Università degli Studi di Padova

Dipartimento di Ingegneria industriale
Corso di Laurea in Ingegneria dell'Energia Elettrica

Small power electric drives: design and digital implementation

Azionamenti elettrici di piccola potenza: progettazione ed implementazione digitale

Relatore
Prof. **Luigi Alberti**

Candidato
Luca Mincato

Correlatore
Dott.**Giuseppe Galati**

A.A. 2021-2022

Data di Laurea: 2 Dicembre 2022

Abstract

L'elaborato finale esposto riguarda l'azionamento elettrico di un motore SPM tramite software.
Il microcontrollore utilizzato è il F28069M sviluppato dalla Texas Instruments mentre il software per il controllo è PLECS.

Lo studio si divide in due parti fondamentali. Nella prima parte, tramite esempi applicativi, si approfondiranno le funzionalità del microcontrollore per gli azionamenti elettrici. Si studieranno in particolare dispositivi come i convertitori analogico digitali, l'encoder, il modulo ePWM, le interfaccie I/O la gestione degli Interrupt ed i canali di comunicazione.
Nella seconda parte, invece, la tesi si focalizzerà sul controllo di un motore SPM tramite software.
Particolare attenzione sarà data alla caratterizzazione dei parametri fondamentali del motore in modo da ottenere un controllo adeguato.

PLECS è un software di simulazione di modelli dinamici per elettronica di potenza sviluppato dall'azienda Plexim.
Per la modellizzazione di sistemi di potenza i software di simulazione devono essere in grado di schematizzare elementi non lineari come switch e diodi (elementi fondamentali per il controllo). Queste non linearità rendono il costo computazionale del microprocessore alto ed il tempo di esecuzione lungo. PLECS elimina i transitori degli elementi di controllo sostituendo i blocchi non lineari con elementi lineari. Per fare questo PLECS ogni volta che vede una discontinuià che rende il sistema non lineare cambia la topologia della rete in modo da non considerare i transitori di elementi non lineari. Per esempio uno switch sarà considerato un cortocircuito mentre è in conduzione e cambierà in un circuito aperto quando arriverà un comando esterno.
In questo modo PLECS per lavorare risolverà solo equazioni differenziali lineari che cambiano in base alla topologia della rete (piecewise-linear system). In questo modo il tempo di esecuzione del programma è più breve di software come Simulink.

Nella prima parte della tesi sarà data molta attenzione a come PLECS lavora con elementi esterni come un microcontrollore.
Per implementare questa modalità è necessario scaricare un pacchetto apposito di blocchi.
PLECS infatti ha un pacchetto di blocchi per la comunicazione con una prestabilita famiglia di target (chiamata TI C2000 target). In questo modo il software non simulerà più il sistema elettrico ma comunicherà con il LaunchPad in real time.
LA comunicazione tra il computer host e il micocontrollore avviene tramite la debug probe USB (XDS100v2). Il LaunchPad contiene due canali distiniti per la comunicazione.
Il primo viene denominato JTAG e serve per caricare il programma all'interno del microprocessore ed il secondo 'e il canale di comunicazione seriale attraverso l'interfaccia SCI (Serial Communication Interface) che usa lo standard RS-232.
Per la corretta comunicazione tra il computer host ed il microcontrollore è nec-

essario creare una VCP (Virtual comunication port) e configurare la correta porta COM della periferica.
PLECS, quando lavora in questo modo, fa eseguire il programma al micorcontrollore ma non è in grado di apportare modifiche a quest'ultimo in tempo reale. Per fare questo bisogna usare la modalità external mode che,tramite comunicazione seriale, permette di cambiare dei parametri del programma da software.

La prima parte dell'elaborato è composto da molti esempi volti alla conoscenza dei vari blocchi del pacchetto TI C2000.
Una PCB (printed circuit board) con elementi di elettronica come un display,un led,un potenziometro,un encoder è stata conessa al microcontrollore in modo da verificare il funzionamento del codice. L'approccio utilizzato è a difficolta crescente. Come primo esempio si accende un led tramite comando da software per arrivare ad uno degli ultimi esempi che è un contatore con un displaya sette segmenti.

La seconda parte della tesi verte nel controllo di un motore SPM di bassa potenza. Viene applicato al microcontrollore un inverter di potenza UBOOSTXL-DRV8305EVM per pilotare le tre fasi del motore.
La postazione di lavoro è composta da un'alimentatore in continua che funge da DC bus, un computer dove è installato PLECS, il microcontrollore con l'inverter ed infine un motore SPM.
Il primo passo dello studio è stato quello di misurare i parametri caratteristici del motore. Con un metodo Volt-Amperometrico si è calcolato il valore della resistenza e dell'induttanza di fase.
Per il calcolo del flusso del magnete si è calettato un secondo motore al motore pilotato e si è misurata la tensione tra due fasi.
Per il calcolo del termine viscoso al motore è stata imposta una velocità di rotazione. Si è provveduto a staccare il controllo disabilitando la PWM del motore così da misurare il transitorio meccanico del motore.
Una volta conosciuti i parametri si è provveduto alla progettazione dei controllori. Il tipo di controllore scelto è stato di tipo proporzionale ed integrativo (PI).
Il controllo è stato progettato prima con un metodo classico, imponendo una banda passante ed un margine di fase. Poi si sono utilizzati i metodi della cancellazione zero-polo e dell'ottimo simmetrico. Il design dei PI è stato fatto sia per l'anello di velocità che per l'anello di corrente.
L'ultimo passo dell'eleaborato è stato quello di costruire il controllo del motore da software.
I controlli sono stati prima un controllo Volt-Hertz con imposizione di frequenza e tensione per poi passare a controlli a catena chiusa. Prima si è sviluppato un controllo di corrente dove come ingresso c'erano le correnti di asse d I_d e le correnti di asse q I_q. In questo controllo particolare attenzione è stata posta nella fase di allineamento del motore in modo da consentire un corretto controllo di quest'ultimo.
Infine si è implementato un controllo di velocità. In questo caso il motore è stato studiato sia a vuoto che a carico per verificare gli andamenti delle correnti e della velocità.

Contents

Chapter 1

Introduction

An electric drive is a system that converts electrical energy into mechanical energy through a control algorithm.
During the last decades, the development of these devices is increased a lot. The spread of new electric technology such as motors for the automotive, generators for electric production, and other applications has increased the number of control algorithms developed as well.
A major contribution to the spread of these technologies was made by low-power motors. Lots of stuff are driven by a little motor nowadays. From a little appliance to a more complex generator, from a hoover to an electric shutter. Everything is driven by an electric motor so it is useful to understand how these electric devices work and how to control them.
To better implement the electric drive, control algorithms are often represented with block schemes in the s-domain where all the parameters are expressed as a function of frequency instead of time. The easiest and most known control algorithms are the open-loop control scheme and the closed-loop control scheme. Both these two methods aim to force the error between the output wanted and the actual signal to zero. A typically closed-loop control scheme is split into two parts. The first part is the control scheme and the second part is the power electronic scheme.

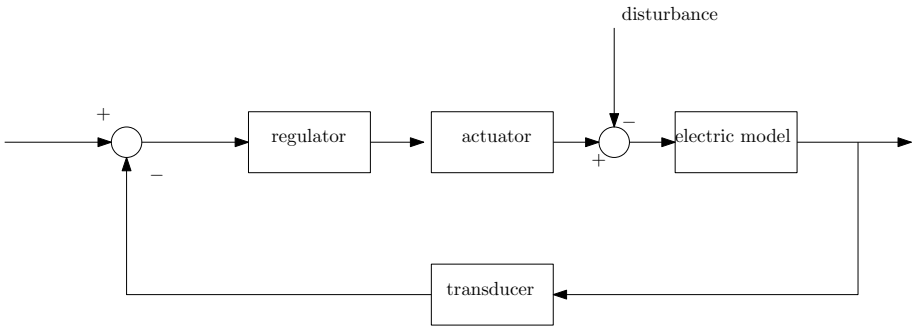


Figure 1.1: typical electric drive closed-loop control scheme

Fig. (1.1) is shown the block scheme of a closed loop control. As input, there is the value the user wants as a reference and as output, there is the actual value. The difference between these two values is the error that feeds the regulator. The output signal of the regulator goes through the actuator(for example an inverter). Then there could be a disturbance signal which modifies the output

of the actuator. The difference between the output of the actuator and the disturbance feeds the electric model of the system controlled and it gave as output the actual output signal of the system.
The development of power electronics systems usually involves the design of both the electrical circuit and the control algorithms.

The study carried out is split into two main parts: the first concerns the study of the main block of PLECS (the software used in this study) and the second concerns the control of an SPM motor from software. The control schemes used are an open-loop control, a closed-loop current control, and a closed-loop speed control.
Throughout the study, the software used is PLECS, a program for modeling and simulating the dynamic system.
Furthermore, PLECS is able to communicate with a specific family of microcontrollers (MCU). To do so a new package of blocks is need to be downloaded. In the first part of the study, several examples are shown in order to explain how these blocks work.
The microcontroller used is the C2000 Piccolo LaunchPad, LAUNCHXL-F28069M an MCU made by Texas Instruments. This low-cost development board is able to communicate with software with a debug probe USB (XDS100v2). The LaunchPad has two different ways of communication. The first is the onboard JTAG emulation tool allowing a direct interface between the TI C2000 target and the host PC. This type of communication is used to run the software program in the MCU. The second type of communication the LaunchPad provides is the SCI (serial communication interface) to run the program in External Mode. Once the program is built it is not possible to change the input parameter from PLECS if this mode is not set.
The main hardware part of the MCU are:

- The ADC module that contains a single 12-bit converter fed by two sample-and-hold circuits, the basic principle of operation is centered around the configurations of individual conversions, called SOC's, or Start-Of-Conversions;

- The enhanced quadrature encoder pulse (eQEP) module is used for direct interface with a linear or rotary incremental encoder to get the position, direction, and speed information from a rotating machine for use in a high-performance motion and position-control system;

- The enhanced pulse width modulator (ePWM).The ePWM peripheral performs a digital-to-analog (DAC) function, where the duty cycle is equivalent to a DAC analog value. The ePWM module represents one complete PWM channel composed of two PWM outputs: EPWMxA and EPWMxB. The letter x within a signal or module name is used to indicate a generic ePWM leg on a device. In this MCU the legs are six;

- The General-Purpose Input/Output (GPIO) multiplexing (MUX) registers are used to select the operation of shared pins. The pins are named by their general purpose I/O name (GPIO0 - GPIO58). These pins can be individually selected to operate as digital I/O, referred to as GPIO,

or connected to one of up to three peripheral I/O signals (via the GPxMUXn registers).

To show the output of the LaunchPad a printed circuit board (PCB) is used. This board is connected to the jumper of the MCU. The main tools connected to the board are:

- A seven-segments Display;
- A potentiometer;
- An encoder;
- A pushbutton;
- three LEDs.

In the second part of the study, an SPM motor is connected to the LaunchPad. From software, this motor has been controlled with different types of control. The MCU used isn't able to control a motor so an external inverter is needed. The PCB is substituted with a BOOSTXL-DRV8305EVM booster pack. This three-phase inverter is suitable for the electric drive needed. The workstation of the study is composed of a host PC, a DC generator, a microcontroller, an SPM motor, and an oscilloscope.

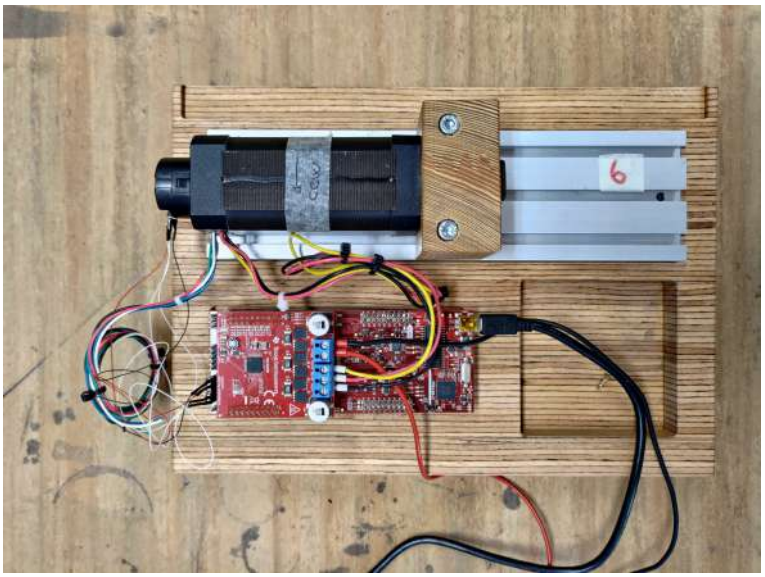


Figure 1.2: Motor Front view

Chapter 2

PLECS Overview

PLECS is a piece-wise linear program for modeling and simulating dynamic systems. This toolbox allows the user to combine the electrical system and the controller in one system model.
The term modeling refers to the process of extracting knowledge from the system we want to study and representing this knowledge in some formal way. These formal ways with which we represent the software are equations, block diagrams, and physical models.
The term simulation refers to the process of performing experiments on a model to predict how the real system would behave under the same conditions. In the context of PLECS, it refers to the computation of the trajectories of the model states and outputs over time using an ordinary differential equation (ODE).
In the PLECS environment can be simulated both ideal resistors, capacitors, transformers, and many electrical components and controller blocks like PID controllers, inverters, and other blocks useful for the electric drives of power electronic system.
The presence of non-linear elements such as switches and GTOs really extends the computation and simulation time. The advantage of using PLECS is that this program doesn't solve the problem by computing non-linear differential equations but it changes the topology of the circuit every time it founds a discontinuity (that could be made by switches or diodes). For this reason, at any time the switches in PLECS are either short or open circuits. This means that before and after the instant of switching the circuit is purely linear and hence its overall behavior is piece-wise linear. With this approach, PLECS contains only linear components thus the circuit can be described mathematically by one set of differential equations.

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx + Du \end{cases} \tag{2.1}$$

2.1 Model execution

The main simulation loop (also called the *major time step*) is divided into two tasks

1. The output functions of all blocks are evaluated in the executive order that was determined during block sorting (the output of a block could

be the input of another block). If a model contains scopes, they will be updated here.

2. The update functions of blocks with discrete state variables are executed to compute the discrete state values for the next simulation step.

Then there are two minor time steps as well that PLECS many uses:

Integration loop

If a model has continuous state variables, it is the task of the solver to numerically integrate the time derivates of the state variables to calculate the momentary values of the state variables.
The integration step is performed in multiple stages (called minor time steps) to increase the accuracy of the integration. In each stage, PLECS computes the derivates at a different intermediate time. Since the derivative function of a block can depend on the block's input the solver must first execute all output functions for that time.

Event detection loop

If a model contains discontinuities, it may register auxiliary event functions to aid a variable-step solver in locating these instants. Event functions are block functions and are specified implicitly as zero-crossing functions depending on the current time and the block's input and internal states.
If one or more event functions change sign during the current simulation step the solver performs a bisection search to locate the time of the zero-crossing. This search involves the evaluation of the event functions at different intermediate times. Since the event function of a block can depend on the block's input the solver must first execute all output functions for a particular time. It is shown that is preferable to simulate the system with a variable-step simulation rather than a fixed-step simulation to avoid serious implications like integration error and event handling.
If one wants to learn more about how PLECS works, refer to the user's PLECS manual available on the PLECS website.

2.2 PLECS with C2000 target support

As an additional package, Plexim can supply target support packages for specific processor families. This target support package enables PLECS to generate specific code for hardware targets such as the TI C2000 family of MCUs. With the PLECS Coder and a target support package, embedded control code can be written down in the PLECS environment, and then it can be uploaded to the target device easily. Moreover, the user can test the embedded control code inside PLECS with a simulation.

2.2.1 Deploy code to C2000 target from PLECS

PLECS allows the users to deploy a code for the C2000 target either directly in the PLECS environment or with a code composer studio.

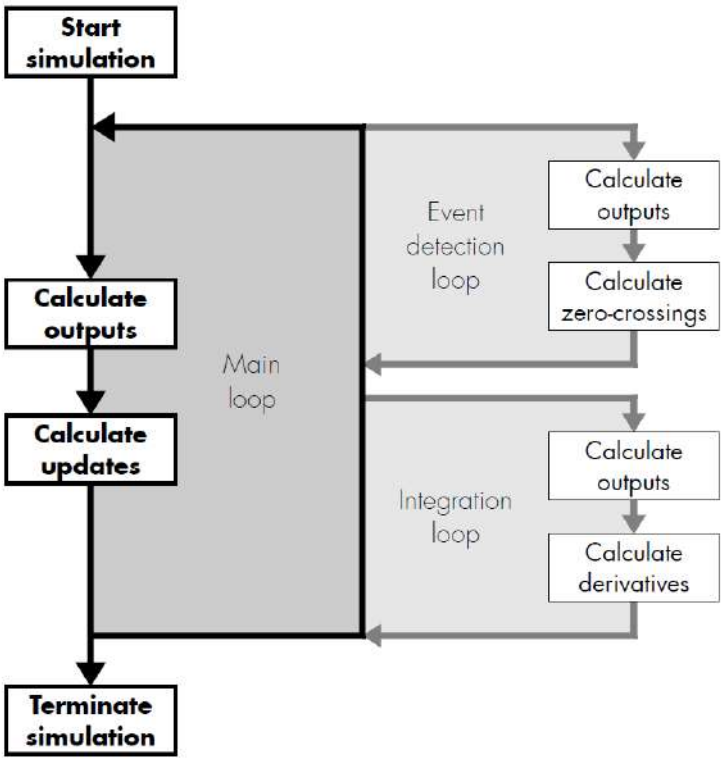


Figure 2.1: Simulation loop

If the user wants to write down the code in PLECS he must go to Coder Option, then to Target, and there he selects the target which in this case is TI2806x. If one downloaded the TI C2000 target support package properly (see appendix 1) one would be able to build the program easily.
The first thing to do is to connect the microcontroller (MCU) to the computer, then the *Coder Option* window has to be opened, and change the precision from "double" to "float" (otherwise the software pops up an error window when we build the program).

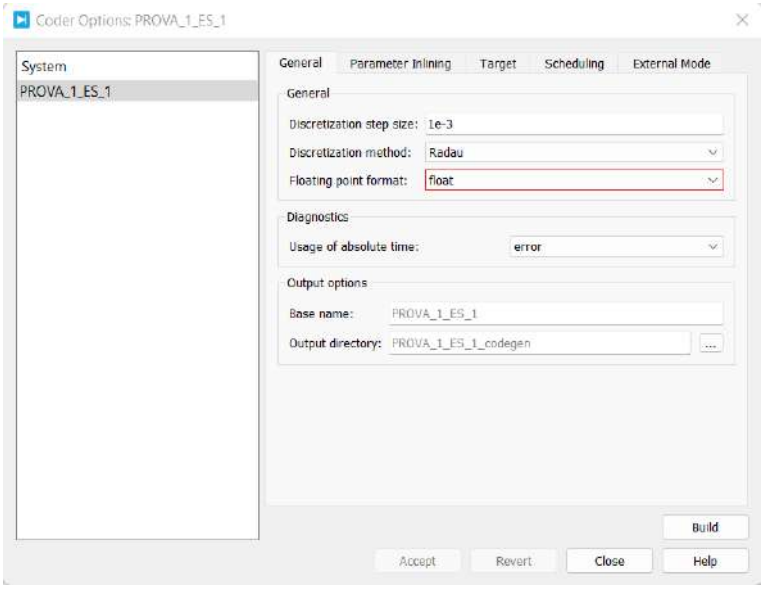


Figure 2.2: setting window

The second thing to do is to set the target of the software. In the Coder Option window, there is the possibility to click on **Target** and choose the MCU that is connected to the computer (In this study the MCU is the "TI2806x"). In the same window, the user can choose the board target. In this study, a launchpad target is used so the board set is "LaunchPad".
While during the simulation PLECS is able to work with either a variable-step solver or with a fixed-step solver when it works with an MCU target it works with a fixed-step solver.
All blocks work at the same frequency which is the same with which the scope shows the samples.
Different from Simulink where all blocks could work at a different time here in PLECS all blocks produce the output in the same moment.

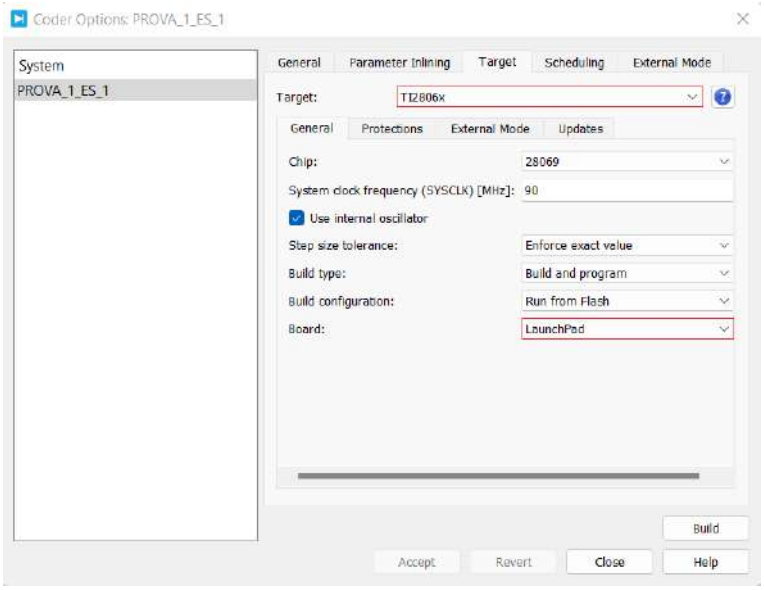


Figure 2.3: Target window

2.2.2 Program the MCU from CCS

PLECS allows the user to write a CCS code and use PLECS as a linker between the C2000 target and CCS.
This is a quick overview of how to configure CCS but if someone wants to learn more about how to use this method properly there is the user's C2000 manual supply by Plexim. Download and install the proper version of CCS (v9.3) at the following link.

`https://processors.wiki.ti.com/index.php/Download$_$CCS`

Once CCS has been installed go to *Coder_target*, then to *tsp_ti_c2000*, then to projects, and at the end locate the template of the desired project.
After that open CCS and open the project window, then click on Import CCS Projects, choose to select the archive file, and select the zip file of the desired target (in our case 28069.zip). Select the discover project and click on finish. Now will appear a new project folder in your workspace.
Right-click on the main folder of the project, select new, and then select the

target configuration file (.cxml file) and select the proper setting. In this case, since a 28069 LaunchPad has been connected, the Texas Instruments XD100v2 USB debug Probe has been selected as the connection and 28069 as the device (write 28069 in Board or Device tab, and it will appear some possible devices and you choose TMS320F28069).
Test the connection with the MCU to be sure the connection is working properly. Next click again in the main project folder and select properties. Then navigate to the general tab and make sure that the compiler version is the one that is compatible with PLECS which is mentioned in the c2000 manual.
There is a folder named 'cg' under the main folder. Here are the PLECS coder files. Right-click on the cg folder and goes to properties. Copy the location of this folder. Next, navigate PLECS and make a program. Once you create a program go to Coder option and then target. Click on Build type and select Generate code into CCS project. Paste the location of 'cg' folder in CCS project directory and build the program. Once the program is run you can go back to CCS and inside the folder 'cg' it will be the code that represents the program you built in PLECS.
Anyway, PLECS generate automatically the codgen folder once a PLECS program is saved.

2.3 External mode

Once the program is built, PLECS allows the users to run the External Mode to update Scopes and Display with real-time data. The External Mode modality could be used only after the building of the program by clicking on "Connect" in the External Mode window.
In the external mode window, the user can choose the type of communication between the software and the MCU. In this specific example, the communication is set to "Serial" in order to allow the serial communication interface.
The default interfaces are GPIO 28 as Rx signal and GPIO 29 as Tx signal.
In the coder option, there is a window called *External Mode*. In order for PLECS to be able to communicate with the external MCU, the user must first click on the pencil image and then select the serial port(in this study the serial port is the COM2).

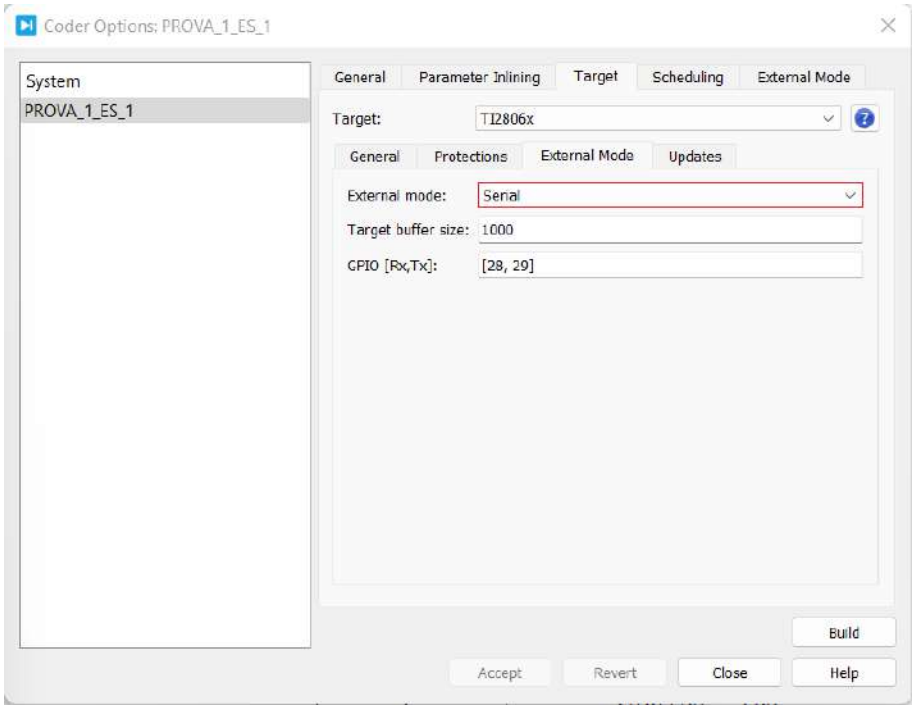


Figure 2.4: external mode window

PLECS builds the program in discrete time. All blocks that are put in the main window run at the same sample time. Simulink in external mode increases the duration of program compilation by increasing the number of scopes entered by the user, but PLECS does not work in this manner. The main advantage of using PLECS is that it is faster than Simulink. So PLECS to ensure a small compilation time decreases the acquisition time by increasing the number of scopes and displays.
The Target buffer size (expressed also in number of words)setting determines how much memory is allocated to store signals for the external mode. The number of words N_w required by the external mode can be calculated as follows:

$$N_w = 2N_{signals}(N_{samples} + 1) \tag{2.2}$$

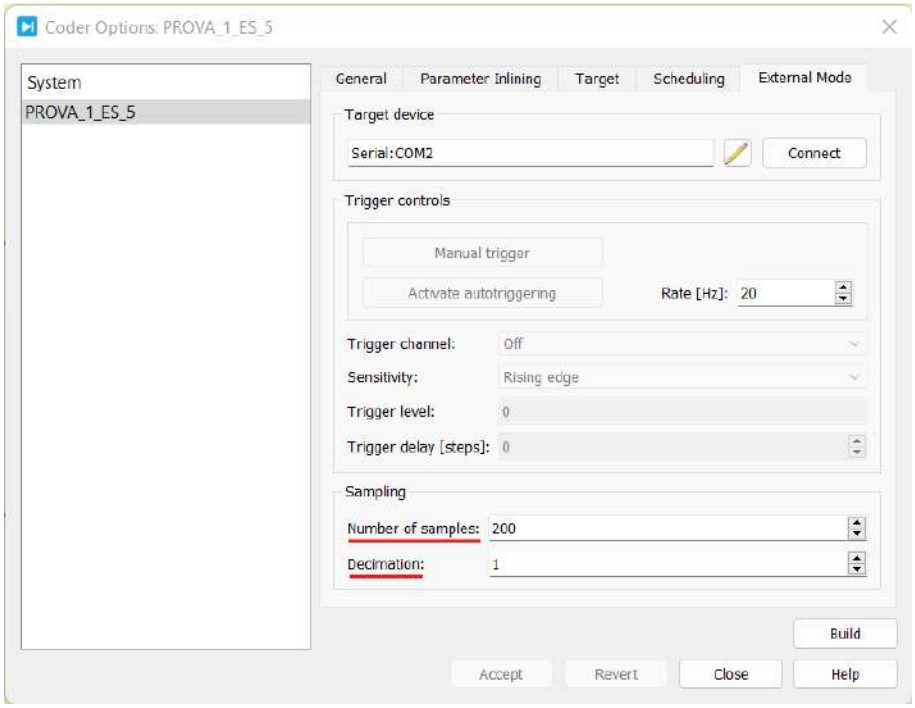


Figure 2.5: external mode setup

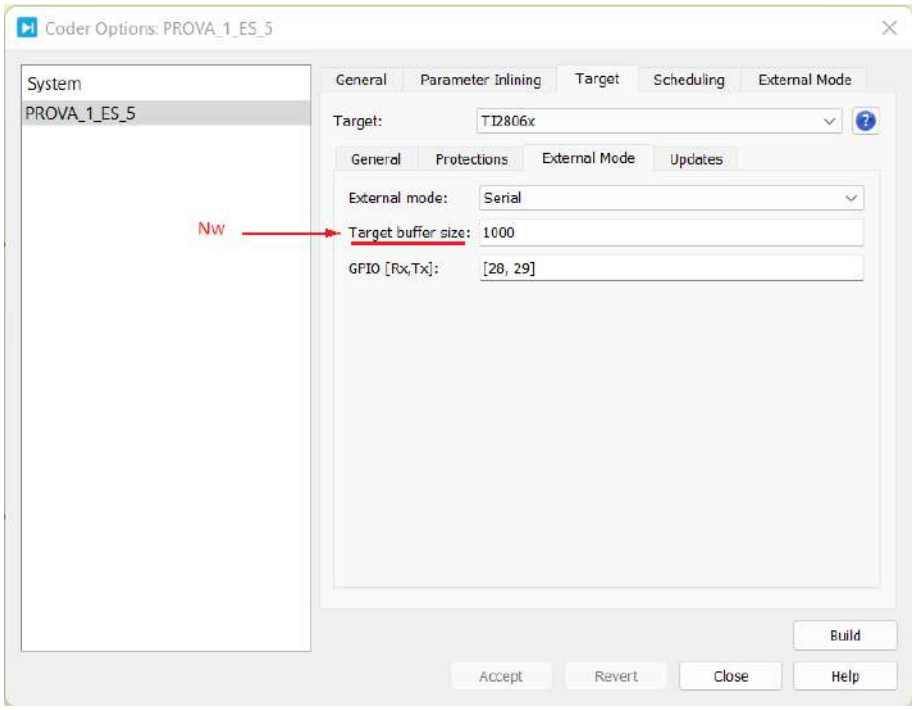


Figure 2.6: external mode setup

The number of samples $N_{samples}$ specifies how many of those signals are displayed. The number of signals is the number of scopes traces and displays. So, once the target buffer size and the number of scopes and displays have been set the number of available samples in the allocated memory can be computed.

$$N_{samples} = \frac{N_w}{2N_{signals}} - 1 \tag{2.3}$$

If the number of samples, it is set in the external mode tab exceeds the number of samples available PLECS will show all the samples available. Notice that if the number of samples has been changed the total time duration of the scope will change as well. In fact.

$$T_{scope} = (N_{samples} - 1)T_{disc}Decimation \tag{2.4}$$

Where T_{disc} is the discretization step size and it is set in the General tab of the coder option window.
If the decimation parameter is set to 1 means that all samples are considered. If that parameter is set to 2, for example, it means that one sample every two is skipped. So, by increasing the decimation the information about the signal is less precise. The benefit of doing this is that the time acquisition is greater and the waveforms of signals can be seen for more time.
This is a little example to show how the acquisition time change by changing the parameters in the setting window.

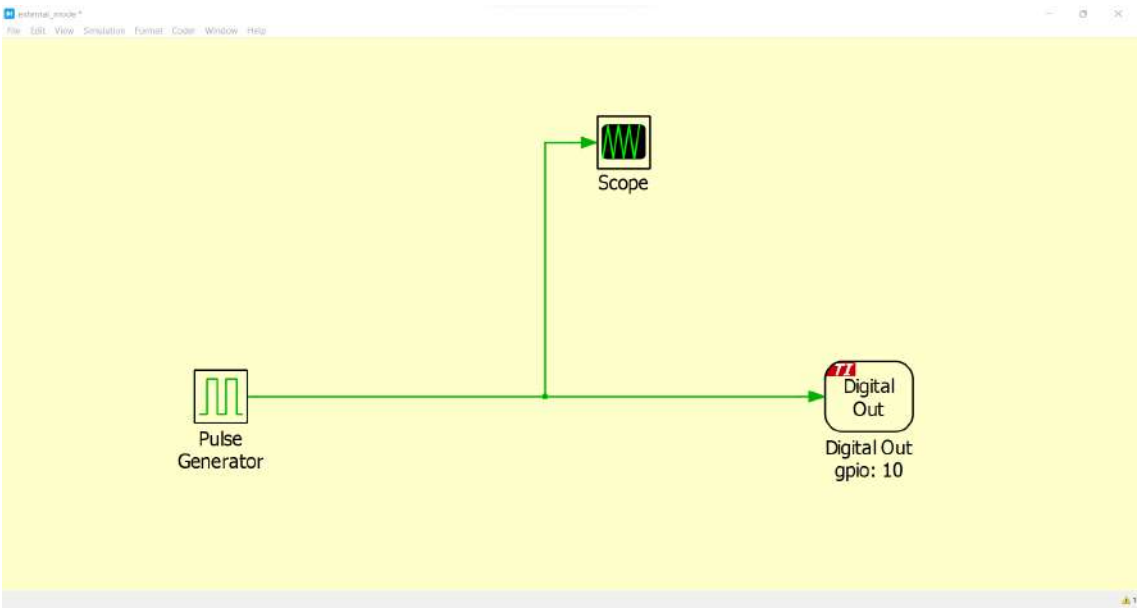


Figure 2.7: overall scheme

Let's impose
$N_w = 1000$
$Decimation = 1$
$T_{disc} = 1e-3 \quad [s]$
$N_{signals} = 1$
The results become:
$T_{scope} = 0.498 \quad [s]$
$N_{samples} = 499$

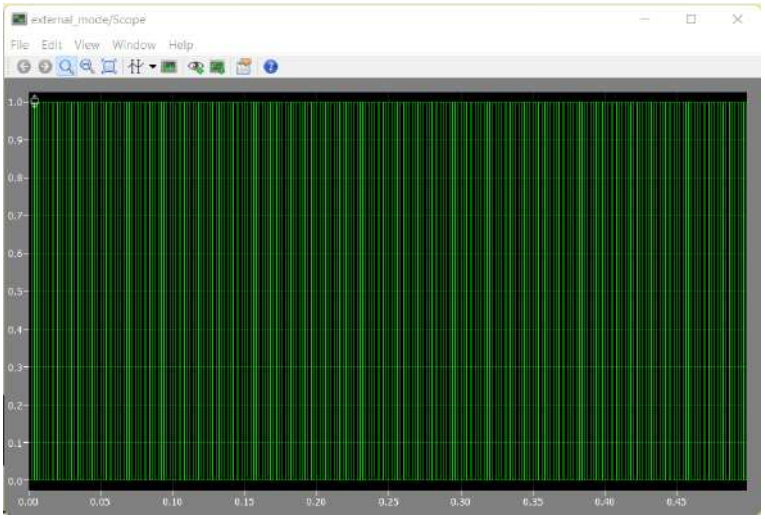


Figure 2.8: $N_{samples} = 499$

Now by imposing the number of samples as:
$N_{samples} = 200$
The total scope time will become:
$T_{scope} = 0.199 \quad [s]$

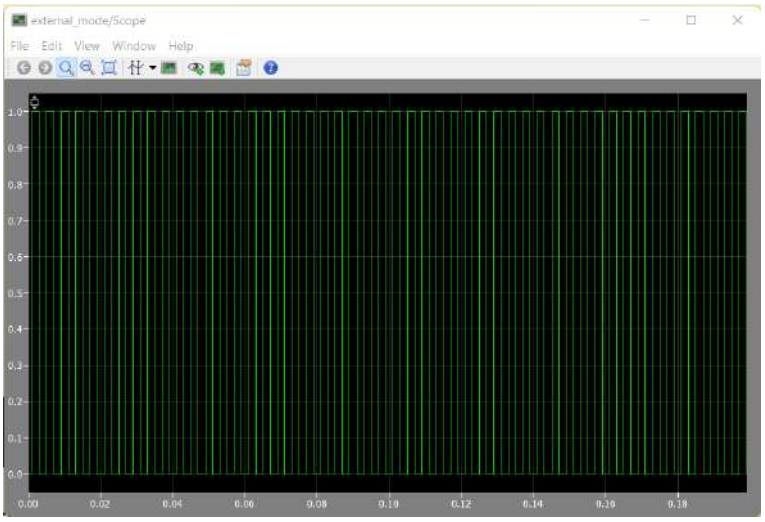


Figure 2.9: $N_{samples} = 200$

But if the number of samples is imposed as:
$N_{samples} = 600$
The total scope time will remain
$T_{scope} = 0.499 \quad [s]$
Because the number of samples exceeds the number of samples available.

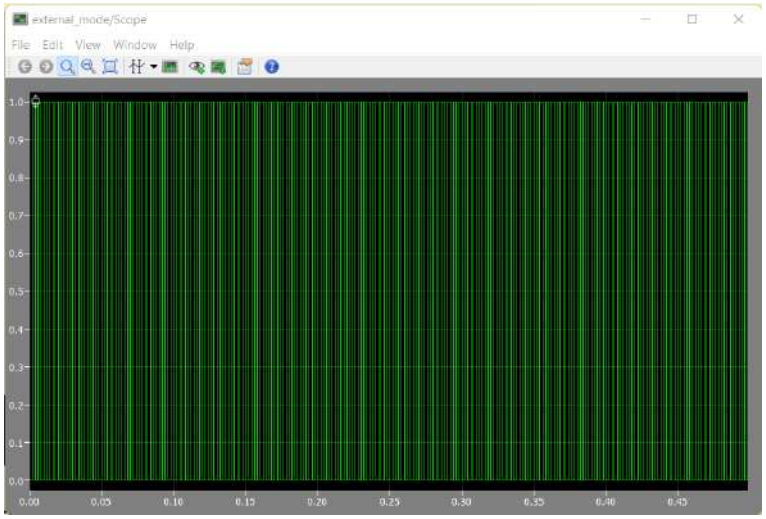


Figure 2.10: $N_{samples} = 600$

If the connection is successful, the Trigger controls will activate. There are two ways of triggering the first one is the auto-triggering mode where PLECS synchronizes the program with the Launch Pad automatically. There Is also the possibility to customize the synchronization between the software and the microcontroller by setting a specific trigger event. To do so the user must change the Trigger channel from "off" to the desired signal. The Scope will now show a small square indicating the trigger level and delay. By dragging the trigger icon the trigger level will change. The second way is the "manual trigger". In this way, the software will start the acquisition every time the user clicks on the icon "manual trigger".

Chapter 3

PLECS examples

In this chapter is shown how PLECS works when linked to a microcontroller. In this study, it is used a LAUNCHXL-F28069M microcontroller made by Texas Instruments. PLECS has designed specific masks for these tasks which are located in the library browser.
PLECS simulates dynamic systems. In order to use this software to communicate in real-time with a LaunchPad a suitable package of blocks is needed. The family of targets that are suitable for this software is called the TI C2000 target. In this family of microcontrollers, there is the LAUNCHXL-F28069M as well.
The C2000 Piccolo LaunchPad, LAUNCHXL-F28069M, is a complete low-cost development board for the Texas Instruments Piccolo F2806x devices and InstaSPIN technology. It offers an on-board JTAG emulation tool allowing direct interface to a PC for easy programming, debugging, and evaluation. In addition to JTAG emulation, the USB interface provides a universal asynchronous receiver/transmitter (UART) serial connection from the F2806x device to the host PC.
The communication between the host PC and TI C2000 Piccolo LaunchPad is made by the debug probe USB(XDS100v2). For proper communication, it has to be created an VCP (Virtual Communication Interface) and configured the correct COM peripheral. It has to pay attention that every LaunchPad has its own COM peripheral and every time a new MCU is connected to the host PC this has to be verified (the interface peripheral of the LaunchPad is COM 2).
In order to study the blocks PLECS uses for the communication between a host PC and a LaunchPad a PCB (printed circuit board)is connected to the microcontroller.
This board contains several hardware tools useful for control. It has a potentiometer, an encoder, a LED, and a seven-segment display.
The Datasheet of the PCB is shown in Appendix 3. This board is used to show the output of the MCU.
PLECS is connected with the LaunchPad with a direct interface but to see the output of the microcontroller the PCB is needed. During the example shown it will be noticeable how the board makes the outputs of the MCU visible to the user. Every tool is connected to the right jumper of the LaunchPad in order to guarantee the right communication between the MCU peripherals and the PBC ones. For instance, a green LED is connected to the jumper J14 where

the microcontroller has set the GPIO 10 for the I/O interface. During the following example, it is explained how the communication between the host PC, the MCU, and the PCB is made paying particular attention to the hardware elements of the PCB and the MCU.

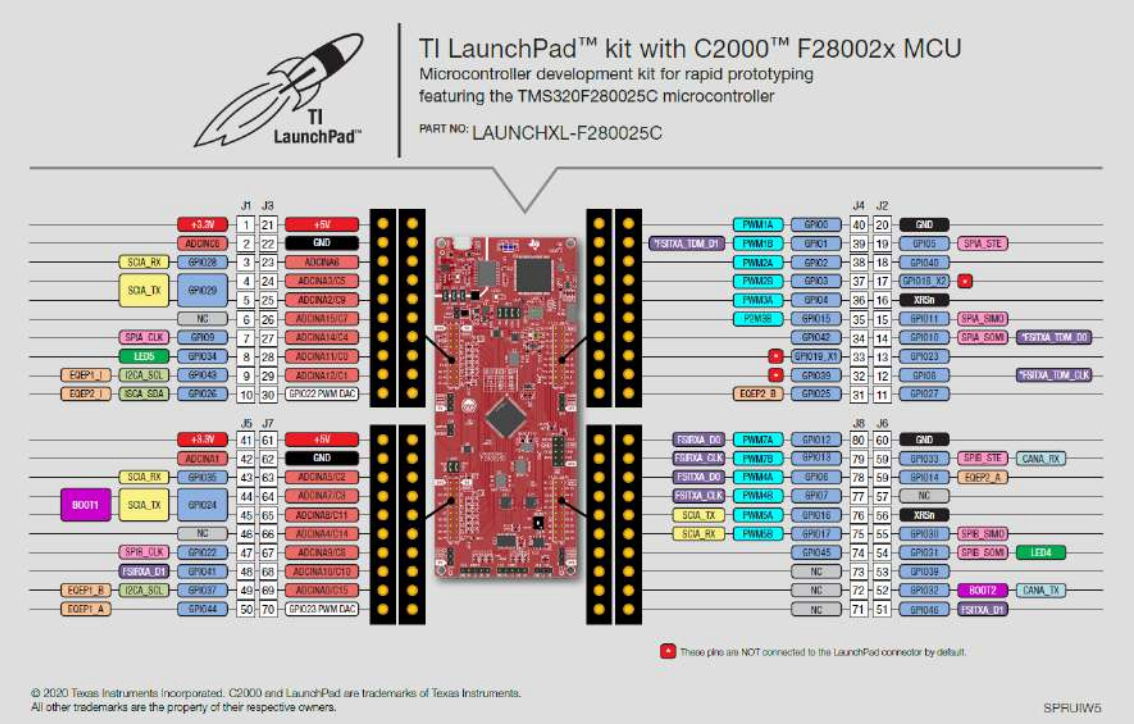


Figure 3.1: MCU peripherals

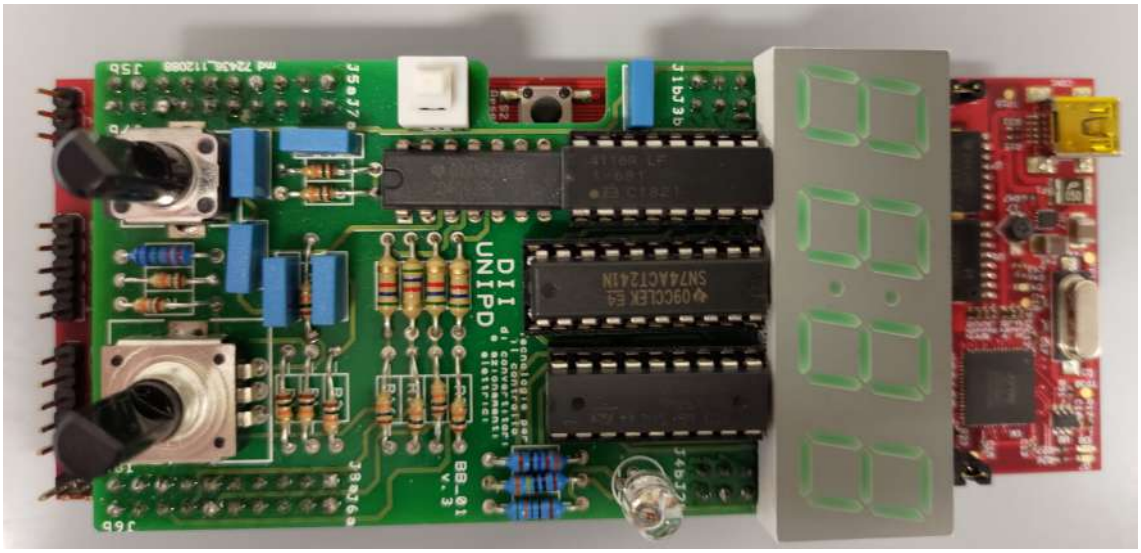


Figure 3.2: MCU and PCB front view

3.1 Example 1: Blinking a LED

The first example shows how to turn on and off a LED. There is a pulse generator block with the frequency and the duty cycle set and a Digital Out

mask all connected together as shown in the following picture. This mask represents the output of our microcontroller. In this case, the outputs are the led we have plugged into the microcontroller that is:

- GPIO 10 green led;
- GPIO 08 blue led;
- GPIO 06 red led.

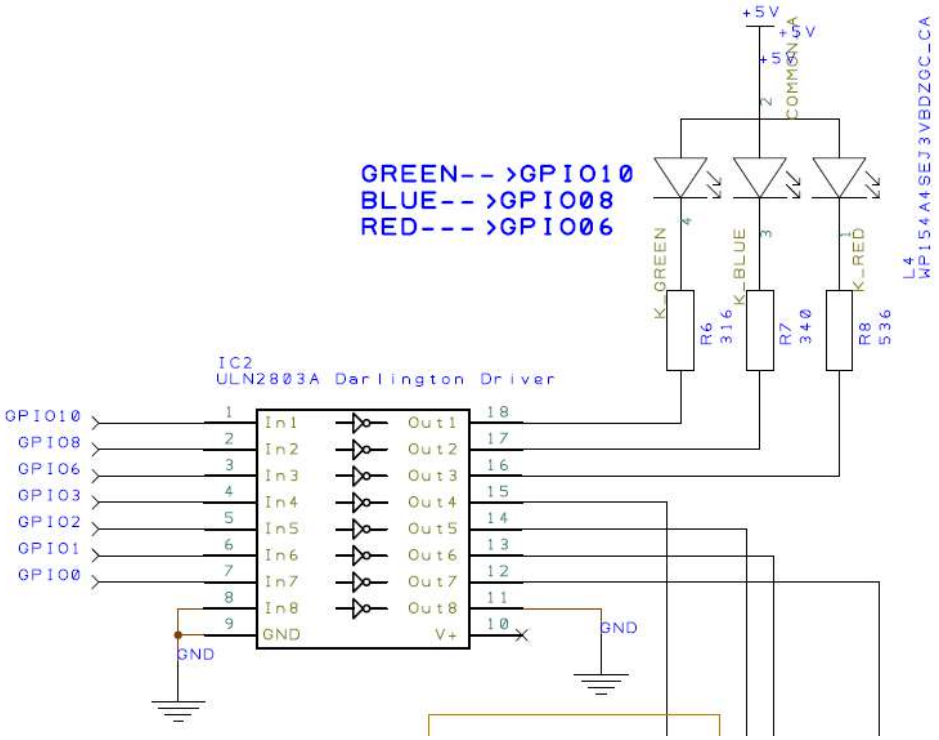


Figure 3.3: PCB LEDs Datasheet

To build the program click on *Coder* and then click on *Build*. The software will run the program and after a few seconds the led will turn on and off at the imposed frequency.

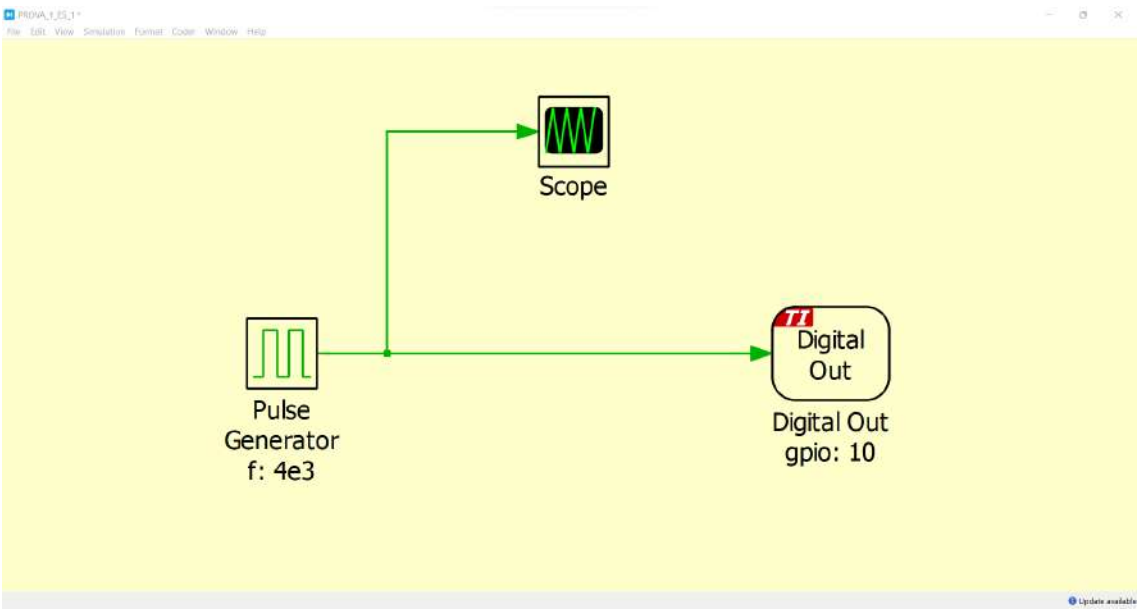


Figure 3.4: First example overall scheme

A scope shows the pulse generator waveform once the program is run in external mode.
Furthermore, the GPIO 10 of the LED has been connected to an oscilloscope to verify the period imposed in the mask.

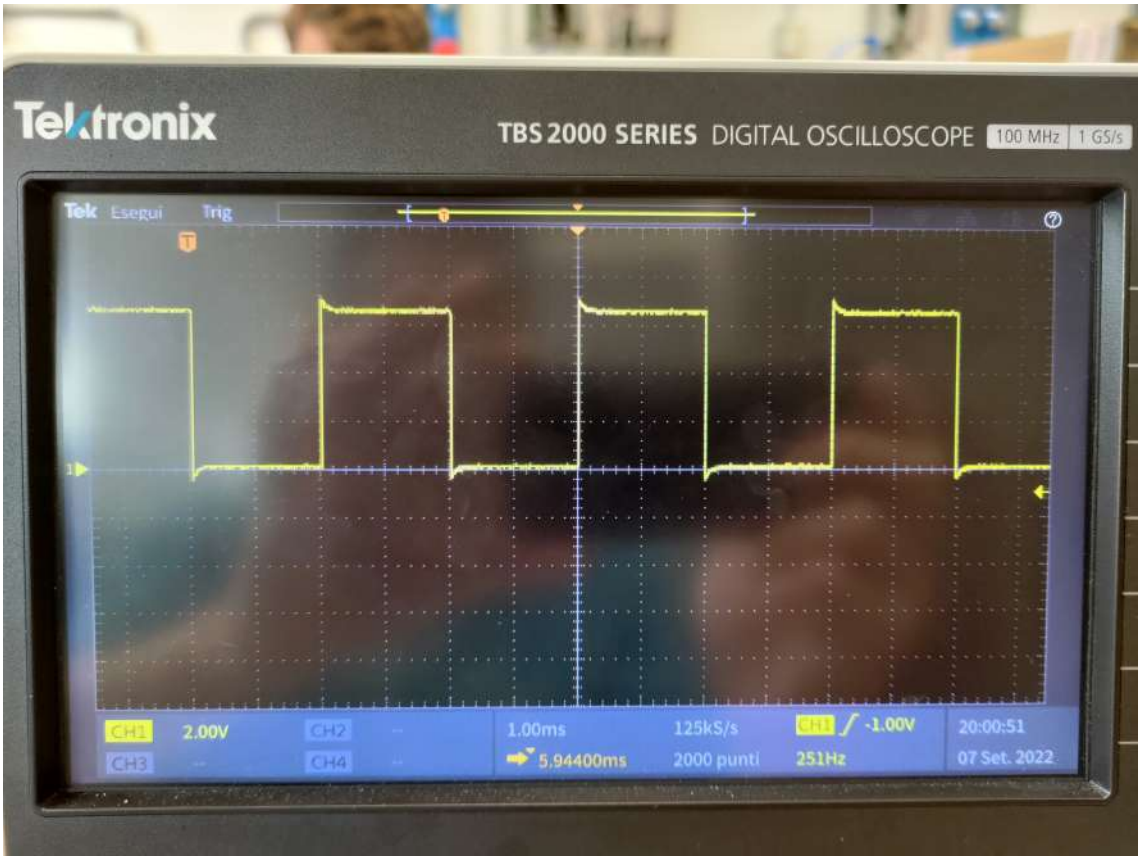


Figure 3.5: Pulse generator waveform shown in the oscilloscope

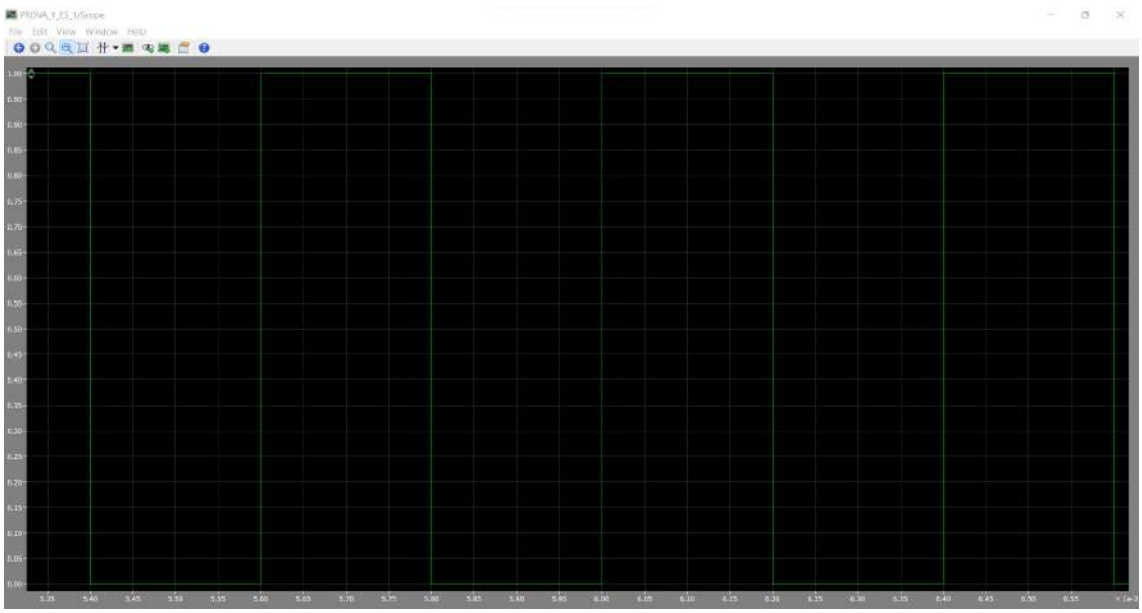


Figure 3.6: Pulse generator waveform shown in the PLECS scope

3.2 Example 2: Blinking two LEDs at different speeds

In this example, it is shown two different outputs with two different inputs. There are two different pulse generators with two different frequencies so that, once one runs the program, two different led turn on and off at different frequencies.
In PLECS there is no need for a sample-based pulse generator, but it can be used a time-based pulse generator. Using Simulink there was the problem of the sampling period was solved by introducing a rate transition that worked as a ZOH or as a unit delay. PLECS may contain blocks with multiple different discrete-periodic samples. In this case, the software calculates the sample time as the greatest common divisor of the periods and offsets of the individual sample times. It means that the software automatically set the sample time of the discrete-time blocks. In this example, the problem does not subsist because a time-based mask is used
The exercise is made by turning on and off both the green and the blue LED with two different pulse generators. The first one with a frequency of 1 Hz and the second one with a frequency of 2 Hz.

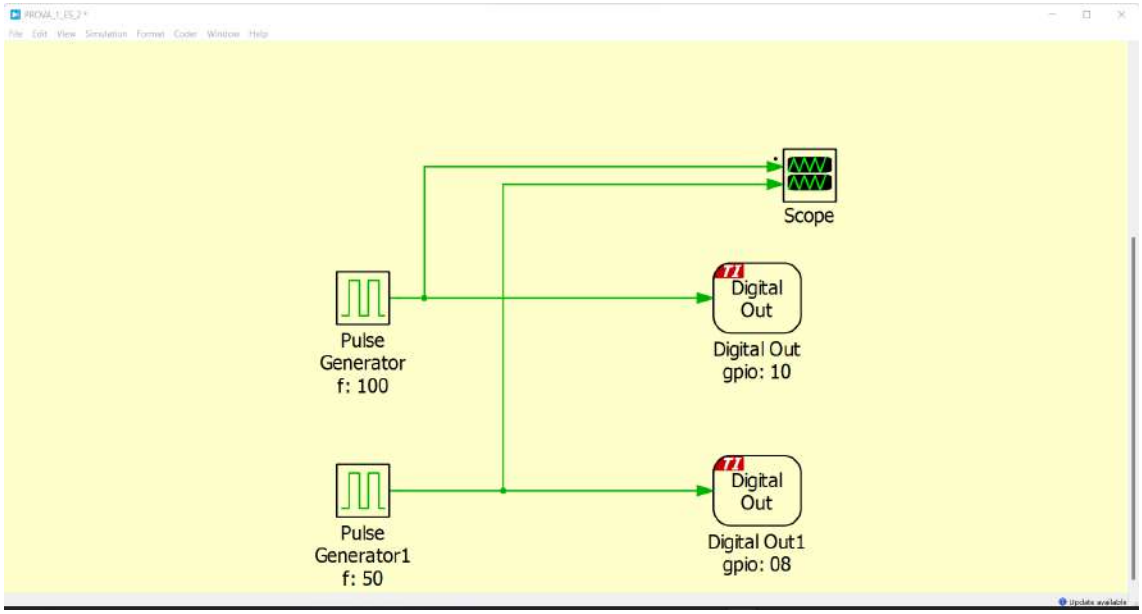


Figure 3.7: Second example overall scheme

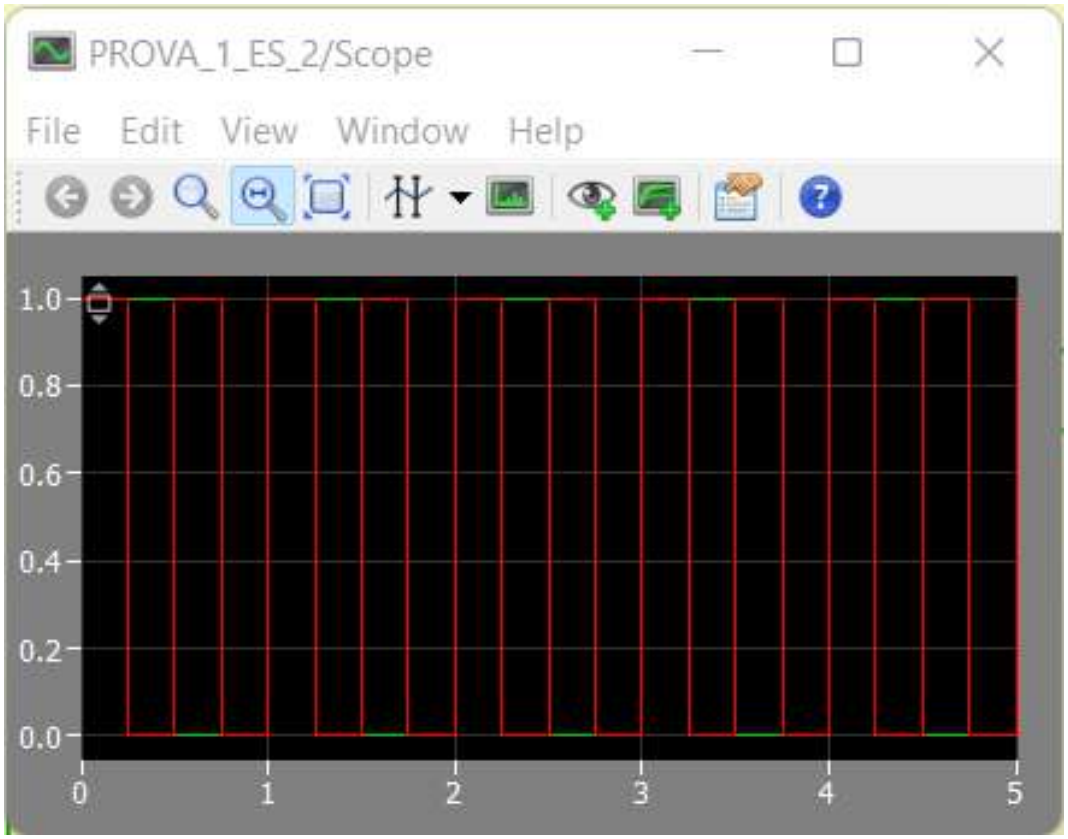


Figure 3.8: Second example scope

PLECS allows the user to build a multi-rate system in simulation mode. Since the discretization time of the software is equal for every block the user must build the multi-rate system by himself. With this mask, one can trigger the system with a sample time which is fixed by a pulse generator with a specific period. Notice that PLECS has two ways of running a simulation.

The first one is with a fixed step solver and the second one is with a variable step solver. If it is used a fixed-step solver, one must pay attention to the period of the pulse generator. The sample time of the triggered subsystem must be a multiple integers of the discretization time (by using the pulse generator the wave period is imposed so this number must be divided by two). In Fig. (3.9), there is a pulse generator with a frequency of 250 Hz with two different triggered subsystems. The first one is connected to a pulse generator with a frequency of 100 Hz and the second one is triggered by a pulse generator with a frequency of 500 Hz (notice that both periods are integer multiple of the default time discretization which is Ts = 1e-3 s).

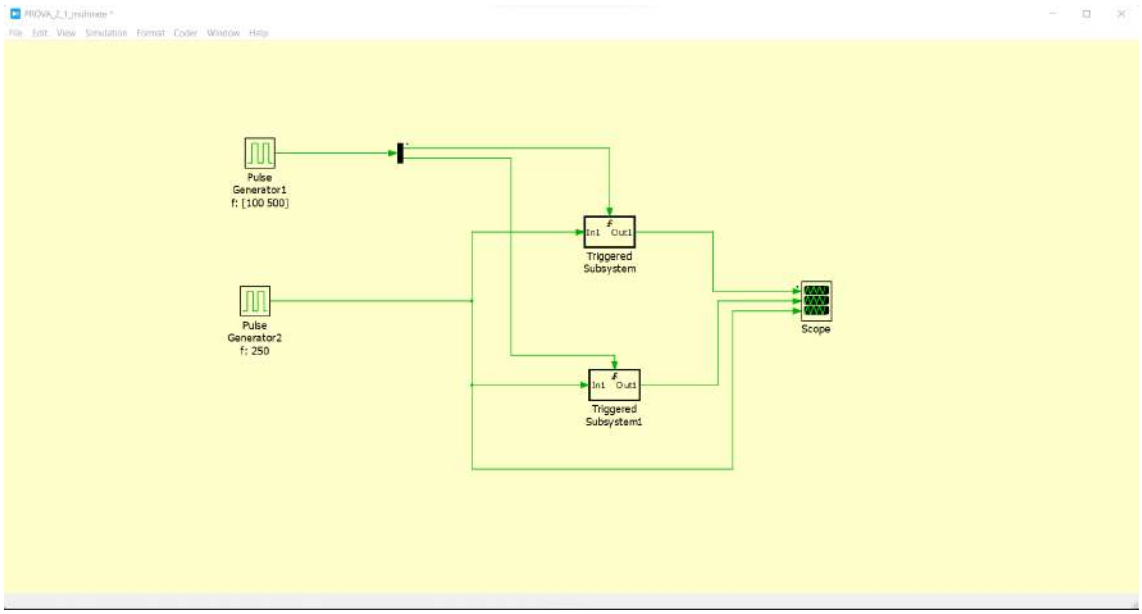


Figure 3.9: Scheme with two triggered subsystems

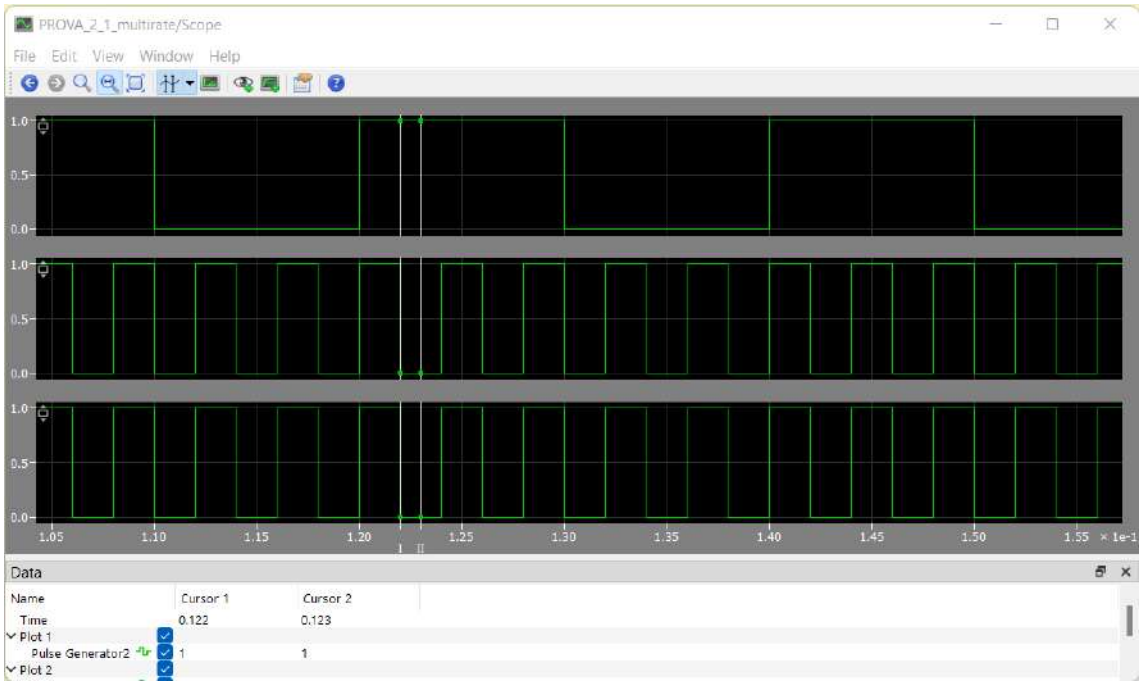


Figure 3.10: Scope waveforms

The first signal has aliasing but the second one no. Furthermore, in the picture above the discretization time is always 1e-3s regardless of the frequency of the triggered subsystem.
Instead of using a fixed-step solver, one can run the software with a variable-step solver. In Fig. (3.10), it is possible to set the period of trigger regardless of the discretization time. PLECS automatically set the sample time. It looks at the different sample times in the main window and then it uses the minor one. In Fig. (3.11) there is the same example shown before but this time the frequency on the triggered subsystems is higher so that one of them has a sample period that is shorter than the max discretization time step which is imposed in the simulation parameters.

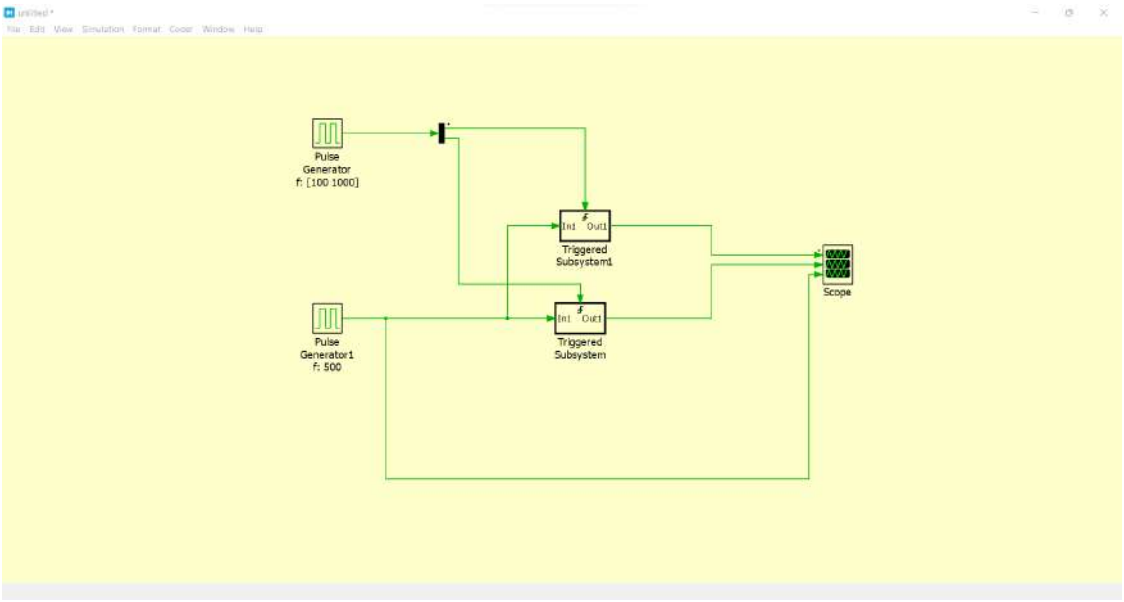


Figure 3.11: overall scheme

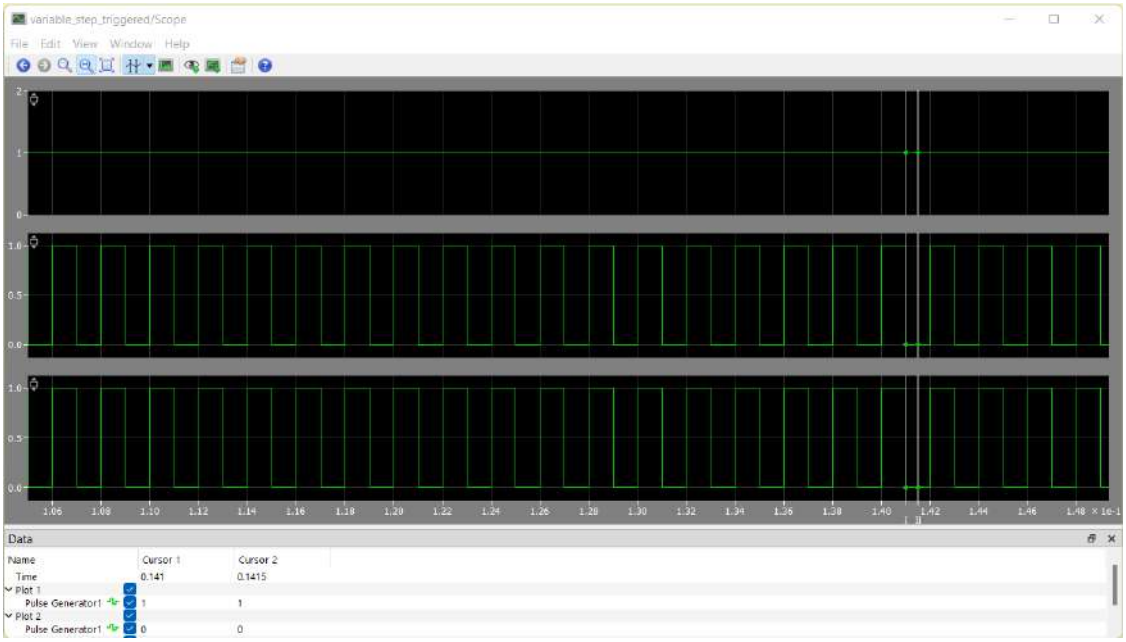


Figure 3.12: scope waveforms

In Fig. (3.13) it is possible to see that the sample time of the solver is 5e-4 s. By setting the frequency of the pulse generator that triggers the subsystem at 1000 Hz we trigger the subsystem every 5e-4 s (Ts = 1e-3 so every half period the wave goes up and down).
Another way to make a multi-rate system is by using the zero-order-hold (ZOH) block. This block the sample constant for a certain amount of time which can be set in the mask. In Fig. (3.13) there is the same code that is shown in Fig. (3.11) but this time there is the ZOH block. By running the program with a variable step solver mode, and by changing the sample time at 1e-3s there will not be a problem with aliasing.

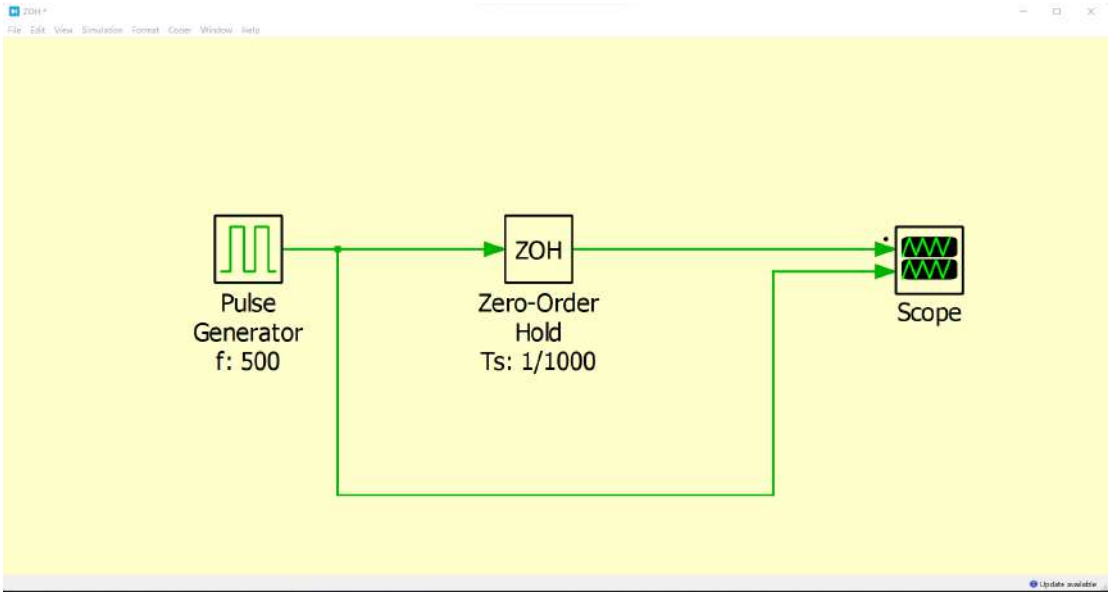


Figure 3.13: scope waveforms with ZOH block

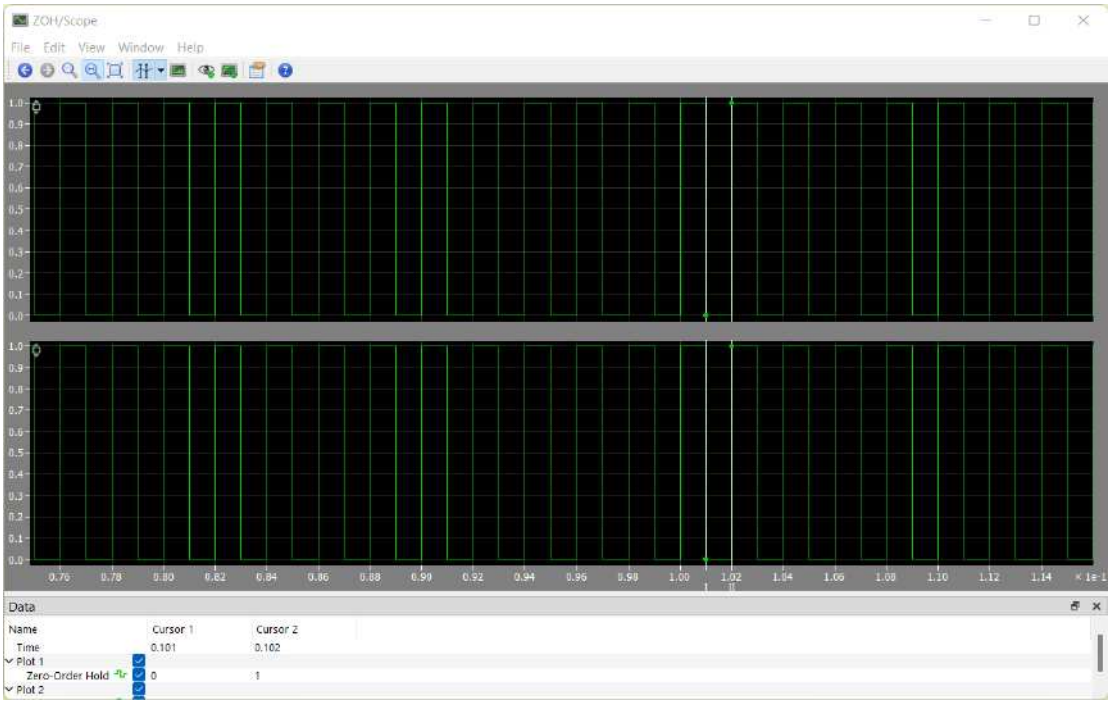


Figure 3.14: Scope waveforms with ZOH block

3.3 Example 3: Blinking a LED with Manual Switch

This exercise shows how to use the software to control LaunchPad. In this example, there are two constant masks one set with the number one and the other set with the number zero. Both are connected to a manual switch which allows the user to control the inputs. Then the manual switch is connected to the digital output which in this case is the green LED.
To control the software input while the program is running, there are two things to be done previously. First, the manual switch mask has to be dragged into the *Parameter Inlining* window and keep the Default behavior set on *Inline parameter values*.

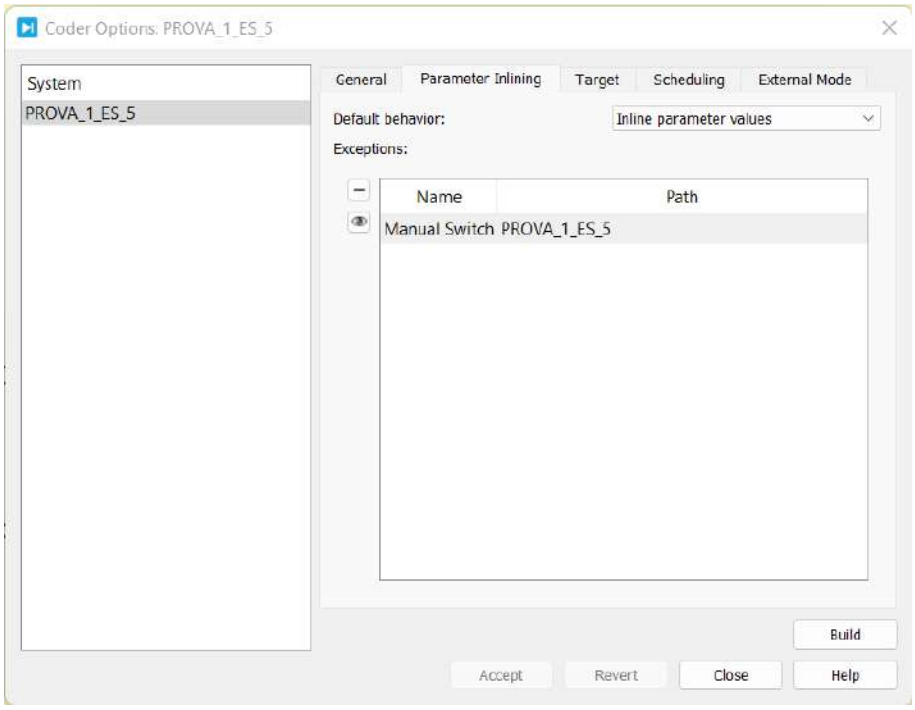


Figure 3.15: Parameter inlining window

Secondly, the program has to be run in External Mode so the user can choose which input value must feed the output. Once these settings are done, it is possible to control the switching on and off of our LED from the software.

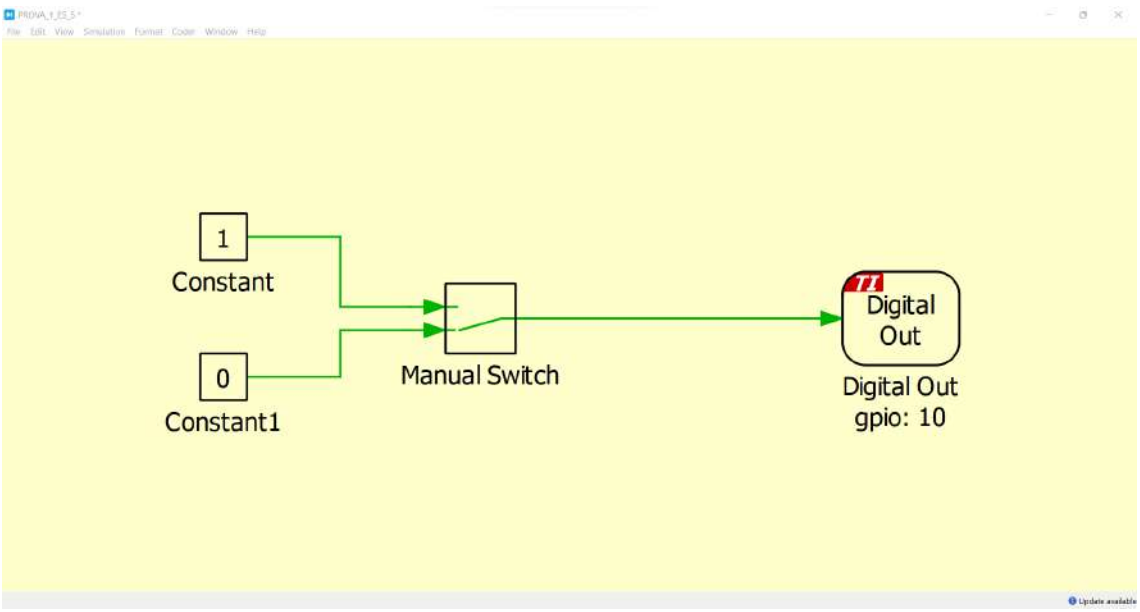


Figure 3.16: Third example overall scheme

3.4 Example 4: Blinking a LED with a hardware button

This simple example shows how to use PLECS to connect a digital input to a digital output. This exercise aims to connect a hardware input which in this case is a button plugged into the GPIO 52 to a hardware output which is the green LED. To do so the TI C2000 support block "Digital In" is used. In the setting block window, the input GPIO is chosen. Fig. (3.18) shows the Digital In block which feeds the Digital Out block.
In Fig. (3.18) it is shown the PCB where the button and the LED are connected. It is possible to see that when a user presses the button the LED blinks.

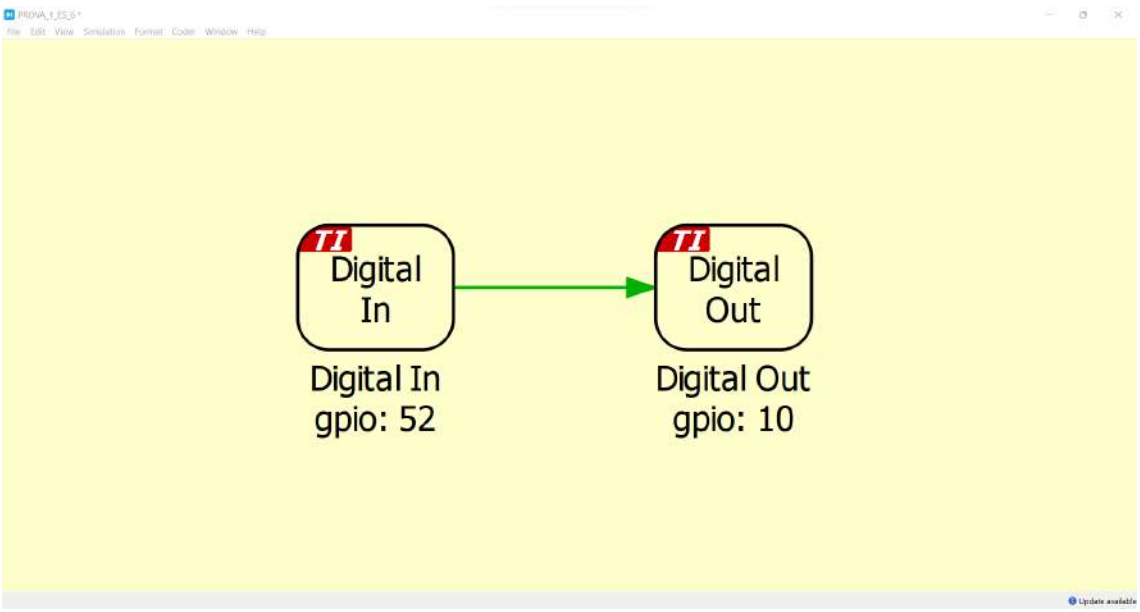


Figure 3.17: Fourth example overall scheme

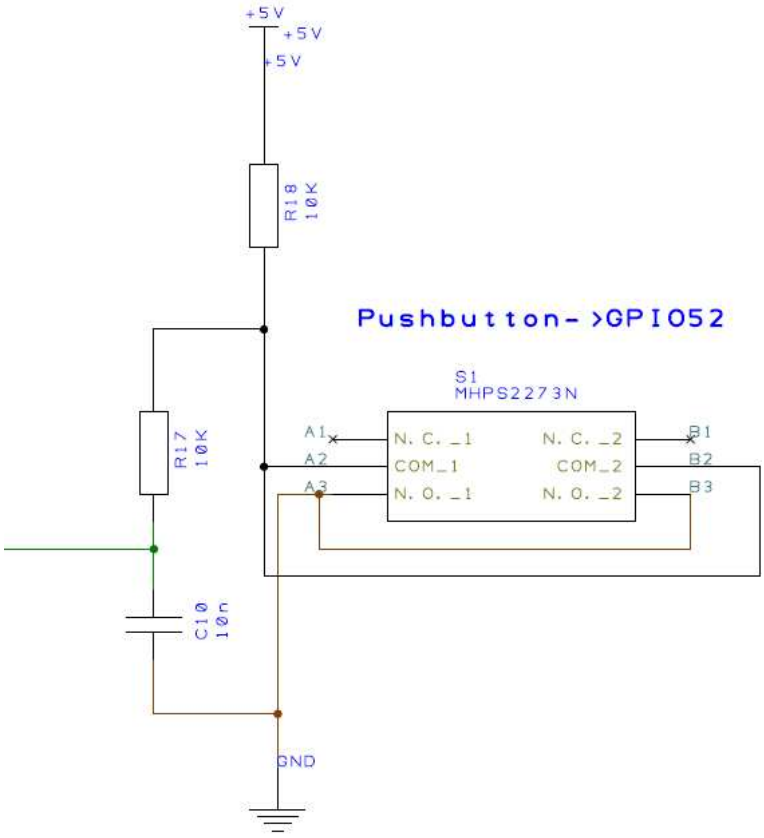


Figure 3.18: PCB Button Datasheet

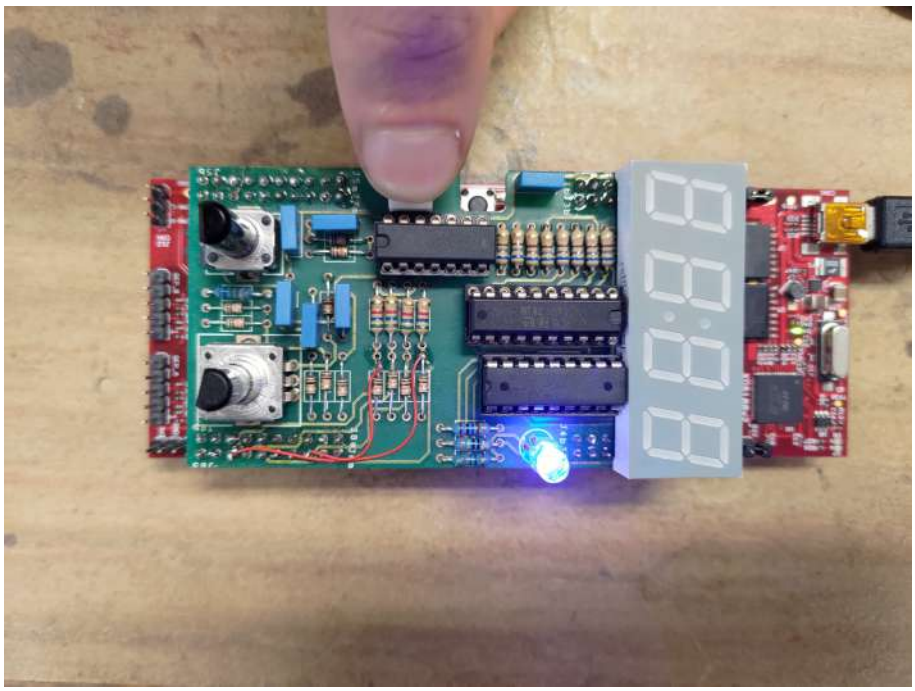


Figure 3.19: LaunchPad front view

3.5 Example 5: Digital Counter with C-Script block

This example implements a counter by using the C-script mask and a triggered subsystem.

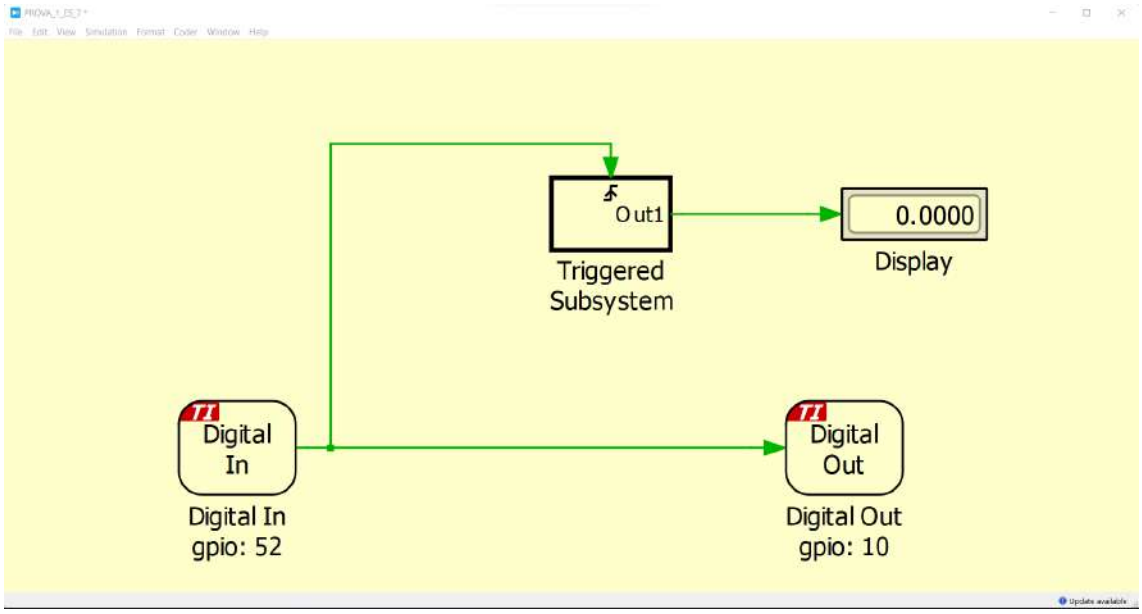


Figure 3.20: fifth example overall scheme

The triggered subsystem makes the inner blocks work every time it is fed with a trigger signal. Notice that it is not allowed to put a TI C2000 output block

inside a subsystem because PLECS wants the digital output block to always be in the main system.

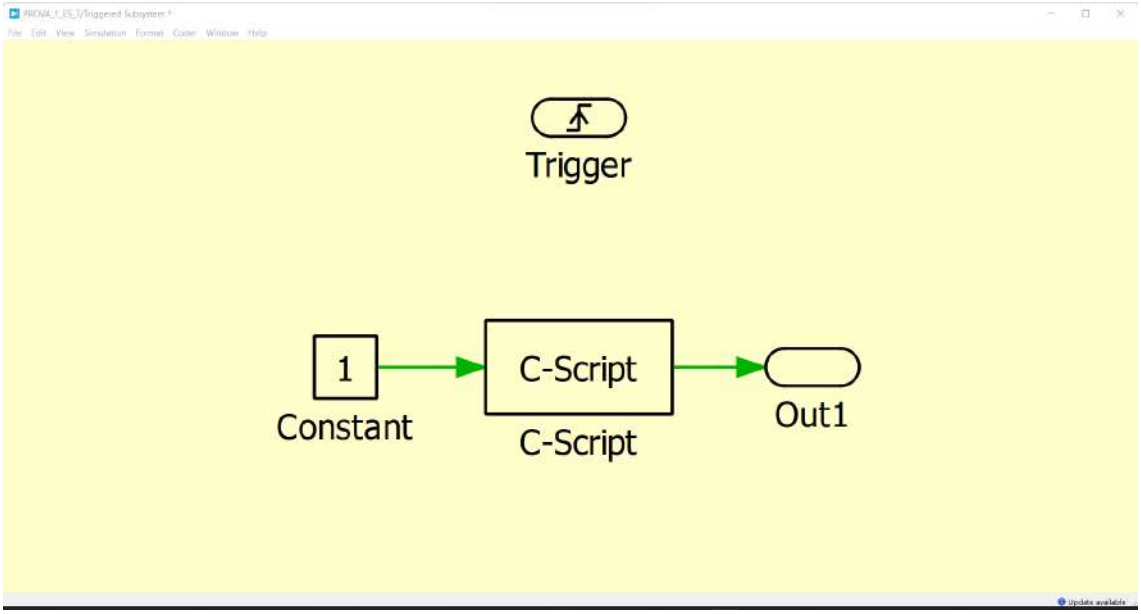


Figure 3.21: triggered subsystem scheme

C-script block can be used for implementing custom controllers and components. C-script is the more versatile mask in PLECS because it allows the user to write down a C code that is used for specific applications. This mask is composed of two windows the first one is the setup window and the second one is the code window.
In the setup window, it can be chosen the number of inputs and the number of outputs. Notice that even if the user changes the length of these terms, there always be only one input connection and one output connection. This means that multiplexers and demultiplexers must be used to feed the C-script block with more signals.
The standard languages that PLECS uses are C99, C90, and C11 and the default language is C90. The sample time setting is a key parameter that controls when the C-script mask is called. If the block works with continuous signals, "sample time" has to be put at zero. In this case, every time the solver takes a step, the C-script block is executed.
If the block work with discrete signals the sample time must be set with a positive number. In this case, the C-Script block is executed at discrete regular intervals defined by this sample time. In this example, a semi-continuous model is used because are used continuous signals, but the C-script block produces only discrete values that do not change in sign (no zero-crossing signal). In this case, the sample time is set to -1.

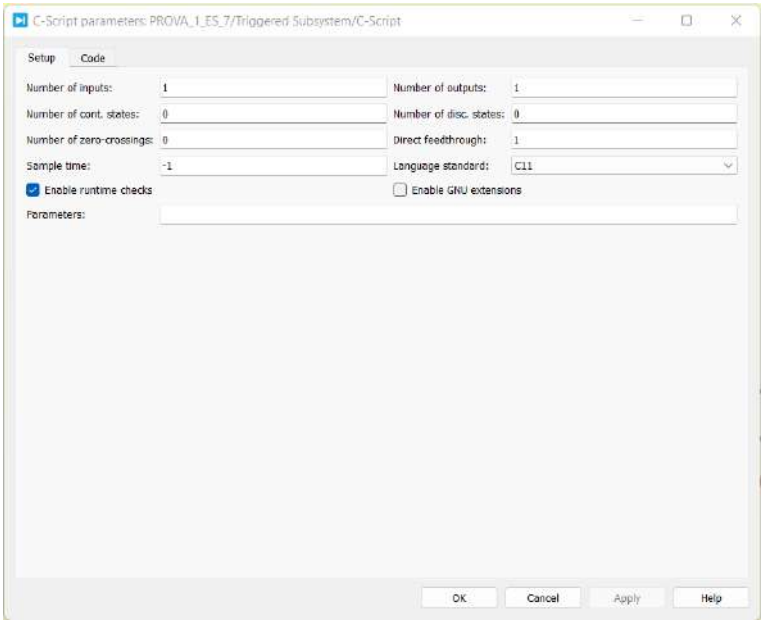


Figure 3.22: C-Script setup window

In the code window there are four majors windows:

- Code declarations
- Start function code
- Output function code
- Update function code

PLECS works with a fixed step equal to the discretization time imposed in the coder option. Every Major step is thus computed with a fixed period as shown in Fig. (2.1).
Code declarations are provided for defining global variables and for including C header files like math.h and stdio.h. All variables that are declared here are globally seen to all functions within the C-Script mask.
The variables used can be of different types. They could be either integers or real numbers for example. The C-Script block as said use the C language and so these values have to be declared in the right way.

- int: it represents an integer with a resolution of 16 bits.
- float: it represents a real number with a precision of 32 bits.
- double: it represents a real number with a precision of 64 bits.
- real_t: it represents whatever data type is handy to handle single precision or higher.

In the start function code, the internal state variables are set. In the Output function code, the C code procedure is written down. Here the initial states can be declared and the discrete values can be updated too. The problem in doing this is that the output function is called at least once during every

simulation. But it can also be called more than once and so the program might update the discrete values unproperly. In the Update function, the program will update the discrete value. This function is called once at the end of the sample time. For this reason, is correct to use this section rather than the output function to update the discrete values.
The C-script block contains several built-in macro functions that can be used to interact with the solver. Some of these macros are:

- InputSignal(i,j) — Reference the i-th signal of the j-th C-Script block input
- OutputSignal(i,j) — Reference the i-th signal of the j-th C-Script block output
- DiscState(i) — Reference a discrete state with index i
- CurrentTime — retrieve the current time
- SetErrorMessage(msg) — Abort the simulation with an error message

This is the code for this exercise.

Listing 3.1: Code declarations

```
static int ik;
```

Listing 3.2: Start function code

```
ik = 0;
```

Listing 3.3: Output function code

```
Output(0) = ik;
```

Listing 3.4: Update function code

```
ik++;
```

3.6 Example 6: Display

The next example shows how to build a counter from 0 to 9999. Once a user presses the button on the Launchpad the display will increase its counter. The microcontroller is plugged into a seven-segment display with four digits. This example aims to use the C-Script block to design a counter.

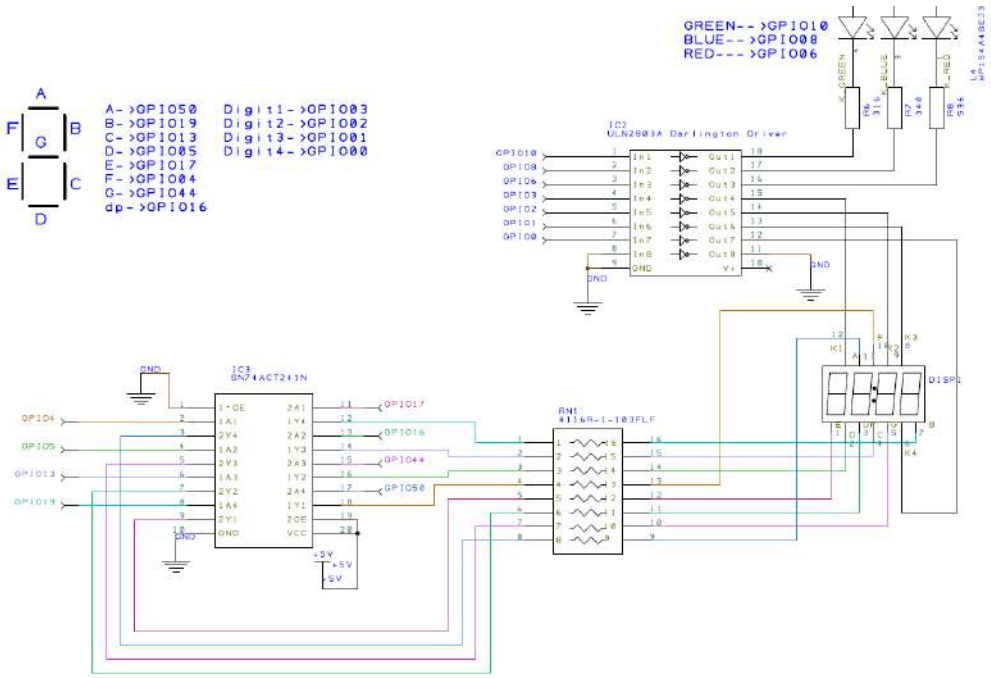


Figure 3.23: PCB Display Datasheet

The exercitation is split into two main tasks. The first one is to control the turning on and off of the digits and the second one is to increase the counter by one each time someone presses the button on the LaunchPad.
PLECS doesn't have a free-running counter block like Simulink, so it is possible either to use a C-Script block as used in the previous example or to build a counter-free running using PLECS blocks. Considering that the display has four digits the counter has to be reset every time it reached the value of four. This is possible by using a C-Script. Notice that the delayed mask has got as default sample time 1. This will cause an error. In this example the C-Script block works in a semi-continuous domain, so the sample time has to be changed from 1 to -1.

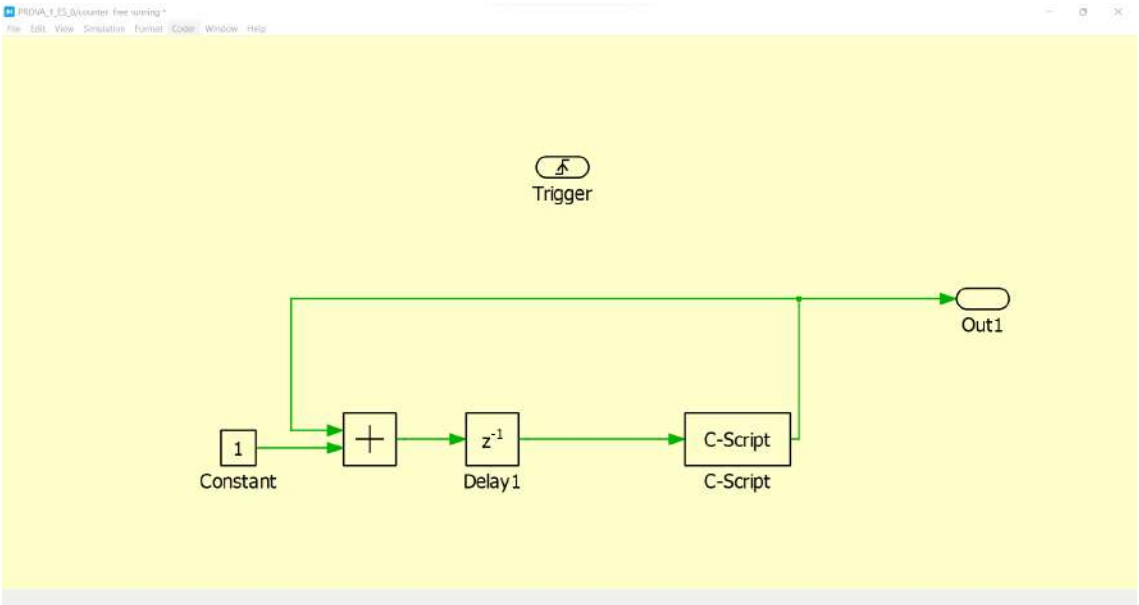


Figure 3.24: counter

Listing 3.5: Output function code

```
if(Input(0)>3)
{
Output(0)=0;
}
else
{
Output(0)= Input(0);
}
```

The digits are regulated by a pulse generator and a counter which turn on and off the digits one by one in succession. The output of the counter feeds a C-Script block which turns on and off the corresponding digit.
The C-Script block has thus four outputs that are connected to the four digits. Furthermore, there is another output that we will use to regulate the segments of each digit.

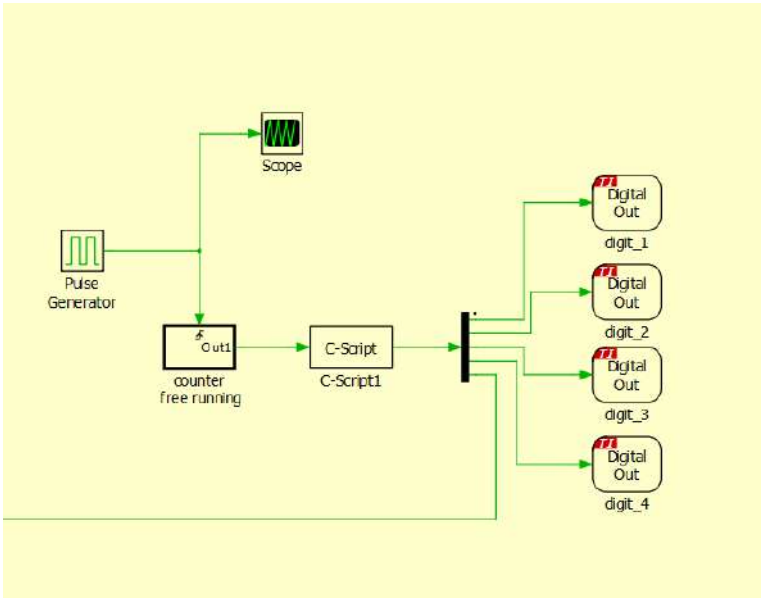


Figure 3.25: part of the scheme where there are the digit

Listing 3.6: Output function code 1

```
int n = Input(0);

switch (n)
{
case 0:

Output(0) = 1;
Output(1) = 0;
Output(2) = 0;
Output(3) = 0;

break;
case 1:

Output(0) = 0;
Output(1) = 1;
Output(2) = 0;
Output(3) = 0;
```

Listing 3.7: Output function code 2

```
break;
case 2:

Output(0) = 0;
Output(1) = 0;
Output(2) = 1;
Output(3) = 0;

break;
case 3:

Output(0) = 0;
Output(1) = 0;
Output(2) = 0;
Output(3) = 1;
break;
}
Output(4) = Input(0);
```

Notice that the frequency of the pulse generator must be chosen properly. The discretization step size of the software is imposed at 0.001 as default. The sample period of the block is set automatically as half of the period of the pulse generator. This term must be an integer multiple of the base step otherwise PLECS will show an error. In this example, I set a frequency of 500 Hz (the sample time is 1/ (500*2) = 0,001).
The main part of this example is to turn on the right segments and digits every time one pushes the button. For doing so a counter is needed that is fed with the input signal so input value can increase. Then the signal of the counter and the output of the C-Script block used to set the digits must be connected to another C-Script block with a multiplexer.
In this new C-Script, the input is split into units, tens, hundreds, and thousands. Each digit from 0 to 3 corresponds to one of these divisions. The number itself is made up by turning on the right segments.
Each segment is a digital output that is pinned in a specific GPIO.

Listing 3.8: Code declaration

```
#include <math.h>

int x = 0;
int th = 0;
int cent = 0;
int un = 0;
int dec = 0;
```

Listing 3.9: Output function code

```
int n = Input(0);
int digit = Input(1);
th = ceil(n/1000);
n=n-th*1000;
cent = ceil(n/100);
n=n-cent*100;
dec = ceil(n/10);
n=n-dec*10;

un = n;
switch (digit)
{
    case 0:
        //unita
        x=un;
        break;
    case 1:
        //decine
        x=dec;
        break;
    case 2:
        //centinaia
        x=cent;
        break;
    case 3:
        //migliaia
        x=th;
        break;
}
```

Listing 3.10: Output function code

```
  switch (x)
{
case 0:
Output(0) = 1;
Output(1) = 1;
Output(2) = 1;
Output(3) = 1;
Output(4) = 1;
Output(5) = 1;
Output(6) = 0;
break;
case 1:
Output(0) = 0;
Output(1) = 1;
Output(2) = 1;
Output(3) = 0;
Output(4) = 0;
Output(5) = 0;
Output(6) = 0;
break;
case 2:
Output(0) = 1;
Output(1) = 1;
Output(2) = 0;
Output(3) = 1;
Output(4) = 1;
Output(5) = 0;
Output(6) = 1;
break;
```

Listing 3.11: Output function code

```
  case 3:
Output(0) = 1;
Output(1) = 1;
Output(2) = 1;
Output(3) = 1;
Output(4) = 0;
Output(5) = 0;
Output(6) = 1;
break;

  case 4:
Output(0) = 0;
Output(1) = 1;
Output(2) = 1;
Output(3) = 0;
Output(4) = 0;
Output(5) = 1;
Output(6) = 1;
break;

  case 5:
Output(0) = 1;
Output(1) = 0;
Output(2) = 1;
Output(3) = 1;
Output(4) = 0;
Output(5) = 1;
Output(6) = 1;
break;
case 6:
Output(0) = 1;
Output(1) = 0;
Output(2) = 1;
```

Listing 3.12: Output function code

```
Output(3) = 1;
Output(4) = 1;
Output(5) = 1;
Output(6) = 1;
break;
case 7:
Output(0) = 1;
Output(1) = 1;
Output(2) = 1;
Output(3) = 0;
Output(4) = 0;
Output(5) = 0;
Output(6) = 0;
break;
case 8:
Output(0) = 1;
Output(1) = 1;
Output(2) = 1;
Output(3) = 1;
Output(4) = 1;
Output(5) = 1;
Output(6) = 1;
break;
case 9:
Output(0) = 1;
Output(1) = 1;
Output(2) = 1;
Output(3) = 1;
Output(4) = 0;
Output(5) = 1;
Output(6) = 1;
break;
}
```

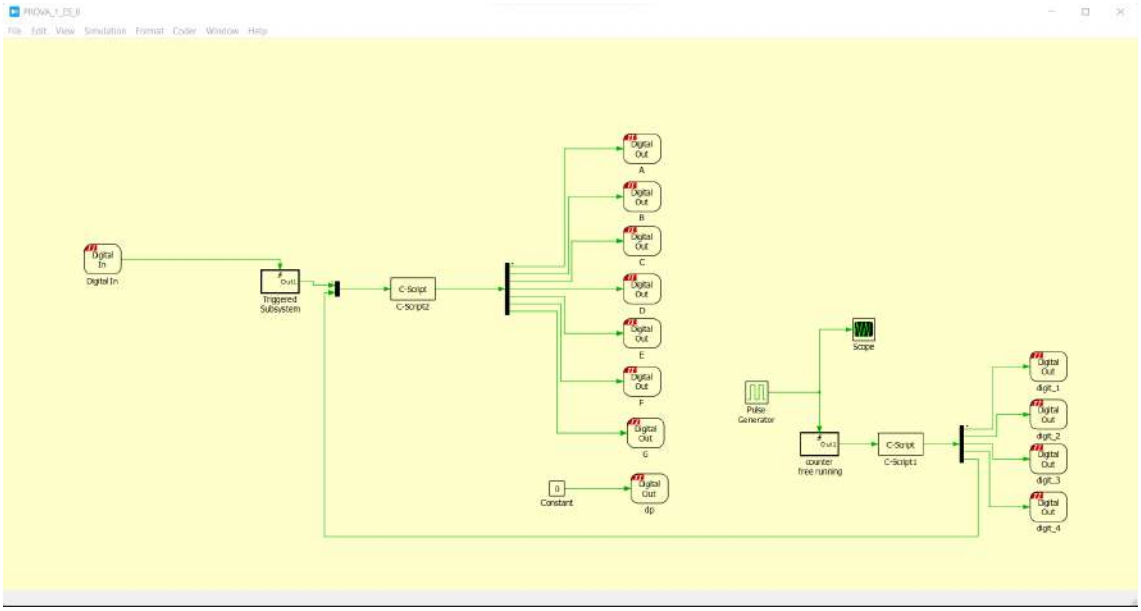


Figure 3.26: Sixth example overall scheme

3.7 Example 7: Display the position counted by an encoder

This example shows how the quadrature encoder counter works. Instead of pushing a button, the position of the encoder is used to increase the display values.

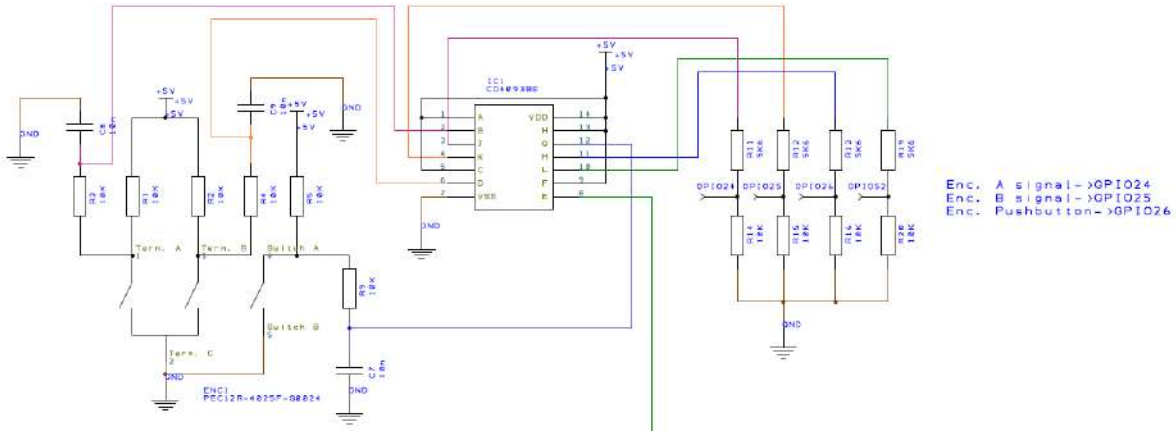


Figure 3.27: PCB Encoder Datasheet

The encoder which is plugged into the microcontroller is a quadrature encoder. It has three different signals. QEPA, QEPB, and QEPI. QEPA and QEPB are shifted 90° out of phase in order to understand the direction of rotation. The counter value will increase when the direction of rotation results in the rising edge of B following the rising edge of A and will decrease when the direction of rotation results in the rising edge of A following the rising edge of B.
QEPI is the reset signal. it is possible to reset the position counter by giving an external signal from the Launchpad. In PLECS there is no possibility of imposing signal inputs to reset the encoder counter from software. It is possible

to choose whether run the encoder on free-wheeling(first method) or reset the counter with an index pulse(second method). By clicking on the counter reset method in the encoder tab, there will be the possibility to choose <reference> as the reset method as well. Instead of writing down numbers, PLECS users can record variables that can be used by blocks. In the simulation parameters window, there is a section called "initialization" where is possible to write down the variable in the Model initialization commands. In this example, the encoder block is used, which can be reset in three different ways. By choosing <reference> PLECS will use the variable stored in the initialization window to select the first or the second method of reset. Let's write down the variable reset_method = 1. That means that if in the encoder mask there is written reset_method in the counter reset method the counter will work as it was set on free running (if reset_method=2 the counter will reset with an index impulse).

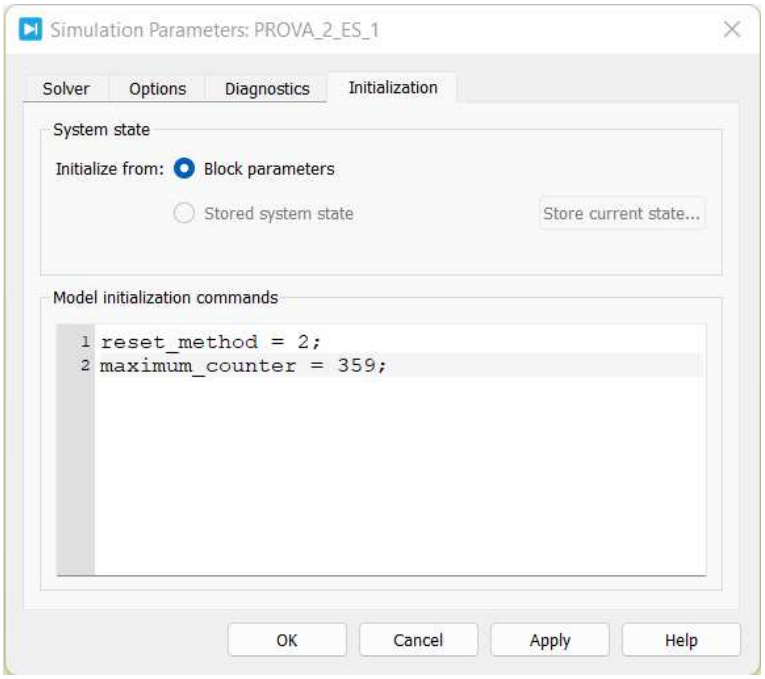


Figure 3.28: Simulation parameters scheme

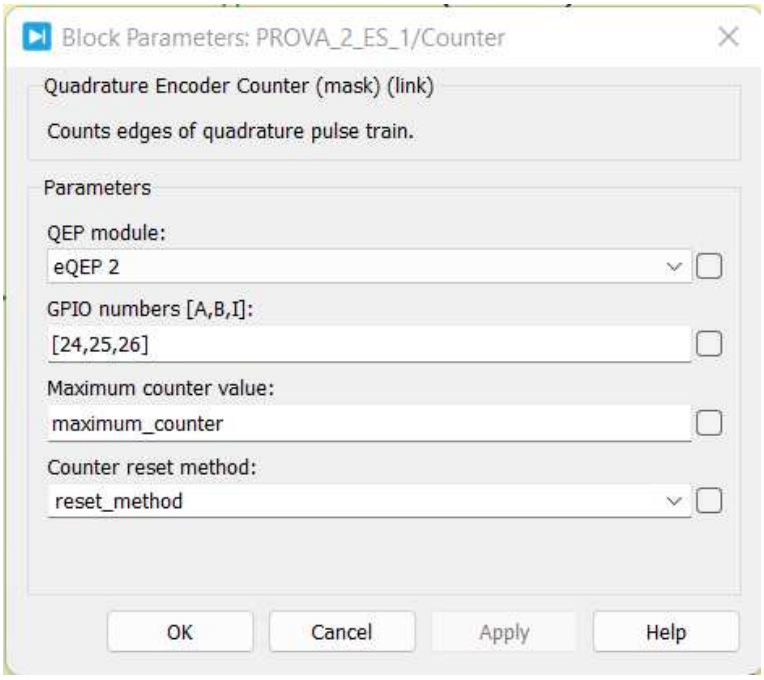


Figure 3.29: QEP block scheme

The microcontroller used in this study has two peripheral modules for the encoder eQEP1 and eQEP2. in the LaunchPad, it is used the eQEP2 and the three signals QEPA, QEPB, and QEPI are plugged into the GPIO24, GPIO25, and GPIO26 respectively.
Another parameter that can be set is the maximum counter value. This value must be set correctly because its value plus 1 must be a multiple of four since the QEP module counts all edges of A and B. In this exercise, the maximum counter value is 9999 so there isn't any problem. The outputs of the encoder are the current counter value(c), the index pulse(i), and the latched counter value from the previous index pulse(ic).

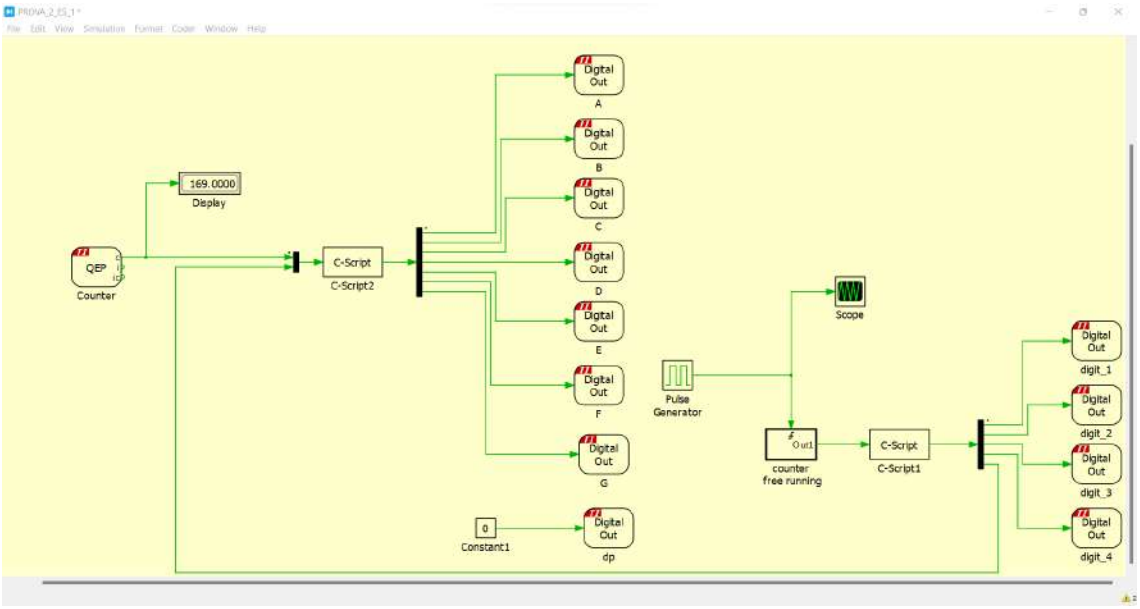


Figure 3.30: seventh example overall scheme

3.8 Example 8: Display a number evaluated by an ADC

The next example shows the previous scheme but in this exercise, the encoder block is replaced with the ADC block.

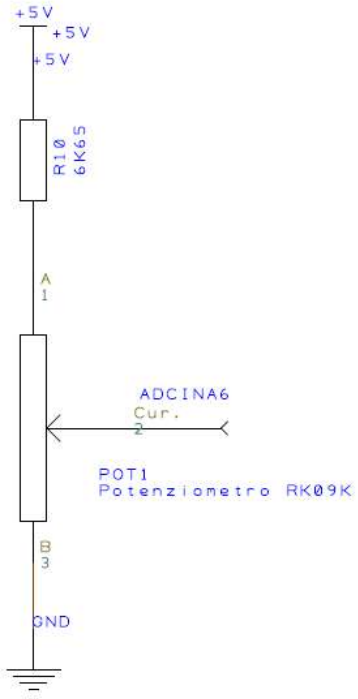


Figure 3.31: PBC Potentiometer Datasheet

The analog-to-digital converter (ADC) that the MCU uses is a 12-bit converter with 16 possible inputs which could be sampled simultaneously or sequentially. Inputs are split into two registers of 8 bits each of which is connected to its sample-and-hold circuit by a multiplexer. The MCU has two sample-and-hold circuits, one connected to the first register and the other to the second register. Since there are 16 different inputs there are also 16 different start-of-conversion (SOCx) triggers available. This MCU can convert one signal at a time. When the SOCx signal triggers the ADCx this one starts converting the analog input. If there are two simultaneous SOCx inputs the ADC converts the signal that is linked with the SOCx lower (SOC0 triggers before SCO7).
Different from MatLab, PLECS does not allow the user to customize the SOCx for the ADCx input and so every ADCx converts with the same SOC0. The reference voltage goes from 0 V to 3.3 V which is also the output of the mask with a precision of 4096 bits. Notice that the board which is connected to the microcontroller has a reference voltage of 5 V so to ensure that the input signal is within the range of the MCU is used a voltage divider.
The ADC block output signals represent the measured voltage at the ADC pin. If the ADC task output is the source of a control task trigger, then the control task will execute once the last ADC channel is covered. The ADC block has several parameters that can be customized.

The first one is the trigger source. Users can select whether the trigger of the starting of conversion is automatic or is made by an external signal. Then it is possible to set the input port from 0 to 15 and the ADC submodules if there are ones. In this case, peripheral 6 is set with the submodule ADC A.
Furthermore, it can be chosen both the offset for the scaled input signal that is leaved at 0 and the scale factor for the input signal (in this exercise is set it to 1000 so the maximum value will be 3300). Only if the user wants to use the offline mode, he must set the resolution in bits and the voltage reference as well.
In this example, the maximum value on the display isn't 3300 because the voltage divider the engineer had connected to the board was designed to reach a maximum value of less than 3300.

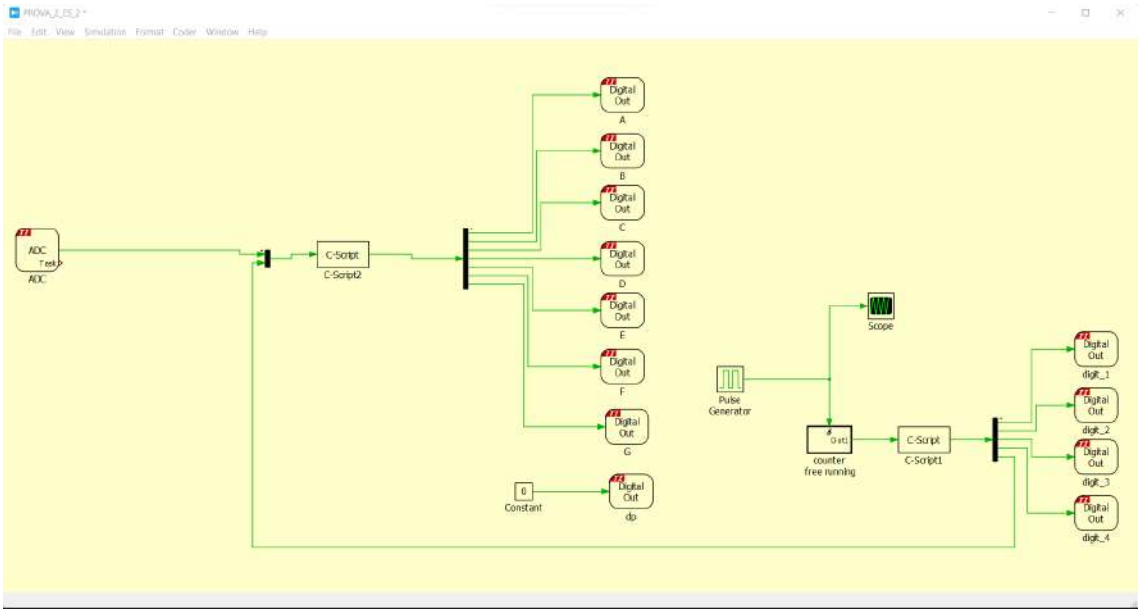


Figure 3.32: eight example overall scheme

3.9 Example 9: LPF filter in order to reject the disturbance

In the previous example, it has been seen that the ADC output signal has high disturbances. To avoid this problem, it is designed a low-pass filter.

$$H(s) = \frac{1}{\tau s + 1} \tag{3.1}$$

Imposing a frequency of bandwidth:

$$f_{bw} = 1 \quad [Hz]$$

$$w_{bw} = 2\pi f_{bw} = 6.2832 \quad [\frac{rad}{s}]$$

The pole of the LPF is:

$$\tau = \frac{1}{w_{bw}} = 0.1592 \quad [s]$$

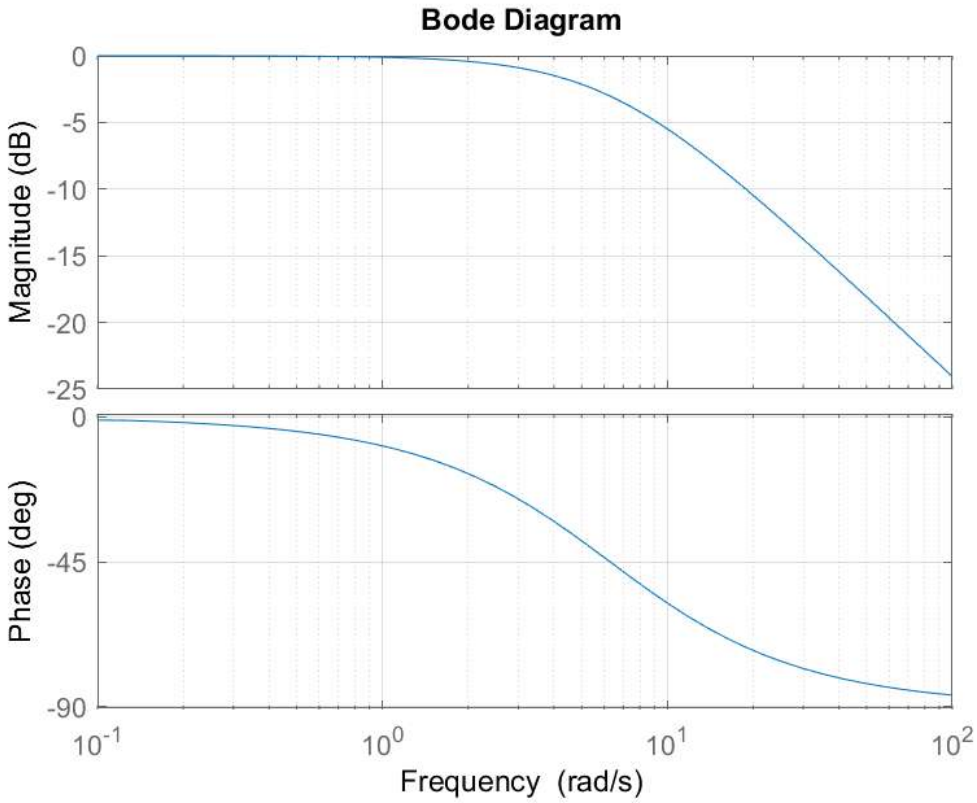


Figure 3.33: Bode diagram of a low-pass filter

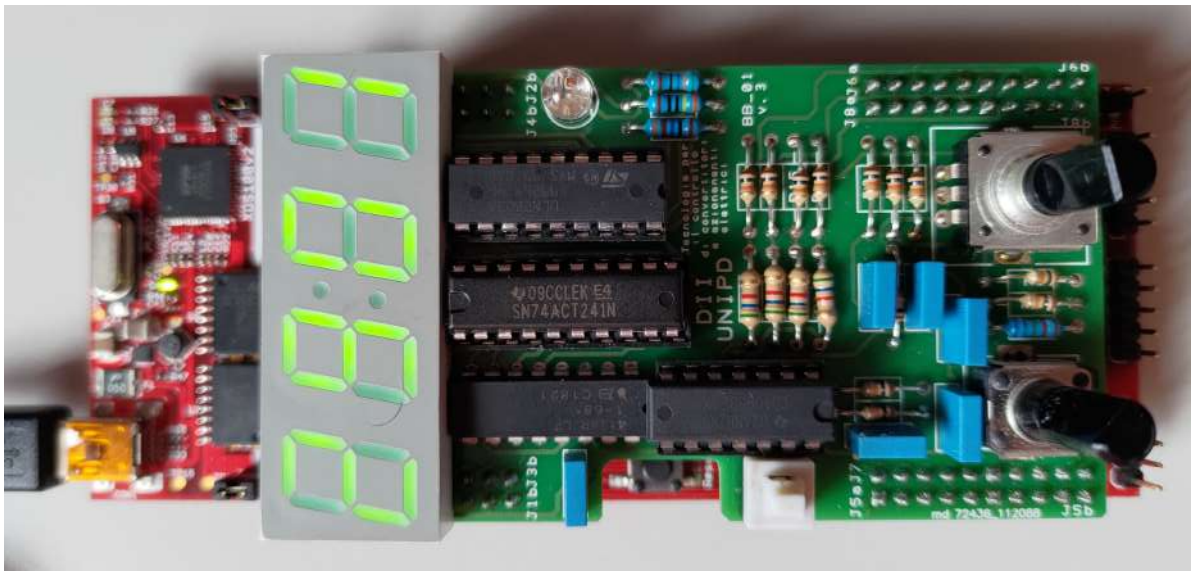


Figure 3.34: LaunchPad

Then by using the Tustin approximation with a sample time of Ts = 1e-3 s

$$H(z) = \frac{0.003132z + 0.003132}{z - 0.9937}$$

With this low pass filter, we are now able to reject the high disturbance of the ADC.

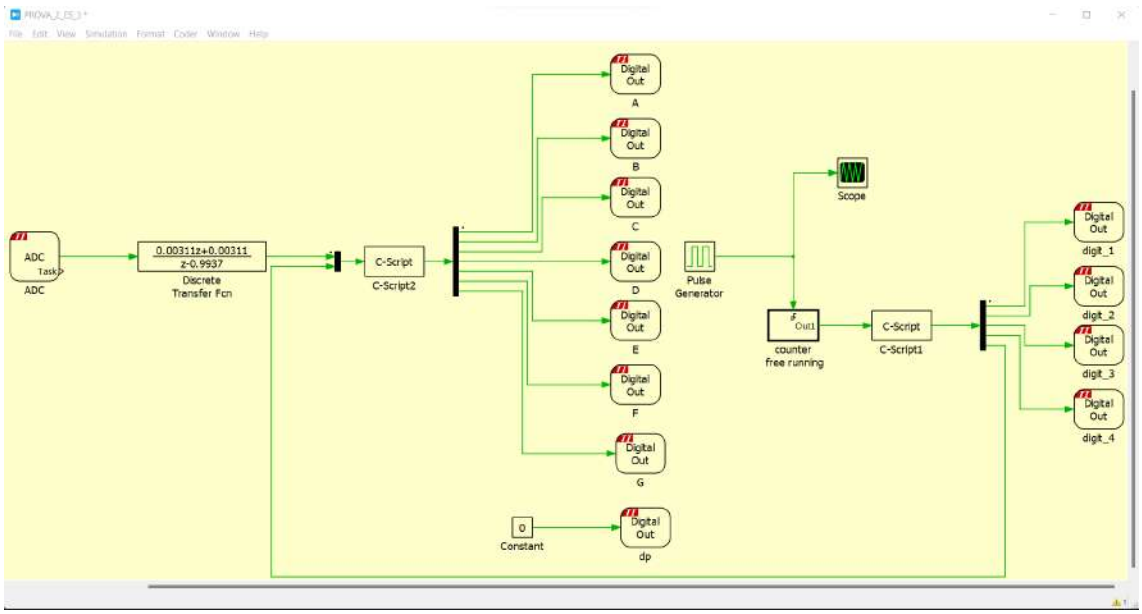


Figure 3.35: ninth example overall scheme

3.10 Example 10: Usage of a PWM for blinking a LED

In this example, It is introduced the PWM mask. The aim of this exercise is to control the brightness of a LED by using the PWM block.
The input signal of this block is the modulation index which can be set from 0 to 1. The PWM mask has a carrier that starts at 0 and varies between 0 and 1.
The PWM block can generate independent interrupts as output. these outputs can be both a SOC for the ADC and a Control Task Trigger. It is possible, thus, to set up two output signals which work as a trigger. These outputs are synchronized with the PWM carrier and will occur at the carrier underflow, overflow, or underflow and overflow events.
Fig. (3.36) shows the help window of the mask where the task trigger is set to underflow and the ADC trigger is set to overflow.

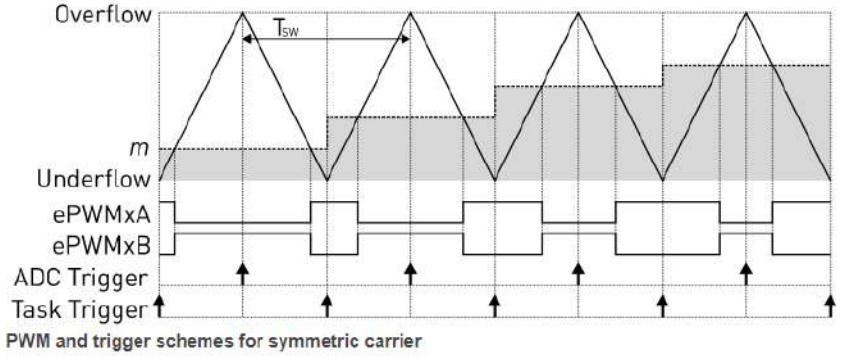


Figure 3.36: PWM block scheme

It is important to choose Overflow as the trigger Output if the PWM block is

used to trigger the start of conversion of the ADCs. In Fig. (3.36) is shown that when the carrier reaches the overflow boundary the ePWMxB switch is conducting. The shunt resistance used to detect the current in the leg of the inverter is located below the switches and so to ensure the correct measure the ePWMxB switch has to be closed. Furthermore, when the carrier reaches the overflow limit the ePWMxB switch is in the middle of its conducting period and so it is far away from transients. In this example, there is the possibility to choose three different registers that are connected to the three different LEDs. There is the register ePWM4A which is connected to the GPIO4 (red LED), the register ePWM5A connects to the GPIO8 (blue LED), and the register ePWM6A connects to the GPIO10 (green LED).
In this example, there is a constant block that feeds the PWM mask and if we change the value of the constant block between 0 and 1 the brightness of the LED will change.

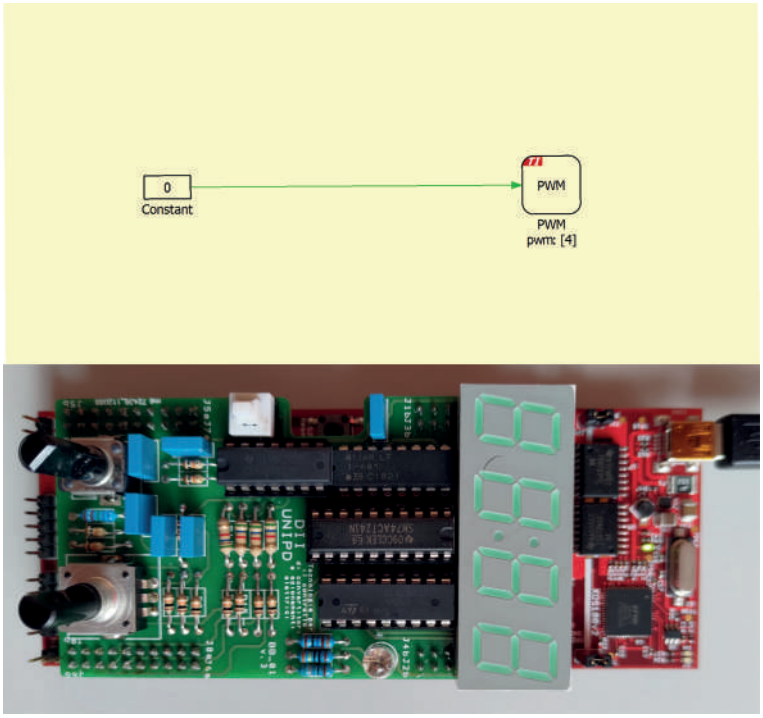


Figure 3.37: modulation index m = 0

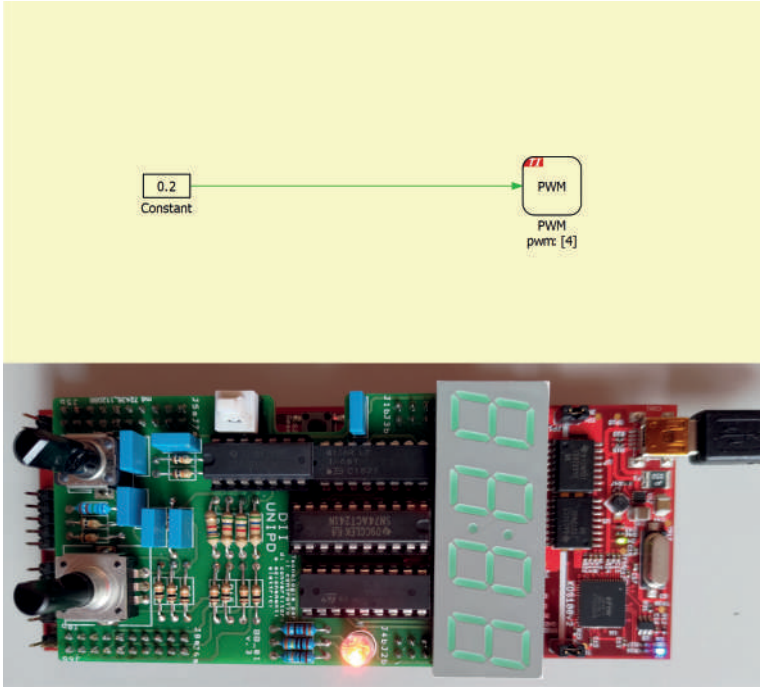


Figure 3.38: modulation index m = 0.2

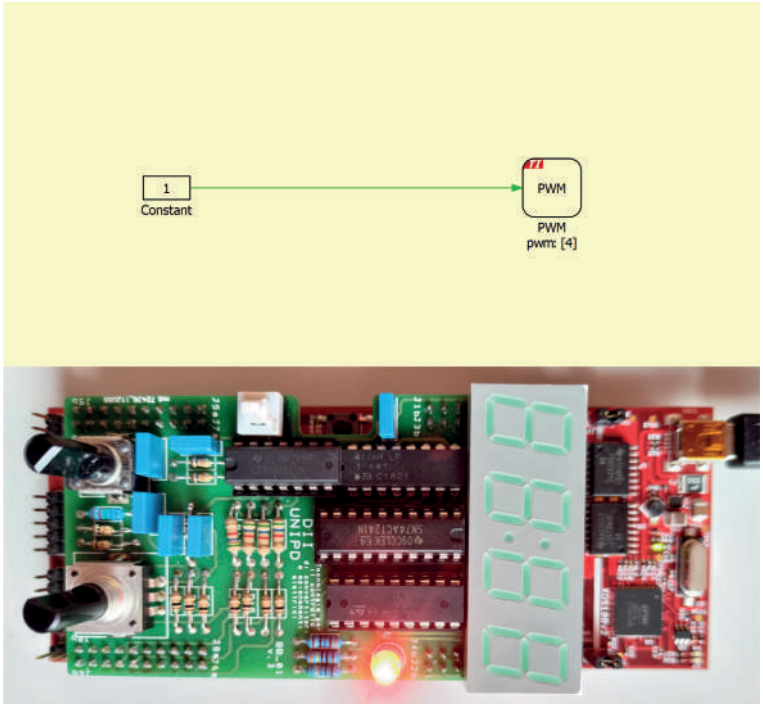


Figure 3.39: modulation index m = 1

3.11 Example 11: Control the modulation index with a potentiometer

In the next example, the constant block is replaced with the ADC mask. In this case, it is possible to regulate the modulation index by turning the

potentiometer. Notice that since the modulation index can change between 0 and 1, the ADC output cannot be connected to the PWM input directly. The output signal of the ADC is the measured voltage at the ADC pins. The maximum voltage reference is 3.3 V so the output signals of the ADC must be divided by the reference voltage in order to normalize the signal. In this case, the constant value is equal to three due to the non-exact value of the ADC output.

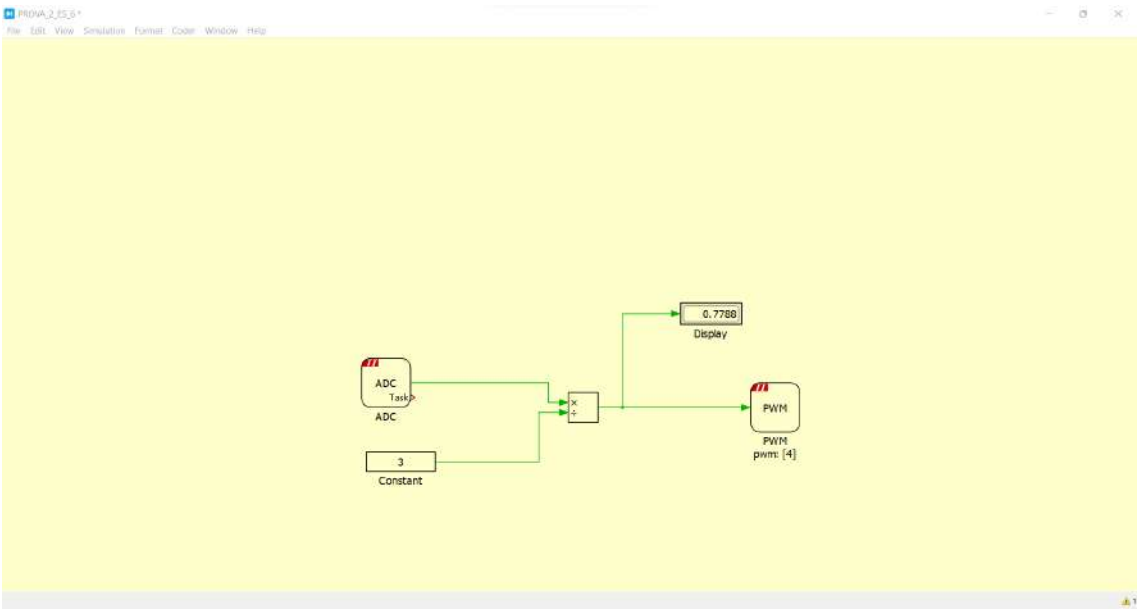


Figure 3.40: eleventh example overall scheme

3.12 Example 12: Three-phase modulation index with blocks

This example demonstrates how to produce three separate modulation indices from an encoder in order to simulate a three-phase system and control three different PWM blocks.
The maximum counter value of the encoder has to be set at 359 so it simulates the phase of the three sin-wave signals. Then since the modulation index must be a number between 0 and 1, a gain block has to be inserted with a constant value equal to $\frac{2\pi}{100}$. So, the three modulation indexes are:

$$m_1 = \frac{sin(x) + 1}{2}$$

$$m_2 = \frac{sin(x + \frac{2\pi}{3}) + 1}{2}$$

$$m_3 = \frac{sin(x - \frac{2\pi}{3}) + 1}{2}$$

Where x is the output signal of the gain block.

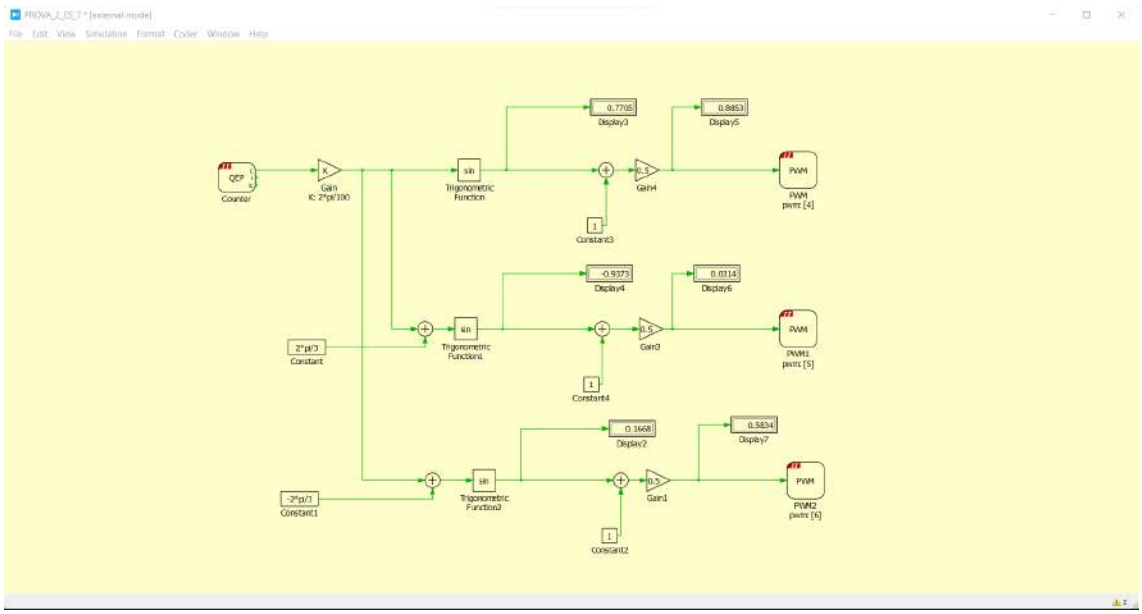


Figure 3.41: twelfth example overall scheme

3.13 Example 13: Three-phase modulation index with C-script block

In this exercise, a C-Script block replaces the scheme of example 12 in order to simplify the overall scheme.

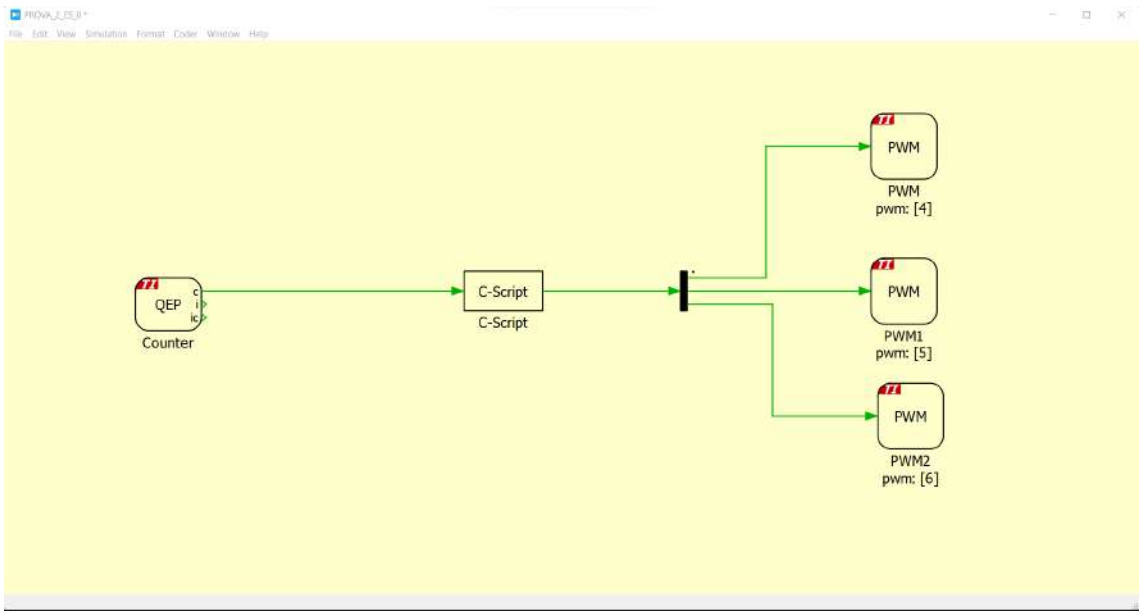


Figure 3.42: thirteenth example overall scheme

Listing 3.13: Code declaration

```
#include <math.h>
int e = 0;
float pi =3.141592653589793;
```

Listing 3.14: Output function code

```
int n = Input(0);
e = n*2*pi/100;
Output(0) = (sin(e)+1)*0.5;
Output(1) = (sin(e+2*pi/3)+1)*0.5;
Output(2) = (sin(e-2*pi/3)+1)*0.5;
```

3.14 Example 14: Control of the blinking of the edge segments of the display

In this example, an encoder is used to scroll the display border segments. The maximum value counter is 359. A C-Script block is used to light the right segment based on the encoder position. For example, if the encoder counter is between zero and thirty the display will light up the upper segments of the first two digits. If the encoder output signal is between thirty and sixty the display will light up the upper segments of the digits two and three and so on.

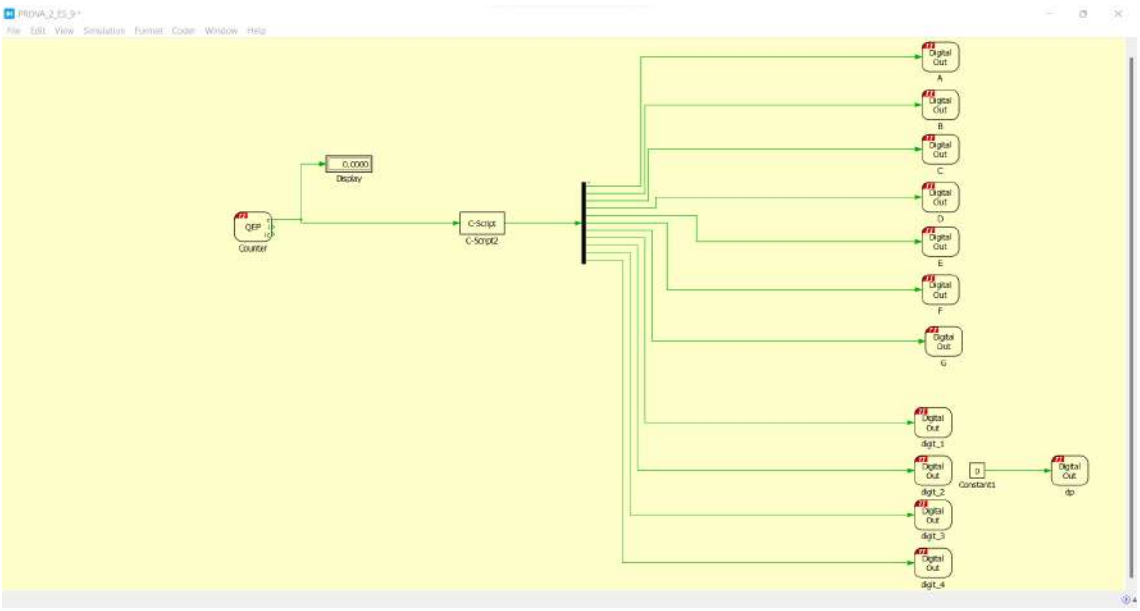
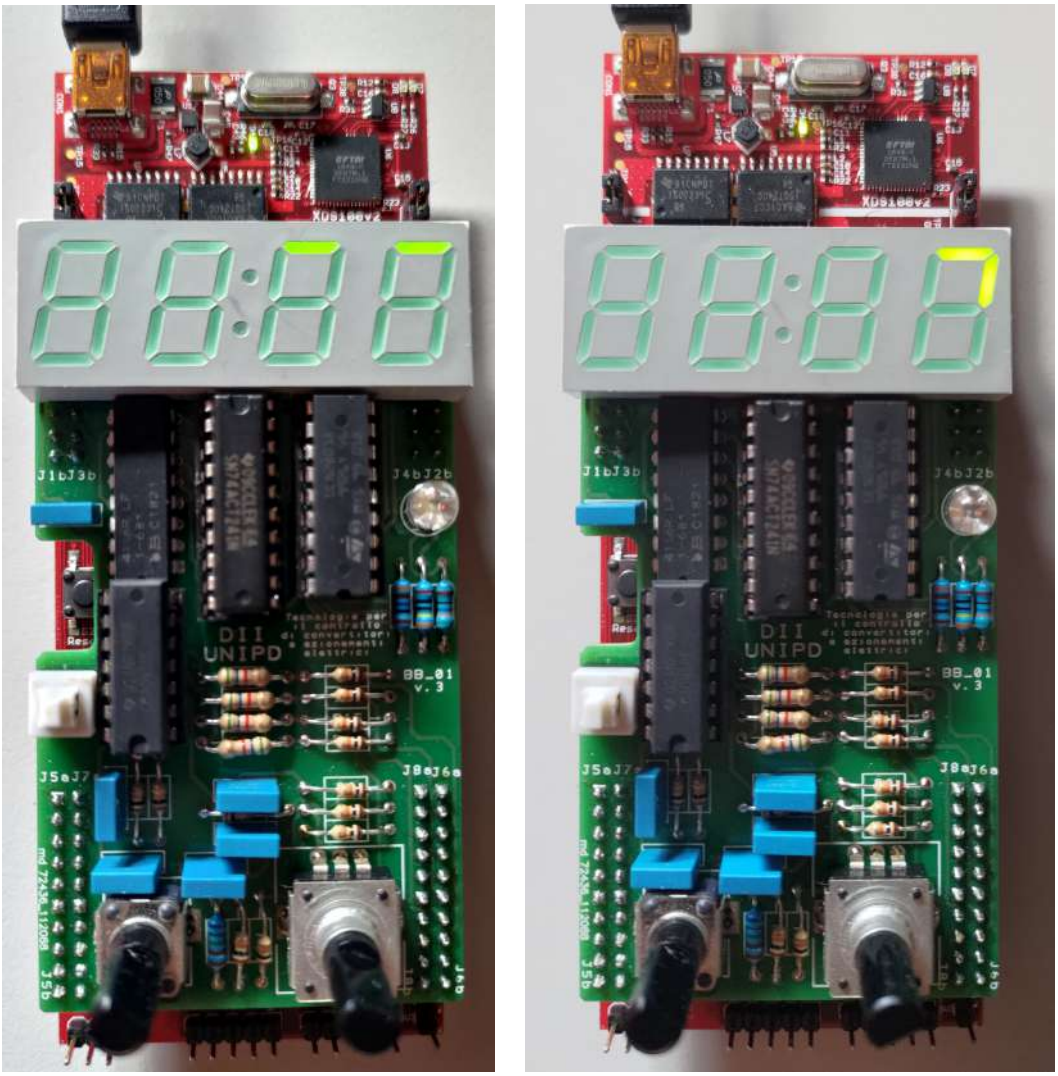


Figure 3.43: Fourteenth example overall scheme



((a)) *Angle between 30° and 60°* ((b)) *Angle between 60° and 90°*

Figure 3.44: LaunchPad

Listing 3.15: Output function code

```
int n = Input(0);
if (n>=0 && n<30)
{
Output(0)=1;
Output(1)=0;
Output(2)=0;
Output(3)=0;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=0;
Output(8)=0;
Output(9)=1;
Output(10)=1;
}
if (n>=30 && n<60)
{
Output(0)=1;
Output(1)=0;
Output(2)=0;
Output(3)=0;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=0;
Output(8)=1;
Output(9)=1;
Output(10)=0;
}
if (n>=60 && n<90)
{
Output(0)=1;
Output(1)=0;
Output(2)=0;
Output(3)=0;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=1;
Output(8)=1;
Output(9)=0;
Output(10)=0;
}
```

Listing 3.16: Output function code

```
if (n>=90 && n<120)
{
Output(0)=1;
Output(1)=1;
Output(2)=0;
Output(3)=0;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=1;
Output(8)=0;
Output(9)=0;
Output(10)=0;
}
if (n>=120 && n<150)
{
Output(0)=0;
Output(1)=1;
Output(2)=1;
Output(3)=0;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=1;
Output(8)=0;
Output(9)=0;
Output(10)=0;
}
if (n>=150 && n<180)
{
Output(0)=0;
Output(1)=0;
Output(2)=1;
Output(3)=1;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=1;
Output(8)=0;
Output(9)=0;
Output(10)=0;

}
```

Listing 3.17: Output function code

```
if (n>=180 && n<210)
{
Output(0)=0;
Output(1)=0;
Output(2)=0;
Output(3)=1;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=1;
Output(8)=1;
Output(9)=0;
Output(10)=0;
}
if (n>=210 && n<240)
{
Output(0)=0;
Output(1)=0;
Output(2)=0;
Output(3)=1;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=0;
Output(8)=1;
Output(9)=1;
Output(10)=0;
}
if (n>=240 && n<270)
{
Output(0)=0;
Output(1)=0;
Output(2)=0;
Output(3)=1;
Output(4)=0;
Output(5)=0;
Output(6)=0;
Output(7)=0;
Output(8)=0;
Output(9)=1;
Output(10)=1;
}
```

Listing 3.18: Output function code

```
if (n>=270 && n<300)
{
Output(0)=0;
Output(1)=0;
Output(2)=0;
Output(3)=1;
Output(4)=1;
Output(5)=0;
Output(6)=0;
Output(7)=0;
Output(8)=0;
Output(9)=0;
Output(10)=1;
}
if (n>=300 && n<330)
{
Output(0)=0;
Output(1)=0;
Output(2)=0;
Output(3)=0;
Output(4)=1;
Output(5)=1;
Output(6)=0;
Output(7)=0;
Output(8)=0;
Output(9)=0;
Output(10)=1;
}
if (n>=330 && n<360)
{
Output(0)=1;
Output(1)=0;
Output(2)=0;
Output(3)=0;
Output(4)=0;
Output(5)=1;
Output(6)=0;
Output(7)=0;
Output(8)=0;
Output(9)=0;
Output(10)=1;
}
```

3.15 Example 15: Check of the PWM block with an oscilloscope

It's now interesting to verify that the parameters set in PLECS are right. To do that the oscilloscope is connected to the LaunchPad. In this example, PWM block parameters have been verified. First the duty cycle. Pin number 80 (where stands the EPWM4A, GPIO6) and the ground are connected to the oscilloscope. A carrier frequency of 1000 kHz has been imposed, the carrier type is symmetrical, the PWM generator is 4 (which is the PWM of pin number 80), and the modulation index is 0.8.
From computation, the PWM output is zero and it is $T_{off} = 2e - 5 \quad [s]$.

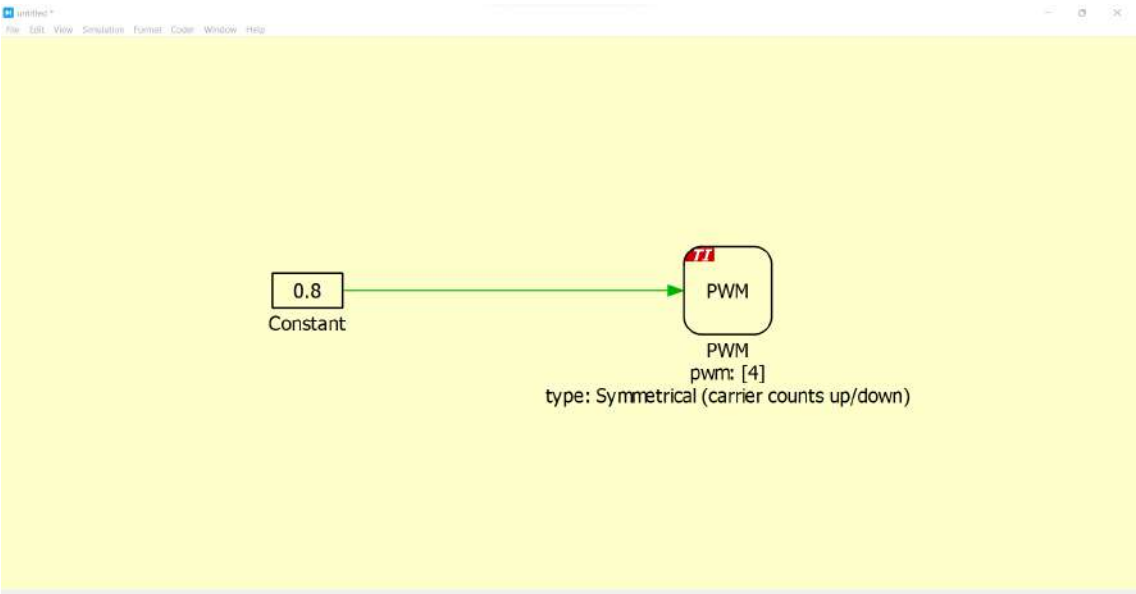


Figure 3.45: Overall scheme

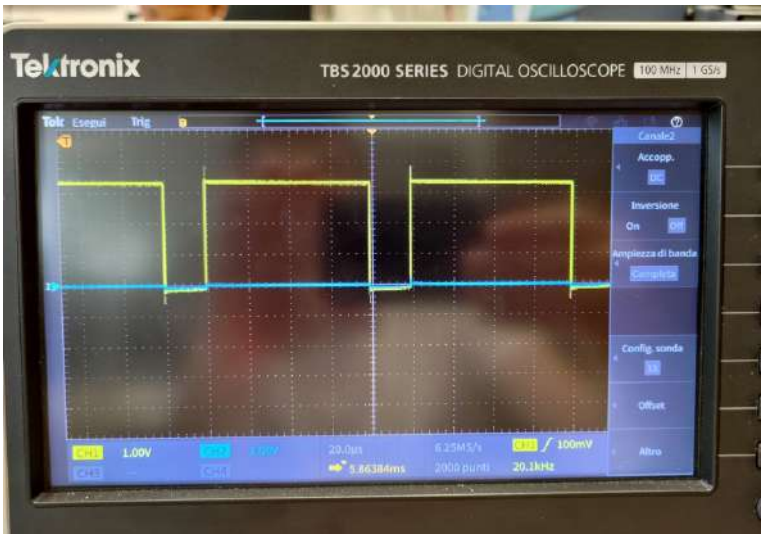


Figure 3.46: Oscilloscope response

3.16 example 16: Blanking time

In this example, it is shown how PLECS manages the blanking time. The first thing to understand is that PLECS does not allow the user to set independently the blanking time of leg A and leg B. So, there are only two ways of setting. Active high complementary (AHC) and active low complementary (ALC).
The output mode is set to complementary. Therefore, it is possible to see in the oscilloscope both leg A (pin 80) and leg B (pin 79) of the EPWM4 register. The blanking time is set at 1e-6 seconds.

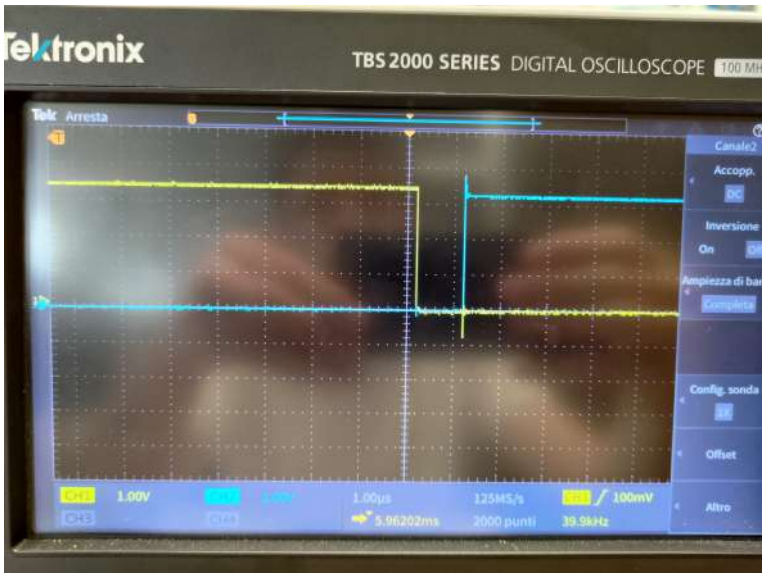


Figure 3.47: Blanking time

Notice that the yellow plot stands for leg A and the blue plot stands for leg B of the register. Now the polarity is set at Active state in logic '1' so it is possible to see AHC blanking mode.

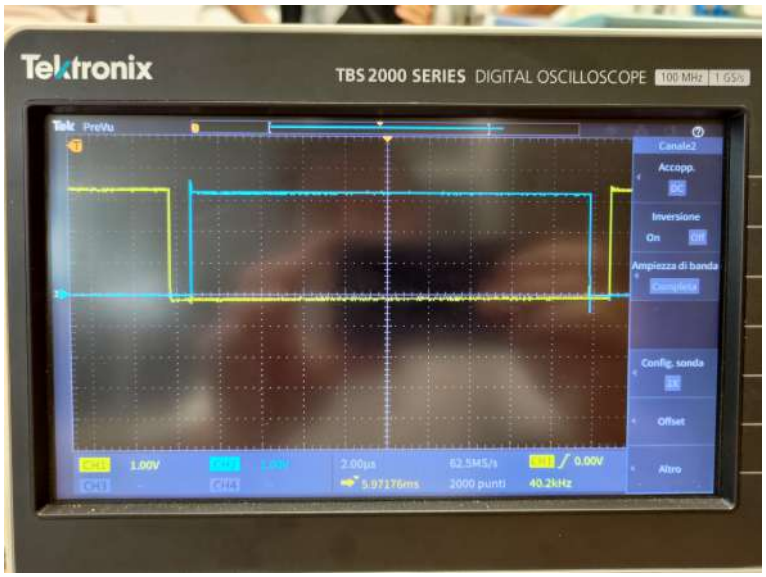


Figure 3.48: AHC behaviour

Otherwise, if the output mode is set at Active state as logic '0' it is possible to see in the oscilloscope the ALC blanking mode.

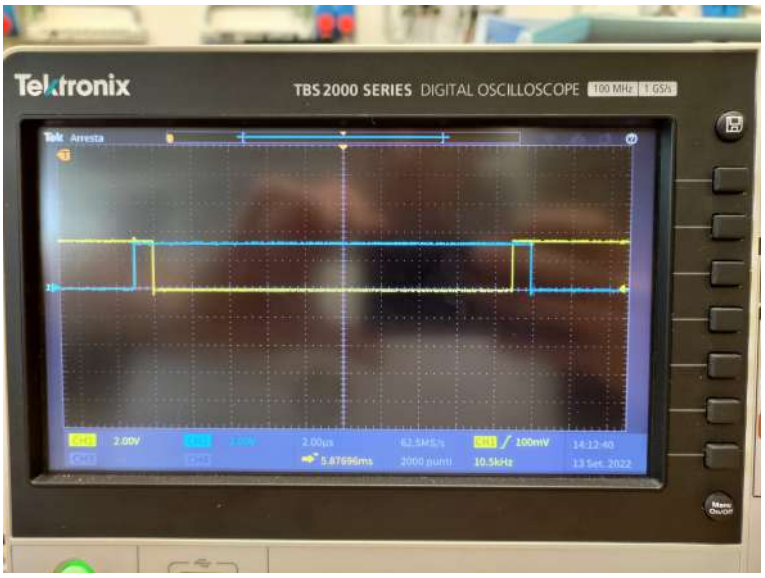


Figure 3.49: ALC behaviour

Chapter 4

Control of an SPM motor

4.1 Synchronous motors

Synchronous motors are electric machines composed of a stator and a rotor. A huge family of these machines is composed of permanent-magnetic (PM) machines. These motors are made up of a three-phase stator connected to an electrical source and a rotor with permanent magnets attached.
The three-phase stator produces a rotating magnetic field that interacts whit the PMs making the rotor move.
The interaction between the rotating magnetic field of the stator and the constant magnetic field made up of the PMs makes the rotor turn at the same speed as the stator's rotating field. For this reason, these kinds of motors are called synchronous motors.
The most known families of Synchronous motors are two: the surface permanent magnetic motors and the interior permanent magnetic motors.
The difference between these motors is in how the magnets are attached to the rotor.SPM motors have magnets affixed to the exterior of the rotor and IPM motors have the permanent magnet embedded into the rotor itself.

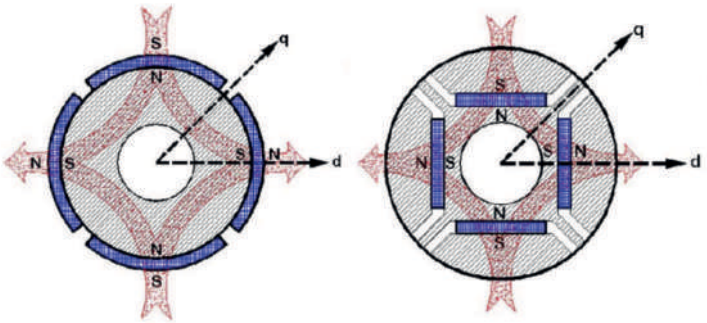


Figure 4.1: SPM and IPM motors

Since the permittivity of magnets is similar to the permittivity of the air SPM motors behave like an isotropic machine. This means that the reluctance of the d-axis and the q-axis are the same (Ld = Lq) so inductance values measured at the rotor terminals are consistent regardless of the rotor position.
This fact provides less torque than an IPM motor because the inductance

torque is neglected

$$T = \frac{3}{2} p \lambda_m i_q \tag{4.1}$$

IPM motors have different reluctances between the d-axis and the q-axis. Because of the position of the magnets attached to the rotor, the flux lines change between the two axes. This means that the machine behaves like an anisotropic machine.
The torque made by these motors has an additional term due to the saliency of the rotor (Ld > Lq). Thus, IPM motors could use fewer magnets for the same torque.

$$T = \frac{3}{2} p [\lambda_m i_q + (L_d - L_q) i_q i_d] \tag{4.2}$$

4.2 SPM motors

The aim of this study is to control an SPM motor. As said before the main characteristic of this motor is that it has Ld = Lq and so the flux produced by the magnet has no preferable paths.
The general equations of an electrical machine are:

$$\begin{cases} v_a = R i_a + \frac{d\lambda_a}{dt} \\ v_b = R i_b + \frac{d\lambda_b}{dt} \\ v_c = R i_c + \frac{d\lambda_c}{dt} \end{cases} \tag{4.3}$$

These equations are made with general terms. The flux term is composed of the flux made by the PM λ_m and the flux made by the stator current λ_a.

$$\begin{cases} \lambda_a = \lambda_{am} + \lambda_{ai} \\ \lambda_b = \lambda_{bm} + \lambda_{bi} \\ \lambda_c = \lambda_{cm} + \lambda_{ci} \end{cases} \tag{4.4}$$

The flux produced by the rotor magnets varies with the angle θ_m^e. Using the a-axis as a reference the electric angle will increase by turning counterclockwise.

$$\begin{cases} \lambda_{am} = \Lambda_m cos(\theta_m^e) \\ \lambda_{bm} = \Lambda_m cos(\theta_m^e - \frac{2\pi}{3}) \\ \lambda_{cm} = \Lambda_m cos(\theta_m^e - \frac{4\pi}{3}) \end{cases} \tag{4.5}$$

Since the motor studied is an SPM is possible to consider all the electrical components symmetric.

$$L = L_a = L_b = L_c M = M_{ab} = M_{bc} = M_{ac} \tag{4.6}$$

Furthermore not having the neutral wire inside the motor the sum of the three currents is equal to 0.

$$i_a + i_b + i_c = 0 \tag{4.7}$$

So the fluxes made by the stator currents have the following expression.

$$\begin{cases} \lambda_{ai} = L_a i_a + M_{ab} i_b + M_{ac} i_c = L i_a + M i_b + M i_c = (L - M) i_a \\ \lambda_{bi} = M_{ab} i_a + L_b i_b + M_{bc} i_c = M i_a + L i_b + M i_c = (L - M) i_b \\ \lambda_{ci} = M_{ac} i_a + M_{bc} i_b + L_c i_c = M i_a + M i_b + L i_c = (L - M) i_c \end{cases} \tag{4.8}$$

the term L - M is also called synchronous inductance L_s (Often it is used L instead of L_s).
By replacing Eq. (4.8) in Eq. (4.3) the general equations become.

$$\begin{cases} v_a = Ri_a + \frac{d\lambda_{ma}}{dt} + L_s\frac{di_a}{dt} \\ v_b = Ri_b + \frac{d\lambda_{mb}}{dt} + L_s\frac{di_b}{dt} \\ v_c = Ri_c + \frac{d\lambda_{mc}}{dt} + L_s\frac{di_c}{dt} \end{cases} \tag{4.9}$$

notice that.

$$\begin{cases} \frac{d\lambda_{ma}}{dt} = -\omega_m^e\Lambda_m sin(\theta_m^e) = \omega_m^e\Lambda_m cos(\theta_m^e + \frac{\pi}{2}) \\ \frac{d\lambda_{mb}}{dt} = -\omega_m^e\Lambda_m sin(\theta_m^e - \frac{2\pi}{3}) = \omega_m^e\Lambda_m cos(\theta_m^e + \frac{\pi}{2} - \frac{2\pi}{3}) \\ \frac{d\lambda_{mc}}{dt} = -\omega_m^e\Lambda_m sin(\theta_m^e - \frac{4\pi}{3}) = \omega_m^e\Lambda_m cos(\theta_m^e + \frac{\pi}{2} - \frac{4\pi}{3}) \end{cases} \tag{4.10}$$

Where ω_m^e is the electric velocity of the rotor synchronous with the rotating magnetic field.
These values produce a three-phase back e.m.f. with peak value equal to:

$$E = \omega_m^e\Lambda_m \tag{4.11}$$

So eventually the system ends up with the following scheme.

$$\begin{cases} v_a = Ri_a + e_a + L\frac{di_a}{dt} \\ v_b = Ri_b + e_b + L\frac{di_b}{dt} \\ v_c = Ri_c + e_c + L\frac{di_c}{dt} \end{cases} \tag{4.12}$$

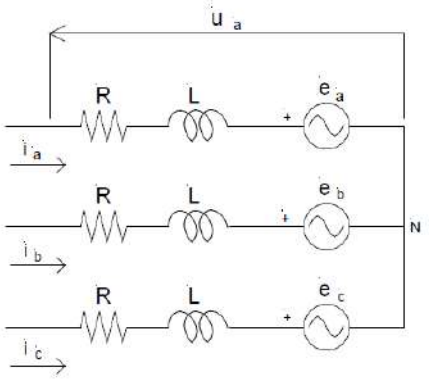


Figure 4.2: Equivalent scheme of an SPM motor

The equations used can be rewritten in terms of spatial phasors by introducing the Clarke and Park transformations(see Appendix 2).

$$\vec{v} = R\vec{i} + L\frac{d\vec{i}}{dt} + \vec{e} \tag{4.13}$$

The term $\vec{e}$ is the derivate of the magnetic flux vector so the back e.m.f. is a spatial vector shifted counterclockwise 90 degrees in phase from the magnetic flux vector at the end of the computation.

$$\vec{e} = \frac{d\vec{\lambda_m}}{dt} = \frac{d}{dt}[\Lambda_m e^{j\theta_m^e}] = j\omega_m^e\vec{\lambda_m} \tag{4.14}$$

So introducing Eq. (4.14) in Eq. (4.13) the general equation becomes:

$$\vec{v} = R\vec{i} + L\frac{d\vec{i}}{dt} + j\omega_m^e \vec{\lambda_m} \tag{4.15}$$

The Clarke transformation split the vectorial equation in the two axes α and β.

$$v_\alpha + jv_\beta = R(i_\alpha + ji_\beta) + L(\frac{di_\alpha}{dt} + j\frac{di_\beta}{dt}) + j\omega_m^e(\lambda_\alpha + j\lambda_\beta) \tag{4.16}$$

$$\begin{cases} v_\alpha = Ri_\alpha + L\frac{di_\alpha}{dt} - \omega_m^e\lambda_\beta \\ v_\beta = Ri_\beta + L\frac{di_\beta}{dt} + \omega_m^e\lambda_\alpha \end{cases} \tag{4.17}$$

The Clarke transformation transforms a three-phase system into a bidimensional domain where the real axis is named alpha and the imaginary axis is named beta. The Alpha axis is solid with the a-axis which represents the reference for the electrical angle.
Park transformation introduces a reference system that rotates with a certain velocity, in this case, ω_m^e. Since this reference system is solid with the velocity of the rotating magnetic field the term of the Park equation will be constant. A general vector written in alpha-beta reference becomes in d-q reference:

$$\vec{g^R} = g^s e^{\vec{-j\theta_m^e}} \tag{4.18}$$

So the equation in a rotating reference system is:

$$\vec{v^R}e^{j\theta_m^e} = R\vec{i^R}e^{j\theta_m^e} + L\frac{d\vec{i^R}e^{j\theta_m^e}}{dt} + j\omega_m^e\vec{\lambda_m}^R e^{j\theta_m^e} \tag{4.19}$$

$$\vec{v^R} = R\vec{i^R} + L\frac{d\vec{i^R}}{dt}j\omega_m^e(L\vec{i^R} + \vec{\lambda_m}^R) \tag{4.20}$$

As said before the d-q reference is solid with the electric magnetic field so the magnetic flux vector is solid with the d-axis as well.

$$\vec{\lambda_m}^R = \lambda_m + j0 \tag{4.21}$$

The general d-q equations become:

$$\begin{cases} v_d = Ri_d + L\frac{di_d}{dt} - \omega_m^e L i_q \\ v_q = Ri_q + L\frac{di_q}{dt} + \omega_m^e(Li_d + \lambda_m) \end{cases} \tag{4.22}$$

Since the equations are made with the d-q transformation invariant with the magnitude (see appendix 2) the power balance becomes:

$$\frac{3}{2}(v_d i_d + v_q i_q) = \frac{3}{2}R(i_d^2 + i_q^2) + \frac{3}{2}L(\frac{di_d}{dt}i_d + \frac{di_q}{dt}i_q) + \frac{3}{2}\omega_m^e i_q \lambda_m \tag{4.23}$$

The first two terms represent the losses of the motor but the last term represents the Torque.
The torque made by an SPM motor can be represented by the sum of a viscous term, an inertia term, and a disturbance term that appears during the on-load

behavior.

$$T = \frac{3}{2} p \lambda_m i_q = T_L + B\omega_m + J\frac{d\omega_m}{dt} \tag{4.24}$$

Summarising the fundamental equations Eq. (4.24) and Eq. (4.22). It is possible to represent the SPM motor with a block scheme.

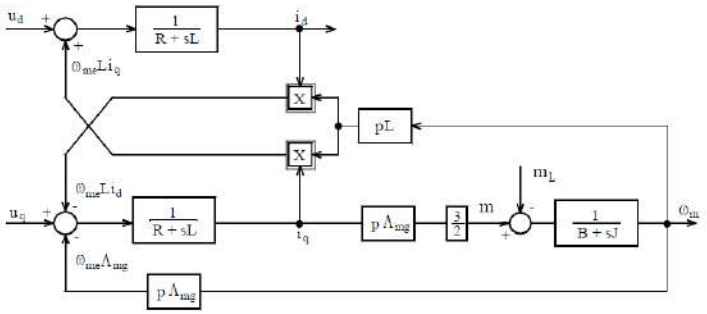


Figure 4.3: blocks scheme of an SPM motor

4.3 Main data

This study's motor is the QBL 4208-100-04-025-DS which is an SPM motor from the QMot QBL4208 family provided by TRINAMIC. Below are shown the plate data.

Variable	value	unit
N° poles	8	
N° phases	3	
Rated Voltage	24	V
Rated phase current	6.95	A
Rated speed	4000	rpm
Rated Torque	0.25	Nm
Max peak torque	0.75	Nm
length	100	mm
Line to line resistance	0.28	Ω
Line to line inductance	0.54	mH
Coil windings	delta connection	
Insulation Class	B	
rotor inertia	96 x 1e-6	kgm^2

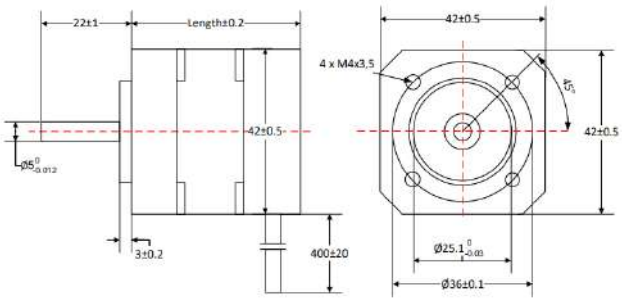


Figure 4.4: dimensions of the motor

4.4 Motor characterization

This section of the study aims to compare the main motor parameters shown in the user manual provided by TRINAMIC with the parameters found in the laboratory.
Parameter values found in this section of the study will be used later to design PI gains.

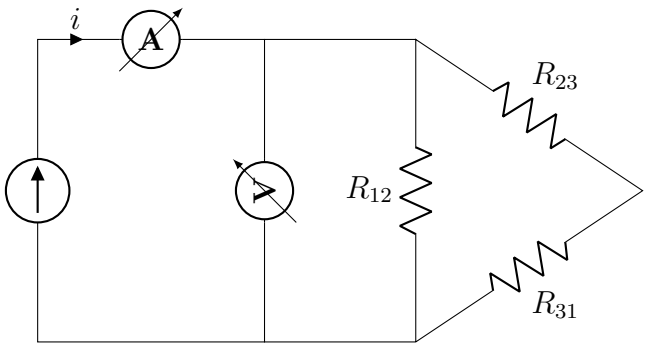
4.4.1 Line-to-line resistance

To measure the line-to-line resistance is used the Volt-amperometric method. Using a current generator it is imposed a fixed DC current on the motor terminals. Then a voltmeter is connected in parallel.
In this case, it is used $I = 5.21 \quad [A]$ as current in order to be very close to the Full scale of the instrument.
To be sure that the current generator forces the same quantity of current that is shown in its display, an ammeter is connected in series.
The datasheet of the motor gives the resistance that the firm measured. It is said that the line-to-line resistance is $R = 0.28 \quad [\Omega]$. This value is far lower than the voltmeter resistance and comparable with the shunt resistance of the ammeter. In order to achieve the lowest error during the measure the voltmeter is put after the ammeter. Since the type of connection between the phases is of delta topology the resistance measured is not the phase resistance but it is in series with the parallel of the two other phase resistances.

$$I = 5.21 \quad [A]$$
$$V = 1.546 \quad [V]$$

$$R_{mis} = \frac{V}{I} = 0.297 \quad [\Omega]$$

$$R_{12} = \frac{3}{2}R = 0.445 \quad [\Omega]$$



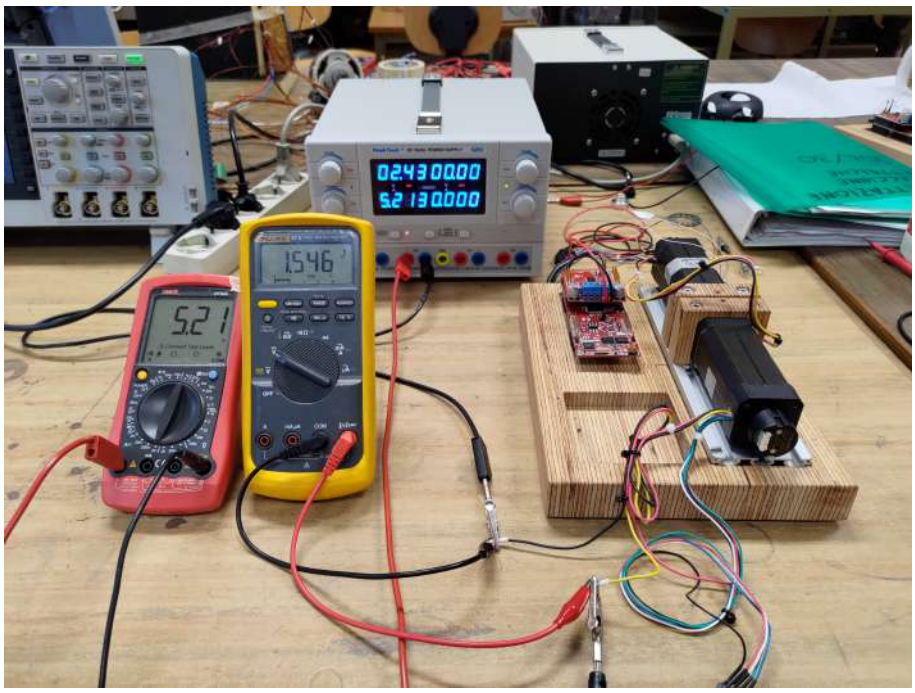


Figure 4.5: volt-amperometric circuit

4.4.2 Flux of the magnet

To compute the magnet flux two different motors are needed. The two motors are connected to each other with a joint so that if one rotates the other one will rotate as well.
Since the second motor is not connected to anything there will not be current in the phases of the motor. So it is possible to detect the back e.m.f. and compute the flux of the magnets.
To ensure a good approximation if the value of the measure is made with different speeds. With a speed loop control, the motor is run at a fixed velocity, and a digital multimeter measures the back e.m.f. from two phases of the second motor. Since the motor has a delta connection the back e.m.f. is the tension between two phases. The multimeter read the RMS value so, to achieve the peak value the measurement has to be multiplied by $\sqrt{2}$.

$$\hat{\lambda}_m = \frac{\hat{V}_{ph}}{p\omega_m} \tag{4.25}$$

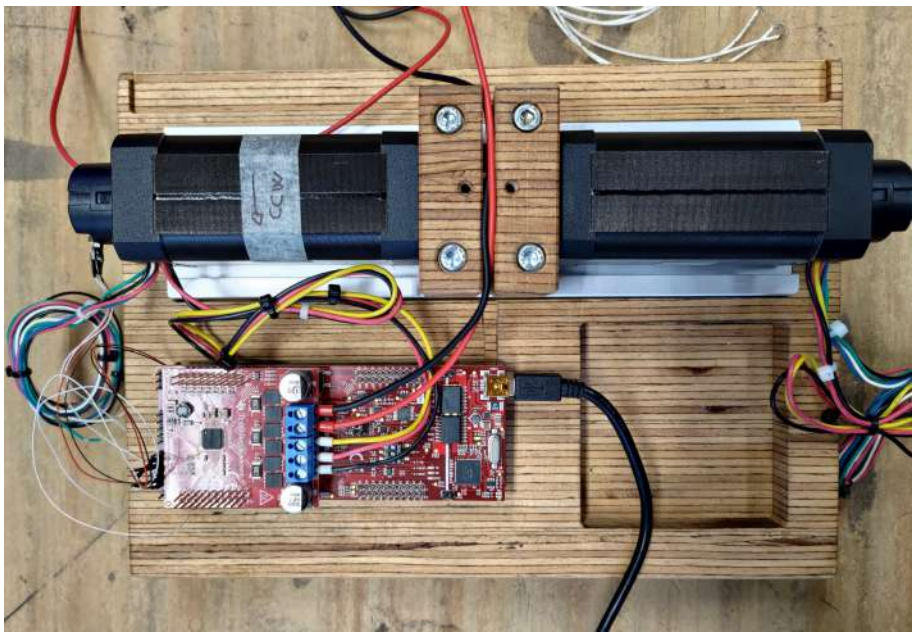


Figure 4.6: workstation

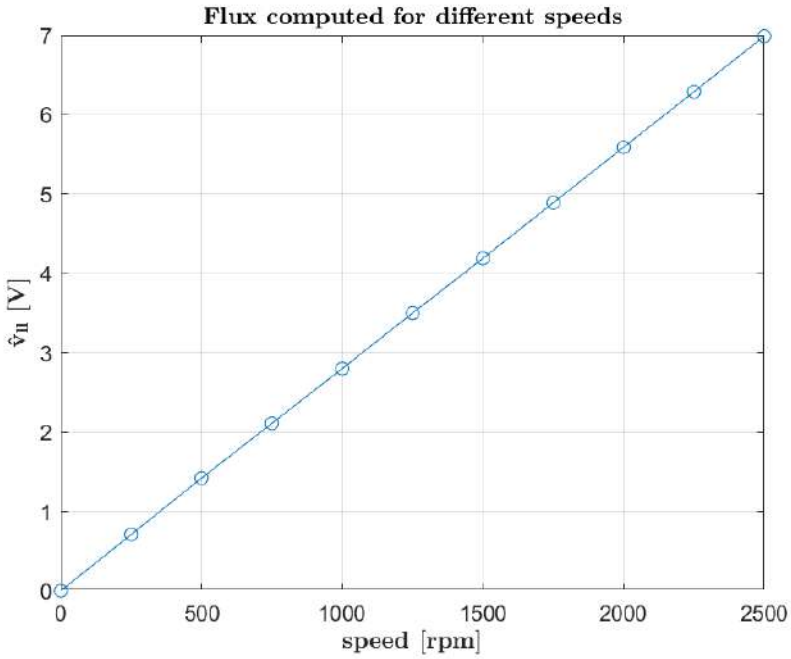


Figure 4.7: back e.m.f. at different speeds

It is possible to see that the flux of the magnets increases linearly with speed.

$$\hat{\lambda}_m = 9.48e-03 \quad [Vs]$$

4.4.3 Line-to-line inductance

An AC voltage generator is connected to two phases of the motor. Once the current is forced into the motor the inductance is calculated with an option of the voltage generator that is able to compute the line-to-line inductance

automatically.
The inductance is calculated sequentially for every fixed step of the angle until the rotor has done a complete turn. The instrument used to compute the line-to-line inductance is the PeakTech 2165 USB and as said before it is able to compute the line-to-line inductance automatically.

Figure 4.8: PeakTech 2165 USB

The instrument has to be set with a frequency of $f = 1 \quad [kHz]$ in order to measure better the inductive term.
Since the inductance is very small the measure is not considered very accurate but it is suitable for the design of the PI.

$$L_{mis} = 0.487 \quad [mH]$$

Since the motor has delta topology the line-to-line inductance is:

$$L_{12} = \frac{3}{2} L_{mis} = 0.7305 \quad [mH]$$

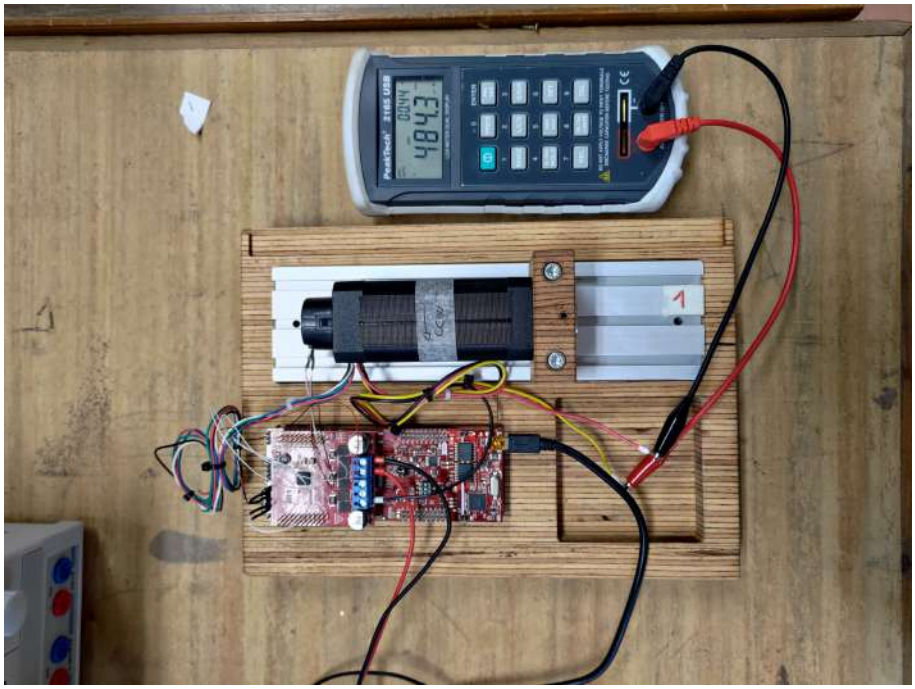


Figure 4.9: workstation, front view

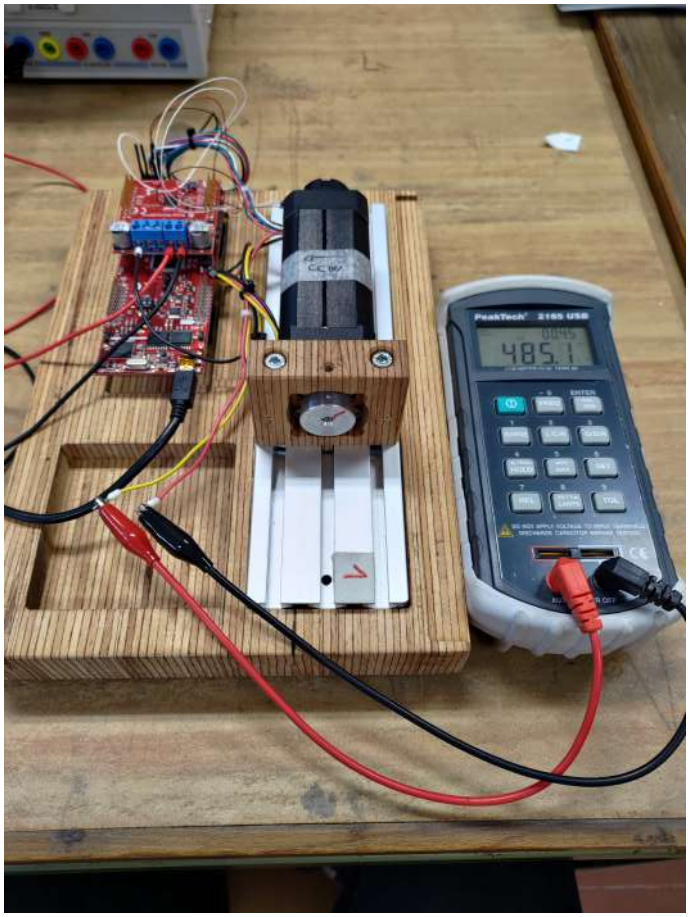


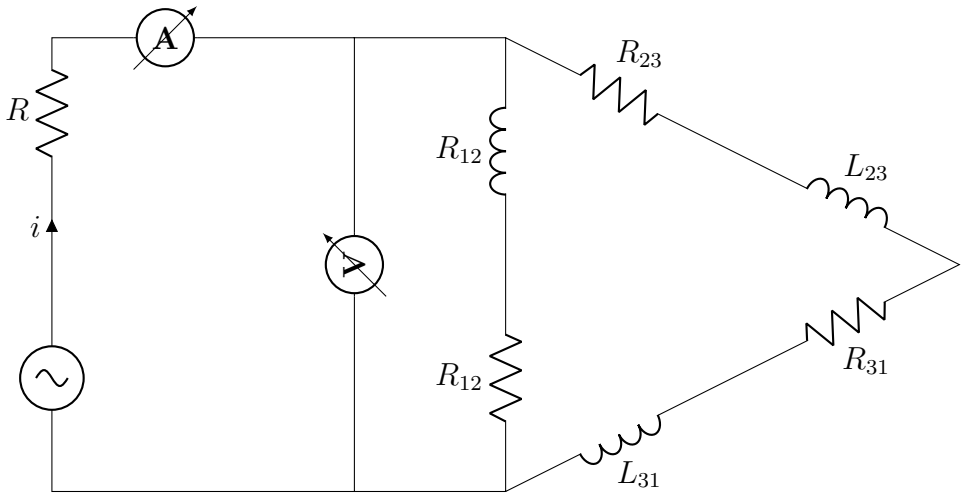
Figure 4.10: workstation, side view

To confirm the result obtained with the LCR meter it is possible to measure the inductance of the motor with a volt-amperometric method.

An AC generator imposed a sine current on the motor at different frequencies. An ammeter is put in series with the AC generator and a voltmeter is put in parallel with the phase of the motor.
Knowing the RMS value of the current and the tension it is possible to know the impedance of the motor. The AC generator has an internal impedance of $R = 50 \quad [\Omega]$.

$$Z = \frac{V}{I} \tag{4.26}$$

$$L = \frac{\sqrt{Z^2 - R_{mis}^2}}{2 * pi * f} \tag{4.27}$$



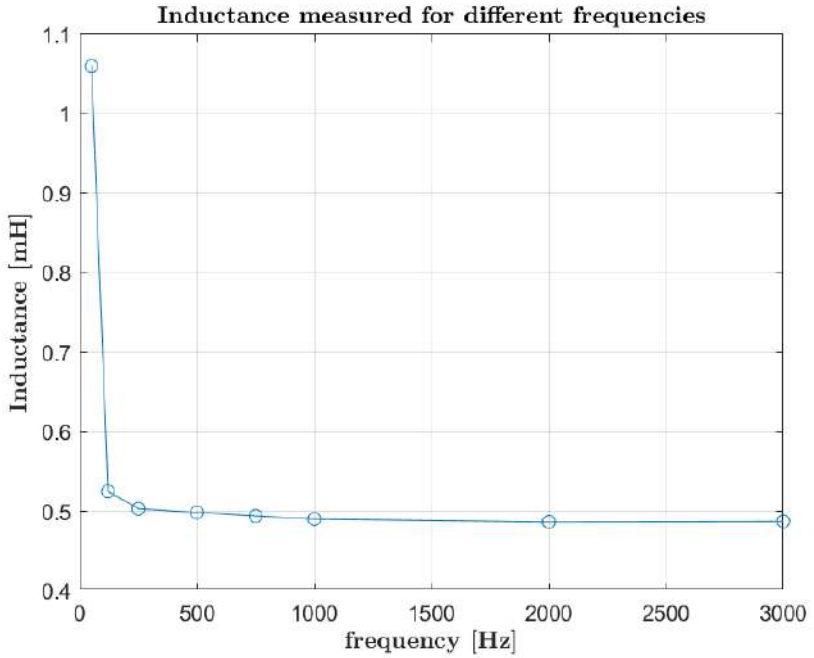


Figure 4.11: Inductance for different frequencies

It is noticeable that the measure has validity for high frequencies. At low frequencies, the ohmic term prevails over the inductive one and the measure

is not valid.
At high frequencies, instead, the inductance value remains constant with a value of:

$$L_{mis} = 0.490 \quad [mH]$$

This value is very close to the one computed with the LCR meter. That confirms the validity of the measure.
Since the motor has a delta topology the phase Inductance becomes:

$$L_{12} = \frac{3}{2} L_{mis} = 0.735 \quad [mH]$$

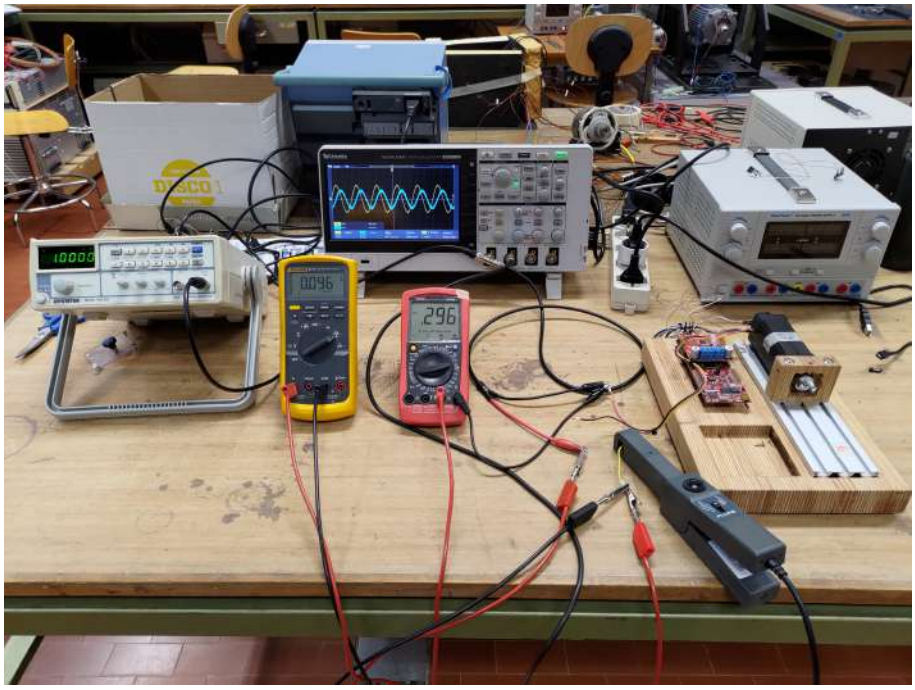


Figure 4.12: Workstation

4.4.4 Viscous term

The viscous term of a motor expressed in Nms is the toughest term to compute because it depends on several quantities. In order to achieve a good result the viscous term is calculated from the mechanical time constant of the motor.
The motor is made to rotate at a fixed speed. Then from the software, the user disabilities the PWM mask so that the motor is no longer controlled. It will now decelerate with its mechanical constant until it will stop.
With a scope is possible to see the speed waveform from the fixed speed to zero.

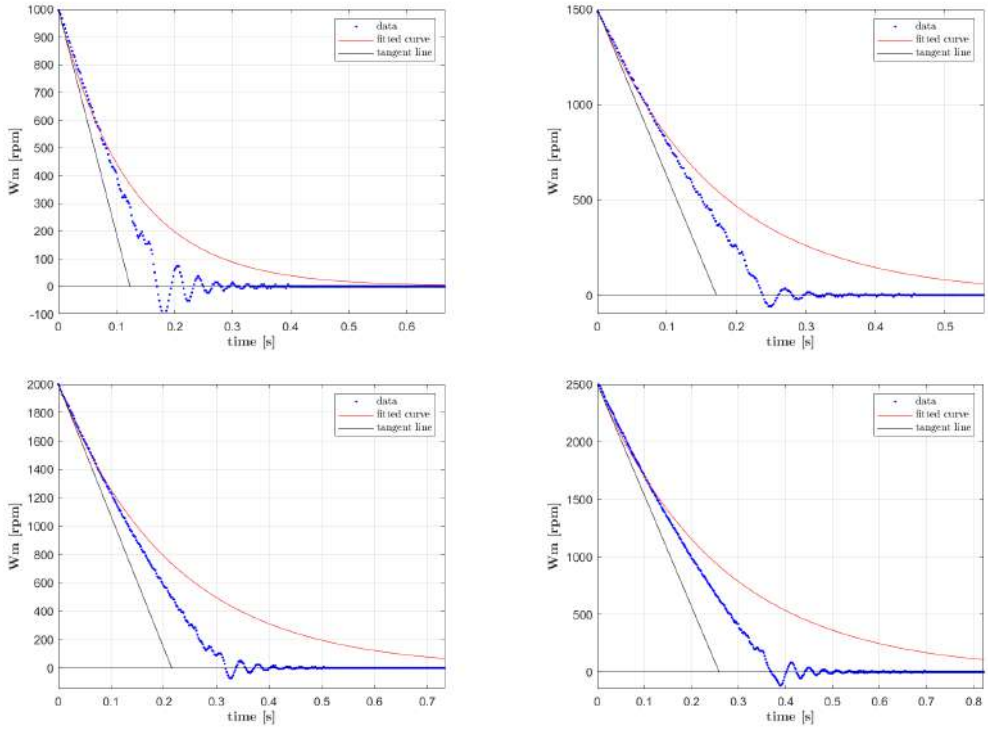


Figure 4.13: Speed decay from different initial value

Fig. (4.13) shows the decay of the speed from different initial points. It is noticeable that the viscous term is not constant but it changes due to the speed of the motor. Furthermore, at low speeds, it is shown an oscillating trend. This is because the rotor tends to get stuck where there are magnets.
The mechanic part of a motor is modeled as a first-order behavior. In this case, it is noticeable that the exponential interpolation of the data points doesn't fit the speed behavior very well.
This means that the assumption that the mechanic part of the motor has a first-order behavior isn't correct.
For this study, this assumption is considered. It means that the speed loop will behave slightly differently from the model to the actual motor.
In order to find the viscous term the Taylor series of the exponential is computed in the initial value:

$$y = Ae^{-xt} = A - Axt \tag{4.28}$$

Where A is the initial speed value and Ax is the slope of the line tangent at the beginning of the exponential function. The point where the line intersects the x-axis is the τ_m of the motor. Once known τ_m and J, B is easily computed.

$$B = \frac{J}{\tau_m} \tag{4.29}$$

n [rpm]	B [Nms]
1000	7.7662e-04
1500	5.6043e-04
2000	4.4598e-04
2500	3.7040e-04

Once B has been computed for all the velocities the term used for the design of the PI of the speed loop is the average.

$$B_{mean} = 5.38e-04 \quad [Nms]$$

4.5 Design of the PI

To control a motor the design of the PI controllers is critical. In Eq. (4.22) is shown that the currents i_d and i_q both affect the v_d and v_q equations. In Fig. (4.3) is possible to see that the two axes are coupled. For better control of the SPM is necessary to decouple these two terms.
To do this is necessary the knowledge of the main parameters of the motor that are computed in section 3.4.

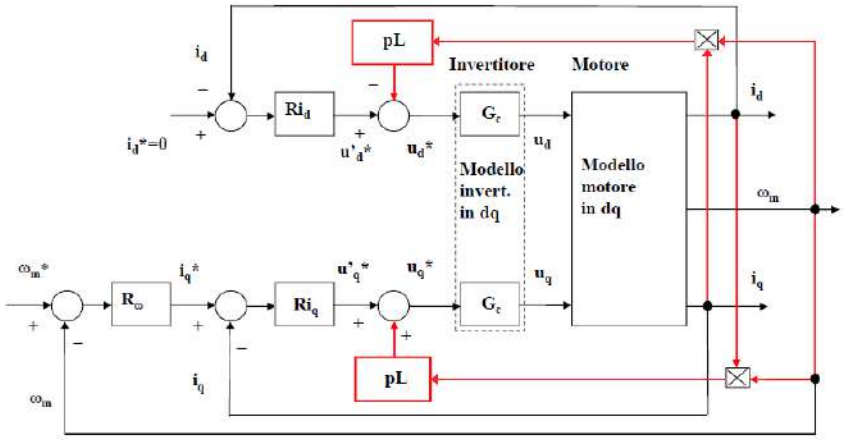


Figure 4.14: d-q axes decoupling

Eq. (4.22) equations represent the behavior of the SPM motor. It is noticeable that in the v_q equation there is the term $p\Lambda_m\omega_m$ that has a negative sign as shown in Fig. (4.3) as well. In order to have the same scheme for the d-axis and the q-axis and thus design the PI of both, in the same way, it is used the e.f.m. compensation method.
This method consists in add the term $p\Lambda_m\omega_m$ after the output of the PI controller in order to compensate the negative quantity of the v_q equation

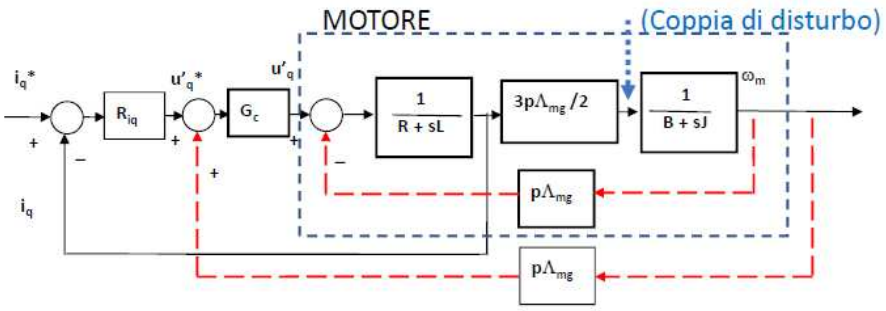


Figure 4.15: E.m.f. compensation

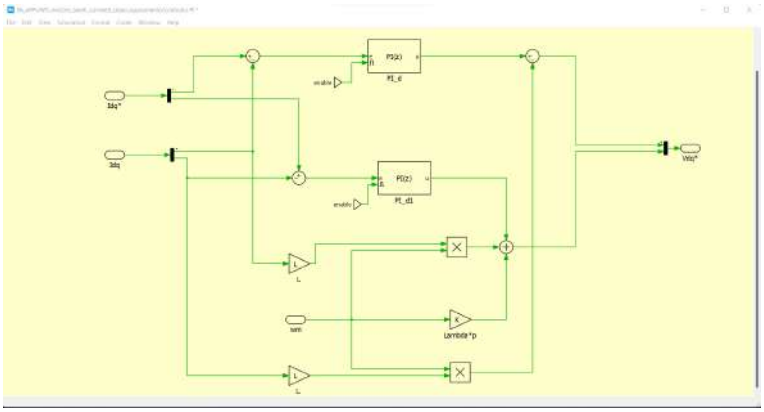


Figure 4.16: Axes decoupling from the PLECS scheme

4.5.1 Current-loop PI design

For the control of the motor, it is shown in section 4.2 that the star topology is assumed. Since the motor used for this study has delta topology the electrical parameters have to be changed.

$$La = \frac{L_{mis}}{2} = 0.245 \quad [mH]$$

$$Ra = \frac{R_{mis}}{2} = 0.1484 \quad [\Omega]$$

$$\lambda = \frac{\hat{\lambda}_m}{\sqrt{3}} = 0.00547 \quad [Vs]$$

PI controllers are made by two terms: the proportional term and the integral term.
In the s-domain the PI(S) is expressed as:

$$R(S) = k_p + \frac{k_i}{s} = k_p\left(\frac{s + \frac{k_i}{k_p}}{s}\right) = k_i\left(\frac{1 + s\tau_r}{s}\right) \tag{4.30}$$

$$\tau_r = \frac{k_p}{k_i} \tag{4.31}$$

The parameter k_p and k_i are called respectively the proportional gain and the integral gain. The aim of the design of a PI controller is to set these two terms in order to satisfy the requirement of the control (bandwidth and phase margin).
The current-loop scheme is shown in Fig. (4.18).

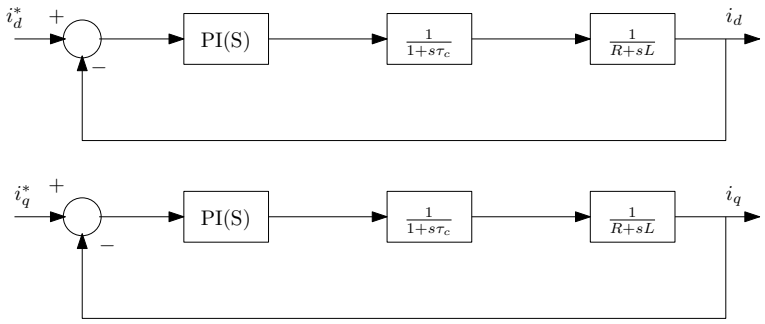


Figure 4.17: I_d and I_q schemes

The inverter carrier period is $Tc = 1e^{-4}[s]$. In the s-domain it is represented as a delay with the following expression:

$$G_c = \frac{1}{1 + s\tau_c} \tag{4.32}$$

$$\tau_c = \frac{3}{2}Tc = \frac{3}{2}1e^{-4} = 150[\mu s]$$

The electrical part of the SPM motor shows the electric behavior of the motor represents a first-order system:

$$\frac{1}{R + sL} = \frac{1}{R}\frac{1}{1 + \tau_e s} \tag{4.33}$$

$$\tau_e = \frac{L}{R} \tag{4.34}$$

The open loop transfer function of the system is:

$$GH(S) = k_i\frac{1 + s\tau_r}{s}\frac{1}{1 + s\tau_c}\frac{1}{R}\frac{1}{1 + s\tau_e} \tag{4.35}$$

In order to achieve a good design for the control gains a bandwidth and a phase margin has to be imposed.
For this study these are:

$$PM = 90$$

$$f_{bw} = 100 \quad [Hz]$$

$$\omega_{bw} = 2\pi 100 = 628.318$$

There are two equations with two unknown terms so a system is needed.
Firstly the magnitude of the open loop transfer function is forced to 1 at the bandwidth frequency.

$$|GH(j\omega_{bw})| = 1 = k_i\frac{\sqrt{1 + \omega_{bw}^2\tau_r^2}}{\omega_{bw}}\frac{1}{\sqrt{1 + \omega_{bw}^2\tau_c^2}}\frac{1}{R}\frac{1}{\sqrt{1 + \omega_{bw}^2\tau_e^2}} \tag{4.36}$$

Then the phase margin is imposed.

$$PM = \frac{\pi}{2} = \pi - arctan(\tau_r) + arctan(\tau_c) + arctan(\tau_e) + \frac{\pi}{2} \tag{4.37}$$

From these equations the gain values become:

$$k_i = 84.1039$$

$$k_p = 0.1679$$

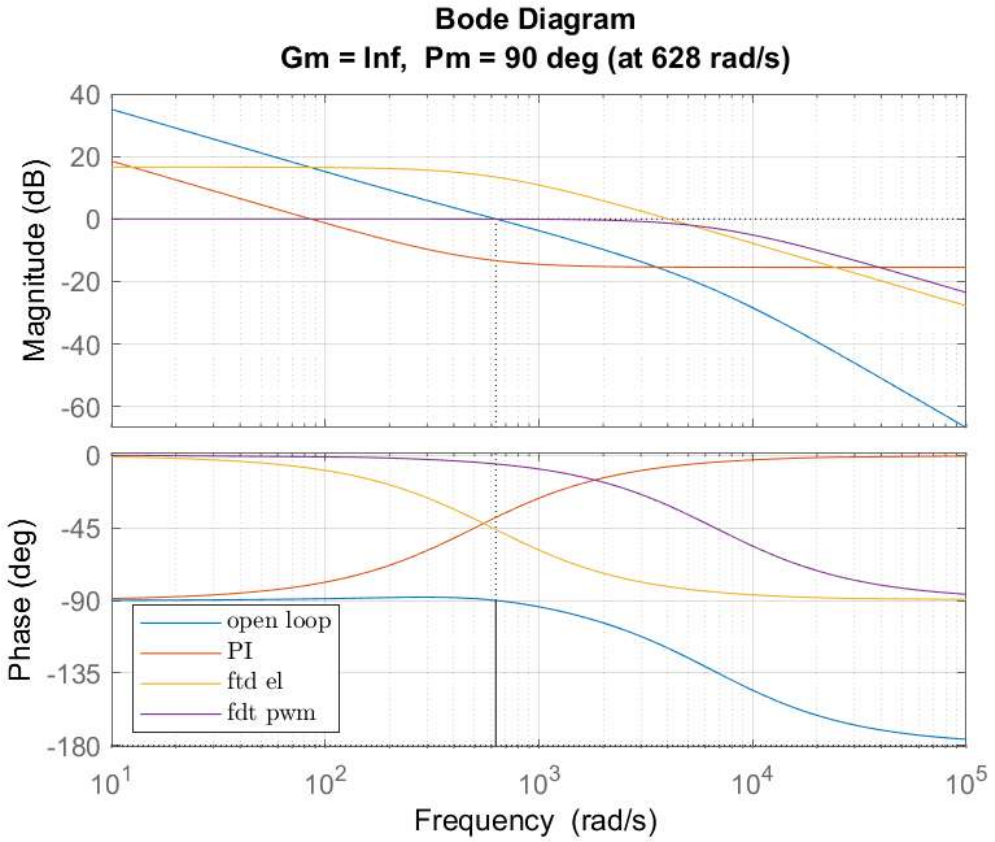


Figure 4.18: bode diagram

Another way of PI design is the zero-pole cancellation. This method consists to delete the main pole in the open loop system with the main zero of the controller. In this way, only one specification is imposed to solve the system. Since the time constant of the inverter is very small compared to the electrical time constant it could be neglected for the computation of k_p and k_i.
So the open loop transfer function becomes:

$$GH(S) = k_i \frac{1 + s\tau_{ri}}{s} \frac{1}{R} \frac{1}{1 + \tau_e s} \tag{4.38}$$

For this system, this method is achieved by equalling τ_{ri} with τ_e.

$$\tau_{ri} = \tau_e \tag{4.39}$$

$$GH(S) = k_i \frac{1}{R} \frac{1}{s} \tag{4.40}$$

$$W_i(S) = \frac{GH(S)}{1 + GH(S)} = \frac{1}{1 + s\tau_i} \tag{4.41}$$

$$\tau_i = \frac{R}{k_i} = \frac{1}{\omega_{bw}} \tag{4.42}$$

Let's impose a bandwidth of:

$$f_{bw} = 100 \quad [Hz]$$

$$\omega_{bw} = 2\pi 100 = 628.318$$

Now by imposing the magnitude of the open-loop transfer function equal to one for an angular frequency equal to the bandwidth.

$$|GH(j\omega_{bw})| = 1 = \frac{k_i}{R}\frac{1}{\omega_{bw}} \tag{4.43}$$

So with the motor specifications k_i and k_p become:

$$k_i = 93.222$$

$$k_p = \tau_{PI} k_i = 0.1539$$

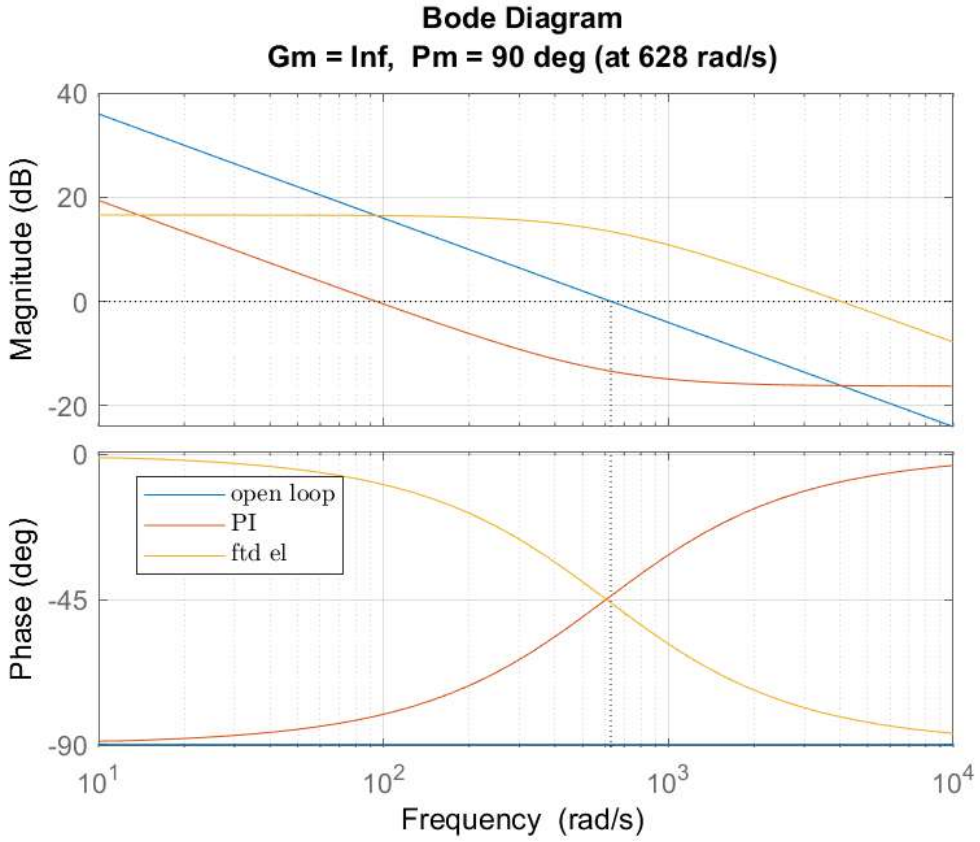


Figure 4.19: current loop bode diagram

Notice that this method significantly simplifies the computation of k_p and k_i but it works well only if the electrical parameters of the machine are well-known.

Since PLECS works in the z-domain because it samples the signal with a fixed-step solver the PI transfer function has to be transformed.
By using the Tustin method of conversion the PI transfer function becomes:

$$GI(Z) = \frac{0.06283}{z-1}$$

Fortunately, PLECS is able to transform the transfer function automatically by specifying within the PI(Z) mask the gains and the discretization method.
The integral part of the controller may have some issues if the input signal saturates. If this happens the feedback loop is broken and the system runs as an open loop because the actuator will remain at its limit regardless of the process output.
If a controller with integrating action is used, the error may continue to be integrated if the algorithm is not properly designed. This means that the integral term may "wind up" and cause a delayed output.
To avoid this problem there are several methods. PLECS allows users to choose between back calculation and conditional integration. In this study, the second method is used.
Conditional Integration switched off the integral term when the controller is saturated and the integrator update is such that it causes the control signal to become more saturated.
As a result, the integration is only turned off if the error is positive rather than if it is negative, for instance, if the controller becomes saturated at the upper limit.

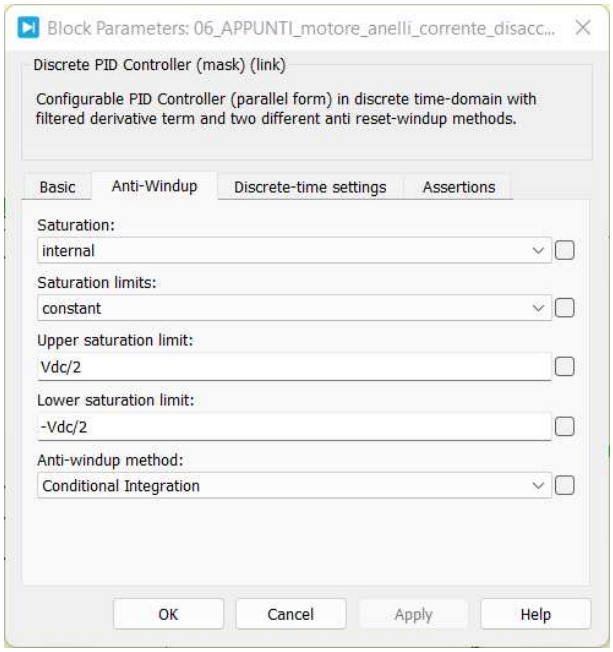


Figure 4.20: PLECS specification for the anti-Windup

4.5.2 Speed loop PI design

The design of the gain of the PI controller of the speed loop is conditioned from the design of the controller of the speed loop. The speed loop appears

only on the q-axis since only i_q produces torque (MTPA behavior).
The block scheme is shown below.

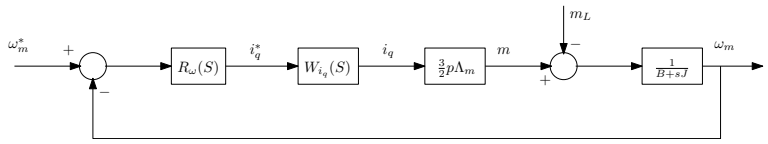


Figure 4.21: speed loop block scheme

The block W_{iq} represents the transfer function of the q-axis current loop shown in Eq. (4.41).
PI has the same general transfer function of Eq. (4.30).
The mechanic term is represented as a first-order system:

$$\frac{1}{B+Js} = \frac{1}{B}\frac{1}{1+s\tau_m} \tag{4.44}$$

$$\tau_m = \frac{J}{B} \tag{4.45}$$

The mechanical and electrical terms have been computed in section 3.4.
The open loop transfer function is:

$$GH_w(S) = \frac{k_{i\omega}}{B}\frac{1+s\tau_{r\omega}}{s}\frac{1}{1+s\tau_i}\frac{3}{2}p\lambda_m\frac{1}{1+s\tau_m} \tag{4.46}$$

There are two ways shown in this study to compute the k_p and k_i gains for the speed loop.
The first one is the zero-pole cancellation previously used for the current loop. This time the zero introduced by the controller has to match the pole of the mechanic block:

$$\tau_{r\omega} = \tau_m \tag{4.47}$$

So that the open loop transfer function becomes:

$$GH_w(S) = \frac{k_{i\omega}3p\lambda_m}{2B}\frac{1}{s}\frac{1}{1+s\tau_i} \tag{4.48}$$

By imposing a bandwidth ten times smaller than the one uses for the current loop the computation of the magnitude in the bandwidth frequency is:

$$f_{bw_\omega} = 10 \quad [Hz]$$

$$\omega_{bw_\omega} = 2\pi f_{bw_\omega} = 62.831$$

$$|GH_w(j\omega_{bw_\omega}| = 1 = \frac{k_{i\omega}3p\lambda_m}{2B}\frac{1}{\omega_{bw_\omega}}\frac{1}{\sqrt{1+(\omega_{bw_\omega}\tau_i)^2}} \tag{4.49}$$

Substituting Eq. (4.42) in Eq. (4.49).

$$|GH_w(j\omega_{bw_\omega}| = 1 = \frac{k_{i\omega}3p\lambda_m}{2B}\frac{1}{\omega_{bw_\omega}}\frac{1}{\sqrt{1+(\frac{\omega_{bw_\omega}}{\omega_{bw_i}})^2}} \tag{4.50}$$

Where ω_{bw_ω} is the bandwidth imposed for the speed loop control and ω_{bw_i} is the bandwidth imposed for the current loop control.
In the end the k_i and k_p are:

$$k_{i\omega} = 1.035$$

$$k_{p\omega} = k_{i\omega}\tau_{r\omega} = 0.1846$$

The phase margin PM becomes:

$$PM = 84.289 \quad [DEG]$$

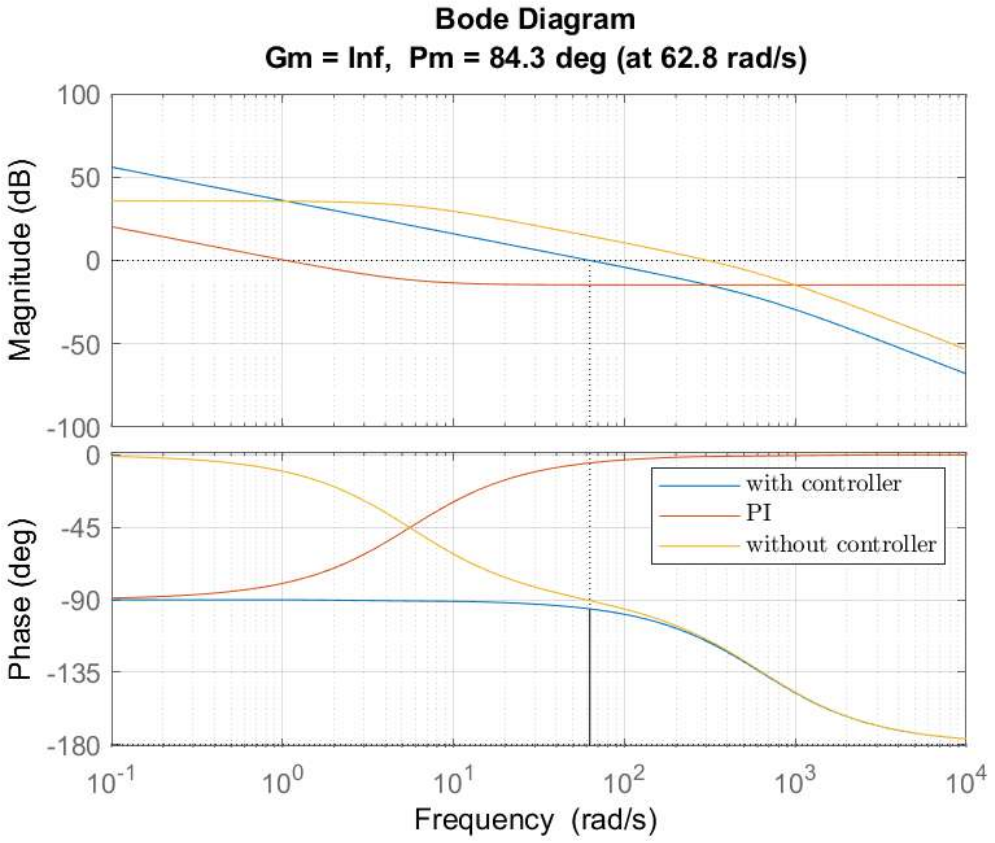


Figure 4.22: speed loop with pole-zero cancelaltion

The second method of design is the symmetrical optimum.
This method aims to maximize the phase margin of the open loop transfer function. For doing so the bandwidth of the open-loop transfer function is chosen to be halfway between the pole of the closed-loop current transfer function and the PI zero.
Since the bandwidth is imposed the term that has to be designed is the controller zero.

$$\omega_{bw_\omega} = \sqrt{\frac{1}{\tau_{r\omega}}\frac{1}{\tau_i}} \tag{4.51}$$

$$\tau_{r\omega} = \frac{1}{\omega_{bw_\omega}^2 \tau_i} = \frac{\omega_{bw_i}}{\omega_{bw_\omega}^2} \tag{4.52}$$

Starting from Eq. (4.49) and introducing Eq. (4.52) the open loop transfer function is:

$$GH_\omega(S) = \frac{3p\lambda_m}{2B}\frac{k_{i\omega}}{s}\frac{1}{1+s\tau_m}\frac{1+s\frac{\omega_{bw_i}}{\omega_{bw_\omega}^2}}{1+\frac{s}{\omega_{bw_i}}} \tag{4.53}$$

It is noticeable that the only unknown term is $k_{i\omega}$ and it is obtainable by forcing to one the magnitude in the bandwidth frequency.

$$|GH_\omega(j\omega_{bw_\omega})| = 1 = \frac{3p\lambda_m}{2B}\frac{k_{i\omega}}{\omega_{bw_\omega}}\sqrt{1+\left(\frac{\omega_{bw_\omega)}\omega_{bw_i}}{\omega_{bw_\omega)}^2}\right)^2}\frac{1}{\sqrt{1+\left(\frac{\omega_{bw_\omega}}{\omega_{bw_i)}}\right)^2}}\frac{1}{\sqrt{1+(\omega_{bw_\omega}\tau_m)^2}} \tag{4.54}$$

Solving Eq. (4.54), $k_{i\omega}$ and $k_{p\omega}$ become:

$$k_{i\omega} = 1.159$$

$$k_{p\omega} = k_{i\omega}\tau_{r\omega} = k_{i\omega}\frac{\omega_{bw_i}}{\omega_{bw_\omega}^2} = 0.1844$$

$$PM = 83.679 \quad [DEG]$$

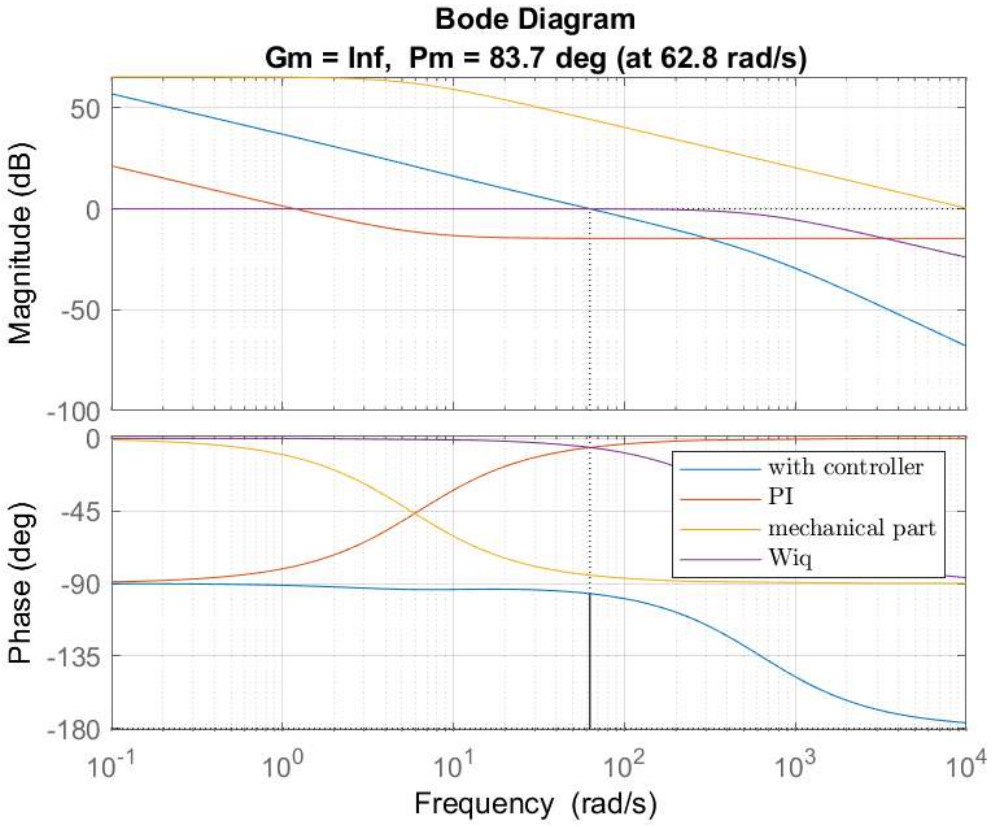


Figure 4.23: Speed loop with symmetrical optimum

The two methods give different gains that act in different ways on the control. It is possible to compare these two results.

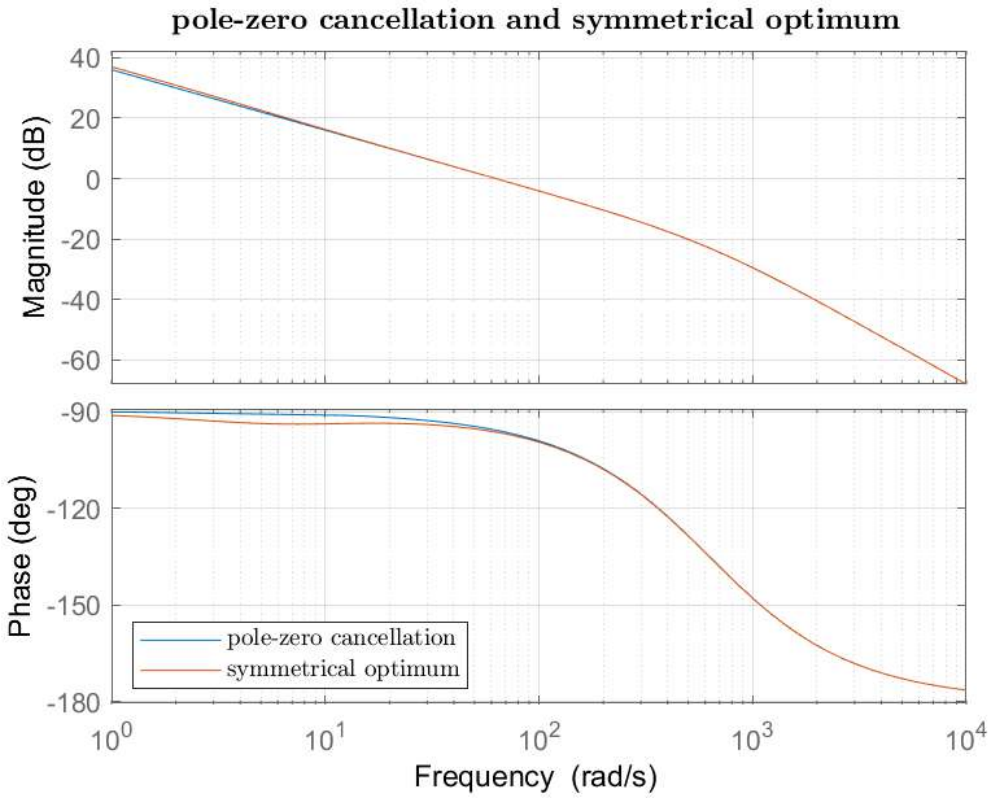


Figure 4.24: Difference between symmetrical optimum and zero-pole cancellation

It is possible to see the two methods of design have similar behavior. It means that for this bandwidth it is possible to choose one of them.

4.6 Control schemes

For the next chapter, a three-phase inverter is plugged onto the LaunchPad. The BOOSTXL-DRV8305EVM BoosterPack is a complete 3-phase driver stage in order to evaluate the motor application with the DRV8305 motor gate driver.

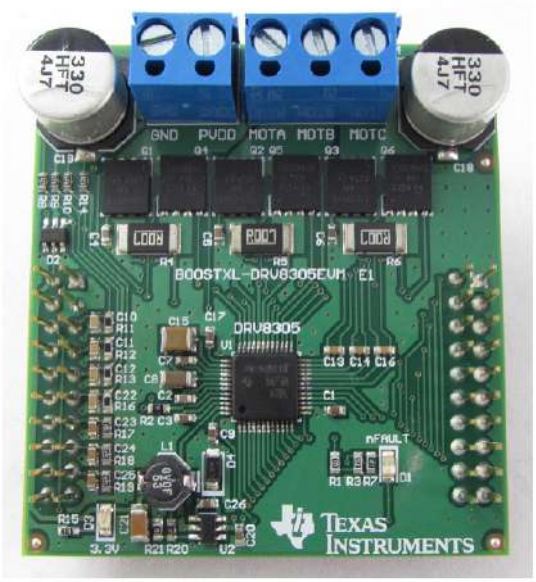


Figure 4.25: BOOSTXL-DRV8305EVM front view

The BOOSTXL-DRV8305EVM is not a standalone motor control kit and requires a compatible XL LaunchPad to provide the appropriate motor control signals. In addition to the BoosterPack and a compatible XL LaunchPad, a 3-phase motor, and a sufficient power supply are required.
The BOOSTXL-DRV8305EVM brings out a mixture of power, control, and feedback signals to the XL LaunchPad headers.

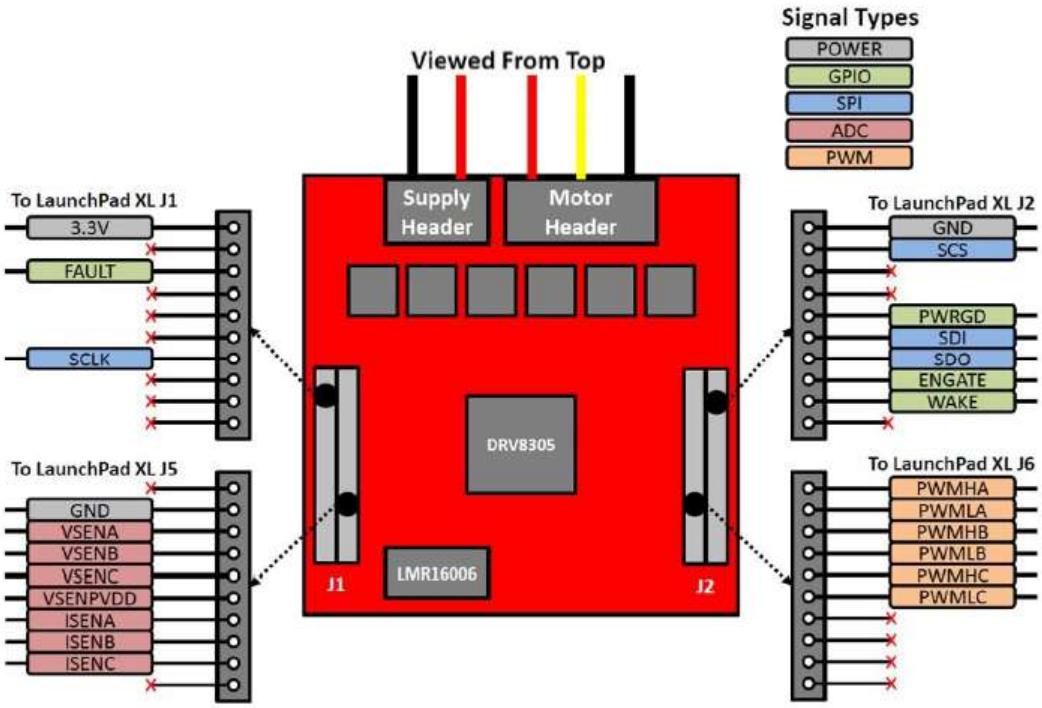


Figure 4.26: Pinout

- Terminal block headers for the power supply and motor connections
- Onboard LM16006 step-down buck regulator to provide 3.3-V power to the LaunchPad
- Fault reporting through the nFAULT and PWRGD signals
- SPI to set device configuration, operating parameters, and read out diagnostic information
- Voltage sense for the voltage supply bus and each phase output (scaled for 4.4- to 45-V operation)
- Low-side current shunt sensing on each phase (scaled for 0- to 20-A peak current operation)

The test bench is composed of a DC power supplier, an oscilloscope, a computer, a BOOSTXL-DRV8305EVM Booster pack connected to a LAUNCHXL-F28069M LauchPad, an SPM motor, and a computer where the software is made run.

Features
The following lists the BOOSTXL-DRV8305EVM features:

- Complete 3-phase drive stage in a compact form factor (2.0 in $\times$ 2.2 in)
- Supports 4.4- to 45-V voltage supply and up to 15-A RMS (20-A peak) drive current
- 6x CSD18540Q5B N-Channel NexFET™ Power MOSFETs (1.8 $m\Omega$)
- Individual motor phase and DC bus voltage sense;
- Low-side current shunt sense for each half-bridge;
- Fully protected drive stage including short circuit, thermal, shoot-through, and undervoltage protection;
- LMR16006 wide voltage input, 0.6-A step down buck regulator for MCU supply;
- Combine with compatible LaunchPad XL kits to create a complete 3-phase motor control platform;
- Optimized for the Piccolo™ LAUNCHXL-F28027F LaunchPad to support the InstaSPIN-FOC™ sensorless motor control solution.

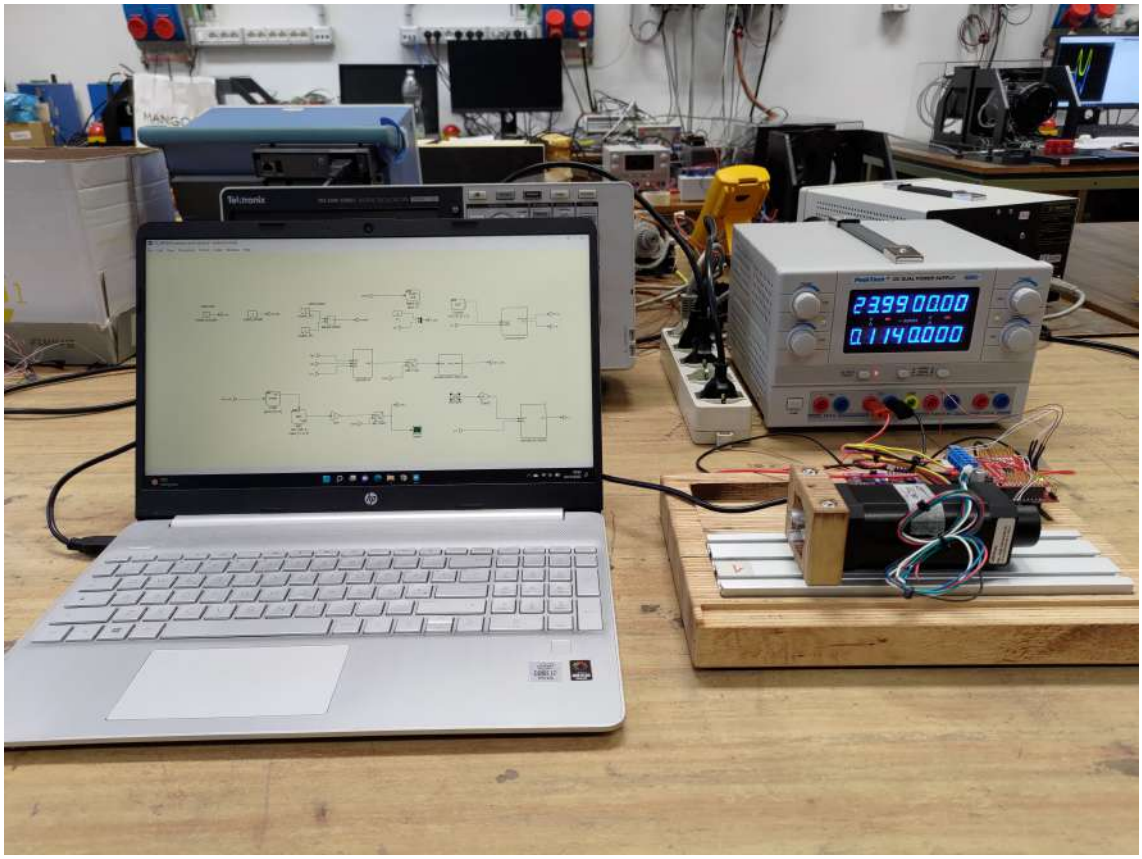


Figure 4.27: Workstation

4.6.1 Open-loop control

The first control make is the open loop control (also known as volt-Hertz control).
Here the inputs of the three phases of the motor are the tension and the frequency. Since the mask of the ePWM has as input the modulation index which can vary from 0 to 1 the input signal of the mask has to be normalized. For this reason, a C-Script mask is introduced in order to both normalize the modulation index and generate the three sine waves.

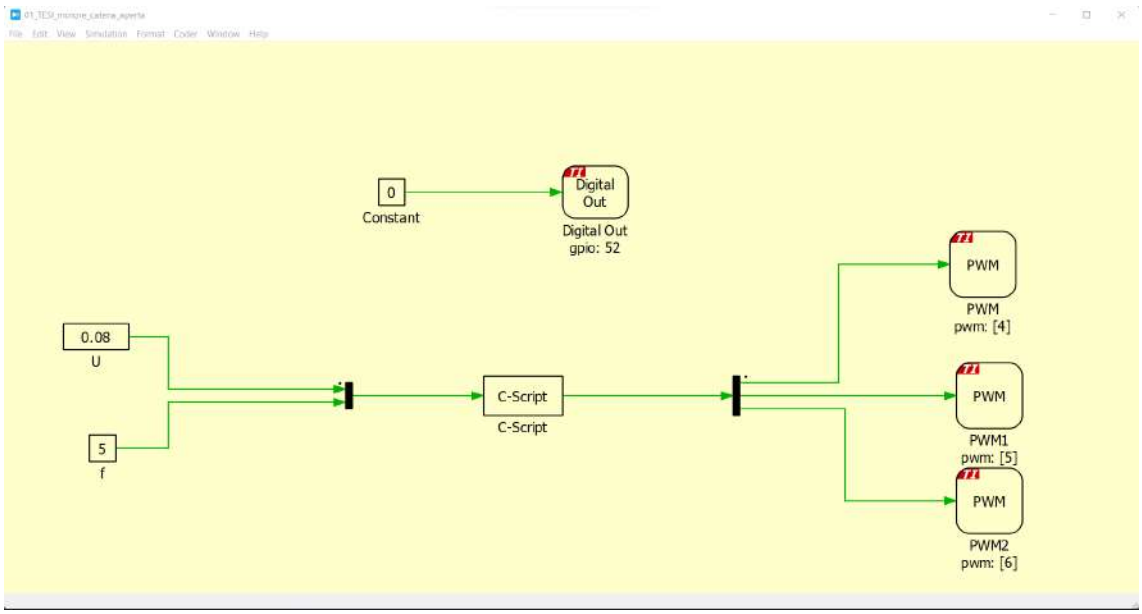


Figure 4.28: V/Hz control scheme

Listing 4.1: Code declaration

```
#include <math.h>
float pi = 3.141592653589793;
```

Listing 4.2: Output function code

```
float U = Input(0);
float f = Input(1);
if (U>0.1) {
U=.1;
}
Output(0) = 0.5 + U*sin(2*pi*f*CurrentTime);
Output(1) = 0.5 + U*sin(2*pi*CurrentTime*f+2*pi/3);
Output(2) = 0.5 + U*sin(2*pi*CurrentTime*f-2*pi/3);
```

Since the modulation index is allowed to vary from 0 to 1 the tension input is a signal that can range from 0 to 0.5.
With a reference of zero volts, the duty cycles are centered at 0.5. With this configuration, the inverter always operates in linearity with a peak first harmonic tension equal to :

$$\hat{V}_{oa} = m\frac{V_{DC}}{2} \tag{4.55}$$

There are several protections in the scheme. The first one is the enabling of the ePWM registers. With a constant which can be 0 or 1, it is possible to enable the ePWM masks.
The second one is a saturation limit of 0.1 in the C-Script block with the aim of avoiding overmodulation by mistake.
The Volt-Hertz control is an easy way to control a motor but it has disadvantages too.
The main disadvantage of open loop control is that it is not able to reject

disturbances. In motor control, it means that if there is a resistance torque applied on the shaft the control can't work properly. For this reason, close-loop controls are used.

4.6.2 Current control

As shown in fig. (4.18) the current control is split into two controls. The first is in relation to the d-axis, while the second is in relation to the q-axis.
The design of the PI for this control is already studied in the dedicated paragraph.

The BoosterPack has low-side current shunt sense for each half-bridge (phases A, B, and C). The current sense setup takes advantage of the DRV8305's triple shunt current amplifiers (phases A, B, and C).
The differential amplifier senses the voltage across a 0.007-Ω power sense resistor with differential connections.
The following image shows the scheme of the current sense

$$\begin{aligned} R_{SENSE} &= 7 \quad [m\Omega] \\ R_{GAIN} &= 50 \quad [k\Omega] \\ R_{REF} &= 50 \quad [k\Omega] \\ R_1 &= 5 \quad [k\Omega] \end{aligned}$$

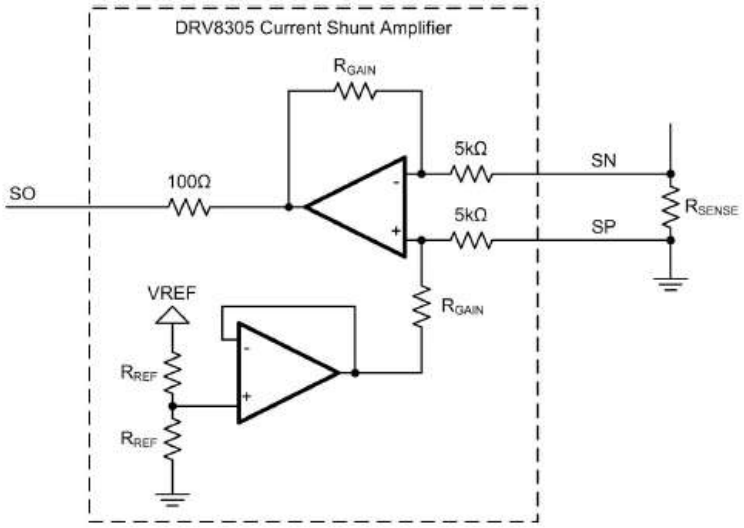


Figure 4.29: Current sense

$$v_{SO} = \frac{v_{REF}}{2}\left(1+\frac{R_{GAIN}}{R_1}\right)\frac{R_1}{R_1+R_{GAIN}}+v_{SP}\left(1+\frac{R_{GAIN}}{R_1}\right)\frac{R_{GAIN}}{R_1+R_{GAIN}}-v_{SN}\frac{R_{GAIN}}{R_1} \tag{4.56}$$

And substituting the resistance parameters with their quantities:

$$v_{SO} = \frac{v_{REF}}{2} + 10(v_{SP} - v_{SN})$$

Where $v_{REF} = 3.3 \quad [V]$ and $v_{SP} - v_{SN}$ can be rewritten as:

$$v_{SP} - v_{SN} = iR_{SENSE}$$

v_{SO} is the tension measured at the pin of the ADC which is the output signal of the ADC block. So the current is computed as:

$$i = \frac{1}{10R_{SENSE}}\left(v_{SO} - \frac{v_{REF}}{2}\right) \tag{4.57}$$

So after the ADC mask for every phase, there will be a bias term of $\frac{v_{REF}}{2}$ and a gain of $\frac{1}{10R_{SENSE}}$.

$$\frac{v_{REF}}{2} = 1.65 \quad [V]$$
$$\frac{1}{10R_{SENSE}} = 0.007 \quad [\Omega]$$

Since every phase has a little variation in the bias quantity it is better to subtract an independent value for each phase. By imposing a reference current of zero the bias component of every ADC register is set by tuning the value until every phase current has a mean value equal to zero. (the waveform of the current isn't constant because of the disturbances).
As said in previous chapters the ADC can start the conversion with an external trigger source. This signal can be a PWM output as well. The PWM block has the possibility to generate a trigger signal for the ADC.
PLECS doesn't allow users to customize the SOC for the ADC. So it uses always SOC0 for every ADC block. This can cause errors during the compilation of the program if it is used three different ADC blocks as current sensors. One SOCx can't be used for ADC of the same register because it is impossible to order the sequence of conversion. To solve this problem the same ADC block is used for every phase. In the Analog input channel, it is written the ADC used and here it will be written the three different inputs. In this way, PLECS order automatically the SOC for every analog input (The first signal is converted with SOC1, the second signal with SOC2, and the third one with SOC3).
The analog input for each phase is ADCINA3 for phase A, ADCINB3 for phase B, and ADCINA4 for phase C. It is important to note that while the TI28069 MCU has two ADC's, ADCINAx and ADCINBx, the ADC units are structured with a common results register.
Therefore, when addressing ADCINBx channels, the ADC unit setting should be "ADC A" and the channel offset by a factor of 8. In this case, ADCINB3 should be entered with an ADC unit value of "ADC A" and an Analog input channel(s) value of 11.

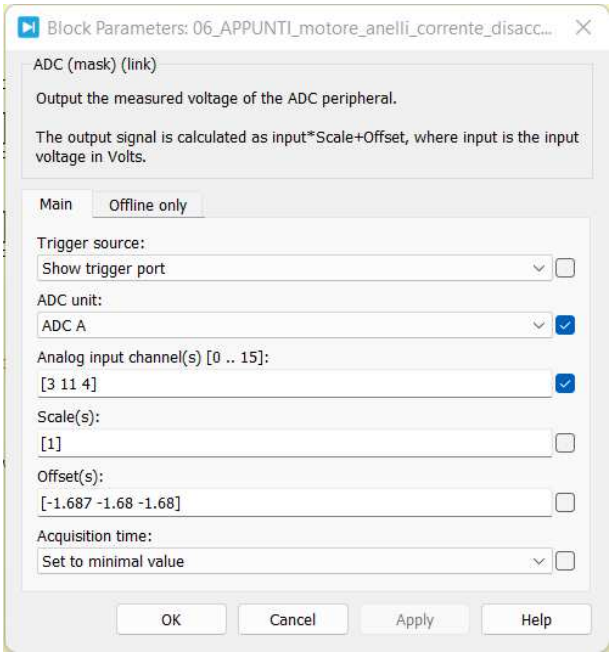


Figure 4.30: ADC block

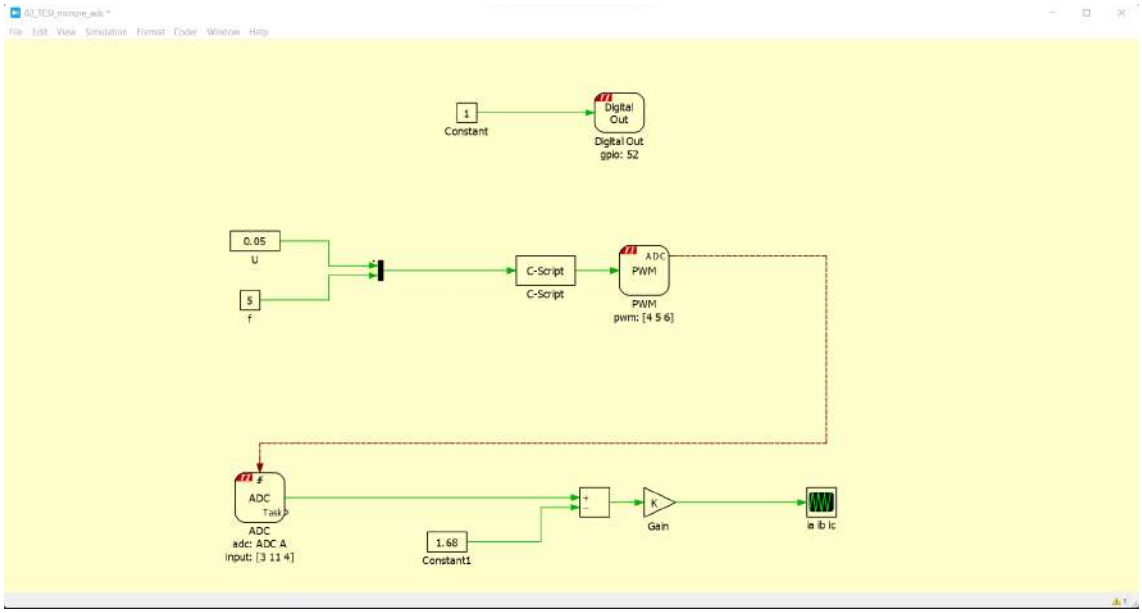


Figure 4.31: V/Hz control scheme with ADCs. In this case, the bias is subtracted outside the ADC block

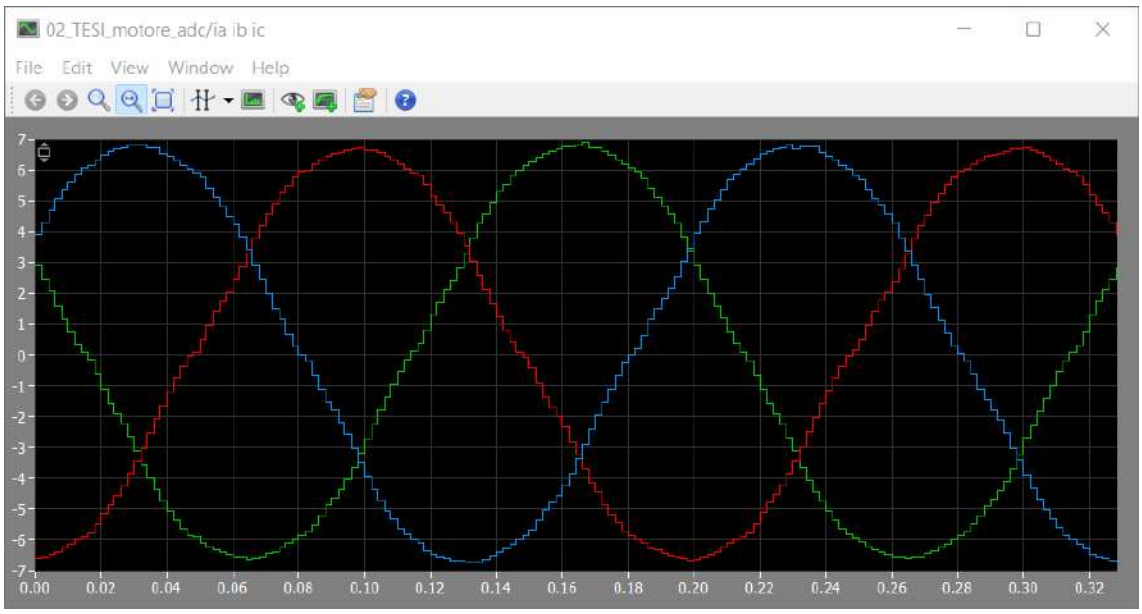


Figure 4.32: Scope of the three currents

The encoder block is used to know the rotor position of the motor. In this case, the QEP module used is the first and the GPIOs where the encoder is plugged in are GPIO20 for the QEPA, GPIO21 for the QEPB, and GPIO23 for the QEPI. In this case, the encoder connected to the motor doesn't have a hardware reset index so GPIO23 is set to the encoder mask but it is connected to nothing.
PLECS doesn't allow users to impose a software command as a reset index. This cause problem with the alignment procedure. It is possible to generate a reset index by using PLECS blocks.

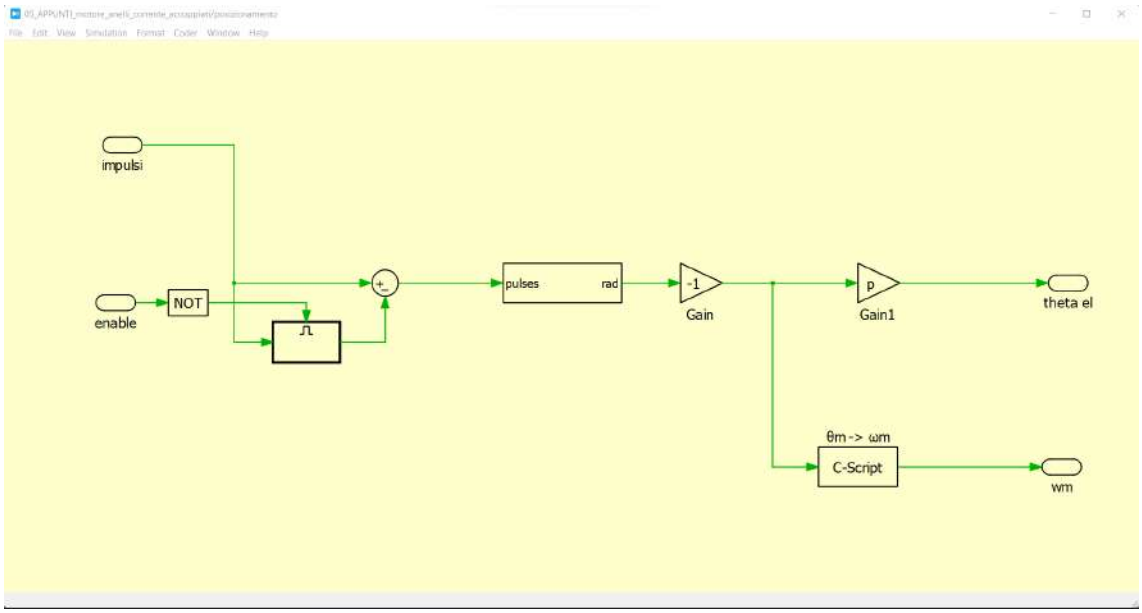


Figure 4.33: Enable of the encoder

This solution is useful for the alignment procedure but it doesn't reset the counting actually. This method fakes the output signal but the encoder is not

reset. To reset properly the motor encoder a button has to be connected to the GPIO23 so that it is possible to use it as a hardware input.
The gain mask set to -1 is introduced in order to match the direction of rotation between the encoder and the motor.

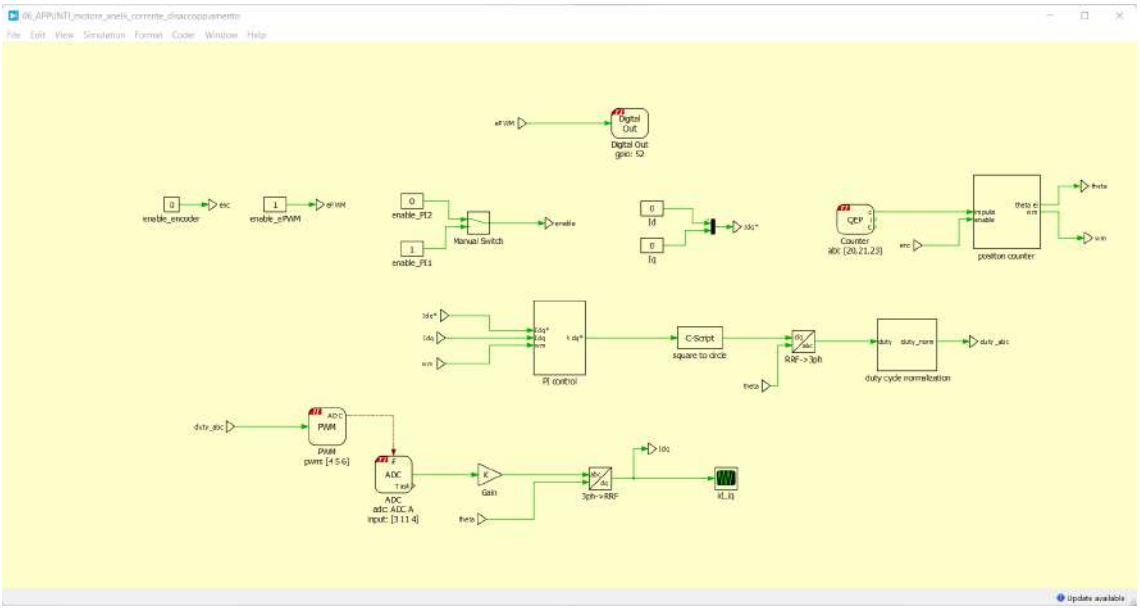


Figure 4.34: Current control overall scheme

Listing 4.3: Model initialization command

```
pi = 3.141592653589793;
k = 2*pi/4095;
p = 4;
L = 0.245e-3;
R = 0.1484;
lambda = 0.00547;
Vdc = 24;
kp = 0.1679;
ki = 84.1039;
Ts = 1e-4;
n = 20;
```

The controller part of the scheme is shown in Fig. (4.16) and the PI gains are already computed in the previous sections.
The output signals of the controller are u_d^* and u_q^*. These signals are both bounded to $\pm\frac{Vdc}{2}$. In the d-q plane, these limitations are represented by a square. If the tension exceeds the boundary of the DC bus the duty cycle will be in over-modulation. This means that the inverter is working in linearity for no longer. The tension u is the magnitude of u_d^* and u_q^* and so the boundary

of tension must be written for u instead of u_d^* and u_q^*.
There is the possibility that one of u_d^* or u_q^* could be lower than $\frac{Vdc}{2}$ but the vector u don't. In this case, the controller believes that the inverter is operating linearly when it is not.
A C-Script block is introduced to bound the output signals within a circle of radius $\frac{Vdc}{2}$ in order to prevent the over-modulation problem.

Listing 4.4: Code declaration

```
#include "math.h"
real_t Vdc;
float u,ud,uq;
```

Listing 4.5: start function code

```
Vdc = ParamRealData(0,0);
u=0;
ud =0;
uq=0;
```

Listing 4.6: Output function code

```
ud = Input(0);
uq = Input(1);
u = sqrt(powf(ud,2) + powf(uq,2));
if (u > Vdc/2){
Output(0) = ud/u*Vdc/2;
Output(1) = uq/u*Vdc/2;}
else{
Output(0) = ud;
Output(1) = uq;}
```

After that, the signal has to be normalized. So the outputs of the C-Script block go into a subsystem that produces the three-phase modulation indexes.

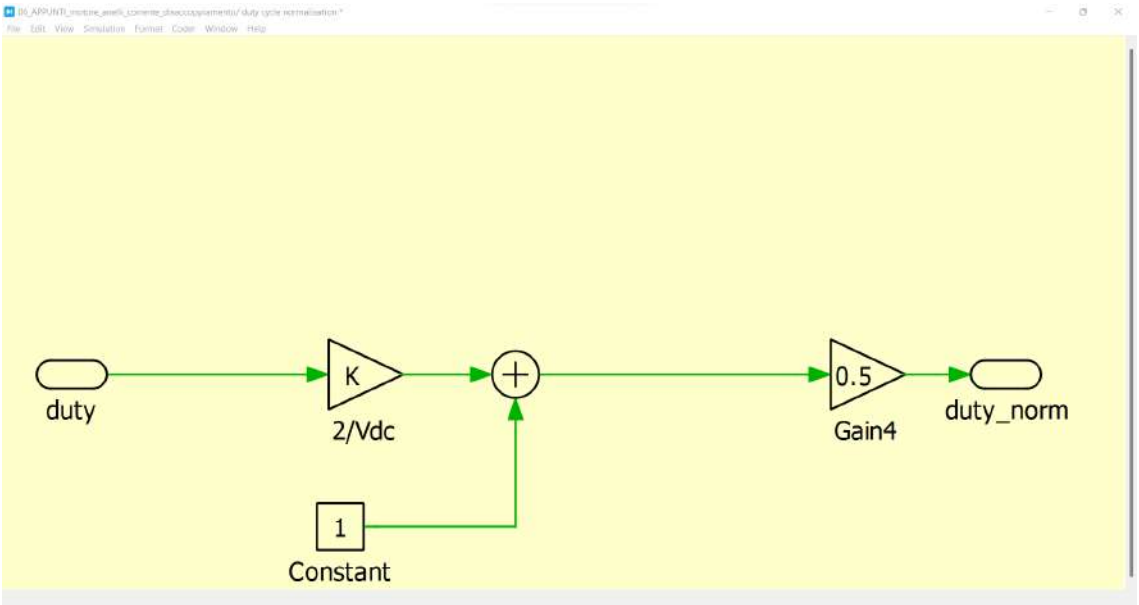


Figure 4.35: Duty cycle normalization

Before running the motor, it is necessary to align the rotor with the d-q axes. Since the d-q transformation has a rotating reference the electrical angle of the motor has to be the same as the d-q reference.
The alignment procedure is made by feeding the motor with a I_d impulse. The rotor will align with the axis of phase A. This will be the reference for the electrical angle. After that, the encoder will be enabled to start counting the position and the rotor will be aligned with the d-q reference.
The procedure is made in steps that are shown here below:

1. The first thing to do is to ensure that all the input signals are off. So every term has to be set to zero except for the enable of the PI which is active low and so it has to be set to one.

2. The first block to switch to one is the enable of the ePWM. By doing this the inverter is turned on and $V_{REF} = 3.3 \quad [V]$. Notice that before the enabling of the ePWM, V_{REF} was equal to zero. This means that the outputs of the ADCs were three constant currents equal to -23.5 [A].To avoid current pulses which can damage the inverter the PI enables is left to one.

3. After turning on the PWM of the three phases the PI is allowed to work by switching the enable command from one to zero.

4. After that a pulse of I_d is imposed in order to align the rotor.

5. Once the rotor is aligned the encoder can start counting the position by switching from zero to one the enable of the encoder.

6. Since the behavior of the motor has to be on MTPA the I_d reference has to turn off and the I_q reference has to turn on.

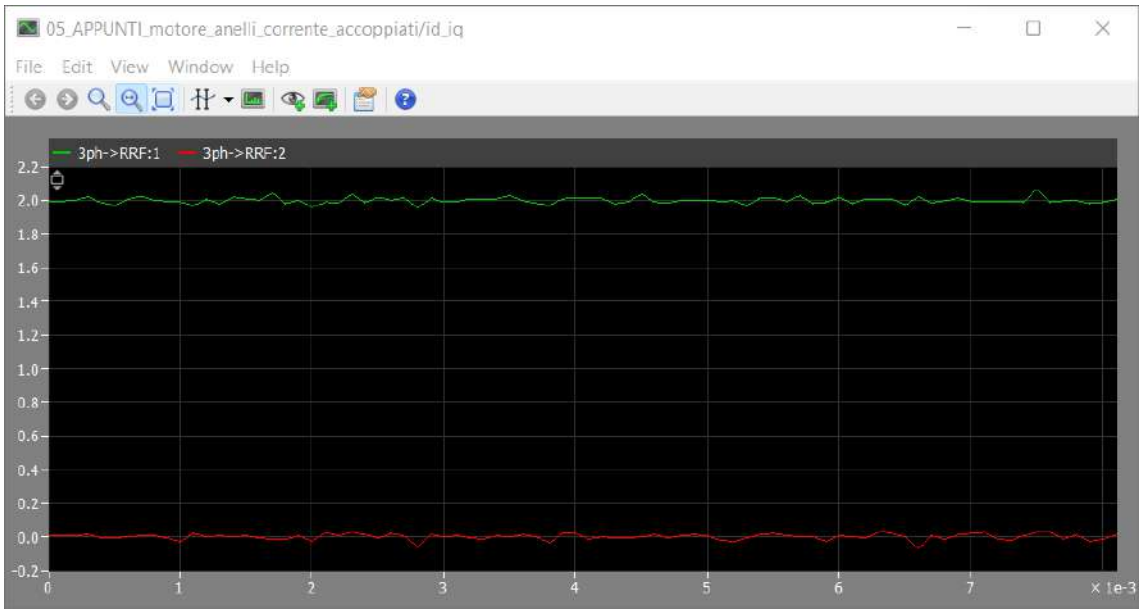


Figure 4.36: I_d and I_q scope with a reference of $I_d = 2$ and $I_q = 0$

PLECS allows Users to use a state machine.
State machines are a formalism for event-driven systems that move from one

discrete state to another in response to discrete events.
PLECS lets the user graphically create and edit state machines using common concepts such as boxes for states and curved arrows for transitions, and simulate them together with a surrounding system.
This important tool given by PLECS is useful to automate the alignment procedure.

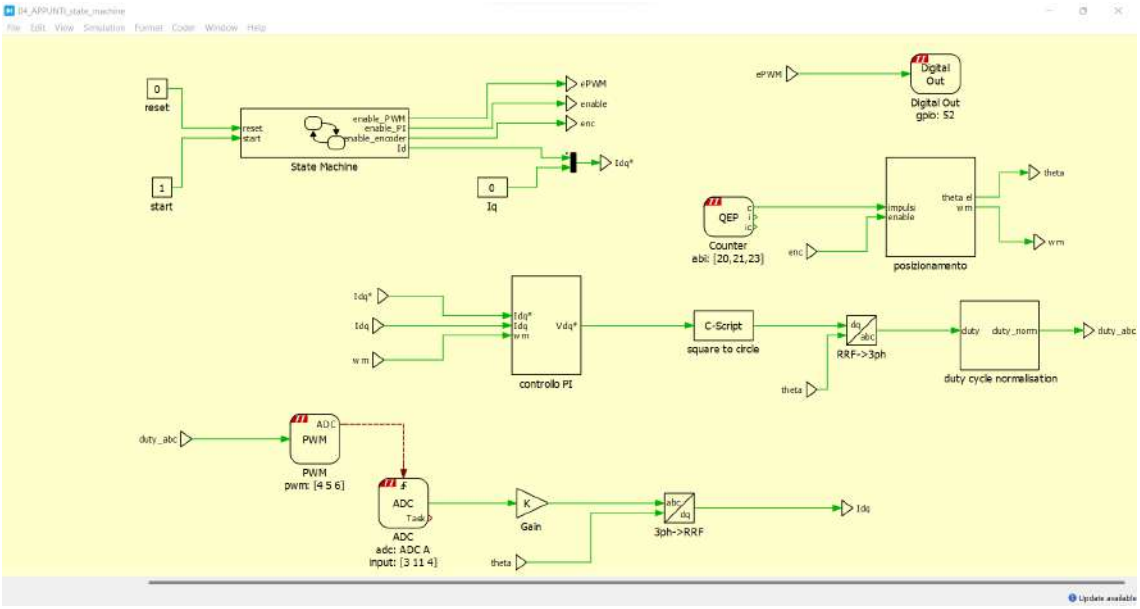


Figure 4.37: Overall scheme with a state machine

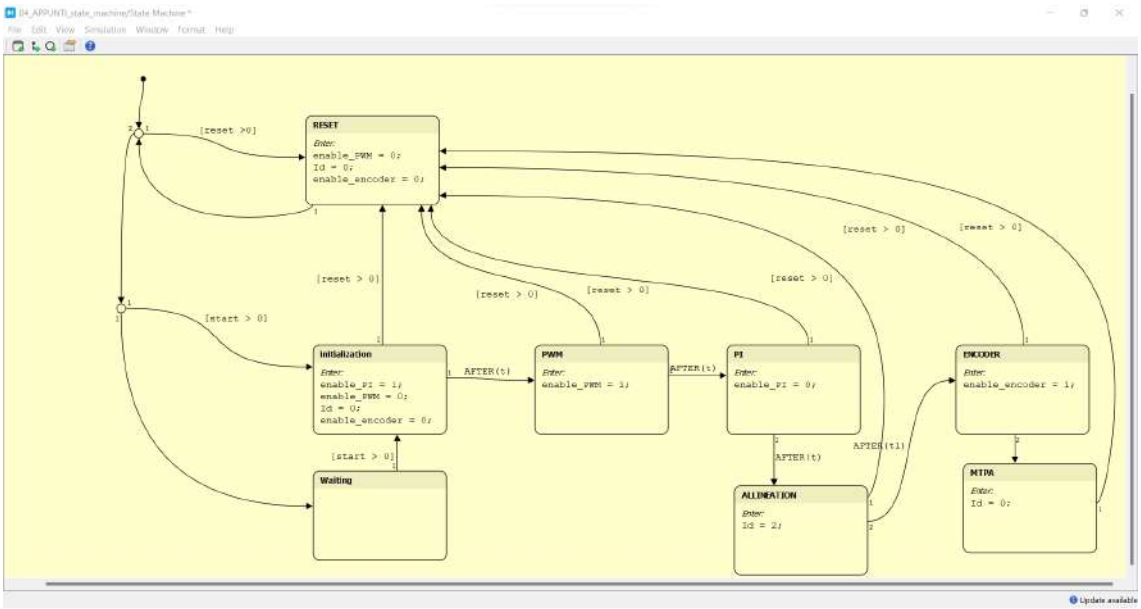


Figure 4.38: State machine

In fig.(4.38) is shown how the alignment procedure is implemented.
As the C-Script block also with the state machine it is requested to declare the input, output, and constant state that are used within the subsystem. In this case, the signals are:

- reset input continuous signal that reset the alignment procedure;
- start input continuous signal that starts the alignment procedure;
- enable_PWM output continuous signal that starts the inverter;
- enable_PI output continuous signal that enables the integrator of the PI controller;
- Id output continuous signal that gives the Id reference current in Ampere;
- enable_encoder output continuous signal that makes the encoder starts counting;
- t = 1 constant value used with the command **AFTER**(t). This command delayed the response of the next box of a certain time expressed in seconds;
- t1 = 3 constant value used with the command **AFTER**(t1). This value is used in order to wait for the rotor until it is aligned with the d-axis.

With the current control, it is possible to regulate the current reference but it is not possible to regulate the velocity. By applying a I_q reference it is possible to have torque in the motor but the velocity is not controllable. For doing that a speed loop is necessary. Since the current loop has been designed with the gains of the PI, it is interesting to compare the output of the controller simulated with Matlab and the actual output measured with an oscilloscope.
A current probe is connected between the A-phase of the motor and an oscilloscope. With the PLECS scheme of the current loop, it is imposed a I_d current equal to one Ampere. Then with the oscilloscope, it is detected the transient of the current with the Normal mode.
After that, the waveform of the current is exported to Matlab where it is compared with the step response simulated.
The step response is taken from $I_d = 3 \quad [A]$ to $I_d = 4 \quad [A]$ in order to avoid the effect of the blanking time that can disturb the current loop for low voltage.

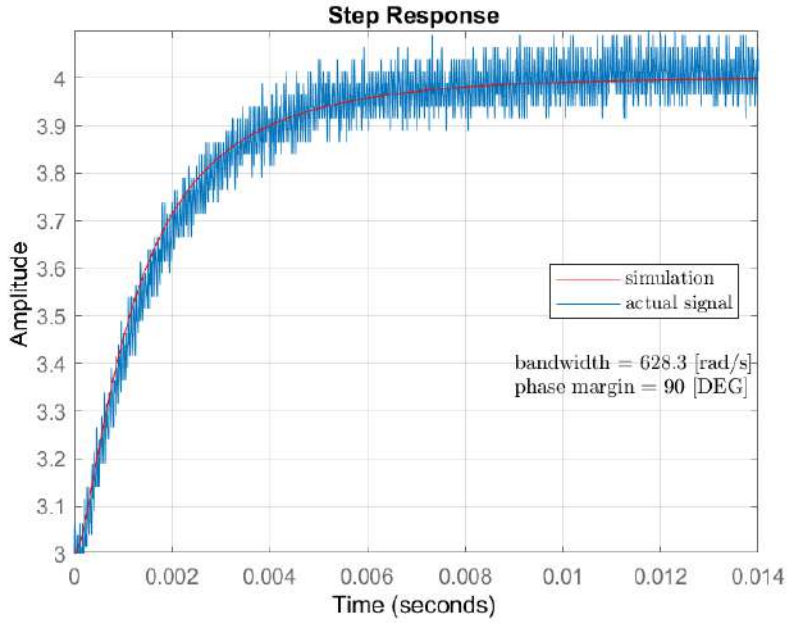


Figure 4.39: Difference between actual output and simulated output

In Fig. (4.39) the simulated step response is compared with the actual signal measured with an oscilloscope. It is noticeable to see that these two signals have the same behavior, this means that the design of the control has been done correctly.
In order to confirm that the electrical parameters of the motor are measured correctly it is possible to design the control for another bandwidth. For example, in Fig. (4.40) the control is designed to fulfill a bandwidth of $f_{bw} = 200 \quad [Hz]$.

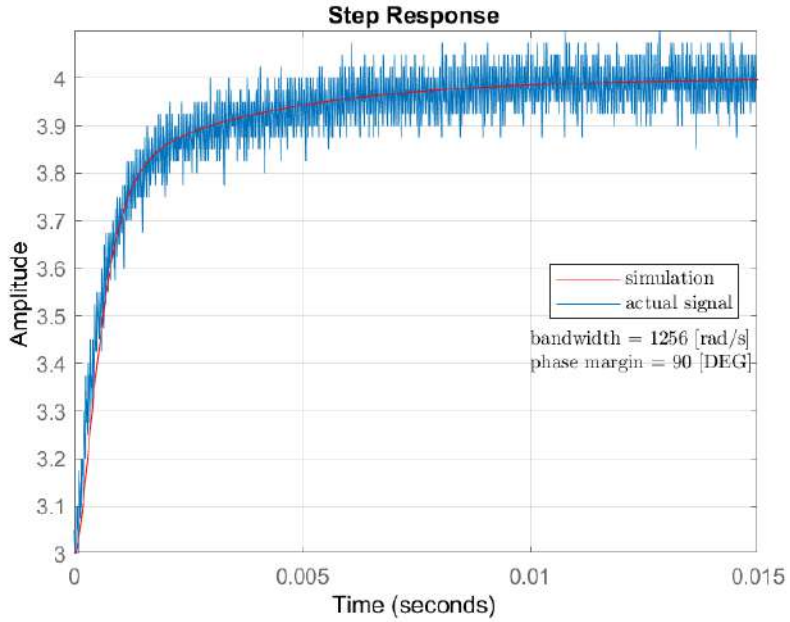


Figure 4.40: Difference between actual output and simulated output

It is possible to verify the control even with a PI more aggressive. By increasing the bandwidth of the controller in order to stay close to the PWM frequency the system will not behave like a first-order system but it will have an overshoot. To simulate this effect the PWM frequency has been lowered to $f_{PWM} = 2 * 1e - 4 \quad [Hz]$ and the gains of the controller have been set at: $K_i = 84.48$, $K_p = 0$.

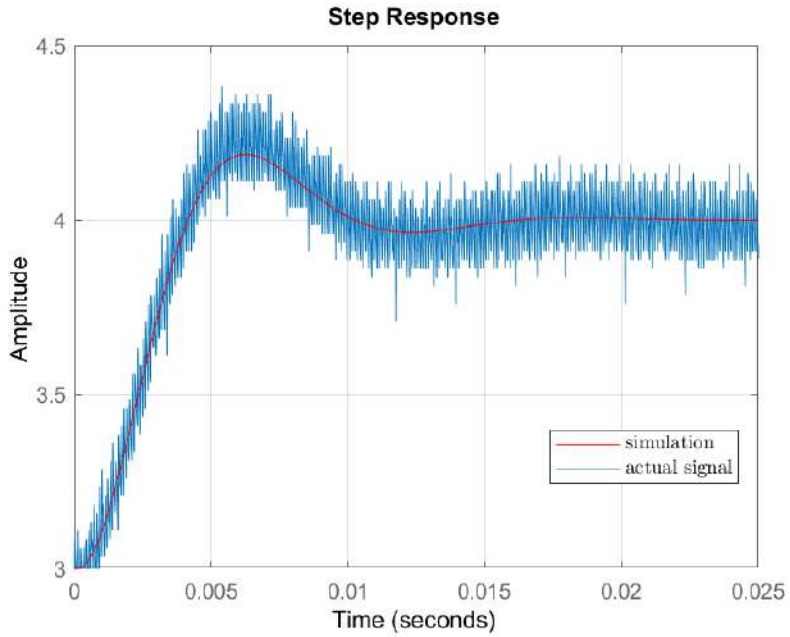


Figure 4.41: Step response of the system with $K_1 = 84.48$ and $k_p = 0$

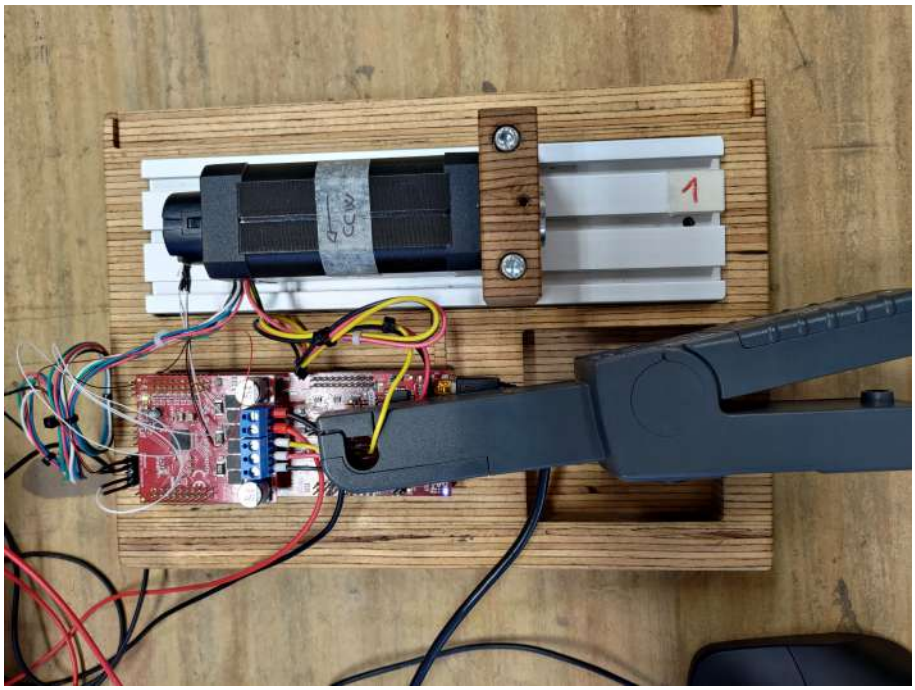


Figure 4.42: Current probe connect to the phase A of the motor

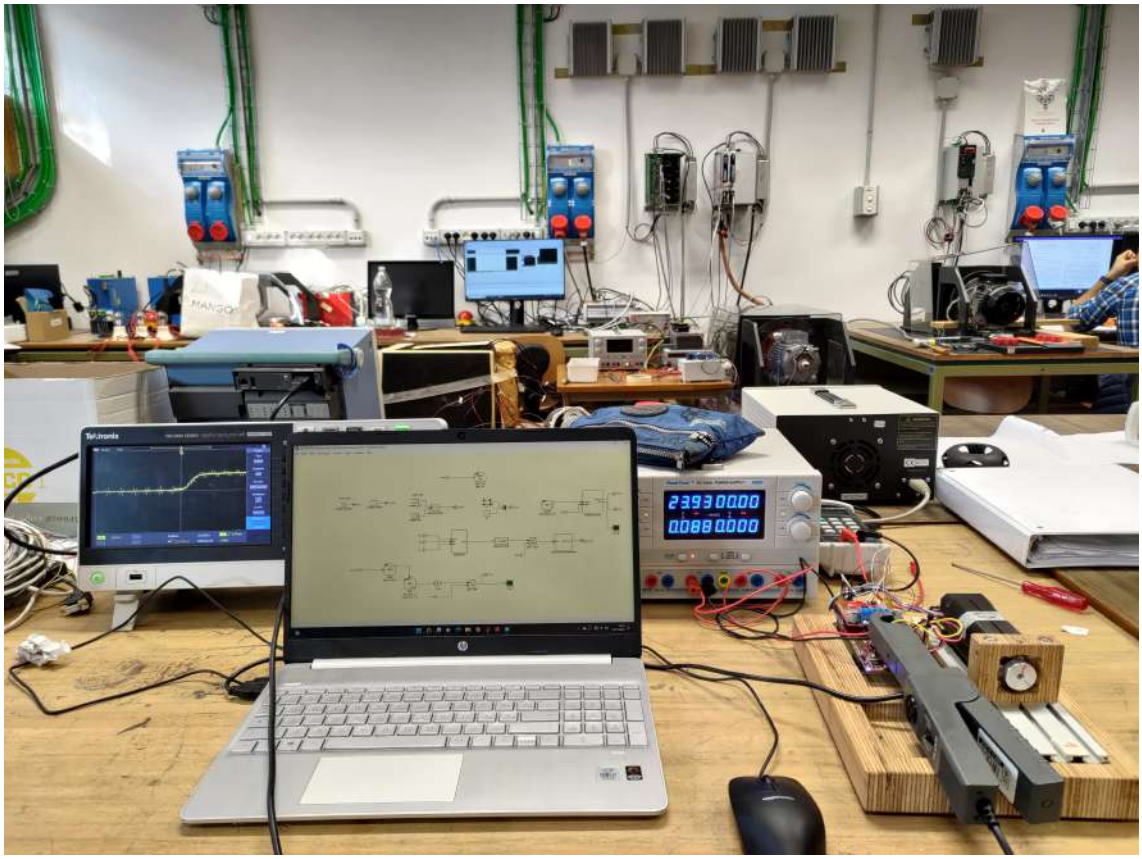


Figure 4.43: Workstation

The step response of the current loop is made in the I_d axis after the alienation procedure in order to take the measure with the rotor stationary.
In this way, there isn't the back e.m.f. produced by the magnet's flux.
The voltages equation Eq. (4.22) becomes:

$$v_d = RI_d + \frac{dI_d}{dt} \tag{4.58}$$

With a step of one voltage the current at a steady state becomes:

$$I_d = \frac{v_d}{R} = 6.7385 \quad [A]$$

It is noticeable that for a current step response the voltage needed is very low. Having a DC bus of $V_{DC} = 24 \quad [V]$ means that the modulation index is low as well.
For this reason, the blanking time of the inverter becomes relevant.
If the step response is measured with a current from $I_d = 0 \quad [A]$ to $I_d = 1 \quad [A]$ the voltage reference at steady state for the inverter is:

$$v_d = RI_d = 0.1484 \quad [V]$$

This value compared to the V_{DC} is very low and the blanking time of the inverter disturbs the control of the motor.

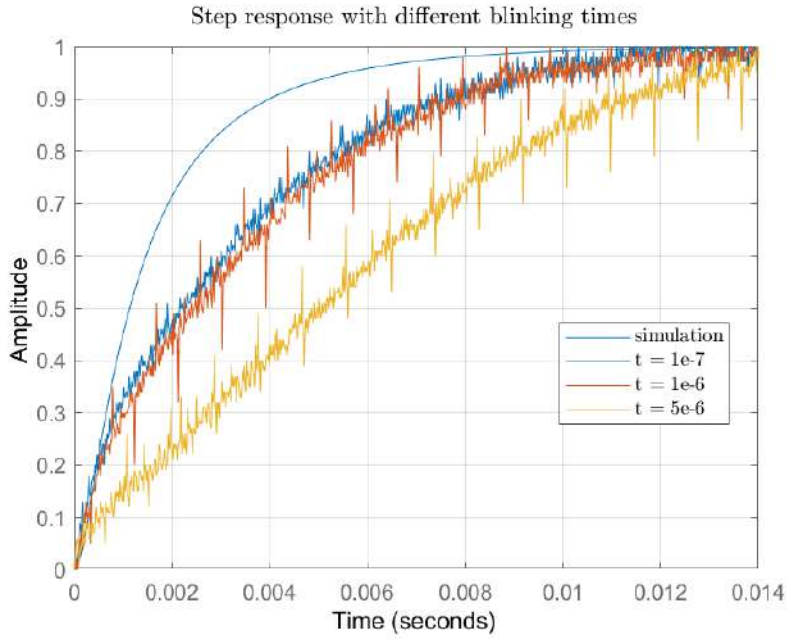


Figure 4.44: Step response with different blanking times

4.6.3 speed loop

Since the control of the velocity of the motor isn't controllable with a current loop, a speed loop is necessary.
For this kind of control, the speed of the rotor is needed as a reference for control.
The encoder mask counts the position of the rotor but doesn't measure the speed.
Angular velocity is the derivative of position in time which in discrete dominion can be re-write as the incremental ratio of the position.

$$\omega(t) = \frac{d\theta(t)}{dt} \quad \omega(k) = \frac{\theta_k - \theta_{k-1}}{T_c} \tag{4.59}$$

The derivative amplifies the rumor oscillation and so the angular speed is more sensitive to noise than the position.
A filter is necessary to ensure a good measurement. There are several filters that are good for this aim. The one shown in this study is based on the counting of samples separated in time. Looking at Eq. (4.59) is shown that the angular velocity is the incremental ratio of the angle θ. If they are taken N samples for the incremental ratio the formula become:

$$\omega(k) = \frac{\theta_k - \theta_{k-N}}{(N-1)T_c} \tag{4.60}$$

In Z-domain this equation becomes:

$$\omega(z) = \frac{\theta(z) - \theta_(z)z^{-N}}{(N-1)T_c} = \frac{\theta(z)(1-z^{-N})}{(N-1)Tc} \tag{4.61}$$

and the transfer function is :

$$\frac{\omega(z)}{\theta(z)} = \frac{1 - z^{-N}}{(N-1)Tc} = \frac{z^N - 1}{z^N(N-1)Tc} \tag{4.62}$$

The last formula of Eq.(4.62) is an LPF in the discrete form that aims to eliminate high-frequency noise. Since PLECS work at the same fixed step to building an embedded code the sample time is the same for every block.

$$T_c = 1e-4 \quad [s] \quad \frac{1}{Tc} = 1e4 \quad [Hz]$$

The bandwidth of the filter is:

$$\omega_{BW} = \frac{1}{NT_c}$$

For example:

$$\begin{cases} N = 20 & \omega_{BW} = 500 \quad [Hz] \\ N = 100 & \omega_{BW} = 100 \quad [Hz] \end{cases}$$

The code for the speed calculation is made by a C-Script block. Here below is shown the code for an angular speed calculation with $N = 20$

Listing 4.7: Code declaration

```
#include "math.h"
static real_t Ts, n_speedcalc;
static real_t theta_diff, omega;
#define pi 3.1415926f
static real_t posizione[20];
static int ii;
```

Listing 4.8: Start function code

```
Ts = ParamRealData(0,0);
n_speedcalc = ParamRealData(1,0);
ii=0;
```

Listing 4.9: Output function code

```
if (IsMajorStep){

        posizione[ii] = Input(0);
if (ii == n_speedcalc-1){
theta_diff = posizione[ii]-posizione[0];
        ii = -1;
        }
        else {
theta_diff = posizione[ii] - posizione[ii+1];
}
if (theta_diff<-pi/2 || theta_diff>pi/2){

if (theta_diff < 0){
```

```
theta_diff = theta_diff+2*pi;
}
else {
theta_diff = theta_diff-2*pi;
}
}

omega = theta_diff/(Ts*n_speedcalc);
Output(0) = omega;
}
```

Listing 4.10: Update function code

```
ii++;
```

The last lines of the code have been made because the difference between two angles can be a negative term if N is too large or the motor is running at high speed.
The speed loop controller is made to regulate the angular speed of the SPM motor.

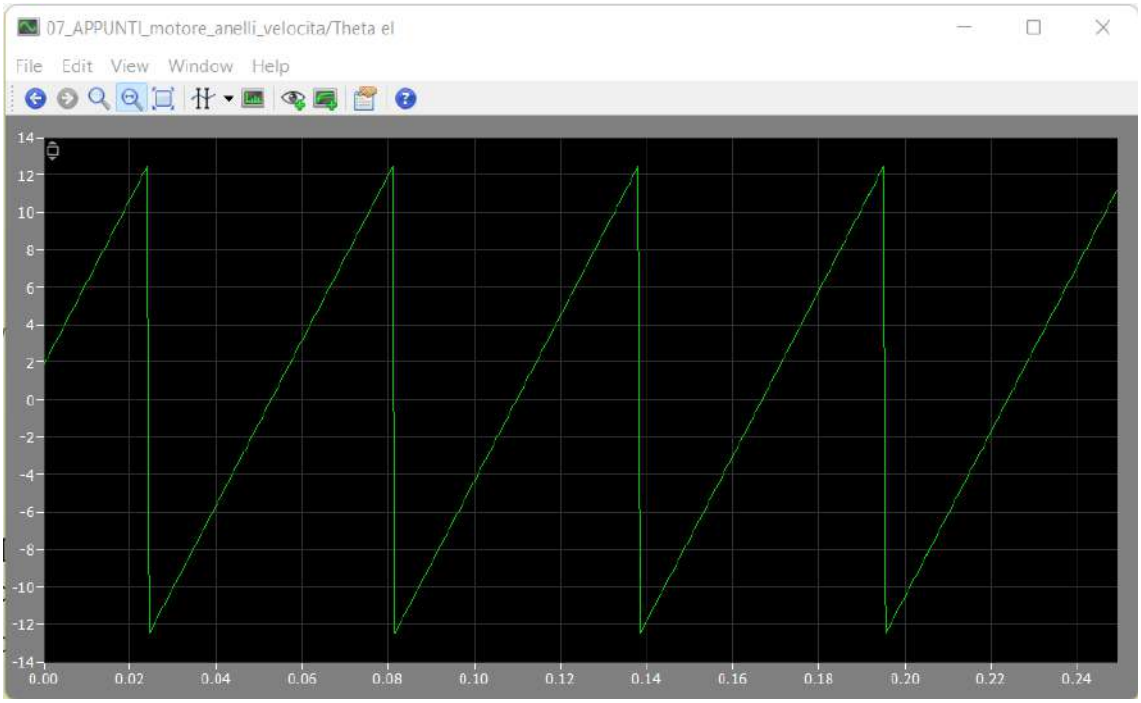


Figure 4.45: Electrical angle

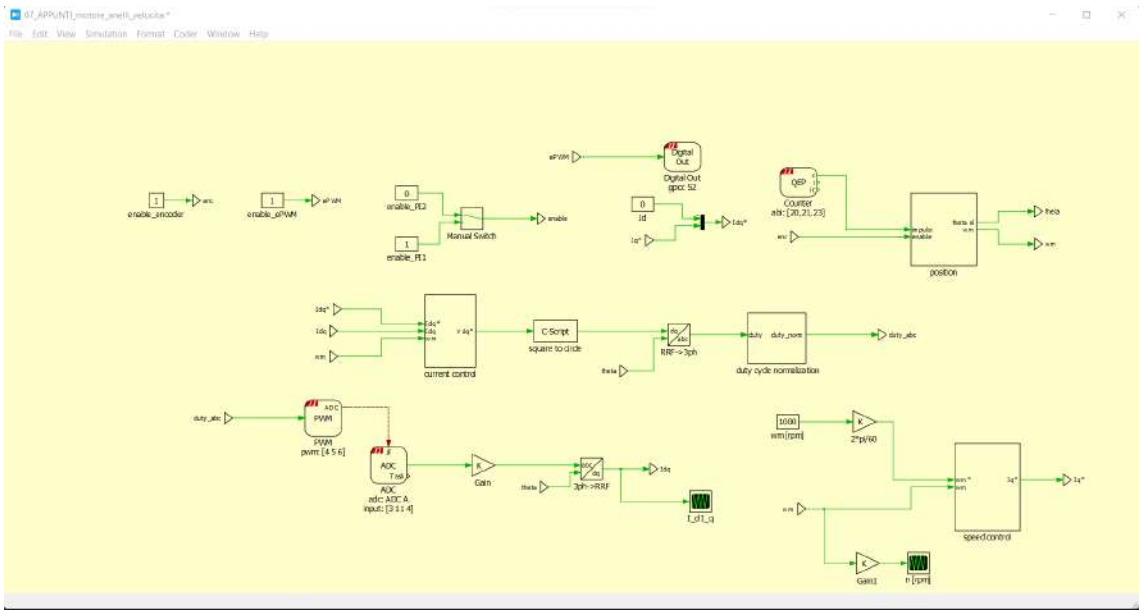


Figure 4.46: Speed control overall scheme

Listing 4.11: Model initialization command

```
pi = 3.141592653589793;
k = 2*pi/4095;
p = 4;
L = 2.4508e-04;
R = 0.1484;
lambda = 0.0055;
Vdc = 24;
kp = 0.1679;
ki = 84.103;
Ts = 1e-4;
n = 20;
% symmetrical optimum
% kp_w = 0.184482;
% ki_w = 1.159132;
% zero-pole cancellation
kp_w =0.184668;
ki_w = 1.035591;
```

The alignment procedure is the same as the current loop scheme. In this exercise, the reference is not I_q but the angular speed in rpm.

Figure 4.47: I_d and I_q

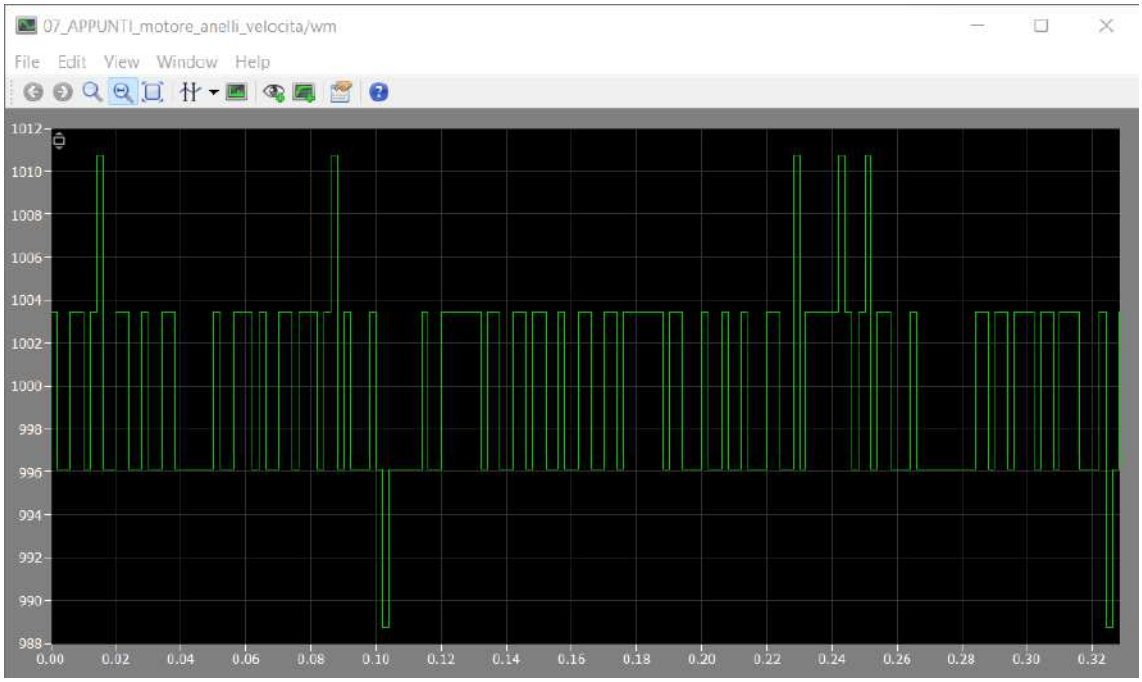


Figure 4.48: Angular speed [rpm]

With the speed loop, it is possible to control the velocity of the motor even in the on-load behavior. It will be the Iq current that will adapt the torque in order to resist the load.

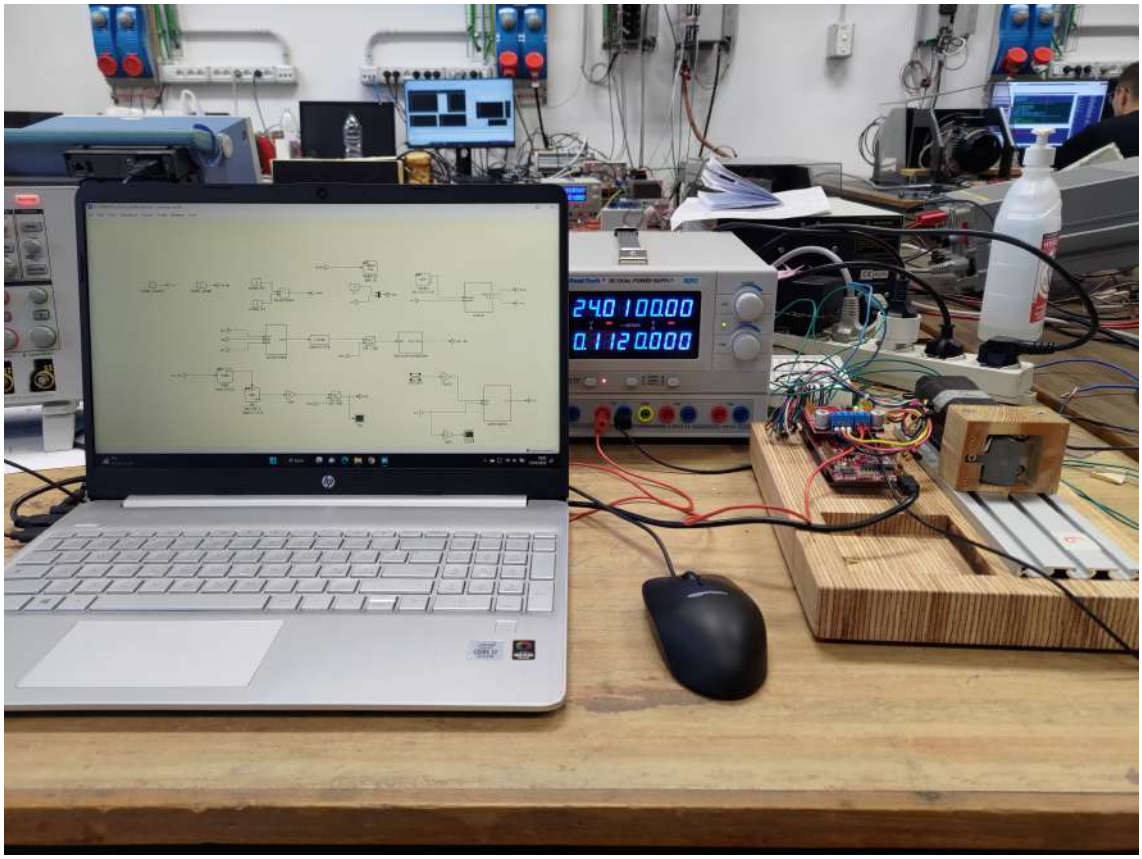


Figure 4.49: Workstation

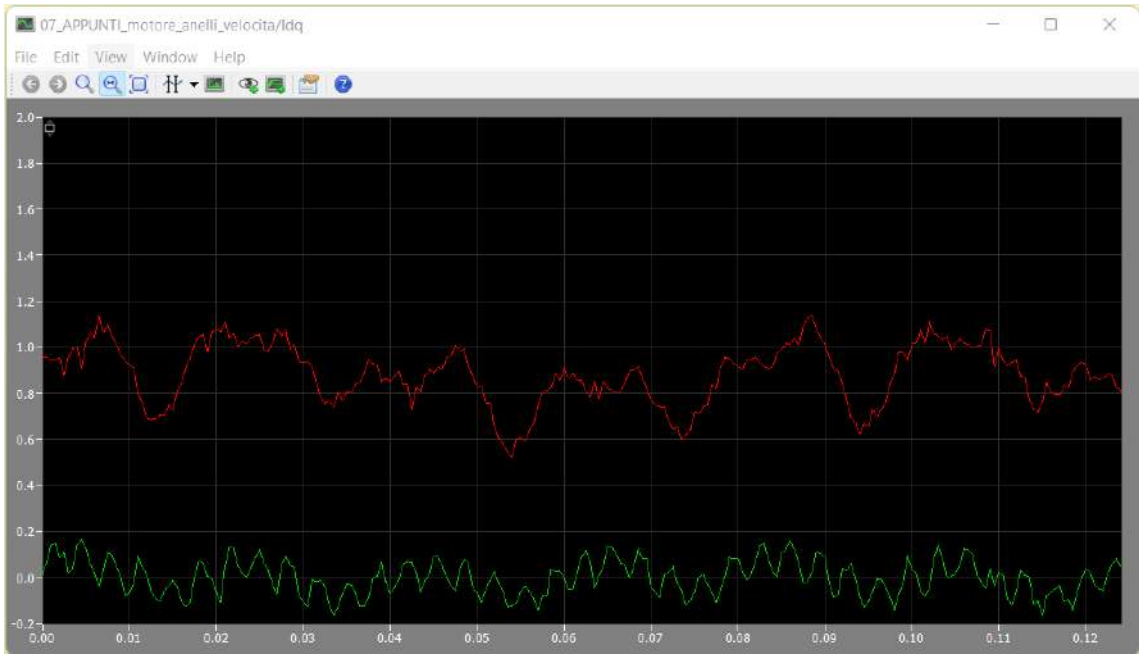


Figure 4.50: I_d and I_q scope on load conditions

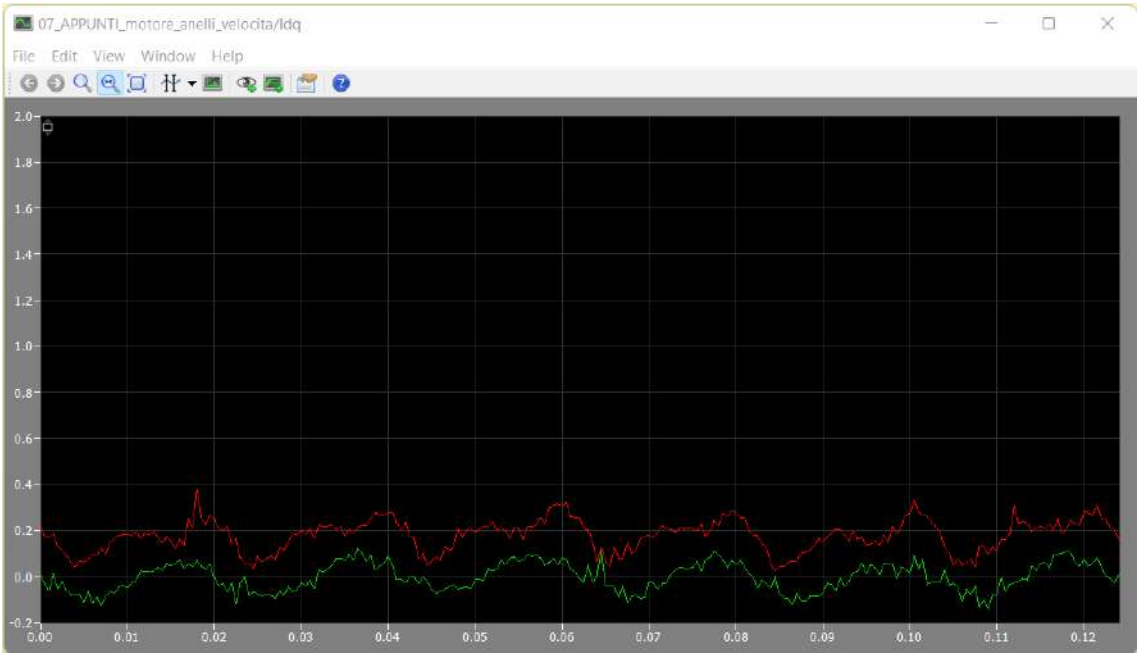


Figure 4.51: I_d and I_q scope with no torque applied to the motor

Once the speed loop is working it is possible to compare the step response simulated with the actual step response. To do so the motor is made to turn at a fixed velocity for example 2000 rpm. Then a small step of 50 rpm is imposed as a reference in order not to saturate the current PI outputs. The waveform is detected by a PLECS scope and then it is exported to MATLAB. In MATLAB is possible to compare the two waveforms.

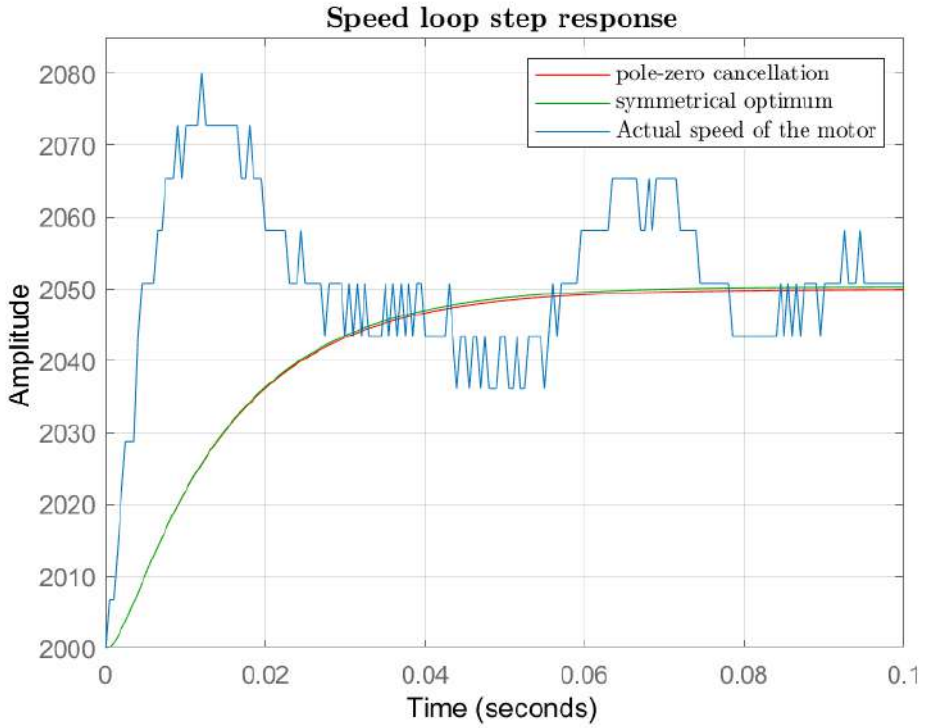


Figure 4.52: Different between simulation and actual speed loop step response

In Fig. (4.6.3) is noticeable that the two waveforms are different. This means that the behavior of the mechanical part of the motor is not representable as a first-order system. This result was already funded in section four when the viscous term was computed.
A more accurate study of the behavior of the motor has to be done in order to design the gain of the PI properly. This effect is more evident in small motors than the huge ones where the mechanical behavior fulfill the first-order dynamic. In this case, the dynamic of the system does not match the dynamic simulated but the error at steady state is zero. Furthermore, the speed loop works when a load is applied to the motor maintaining the reference speed.

Chapter 5

Conclusion

This study had the aim to discover PLECS as software for the electric drive for small power motors. The first part was focused on the study of fundamental blocks that the software provides the users to communicate with external MCU. This part of the study helps the reader to become familiar with PLECS and the way the software works. It is possible to see that the graphical interface of PLECS is very similar to Simulink with the same way to represent the blocks. Since the first example, it became clear that this software builds the program faster than Simulink and for this reason, it is preferable.
In the second part of the study, an SPM motor was driven from software. In this part, PLECS was used for power electronics. It turns out that it is very suitable for this study since it is able to connect power blocks with control algorithms.
The speed of computation of the software reduces the amount of time that the samples are achievable in the scope.
All blocks are thought for power control. They are all less customizable than Simulink's. PLECS saves every file both as PLECS code and a CCS project so it is possible to check how the code works by looking at the CCS project. This way of work is more complicated than Simulink.
A huge problem concerns the alignment procedure. The fact that the encoder block couldn't be reset from the software makes this procedure complicated and inaccurate. The output signal of the block, in fact, fakes the output of the encoder but doesn't actually reset its counter.
A very useful tool that PLECS provides is the user forum. This page is available on the PLECS website and allows Users to share problems with each other. This forum is very popular and if someone has a problem in a few days a person will resolve it. Even if no one answers a question asked by a user a PLECS engineer will provide a suitable answer.
This software is relatively new and it is constantly under development so every problem shared with the community is taken into account by PLECS engineers in order to make the software better.
Throughout the study, there were a lot of problems that have been solved step by step. Some of them were more difficult and others not but in the end, the aim of this paper has been fulfilled correctly.
It has been demonstrated that PLECS is an optimum software for electric drives and it is suitable for the simulation of power electronics too.

Chapter 6

Appendixes

6.1 Appendix 1

The following link sends the user to the PLECS download page.

`https://www.plexim.com/download`

There is a link: 'PLECS standalone package'. By clicking it, it is possible to download the version more suitable for the operating system. (Every software is designed for a different platform) Furthermore, on the download page, the 'TI C2000 Support Package' has to be downloaded too which permits the user to connect the microcontroller with PLECS and gives the blocks which are suitable for the microcontroller.
The installation program has to be run. There will be instructions to complete the installation process that have to be followed. After that, with the university credentials and the code provided by a supervisor(in this study the credentials are provided by the professor), it is possible to take an annual license.
To download the TI C2000 Support Package there are two methods. The first one is to download the installation application "$ti_c2000_1.5.5.exe$" which once run automatically gives the masks package. This method is supported by Windows but not by other platforms. Therefore, the second method is supported by every platform and consists of downloading a zip file("$ti_c2000_1.5.4.zip$"). Then it is possible to manually install the TI C2000 Support Package. This method is well described by the following video:

`https://www.plexim.com/support/videos/intro_to_ti_c2000_tsp`

Once both the TI C2000 Support Package and the PLECS standalone package are downloaded program is available. To be sure that the procedure has been done correctly on the source page of the program there will be the written:

PLECS RT BOX
TI C2000 TARGET

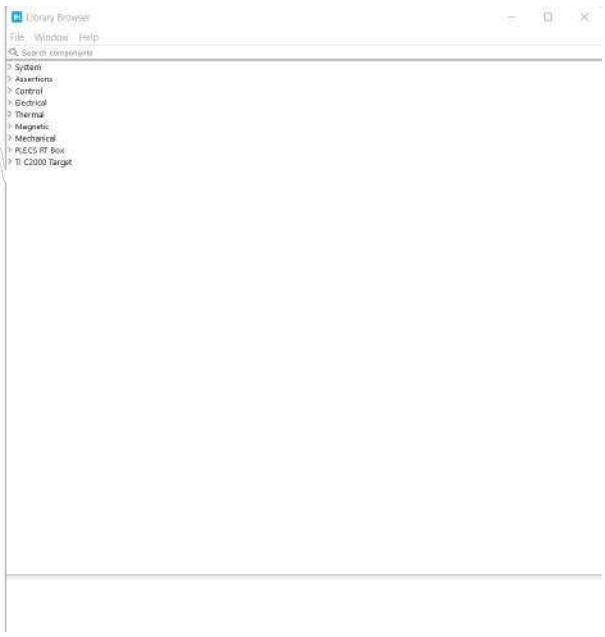


Figure 6.1: Source page

6.2 Appendix 2

6.2.1 Clarke transformation

Clarke transformation transforms a three-phase system of quantities into another three-phase system. The Clarke transformation in matrix form is defined as:

$$x_{\alpha\beta\gamma} = Cx_{abc} \tag{6.1}$$

where:

$$x_{\alpha\beta\gamma} = \begin{bmatrix} x_\alpha \\ x_\beta \\ x_\gamma \end{bmatrix} \tag{6.2}$$

$$C = k_1 \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ k_2 & k_2 & k_2 \end{bmatrix} \tag{6.3}$$

where:

$$\begin{aligned} x_\alpha &= k_1(1x_a - \frac{1}{2}x_b - \frac{1}{2}x_c) \\ x_\beta &= k_1(\frac{\sqrt{3}}{2}x_b - \frac{\sqrt{3}}{2}x_c) \\ x_\gamma &= k_1k_2(x_a + x_b + x_c) \end{aligned} \tag{6.4}$$

k_1 and k_2 are two constant coefficients. All the coefficients of the Clarke matrix are constant,i.e. they do not depend on time. If the three-phase system x_{abc} has no bias component it can be proven that $x_\gamma = 0$.

$$x_a + x_b + x_c = 0$$

$$x_\gamma = k_1k_2(x_a + x_b + x_c) = 0$$

And so the Clarke matrix is made up of two terms. This means that x_α and x_β don't have bias component whatever are x_a,x_b and x_c as well.
Let's consider a three-phase system with no bias component.
In this case, it is possible to represent a vector from a three dimensions Cartesian plane (x_{abc}) to two dimensions Cartesian plane ($x_{\alpha\beta}$).

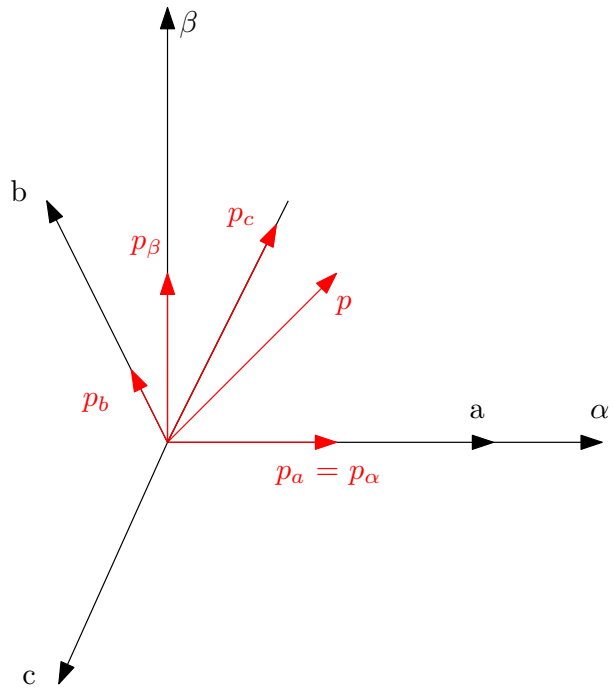


Figure 6.2: Clarke transformation

The power expressions of the two systems are:

$$p_{abc} = v_{abc}^T i_{abc} = v_a i_a + v_b i_b + v_c i_c \tag{6.5}$$

$$p_{\alpha\beta\gamma} = v_{\alpha\beta\gamma}^T i_{\alpha\beta\gamma} = v_\alpha i_\alpha + v_\beta i_\beta + v_\gamma i_\gamma \tag{6.6}$$

Constant values k_1 and k_2 can be chosen arbitrarily. There are two main ways to choose those terms. The first one is the "magnitude invariant transformation" and the second one is the "power invariant transformation".
For the magnitude invariant transformation, the constant k_1 and k_2 are:

$$k_1 = \frac{2}{3} \quad k_2 = \frac{1}{2}$$

and so:

$$C = \frac{2}{3}\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \quad C^{-1} = \begin{bmatrix} 1 & 0 & 1 \\ -\frac{1}{2} 0 & \frac{\sqrt{3}}{2} & 1 \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} & 1 \end{bmatrix}$$

The power expression in this case becomes:

$$p_{abc} = \frac{3}{2}(v_\alpha i_{\alpha v_\beta i_\beta}) + 3 v_\gamma i_\gamma \tag{6.7}$$

For a three-system with no neutral connection, it is:

$$p_{abc} = \frac{3}{2}(v_\alpha i_{\alpha v_\beta i_\beta}) \tag{6.8}$$

If we consider a sinusoidal three system:

$$\begin{aligned} x_a &= \sqrt{2}Xcos(\Omega t) \\ x_b &= \sqrt{2}Xcos(\Omega t - \frac{2\pi}{3}) \\ x_c &= \sqrt{2}Xcos(\Omega t - \frac{4\pi}{3}) \end{aligned} \tag{6.9}$$

The transformed quantities become:

$$x_\alpha = \sqrt{2}Xcos(\Omega t)$$

$$x_\beta = \sqrt{2}Xsin(\Omega t)$$

It is possible to see that the power expression has changed but the magnitude of the sine waves doesn't. That's why this transformation is called invariant to the magnitude.
If we choose k_1 and k_2 equal to:

$$k_1 = \sqrt{\frac{2}{3}} \quad k_2 = \frac{1}{\sqrt{2}}$$

The Clarke matrix becomes:

$$C = \sqrt{\frac{3}{2}} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \quad C^{-1} = C^T = \begin{bmatrix} 1 & 0 & \frac{1}{\sqrt{2}} \\ -\frac{1}{2}0 & \frac{\sqrt{3}}{2} & \frac{1}{\sqrt{2}} \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

It is possible to see that $C^{-1} = C^T$.
So:

$$p_{abc} = v^T_{\alpha\beta\gamma}(C^{-1})^T C^{-1})i_{\alpha\beta\gamma} = v^T_{\alpha\beta\gamma}i_{\alpha\beta\gamma} = p_{\alpha\beta\gamma}$$

Let's consider the three-phase system shown in Eq. (6.9). This time the transformation becomes:

$$x_\alpha = \sqrt{3}Xcos(\Omega t)$$

$$x_\beta = \sqrt{3}Xsin(\Omega t)$$

It is noticeable that this time the magnitude of the sine waves has been changed but the power in the transformed system is unchanged. This is why this transformation is called "invariant to the power".

6.2.2 Park transformation

It is possible to represent the Clarke transformation by space phasors. If the plane $\alpha\beta$ is interpreted as the complex plane ($\alpha \to Re$,$\beta \to Im$) the vector $\vec{p}$ showed in figure 5.1 can be described by means of the complex quantity $p^-_{\alpha\beta}$.

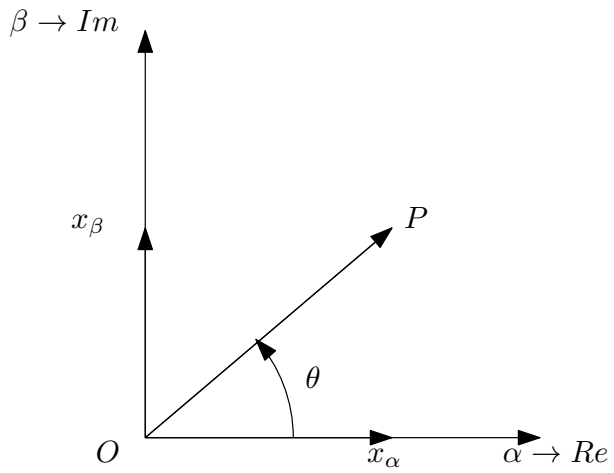


Figure 6.3: Clarke transformation in complex plane

For a general complex quantity:

$$\bar{x}_{\alpha\beta} = x_\alpha + jx_\beta \tag{6.10}$$

$$x_\alpha = Re(x_{\alpha\beta}) \tag{6.11}$$

$$x_\beta = Im(x_{\alpha\beta}) \tag{6.12}$$

In polar form, the expression of a space phasor is:

$$\bar{x}_{\alpha\beta} = x_{\alpha\beta} e^{j\theta_{\alpha\beta}} \tag{6.13}$$

Where $x_{\alpha\beta}$ is the magnitude and $\theta_{\alpha\beta}$ is the argument of the space phasor.
It is noticeable that the quantities made by the Clarke transformation produce quantities that are changeable with the angle θ as well.
This means that if there are three sine waves the Clarke transformation produces two sine waves.
Park transformation instead, transform the quantities in a reference system which rotates with an angle *theta* too.
Park transformation transforms a three-phase system of quantities in another three-phase system. Starting from a system of equations expressed in $\alpha\beta\gamma$ it is possible to find the quantities in $dq0$.

$$x_{dq0} = P x_{\alpha\beta\gamma} \tag{6.14}$$

Where:

$$x_{dq0} = \begin{bmatrix} x_d \\ x_q \\ x_0 \end{bmatrix} \quad P = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.15}$$

And so the quantities in dq0 are:

$$\begin{aligned} x_d &= x_\alpha cos(\theta_p) + x_\beta sin(\theta_p) \\ x_q &= -x_\alpha sin(\theta_p) + x_\beta cos(\theta_p) \\ x0 &= x_\gamma \end{aligned} \tag{6.16}$$

Also in the dq0 reference is possible to represent quantities that don't have homopolar components as space phasors:

$$\bar{x}_{dq} = x_{dq} e^{j\theta_p} \tag{6.17}$$

$$\bar{x_{dq}} = x_d + jx_q \tag{6.18}$$

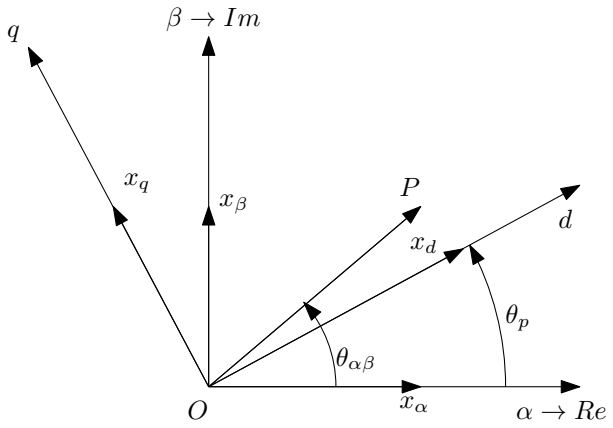


Figure 6.4: Park transformation in complex plane

By substituting the expression of x_d and x_q as a function of x_α and x_β it is:

$$\begin{aligned}\bar{x}_{dq} &= x_d + jx_q = x_\alpha cos(\theta_p) + x_\beta sin(\theta_p) + j(-x_\alpha sin(\theta_p) + x_\beta cos(\theta_p)) = \\ &= x_\alpha(cos(\theta_p) - jsin(\theta_p)) + jx_\beta(cos(\theta_p) - jsin(\theta_p)) = (x_\alpha + jx_\beta)(cos(\theta_p) - jsin(\theta_p))) = \\ &= \bar{x}_{\alpha\beta}e^{-j\theta_p} = x_{\alpha\beta}e^{\theta_{\alpha\beta}-\theta_p}\end{aligned} \tag{6.19}$$

For example the two terms :

$$x_\alpha = \sqrt{3}Xcos(\Omega t + \theta_0)$$
$$x_\beta = \sqrt{3}Xsin(\Omega t + \theta_0)$$

in dq by choosing:

$$\theta_p = \Omega t$$

The Park transformation of the space phasor on the α, β plane becomes:

$$\bar{x}_{dq} = \sqrt{3}Xe^{j\theta_0}$$

This equation shows that the space phasor in the d,q plane is stationary. This is because the dq reference system is rotating at the same speed of the $\alpha\beta$ phasor.

6.3 PBC Datasheet

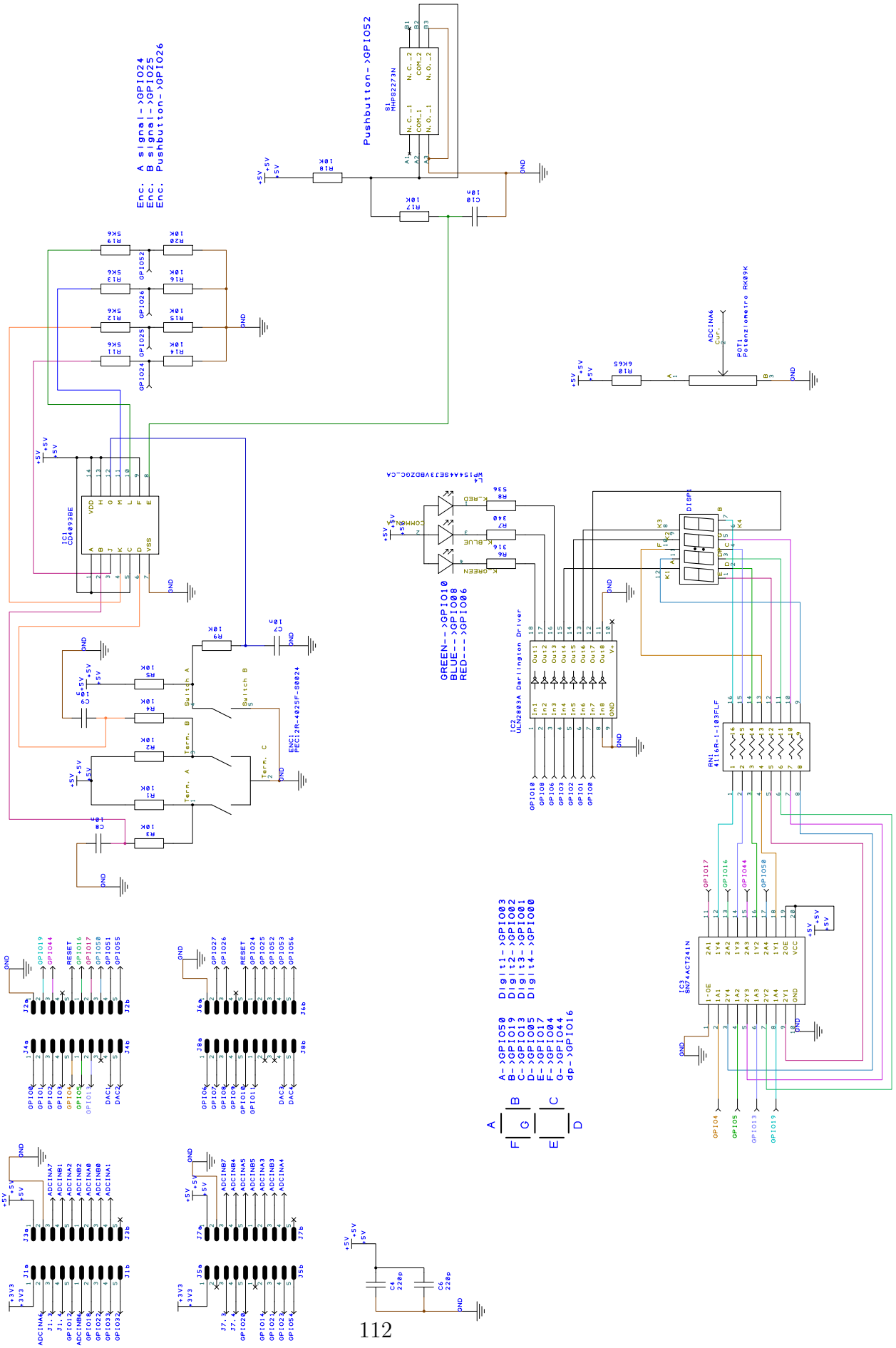


Figure 6.5: PCB peripherals

Bibliography

[1] Ogata, Katsuhiko (1967), *State space analysis of control systems*, Prentice-Hall.

[2] *TMS320x2806xTechnical Reference Manual*,(https://www.ti.com/lit/ug/spruh18i/spruh18i.pdf?ts=1669366677479)

[3] *LAUNCHXL-F28069M Overview*,https://www.ti.com/lit/ug/sprui11b/sprui11b.pdf?ts=1599060816210

[4] Plexim GmbH, *PLECS User Manual*.

[5] Plexim GmbH, *I C2000 Target Support User Manual Version 1.5*

[6] *BOOSTXL-DRV8305EVM User's Guide*,(https://www.ti.com/lit/ug/slvuai8a/slvuai8a.pdf?ts=1669367062319&ref_url=https%253A%252F%252Fwww.google.com%252F)

[7] S.Bolognani, *Azionamenti Elettrici parte tre*

[8] J. H. Allmeling and W. P. Hammer, *PLECS-piece-wise linear electrical circuit simulation for Simulink*. Proceedings of the IEEE 1999 International Conference on Power Electronics and Drive Systems. PEDS'99 (Cat. No.99TH8475), 1999, pp. 355-360 vol.1, doi: 10.1109/PEDS.1999.794588.