Department of Information Engineering and Computer Science

Bachelor's Degree in
Ingegneria Informatica, delle Comunicazioni ed Elettronica

Final dissertation

Firmware development and Graphical User Interface design for a stepper motor test bench prototype

Supervisor

Roberto Passerone

Student

Tommaso Canova

Academic year 2021/2022

Acknowledgements

First of all I would like to express my deepest gratitude to ProM Facility for the internship experience and the opportunity that they gave me, allowing me to participate in a such innovative project. In particular I would like to thank my company tutor, Maurizio Rossi, who always helped me when I was stuck into some problems. I am also particularly grated to my thesis supervisor, professor Roberto Passerone, which convey to me his electronics passion, whose naturally pushed me to undertake the electronics curriculum.

A special thank goes to my family, which allowed me to study far from home, while supporting me every day and never making me miss anything. Along with them, I would like to thank Giorgia, who always sustained and encouraged me to see the things from a different perspective, especially in the toughest times. Thanks should also go to all the friends met during this university journey, with whom I have shared both moments of joy and stress!

Last but not the least I would like to thank E-Agle Trento Racing Team, a team made of friends, in which I could learn new skills while having fun and bonding new friendships.

Trento, September 9, 2022 *Tommaso Canova*

Contents

Abstract

This thesis reports the work done during my internship period at *ProM Facility*, the company - based in Rovereto (TN) - is focused on mechanics, electronics and informatics prototyping. This work principally covers the high level software development and at the same time the *baremetal firmware* development to support the innovative company project. In particular, ProM Facility had the necessity to prototype a **stepper motor test bench** for an internal usage. This measurement tool is able to determine the **maximum power** and **maximum torque** provided by the under test motor.

At my arrival in the company the test bench mechanical structure was already built, ensuring the connection between a magnetic brake, a torque sensor and a stepper motor. To make the prototype actually working it has been necessary to integrate the hardware development with the software one. Obtaining the characteristic curves that describes the performance of the motor has been one of the main objectives of my work, in order to get this results sensors of voltage, current and torque have been used. Moreover, a wide variety of motor driver parameters selection has been implemented, in such a way to build a custom configuration with the purpose of optimize the produced curves by operating on the single parameters. However, this curve optimization task will be studied by the future company interns.

It has also been developed a dedicated graphical user interface to let the end user interacting with the test bench, with this software it is possible to change the motor driver parameters and test it under load; on the other hand, to read the sensors values and drive the motor, two units have been projected: *master* and *slave*. The first one is made up of a *STM Nucleo F7* board with a custom developed *shield*, where sensors and signal conditioning circuits have been soldered. The second one, instead, is composed of a *STM Nucleo F4* board and an expansion driver board - the *STM X-NUCLEO-IHM03A1* - necessary to control the motor under the F4 *microcontroller* rules.

The thesis especially describes the development process to create the graphical interface using the Python programming language and to write the firmware of the master unit. For the last one a duo formed by the *STM Cube MX* software and the *C* language has been selected. Eventually, the remaining work, consisting of the slave firmware development and the prototyping of the expansion board for the master unit, has been up to the colleague with whom I have shared the internship experience, *Lisa Santarossa*.

As planned, the project has been concluded successfully, gathering the curves of **torque-power** based on the custom configuration chosen by the end user.

1 Introduction

In this first chapter the entire project is briefly explained, contextualizing my contribution for the prototype realization and then illustrating the core components involved.

1.1 Motor test bench

A motor test bench is a measuring tool aimed to detect the main characteristics of the device under test, including: **maximum power** and **maximum torque**. It can be represented by a system typically composed of:

- a motor to be tested,
- a torque sensor,
- a brake - or another motor - able to oppose to the torque wielded by the under test motor.

In other typologies of test benches it is common to directly use a *dynamo-metric* brake, which integrates a rotational speed and torque sensor integrates inside of it. Moreover, in the automotive tests, a roller that supports the car tyre is present; this note is important because it lets us distinguish two types of motor testing: **dynamo-metric** and **inertial** one. In the first case brake points and interval are decided in order to obtain the motor curves related at every brake value, while, in the second case, the brake is not involved, but the mass of the roller and its inertia is exploited to obtain the curves, however the results are less precise compared to the first mode. The block diagram which represents the chosen components configuration for the test bench built by *ProM Facility* is reported in figure 1.1. As it can be seen, it is characterized by the presence of a *stepper* motor, a rotary torque sensor and a magnetic brake.

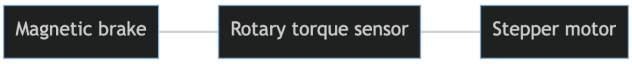
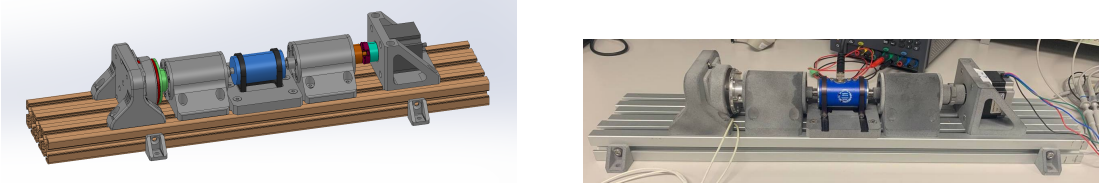


Figure 1.1: ProM Facility test bench block diagram

1.2 Problem statement

During my internship at *ProM Facility* - leader company in mechanics, electronics and informatics prototyping - I had been involved in the prototyping of a **stepper motor test bench**. This project arises from a corporate need, in fact ProM Facility necessitated for a long time to have a test bench to **measure the maximum torque** provided by a kit made up of a driver and a stepper motor (with a **dynamo-metric test**). By the time the internship started, the electronics division of ProM had already had a calibrated torque sensor with analog output and a magnetic brake, with the related mechanical structure capable to connect the components each other.



(a) Test bench CAD model

(b) Test bench prototype

Figure 1.2: Stepper motor test bench

1.2.1 Requirements

In order to test the motor, it has been fundamental to divide the work in the following macro-functionalities:

1. Driving the motor
2. Driving the magnetic brake
3. Measuring the involved physical quantities such as: torque, voltage and current of the motor
4. Developing an user interface to set the motor driver parameters and show the plots related to the obtained motor curves

To satisfy the point N°1 the couple board-driver composed of a *STM NUCLEO F401RE*[1] and a stepper motor driver (*X-NUCLEO-IHM03A1*[2]) has been individuated. This sub-system takes the name of **slave**. To meet the points N°2 and N°3 it has been required to prototype a custom *PCB* (*Printed Circuit Board*), because, in order to drive the magnetic brake - using *PWM* (*Pulse Width Modulation*) - and read the sensors, it has been necessary to develop *ad hoc* circuits. Hence, those two points allowed to identify another couple made up of a *STM NUCLEO F746ZG*[3] and a PCB, this sub-system takes the name of **master**.
Lastly, referring to the point N°4, a dedicated *GUI* (*Graphical User Interface*) has been designed to let the user change the motor configuration parameters and correctly display the curves. The interface can be used by a laptop connected via *USB* (*Universal Serial Bus*) to the two units, which are in their turn connected each other by a serial connection based on the *UART* protocol (*Universal Asynchronous Receiver-Transmitter*).

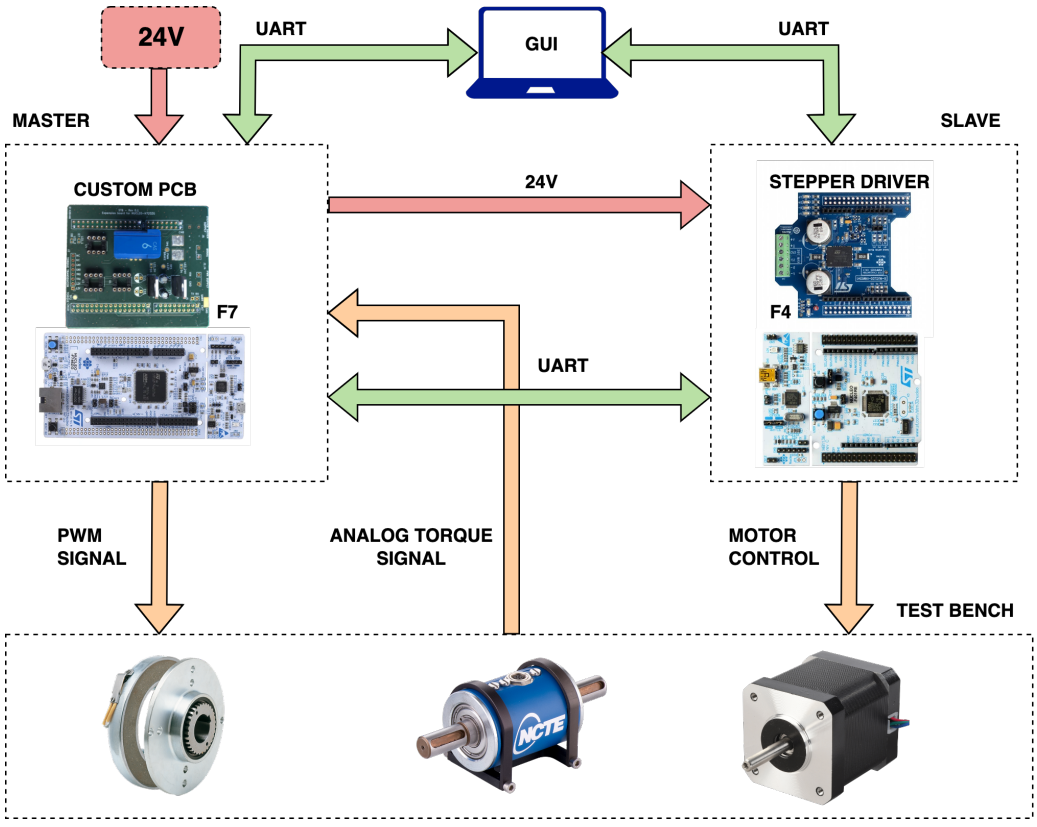


Figure 1.3: Block diagram of the entire project

[1] https://www.st.com/en/microcontrollers-microprocessors/stm32f401re.html
[2] https://www.st.com/en/ecosystems/x-nucleo-ihm03a1.html
[3] https://www.st.com/en/evaluation-tools/nucleo-f746zg.html

1.2.2 Task to be performed

In order to get an operating test bench, it has been necessary to divide the work into the following tasks:

1. Firmware development of the *STM32F7* master board
2. Firmware development for the *microcontroler* mounted on the *STM32F4* slave board, based on *Microsoft Azure RTOS* (*Real Time Operating System*)
3. Software development of the GUI in order to interact to the boards using a laptop
4. Expansion board PCB development to acquire the sensors signals

I have personally carried out the task N°1 and N°3, while task N°2 and N°4 have been completed by the colleague, with whom I have shared the internship experience and the project development, *Lisa Santarossa*.

1.2.3 Expected goals

The main goal of this thesis is thus to validate the company proposed idea, verifying the functionalities of the provided components and the eventual *modularity* of the system, especially in the case of upgrade or requirements change. The actual motor characterization, the detailed comprehension of every single configuration parameter and its impact on the performance will be the next project step, that will be made by the future interns.

1.3 Thesis outline

In the next chapters will be discussed the following themes.

- **Chapter 2** provides a theoretical background regarding the stepper motor and the magnetic brake.
- **Chapter 3** describes the tools used to develop the project.
- **Chapter 4** discusses the design and implementation choices for the GUI.
- **Chapter 5** covers the firmware development of the master microcontroller.
- **Chapter 6** reports the main results obtained at the end of the project developing.
- **Chapter 7** discusses the future works aimed to bring a second revision of the project, with a particular focus on the encountered obstacles during the developing of the first prototype.
- **Chapter 8** reports some final considerations about the thesis experience.

2 Theoretical background

This section gives a theroretical background regarding the most important components involved in the project: the stepper motor and the magnetic brake.

2.1 Stepper motor

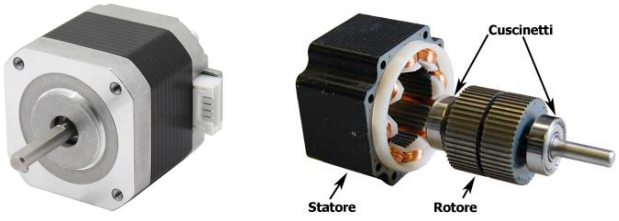
The stepper motor is a a type of motor that is part of the electric motor family.

> An electric motor is a device able to convert electrical energy into mechanical energy, by exerting a force called torque. The conversion is obtained by the generation of a magnetic field, caused by the current flowing in one or more winding of the motor.

Every motor is made up of a *stator* and a *rotor*, respectively the fixed and the rotating body of the device. Unlike the other direct current motors, the stepper motor guarantees a high precision movement because the complete rotation is divided into a fixed number of *steps* (generally around 200), that allow to control the shaft position. This kind of motor is particularly used in precision applications such as in the: automation industry, robotics, 3D printing, etc. The most common version is the *bipolar* stepper motor, which can be driven following four different modes, based on the desired precision:

- *Full-step wave mode*, where just one phase per time is supplied, obtaining a resolution of $\frac{360°}{steps}$
- *Full-step normal mode*, where the two phases per time are supplied, obtaining the same resolution as the *full-step wave* mode. Furthermore, for the same required torque, this configuration needs $\sqrt{2}$ less current rather than the *full-step wave* mode (with the same power consumption) [1]
- *Half-step*, where the phases are supplied alternatively following a specific sequence which involves the two modes mentioned before (one phase on, two phases on, one phase on and so on), obtaining a resolution of $\frac{360°}{2 \cdot steps}$
- *Microstepping*, where both phases are supplied with varying the current continuation, as a consequence being able to divide a step into more *microsteps*. It is possible to gain a resolution which varies from $\frac{360°}{4 \cdot steps}$ to $\frac{360°}{256 \cdot steps}$ based on the chosen driver hardware.

For the prototype a *DPM57SH51-4B*[1] custom step motor has been used, it has a resolution of 1.8° in full step mode, with an accuracy of 5% , moreover its holding torque is 1.01 Nm.
Since that the goal of the thesis work is not oriented to characterization of the motor, but only on the prototyping of the test bench, other details about stepper motor operation will not provided.



(a) Stepper motor (b) Stepper motor in section

Figure 2.1: Stepper motors

[1] `https://www.fullingmotor.eu/a.pag/57sh51-4a-pzk2889kzczk338.html`

2.2 Magnetic brake

The magnetic brake is a device able to contrast the displacement of an object exploiting the magnetic field principle. In the studied case it opposes to the torque exerted by the motor. It is typically composed by a rotating conductive disc and a stator, inside which are placed some winding needed to generate a magnetic field proportional to the current which flows into them. The presence of magnetic field generates induced voltages on the disc which generate parasitic currents, which in turn induce a magnetic field in opposition to the one made by the generator, as consequence a slowing down of the object is obtained. Between the two components there is an air gap that becomes null when the braking level reaches its maximum. In this situation the rotating disc is completely attracted to the stator.

For the prototype a *MWM EMSL060*[2] magnetic brake has been used, it can resist up to 5 Nm until a speed of 8000 *rpm*.

(a) Mono-disc magnetic brake (b) Magnetic brake in section

Figure 2.2: Magnetic brakes

[2] https://mwmfrenifrizioni.it/wp-content/uploads/2020/12/catalogo-generale-elettromagnetici.pdf

3 Used tools

In this chapter is presented a brief description of every tool used to develop the project.

- **STM32-CubeMX** [1]: is a graphical tool that allows a very easy configuration of STM32 microcontrollers and microprocessors, as well as the generation of the corresponding initialization C code for the Arm® Cortex®-M core or a partial Linux® Device Tree for Arm® Cortex®-A core.
- **Visual Studio Code** [2]: is a code editor with many extensions.
- **stm32-vscode** [3]: is an extension to compile, *debug* and *flash* STM32 projects with Visual Studio Code.
- **STM32 NUCLEOF746ZG** [4]: is a STM develop board with a *ARM® Cortex®-M7* micontroller, running at 216 *MHz*, with a flash memory of 1024 *kB* and a *SRAM* of 320*kB*.
- **DearPyGui** [5]: is a fast and powerful Graphical User Interface Toolkit for Python [6] with minimal dependencies.
- **Pyserial** [7]: is a Python library used to manage the serial ports access and the communication.
- **Matplotlib** [8]: is a comprehensive library for creating static, animated, and interactive visualizations in Python.
- **Pandas** [9]: is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.
- **Keysight Infiniium S-Series Oscilloscope** [10]: is an electronic measurement tool able to display the variation of one or more signals in the time domain.
- **NCTE Series 2200** [11]: is a torque sensor which can measure up to 2.5 Nm with an accuracy $\leq \pm 1\%$, until a maximum speed of 5000 *rpm*.

[1] https://www.st.com/en/development-tools/stm32cubemx.html
[2] https://code.visualstudio.com
[3] https://marketplace.visualstudio.com/items?itemName=bmd.stm32-for-vscode
[4] https://www.st.com/en/evaluation-tools/nucleo-f746zg.html
[5] https://github.com/hoffstadt/DearPyGui
[6] https://www.python.org
[7] https://github.com/pyserial/pyserial
[8] https://matplotlib.org
[9] https://pandas.pydata.org/
[10] https://www.keysight.com/us/en/assets/7018-04295/product-fact-sheets/5991-4028.pdf
[11] https://ncte.com/wp-content/uploads/2022/02/Instruction_manual_data_sheet_series2000_EN_V22_02.pdf

4 Graphical user interface development

The development of the graphical interface (GUI) was born from the necessity to allow the test-bench end user to interact directly to the master and slave unities, in order to run the tests and obtain the related motor curves. The GUI is a Python based application which has been mainly developed with the *DearPyGui* library.

4.1 Requirements

The main functionalities required for the GUI are summed up in the following list:

i Communicate in a separate way with each single unit by *UART-USB* serial protocol.

ii Modify each motor driver parameter and load the entire configuration into the slave memory.

iii Read and write on file the motor configuration.

iv Run an automatic test to obtain the motor characteristic curve, while the master reads the sensors values and drive the brake following a linear scale. Between each brake level ($0 \rightarrow 100\%$) a specific period of time passes, based on the automatic test duration input divided by the brake step increase factor, and added to 1.

v Run a manual test, without time constraints, driving the motor and the magnetic brake directly from the dedicated panel.

vi Save the acquired data in a *CSV* (Comma-Separated Values) file and visualize the related plots in an interactive way, letting the user store them on files.

4.2 Project structure

The *divide et impera* principle has been followed to handle efficiently the complexity of the graphical user interface, partitioning it in two separate modules, which in turn present inside separations into panels.

4.2.1 ProM Motor driver configurator

The first module is encapsulated in a window called *ProM Motor driver configurator* as it includes a GUI entirely dedicated to the motor configuration, which is also split in three panels:

- **Motor settings** where all the stepper motor driver parameters are placed. Since the chosen driver can control the motor in two way - *voltage mode* and *current mode* - it has been necessary to divide the common settings between the two modes from those of each one.
- **Serial interface** where it is possible to establish a connection with the slave and the master, allowing custom message sending to the specific units and simultaneously visualize what they send to the computer (such as acquired data from the ADC or the its operating status)
- **Manual interface** where it is possible to send to the motor and brake control commands respectively to the slave and the master, exploiting the serial connection.

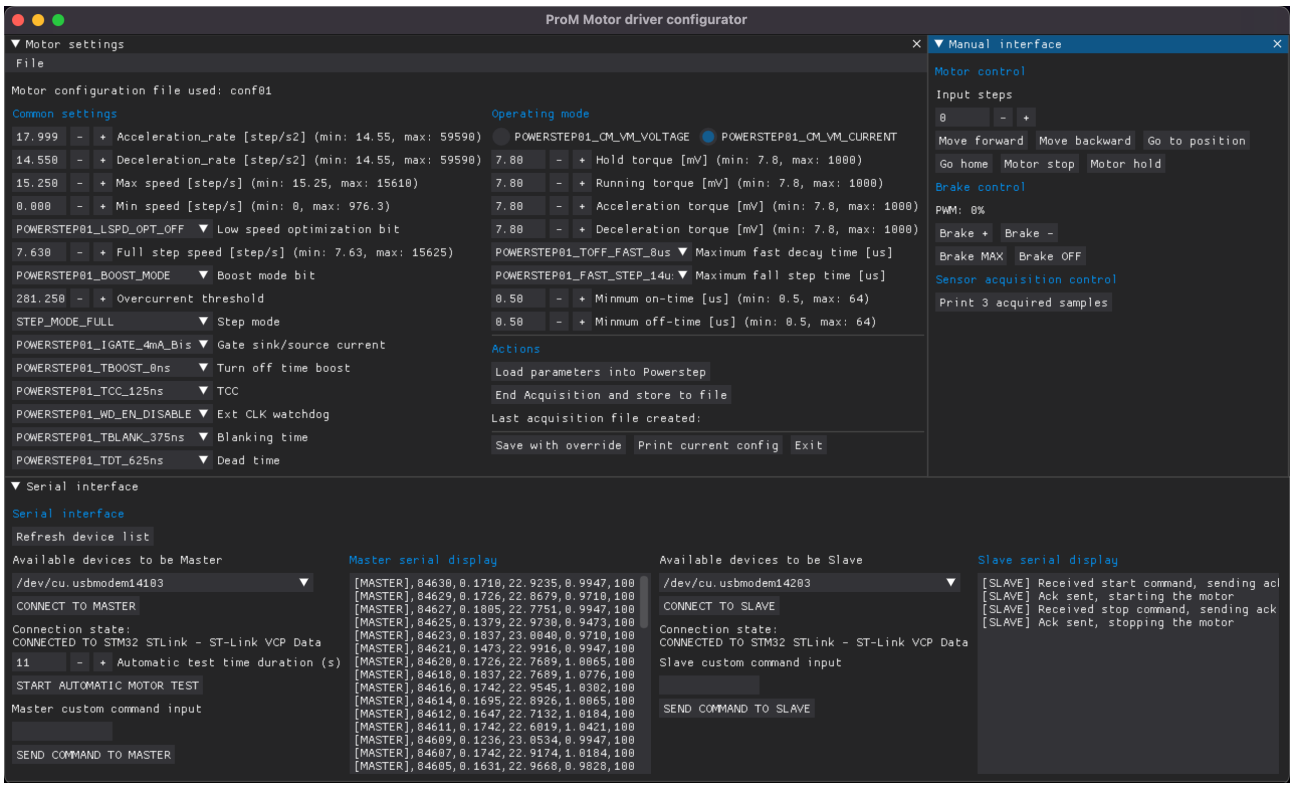


Figure 4.1: ProM Motor driver configurator GUI

In the **Motor settings panel** all the motor driver configuration parameters are showed. They can be identified into two main input typologies: **numerical** or **constrained choice**. To spot each parameter in the design phase and its correspondent data type (*float* or integer), the library used to drive the motor [1] has been consulted, which in many cases uses C *enums*, a specific data type that links a label to an integer number. Consequently, every enum can be represented with a constrained choice input. To accomplish this task from C to Python code a dictionary[2] has been used, which is a data structure that links a key to a value. The two code snippets are reported in listing 3. On the other hand, the numerical inputs can be chosen from the user in a specific range defined by an upper and lower bounds.

```c
/// Stepping options enum
  typedef enum {
    STEP_MODE_FULL    = ((uint8_t)0x00),
    STEP_MODE_HALF    = ((uint8_t)0x01),
    STEP_MODE_1_4     = ((uint8_t)0x02),
    STEP_MODE_1_8     = ((uint8_t)0x03),
    STEP_MODE_1_16    = ((uint8_t)0x04),
    STEP_MODE_1_32    = ((uint8_t)0x05),
    STEP_MODE_1_64    = ((uint8_t)0x06),
    STEP_MODE_1_128   = ((uint8_t)0x07),
    STEP_MODE_1_256   = ((uint8_t)0x08),
    STEP_MODE_UNKNOW  = ((uint8_t)0xFE),
    STEP_MODE_WAVE    = ((uint8_t)0xFF)
  } motorStepMode_t;
```

Listing 1: C code for the Step Mode enum

```python
# Stepping options dictionary
self.motorStepMode_t = {
          "STEP_MODE_FULL":    0,
          "STEP_MODE_HALF":    1,
          "STEP_MODE_1_4 ":    2,
          "STEP_MODE_1_8 ":    3,
          "STEP_MODE_1_16":    4,
          "STEP_MODE_1_32":    5,
          "STEP_MODE_1_64":    6,
          "STEP_MODE_1_128":   7,
          "STEP_MODE_1_256":   8,
          "STEP_MODE_UNKNOW": 254,
          "STEP_MODE_WAVE":    255,
}
```

Listing 2: Python code for the Step Mode dictionary

Listing 3: C and Python snippets compared

[1] STM X-CUBE-SPN3

[2] https://docs.python.org/3/tutorial/datastructures.html

The interface can also create and store new motor configurations with a dedicated file picker. In the design phase *JSON*[3] (*JavaScript Object Notation*) format has been selected to encode the settings, because of its ease of use and versatility. At the end of the input section, in the *Actions sub-panel*, it is possible to use some predefined routines such as:

- loading the configuration to the slave, serializing every parameter and send the whole packet to the slave using the serial connection,
- ending the automatic data acquisition and storing it in a file named with the date and the time referred to this operation (following the format *Day-Month-Year_Hours-Minutes-Seconds.csv*),
- overriding the current configuration,
- printing the current settings in a window popup,
- quitting the program.

The **Serial Interface panel** offers the possibility to connect to both units in order to send commands and, at the same time, display the devices serial output. The communication *baudrate* is fixed at 115200 bit/s. Then, looking at the panel, it is possible to note that the master section provides an input where the duration of the automatic test can be set. In particular, with this value the master unit can split the total acquisition time into different slots, where different brake values are set. To give an example, in the case where the user decides to set the test duration to 55 seconds, the master will increase the brake force of 10% every 5 seconds (in the case where the brake step increase factor is 10%). The logic behind the brake time slicing is reported in section 5.2.2.

Then, the last panel of the window is the **Manual Interface**, which is useful to do a manual test, as the name suggests. In fact, it is possible to drive the motor directly sending the commands to the slave unit, while at the same time it can be possible to set the brake intensity and read the sensors values by communicating with the master unit.

Because the performance impact on the *CPU* of multiple concurrent tasks is not negligible, the usage of *threads* has become necessary to handle the dual serial communication. It has been decided to redirect the output of the units to the GUI using two separate queues[4], one for the master and one for the slave, which are filled and emptied by four threads (two for every unit). Therefore, the first two threads read respectively the output messages from the devices, *en-queuing* them to each own queue, while the other two threads do the *de-queuing* operation to display the received payload in each device section.

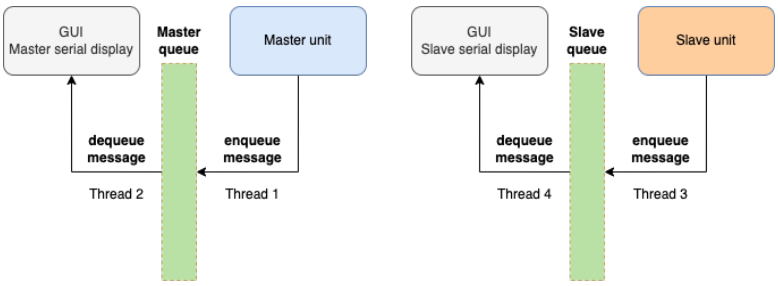


Figure 4.2: Thread and queues usage to read and display the serial messages

When an automatic test is launched and the master device starts to send the strings with the sensors values to the GUI, a list with the related information proceeds to be filled by the interface. The list used as buffer will be parsed into a CSV file - and stored in the csv folder - only when the *End acquisition and store to file* button will be pressed.

[3] `https://www.json.org/json-en.html`
[4] `https://docs.python.org/3/library/queue.html`

4.2.2 CSV to Plot converter

The second module (showed in Figure 4.4) contains three panels, one dedicated to the selection and preview of the CSV file with the data and the other two to display the plots. Unfortunately the plot module implemented in the GUI library can't handle more than three y-axis, thus in order to display all the categories of sensors (five in total), it has been necessary to use two different plot windows. In addition, due to the fact that the DearPyGui graphical motor is *thread-based* and the matplotlib library is not *thread-safe*, it has not been possible to implement the plot saving on file feature directly in the GUI. Therefore, a little Python script to handle this task has been developed. This script takes as first argument the CSV file - which is located in the same directory - used to plot the curves, otherwise it is possible to use the **last** keyword to pick the last added CSV file. Then it creates a folder and sub-folder based on the date and time of the file creation (as mentioned before the file name follows the format *Day-Month-Year_Hours-Minutes-Seconds.csv*). The relative code is showed in listing 4, while the CSV file header is reported in Figure 4.3.

Sender	Time [ms]	Torque [Nm]	Voltage [V]	Current [A]	Brake [%]

Figure 4.3: CSV file header used to store the sensors values

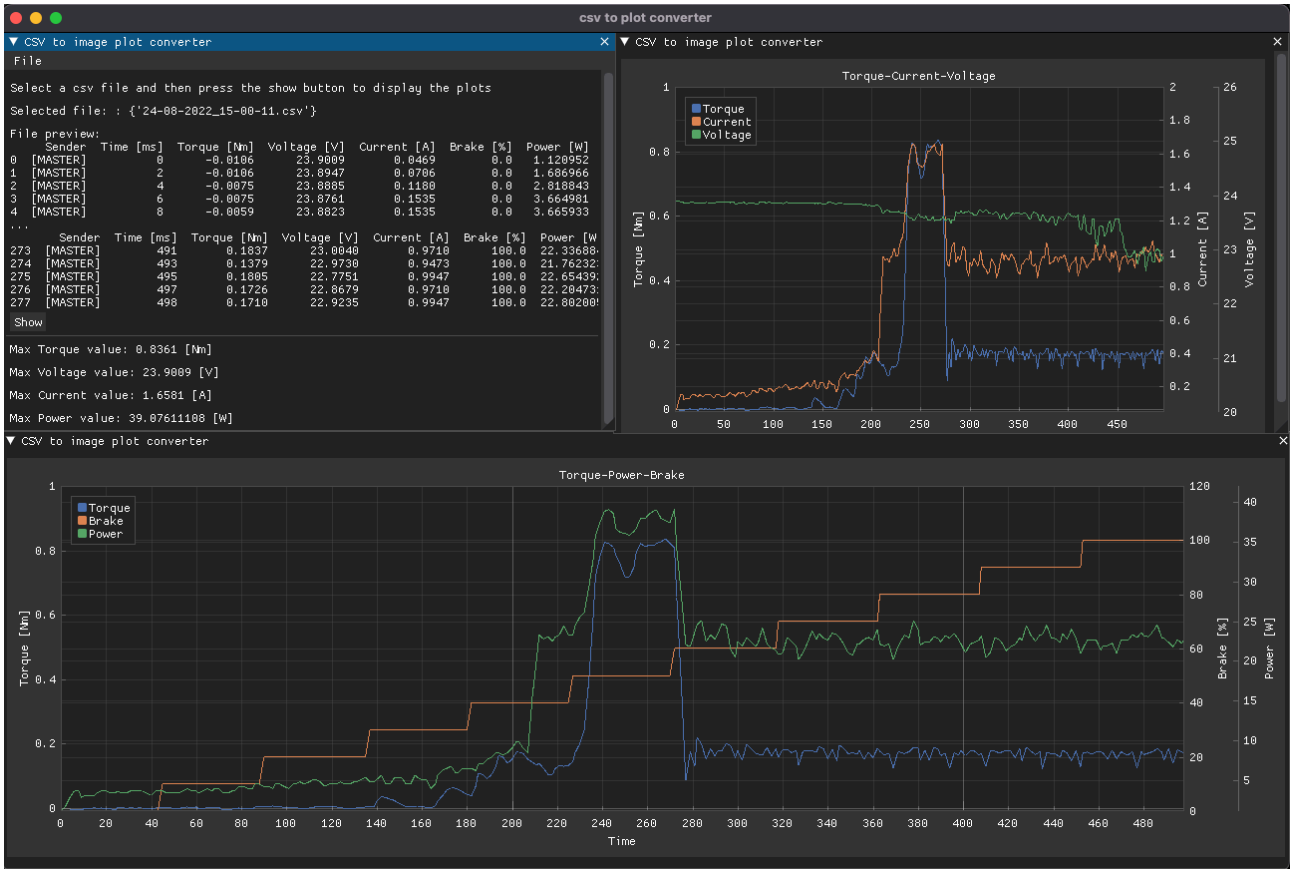


Figure 4.4: CSV to Plot GUI used to read and visualize the acquired data

```python
import pandas as pd
import sys
import os

# If keyword last is detected the last csv file added will be read
if str(sys.argv[1]) == "last":
    # Get the list of all the csv file in the directory
    files = os.listdir()
    # Remove save_plot.py from the list
    files.remove(sys.argv[0])
    files.sort()
    file_name = (files[-1])
# Otherwise the file name given will be selected
else:
    file_name = str(sys.argv[1])
df = pd.read_csv(file_name)
initial_time = df.iloc[0]['Time [ms]']
# if time is relative then convert to absolute
if initial_time != 0:
    df['Time [ms]'] -= df.iloc[0]['Time [ms]']
file_name = file_name.replace(".csv", "")
df = df.dropna()
# If there isn't a Power column it will be created
if 'Power [W]' not in df.columns:
    df["Power [W]"] = df["Voltage [V]"] * df["Current [A]"]
# Split the directories name in DD_MM_YYYY and HH_MM_SS from the original file name
directory_name = file_name.split("_")
# Check whether plots folder exists, if not one will be created
if not os.path.exists(f"../plots"):
    print("Creating plots dir")
    os.mkdir(f"../plots")
# Check whether plot/directory_name[0] folder exists, if not one will be created
if not os.path.exists(f"../plots/{directory_name[0]}"):
    os.mkdir(f"../plots/{directory_name[0]}")
    print(f"Creating plots dir based on date {directory_name[0]}")
# Check whether plot/directory_name[0]/directory_name[1] folder exists,
# if not one will be created
if not os.path.exists(f"../plots/{directory_name[0]}/{directory_name[1]}"):
    os.mkdir(f"../plots/{directory_name[0]}/{directory_name[1]}")
    print(f"Creating plots sub dir based on time {directory_name[1]}")
# Whether plot/directory_name[0]/directory_name[1] exists it will be selected
# as current work path
if os.path.exists(f"../plots/{directory_name[0]}/{directory_name[1]}"):
    os.chdir(f"../plots/{directory_name[0]}")
print(f"Using path: {os.getcwd()}")
# Save all sensor values in the same plot (using Time as x-axis)
df.plot("Time [ms]").get_figure().savefig(
    f"{directory_name[1]}/All.pdf")
# Plot and save separate values during Time
# eg: Brake.pdf, Current.pdf etc
for i in df.columns:
    if i != "Time [ms]" and i != "Sender":
        df.plot("Time [ms]", f"{i}").get_figure().savefig(f"{directory_name[1]}/{i}.pdf")
```

Listing 4: Python snippet of the save_plot.py script

5 Master unit firmware development

This chapter covers the firmware development of the master unit. As introduced before, this unit has a fundamental role for the application, indeed it is necessary for the analog sensors readings besides the brake and slave control.

5.1 Requirements

The main requirements selected for the master unit are:

1. Brake control using PWM,
2. Analog reading of the torque, voltage and current sensors, using the *ADC* peripheral (*Analog to Digital Converter*),
3. Transfer of converted values to the GUI in order to obtain the plots,
4. Automatic test functionality - based on a given acquisition time - exploiting the UART communication protocol with the slave unit,
5. Manual test functionality - slave independent - where brake and sensors are handled from GUI without asking to the slave unit its state of working.

5.2 Project structure

5.2.1 FSM

A finite-state machine (*FSM*) model has been chosen to describe and formalize the master unit behaviour. Since the device has to deal with an **automatic test** and a **manual test**, the FSM is basically divided in two sequential branches. The first one is aimed to handle a synchronous event (the automatic test), where every action of the unit is time dependent and at the same time is slave-message subordinate. To give an example, in the first branch it is not possible to read the analog sensors before the receiving of a *start-motor* message acknowledgment from the slave. On the other hand the second branch is **slave-independent**, so it is possible to manage a braking command or a reading sensors in an *asynchronous* way, without considering the current state of the slave. To give another example, during a manual test, it is possible to read the sensors or pilot the brake even when the slave is not driving the motor, as this branch is totally slave-independent. Note that a manual test has no need to start from a specific condition, indeed when the master is not operating an automatic test, every command is interpreted as a manual test command. The entire FSM flowchart is reported in figure 5.1.
When an automatic test is launched with the GUI, the master unit sends to the slave a *start_motor* command and then waits for a fixed time in order to receive an acknowledging message from the slave to know if the motor is running. The sending and waiting operation respects a fixed timing defined by the multiplication of the macros MAX_CMD_SEND_ATTEMPTS and MAX_ACK_ALLOWED_DELAY_MS. If the slave does not respect this timing, the test must be re-launched by the GUI; otherwise if it is respected, the master starts to acquire the ADCs values and sends them to the GUI, while piloting gradually the brake, until the total duration of the test. Then the brake is released and a *stop_motor* command is sent with the same acknowledgement procedure.

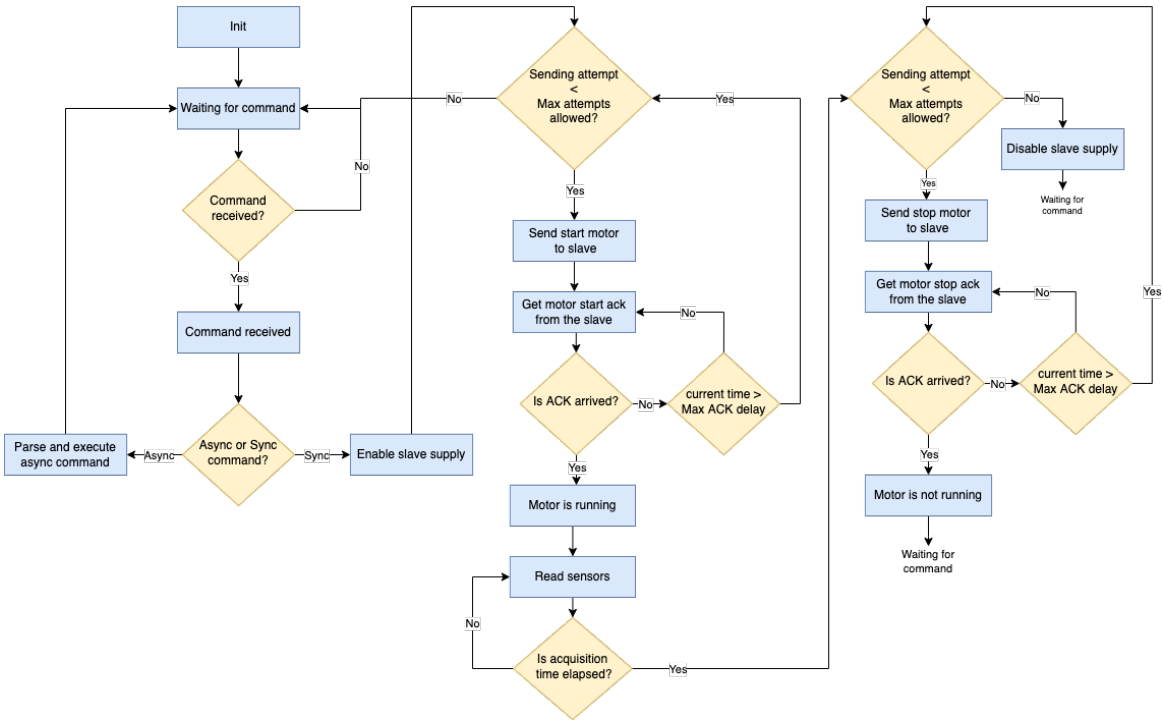
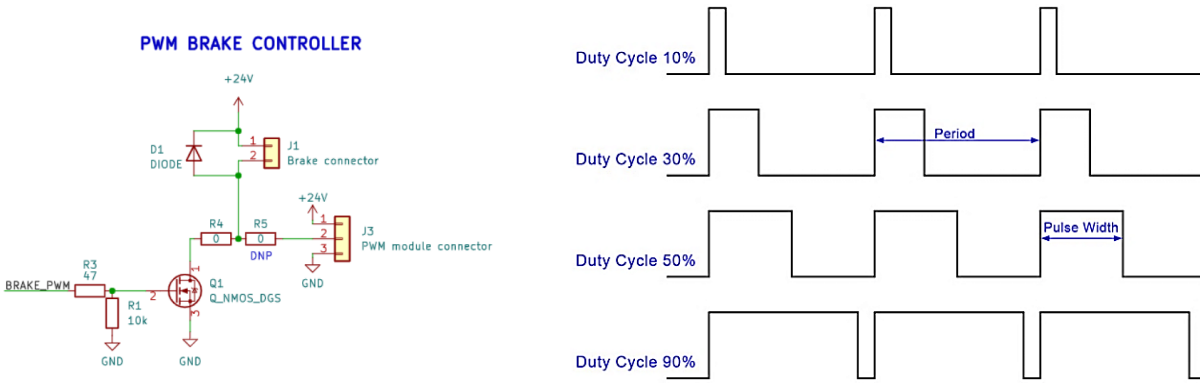


Figure 5.1: Master unit FSM flowchart

5.2.2 Brake control

In order to handle this task, the colleague who was in charge to develop the hardware of the expansion board has designed a little circuit to properly supply the magnetic brake (shown in figure 5.2). The brake connector (*J1*) is connected between the brake supply voltage (24 V) and a *NMOS* transistor, which has its gate driven by the *BRAKE_PWM* signal. The pulse width modulation allows to drive the brake in a full range of voltages (0-24V), cutting the other pole voltage of the connector based on the given duty cycle.



(a) PWM circuit of the extension board

(b) PWM signals with different duty cycle

Figure 5.2: PWM brake circuit and PWM signals

A brake library has been developed to wrap all the PWM HAL (Hardware Abstraction Layer) functionalities offered by STM32 suite and, at the same time, to store every status change of the brake, such as its duty cycle, its period and so on. The brake C struct and function prototypes are reported in listing 7. When the *brake_handler* is instantiated with the default settings the PWM signals runs with a frequency of 25 KHz, and its duty cycle can be set from 0% to 100% with a 10% fixed step variation between each level. Note that **steps are equidistant in time**.

```c
// Brake struct
  typedef struct {
    float pwm_duty_cycle;
    float pwm_period;
    bool is_braking;
    bool is_top_limit_reached;
} brake_handler;
```

Listing 5: C code for the brake struct

```c
// Functions prototypes
void brake_init(brake_handler *brake);
float brake_get_pwm_duty_cycle(brake_handler *brake);
void brake_set_pwm_duty_cycle(brake_handler *brake, float value);
void brake_on(brake_handler *brake);
void brake_off(brake_handler *brake);
void brake_increase_duty(brake_handler *brake, float factor);
void brake_decrease_duty(brake_handler *brake, float factor);
void brake_oc_handler(TIM_HandleTypeDef *htim);
void brake_clear_top_limit_flag(brake_handler *brake);
```

Listing 6: C code for the brake function prototypes

Listing 7: C code of brake library

During a **manual test** the master unit can receive from the GUI a command between: *brake+*, *brake-*, *brake_on_max*, *brake_off*, which allows to modify the duty cycle of the PWM signal. While, during an **automatic test** the brake is not driven by the previous GUI commands, the unit interprets just a specific GUI command instead (e.g. *start_motor+11* which stands for "send a start motor command to the slave and when *ack* arrives start the test for 11 seconds"), and then splits the total test time into braking levels based on the parsed received time. So, in the case of an 11 seconds test, to reach the 100% braking level starting from 0% 11 steps are presents (when a 10% increment is chosen for a step), therefore the *MCU* (*MicroController Unit*) configures a specific timer with a period of:

$$\frac{\text{Total test time [s]}}{\frac{\text{(Max-Min) duty cycle allowed [\%]}}{\text{Step increment [\%]}}} + 1$$

which in this example is equal to $\frac{11s}{11} = 1s$. Every time the timer reaches its full-scale (every 1 second) an interrupt is generated, increasing the brake voltage until when 100% braking level has been achieved for at least one step time, which explains why 11 steps are mentioned. Then it stops the timer and releases the brake, setting its duty cycle to 0% in order to return in a *waiting for command* state.

5.2.3 Analog sensors

As briefly described in the introduction, the master unit needs to acquire the analog signals coming from the torque and current sensors while the voltage supply for the slave - that is still an analog signal - it is obtained only with an appropriate voltage divider. With the purpose of reducing the overall overhead induced by the ADCs operations an hardware approach has been chosen to solve this task. Accordingly to this aim every analog signal is handled by a dedicated ADC peripheral (three in total). The hardware solution consists in the exploitation of three timer channels and *DMA* (*Direct Memory Access*) peripheral, which are fundamental to avoid an instructions overload to the CPU due to the high rate of the conversions. Indeed a single ADC conversion starts when the peripheral gets triggered by a timer event - that is generated with a frequency of 200Hz (5 ms) -, then, when the conversion is completed, the digital value is moved out from the ADC to the memory by the DMA peripheral. To give some metrics the total conversion time of an ADC can be obtained by the following formula [3]:

$$T_{conversion} = \frac{\text{Cycles due to resolution+Sampling cycles}}{ADC_{clock}}$$

Given that the ADC clock runs at 27 MHz, 15 clock cycles are required to obtain a 12-bit resolution and the sampling cycles for a single *rank* (channel read by the ADC) have been set to 480 cycles, it is easy to get the conversion time value for a single read, which is equal to $\frac{15+480}{27 \cdot 10^6} = 18.33\mu s$. Because of the converted values are stored in a circular buffer of 16 elements the total conversion time to get 16 samples - in the case of continuous conversions - would be $18.33\mu \cdot 16 = 293.33\mu s$, so the timer frequency is appropriate even with this timing constraints. When the master request a digital value an arithmetic average is computed over the buffer elements.
Every *raw* digital value is converted into a *GPIO* (*General Purpose Input/Output*) voltage level with a simple proportion: $Voltage = \frac{\text{raw value} \cdot \text{GPIO high voltage level}}{2^{resolution-bits}}$, where in this case the high voltage for a GPIO pin is set at 3.3V and the resolution bits are 12. Then, because the sensors output voltage are

not matched with the GPIO voltage levels, my colleague has designed appropriate circuits connected to the output pins of the sensors in order to obtain this level matching. Afterwards, to convert the torque and current sensors value to a different physical quantity, it is important to know the exact voltage before the latest circuit stage, which is a voltage divider; to do this, the previous obtained voltage is multiplied by $\frac{R_{top}+R_{bottom}}{R_{bottom}}$ factor.
In the current sensor case the value must be subtracted by the voltage read in a zero current condition and then divided by a factor called *theoretical sensitivity* which determines the voltage sensitivity to a change of the current flow and it is expressed in mV/A. The same approach has been followed for the torque sensor: the voltage value needs to be subtracted by an offset and then divided with the sensor sensitivity [mV/Nm]. In this situation the offset indicates the sensor voltage output in the case of zero Nm torque. Lastly, to get the voltage supply of the slave, the voltage matched with the GPIO levels is multiplied by the maximum voltage that can be given by design (in this case 24 V). Some code snippets of adc measurements are reported in the listing 8 and 9.

```
// ADC get torque function
float ADC_get_torque_converted() {
// raw digital value obtained by computing an arithmetic average over the buffer
uint16_t raw_val = ADC_get_voltage_raw();
// raw digital value : 4095 = voltage : 3.3
float converted_voltage = (float)(raw_val * GPIO_MAX_VOLTAGE / ADC_MAX_RESOLUTION);
return ((converted_voltage * TORQUE_DIVIDER_FACTOR) - TORQUE_ZERO_IN_VOLTS) /
(SLOPE_TORQUE_SENSOR / 1000);  // [V]/([mV/Nm]/1000) -> Nm
}
```

Listing 8: C code of the function which returns torque in Nm

```
// ADC get current function
float ADC_get_current_converted() {
// raw digital value obtained by computing an arithmetic average over the buffer
uint16_t raw_val = ADC_get_current_raw();
// raw digital value : 4095 = voltage : 3.3
float converted_voltage = (float)(raw_val * GPIO_MAX_VOLTAGE / ADC_MAX_RESOLUTION);
return ((converted_voltage - CURRENT_SENSOR_OFFSET) /
LEM_CAS_6_NP_THEORETICAL_SENSITIVITY) * CURRENT_DIVIDER_FACTOR * 1000;
// ([V]/[mV/A])*1000 -> [A]
}
```

Listing 9: C code of the function which returns current in Ampere

5.2.4 Serial communication

Two different UART peripherals have been involved to handle the communication with the GUI and the slave. For both channels the following configuration has been used:

Parameter	Value
Baudrate	115200 Bit/s
Word length	8 Bits
Parity bit	None
Stop bit	1

Whenever the master unit is waiting for a command, the *HAL_UART_Receive_IT* function is used to detect if a single character is received, this approach has been chosen rather than wait for a fixed length string in order to allow the MCU to receive variable length messages. When a character is received, the UART peripheral generates an interrupt, afterwards the microcontroller stores the character in a buffer and only when a $\backslash r$ by the GUI or $\backslash n$ by the slave is received, the corresponding command reading is considered completed and a flag is set. To handle properly this task a specific struct has been designed to wrap all the related variables involved in the message building, and it is reported in

Figure 5.3. Even though this solution is not totally optimized since the MCU needs n clock cycles to read a *n-lenght* command, any notable overhead has been introduced.

```
typedef struct {
    bool is_word_complete;
    char cmd_received[CMD_PAYLOAD_LENGHT];
    char last_char_received;
    uint8_t char_index;
} uart_rx_handler;
```

Figure 5.3: C struct of the custom UART message handler

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == UART_GUI.Instance) {
        gui_cmd_handler.cmd_received[gui_cmd_handler.char_index] = gui_cmd_handler.last_char_received;
        gui_cmd_handler.char_index++;
        if (gui_cmd_handler.last_char_received == '\r') {
            gui_cmd_handler.is_word_complete = true;
        }
    } else if (huart->Instance == UART_SLAVE.Instance) {
        slave_cmd_handler.cmd_received[slave_cmd_handler.char_index] = slave_cmd_handler.last_char_received;
        slave_cmd_handler.char_index++;
        if (slave_cmd_handler.last_char_received == '\n') {
            slave_cmd_handler.is_word_complete = true;
        }
    }
}
```

Figure 5.4: C code of the interrupt callback invoked when a char is received

6 Obtained results

This chapter reports some metrics regarding the performance of the master unit and the overall obtained results.

6.1 Flash and RAM usage

The STM32F746ZG MCU includes a 1024 KB flash memory and a 320 KB RAM. The latest build showed in figure 6.1 highlights the sizes in bytes of the different memory areas. Knowing that the flash memory is made up of *data* and *text* sections and the RAM is composed of *bss* and *data* [2], is easy to obtain that:

- **Flash memory used** $= .data + .text = 484 + 38504 = 38.988KB$
- **Total Flash memory usage** $= \frac{38.988K}{1024K} \cong 0.038\%$
- **Ram used** $= .bss + .data = 3124 + 484 = 3.608KB.$
- **Total Ram usage** $= \frac{3.608K}{320K} \cong 0.011\%$

```
 text    data     bss     dec     hex filename
38504     484    3124   42112    a480 build/STB-master.elf
```

Figure 6.1: Memory areas sizes of the project

6.2 Peripherals speed

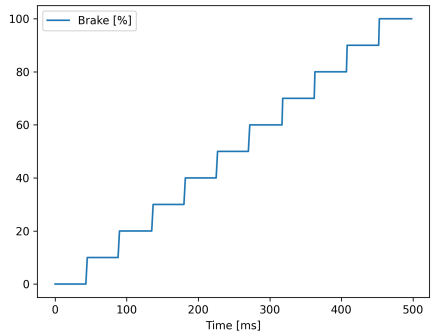
After every 16 samples converted by the ADC, the DMA peripheral needs just 52μs to move all the raw samples from ADC to the related memory buffer. Knowing that, while the brake value does not change, the physical quantities involved are approximately constant, it is possible to analyze the following hypothetical scenario: considering that the minimum acquisition time for an automatic test is 5 seconds (following the GUI constraints), in the case of a PWM step increment of 1%, it is possible to say that the signals variations occur every $\frac{5}{101} \cong 50ms$, which corresponds with a frequency of 20Hz. As a consequence it can be said that the *Nysquist-Shannon* theorem is respected because the $f_{sampling}$ chosen, which is 200 Hz, is at least two times greater than $\frac{1}{50ms} = 20Hz$. So this consideration in addition with the ADC conversion time - computed in section 5.2.3 - allow us to say that the sampling frequency of 200 Hz can be considered suitable for the application requirements.

Parsing a received message and using the *UART Transmit* function require the same amount of time, which stands around on 1ms.

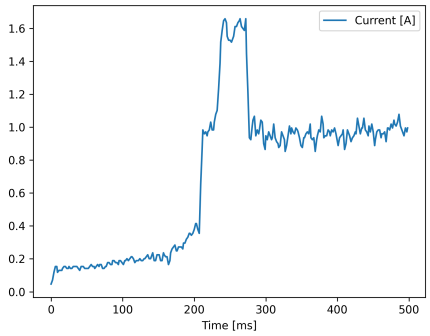
The delay between a start or a stop command from the master and an ack message sent back by the slave is very stochastic: this is due to the fact that the slave firmware uses an RTOS that assigns a lower priority to the task used for the serial communication rather than the one used to control the motor, so as design choice some messages can be dropped. Because of it, the delay between an action command and an acknowledgment message can vary. However the mean delay time between the master and the slave units is around 500ms. In order to reduce this problem it could be necessary to modify the slave firmware, changing the tasks settings and observing if a related changing in the performance can be obtained.

6.3 Motor characteristic curve

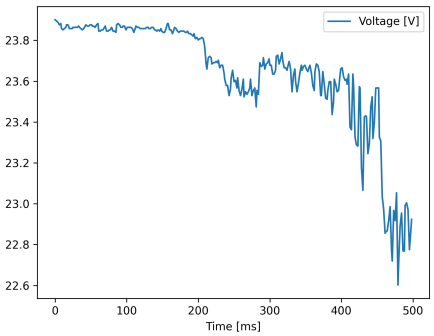
An overall noise reduction for the measurements has been obtained combining a good hardware design and an arithmetic mean over the sampled values. During the testing phase the standard motor configuration has been used, obtaining a **torque** and **power peak** of **0.82 Nm** and **39 W** towards the 50% braking value. In particular, the torque value is kept lower than the maximum one acceptable by the brake and the torque sensor, so that a mechanical break is surely avoided. It is important to remind that the motor runs at fixed speed so the sensors values are related to the acquisition time and the braking level. With this standard configuration the mean torque value, after the spike, is around 0.16 Nm while the mean power is 22.3 W. In the next phase of the project, the configuration will be changed and those mean torque values will be stored in order to draw a characteristic curve based on the motor speed variation.



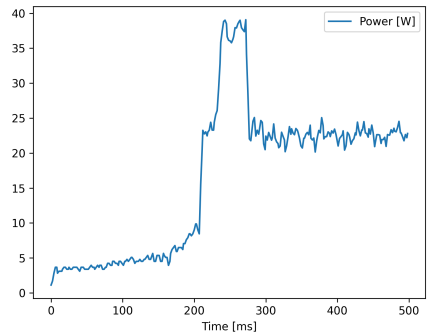
(a) Braking level plot, $Time(ms) - Brake(\%)$



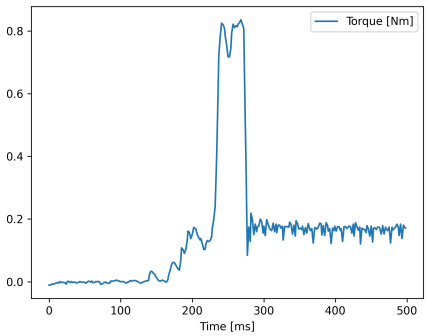
(b) Current plot, $Time(ms) - Current(A)$



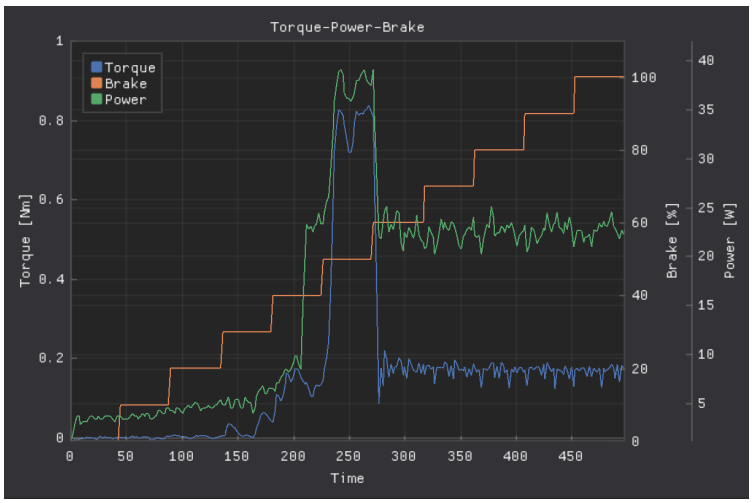
(c) Voltage plot, $Time(ms) - Voltage(V)$



(d) Power plot, $Time(ms) - Power(W)$



(e) Torque plot, $Time(ms) - Torque(Nm)$



(f) Torque - Brake - Power plot

Figure 6.2: Plots of the physical quantities involved in the motor curve creation

7 Future works

This chapter covers the main obstacles encountered during the development phase and the intended implementations to avoid them in the future.

Originally the master unit was designed around the *STM32H7* MCU which would have offered twice the speed and the memory of the current mounted MCU, but due to the chip shortage the project has been moved towards the STM32F7 MCU. Nevertheless as illustrated in Figure 6.1, the firmware can be considered pretty light, so this deficiency of powerful microcontrollers can be considered negligible. However in order to offer more compatibility for the next revisions the *Nucleo-144* board format will be kept.

In the prototyping phase the usage of variable length string has been useful to get more flexibility during the debugging and to avoid to draw up a predefined list of commands since the beginning of the project. Nevertheless - after a new requirements specification - it will be worthwhile to build a shared library between both units and the GUI, using fixed-length messages. This solution could introduce more standardization, letting the programmer operate on all devices while the commands structure is edited in just one file. Moreover, an industrial serial communication protocol such as *RS-485* or *SPI* (*Serial Peripheral Interface*) will be used, replacing the UART one.

Unfortunately, despite reducing the PWM increment step until 1%, it is very hard to achieve a linear behavior for the brake: in fact, after the 50-60% braking level its torque follows an exponential curve, this is caused by its own nature, so the only way to fix is replacing it with another component. In addition, another problem with the magnetic brake is the easy overheating phenomenon that arises after a few consecutive tests and the hardship to dissipate it efficiently. So, in the next revision the magnetic brake will surely be replaced with an AC motor, also because a more precise control to the load will be necessary to push the stepper motor to its limits, in order to optimize its parameters and the related curves. After validating the AC motor as dynamic load, the successive step will be to build a different test bench, that will be given to the **E-Agle TRT Formula SAE Team** of the University of Trento, to which I belong. So the stepper motor will be replaced with a synchronous permanent magnet electric motor, which is the motor used on the team latest car: *Fenice*. Afterwards, the whole mechanical structure will be affected by some changes: the car motor will be piloted by Fenice's *ECU* (Engine Control Unit), while the expansion board of the slave unit will be replaced with a specific one specialized in AC motor piloting, so the load control task will be transferred from the master to the slave. As a consequence the master will become just a *DAS* (Data Acquisition System) for the involved sensors.

Lastly, an important study oriented on motor parameters optimization will be made by the future interns and thesis students of ProM Facility. In fact at the current state of the art, due to the lack of available time, it was not be possible to deeply understand the contribute of every single parameter and how the motor's performance can be really affected by them.

8 Conclusions

Starting from the test bench requirements, the project has been completed as planned, validating the company idea. The system - composed of a magnetic brake, a torque sensor, a stepper motor and two embedded units - is capable of obtaining a **torque-power** curve based on the motor configuration parameters. A custom Graphical User Interface to interact with the test bench has been created, in order to have a full control over any involved unit and plot the motor curves. Another part of the project has been the master unit firmware, which provides two main functionalities: the control of the magnetic brake and the slave unit, and the torque, voltage and current signals acquisition. The development of the slave unit firmware and the master unit expansion board has been completed by my colleague, *Lisa Santarossa*.

The electronics curriculum, which I have chosen during this bachelor, gave me a strong theoretical background, allowing me to approach to many firmware problems with more awareness, precisely because the hardware is always involved. Moreover, the experience with **E-Agle TRT**, the University Formula SAE team, in addition with the one in **ProM Facility**, boosted my learning process, giving me the possibility to make a lot of mistakes and, at the same time, to acquire new competencies from them.

This thesis work lays the foundation for the next challenges which will be held by the future company interns, such as the motor parameter understanding and the related motor curves optimization.

Bibliography

[1] Dr. Fritz Faulhaber GmbH & Co. KG. *Stepper motor basics*. Application Note 001.

[2] Carmine Noviello. *Mastering STM32: A step-by-step guide to the most complete ARM Cortex-M platform, using the official STM32Cube development environment*. Leanpub, second edition edition, 2022.

[3] STMicroelectronics. *How to get the best ADC accuracy in STM32 microcontrollers*. Application note 2834.