc_config.c`: default PID handles, speed/torque controller configuration, observer parameters, PWM handle, voltage and temperature sensing structures;
- `mc_parameters.c`: generated hardware configuration for TIM1, ADC1, three-shunt current sensing, DAC thresholds, and protection signals.

4.2.2 Manual USART2 Integration

The original motor-control project was generated from ST Motor Control Workbench and did not initially contain a generated CubeMX USART2 layer. When UART commands were first added to `main.c`, the build failed because the UART handle, header, initialization function, and HAL driver activation were missing. The errors included undeclared `huart2`, unknown `UART_HandleTypeDef`, implicit declarations of `HAL_UART_Receive_IT()` and `HAL_UART_Transmit()`, and linker errors for the HAL UART functions.

The integration was solved by manually adding `usart.h` and `usart.c`, defining `UART_HandleTypeDef huart2`, enabling the HAL UART module, configuring PA2/PA3 as USART2 TX/RX, enabling the USART2 interrupt, and calling `MX_USART2_UART_Init()` before activating interrupt reception. This step converted the project from a pure MCSDK-generated motor-control application into an interactive embedded test-bench firmware.

```
1 UART_HandleTypeDef huart2;
2
3 void MX_USART2_UART_Init(void)
4 {
5     huart2.Instance = USART2;
6     huart2.Init.BaudRate = 115200;
7     huart2.Init.WordLength = UART_WORDLENGTH_8B;
```

```
    huart2.Init.StopBits = UART_STOPBITS_1;
    huart2.Init.Parity = UART_PARITY_NONE;
    huart2.Init.Mode = UART_MODE_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;

    if (HAL_UART_Init(&huart2) != HAL_OK)
    {
        Error_Handler();
    }
}
```

Listing 4.1: Manual USART2 initialization added to the motor-control project.

```
void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2);
}
```

Listing 4.2: USART2 interrupt handler added for interrupt-based UART reception.

4.2.3 UART Communication Parameters

The first command interface was implemented with PuTTY. The terminal was configured on COM22 with 115200 baud, 8 data bits, one stop bit, no parity, and no flow control. These settings match the USART2 firmware initialization and were used for all manual tests.

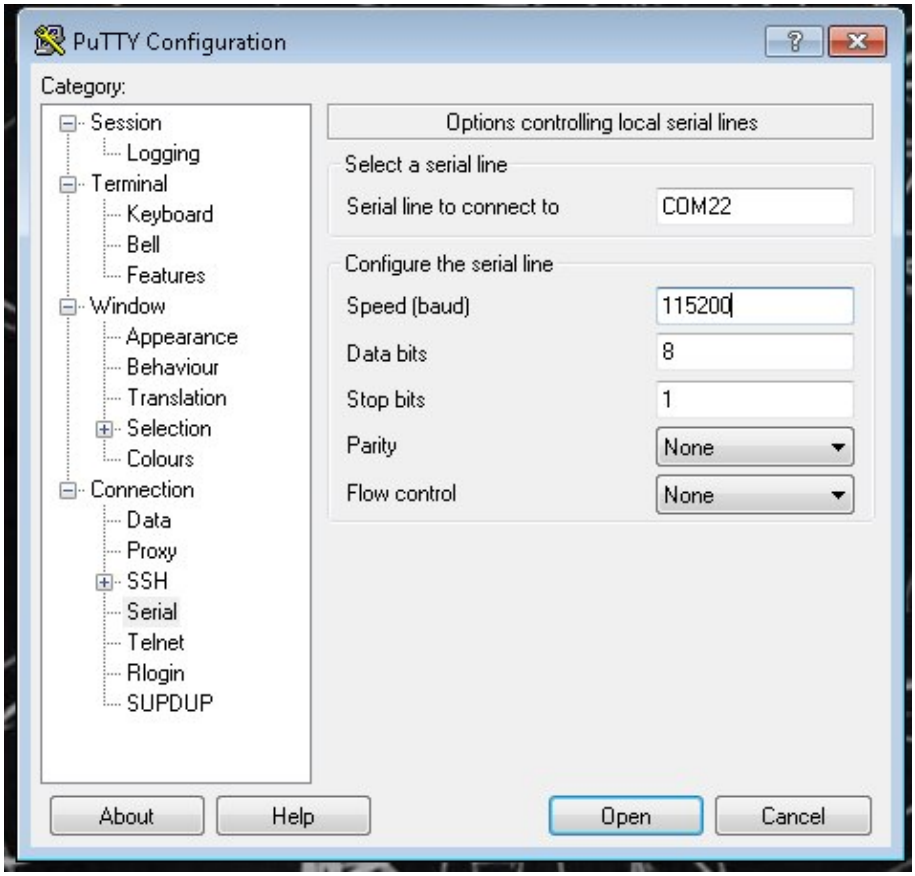


Figure 4.4: PuTTY serial configuration used for the STM32 test bench: COM22, 115200 baud, 8 data bits, 1 stop bit, no parity, and no flow control.

Parameter	Value
Terminal	PuTTY
Port	COM22
Baud rate	115200 bit/s
Word length	8 bit
Parity	None
Stop bits	1
Flow control	None
Peripheral	USART2
TX/RX pins	PA2 / PA3

Table 4.1: Final serial terminal configuration used for UART motor commands.

4.2.4 Serial Command Interpreter and Runtime Feedback

The first working interpretation rules implemented three commands: Start, Stop, and Ack. The command set was then extended with numeric speed references and PID-related debugging commands. The final command philosophy is deterministic:

- if the motor is stopped, a numeric speed command is stored and applied only after

the `S` command;

- if the motor is already running, a numeric speed command is applied as a new speed ramp;
- empty terminal lines are ignored, preventing accidental zero-speed commands;
- the firmware prints motor-state changes and MCSDK fault codes in PuTTY.

This improvement was important because the system could previously accelerate and then return to zero without clear information in the terminal. After adding state and fault printing, the test bench became easier to debug remotely.

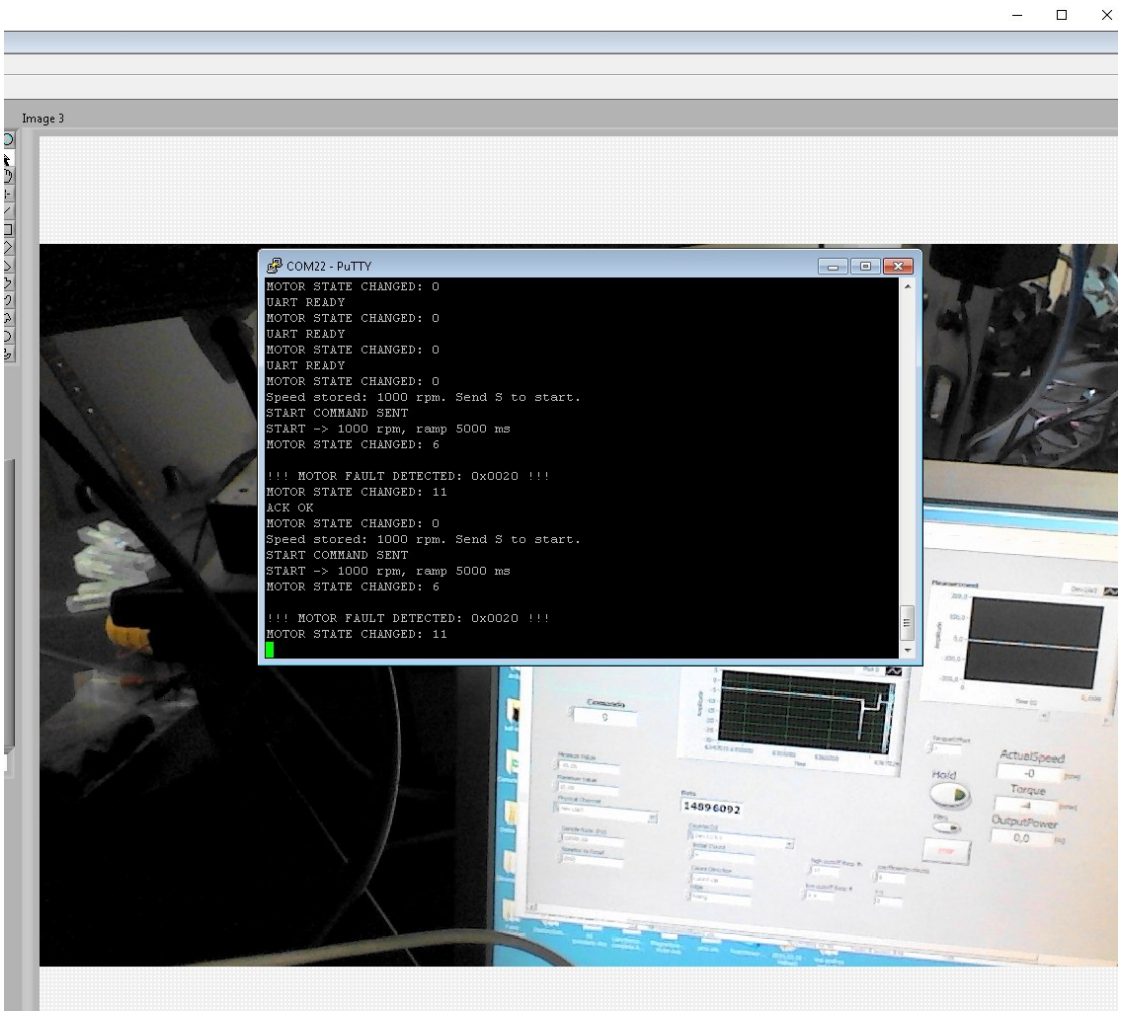


Figure 4.5: Final PuTTY runtime debug screen showing UART READY messages, stored speed command, start command, motor-state transitions, ACK OK response, and fault code 0x0020.

4.2.5 Main Variables

The following variables were added to store command flags, the received command string, speed reference, and PID tuning parameters.

```
uint8_t rx_data[1];

uint8_t cmd_start = 0;
uint8_t cmd_stop = 0;
uint8_t cmd_ack = 0;

char cmd_string[20];
uint8_t cmd_index = 0;

int speed_ref = 0;
uint8_t speed_cmd_ready = 0;

float Kp_iq = 0.05f;
float Ki_iq = 0.01f;
float Kp_id = 0.05f;
float Ki_id = 0.01f;

uint8_t pid_update_flag = 0;
```

Listing 4.3: Main user variables added in main.c.

4.2.6 Runtime Feedback and PID Update Functions

The final firmware prints motor-state changes and fault codes in PuTTY. This was introduced to make the experimental results repeatable and to distinguish a normal stop from an MCSDK fault state. The optional runtime PID update function was kept for current-loop debugging, while the final speed PI tuning was performed in extttmc_config.c.

```
void PrintMotorFault(uint16_t faults)
{
    char msg[100];
    sprintf(msg, "\r\n!!! MOTOR FAULT DETECTED: 0x%04X !!!\r\n", faults);
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), 100);
}

void PrintMotorState(uint8_t state)
{
    char msg[80];
    sprintf(msg, "MOTOR STATE CHANGED: %u\r\n", state);
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), 100);
}
```

Listing 4.4: Motor-state and fault reporting functions added in main.c.

```
void UpdateFOCPID(void)
{
    PID_SetKP(&PIDIqHandle_M1, Kp_iq);
    PID_SetKI(&PIDIqHandle_M1, Ki_iq);

    PID_SetKP(&PIDIdHandle_M1, Kp_id);
    PID_SetKI(&PIDIdHandle_M1, Ki_id);
}
```

Listing 4.5: Runtime update function for Iq and Id PID gains.

4.2.7 Main Loop Logic

The main loop processes the flags set by the UART interrupt. This design avoids executing heavy commands inside the interrupt routine. The interrupt only parses the command and sets a flag; the main loop performs the motor-control API calls.

```
while (1)
{
    uint8_t motor_state = (uint8_t)MC_GetSTMStateMotor1();
    uint16_t current_faults  = MC_GetCurrentFaultsMotor1();
    uint16_t occurred_faults = MC_GetOccurredFaultsMotor1();
    uint16_t faults_to_print = current_faults | occurred_faults;

    if (motor_state != last_state)
    {
        PrintMotorState(motor_state);
        last_state = motor_state;
    }

    if ((faults_to_print != 0) && (faults_to_print != last_faults))
    {
        PrintMotorFault(faults_to_print);
        last_faults = faults_to_print;
        motor_running_cmd = 0;
    }

    if (faults_to_print == 0)
    {
```

```
23          last_faults = 0;
24      }
25
26      if (cmd_ack)
27      {
28          MC_AcknowledgeFaultMotor1();
29          last_faults = 0;
30          last_state = 255;
31          motor_running_cmd = 0;
32          HAL_UART_Transmit(&huart2, (uint8_t*)"ACK␣OK\r\n", 8, 100);
33          cmd_ack = 0;
34      }
35
36      if (cmd_start)
37      {
38          if (speed_ref > 3000) speed_ref = 3000;
39          if (speed_ref < 500)  speed_ref = 1000;
40
41          int16_t target_speed = (int16_t)(speed_ref / 6);
42          MC_StartMotor1();
43          HAL_Delay(3500);
44
45          uint16_t faults_after_start =
46              MC_GetCurrentFaultsMotor1() | MC_GetOccurredFaultsMotor1();
47
48          if (faults_after_start != 0)
49          {
50              PrintMotorFault(faults_after_start);
51              motor_running_cmd = 0;
52          }
53          else
54          {
55              MC_ProgramSpeedRampMotor1(target_speed, SPEED_RAMP_TIME_MS);
56              motor_running_cmd = 1;
57          }
58          cmd_start = 0;
59      }
60
61      if (cmd_stop)
62      {
63          MC_StopMotor1();
```

```
64          motor_running_cmd = 0;
65          HAL_UART_Transmit(&huart2, (uint8_t*)"STOP␣OK\r\n", 9, 100);
66          cmd_stop = 0;
67      }
68
69      if (speed_cmd_ready)
70      {
71          if (speed_ref > 3000) speed_ref = 3000;
72          if (speed_ref < 500)  speed_ref = 500;
73
74          int16_t final_speed = (int16_t)(speed_ref / 6);
75
76          if (motor_running_cmd)
77          {
78              MC_ProgramSpeedRampMotor1(final_speed, SPEED_RAMP_TIME_MS);
79          }
80          else
81          {
82              // Store the command and apply it after the next Start command.
83          }
84          speed_cmd_ready = 0;
85      }
86  }
```

Listing 4.6: Final main-loop logic for state/fault monitoring, Start, Stop, Ack, and speed commands.

4.2.8 UART Receive Callback

The UART callback receives one character at a time. When a carriage return or newline is received, the buffer is interpreted as a complete command. This approach is simple and robust for terminal-based control.

```
1  void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
2  {
3      if (huart->Instance == USART2)
4      {
5          char c = rx_data[0];
6
7          if (c == '\r' || c == '\n')
8          {
9              if (cmd_index == 0)
```

```
10              {
11                  HAL_UART_Receive_IT(&huart2, rx_data, 1);
12                  return;
13              }
14
15              cmd_string[cmd_index] = '\0';
16
17              if ((strcmp(cmd_string, "S") == 0) || (strcmp(cmd_string, "s") ==
    0))
18              {
19                  cmd_start = 1;
20              }
21              else if ((strcmp(cmd_string, "T") == 0) || (strcmp(cmd_string, "t")
     == 0))
22              {
23                  cmd_stop = 1;
24              }
25              else if ((strcmp(cmd_string, "ACK") == 0) ||
26                       (strcmp(cmd_string, "ack") == 0) ||
27                       (strcmp(cmd_string, "Ack") == 0))
28              {
29                  cmd_ack = 1;
30              }
31              else
32              {
33                  uint8_t is_number = 1;
34                  for (uint8_t i = 0; cmd_string[i] != '\0'; i++)
35                  {
36                      if (cmd_string[i] < '0' || cmd_string[i] > '9')
37                      {
38                          is_number = 0;
39                          break;
40                      }
41                  }
42
43                  if (is_number)
44                  {
45                      speed_ref = atoi(cmd_string);
46                      if (speed_ref < 500)  speed_ref = 500;
47                      if (speed_ref > 3000) speed_ref = 3000;
48                      speed_cmd_ready = 1;
```

```
                }
            }
            cmd_index = 0;
        }
        else if (cmd_index < sizeof(cmd_string) - 1)
        {
            cmd_string[cmd_index++] = c;
        }

        HAL_UART_Receive_IT(&huart2, rx_data, 1);
    }
}
```

Listing 4.7: Final UART receive callback with empty-line filtering and numeric-command validation.

4.2.9 Debugging Build Errors

During the development, compilation errors were encountered and solved. One example is shown in Fig. 4.6. The error was related to an undeclared UART handle in an earlier version of the code. Such errors are common when integrating user code into an automatically generated project because peripheral names and user variable names must match the generated HAL structures.

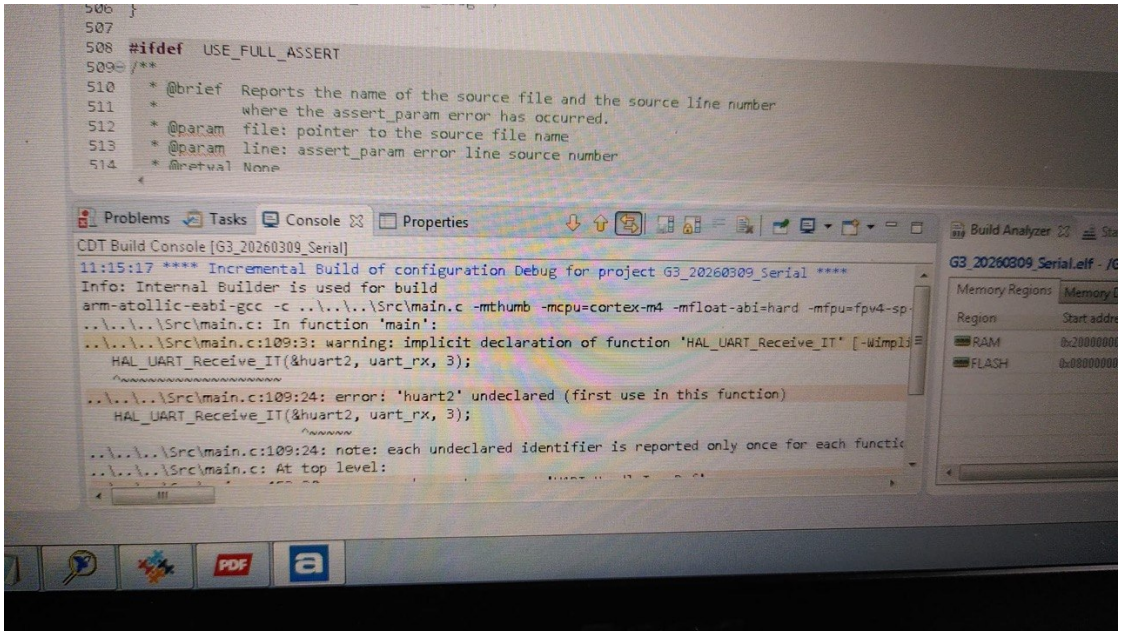


Figure 4.6: Example of build error during UART integration. The issue was related to an undeclared UART handle in an earlier implementation stage.

4.2.10 PID Configuration in mc_config.c

The file `mc_config.c` defines the speed, torque-current, and flux-current PID structures. In the final campaign these three handles were actively modified. The speed loop was changed to `Kp = 290` and `Ki = 20`; the Iq torque-current loop was changed to `Kp = 200` and `Ki = 300`; and the Id flux-current loop was changed to `Kp = 200` and `Ki = 300`. The generated divisors, output limits, and saturation limits were kept consistent with the MCSDK project.

```
PID_Handle_t PIDSpeedHandle_M1 =
{
  .hDefKpGain = 290,  // speed-loop Kp, previously 250
  .hDefKiGain = 20,   // speed-loop Ki, previously 50
  .wUpperIntegralLimit = (int32_t)IQMAX * (int32_t)SP_KIDIV,
  .wLowerIntegralLimit = -(int32_t)IQMAX * (int32_t)SP_KIDIV,
  .hUpperOutputLimit = (int16_t)IQMAX,
  .hLowerOutputLimit = -(int16_t)IQMAX,
  .hKpDivisor = (uint16_t)SP_KPDIV,
  .hKiDivisor = (uint16_t)SP_KIDIV,
};

PID_Handle_t PIDIqHandle_M1 =
{
  .hDefKpGain = 200,  // torque loop Kp, previously default 2800
  .hDefKiGain = 300,  // torque loop Ki, previously default 1606
  .hKpDivisor = (uint16_t)TF_KPDIV,
  .hKiDivisor = (uint16_t)TF_KIDIV,
};

PID_Handle_t PIDIdHandle_M1 =
{
  .hDefKpGain = 200,  // flux loop Kp
  .hDefKiGain = 300,  // flux loop Ki
  .hKpDivisor = (uint16_t)TF_KPDIV,
  .hKiDivisor = (uint16_t)TF_KIDIV,
};
```

Listing 4.8: Final PID handles modified in mc_config.c.

The speed PID is especially important because it defines how the speed error is converted into a torque-producing current reference. If the speed-loop gain is too low, the response becomes slow. If the gain is too high, oscillations can increase. If the integral gain is excessive, integral wind-up can produce overshoot and long settling time.

4.2.11 Additional `mc_config.c` Structures Used by the Drive

Besides the speed PID handle, `mc_config.c` defines several structures that determine the real behavior of the drive:

- **`PIDSpeedHandle_M1`** defines the outer speed-loop controller. In the final tested project the selected speed PI values were `hDefKpGain = 290` and `hDefKiGain = 20`, used together with the generated divisors `SP_KPDIV` and `SP_KIDIV`. These are fixed-point gains, not floating-point physical gains.
- **`PIDIqHandle_M1`** regulates the torque-producing current component.
- **`PIDIdHandle_M1`** regulates the flux current component. In a surface-mounted PMSM, the reference is normally close to zero in the base-speed region.
- **`SpeednTorqCtrlM1`** defines speed and torque references, maximum and minimum mechanical-speed units, current limits, default control mode, and default flux component.
- **`RevUpControlM1`** defines the start-up sequence. This is essential in a sensorless system because the observer requires enough back-EMF or a valid transition procedure before closed-loop operation.
- **`STO_PLL_M1`** defines the state observer and PLL parameters used to estimate rotor speed and position.
- **`PWM_Handle_M1`** connects the FOC algorithm to the three-shunt PWM/current-feedback driver.
- **Voltage and temperature sensor structures** define DC-bus and thermal feedback thresholds used for protection.

The distinction between generated configuration code and user code is important. The generated configuration defines the baseline drive, while the user code developed in this thesis adds interaction, testing, and tuning capability. In the final project, maximum speed, nominal current, and observer gains were interpreted using the Workbench project and the profiler/firmware configuration values used during the laboratory campaign.

```
1 SpeednTorqCtrl_Handle_t SpeednTorqCtrlM1 =
2 {
3     .STCFrequencyHz = MEDIUM_FREQUENCY_TASK_RATE,
4     .MaxAppPositiveMecSpeedUnit = (uint16_t)(MAX_APPLICATION_SPEED_UNIT),
5     .MinAppPositiveMecSpeedUnit = (uint16_t)(MIN_APPLICATION_SPEED_UNIT),
6     .MaxPositiveTorque = (int16_t)NOMINAL_CURRENT,
```

```
    .MinNegativeTorque = -(int16_t)NOMINAL_CURRENT,
    .ModeDefault = DEFAULT_CONTROL_MODE,
    .MecSpeedRefUnitDefault = (int16_t)(DEFAULT_TARGET_SPEED_UNIT),
    .TorqueRefDefault = (int16_t)DEFAULT_TORQUE_COMPONENT,
    .IdrefDefault = (int16_t)DEFAULT_FLUX_COMPONENT,
};
```

Listing 4.9: Speed and torque controller structure from mc-config.c, simplified.

The speed and torque controller structure explains why a direct user command cannot be treated as an isolated number: it is constrained by generated speed-unit limits, default control mode, and torque/current boundaries.

Chapter 5

Speed Control Synchronization and PID Tuning

5.1 Speed Control Synchronization

5.1.1 Objective

The objective of the speed-control synchronization experiment was to make the speed entered by the user in PuTTY correspond to the real mechanical speed of the motor. This was essential because the test bench is intended to be used as an experimental tool: a command such as `1000` produces a stable speed close to 1000 rpm, not an arbitrary internal firmware value.

5.1.2 Initial Problem

During the first tests, the motor did not follow the user command correctly. Small terminal inputs produced much higher real speed. The observed behavior was approximately:

- input `200` produced approximately 1200 rpm;
- input `500` produced approximately 3000 rpm;
- input `800` produced approximately 4200 rpm;
- commands around 2000–3000 rpm were not applied correctly.

This indicated that the custom UART layer was sending a value that was interpreted by the MCSDK speed-ramp function as an internal speed unit rather than as direct mechanical rpm.

5.1.3 Root Cause

The direct firmware call:

`MC_ProgramSpeedRampMotor1(final_speed, duration_ms);`

requires the value passed by the user layer to be compatible with MCSDK internal speed units. The test bench user, however, enters mechanical speed in rpm. The mismatch was therefore a unit-conversion problem between the terminal-level representation and the firmware-level representation. The practical experimental relation was:

$$\omega_{real} \approx 6\,\omega_{input}. \tag{5.1}$$

5.1.4 Implemented Scaling

The correction was implemented by dividing the requested mechanical speed by 6 before calling the MCSDK ramp function:

$$Speed_{MCSDK} = \frac{Speed_{command}}{6}. \tag{5.2}$$

A software safety limit was also applied so that the user cannot command more than 3000 rpm from the terminal.

```
if (speed_cmd_ready)
{
    if (speed_ref > 3000) speed_ref = 3000;
    if (speed_ref < 0) speed_ref = 0;

    int16_t final_speed = (int16_t)(speed_ref / 6);
    MC_ProgramSpeedRampMotor1(final_speed, SPEED_RAMP_TIME_MS);

    char msg[50];
    sprintf(msg, "Speed␣set:␣%d␣rpm\r\n", speed_ref);
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), 100);

    speed_cmd_ready = 0;
}
```

Listing 5.1: Final speed scaling implementation used in the UART command layer.

5.1.5 Final Speed Accuracy Measurements

The final speed validation was performed on project `G3_20260309_Serial_4` with the PMSM and brake attached. The command sequence was `ACK`, numeric speed, `S`, observation

of stable speed and torque, and `T`. The final PI setting used after the tuning campaign was $K_p = 290$, $K_i = 20$ for the speed loop, with $K_p = 200$, $K_i = 300$ for both Iq and Id current loops; the implemented speed scaling was MC command = rpm/6.

The percentage error was calculated as:

$$Error(\%) = \frac{|Speed_{measured} - Speed_{commanded}|}{Speed_{commanded}} \times 100. \tag{5.3}$$

Test	Commanded speed (rpm)	Scaled command	Measured speed (rpm)	Error (%)	Torque value	Fault
P1	500	83	505	1.00	-7	No
P2	1000	167	1005	0.50	-7	No
P3	1500	250	1509	0.60	-7	No
P4	2000	333	2007	0.35	-7	No
P5	2500	417	2505	0.20	-7	No
P6	3000	500	3004	0.13	-7	No

Table 5.1: Final speed-synchronization validation after applying the rpm/6 scaling correction.

The largest absolute speed error in Table 5.1 is 1.00% at 500 rpm, while the average absolute error over the six tested points is approximately 0.46%. This confirms that the implemented scaling correction gives a nearly linear command-to-speed relation across the final operating range.

5.1.6 Graphical Analysis

The measurements show that the actual stable speed is very close to the commanded value over the full tested range. The error remains below 1% for all tested commands and decreases at higher speed.

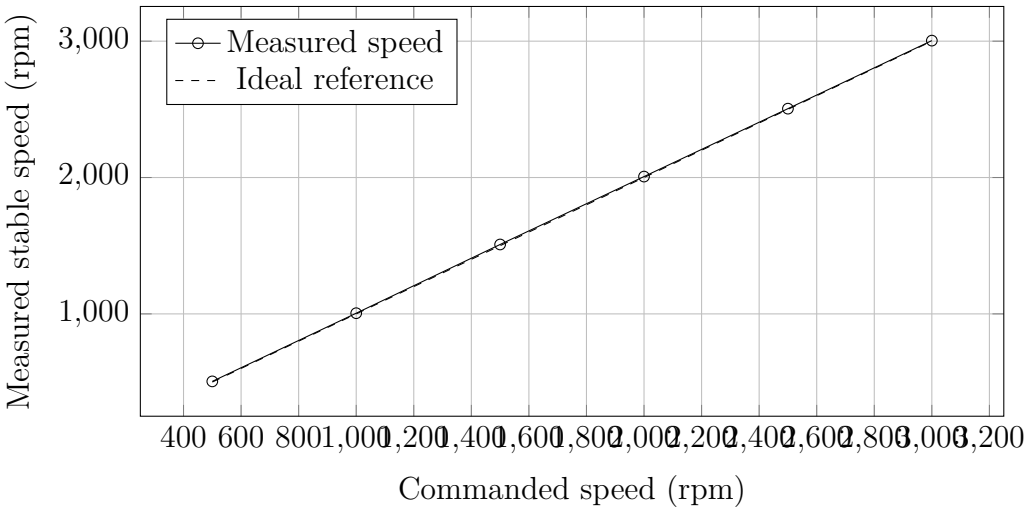


Figure 5.1: Measured stable speed compared with the ideal commanded speed after synchronization.

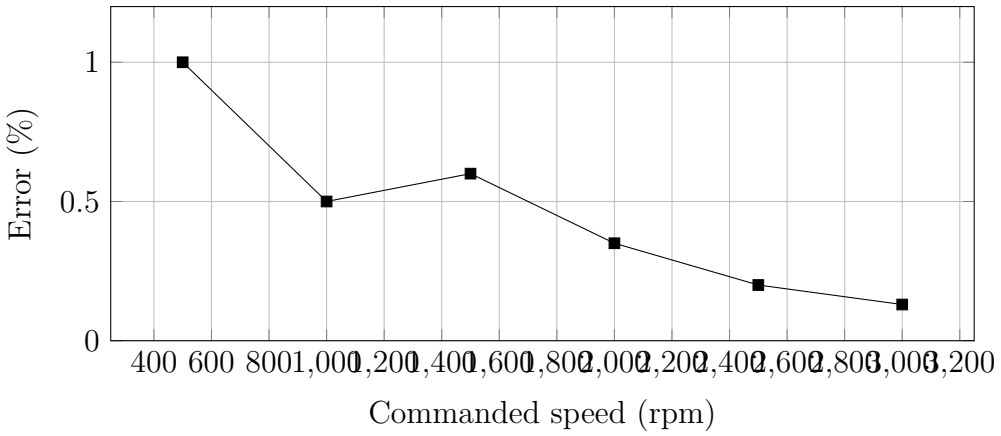


Figure 5.2: Speed tracking error for the final synchronized UART speed commands.

5.1.7 Discussion

The final speed-synchronization result confirms that the implemented UART command layer is suitable for controlled laboratory operation. The terminal command is now meaningful for the operator: entering 500, 1000, 1500, 2000, 2500, or 3000 produces a stable measured speed close to the same rpm value. The absence of faults during the validation campaign also confirms that the scaling correction did not destabilize the MCSDK speed-control state machine.

5.2 PID Control Implementation and Tuning

5.2.1 Control Structure

The implemented motor control system uses nested control loops. The outer speed PI controller converts the speed error into a torque-producing current reference i_q^*. The inner current loops regulate i_q and i_d, and the inverter applies the resulting voltage vector through SVPWM. This structure is standard in FOC and is directly supported by the STM32 MCSDK.

$$\text{Speed reference} \rightarrow \text{Speed PI} \rightarrow i_q^* \rightarrow \text{Iq PI} \rightarrow v_q^* \rightarrow \text{SVPWM} \rightarrow \text{Motor}.$$

5.2.2 Purpose of PID Tuning

The motor was tested with a brake attached. The profiler identified a relatively high inertia and friction for the complete motor-brake system. Consequently, the speed loop is sensitive to proportional gain, integral gain, and ramp time. PID tuning was therefore necessary to reduce overshoot, avoid oscillations, prevent observer faults, and reach a stable speed after each command.

5.2.3 UART Tuning and Debugging Commands

The command interface accepts the following commands:

Command	Function
`S`	Start motor after stored speed command and rev-up delay
`T`	Stop motor safely
`ACK`	Clear fault state before a new start
`500-3000`	Store or apply speed reference in rpm
`KP_IQ x`	Runtime command for Iq proportional gain in the user interface
`KI_IQ x`	Runtime command for Iq integral gain in the user interface
`KP_ID x`	Runtime command for Id proportional gain in the user interface
`KI_ID x`	Runtime command for Id integral gain in the user interface

Table 5.2: UART commands implemented for motor operation and control-loop debugging.

The final firmware also prints state changes and fault codes. This is essential for PID tuning because a speed drop to 0 rpm can be caused by a controller fault rather than by a simple mechanical slowdown.

5.2.4 Speed PI Parameters in `mc_config.c`

The speed PI gains are defined in `PIDSpeedHandle_M1`. In the MCSDK generated project, these values are fixed-point controller parameters and are used together with the generated

divisors. During the tuning campaign the speed PI proportional and integral gains were changed in `mc_config.c`, while the generated divisors and current limits were kept unchanged. The final speed-ramp definition used in `main.c` was `SPEED_RAMP_TIME_MS = 10000`, which gave a smoother response with the brake/load attached.

```
PID_Handle_t PIDSpeedHandle_M1 =
{
  .hDefKpGain          = 290,  // final speed-loop Kp
  .hDefKiGain          = 20,   // final speed-loop Ki
  .wUpperIntegralLimit = (int32_t)IQMAX * (int32_t)SP_KIDIV,
  .wLowerIntegralLimit = -(int32_t)IQMAX * (int32_t)SP_KIDIV,
  .hUpperOutputLimit   = (int16_t)IQMAX,
  .hLowerOutputLimit   = -(int16_t)IQMAX,
  .hKpDivisor          = (uint16_t)SP_KPDIV,
  .hKiDivisor          = (uint16_t)SP_KIDIV,
  .hKpDivisorPOW2      = (uint16_t)SP_KPDIV_LOG,
  .hKiDivisorPOW2      = (uint16_t)SP_KIDIV_LOG,
};

PID_Handle_t PIDIqHandle_M1 =
{
  .hDefKpGain          = 200,  // final torque-current Kp
  .hDefKiGain          = 300,  // final torque-current Ki
  .wUpperIntegralLimit = (int32_t)INT16_MAX * TF_KIDIV,
  .wLowerIntegralLimit = (int32_t)-INT16_MAX * TF_KIDIV,
  .hUpperOutputLimit   = INT16_MAX,
  .hLowerOutputLimit   = -INT16_MAX,
  .hKpDivisor          = (uint16_t)TF_KPDIV,
  .hKiDivisor          = (uint16_t)TF_KIDIV,
};

PID_Handle_t PIDIdHandle_M1 =
{
  .hDefKpGain          = 200,  // final flux-current Kp
  .hDefKiGain          = 300,  // final flux-current Ki
  .wUpperIntegralLimit = (int32_t)INT16_MAX * TF_KIDIV,
  .wLowerIntegralLimit = (int32_t)-INT16_MAX * TF_KIDIV,
  .hUpperOutputLimit   = INT16_MAX,
  .hLowerOutputLimit   = -INT16_MAX,
  .hKpDivisor          = (uint16_t)TF_KPDIV,
  .hKiDivisor          = (uint16_t)TF_KIDIV,
};
```

Listing 5.2: Final PI/PID configuration used for speed, Iq, and Id control in mc_config.c.

5.2.5 Effect of Kp, Ki, and Ramp Time

The speed PI control law can be interpreted as:

$$T_{cmd} = K_p(\omega^* - \omega) + K_i \int (\omega^* - \omega)dt, \tag{5.4}$$

where T_{cmd} is represented in the firmware by a torque-producing current reference. Increasing K_p usually makes the response faster, but too much proportional action increases overshoot and oscillation. Increasing K_i reduces steady-state error, but excessive integral action can cause wind-up. In a high-inertia system, wind-up can continue to accelerate the motor even after the speed target has been reached.

Ramp time acts on the reference trajectory rather than on the controller itself. A longer ramp reduces acceleration demand and mechanical shock. This is why a 10000 ms ramp produced more usable results than several 5000 ms tests with the brake attached.

5.2.6 LabVIEW/Excel-Based PID Response Evaluation

After each PID modification, the response was observed in LabVIEW and then exported as spreadsheet data. The exported files were used to verify that the curves seen in LabVIEW were consistent with the data opened in Excel. For this thesis, the first column of each export was used as the speed-response signal. The file names identify the controller parameters; for example, `Kpsp` and `Kisp` represent the speed-loop proportional and integral gains, while `Kpiq`, `Kiiq`, `Kpid`, and `Kiid` represent torque-current and flux-current-loop parameters.

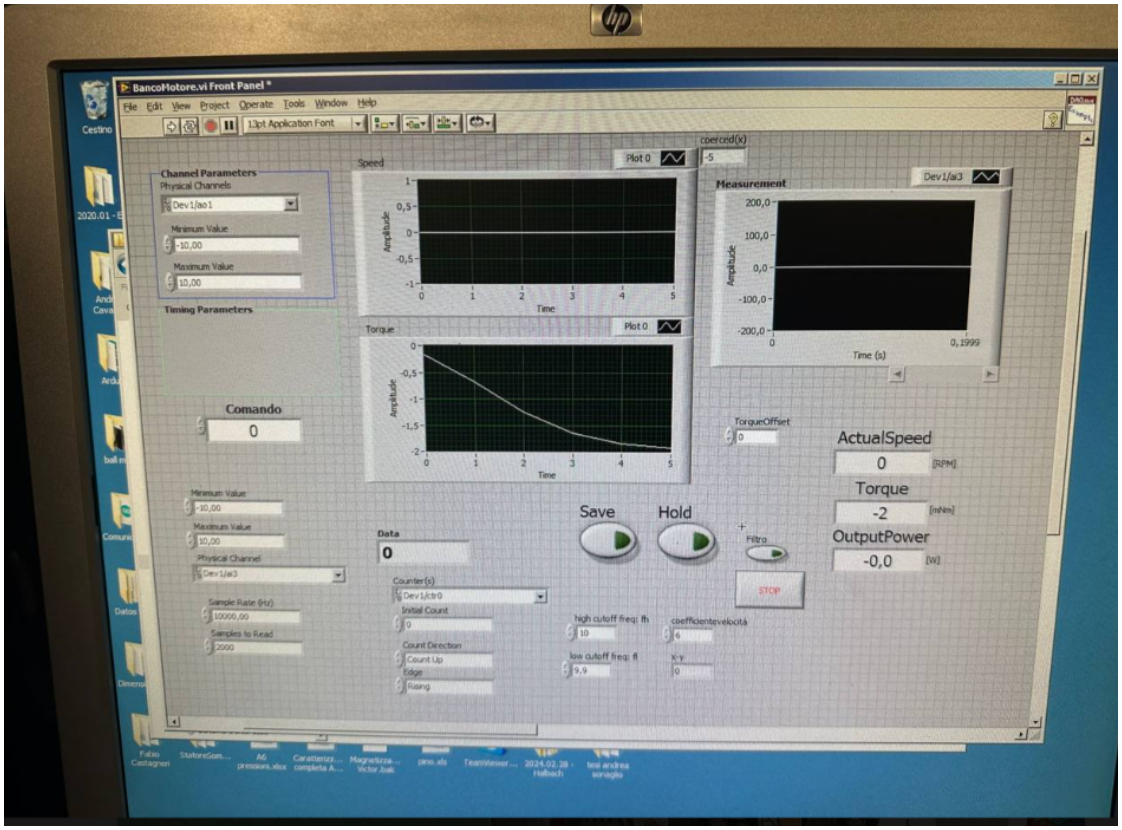


Figure 5.3: LabVIEW front panel used to monitor speed, torque, output power, and the acquired curves during the motor test-bench experiments.

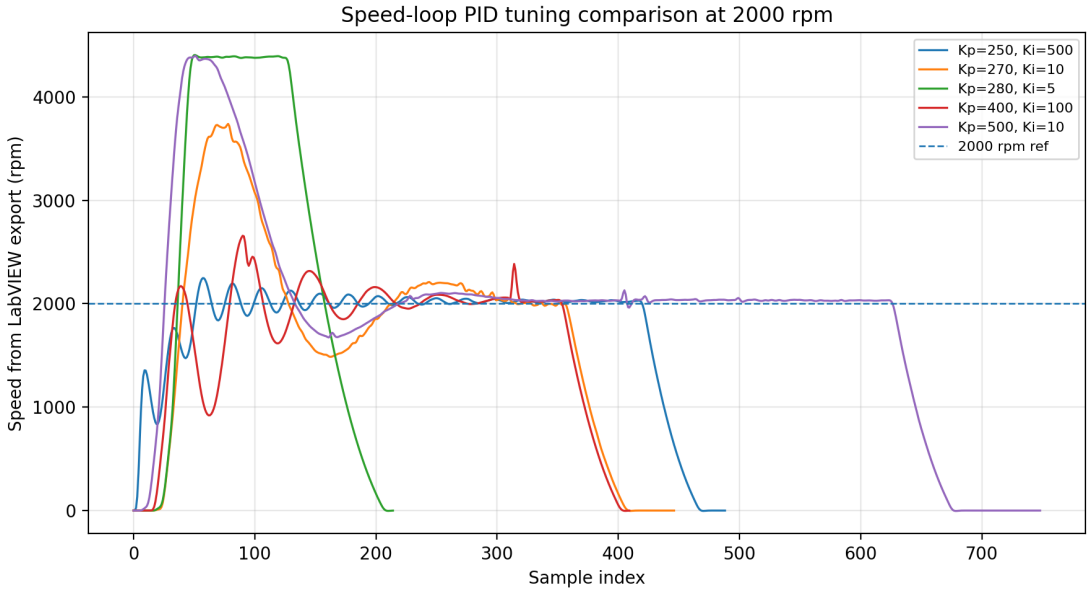


Figure 5.4: Speed-loop PID comparison obtained from the LabVIEW/Excel exports at a 2000 rpm command. The first data column was used as the speed signal.

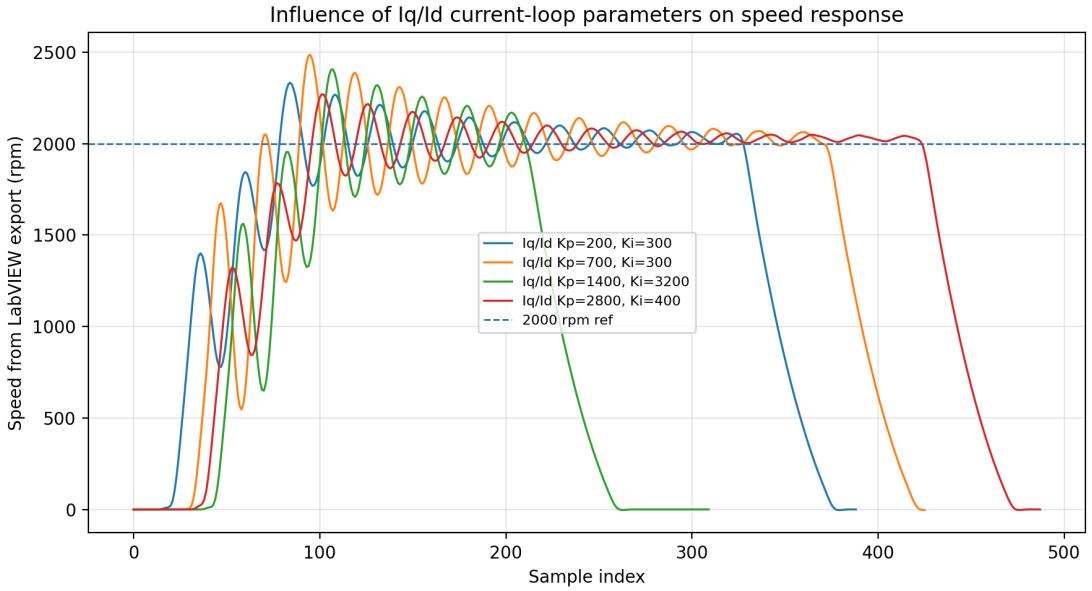


Figure 5.5: Effect of changing Iq/Id current-loop parameters while observing the speed response. The curves show why the torque and flux current loops also influence the final speed behavior.

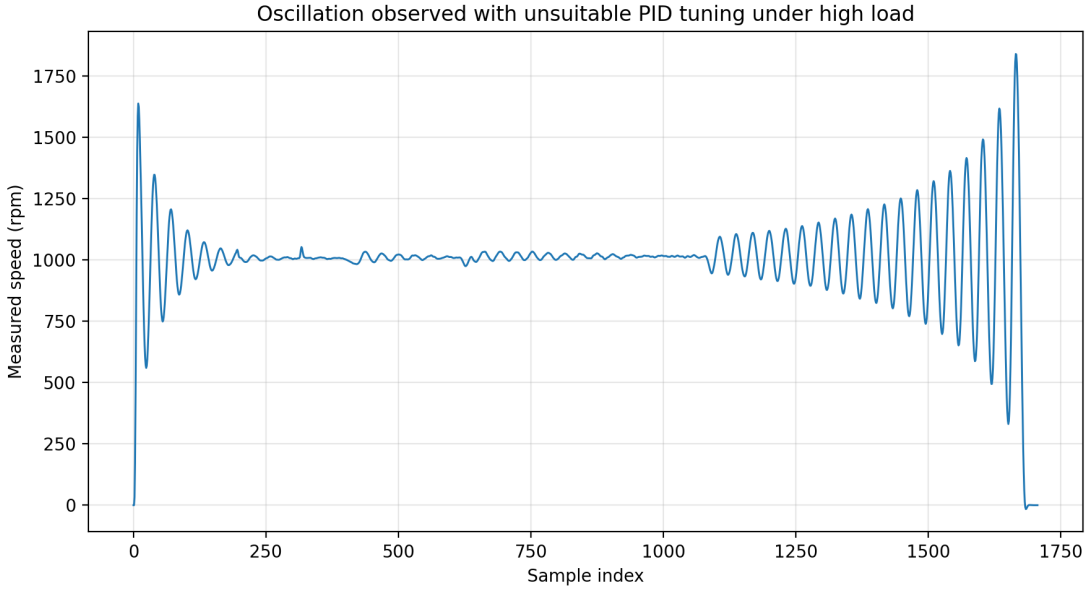


Figure 5.6: Example of oscillating behavior with unsuitable PID tuning under high-load conditions.

Test group	**Main parameter change**	**Observed behaviour**	**Engineering interpretation**
Speed-loop PI tests at 2000 rpm	Different speed-loop K_p and K_i values	Overshoot, oscillation, different settling times, and in some cases fault events	The outer speed loop is strongly affected by the inertia and friction of the motor–brake assembly. Excessive integral action increases wind-up and destabilizes the response.
I_q/I_d current-loop tests at 2000 rpm	Modified torque-current and flux-current gains	Different oscillation amplitude, recovery behaviour, and final speed quality	The inner current loops influence how accurately the requested torque-producing and flux currents follow their references; therefore they also affect the measured speed response.
Ramp-time tests at 2000 rpm	Longer command ramp, especially 10000 ms	Improved stability and fewer fault events compared with aggressive ramps	A slower reference variation reduces mechanical shock and is more suitable for the high-inertia motor–brake configuration.
Final stable configuration	Reduced speed integral action and selected current-loop gains	Stable operation in the validated 500–3000 rpm speed range	The final configuration is appropriate for repeatable experimental measurements and for the subsequent efficiency-map campaign.

Table 5.3: Summary of the LabVIEW/Excel PID response exports used to compare the influence of speed-loop and current-loop parameters.

The LabVIEW/Excel campaign shows that the best behavior was not obtained only by changing the speed-loop proportional gain. The torque-current and flux-current PI values also affected the transient response because they determine how accurately the requested torque-producing and flux currents follow their references. The final selected firmware therefore used a combined configuration: speed loop $K_p = 290$, $K_i = 20$, Iq loop $K_p = 200$, $K_i = 300$, and Id loop $K_p = 200$, $K_i = 300$. The reduction of the speed-loop integral gain compared with earlier tests reduced overshoot tendency, while the selected current-loop gains provided a stable current response for the no-load 2000 rpm experiments.

5.2.7 Runtime Fault Evidence

During tuning, the terminal output showed motor-state transitions and fault code 0x0020 in some unstable cases. This objective feedback confirmed that the observed speed drop was linked to the MCSDK state machine and speed-feedback/observer reliability rather than only to visual speed oscillation.

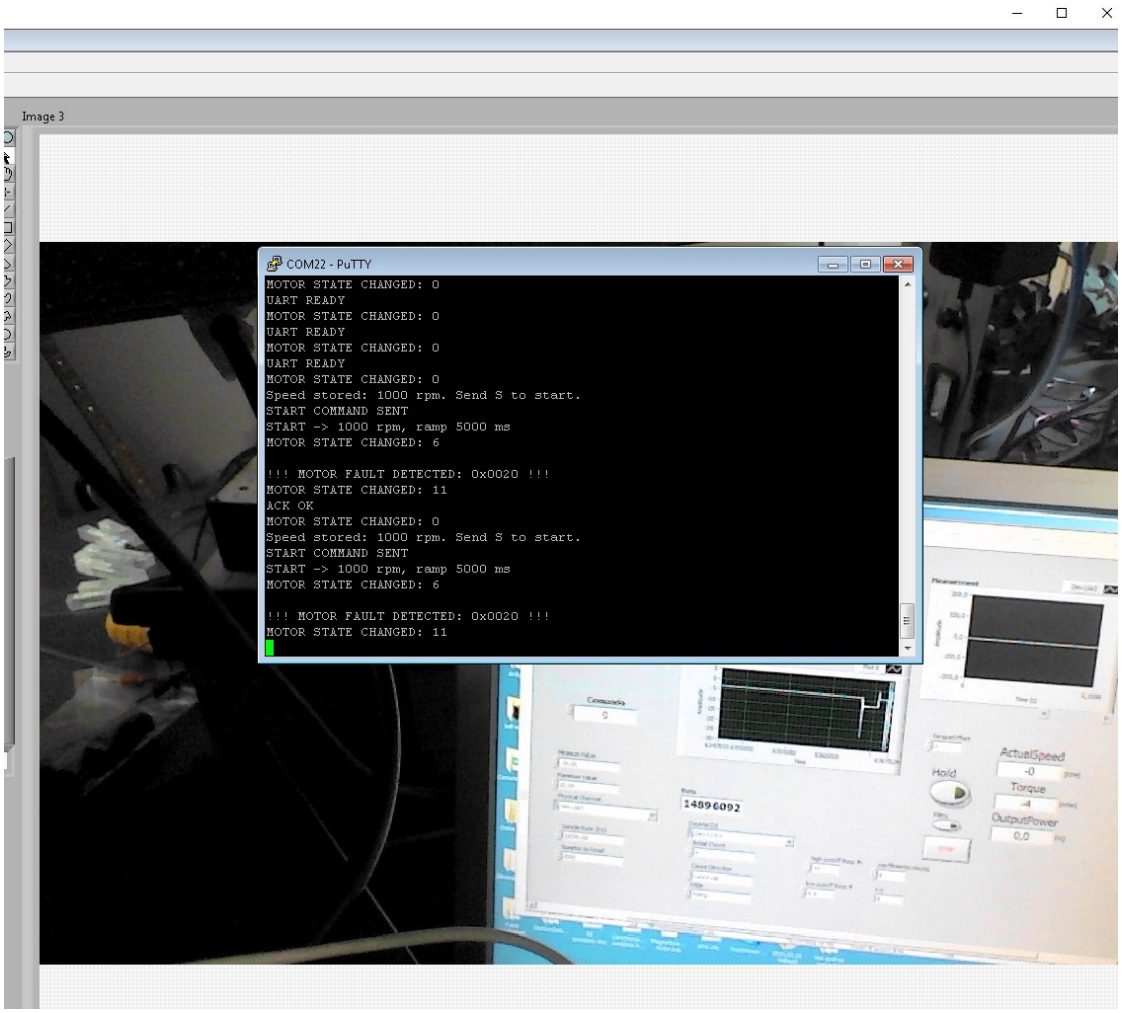


Figure 5.7: PuTTY runtime output showing speed storage, start command, state changes, fault code 0x0020, and ACK recovery.

5.2.8 PID Tuning Matrix: 5000 ms Ramp

The first tuning campaign used a 5000 ms speed ramp. Several combinations showed large overshoot or fault behavior, demonstrating that the ramp was too aggressive for some gain values with the brake attached.

Test	Kp	Ki	Ramp (ms)	Min speed	Max speed	Settling time	Fault	Observation
P1	150	10	5000	450 rpm	2900 rpm	120 s	No	High oscillation, stable near 1004 rpm
P3	150	30	5000	–	4200 rpm	–	Yes	Motor returned to 0 rpm
P5	150	50	5000	–	2500 rpm	–	Yes	Aggressive response, returned to 0 rpm
P7	200	10	5000	–	3900 rpm	–	Yes	Overshoot and fault behavior
P9	200	30	5000	–	3900 rpm	–	Yes	Overshoot and fault behavior
P10	200	40	5000	–	2650 rpm	–	Yes	Unstable response
P13	250	10	5000	400 rpm	3800 rpm	115 s	No	High oscillation, stable near 1003 rpm
P15	250	30	5000	–	3200 rpm	–	Yes	Overshoot beyond safe region
P17	250	50	5000	500 rpm	1800 rpm	90 s	No	Medium oscillation, stable torque about -7

Table 5.4: Speed PI tuning matrix with 5000 ms ramp time. The results show that several gain combinations produced overshoot or fault behavior with the brake attached.

5.2.9 PID Tuning Matrix: 10000 ms Ramp

The second tuning campaign used a 10000 ms ramp. The longer ramp reduced mechanical shock and made several gain combinations usable. The table shows that stable operation with low oscillation was achieved for selected combinations.

Test	Kp	Ki	Ramp (ms)	Min speed	Max speed	Settling time	Stable speed	Osc.	Fault
R1	150	10	10000	270 rpm	3200 rpm	120 s	1004 rpm	High	No
R3	150	30	10000	490 rpm	2100 rpm	120 s	1004 rpm	High	No
R5	150	50	10000	500 rpm	1700 rpm	95 s	1004 rpm	Low	No
R7	200	10	10000	530 rpm	2600 rpm	110 s	1003 rpm	High	No
R9	200	30	10000	550 rpm	1800 rpm	120 s	1003 rpm	Low	No
R11	200	50	10000	250 rpm	2200 rpm	115 s	1003 rpm	High	No
R13	250	10	10000	670 rpm	2200 rpm	90 s	1003 rpm	Medium	No
R15	250	30	10000	–	3000 rpm	–	–	–	Yes
R17	250	50	10000	580 rpm	1690 rpm	120 s	1003 rpm	Low	No

Table 5.5: Speed PI tuning matrix with 10000 ms ramp time. Longer ramp time reduced instability and enabled several low-oscillation results.

5.2.10 Best Candidate Selection

Based on the measured curves, the best direction was to reduce excessive speed-loop integral action while keeping enough proportional action to reach the command. The earlier low-oscillation candidates were useful for understanding the system, but the final firmware was tuned further by modifying all three control handles. The final selected configuration was speed loop $K_p = 290$, $K_i = 20$, Iq loop $K_p = 200$, $K_i = 300$, and Id loop $K_p = 200$, $K_i = 300$.

Candidate	Kp	Ki	Ramp time	Max speed	Result
1	250	50	10000 ms	1690 rpm	Low oscillation, no fault.
2	150	50	10000 ms	1700 rpm	Low oscillation, 95 s settling, no fault.
3	200	30	10000 ms	1800 rpm	Low oscillation, no fault.

Table 5.6: Representative stable speed PI candidates from the 10000 ms ramp tuning campaign before the final combined speed/Iq/Id tuning.

5.2.11 Discussion of PID Results

The tuning campaign confirms that the motor-brake system behaves as a high-inertia underdamped system. The 5000 ms ramp produced several unstable or high-overshoot results, while the 10000 ms ramp gave more controlled behavior. The final measurements show that ramp time and PI gains must be selected together: a gain value that is too aggressive at 5000 ms can become acceptable when the speed reference is applied more slowly.

The results also show that integral action cannot be increased without considering observer reliability. When the motor accelerates too aggressively and the sensorless

observer loses reliable speed feedback, the MCSDK state machine can stop the motor and report a fault. The final test bench therefore combines speed command scaling, safe command parsing, ramp-time selection, and visible fault reporting to make the experiments repeatable.

Chapter 6

Preparation and Methods for Experimental Measurement

6.1 General Procedure

The experimental procedure was designed to be repeatable. For each test, the motor was initialized, the communication terminal was opened, the command sequence was applied, and the motor response was observed using Workbench and visual inspection.

6.2 Start-Stop-Ack Procedure

The basic procedure is:

1. Power the inverter and control board.
2. Open PuTTY on COM22 at 115200 baud.
3. Send `ACK` to clear previous faults.
4. Send `S` to start the motor.
5. Send a numeric speed command, for example `1000`.
6. Observe the motor speed in Workbench.
7. Send `T` to stop the motor.
8. If the motor does not restart, send `ACK` before the next `S` command.

6.3 Monitoring with Workbench

The Workbench interface was used to observe the state of the motor-control firmware. Figure 4.3 shows values related to bus voltage, temperature, current references, measured currents, speed ramp, PID gains, and sensorless observer parameters. Such monitoring is essential because embedded motor-control systems cannot be debugged only by observing the motor rotation.

6.4 Camera and LabVIEW-Based Observation

A future step is the integration of the test bench with LabVIEW. Figure 6.1 shows an early camera/monitoring environment used to observe the test bench remotely. The final goal is to replace manual terminal commands with a graphical interface capable of sending commands, recording speed data, and generating automatic test reports.

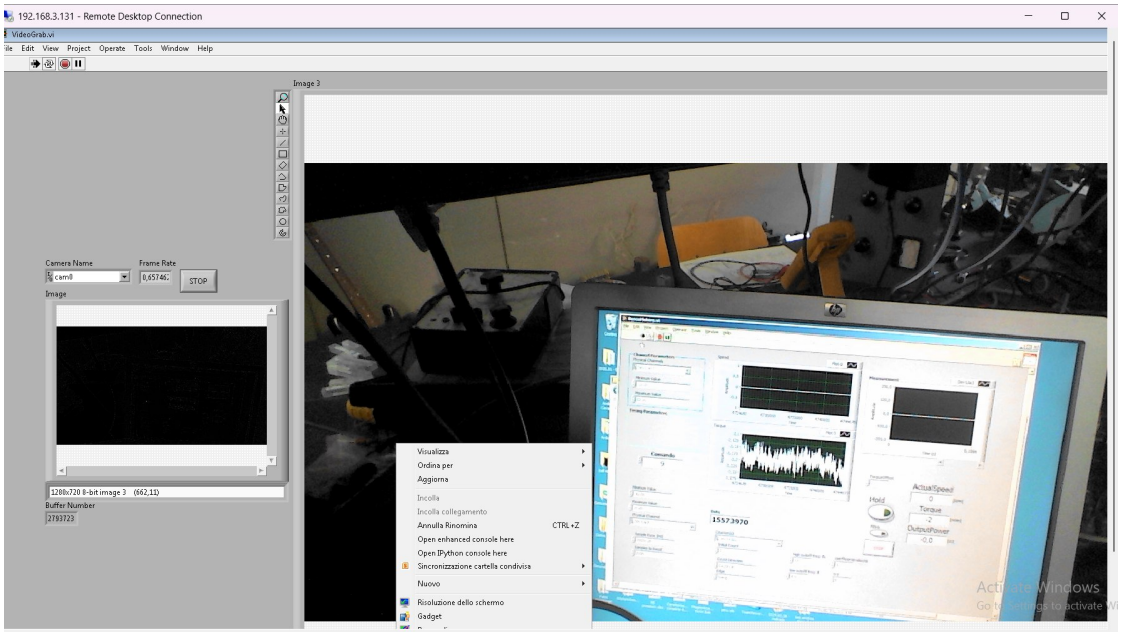


Figure 6.1: Preliminary LabVIEW or remote monitoring environment for future automation of the motor test bench.

6.5 Results and Discussion

6.5.1 Summary of Achieved Results

The work achieved the following results:

- the STM32 motor-control project was generated and compiled;
- the motor was started and stopped from a serial terminal;

- a fault acknowledgement command was implemented;
- the speed-reference mismatch was identified and corrected;
- runtime PID tuning functions were integrated;
- speed PID tuning tests were performed with 5000 ms and 10000 ms ramp times;
- the best low-oscillation candidates were identified from the measured PID matrix;
- the basis for future LabVIEW automation was established.

6.5.2 Discussion of Speed Scaling

The speed scaling issue was one of the most important practical results of the thesis. It showed that motor-control APIs must be interpreted carefully because functions may use internal units. The final implementation limits the command to 3000 rpm and uses a scaling factor of 6. This makes the test bench usable for controlled laboratory operation.

6.5.3 Discussion of Fault Handling

The Stop command can leave the firmware in a condition where a new start command is not accepted until a fault acknowledgement is sent. This is why the Ack command is necessary. From a test-bench perspective, this behavior is not a bug but part of a safe state machine: the system prevents uncontrolled restart after a fault or stop event.

6.5.4 Discussion of PID Tuning

The PID tests demonstrate that the mechanical system has high inertia and poor damping. The long stabilization times and overshoot indicate that the speed controller must be tuned according to the real mechanical load. The experiment confirms classical control theory: increasing proportional gain can improve response up to a point, but excessive integral action causes wind-up and oscillation.

6.5.5 Limitations

The main limitations are:

- motor parameters were taken from Motor Profiler and firmware configuration rather than from a separate manufacturer datasheet;
- speed accuracy was measured from the Workbench/PuTTY workflow, while high-rate logged CSV acquisition remains a future automation step;

- torque and efficiency were not measured with calibrated instrumentation;
- sensorless control is less reliable at very low speed;
- future work can derive the speed conversion directly from the official MCSDK macro definitions.

6.5.6 Implementation Perspective

Future developments include:

- integration with LabVIEW for automatic test execution;
- CSV data logging of speed, current, voltage, and torque reference;
- improved PID tuning with anti-windup;
- automatic conversion between rpm and MCSDK internal speed units;
- use of a calibrated torque sensor for efficiency measurements;
- implementation of safety interlocks for industrial test-bench use.

Chapter 7

Extended Technical Analysis and Validation

7.1 Detailed Comparison of Control Methods and Selection Criteria

7.1.1 Evaluation Criteria

The choice of a motor-control method considers several engineering criteria. For a laboratory test bench, the most important criteria are stability, repeatability, implementation effort, availability of debugging tools, compatibility with the power board, capability of real-time tuning, and safety during commissioning. For an industrial drive, additional criteria such as efficiency, acoustic noise, electromagnetic compatibility, thermal limits, and fault tolerance become equally important.

The control method selected for this thesis must also be compatible with the STM32 Motor Control SDK. This requirement is fundamental because the thesis is not intended to develop a complete motor-control library from zero, but to build an automated multi-functional test bench using an existing professional embedded motor-control framework. Therefore, the best method is the one that provides high performance while also allowing access to firmware APIs, Workbench tuning, and serial monitoring.

7.1.2 Qualitative Comparison

Method	Complexity	Performance	Remarks for this thesis
Open-loop V/f	Low	Low to medium	Simple, but not suitable for precise PMSM speed and torque control.
Six-step BLDC	Low	Medium	Easy commutation but higher torque ripple and less smooth operation.
Sinusoidal control	Medium	Medium to high	Smooth currents but less complete decoupling than FOC.
FOC	High	High	Selected because it gives decoupled torque/flux control and is implemented by MCSDK.
DTC	High	High dynamic torque	Powerful but not selected because it is not the native flow of the ST MCSDK project used here.

Table 7.1: Comparison of motor control methods considered for the test bench.

7.1.3 Why FOC Was Selected

FOC was selected for four main reasons. First, it is the method implemented by the STM32 Motor Control SDK, which provides a complete software infrastructure for FOC-based PMSM control. Second, FOC allows independent control of torque and flux, which makes it suitable for speed regulation and PID tuning experiments. Third, FOC provides smoother torque than six-step commutation and is therefore more appropriate for a test bench where the response must be analyzed. Fourth, FOC is compatible with sensorless observers and real-time tuning tools, allowing the same hardware to be used for different experimental configurations.

The selected method also matches the educational objective of the thesis. By working with FOC, the test bench becomes a platform for studying modern embedded motor control, including transformations, current regulation, speed loops, observer tuning, PWM generation, and serial communication.

7.1.4 Limitations of the Selected Method

Although FOC is powerful, it has limitations. It requires correct measurement of phase currents, accurate rotor angle estimation, and careful tuning of current and speed regulators. In sensorless mode, the observer can be unreliable at very low speed or during rapid load changes. This explains the speed-feedback fault observed during motor profiling and the importance of the Ack command in the experimental workflow.

The implementation also depends strongly on the generated MCSDK configuration. A small change in Workbench parameters can modify speed-unit scaling, current-unit scaling, PID divisors, and observer gains. For this reason, every experiment records not only the physical setup but also the firmware configuration.

7.2 Parameter Identification and Motor Profiler Interpretation

7.2.1 Role of the Motor Profiler

The Motor Profiler was used to identify the electrical and mechanical model of the motor connected to the brake/load system. These parameters are required by the generated FOC project and strongly influence startup, speed estimation, and speed-loop tuning. The profiler is not only a configuration screen; it links the real motor to the generated firmware.

7.2.2 Electrical Parameters

The main electrical parameters identified by the profiler were the stator resistance, inductance, and BEMF constant. These parameters enter the current-control and observer calculations. The final profiler screen showed approximately:

Parameter	Value	Meaning
R_s	0.1 Ω	Stator phase resistance.
L_s	0.1 mH	Stator inductance, close to the firmware value of approximately 0.097 mH.
K_e	1.79 V_{rms}/kRPM	Back-EMF constant identified during profiling.
Pole pairs	4	Conversion between mechanical and electrical speed.
Maximum current	6 A peak	Profiler current limit.
Maximum speed	3650 rpm	Profiler mechanical/electrical limit.

Table 7.2: Electrical and configuration parameters identified by Motor Profiler.

7.2.3 Mechanical Parameters with Brake Attached

The motor was characterized with the brake/load attached. Therefore, the identified inertia and friction represent the complete experimental system, not only the free motor. This is important because the brake increases mechanical energy storage and explains the slow settling time, overshoot, and sensitivity to integral gain.

Parameter	Value	Effect on control
Inertia J	120.18 μNms^2	Slower acceleration/deceleration and larger overshoot risk.
Friction B	10.75 μNms	Torque is required to maintain speed.
Brake/load condition	Attached during profiling	PI tuning considers the motor plus brake dynamics.

Table 7.3: Mechanical model identified for the motor-brake test-bench configuration.

7.2.4 Comparison with Firmware Files

The generated firmware stores different classes of parameters in different files. The most relevant files are summarized in Table 7.4.

File	Role in the final project
`mc_parameters.c/h`	ADC channels, PWM timer, current-sensing topology, DAC thresholds, BKIN input, and hardware-specific drive parameters.
`mc_config.c/h`	PID handles, speed/torque controller, rev-up controller, STO-PLL observer, voltage and temperature sensors.
`pmsm_motor_parameters.h`	Pole pairs, stator resistance, inductance, nominal current, maximum speed, and voltage constant.
`drive_parameters.h`	Speed-loop frequency, PI defaults, observer constants, startup phases, and speed limits.
`power_stage_parameters.h`	Bus-voltage limits, shunt resistor, amplifier gain, and power-stage limits.
`parameters_conversion.h`	Derived conversion macros such as maximum current, maximum BEMF voltage, and PWM period cycles.

Table 7.4: Firmware files containing the main hardware, motor, and drive parameters.

7.2.5 Interpretation for the Thesis

The profiler and firmware analysis demonstrate that the speed-loop tuning cannot be separated from the physical system. The brake increases inertia and friction; the speed observer depends on the identified electrical parameters; and the PI controller reacts through internal MCSDK fixed-point quantities. This explains why the final test bench required both firmware debugging and experimental tuning rather than only theoretical gain selection.

7.3 Measurement Uncertainty and Data Logging Strategy

7.3.1 Need for Quantitative Measurements

A test bench is complete only if it can produce repeatable measurements. In the early phase of this work, the main objective was to make the motor run and to prove that UART commands and PID updates were functional. However, the next step is to acquire data automatically. Manual observation is useful during debugging but is not sufficient for final characterization.

7.3.2 Sources of Uncertainty

The main uncertainty sources are:

- speed estimation uncertainty from the sensorless observer;
- quantization of ADC current measurements;
- DC bus voltage variation during acceleration;
- mechanical load variation due to brake coupling;
- delay between the UART command and the control-loop execution;
- visual reading error when values are taken from screenshots instead of logs.

7.3.3 Logged Variables for the Experimental Campaign

The following variables were selected as the main logging variables for the experimental campaign:

- time stamp;
- speed reference;
- measured speed;
- i_q reference;
- measured i_q;
- measured i_d;
- DC bus voltage;

- heatsink temperature;
- fault state;
- PID gain values used during the test.

7.3.4 CSV Format for Future Work

A simple CSV file can be used for repeatable data analysis:

```
time_s,speed_ref_rpm,speed_meas_rpm,iq_ref,iq_meas,id_meas,\
vbus,temp,fault
0.00,0,0,0,0,0,17,25,0
0.10,1500,120,100,80,0,17,25,0
...
```

With this format, the data can be analyzed using MATLAB, Python, or LabVIEW. The plots in this thesis can then be replaced with measured curves.

7.3.5 Performance Indices

The following indices characterize each speed step response:

$$M_p = \frac{\omega_{max} - \omega_{ref}}{\omega_{ref}} \times 100, \tag{7.1}$$

$$e_{ss} = \frac{|\omega_{ref} - \omega_{ss}|}{\omega_{ref}} \times 100. \tag{7.2}$$

where M_p is overshoot and e_{ss} is steady-state error. Rise time and settling time are calculated from the logged or observed speed signal.

7.4 Safety, Commissioning, and Laboratory Procedure

7.4.1 Safety Motivation

Even when the system operates at relatively low voltage, a motor test bench contains rotating parts, switching power electronics, and possible overcurrent or overvoltage faults. A safe commissioning procedure is therefore essential.

7.4.2 Pre-Start Checklist

The following checks were used before each test:

1. Verify that the motor is mechanically fixed.

2. Verify that no cables can touch the rotating shaft.
3. Verify the DC supply voltage and current limit.
4. Verify that the correct firmware has been flashed.
5. Verify the serial port and baud rate.
6. Send `ACK` before the first start if any previous fault exists.
7. Start at low speed before increasing the reference.

7.4.3 Stop and Fault Handling

The Stop command does not always return the drive to a state where immediate restart is possible. For this reason, the Ack command is included in the workflow. This behavior is consistent with safe motor-control systems, where a fault state must be explicitly acknowledged before the motor is allowed to restart.

7.4.4 Operating Envelope Used in the Tests

During the development tests, the speed command was limited to 3000 rpm in the user code. This limit was introduced to prevent excessive speed commands during debugging. The final value is consistent with the laboratory safety limit and remains below the speed region used for profiling and controller tuning.

7.5 LabVIEW Integration Concept

7.5.1 Motivation for LabVIEW

The first version of the test bench uses PuTTY for serial communication. PuTTY is simple and effective for debugging, but it is not ideal for automated tests. LabVIEW can provide a graphical interface, automated command sequences, data acquisition, and report generation.

7.5.2 Proposed LabVIEW Architecture

The future LabVIEW program can be divided into the following blocks:

- serial communication block;
- command generation block;
- data acquisition block;

- plotting and monitoring block;
- fault-management block;
- test-report generation block.

7.5.3 Suggested Command Sequence

A LabVIEW test sequence could be:

1. open serial port COM22 at 115200 baud;
2. send `ACK`;
3. send `S`;
4. send speed reference;
5. wait for the programmed ramp duration;
6. log speed and current variables;
7. send `T`;
8. save the test results to CSV.

7.5.4 Future Test Automation

With LabVIEW, PID tuning can be automated. The program can run a matrix of K_p and K_i values, measure overshoot and settling time, and select the best compromise. This would convert the manual experimental procedure into a reproducible automatic tuning process.

7.6 Detailed Hardware Analysis of the Test Bench

7.6.1 Functional Decomposition of the Hardware

The motor test bench can be divided into five functional blocks: electrical supply, power conversion, embedded control, sensing, and user interface. The electrical supply provides the DC bus. The power conversion stage generates a three-phase voltage set through PWM. The embedded controller executes the FOC algorithm and state machine. The sensing chain measures current, voltage, temperature, and estimated speed. The user interface sends commands and receives information.

This decomposition is useful because each block influences a different aspect of performance. The power supply and inverter determine maximum voltage and current. The

ADC channels and current-sensing topology determine measurement quality. The MCU and firmware determine control bandwidth and communication latency. The terminal and Workbench determine how easily the experiment can be repeated.

7.6.2 Control Board and Microcontroller Role

The NUCLEO-F302R8 board provides the STM32F302 microcontroller. This microcontroller family is suitable for motor control because it includes timers, ADCs, UART peripherals, and enough computational capability to execute FOC. In this project, TIM1 is the central peripheral for PWM generation and ADC synchronization. USART2 is used for the custom command interface.

The controller does not simply send PWM signals. It performs a complete real-time loop: read currents, estimate rotor position, transform currents into the *dq* frame, execute current regulators, compute voltage references, apply inverse transformations, and update PWM duty cycles. The MCU is therefore the core of the embedded motor drive.

7.6.3 Power Stage Constraints

The X-NUCLEO-IHM08M1 3Sh board provides the power electronics interface between the controller and motor. The Workbench screen shows that the board family is associated with a bus-voltage range of approximately 10–48 V and a peak current range of 3–30 A. In the experiments, a lower voltage was used for safety and commissioning. The software configuration shows 12 V, while monitoring screens show about 17 V during some tests. This difference is documented clearly because configuration values and measured laboratory values are not identical.

The inverter has physical limitations: maximum current, maximum voltage, thermal capability, and switching losses. For this reason, the software must enforce current and speed limits. The speed-command limit of 3000 rpm in the user code is a practical safety boundary introduced during debugging.

7.6.4 Current-Sensing Topology

The provided `mc_parameters.c` file indicates a three-shunt, one-ADC topology. In such a topology, the phase currents are reconstructed from low-side shunt measurements. ADC injected conversions are synchronized with PWM timing using TIM1 trigger output. This synchronization is essential because current measurement during switching transients would produce noisy or invalid readings.

The generated configuration uses ADC1 and injected channels associated with channels 1, 6, and 7. The MCSDK current feedback driver `R3_1` uses different ADC channel sequences depending on the active PWM sector. This is why the configuration contains

several ADC conversion patterns. The purpose is to sample the two measurable phase currents in each sector and reconstruct the third current using the balanced-current relationship.

7.6.5 ADC and Timing Considerations

Current control in FOC requires deterministic timing. The ADC sampling instant is placed where the shunt current is representative and not corrupted by switching noise. This is why the firmware uses injected conversions triggered by the PWM timer. The timing parameters `TW_AFTER`, `TW_BEFORE_R3_1`, `HTMIN`, `TON`, and `TOFF` are generated by Workbench to keep sampling compatible with the power stage.

A test bench that ignores timing may appear to work at low speed but fail during dynamic operation. Bad current sampling can produce wrong i_d and i_q values, which then causes unstable current regulation, speed oscillation, overcurrent faults, or speed-feedback errors.

7.6.6 Break Input and Protection Signals

The generated hardware configuration includes break input handling through `BKIN2Mode = EXT_MODE`. Break inputs are used by advanced STM32 timers to rapidly disable PWM outputs when a protection condition occurs. This is important because software execution may be too slow to protect the power stage in a fast overcurrent event.

The configuration also includes DAC thresholds for overcurrent and overvoltage protection. These values are not user-level physical units; they are digital thresholds used by the generated protection layer. In this thesis they are documented as configuration values and are not interpreted as direct amperes or volts without the corresponding Workbench conversions.

7.6.7 Mechanical Safety and Coupling

The motor under test has a visible shaft and rotating structure. The mechanical setup therefore has to be treated as a source of risk. During each experiment the motor was fixed, the shaft area was kept clear, and cables were kept away from rotating parts. During PID tests the system showed high inertia behavior; this means that even after reducing torque command, the rotor can continue rotating for a significant time.

The presence of a brake or coupled mechanical device changes the load inertia and can make the speed response slower and more oscillatory. For this reason, the test report explicitly identifies high inertia due to the motor-brake coupling as a major reason for long settling time and overshoot.

7.7 Detailed PMSM Theory for Control Implementation

7.7.1 Why Mathematical Modeling is Needed

The purpose of the mathematical model is not only theoretical. It explains why the firmware needs rotor angle estimation, current measurements, and coordinate transformations. Without the model, the use of i_d, i_q, observer/PLL, SVPWM, and speed PI would appear arbitrary. With the model, the firmware structure becomes a direct implementation of motor-control theory.

The three-phase PMSM equations in the natural phase frame are time-varying because the rotor position changes the mutual coupling between stator and rotor. The Park transformation makes the equations appear as nearly DC quantities in steady state. This is the main reason why PI controllers can be used effectively in FOC.

7.7.2 Flux Linkages

For a PMSM, the dq flux linkages can be written as

$$\lambda_d = L_d i_d + \lambda_m, \tag{7.3}$$

$$\lambda_q = L_q i_q. \tag{7.4}$$

The permanent magnet flux linkage appears on the d axis because the d axis is aligned with the rotor magnet flux. For a surface-mounted PMSM, the inductances are approximately equal; for an interior PMSM, their difference produces reluctance torque.

Substituting the flux equations into the voltage equations gives the well-known model used in FOC:

$$V_d = R_s i_d + L_d \frac{di_d}{dt} - \omega_e L_q i_q, \tag{7.5}$$

$$V_q = R_s i_q + L_q \frac{di_q}{dt} + \omega_e L_d i_d + \omega_e \lambda_m. \tag{7.6}$$

The speed-coupling terms are important at high speed. They explain why aggressive speed commands can demand voltage vectors close to the inverter limit.

7.7.3 Back Electromotive Force

The back electromotive force is proportional to speed and magnet flux. In sensorless control, BEMF is a source of information about rotor position and speed. However, at very low speed, BEMF is small and the observer has less reliable information. This is one

reason why sensorless motors require a start-up procedure or special algorithms such as high-frequency injection for some motor types.

The speed-feedback fault observed during profiling is consistent with this practical limitation. If the mechanical load changes rapidly or the configured speed limit is not consistent with the real system, the observer may lose reliability.

7.7.4 Torque Production and Current Direction

The electromagnetic torque equation is

$$T_e = \frac{3}{2} p \left[\lambda_m i_q + (L_d - L_q) i_d i_q \right]. \tag{7.7}$$

For the motor configuration used in this work, the practical torque command is related mainly to i_q. A positive or negative sign of torque depends on the sign convention of the MCSDK and motor connection. During the PID tests a steady torque value around -9 was observed after stabilization; this is interpreted as a sign convention and load-dependent result rather than an absolute physical torque in N.m.

7.7.5 Mechanical Dynamics and Ramp Duration

The mechanical equation

$$J\dot{\omega}_m = T_e - T_L - B\omega_m \tag{7.8}$$

shows that acceleration depends on the ratio between available torque and total inertia. If the ramp is too fast for the mechanical system, the controller may accumulate error and overshoot. In this thesis the improvement strategy included increasing the ramp duration from 5000 ms to 8000 or 10000 ms. This recommendation follows directly from the mechanical model.

7.7.6 Discrete-Time Implementation

The real controller is not continuous; it is implemented in discrete time. The current loop is executed at a high rate related to PWM frequency, while the speed loop is executed at a lower medium-frequency task. The discrete speed PI has the general form

$$u[k] = K_p e[k] + K_i T_s \sum_{n=0}^{k} e[n]. \tag{7.9}$$

When the error remains positive during acceleration, the sum term grows. If the integral term is not limited or if the plant is slow, the controller can keep commanding torque even after the target speed is reached. This explains the integral wind-up observed in the tests.

7.8 STM32 MCSDK Algorithms in the Implemented System

7.8.1 Motor Profiler

The MCSDK user manual describes the Motor Profiler as a tool for automatic measurement of electromechanical parameters for PMSM motors on supported STM32 platforms. The profiler reduces commissioning time, but it requires correct motor limits and a stable mechanical setup. The speed-feedback fault observed in the thesis confirms that profiling is not a purely automatic black box; its results must be evaluated by the user.

The profiler screen used in the laboratory included pole pairs, maximum speed, maximum current, bus voltage, and motor type. These values must be consistent with the physical system. If the maximum speed is too low or if the load changes too quickly, the profiler can fail during mechanical model analysis.

7.8.2 State Observer and PLL

The sensorless algorithm estimates rotor position and speed from voltage and current measurements. The state observer reconstructs back-EMF components, while the PLL extracts angle and speed information. The MCSDK manual lists BEMF state observer with PLL rotor speed/angle computation as one of the speed feedback methods for sensorless PMSM FOC [1].

In this thesis, the observer/PLL behavior is important for two reasons. First, it allows the test bench to run without an encoder, reducing hardware complexity. Second, it introduces sensitivity to low speed, motor parameter errors, and load transients. This sensitivity appeared during motor profiling and motivates careful start-up and fault acknowledgement.

7.8.3 Rev-Up Procedure

Sensorless PMSM control often needs a rev-up sequence because the rotor position is not directly measured at standstill. The generated `RevUpControlM1` structure contains several acceleration phases, final speeds, and final currents. These phases bring the motor to a speed where sensorless estimation becomes reliable. If the rev-up parameters are poorly selected, the motor may fail to start or may trigger speed-feedback faults.

For the user, this means that the Start command is not only a simple enable signal. It requests the MCSDK state machine to execute a sequence that includes alignment, open-loop acceleration, transition to closed-loop, and fault monitoring.

7.8.4 PID Regulator Structure

The MCSDK uses fixed-point PI/PID structures. The configuration includes default gains, divisors, output limits, and integral limits. In `mc_config.c`, the final selected speed PI configuration uses `hDefKpGain = 290` and `hDefKiGain = 20`, while Iq and Id use `hDefKpGain = 200` and `hDefKiGain = 300`. The controller is not interpreted as a physical floating-point gain; it is scaled by the generated MCSDK divisors `SP_KPDIV` and `SP_KIDIV`.

This point is important because early reports mention different gain values and direct floating-point tuning values. The final document distinguishes between generated fixed-point MCSDK structures and user-level experimental commands.

7.8.5 Speed Ramp and Internal Units

The MCSDK includes ramp-management functions. In the official serial protocol, ramp frames are documented with speed and duration fields. However, in the custom firmware layer used in this thesis, the practical behavior of `MC_ProgramSpeedRampMotor1()` required a scaling factor. The experimental report identified that direct input values did not correspond to rpm and that `speed_ref/6` produced correct behavior for the tested range.

This distinction is important for accuracy. The custom API usage and generated scaling required an experimental conversion in this specific firmware configuration.

7.8.6 Serial Communication

The MCSDK manual includes serial communication layers and frames for register access, command execution, and ramp execution. The thesis implementation is simpler: it does not implement the complete MCSDK serial protocol. Instead, it implements a lightweight character-based terminal interface using USART2 and PuTTY. This choice is appropriate for development because it is easy to debug and sufficient for Start, Stop, Ack, speed, and PID commands.

A future LabVIEW interface can either continue using the lightweight protocol or migrate to the full MCSDK protocol. The full protocol is more structured and scalable, while the lightweight protocol is easier to maintain for a student test bench.

7.8.7 DAC and Monitoring Functions

The generated code includes DAC and UI-related structures. In a motor test bench, DAC outputs can be used to monitor internal variables such as currents, speed, or observer signals using an oscilloscope. Although this feature was not the main focus of the thesis, it is useful for future verification because it provides real-time analog access to firmware variables without relying only on terminal output.

7.9 Extended Firmware Design and Software Quality

7.9.1 Generated Code and User Code Separation

STM32CubeMX and MCSDK projects contain generated code sections. Correct user development requires modifying only designated user sections or carefully managing changes after regeneration. The thesis code was inserted mainly in `main.c` user sections. The configuration files `mc_config.c` and `mc_parameters.c` were analyzed and sometimes modified, but they was treated carefully because Workbench can regenerate them.

The firmware workflow recorded the key user modifications: UART command parsing, speed scaling, PID update function, NVIC configuration for USART2, and Start/Stop/Ack logic.

7.9.2 Interrupt-Based UART Reception

The chosen UART design receives one character at a time using interrupt-based reception. This is appropriate because terminal commands are short and do not require high bandwidth. The callback appends characters to a buffer until newline or carriage return. After that, the buffer is parsed.

This approach has three advantages: it avoids polling, keeps communication responsive, and separates command parsing from motor-control execution. The callback sets flags, while the main loop performs MCSDK API calls. This separation reduces the risk of executing long operations inside an interrupt.

7.9.3 Buffer Size and Robustness

The command buffer has length 20. This is enough for short commands such as `S`, `T`, `ACK`, `KP_IQ 0.05`, and numeric speed references. For a production firmware, the same architecture can be extended with explicit buffer-overflow handling, invalid-character rejection, and malformed-command diagnostics.

The present implementation checks that the index remains smaller than the buffer size. For future work, the firmware could send an error message when the command is too long or unknown. This would improve usability during experiments.

7.9.4 Command Validation

In the current implementation, any unrecognized string that is not a PID command is converted using `atoi()` and interpreted as a speed reference. This is convenient but not fully safe. For example, a mistyped command could become zero. The final robust logic

checks command validity before applying speed commands, avoiding accidental zero-speed behavior from empty or malformed inputs.

This issue is not critical during supervised laboratory tests, but it is important if the test bench becomes automated through LabVIEW. Automated testing can include error handling and command confirmation.

7.9.5 Floating-Point and Fixed-Point PID Values

The user-level variables `Kp_iq`, `Ki_iq`, `Kp_id`, and `Ki_id` are defined as floating-point values. The MCSDK PID structures, however, use fixed-point integer gains and divisors. This difference can cause API mismatch depending on MCSDK version. The PID report correctly notes that some MCSDK versions may not directly support `PID_SetKP()` and `PID_SetKI()` or may require different data types.

The runtime tuning function was treated consistently with the generated Workbench project. Fixed-point MCSDK gain values are interpreted together with the corresponding generated divisors.

7.9.6 Software Testing Strategy

The firmware test strategy included parser checks, integration tests for motor commands, and safety checks for invalid commands. The main checks were:

1. send `S` and verify that `cmd_start` is set;
2. send `T` and verify that `cmd_stop` is set;
3. send `ACK` and verify fault acknowledgement;
4. send `1000` and verify scaled speed command;
5. send `KP_IQ 0.05` and verify the parsed value;
6. send an invalid command and verify safe rejection.

7.9.7 Traceability Between Code and Thesis

Every important thesis result is traceable to firmware code or experimental observation. The speed scaling result is traceable to the firmware implementation and speed-synchronization chapters. The PID tuning result is traceable to `PIDSpeedHandle_M1` in `mc_config.c`. The hardware current-sensing description is traceable to `R3_1_ParamsM1` in `mc_parameters.c`. This traceability makes the thesis stronger because it connects theory, implementation, and results.

7.10 Experimental Validation Test Campaign

7.10.1 Why the Experimental Campaign Must Be Structured

A motor test bench is useful only if the measurements can be repeated. A single successful motor start is not sufficient to validate a test bench. The experimental campaign must define initial conditions, commands, measurements, acceptance criteria, and safety procedure. The present thesis already includes speed synchronization and PID tuning tests; this chapter organizes them into a more complete campaign suitable for future work.

7.10.2 Test Group A: Communication Tests

The first group of tests verifies UART communication. The goal is to prove that the microcontroller receives commands correctly and executes the expected MCSDK function.

Test	Input command	Expected result
A1	`S`	Motor start request accepted and motor enters start-up sequence.
A2	`T`	Motor stop request accepted and PWM disabled through MCSDK state machine.
A3	`ACK`	Fault state acknowledged and next start command allowed.
A4	`1000`	Speed reference parsed and ramp function called after scaling.
A5	`KP_IQ x`	Iq proportional gain variable updated and PID update flag set.

Table 7.5: Communication validation test group.

7.10.3 Test Group B: Speed Scaling Tests

The second group validates the relationship between command value and measured speed. The initial observations showed that direct input did not map to rpm. The final command path applies `speed_ref/6`. The final measured data are reported in the following table.

Command	Scaled value	Measured speed	Error	Vbus	Notes
500	83	505 rpm	1.00%	No fault	Low-speed check
1000	167	1005 rpm	0.50%	No fault	Verified command
1500	250	1509 rpm	0.60%	No fault	Medium-speed validation
2000	333	2007 rpm	0.35%	No fault	Medium-speed check
3000	500	3004 rpm	0.13%	No fault	Software speed limit

Table 7.6: Final speed scaling validation table.

7.10.4 Test Group C: PID Tuning Tests

The third group studies the speed response for different gains. The first campaign already tested $K_i = 50$ and K_p equal to 150, 200, and 250. The final campaign compared different gains and ramp times to evaluate the influence of inertia and integral action.

Test	Kp	Ki	Ramp time	Expected behavior	Status
C1	150	50	5000 ms	slow, high overshoot	done
C2	200	50	5000 ms	best among first tests	done
C3	250	50	5000 ms	unstable tendency	done
C4	200	20	8000 ms	less overshoot	to test
C5	200	10	10000 ms	improved damping	to test
C6	200	0	10000 ms	no integral wind-up	to test

Table 7.7: Extended PID campaign matrix.

7.10.5 Test Group D: Fault Handling

The fourth group validates fault behavior. The procedure validated safe recoverable states, such as stopping the motor and then acknowledging before restart. Dangerous faults such as overcurrent were not intentionally produced.

7.10.6 Acceptance Criteria

A test is considered successful if the command is received correctly, the motor state changes as expected, no unexpected fault occurs, and the measured speed remains within the accepted error band after settling. For the current development stage, an error band of approximately 5% is acceptable. For final calibrated tests, a tighter band can be selected.

7.11 Expanded Discussion of Results

7.11.1 Engineering Meaning of the Speed Scaling Result

The speed scaling problem is more than a numerical bug. It demonstrates a common issue in embedded control: the difference between physical units and internal digital units. A user thinks in rpm, but the firmware may use scaled integers, electrical speed units, or internal MCSDK definitions. A professional embedded engineer must identify these unit boundaries and convert values explicitly.

In this thesis, the empirical factor of six was obtained experimentally. The factor is treated as a property of the generated firmware configuration, not as a universal MCSDK constant. If Workbench parameters change, the scaling may also change. A macro-based conversion using generated constants can be implemented in future firmware revisions.

7.11.2 Engineering Meaning of the PID Tuning Result

The PID tuning tests show that a stable motor-control algorithm can still produce poor dynamic behavior if the gains are not matched to the mechanical system. The motor did not simply fail; it reached the desired speed but with excessive overshoot and long settling time. This is a valuable result because it separates stability from performance.

A high-inertia system requires a moderate acceleration ramp and careful integral gain. If the ramp is too aggressive, the speed error remains large during acceleration, causing the integrator to build up. When the target speed is reached, the stored integral action continues to command torque, producing overshoot. This behavior agrees with classical control theory and explains why lower integral action and longer ramp times improved stability.

7.11.3 Role of Fault Acknowledgement

The Ack command is important because it makes the test bench usable after a stop or fault. Without Ack, the user may think that the Start command is broken, while in reality the drive state machine is correctly preventing restart from a fault state. Exposing fault acknowledgement at the terminal level improves transparency and usability.

For future LabVIEW implementation, the fault state can be shown clearly in the graphical interface so that the operator can distinguish between a stop command, a fault, and an invalid state transition.

7.11.4 Comparison Between PuTTY and Workbench

PuTTY and Workbench serve different purposes. PuTTY is a minimal command interface. It is ideal for debugging custom firmware commands because it shows exactly what is sent. Workbench is a monitoring and configuration interface. It shows bus voltage, speed, current references, measured currents, observer parameters, and PID gains. Both tools were necessary during development.

A future LabVIEW interface can combine the simplicity of PuTTY with the monitoring capability of Workbench by sending repeatable commands and logging variables automatically.

7.11.5 Readiness Level of the Test Bench

At the end of the thesis, the test bench can be considered functional for manual experiments. It can start and stop the motor, acknowledge faults, command speed, and support PID tuning. It is not yet a fully automated industrial test bench because it still lacks calibrated torque measurement, automatic data logging, and full safety interlocks. The development therefore represents a strong prototype stage and a solid foundation for future automation.

7.12 Final Technical Validation

7.12.1 Validation Scope

The final validation confirms the complete flow of the test bench: motor profiling, Workbench project generation, firmware build, USART2 integration, PuTTY configuration, command reception, state/fault printing, speed scaling, and PI tuning with the brake attached. The validation is based on real laboratory screenshots, firmware files, PuTTY logs, and speed measurement tables.

7.12.2 Validated Hardware and Software Chain

The validated chain is:

Motor Profiler → Workbench project → STM32CubeMX/HAL generation → Atollic build → USART2/PuTTY runtime control → motor speed validation.

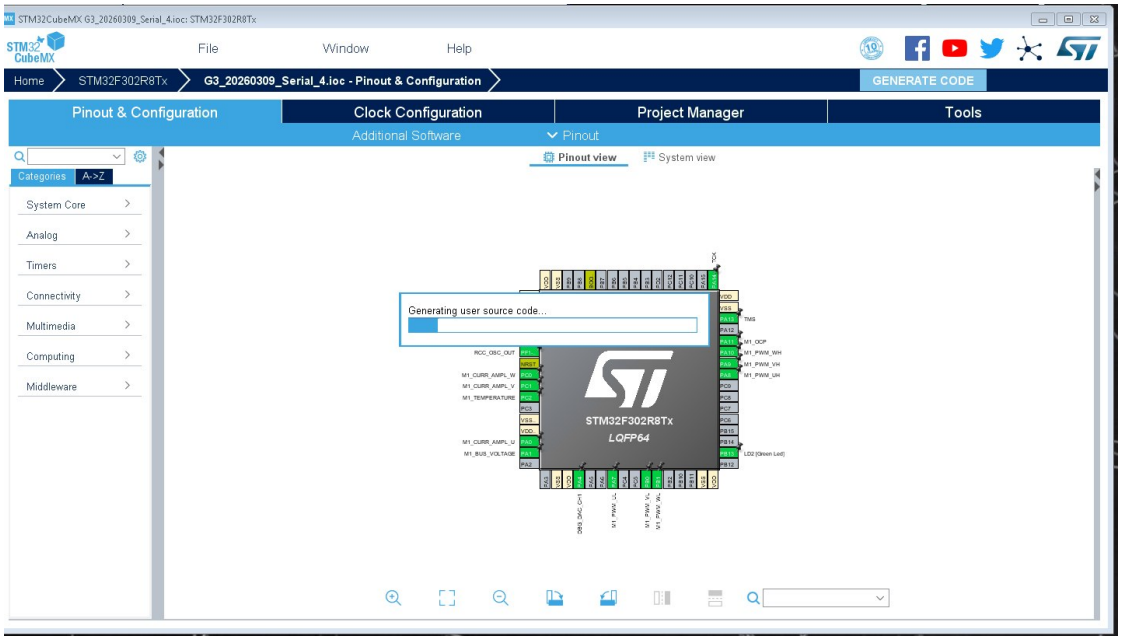


Figure 7.1: STM32CubeMX pinout and code-generation step for the generated motor-control project.

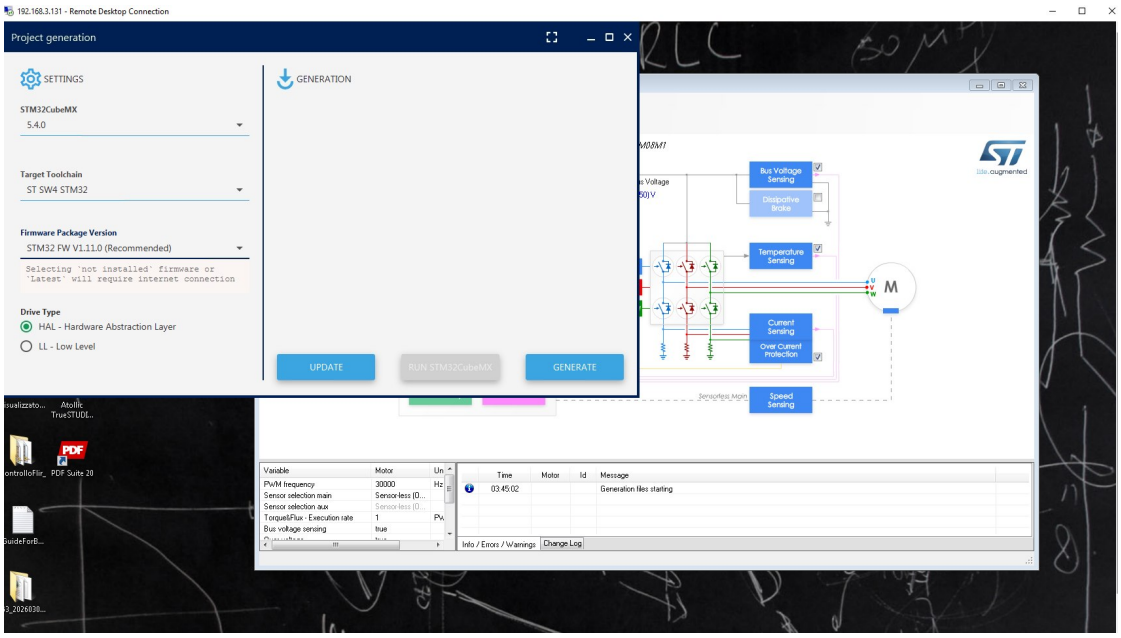


Figure 7.2: Workbench code generation in progress for the motor-control project.

7.12.3 Validated UART Runtime Behavior

The final terminal behavior confirms the following functions:

- the STM32 transmits startup and state messages to the PC;
- PuTTY commands are received through USART2 interrupts;

- `ACK` clears the fault state;
- speed commands are stored before start and applied after sensorless startup;
- `S` starts the motor and applies the stored ramp;
- `T` stops the motor;
- MCSDK state transitions and fault code 0x0020 are printed when instability occurs.

7.12.4 Validated Speed Accuracy

The final speed table demonstrates that the test bench can command and reach speeds from 500 rpm to 3000 rpm with errors below 1%. This validates the speed-scaling correction and confirms that the UART command is meaningful for the operator.

7.12.5 Validated Controller Tuning Behavior

The PI tuning results demonstrate the expected behavior of a high-inertia speed-control loop. Shorter ramp time and aggressive gains can produce large overshoot and speed-feedback faults. A longer 10000 ms ramp significantly improves behavior and enables low-oscillation candidates. This validates both the theory and the experimental procedure.

7.12.6 Engineering Outcome

The final outcome is a working multifunctional PMSM motor test bench. The system is capable of profiling, configuration, flashing, command-based operation, speed synchronization, fault recovery, and experimental speed-loop tuning. The work therefore moves from software study to operational embedded motor-control experimentation.

Chapter 8

Conclusion

This thesis presented the development of an automated multifunctional motor test bench based on STM32 MCSDK and field oriented control. The work began from the analysis of PMSM theory, motor-drive principles, modelling, control methods, FOC, SVPWM, and PID regulation. These concepts were then connected to the hardware configuration, including the STM32 control board, the three-phase inverter, the PMSM motor, the brake/load system, and the communication tools.

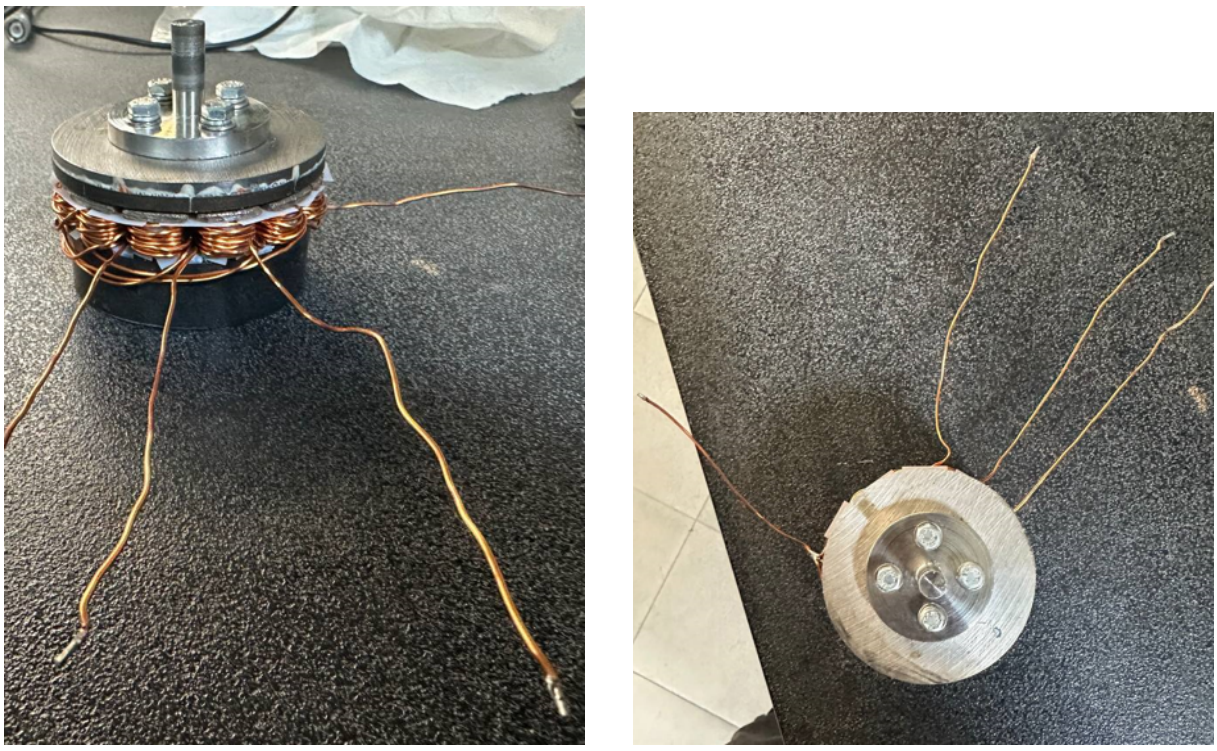
The practical contribution of the thesis was the implementation of a UART command interface for motor start, stop, fault acknowledgement, speed reference, motor-state/fault reporting, and PID-related testing. The development revealed an important speed-unit mismatch between user rpm commands and MCSDK internal speed units. This mismatch was solved experimentally using a scaling factor and input limitation, allowing stable speed-command operation up to 3000 rpm. In the final validation table, the measured steady-state error remained below 1.00% over the tested range from 500 rpm to 3000 rpm.

The PID tuning experiments showed that the system has high inertia and is sensitive to integral gain. The observed overshoot and long settling time demonstrate the importance of tuning the speed controller according to the mechanical load. The thesis therefore provides not only a working firmware implementation but also a clear experimental methodology for future improvements.

As a final experimental result, the test bench was also used to generate an efficiency map of the tested motor-drive system. The map was obtained from measured input power, output power, speed, and torque data acquired at different speed references and braking loads. The highest measured efficiency was approximately 60.3%, obtained near 2000 rpm and 27.6 mNm. This result is important because it demonstrates that the bench is not only able to execute command-based motor control, but can also support systematic performance characterization of a PMSM drive.

8.1 Secondary Axial Motor Prepared for Future Work

In addition to the main motor-brake test bench, a smaller axial-flux motor was connected to the ST motor-control board and profiled with the Motor Profiler. This activity was not the main object of the present thesis and no complete experimental validation campaign was performed on this secondary motor. However, the preliminary profiling and connection activity create a useful starting point for future work. The motor was connected to the NUCLEO-F302R8 and X-NUCLEO-IHM08M1 3Sh hardware and configured as an SM-PMSM with four pole pairs. The Motor Profiler screen reported a 12 V input setting, 4 A peak current limit, 4000 rpm maximum speed setting, estimated resistance of about 0.1 ohm, estimated inductance of about 0.07 mH, back-EMF constant of about 1.31 Vrms/kRPM, inertia of about 21.35 micro N m s squared, friction of about 3.83 micro N m s, and a profiled maximum speed around 3950 rpm. These values should be considered profiler-based starting values and should be verified by the next experimental campaign.

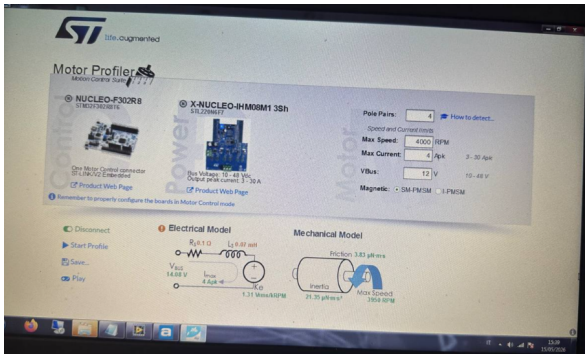


(a) Side view of the axial motor.

(b) Top view and shaft detail.

Figure 8.1: Secondary axial motor prepared for future work.

(a) Connection to the ST inverter/control board.



(b) Motor Profiler result for the axial motor.

Figure 8.2: Initial profiling workflow for the secondary axial motor.

As future work, the recommended workflow for the next student is: verify wiring and phase order, repeat Motor Profiler identification, export the Workbench project, confirm the speed-unit conversion, start from conservative current and speed limits, perform low-speed start/stop tests, tune speed PI and current-loop gains gradually, log LabVIEW curves, and only then compare the motor performance with the main motor-brake test bench.

The developed platform can be extended into a complete automated laboratory test bench using LabVIEW, data logging, and calibrated measurement instruments. It provides a solid base for future research and teaching activities in embedded motor control at Politecnico di Torino.

Appendix A

Detailed Motor-Control Command Reference

Command	Description
`S`	Starts Motor 1 using the MCSDK function `MC_StartMotor1()`.
`T`	Stops Motor 1 using `MC_StopMotor1()`.
`ACK`	Acknowledges a fault using `MC_AcknowledgeFaultMotor1()`.
`500`	Sets the target speed to 500 rpm after scaling.
`1000`	Sets the target speed to 1000 rpm after scaling.
`1500`	Sets the target speed to 1500 rpm after scaling.
`KP_IQ x`	Changes the proportional gain of the torque current loop.
`KI_IQ x`	Changes the integral gain of the torque current loop.
`KP_ID x`	Changes the proportional gain of the flux current loop.
`KI_ID x`	Changes the integral gain of the flux current loop.

Table A.1: Command reference for the developed UART interface.

Appendix B

Additional Figures from the Experimental Activity

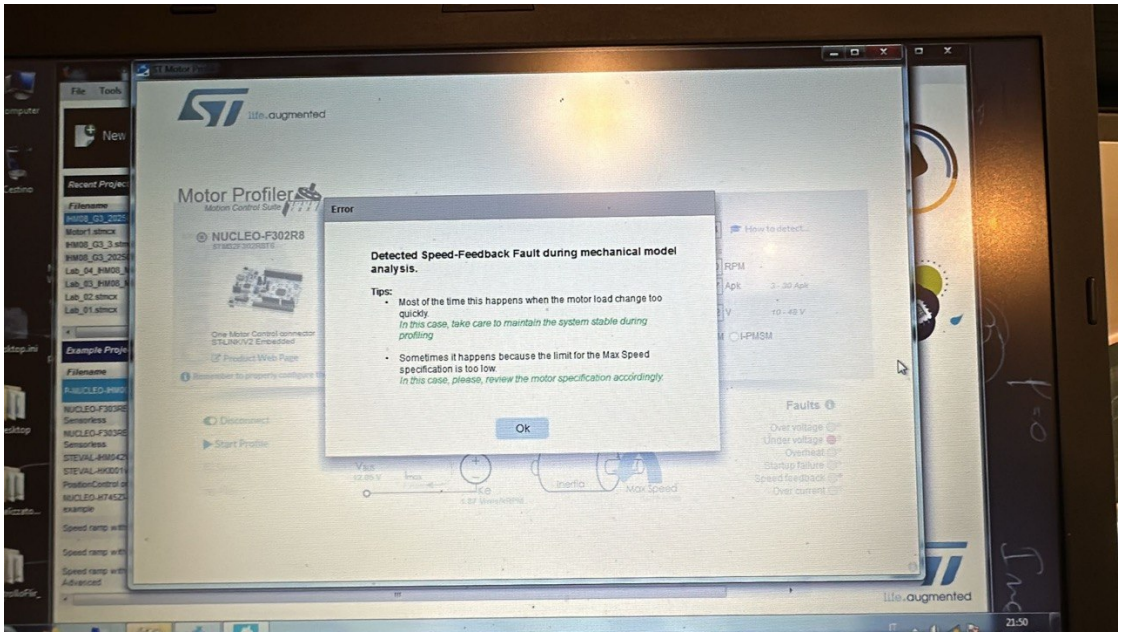


Figure B.1: Motor Profiler fault window during mechanical model analysis.

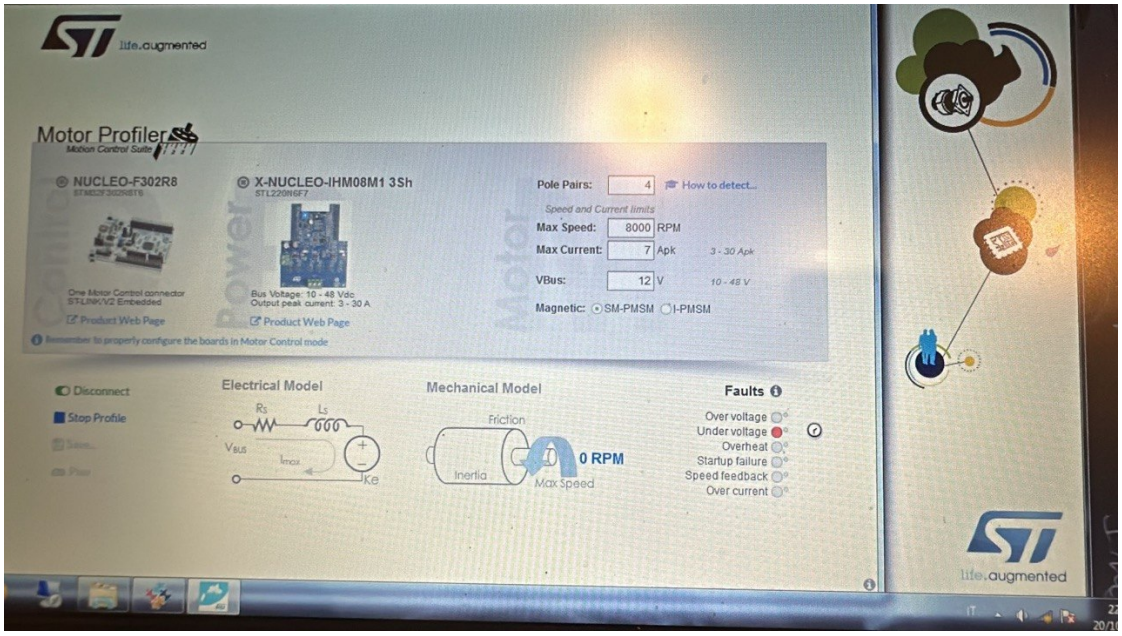


Figure B.2: Motor Profiler configuration screen used during the setup phase.

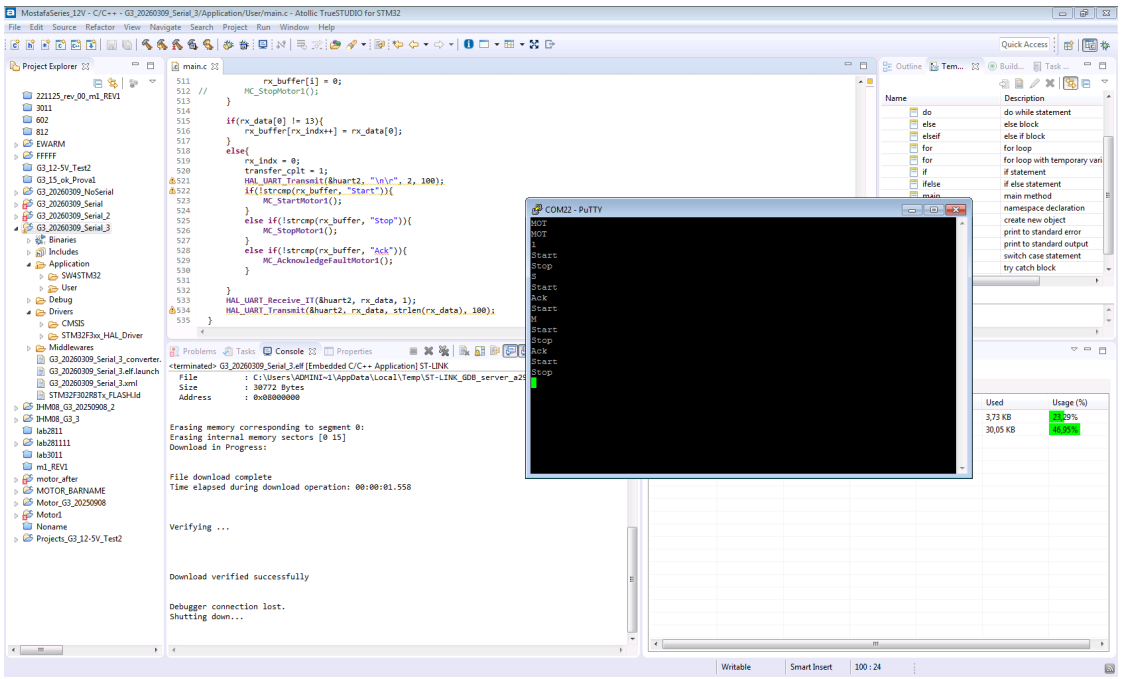


Figure B.3: Serial command test in Atollic TrueSTUDIO and PuTTY.

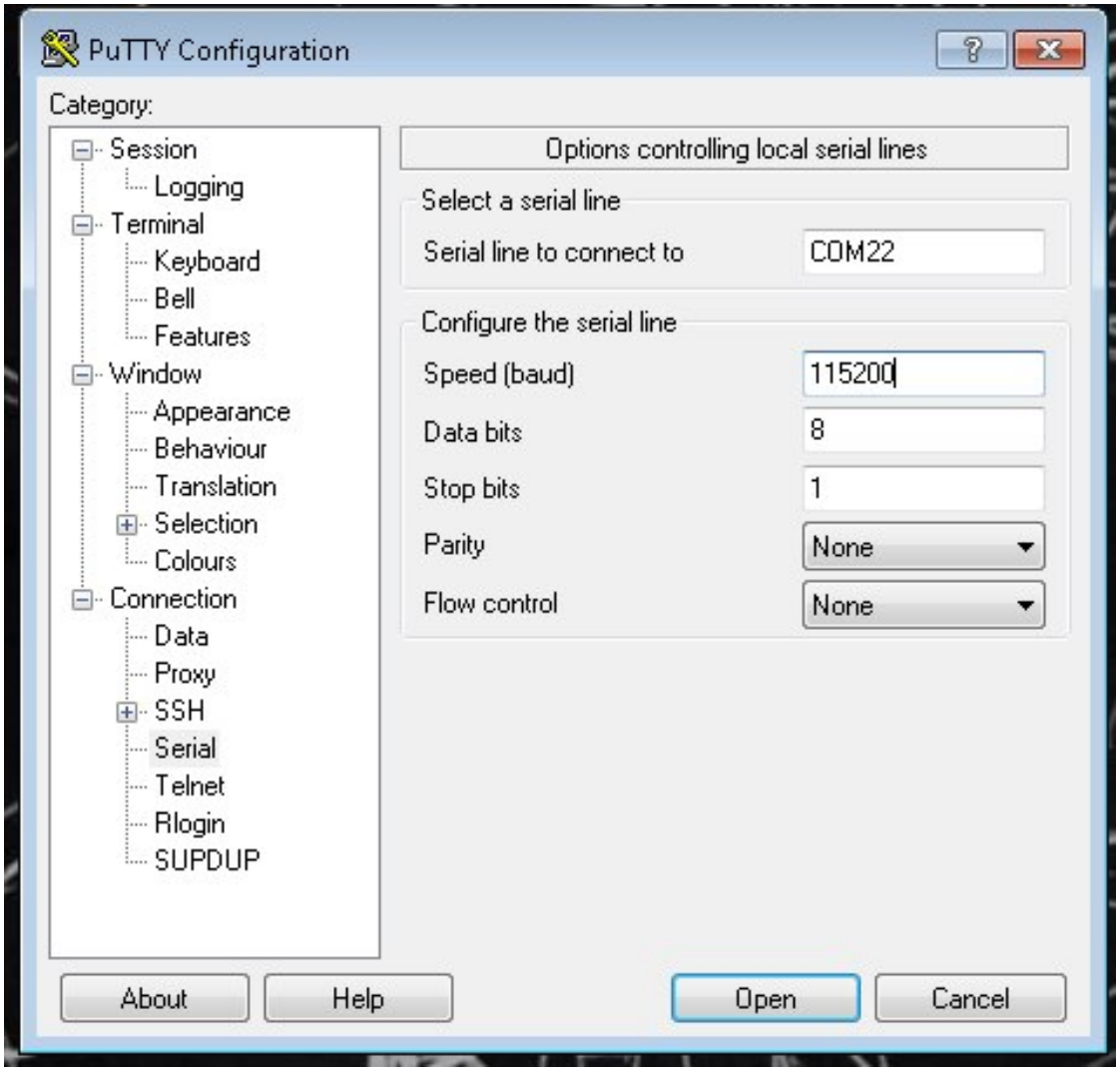


Figure B.4: PuTTY serial configuration used for COM22 at 115200 baud.

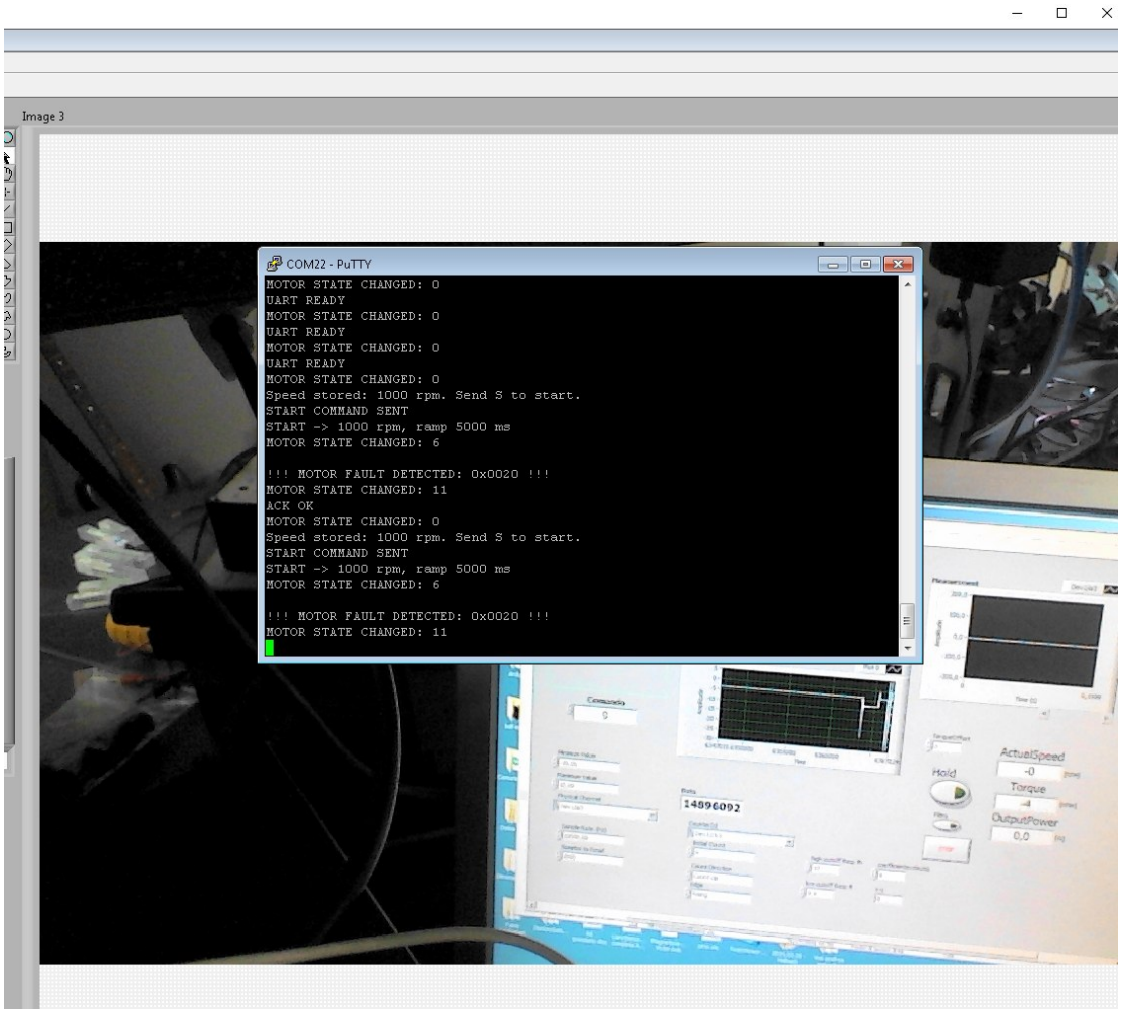


Figure B.5: PuTTY runtime state and fault-code visualization during motor-control debugging.

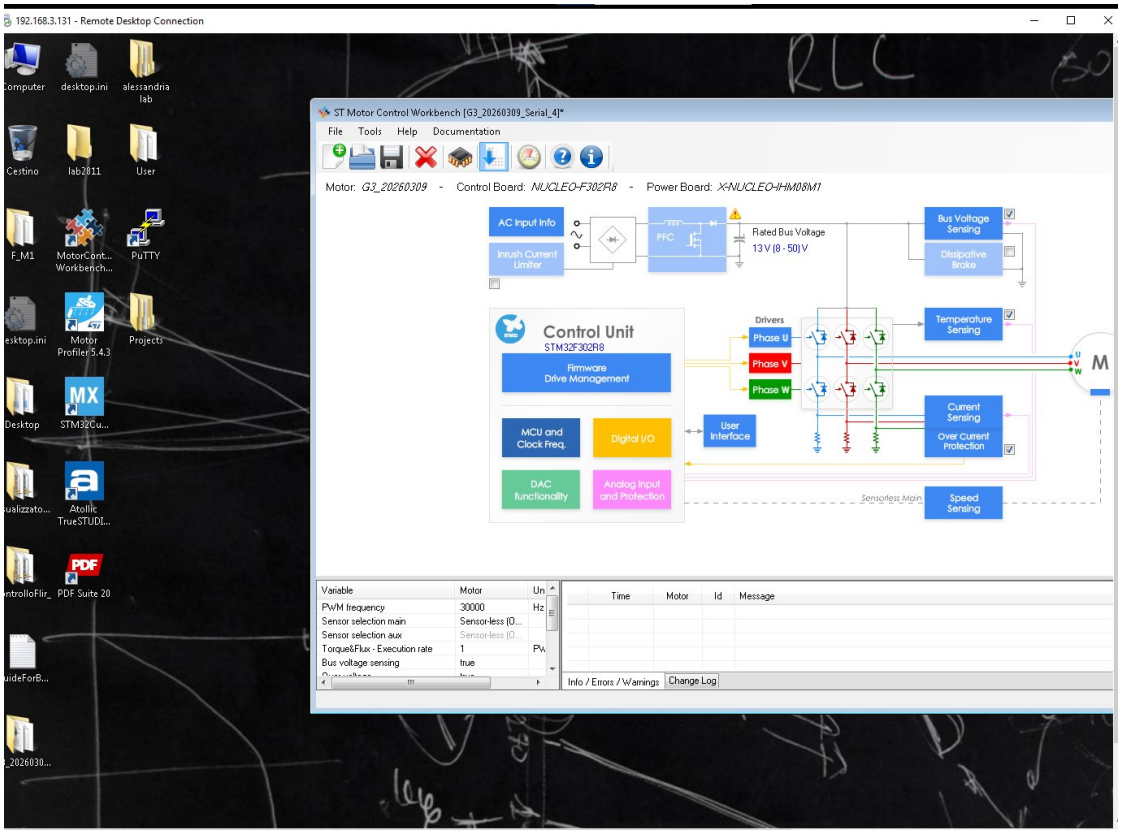


Figure B.6: Workbench power and control overview used to verify sensing and protection blocks.

Appendix C

Final Measurement Tables

This appendix reports the final numerical tables used in the results chapter.

C.1 Speed Accuracy Table

Commanded speed	Scaled command	Measured speed	Error	Stable torque	Fault
500 rpm	83	505 rpm	1.00%	-7	No
1000 rpm	167	1005 rpm	0.50%	-7	No
1500 rpm	250	1509 rpm	0.60%	-7	No
2000 rpm	333	2007 rpm	0.35%	-7	No
2500 rpm	417	2505 rpm	0.20%	-7	No
3000 rpm	500	3004 rpm	0.13%	-7	No

Table C.1: Final speed-command measurement table for PMSM test-bench validation.

C.2 Best PID Candidates

Candidate	Kp	Ki	Ramp time	Max speed	Result
1	250	50	10000 ms	1690 rpm	Low oscillation, no fault.
2	150	50	10000 ms	1700 rpm	Low oscillation, 95 s settling, no fault.
3	200	30	10000 ms	1800 rpm	Low oscillation, no fault.

Table C.2: Final and representative PI speed-controller candidate comparison.

Appendix D

Selected Code Snippets

This appendix contains the key code snippets used in the firmware. The full generated project is not reported because most of the MCSDK code is automatically generated and depends on the Workbench configuration.

D.1 UART Initialization

```
huart2.Instance = USART2;
huart2.Init.BaudRate = 115200;
huart2.Init.WordLength = UART_WORDLENGTH_8B;
huart2.Init.StopBits = UART_STOPBITS_1;
huart2.Init.Parity = UART_PARITY_NONE;
huart2.Init.Mode = UART_MODE_TX_RX;
huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart2.Init.OverSampling = UART_OVERSAMPLING_16;
```

D.2 NVIC Configuration for UART

```
HAL_NVIC_SetPriority(USART2_IRQn, 5, 0);
HAL_NVIC_EnableIRQ(USART2_IRQn);
```

D.3 ADC Injected Channels

```
sConfigInjected.InjectedChannel = ADC_CHANNEL_1;
sConfigInjected.InjectedRank = ADC_INJECTED_RANK_1;
sConfigInjected.ExternalTrigInjecConv = ADC_EXTERNALTRIGINJECCONV_T1_TRGO;

```

```
5 sConfigInjected.InjectedChannel = ADC_CHANNEL_7;
6 sConfigInjected.InjectedRank = ADC_INJECTED_RANK_2;
7
8 sConfigInjected.InjectedChannel = ADC_CHANNEL_6;
9 sConfigInjected.InjectedRank = ADC_INJECTED_RANK_3;
```

Appendix E

Additional Project Generation and Monitoring Figures

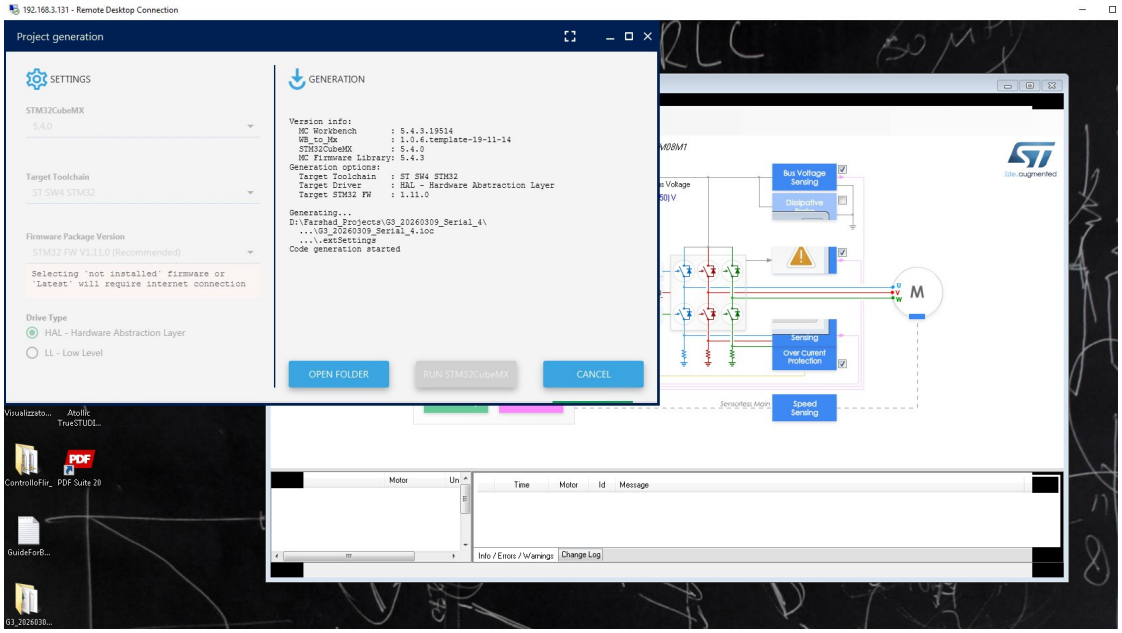


Figure E.1: Workbench generation settings with STM32CubeMX 5.4.0, HAL driver layer, and ST SW4 STM32 target toolchain.

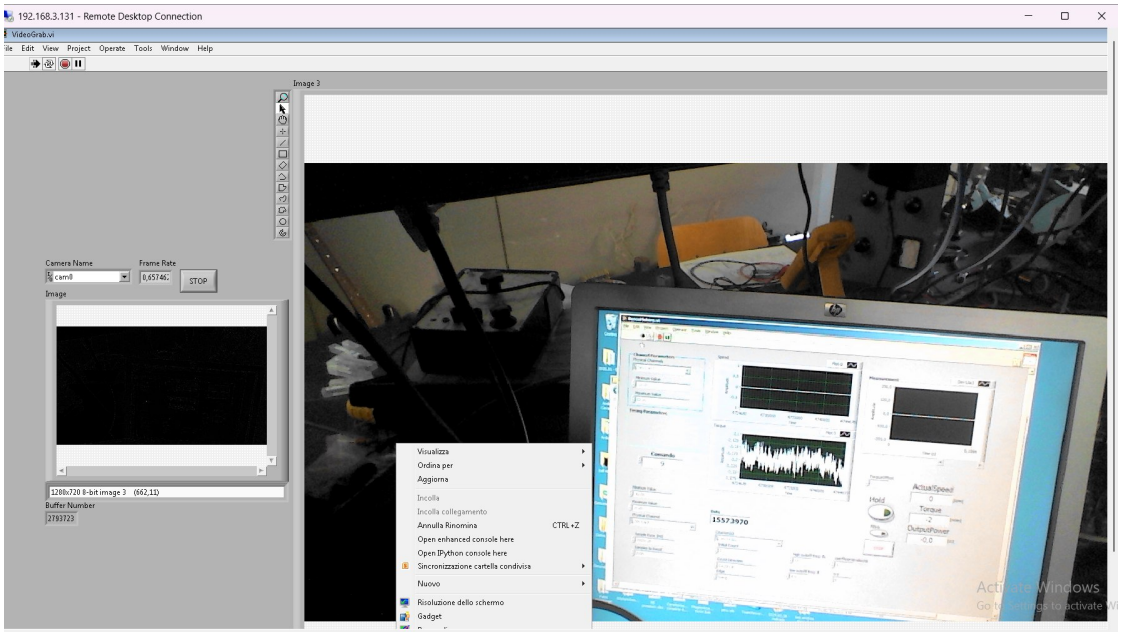


Figure E.2: Remote/camera monitoring environment used during test-bench observation.

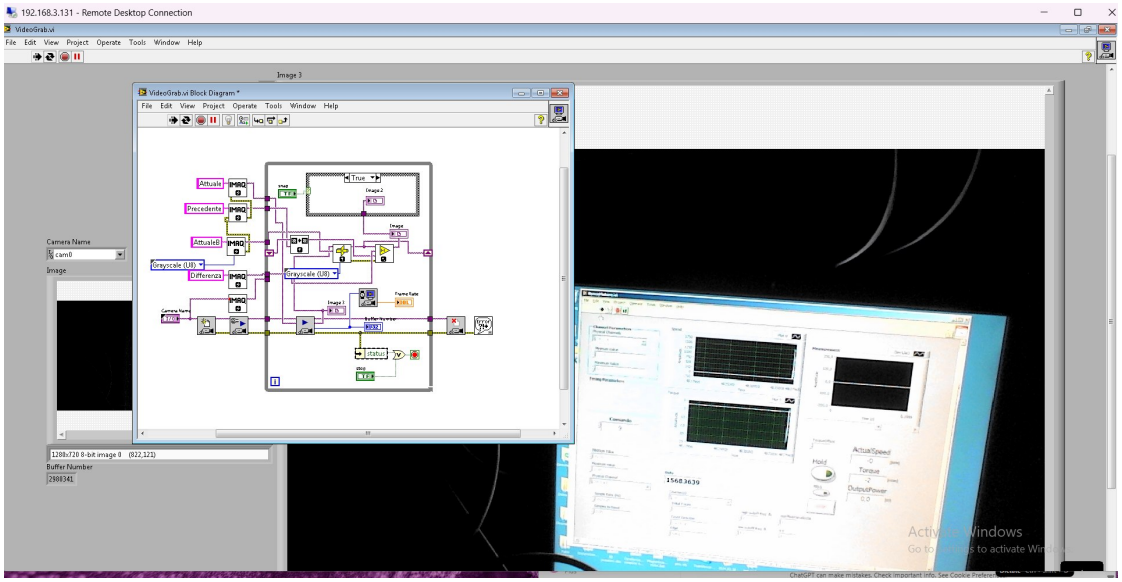


Figure E.3: LabVIEW block diagram associated with the video acquisition and monitoring prototype.

Appendix F

Final Experimental Efficiency Mapping and Performance Assessment

F.1 Purpose of the Final Efficiency Activity

As a final experimental result of the thesis, the developed test bench was used to generate an efficiency map of the tested PMSM drive. This activity represents an important outcome because the goal of the work was not limited to starting the motor or tuning the firmware, but also to obtain meaningful performance information from the complete laboratory drive. The efficiency map was generated from the measured electrical input power, mechanical output power, speed, and torque acquired during the experimental campaign.

For each operating point, the efficiency was calculated as

$$\eta = \frac{P_{out}}{P_{in}}, \tag{F.1}$$

where P_{out} is the mechanical output power and P_{in} is the electrical input power measured by the acquisition system. Therefore, an efficiency value of 0.60 corresponds to 60%.

F.2 Experimental Data Acquisition

The efficiency campaign was performed by changing the speed reference and progressively increasing the braking load. The final dataset includes 45 operating points acquired at nominal speeds of 500, 1000, 1500, 2000, 2500, and 3000 rpm. The 2000 rpm dataset was extracted from the LabVIEW table screenshots, while the other speed series were obtained from the recorded measurement files. The measured quantities used for the analysis were voltage, current, input power, speed, torque, output power, and efficiency.

At low speed, only a limited number of stable torque points could be acquired because a higher braking load caused rapid instability, speed-feedback fault conditions, and motor stop. This limitation is related to the behavior of the brake at low speed and high torque. The brake used in the bench is an asynchronous machine with DC current excitation; in this operating region its torque-speed characteristic is locally very steep. As a consequence, a small variation of speed can produce a large variation of braking torque, making the speed-control loop difficult to stabilize. Therefore, the low-speed/high-torque tests were not forced beyond the stable region, and the low-speed part of the map includes only the operating points that were stable and repeatable during the measurement campaign. The maximum measured torque in the dataset is approximately 28.05 mNm, obtained in the 2500 rpm speed series, while the best efficiency point is located around 2000 rpm and 27.59 mNm.

Figure F.1 shows the final acquisition environment used during the efficiency campaign. The LabVIEW interface was used to monitor speed, torque, input power, output power, and efficiency, while PuTTY and the STM32 development environment were used to observe the firmware state, command execution, and fault handling.

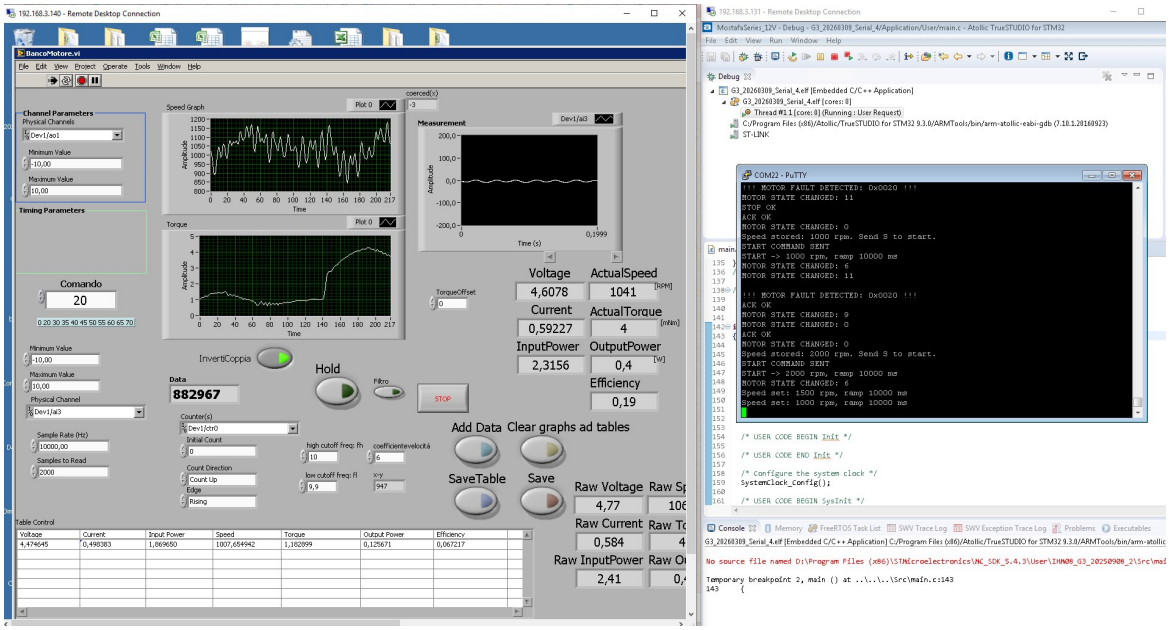


Figure F.1: Final LabVIEW and STM32 monitoring environment used for efficiency-map acquisition. The figure shows the LabVIEW measurement interface together with the PuTTY/-firmware debug environment used during the tests.

F.3 Final Efficiency Map

The final efficiency map is shown in Figure F.2. The blue points correspond to the experimental operating points, while the colored regions are obtained by interpolation between the measured data. The map shows the expected trend of a small laboratory

motor-drive system: efficiency is low at low torque because fixed losses dominate, and it increases when the motor is loaded and the useful mechanical output power becomes larger.

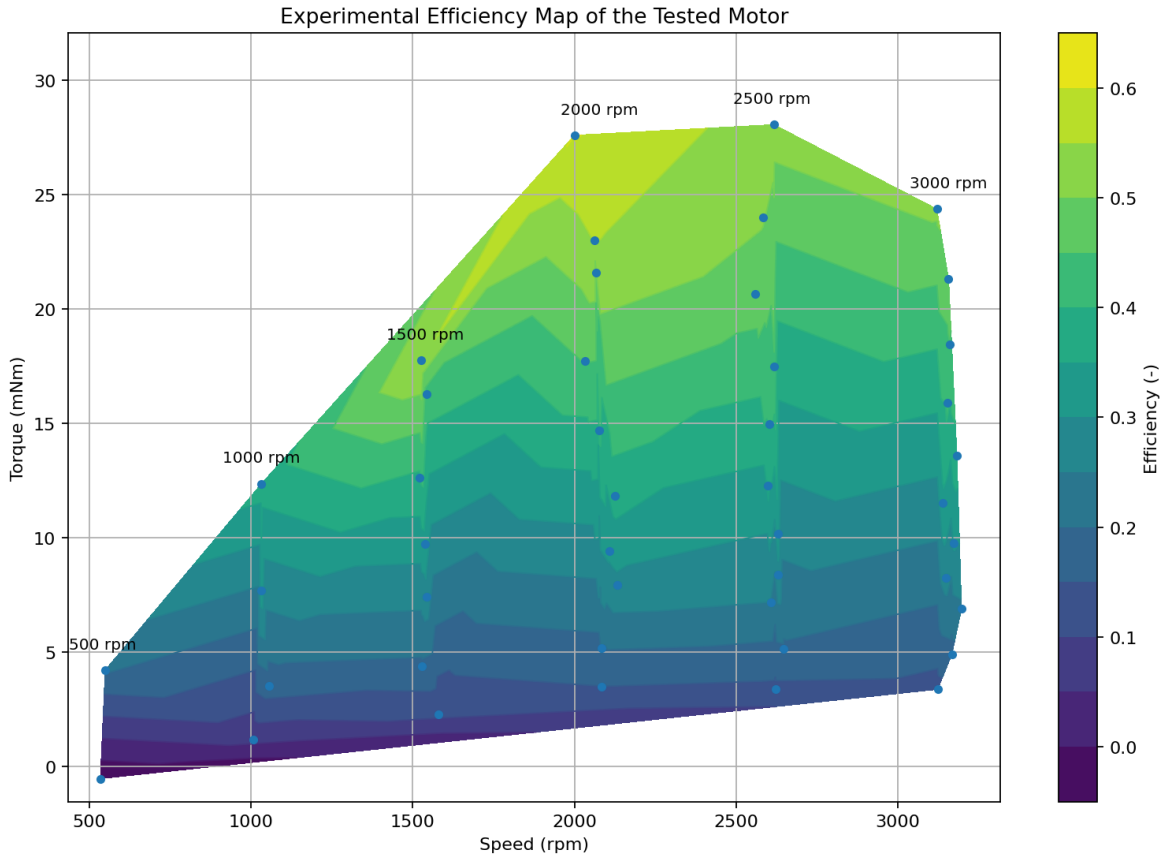


Figure F.2: Experimental efficiency map of the tested PMSM drive. The highest measured efficiency is approximately 0.603, corresponding to 60.3%, near 2000 rpm and 27.6 mNm.

F.4 Summary of Measured Results

Table F.1 summarizes the maximum and average efficiency for each nominal speed. The best measured efficiency was obtained at 2000 rpm, where the peak value reached approximately 60.3%. The maximum measured torque of the complete campaign is approximately 28.05 mNm in the 2500 rpm series. The 2500 rpm and 3000 rpm regions also show good performance, but their peak efficiencies are lower than the best value measured at 2000 rpm.

Nominal speed (rpm)	Measured points	Max. efficiency (%)	Average efficiency (%)	Max. torque (mNm)
500	2	20.4	8.2	4.20
1000	4	37.3	23.6	12.36
1500	7	54.5	32.7	17.74
2000	10	60.3	38.2	27.59
2500	11	52.4	35.6	28.05
3000	11	51.0	33.7	24.36

Table F.1: Summary of the measured efficiency campaign by nominal speed.

Table F.2 reports the best-efficiency point for each speed series. These values show that the highest recorded efficiency is not obtained at the highest speed, but in the high-load region around 2000 rpm. This result is physically reasonable because at very high speed additional losses can increase, while at very low load the useful output power is too small compared with the fixed losses of the complete drive.

Nominal speed (rpm)	Measured speed (rpm)	Torque (mNm)	P_{in} (W)	P_{out} (W)	η (-)	η (%)
500	549.0	4.20	1.191	0.243	0.204	20.4
1000	1032.3	12.36	3.565	1.330	0.373	37.3
1500	1524.9	17.74	5.024	2.737	0.545	54.5
2000	2000.0	27.59	9.459	5.703	0.603	60.3
2500	2616.3	28.05	14.662	7.678	0.524	52.4
3000	3120.4	24.36	15.620	7.960	0.510	51.0

Table F.2: Best measured efficiency point for each nominal speed series.

F.5 Interpretation of the Final Result

The efficiency map confirms that the developed system can be used as a complete experimental test bench, not only as a firmware demonstration. At low speed and low torque, the efficiency is limited because the measured output power is small and the fixed losses of the inverter, motor, brake, and measurement chain have a larger relative effect. Moreover, the low-speed/high-torque region was experimentally critical because the brake did not allow the speed to settle reliably when high braking torque was applied. Since the brake behaves as an asynchronous machine with DC excitation, its torque curve can be very steep in this region; this produces abrupt load variations and makes stabilization more difficult for the sensorless speed controller. For this reason, only reliable and repeatable low-speed points were used in the final map. When the torque increases in the stable operating region, the useful mechanical output power increases and the efficiency improves.

The most favorable measured operating point is located around 2000 rpm and 27.6 mNm, where the measured efficiency reaches approximately 60.3%. This value should be interpreted as the practical efficiency of the complete laboratory drive system, including the low-voltage inverter board, sensorless control, mechanical brake/load unit, and mea-

surement chain. Therefore, it is a meaningful final experimental result for the test-bench activity, rather than a manufacturer-level certification of the standalone motor.

The final efficiency map completes the thesis validation because it demonstrates that the developed bench can command the motor, monitor the firmware state, recover from faults, tune the speed and current loops, acquire electrical and mechanical quantities, and finally transform the measured data into a professional performance map. This result provides a solid basis for future automatic test procedures, calibrated data logging, and comparison of different motors or control strategies.

Bibliography

[1] STMicroelectronics, *STM32F PMSM single/dual FOC SDK v4.3 User Manual*, UM1052, Rev. 9, 2016.

[2] J. F. Gieras, *Permanent Magnet Motor Technology: Design and Applications*, 3rd ed., CRC Press, 2009.

[3] J. F. Gieras, *Advancements in Electric Machines*, Springer, 2008.

[4] A. Cavagnino, M. Lazzari, F. Profumo, and A. Tenconi, "A comparison between the axial flux and the radial flux structures for PM synchronous motors," *IEEE Transactions on Industry Applications*, vol. 38, no. 6, pp. 1517–1524, Nov./Dec. 2002.

[5] L. Shao, R. Navaratne, M. Popescu, and G. Liu, "Design and construction of axial-flux permanent magnet motors for electric propulsion applications: A review," *IEEE Access*, vol. 9, pp. 158998–159017, 2021.

[6] F. Hassanalizadeh, *Motor Speed Control Synchronization Report*, experimental report, 2026.

[7] F. Hassanalizadeh, *Implementation of PID Control in STM32 Motor Control System*, experimental report, 2026.

[8] F. Hassanalizadeh, *Motor Control PID Tuning Report*, experimental report, 2026.

[9] P. C. Krause, O. Wasynczuk, and S. D. Sudhoff, *Analysis of Electric Machinery and Drive Systems*, Wiley-IEEE Press, 2002.

[10] P. Vas, *Sensorless Vector and Direct Torque Control*, Oxford University Press, 1998.