POLITECNICO DI TORINO

Master's Degree Course in Electrical Engineering

Master's Degree Thesis

Model-Based Design for STM32 Nucleo boards applied to motion control

Supervisor:
prof. Gian-Mario Pellegrino

Candidate:
Cesar Olivera
Student Id: 238578

Academic Year 2017-2018

Abstract

In the field of Power Electronics, several types of motor control systems have been developed using microprocessors. Many of them are based on systems of differential equations, use state observers and need to handle multiple tasks in order to execute an optimal operation of the motor. The more complex is the system, the harder is to implement. Traditionally, many steps are carried out from the modelling stage to the real operation of the control systems, and require iterative tests to ensure the absence of error. A major part of the effort is focused on embedded software representation of the control system, which demands a large amount of code lines that may contain hard-to-detect bugs.

The Model-Based Design is a method in which most of the design is centred in the model, while the code representation and embedded integration is automatically carried out. In this work, simplified implementation and validation methods are proposed using Model-Based Design for motor control systems, exemplified with an I-Hz control. Taking advantage of the Matlab/Simulink graphical tools, a control system can be modelled as a combination of block diagrams and state machines to reduce the mathematical complexity and schedule multiple tasks. Also, the STM32 Embedded Target toolbox is used for automating the generation and integration of the modelled control algorithm to an embedded software representation compatible with the STM32 Nucleo boards, avoiding to write any code manually and carry errors.

Validation of the designed control system is eased when exactly the same simulated control algorithm is integrated to a microprocessor because the names of the variables are shared. The variables involved in the operation of the control system can be read from the microprocessor's memory using a serial monitor with USART communication protocol. Minimal differences took place when the real and simulated results were confronted due to some nonidealities, but under certain tolerance ranges that allowed the model to be endorsed.

In conclusion, the proposed method for designing motor control systems allowed to skip or to simplify many steps that are traditionally carried out, specially what regards to the mathematical modelling and code programming, so time and effort reduction was successfully accomplished.

To the Olivera de la Calzada family
around the world...

Acknowledgements

Un aroma a incienso, mi tacita de café y una lluvia primaveral por mi ventana me acompañan este día en Torino... Aquí me encuentro, tan lejos y tan cerca de todo, buscando las palabras para explicar cómo es que llegué a este punto. No sé si es la meta o un punto de partida, o si tan solo es una de las cosas que ocurren entre medio. Lo que sí tengo certeza es de que esto no lo siento normal, un poco atormentado por el cambio que llegará a suceder. ¡Un momento! ¿Esto no lo había vivido antes?

Me parece que fue a los 9 años cuando me despedí de mi ciudad natal, Lima, para ir a Santiago con mis padres y hermanos. Tal vez me enojé mucho con mis padres por este gran cambio que me atormentó desde temprana edad. Sin embargo, ellos siempre estuvieron firmes con la idea de que estaban haciendo lo mejor para sus hijos, y efectivamente así fue. Nunca fueron padres ausentes, siempre nos entregaron motivación y apoyo en nuestro crecimiento como personas. Recuerdo que yo quería hacer de todo... ser músico, deportista y buen estudiante, y allí ellos estaban... en mis presentaciones tocando la guitarra, en mis torneos de basketball y en mi graduación del colegio.

Creo que me desvié del tema... ¿Cómo es que llegué a este punto?... ¡Ah sí! Fue un tema de querer poner mis propias decisiones en práctica con respecto a lo que sería mi futuro. Recuerdo momentos sentado en el parque con mi guitarra, aprovechando el silencio del barrio para pensar qué haría. Mis abuelos desde Perú me decían que tome la decisión que yo quisiera, pero que siempre trate de ser el mejor en ello, y así fue... Decidí irme a estudiar afuera, bien lejos, y empezar una nueva vida armada por mí mismo... Ok, eso no fue exactamente lo que ocurrió.

Entré a estudiar ingeniería eléctrica en la UC, para nada arrepentido, incluso llegué a conocer amistades increíbles, profesores muy buenos, y por su puesto... grandes experiencias. El deporte y la música nunca los dejé de lado por los estudios, a ello se sumaron tantos viajes y fiestas. Pero el foco nunca lo perdí, poco más de cuatro años pasaron cuando logré continuar mis estudios y experiencias en Italia.

Fueron muchos amigos de los que me tuve que despedir, con los que compartí tiempos de estudio, fiestas, paseos en bicicleta, subidas a las montañas, partidos de basket, presentaciones musicales, viajes a la playa o al sur... Es realmente difícil despedirse, agradezco todo lo que he vivido con ellos y les deseo lo mejor también en sus vidas. ¿Despedida? Cavolo! esto es solo un “hasta luego”, porque les aseguro que nos volveremos a ver muchas veces más... Creo que ya solté una palabra en italiano...

Aparentemente no vine solo a Torino, fueron muchos otros compañeros de la UC que tomaron la misma decisión, y ahora también son queridos amigos míos. En general, me gusta referirme al grupo como la familia Baretti, porque es la dirección de la casa que nos acoge... No, no... No todos vivimos en esta casa, pero sí compartimos acá gran parte del tiempo, nos reunimos a conversar, hacer ejercicios, comer, tirar la casa por la ventana... Realmente los he considerado una familia para mí, no sé que sería sin ellos en este momento. Todos nos apoyamos, nos felicitamos, nos escuchamos en los momentos difíciles, nos aconsejamos, viajamos juntos por Europa... Gracias a todos por estar conmigo, y hacer de esta experiencia “La Dolce Vita”. Y cómo no mencionar a los amigos del viejo

continente que he hecho este último par de años. También los considero parte de la familia Baretti. Tanto compartimos todos juntos que veces no nos damos cuenta si estamos hablando en español o italiano. Grazie mille per tutto, ragazzi!

Ya se ha consumido el incienso, me terminé el café, y ahora hay truenos acompañando las fuertes lluvias primaverales. Este agitado día está llegando a su fin y se siente como si hubiesen sido seis años... ¡Efectivamente! Fueron seis años muy agitados que me han marcado como persona, dejándome ante un futuro incierto pero emocionante. Finalmente, para cerrar esta dedicatoria, quisiera reiterar mi agradecimiento infinito a mis padres y hermanos, que dentro de poco atravesarán medio mundo para venir a verme y celebrar este valioso hito.

¡Gracias Totales!

Contents

List of Figures

List of Tables

1 Introduction

An AC electric motor is a machine that carries out electromechanical conversion. Being supplied by alternate current the machine is capable to apply a certain power and speed on the rotor, depending on its mechanical load. Control systems can be implemented to control the states of the motor such as the applied torque, rotor speed, stator voltage and current, among others. Generally, the logic of a control system is executed in a Micro-Controller Unit (MCU) to process the motor readings and drive the inverter for supplying power to the motor, however, this logic has to be properly tested before the implementation to ensure a safe performance. There are multiple methods for correctly implementing an AC motor control system, among which the Model-Based Design will be presented in this work. The purpose of this work is to use the Model-Based Design method for motor control implementation using Matlab/Simulink and the STM32 Embedded Target toolbox. This particular toolbox incorporates full compatibility with the STM32 MCUs, so the simulation of the control system and the code generation can be achieved from the Simulink environment, targeting in this case the STM32 Nucleo boards. In this work is also explained how to deal with the debugging process of the implemented control system.

1.1 State of art

Before introducing the Model-Based Design method, common implementation techniques of control systems are going to be explained. A control system involves a plant (an inverter and an AC motor in this case) and a controller (the MCU) to control the states of the plant using a feedback loop. This is graphically represented in block diagrams and mathematically validated. As the control system becomes more complex, computational tools are required to validate the system using numerical methods and simulation. There are some platforms for graphical modelling and simulating control systems like Matlab/Simulink and PLECS. As said before, the MCU executes the control logic. This logic must come from a C file that is a transcription of the modelled controller in programming language. A handwritten transcription from the block diagram to C code is susceptible to carry a lot of errors, specially when the system is very complex. So testing processes are also required for validating the C code.

From the Simulink environment, handwritten C code can be simulated with a modelled plant to test the performance of the control logic before being integrated to the MCU. This can be done using the S-Function block[6] from the Simulink Library. An S-Function is a C file that carries the same algorithm that is going to be loaded in the MCU. S-Functions are compiled as MEX files to be called from the Simulink environment and to interact with other blocks. Once the simulation works properly, the control algorithm is integrated to the MCU using an Integrated Development Environment (IDE). This integration consists in linking the inputs and outputs of the control algorithm to the peripherals of the MCU in order to interface the real plant. The integration is manually written and requires the inclusion of the MCU's drivers in the C project. Figure 1 shows how the control algorithm is first simulated and then integrated to MCU from the C project.

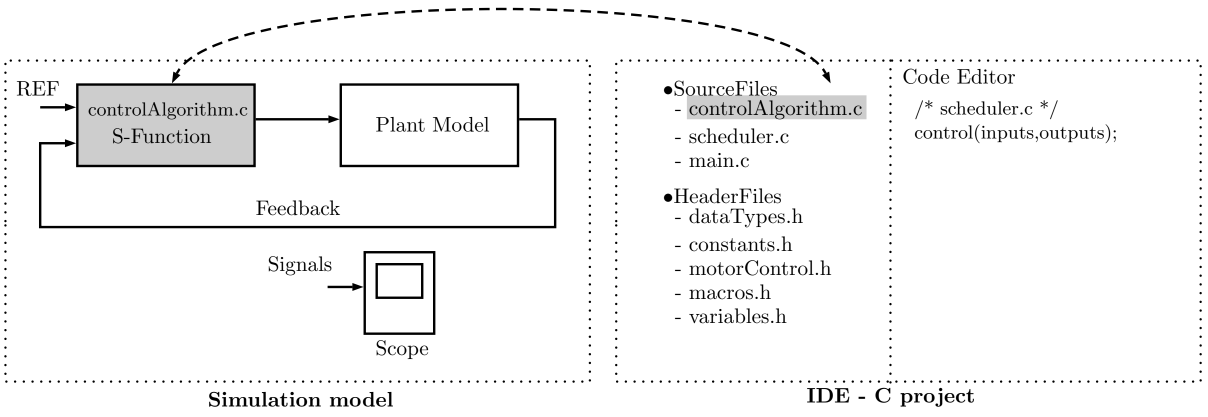


Figure 1: S-Function approach

One thing to keep in mind is that the S-Function C file is not exactly the same control algorithm file. The S-Function contains some additional functions for interfacing the Simulink inputs/outputs and for being compiled and executed in the simulation model. The code adaptation between the common C file and the S-Function can be handwritten or generated using the Legacy Code Tool[7]. The Legacy Code Tool generates the S-Function by receiving the C file and the specifications of which are the inputs/outputs and other internal parameters or data types.

After loading the control algorithm onto the MCU, a monitoring and debugging process has to be carried out in order to validate the correctness of the real execution. Debugging and measurements for the real-time execution can be done using external lab instruments (such as oscilloscope and tachometer), or establishing a communication protocol with the MCU to read the state of the internal variables from a serial terminal. Some debugging tools available in the market for real-time execution analysis are dSpace, Speedgoat, OPAL-RT and RT Box. These tools let the MCU interface to an emulated or real plant[1] for monitoring the internal values of the MCU and the states of the plant.

As competitiveness in the technology market increases, faster and more optimised designs are being pursued. Important companies like Airbus, Eurocopter, Schneider Electric and CSEE Transport are using automatic Code Generation tools for skipping the complexity of the C code handwriting and testing in order to accelerate their developments. Some of these tools are SCADE, ADI Beacon, MATRIXx and Matlab/Simulink. One of the methods that takes advantage of Code Generation is the Model-Based Design (sec. 1.2).

[1]Emulation hardware are used to emulate the plant for the Hardware-in-the-loop (HIL) test.

1.2 Model-Based Design overview

The Model-Based Design is a methodology to design software for embedded systems[8]. As its name says, the development process of a control system is focused on the model by exploiting the graphical modelling and Code Generation capabilities. While the graphical modelling reduces the complexity of the mathematical representation of the system, the Code Generation avoids the transcription process to programming language that might carry errors hard to detect. In this sense, the designing problem is simplified to the creation of the graphical model and its validation with the proper tests and simulations, while the rest of the process is practically automated.

The C code handwriting, verification and tests are time consuming, specially when the target system becomes more complex. These steps can be skipped with the Code Generation in order to achieve an important time saving. Error detection and mathematical details are easier to handle from the high level perspective of the graphical modelling.

The software used in this work for modelling and simulating control systems is Matlab/Simulink, which offers graphical tools like block diagrams and state machines to reduce the complexity of the control system modelling. The block diagrams eliminate the need to formulate differential equations and the states machines are an easy way to deal with conditional statements. At this point, a properly working simulation is enough to start generating the C code and loading it onto the MCU.

The proposal of this work is to use the Matlab/Simulink tools and the STM32 Embedded Target toolbox for modelling, implementing and debugging a control system for motor control. A summary of the steps detailed in this work are explained below.

1. The control algorithm is modelled in Simulink (sec. 3.1). The model (fig. 2) receives the reference parameters and the plant readings in order to determine the input of the plant. In this work, an I-Hz control will be developed using the Model-Based Design method. The I-Hz control is executed by the control algorithm in order to regulate the stator current vector of the AC motor, based on the scheme of figure 3.

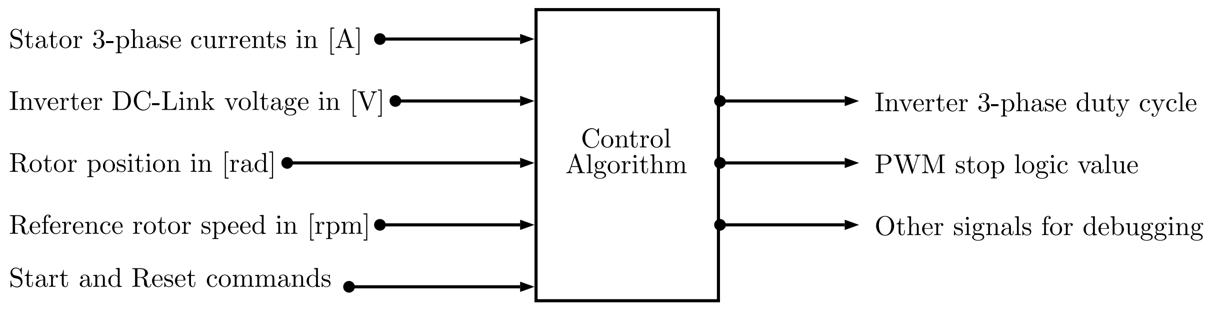

Stator 3-phase currents in [A]
Inverter DC-Link voltage in [V]
Rotor position in [rad]
Reference rotor speed in [rpm]
Start and Reset commands
Control Algorithm
Inverter 3-phase duty cycle
PWM stop logic value
Other signals for debugging

Figure 2: Inputs/Outputs of the control algorithm

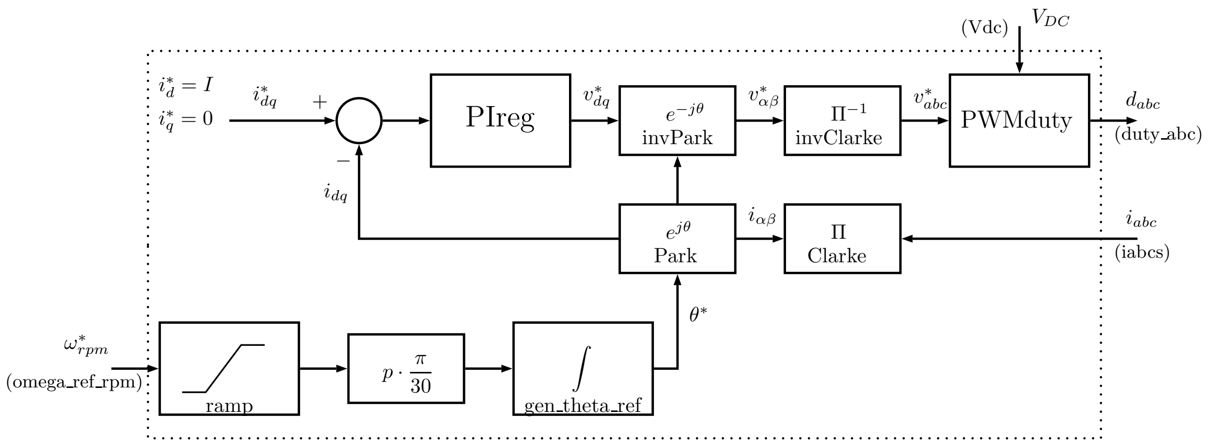


Figure 3: I-Hz control

2. Test of the control algorithm with a modelled plant in order to verify the correctness of the control logic execution (sec. 3.2).

3. Use the STM32 Embedded Target toolbox for creating a Code Generation environment in Simulink (sec 4.2). This toolbox enables Simulink blocks for representing the MCU's peripherals, so the integration between the control algorithm model and the peripherals are graphically done and it avoids the handwritten integration (see fig. 4). It may be noted that the same control algorithm model is used for both simulation and integration.

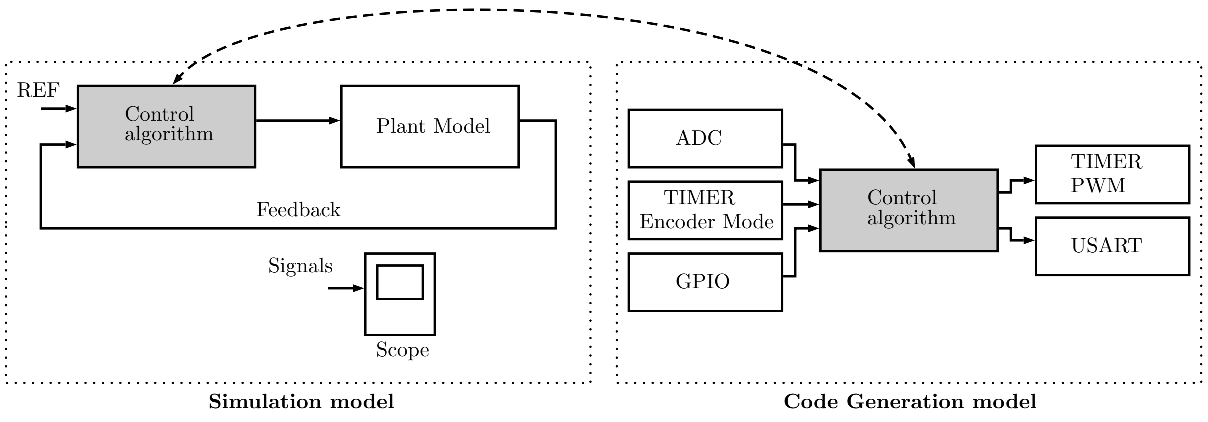


Figure 4: Model-Based Design approach

4. The Code Generation environment from the previous step will automatically generate a C project with all the control logic and the integration to the MCU, so no manual modification to the C code is required (sec. 5.1). The C project is opened from an IDE with the capability of building and loading it onto the MCU.

5. The MCU interfaces the real plant, which consists of the inverter and the AC motor (sec. 5.2). A debugging process is done using a serial monitor or virtual oscilloscope in order to visualise the relevant signals of the plant in real-time such as stator currents, voltages, duty cycles, rotor position, etc. The debugging process requires a serial communication protocol that for this case is carried out by the USART peripheral of

the MCU. In figure 5 is shown how the generated C project is loaded onto the MCU that drives the real plant and its states are being monitored using serial communication for the debug.

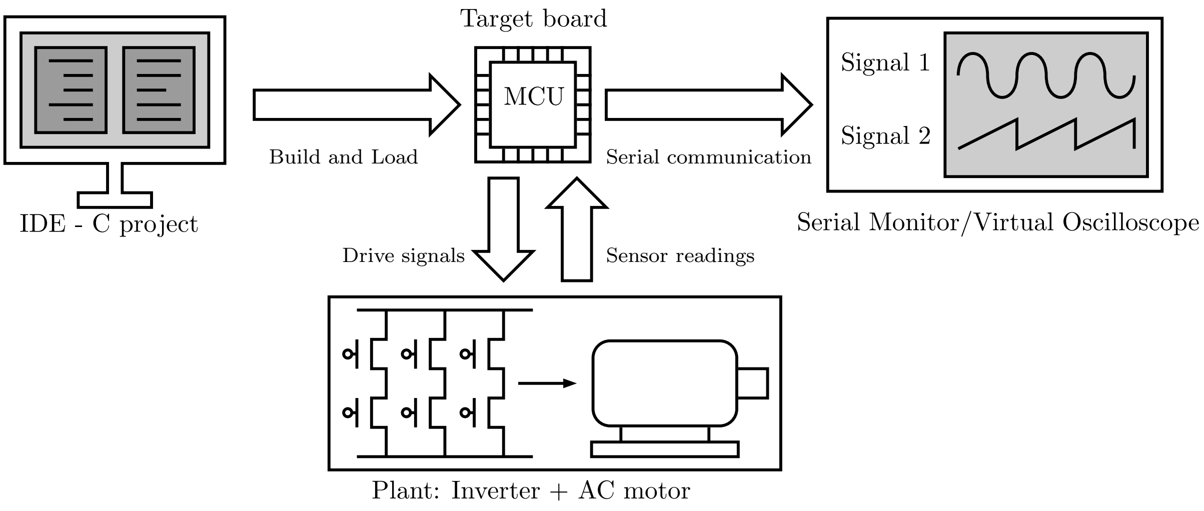


Figure 5: The operation and debugging of the real control system

2 Software and hardware requirements

This section aims to clarify which software and hardware are required in order to carry out the presented project. The set-up of the project is based on the STM32 evaluation kit, consisting of a STM32 Nucleo board and the X-NUCLEO-IHM08M low-voltage-motor driver, which are meant for educational purposes.

2.1 Software

- Matlab/Simulink from MathWorks (version R2015b or later): Environment for graphical modelling, simulating and analysing dynamical systems, based on the interconnection of high-level blocks that represents logical and mathematical functions. Some add-ons are required for this work:
  - Stateflow[9]: Provides an environment to design a state machine in Simulink.
  - Simscape[10]: Includes pre-modelled electrical systems, such as voltage sources and electrical motors.
  - Embedded Coder[12]: Tool for translating a system representation from a Simulink model into C code.
- STM32CubeMX[5] from STMicroelectronics (version 4.21.0 or later): Graphical software configuration tool that allows the generation of C initialisation code for STM32 MCUs using graphical wizards.
- STM32 Embedded Target toolbox[1] from STMicroelectronics (version 4.4.2): Is a Matlab extension that links a Simulink model to an STM32CubeMX file in order to enable the peripheral blocks of the STM32 MCUs. Its purpose is to graphically connect a modelled system to the MCU for generating a totally integrated code using Embedded Coder.
- KEIL uVision IDE[13] from KEIL (version 5 or later): Integrated Development Environment with the MDK-ARM toolchain to compile and load a C project onto the STM32 MCUs.

2.2 Hardware

- STM32F303RETx Nucleo board[2] from STMicroelectronics. The main specifications of this board are:
  - 72 [MHz] clock source
  - 4 ultra-fast 12-bit ADCs with 5 MSPS
  - 3 Timers with PWM and encoder mode
  - Full-speed USB for USART communication up to a 2,000,000 baud rate

Figure 6: STM32F3RETx Nucleo board

  Alternative boards with these minimum requirements can be used, such as the STM32F4 and STM32F7 Nucleo board series.

- X-NUCLEO-IHM08M[4] 3-phase motor driver from STMicroelectronics. The main characteristics of this device are:
  - 2-level output signal per leg
  - DC-link from 8 to 48 [VDC]
  - $I_{out}^{max} = 15$ [ARMS] in each phase
  - 3.3 [V] compatible NMOS gate driver
  - Power NMOS with $R_{DS} = 0.0014$ [Ω]
  - 3 current shunt sensor
  - DC-link voltage sensor
  - Shield compatible with STM32 Nucleo boards

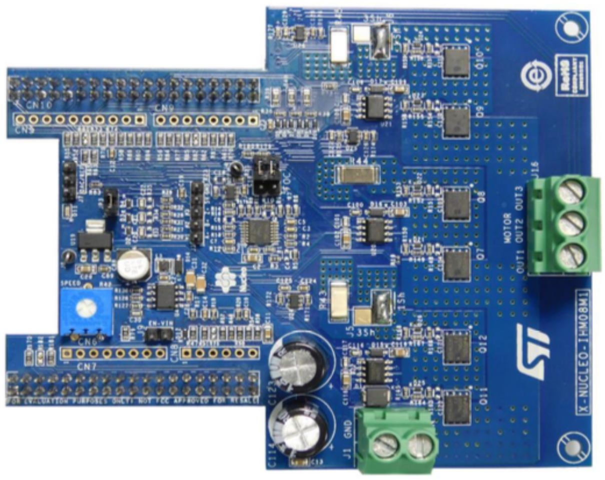


Figure 7: X-NUCLEO-IHM08M expansion board

- S140-2B353[14] Permanent Magnet Synchronous Motor with encoder from Microphase. The main ratings of this motor are:
  - $V_{\text{supply}} = 17$ [VAC]
  - $I_{\text{stall}} = 6.7$ [A]
  - $P_{\text{nom}} = 100$ [W]
  - $T_{\text{nom}} = 0.32$ [Nm]
  - $\omega_{\text{nom}}^{\text{mec}} = 3000$ [rpm]
  - $K_T = 0.05$ [Nm/A]
  - $R_{u-v} = 0.5$ [Ω]
  - $L_{u-v} = 0.53$ [mH]

Figure 8: S140-2B353 Permanent Magnet Synchronous Motor

  Alternative AC motors that can operate with less than 24 [VAC] and 100 [W] can be also used.

- DC power supply with at least 24 [V] and 3[A].

Figure 9 shows the hardware set-up based on the STM32 evaluation kit, where the inverter converts the DC power into a 3-phase sinusoidal waveform to supply the motor using a wye connection.

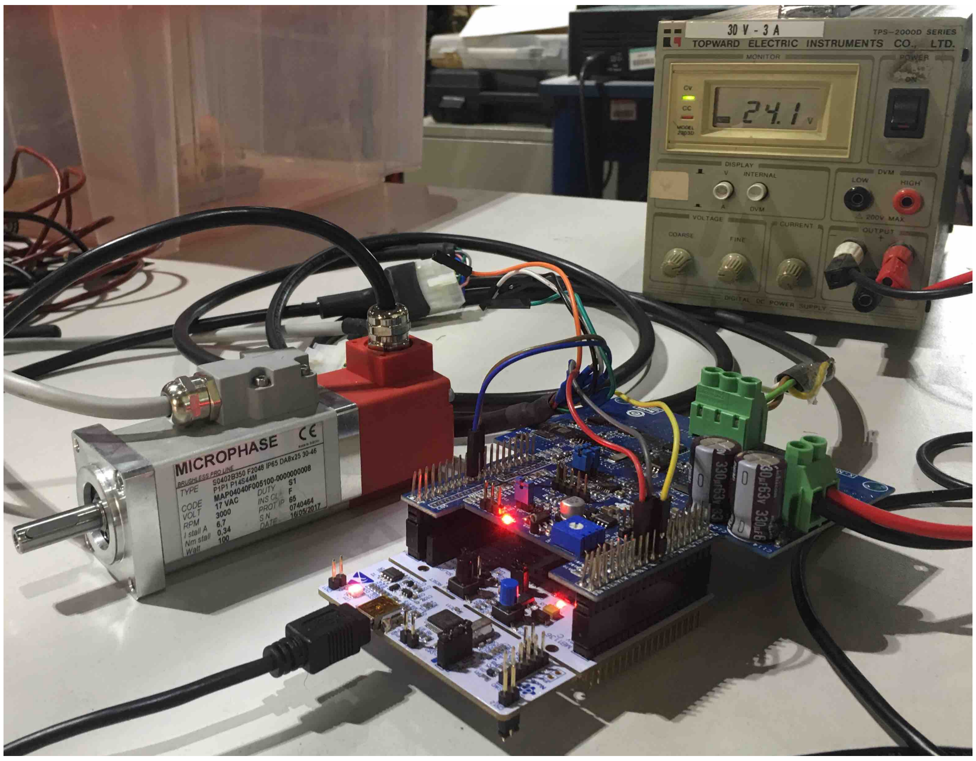


Figure 9: Hardware set-up

3 Modelling the control algorithm

3.1 The Simulink model of the control algorithm

A control algorithm is a set of instructions that are executed depending on the input readings and the current state of the internal variables, making changes on these internal variables and updating the output values. This is called once at each cycle of a clock source or timer, triggered by its rising edge.

The purpose of the control algorithm developed in this work (fig. 10) is to control the behaviour of a Permanent Magnet Synchronous Motor (PMSM) by an MCU, exemplified in this case with an I-Hz control[2] and shown in fig. 3 as a block diagram. However, the algorithm should be organised as a state machine to schedule the tasks execution properly. This can be done using the Stateflow toolbox[9] of Simulink.

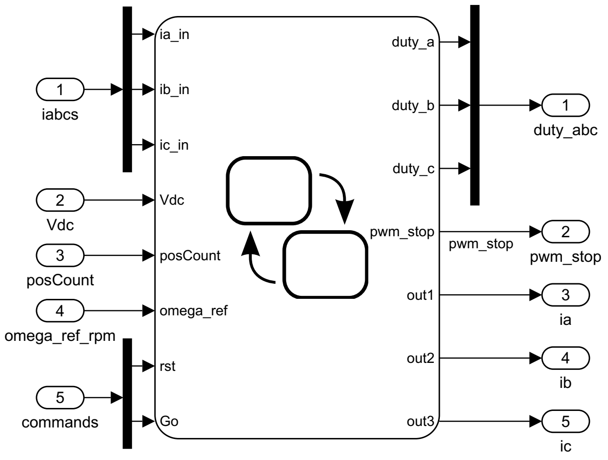


Figure 10: The control algorithm model in Simulink: *IHz_model.slx*

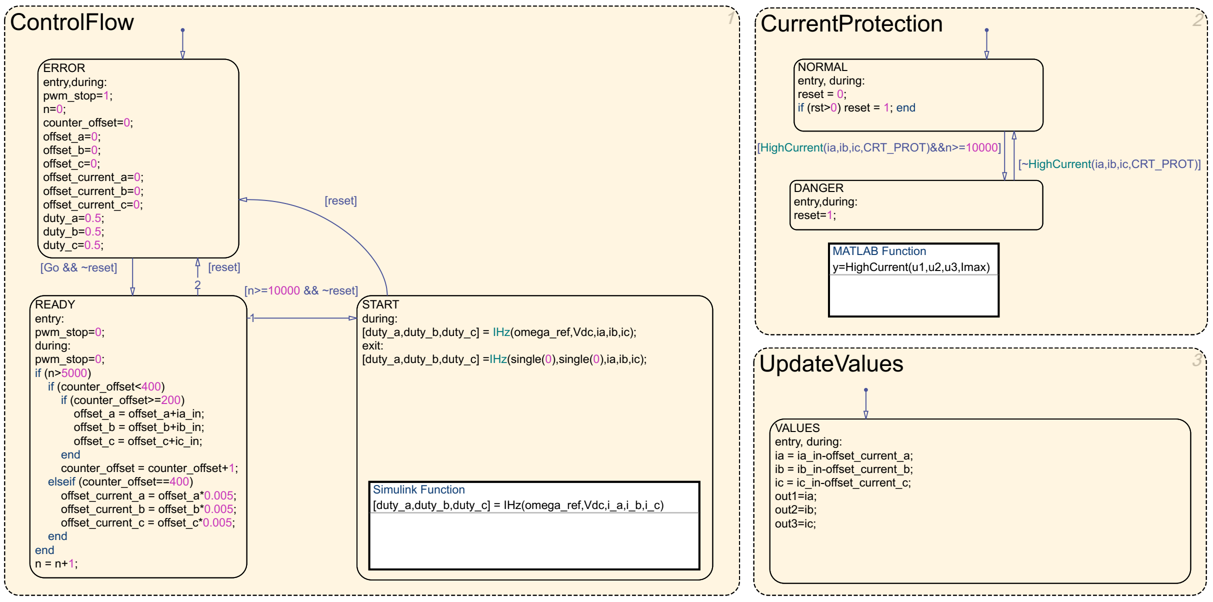
The Chart block of Stateflow (available in the Simulink Library Browser) is a kind of subsystem were states machines can be drawn. In fig. 11 are shown three parallel state machines:

1. **ControlFlow**: There are three internal states. Firstly, the ERROR state in which the internal variables are initialised and the PWM ports are set to 3-state logic (turned off). When the *Go* command goes HIGH it proceeds to the READY state where the PWM ports are enabled, the bootstrap capacitors of the inverter are charged and the offsets of the ADCs are measured. After some cycles, the START state becomes active and the I-Hz control is executed in a Simulink Function. Whenever the *reset* command is HIGH, the ERROR state becomes active and the Simulink Function variables are cleared.

2. **CurrentProtection**: This state machine checks the intensity of the currents to decide whether the operation is within the safety limits or it should be turned off.

3. **UpdateValues**: In this part the offset is subtracted from the input currents. Also, the output values of the chart are updated.

[2]The I-Hz control regulates the position and amplitude of the stator current vector in rotating axes.



```
ERROR
entry,during:
pwm_stop=1;
n=0;
counter_offset=0;
offset_a=0;
offset_b=0;
offset_c=0;
offset_current_a=0;
offset_current_b=0;
offset_current_c=0;
duty_a=0.5;
duty_b=0.5;
duty_c=0.5;
```

```
READY
entry:
pwm_stop=0;
during:
pwm_stop=0;
if (n>5000)
   if (counter_offset<400)
      if (counter_offset>=200)
         offset_a = offset_a+ia_in;
         offset_b = offset_b+ib_in;
         offset_c = offset_c+ic_in;
      end
      counter_offset = counter_offset+1;
   elseif (counter_offset==400)
      offset_current_a = offset_a*0.005;
      offset_current_b = offset_b*0.005;
      offset_current_c = offset_c*0.005;
   end
end
n = n+1;
```

```
START
during:
[duty_a,duty_b,duty_c] = IHz(omega_ref,Vdc,ia,ib,ic);
exit:
[duty_a,duty_b,duty_c] =IHz(single(0),single(0),ia,ib,ic);
```

```
Simulink Function
[duty_a,duty_b,duty_c] = IHz(omega_ref,Vdc,i_a,i_b,i_c)
```

```
NORMAL
entry, during:
reset = 0;
if (rst>0) reset = 1; end
```

```
DANGER
entry,during:
reset=1;
```

```
MATLAB Function
y=HighCurrent(u1,u2,u3,Imax)
```

```
VALUES
entry, during:
ia = ia_in-offset_current_a;
ib = ib_in-offset_current_b;
ic = ic_in-offset_current_c;
out1=ia;
out2=ib;
out3=ic;
```

Figure 11: Stateflow state machines

The Simulink Function inside the START state (fig. 12) represents the same idea of the I-Hz block diagram (fig. 3). It consists of the interconnection of subsystems (white blocks) and Matlab functions (grey blocks that contain Matlab code), whose modes of operation are explained in section 3.1.1.

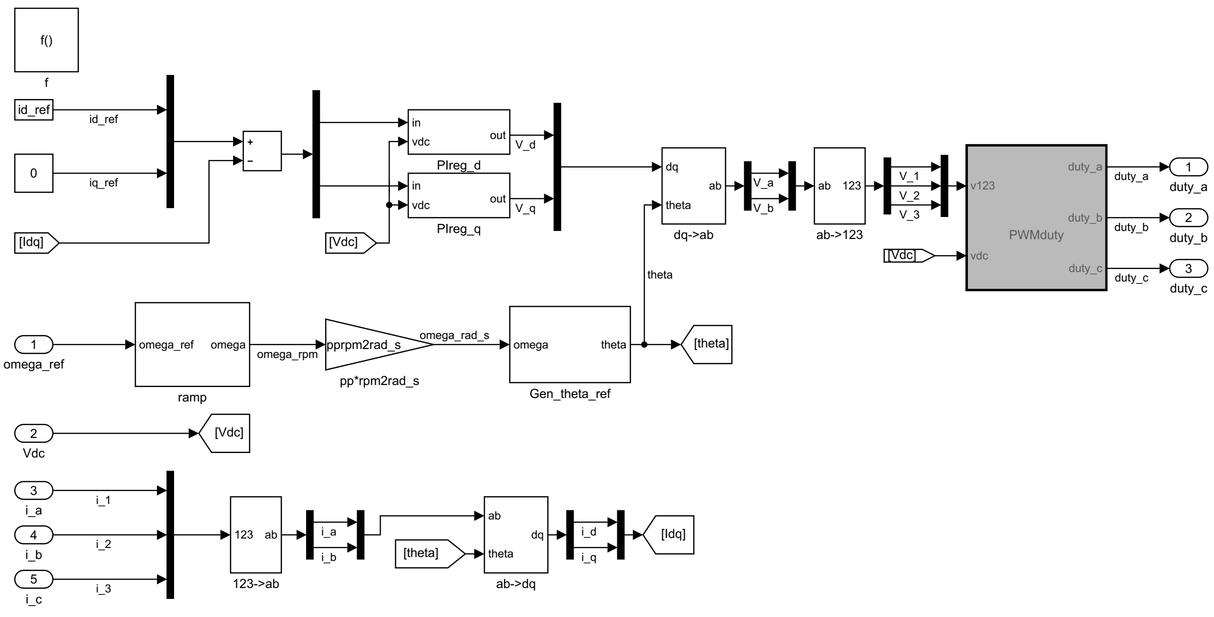


Figure 12: I-Hz Simulink Function

3.1.1 Description of the subsystems and Matlab functions

- **ramp**: Generates a ramp reference input instead of a step to avoid an abrupt response of the plant. It uses a Matlab Function inside.

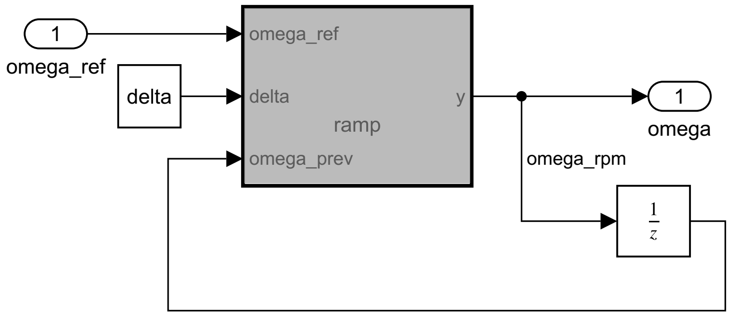


Figure 13: ramp block

```
function y = ramp(omega_ref, delta, omega_prev)
    tmp = single(0);
    tmp = omega_prev+delta;
    if (tmp>omega_ref) tmp=omega_ref; end
y = tmp;
```

- **pp*rpm2rad_s**: Gain that converts the rotor speed reference in [rpm] to an electrical speed reference in [rad/s].

- **Gen_theta_ref**: Calculates the angle from the reference speed integrating its value.

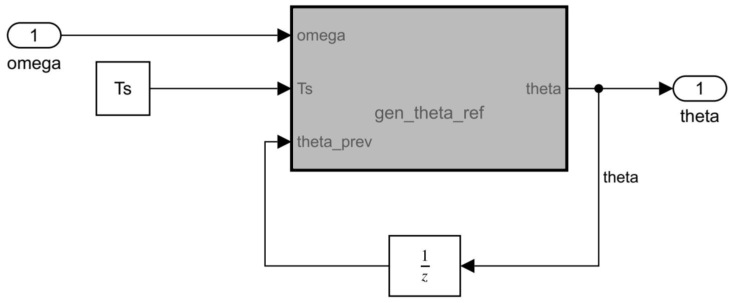


Figure 14: gen_theta_ref block

```
function theta = gen_theta_ref(omega,Ts,theta_prev)
    tmp = single(0);
    tmp = theta_prev + omega*Ts;
    if (tmp>=2*pi) tmp=tmp-2*pi; end
    if (tmp<0) tmp=tmp+2*pi; end
    if (omega==single(0)) tmp=single(0); end
theta = tmp;
```

- **PIreg_d** and **PIreg_q**: Proportional-Integral controllers for the stator currents.

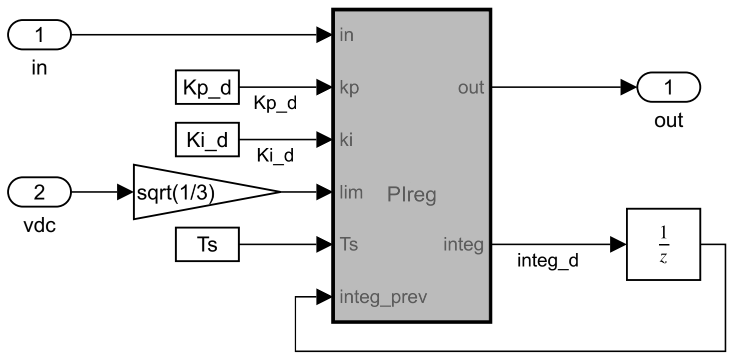


Figure 15: PIreg_d block

```
function [out,integ] = PIreg(in,kp,ki,lim,Ts,integ_prev)
    prop = single(0);
    int_lim = single(0);
    tmp = single(0);

    prop = in*kp;
    if (prop>lim) prop=lim; end
    if (prop<-lim) prop=-lim; end

    int_lim = lim-abs(prop);
    tmp = integ_prev+in*ki*Ts;
    if (tmp>int_lim) tmp=int_lim; end
    if (tmp<-int_lim) tmp=-int_lim; end

    out = prop+tmp;
integ = tmp;
```

- **123->ab**: Clarke transformation (signal representation from 3-phase to 2-phase).

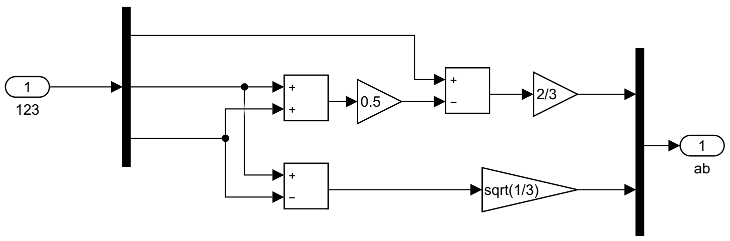


Figure 16: Clarke transformation

$$\begin{bmatrix} X_a \\ X_b \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} & -\frac{1}{3} \\ 0 & \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

- **ab->123**: Inverse Clarke transformation.

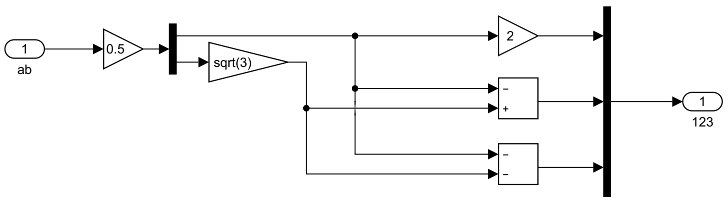


Figure 17: Inverse Clarke transformation

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} X_a \\ X_b \end{bmatrix}$$

- **ab->dq**: Park transformation (signals on rotating axes).

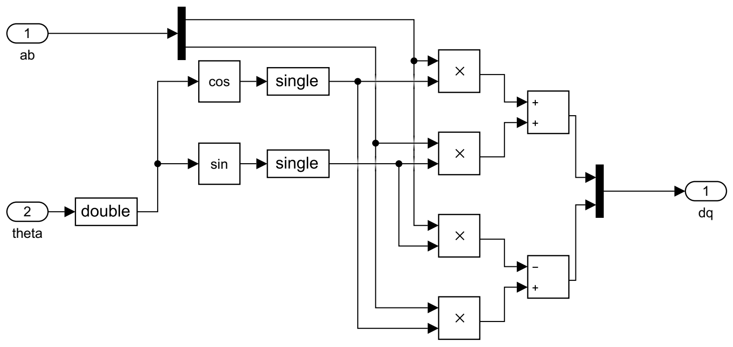


Figure 18: Park transformation

$$\begin{bmatrix} X_a \\ X_b \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} X_d \\ X_q \end{bmatrix}$$

- **dq->ab**: Inverse Park transformation.

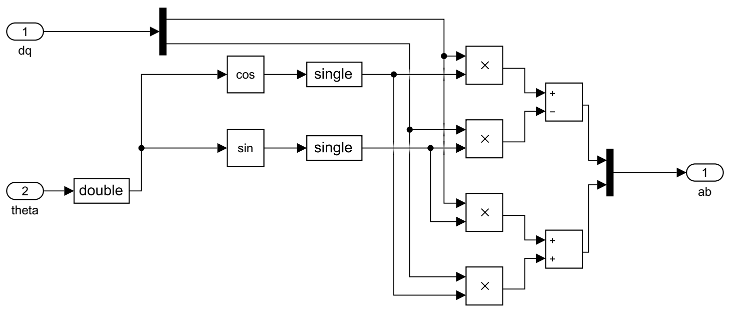


Figure 19: Inverse Park transformation

$$\begin{bmatrix} X_d \\ X_q \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} X_a \\ X_b \end{bmatrix}$$

- **PWMduty**: Generates the duty cycle for the PWM, comparing the desired voltages with the DC-Link level and subtracting the zero-sequence voltage.

```
function [duty_a,duty_b,duty_c] = PWMduty(v123,vdc)
    tmp1=single(0);
    tmp2=single(0);
    tmp3=single(0);
    vdc_inv=single(0);
    pwm_zero_seq=single(0);

    if (vdc>0) vdc_inv = 1/vdc;
    else vdc_inv=single(0);end

    if (v123(1) > v123(2))
        tmp1 = v123(1);
        tmp2 = v123(2);
    else
        tmp1 = v123(2);
        tmp2 = v123(1);
    end

    if (tmp1<v123(3)) tmp3 = tmp1;
    else
        if (tmp2>v123(3)) tmp3 = tmp2;
        else tmp3 = v123(3);end
    end

    pwm_zero_seq=tmp3*0.5;

    duty_a = 0.5 + (v123(1) + pwm_zero_seq)*vdc_inv;
    duty_b = 0.5 + (v123(2) + pwm_zero_seq)*vdc_inv;
    duty_c = 0.5 + (v123(3) + pwm_zero_seq)*vdc_inv;

    if (duty_a>1) duty_a = single(1);end
    if (duty_b>1) duty_b = single(1);end
    if (duty_c>1) duty_c = single(1);end

    if (duty_a<0) duty_a = single(0);end
    if (duty_b<0) duty_b = single(0);end
    if (duty_c<0) duty_c = single(0);end
```

3.1.2 Signal classification

The components of the control algorithm, called *IHz_model.slx* (fig. 10), have to be classified as inputs, outputs, local variables or global variables in the Model Explorer window.

- In the IHz_model of fig. 10, the inports and outports are named and set as single data type (see fig 52 of annex A1). The inports of the block are *iabcs* for the sensed stator currents, *Vdc* for the DC-Link level, *posCount* for the position counter given by the encoder, *omega_ref_rpm* for the reference angular speed in [rpm] and *commands* for the start and reset commands. The outports are *duty_abc* for the duty cycles of the 3-phase PWM inverter, *pwm_stop* for disabling the gates of the inverter, and three general outputs for monitoring other variables.

- In the Stateflow chart of fig. 11, the variables are configured as inputs, outputs or local variables with their initial conditions. All of them are treated as single data type (see fig 53 of annex A1).

- In the Simulink Function of fig. 12, the relevant signals are named and configured to *ExportedGlobal* for being treated as global variables during the Code Generation (see fig 54 of annex A1). The values of some Constant and Gain blocks are called from the Matlab workspace. One last setting has to be done for enabling the *ExportedGlobal* mode:

  ```
  > Configuration Parameters
      > Model Referencing
          > Total number of instances allowed per top model
              > one
  ```

It is important to recall that all variables and operations should work as *single* type that is equivalent to the *float32* type of the MCU.

3.2 Model-in-the-loop test

The first testing stage of the Model-based Design approach consists in running a simulation of the control algorithm and a modelled plant, this test is called Model-in-the-loop (MIL). Both the inverter and the AC motor are considered the plant of the system, where the PWM signals are the inputs and the stator currents are the outputs (see fig. 20). The rotor position given by the encoder will not be used for the I-Hz control but can be monitored.

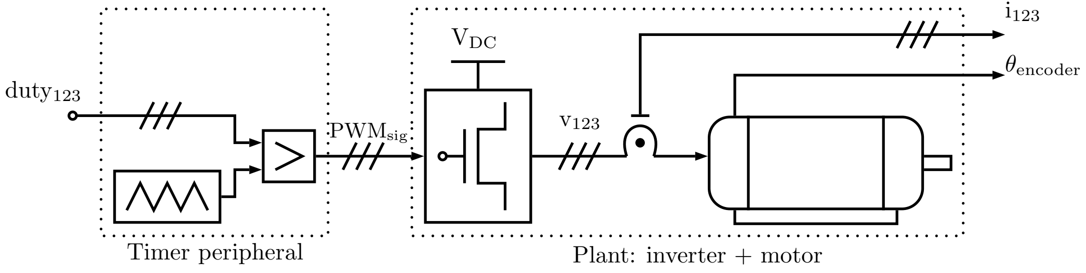


Figure 20: Plant model

In a simulation model of Simulink (see fig. 21) the inverter is represented by an amplifier of PWM signals and the AC motor is represented by a Permanent Magnet Synchronous Machine block[11] set with the manufacturer data of the real motor. The PWM signals are given by the timer peripheral of the MCU, comparing the duty cycles with an up-down triangular carrier wave at the sampling frequency of 4 [kHz]. This peripheral is depicted in the simulation by comparators and a repeating sequence wave. The inverter amplifies the signal to the levels 0-Vdc in order to supply voltage to the motor. After the plant is modelled, the control algorithm is placed on the MIL test using the Model Reference block[3] and triggered at the sampling frequency by a 4 [kHz] clock source. It is important that the sample frequency of the simulation is enough higher than the clock source to ensure an approach to a continuous behaviour of the plant.

PMSM model parameters:	Control system constants:
• 4 pole pairs in each of the 3 phases	• $\text{Kp}_d = 0.4$
• Voltage constant (K_E): 3.15 [Vrms/Krpm]	• $\text{Ki}_q = 80$
• Rotor inertia (J_R): 0.06 [kg·cm^2]	• $\text{Kp}_d = 0.4$
• Phase-Phase Winding resistance (R_{u-v}): 0.5 [Ω]	• $\text{Ki}_q = 80$
• Phase-Phase Winding inductance (L_{u-v}): 2.2 [mH]	• $V_{DC} = 24$ [V]

Table 1: Simulation parameters

[3]The Model Reference block calls an external Simulink model.

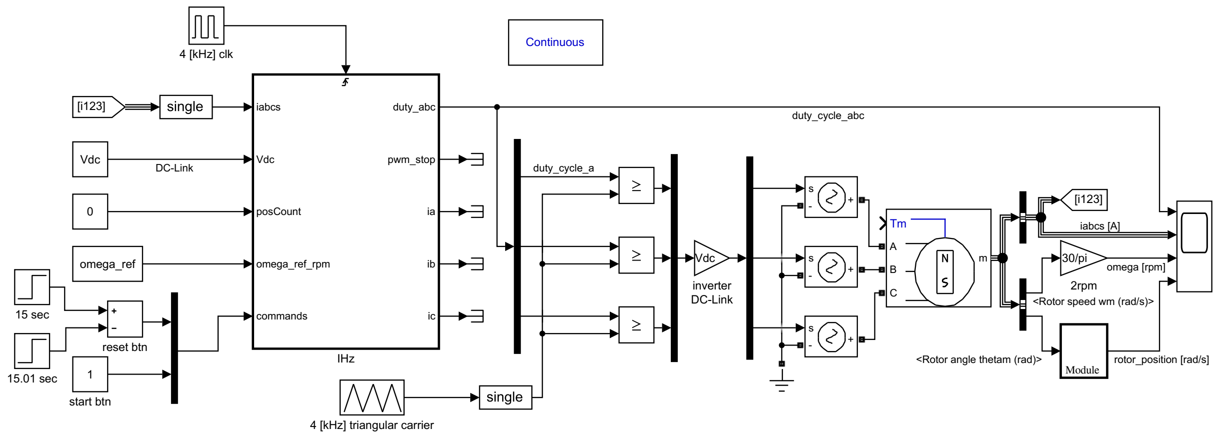


Figure 21: MIL test

Once the simulation environment is set, the relevant signals are chosen to be visualised in the scope. These are the duty cycles, stator currents, rotating speed and position of the rotor. Three simulations are run to check if the motor readings follow the set of control references. The signal visualisation is shown below for each set of control reference.

- $i_d^* = 0.8$ [A] and $\omega^*_{\text{mec}} = 400$ [rpm]
- $i_d^* = 1.0$ [A] and $\omega^*_{\text{mec}} = 500$ [rpm]
- $i_d^* = 1.2$ [A] and $\omega^*_{\text{mec}} = 600$ [rpm]

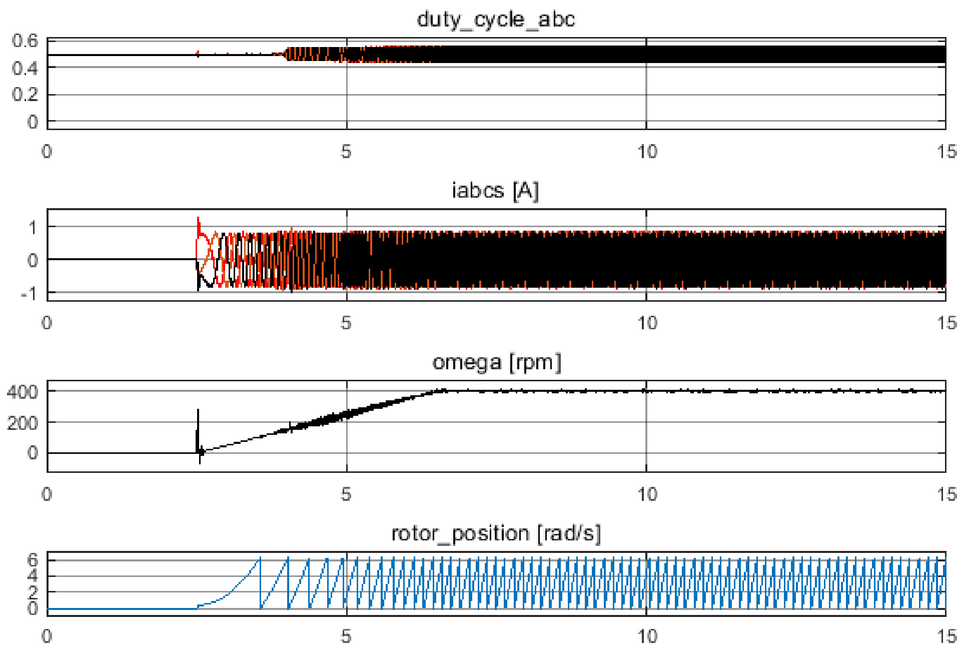


Figure 22: Signal visualisation, $i_d^* = 0.8$ [A] and $\omega^*_{\text{mec}} = 400$ [rpm]

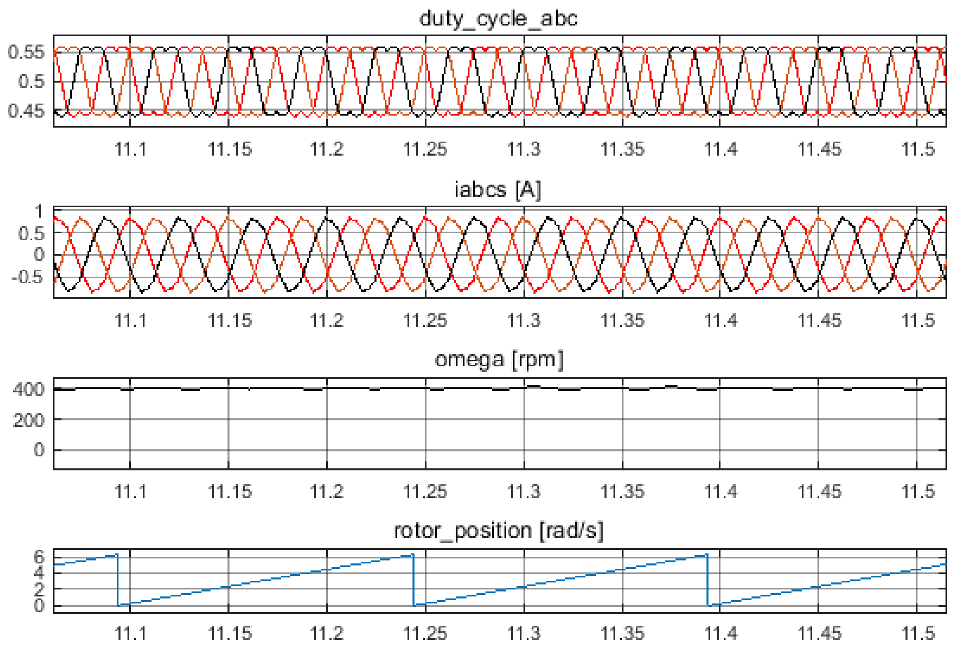


Figure 23: Signal visualisation in steady state, $i_d^* = 0.8$ [A] and $\omega_{\text{mec}}^* = 400$ [rpm]

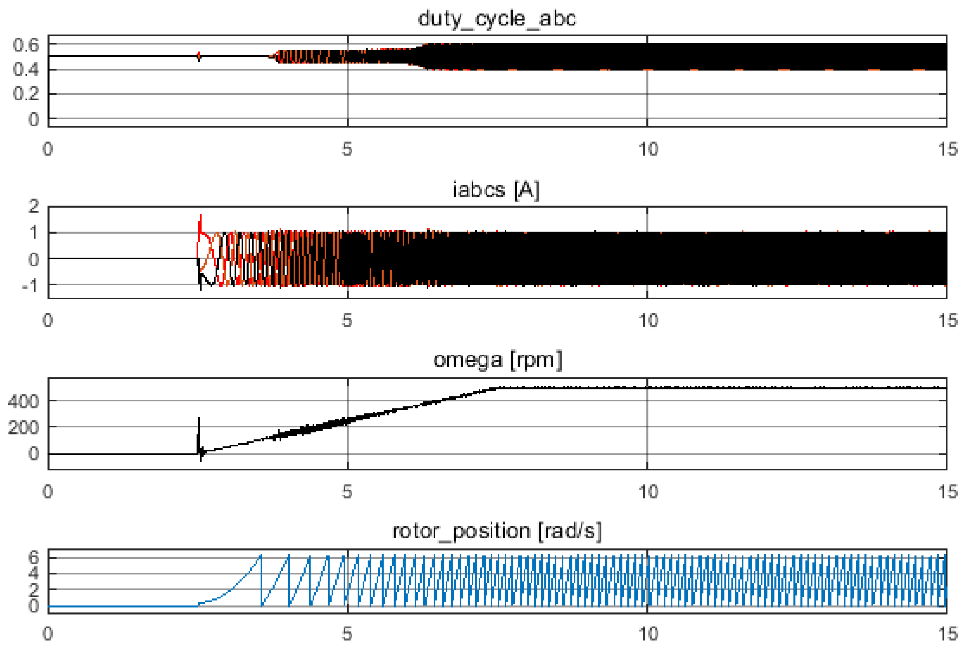


Figure 24: Signal visualisation, $i_d^* = 1.0$ [A] and $\omega_{\text{mec}}^* = 500$ [rpm]

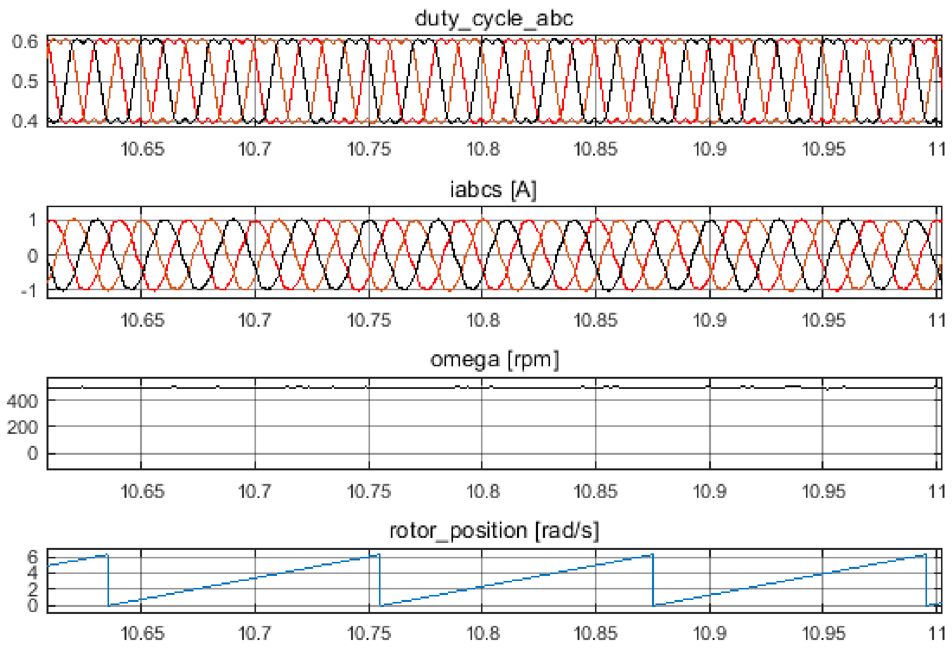


Figure 25: Signal visualisation in steady state, $i_d^* = 1.0$ [A] and $\omega_{\text{mec}}^* = 500$ [rpm]

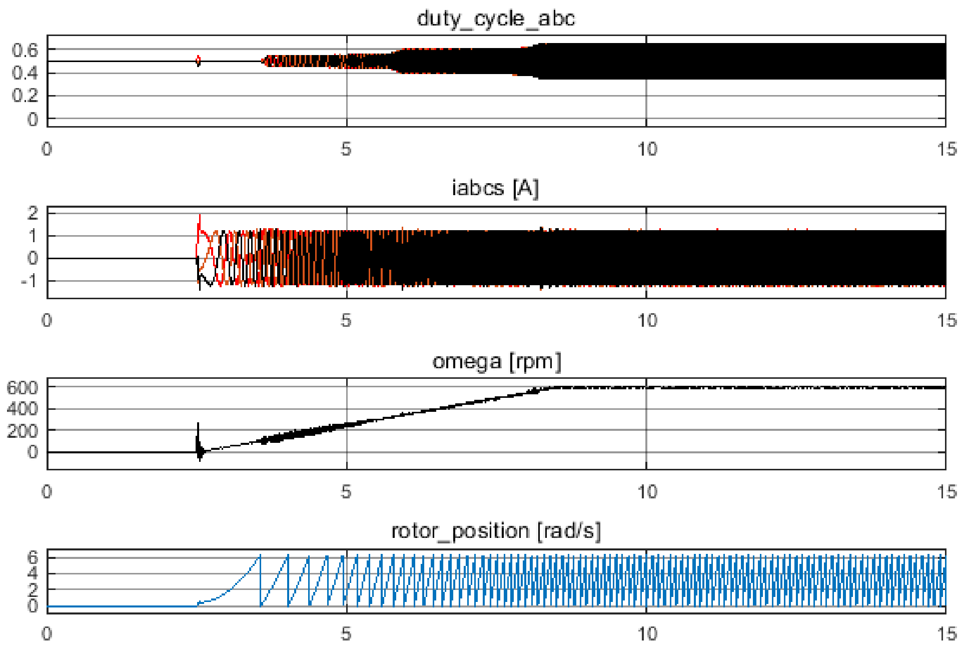


Figure 26: Signal visualisation, $i_d^* = 1.2$ [A] and $\omega_{\text{mec}}^* = 600$ [rpm]

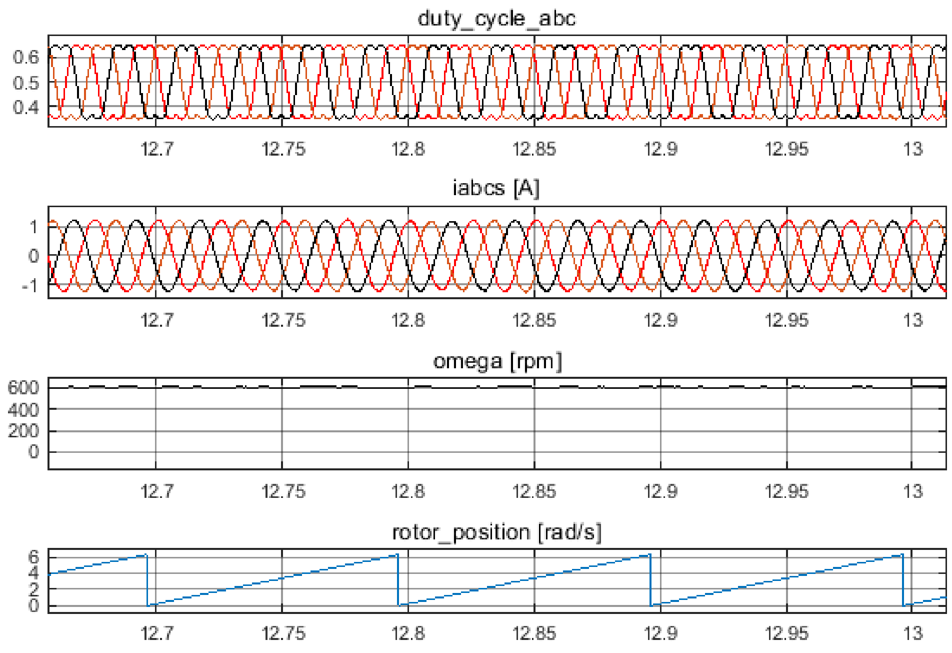


Figure 27: Signal visualisation in steady state, $i_d^* = 1.2$ [A] and $\omega_{\text{mec}}^* = 600$ [rpm]

The simulation results show how the current amplitude and the rotor speed track the i_d^* and ω_{mec}^* references in the steady state. Now that a correct operation was ensured in the simulations, it is time to proceed to the next stage: the Code Generation (sec. 4).

4 Code Generation

4.1 Overview of the Code Generation

After designing the control algorithm graphically, it is necessary to represent it in C code to be compiled and then loaded onto the MCU. The Code Generation is the capability of a software to create the code representation of an algorithm automatically. Compared to the handwritten code, the Code Generation brings a number of advantages:

- **High level error detection**: As the graphical algorithm is designed, it is easier to detect errors visually or by simulations. The errors propagated to the code are harder to find.
- **Ease of dealing with complexity**: The machine uses advanced optimisations for precise control of the generated code. As the algorithm becomes more complex, the code representation becomes harder and it is more likely to carry errors due to the handwriting.
- **Traceability**: Optimisation settings offers the possibility of generating readable and compact code, in order to trace which part of the block diagram is being represented in each part of the C code.
- **Cost and time reduction**: Easier designs and early error detection let save time and money from the debugging process and the implementation stage of the model.

The tool used in this context is Embedded Coder[12] for Matlab/Simulink. Given a Simulink model or subsystem, Embedded Coder is capable of building the C code. The way in which the code is generated and integrated to a processor will be explained with an example model and some pseudo codes in section 4.1.1.

4.1.1 Code Generation example

A simple example of a simulink model is presented, consisting of a subsystem that sums the values of two inputs and the result is multiplied by 10 (fig. 28).

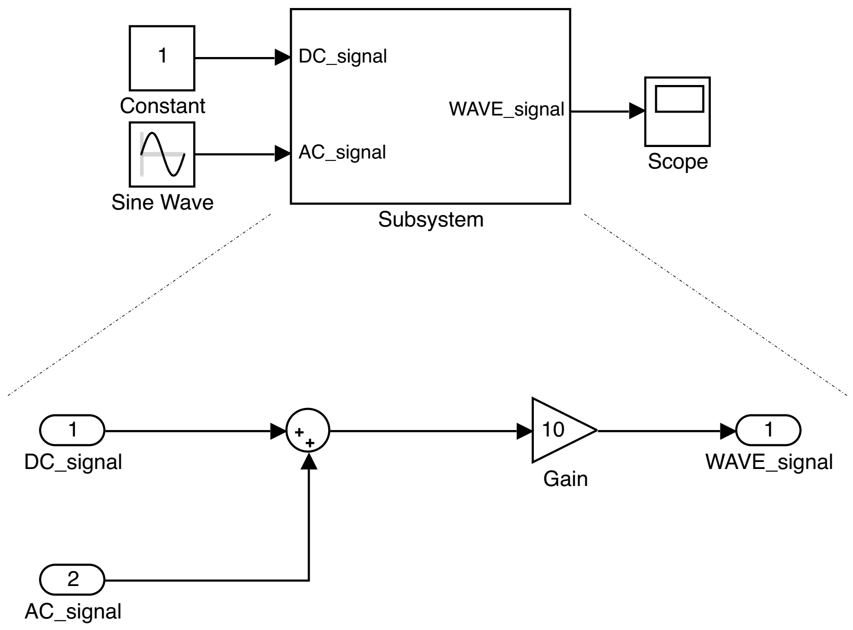


Figure 28: Code Generation example

The steps for generating the code of the subsystem are:

```
> Right-click on the Subsystem
    > C/C++ Code
        > Build This Subsystem
```

Then, three files are generated:

- ***rtwtypes.h***: Contains the hardware specific data-types.

```
/* ... code ... */
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef int int32_T;
typedef unsigned int uint32_T;
typedef float real32_T;
typedef double real64_T;
/* ... code ... */
```

- ***Subsystem.h***: Contains the model specific data-types and the function declaration.

```c
/* ... code ... */

/* model specific data-types */
typedef struct {
        const char_T * volatile errorStatus;
} M;
/* ... code ... */

/* function declaration */
extern void Subsystem_initialize(M *const Subsystem_M,
        real_T *input1, real_T *input2, real_T *output);
extern void Subsystem_step(M *const Subsystem_M,
        real_T input1, real_T input2, real_T *output);
extern void Subsystem_terminate(M *const Subsystem_M);

/* ... code ... */
```

- ***Subsystem.c***: Contains the source code for the model. Three functions are created:
  - *initialize*: Executed once when the MCU is turned ON to reset the model state
  - *step*: To execute one integration step for the model. This is executed at each cycle of the task scheduler or timer ISR with a fixed-step time.
  - *terminate*: Executed once to clean-up memory after the last execution of the model.

```c
/* ... code ... */

void Subsystem_initialize(M *const Subsystem_M,
        real_T *input1, real_T *input2, real_T *output)
{/* ... */}

void Subsystem_step(M *const Subsystem_M,
        real_T input1, real_T input2, real_T *output)
{
        /* ... */
        *output = (input1 + input2) * 10.0;
        /* ... */
}

void Subsystem_terminate(M *const Subsystem_M)
{/* ... */}

/* ... code ... */
```

These files are placed in the C project that will be compiled and loaded onto the MCU. The three functions are called from the source code of the project, using the MCU's peripherals as inputs and outputs. An example pseudo code is shown below:

```
/* ... code ... */

int main (void) {
        /* peripherals initialisation */
        TIMER_init();
        USART_init();
        ADC_init();
        GPIO_init();

        /* initialise generated model */
        Subsystem_initialize(Model,GPIO_input,
                ADC_input,USART_output);

        while (1) {
                /* end of model execution */
                if ( stop() ) break;
        }

        /* terminate generated model */
        Subsystem_terminate(Model);
        return 0;
}

void TIMER_ISR (void) {
        /* integration step for the model */
        Subsystem_step(Model,GPIO_input,
                ADC_input,USART_output);
}

/* ... code ... */
```

4.2 Integrating Code Generation with the STM32 Nucleo boards

4.2.1 Peripherals initialisation using STM32CubeMX

As explained before, a C project consists of a set of C files in which the control algorithm is called and linked to the MCU's peripherals. A toolchain is required in order to compile and link the C project to the MCU. Some of the toolchains are EWARM, TrueSTUDIO, SW4STM32 and MDK-ARM. For this case, the chosen toolchain was MDK-ARM that is compatible with the KEIL uVision IDE[13].

It was already shown how to generate the C code corresponding to the control algorithm, but the connection with the peripherals requires a previous initialisation of them from the chosen IDE. For the peripherals initialisation, it is necessary to include a set of header files from the STM32 HAL Library[4] and manipulate the values of some registers to have the desired configuration. The peripherals communicating with the *IHz_model.slx* control algorithm (sec. 3.1) and its settings are the following:

- TIM1: Timer peripheral based on an up-down counter at a 4 [kHz] frequency. It makes an Interrupt Service Routine (ISR) at the beginning of each cycle and sets a PWM output in three physical ports of the MCU.
- TIM4: Timer peripheral in encoder mode. It counts the pulses given by the encoder to compute the rotor position of the motor. Two physical ports are enabled to receive both channels A and B from the encoder.
- ADC1: 12 bit ADC registers that receive the current and voltage sensor measurements.
- USART2: USART register for communicating to the PC at a baud rate of 2,000,000 for monitoring the state of the internal variables of the MCU in real-time.

The PC application STM32CubeMX[5] is capable of auto-generating the peripherals initialisation code, providing a GUI (Graphic User Interface) for doing all the settings. These configurations are targeted to a specific MCU, in this case the STM32F303RETx Nucleo[2] was chosen. After the MCU selection in the application, the HCLK is set to 64 [MHz] from the Clock Configuration tab and the physical ports are interfaced with the peripherals from the Pinout tab (see fig. 29).

[4]The Hardware Abstraction Layer (or HAL) Library contains high level C functions for the STM32 MCUs in order to simplify the initialisation and configuration of the peripherals

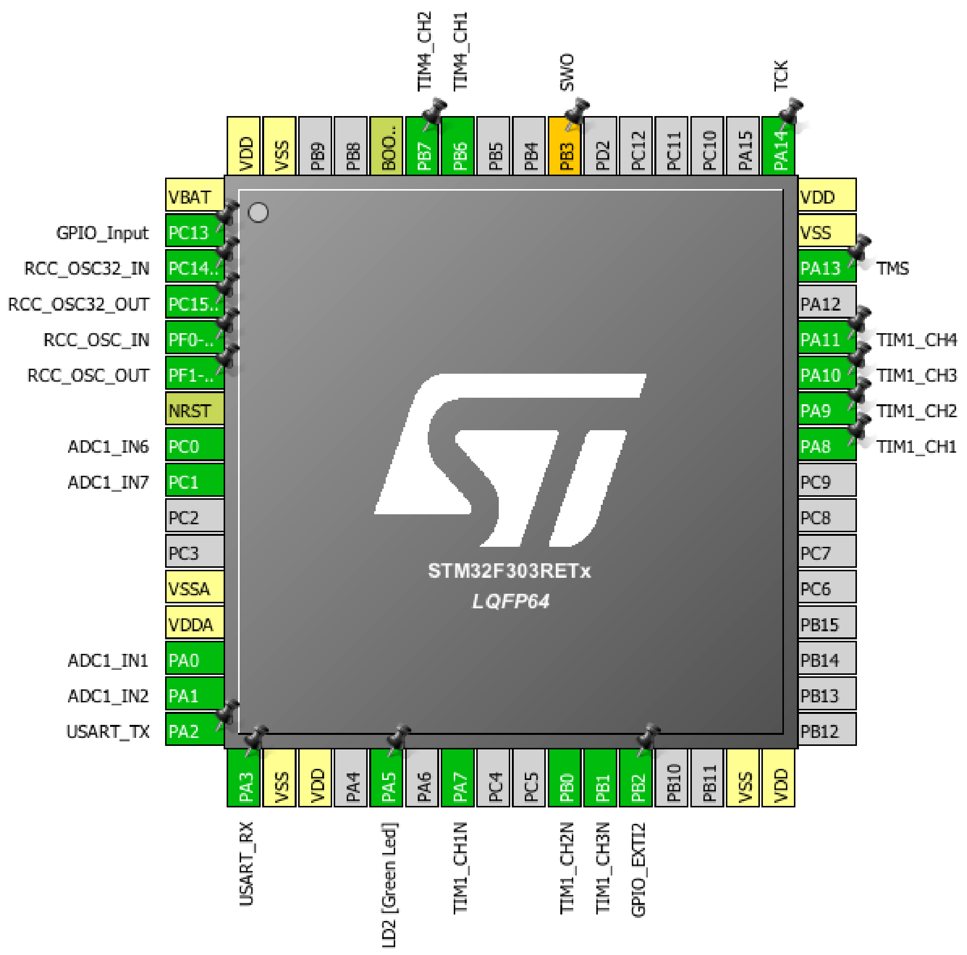


Figure 29: Interfacing the physical ports and peripherals from STM32CubeMX

From the Pinout tab, the channels of the peripherals are chosen and from the Configuration tab, their registers are configured to work as explained before (see annex A2). Finally, the MDK-ARM V5 toolchain is selected from the Project Settings and the code is generated. The auto-generated *main.c* file contains all the initialisation functions of the peripherals and here the functions of the control algorithm may be added.

```
/* ... code ... */
int main (void) {

        MX_GPIO_Init();
        MX_USART2_UART_Init();
        MX_TIM1_Init();
        MX_ADC1_Init();
        MX_TIM4_Init();

        while (1);
        return 0;
}
/* ... code ... */
```

4.2.2 Linking to the peripherals via STM32 Embedded Target

Until now, it is presumed that the Code Generation of the control algorithm and the linking to the MCU's peripherals are separated processes. However, a useful tool is provided by STMicroelectronics to perform all parts of the Code Generation together from a single platform, this is the STM32 Embedded Target[1] for Matlab and Simulink. This tool provides a set of Simulink blocks in the Library Browser that represents the peripherals of the STM32 MCUs, whose ports can be graphically connected to the control algorithm in a Simulink model. STM32 Embedded Target is linked to Embedded Coder, so the entire C project (control algorithm, peripherals initialisation and their connection) is auto-generated and ready to be compiled, with no need to add handwritten code.

The installation instructions for the STM32 Embedded Target is explained in [1]. After the installation is done, an empty Simulink model is created with the following configuration:

```
> Configuration Parameters
    > Code Generation
        > System target file
            > stm32.tcl
```

The peripheral blocks are under the label Target Support Package - STM32 Adapter in the Browser Library. Drag and Drop the STM32_Config block to the model (see fig. 30). This block calls an ioc file[5] to enable the Simulink blocks corresponding to the configured peripherals in section 4.2.1. The name of the STM32_Config block will change to the targeted MCU.

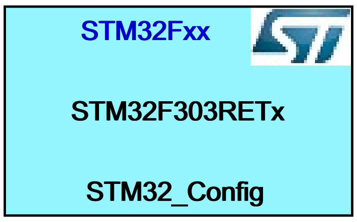


Figure 30: STM32 configuration

4.2.3 Description of the Simulink model for the Code Generation

- According to the project, two ISR are executed: a Timer ISR for the step function of the control algorithm and an External Interrupt for the Z channel of the encoder. Drag both the Timer and the GPIO_Exti blocks to the model, whose configurations are detailed in annex A3. The interrupt signals of these peripherals are updated in their outports. Each of these signals triggers an ISR inside a Function-Call block[6] (see fig. 31).

[5]The application STM32CubeMX creates files with ioc extension.
[6]This is a kind of subsystem that is executed when it is triggered by an external signal.

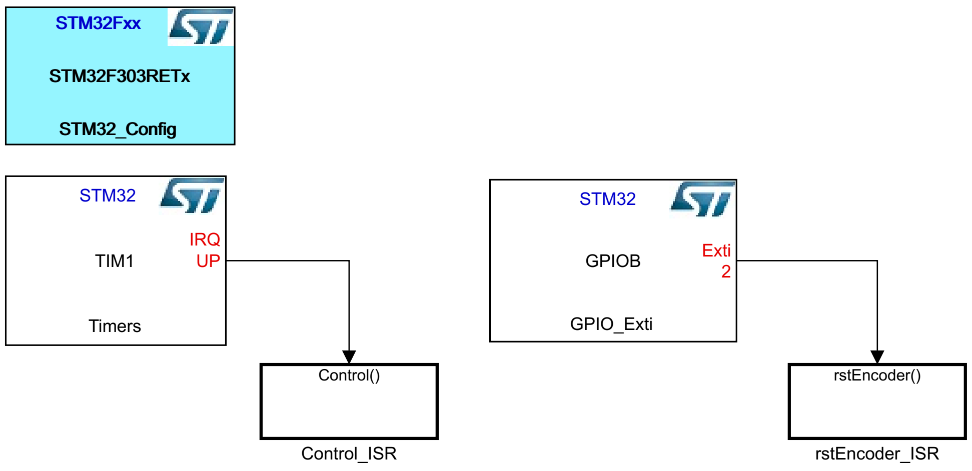


Figure 31: STM32 configuration and interrupt service routines

- The ISR triggered by the Z channel of the encoder resets the counting of the timer TIM4. In figure 32, the Reset_CNT inport of the Timer block corresponding to TIM4 is set.

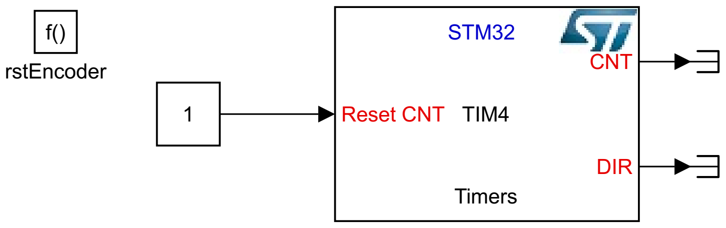


Figure 32: rstEncoder_ISR

- The ISR triggered by TIM1 contains the Model Reference block that calls the *IHz_model.slx* control algorithm (see fig. 33).

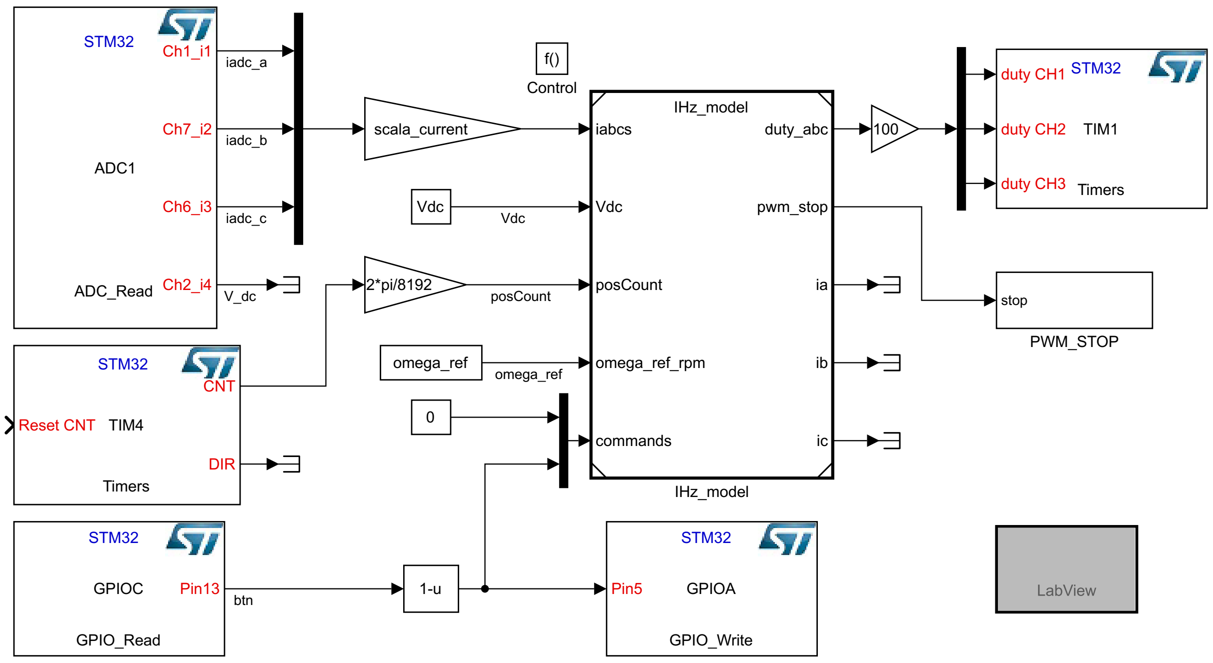


Figure 33: Control_ISR

This ISR also contains the following peripheral blocks[7]:

- **ADC1**: Called from the ADC_Read block. It reads the injected channels from the corresponding physical inports of the MCU and gives a 12-bit value (0 to 4095). The channels corresponding to the current measurements are multiplied by a scale factor, so they can be treated as the actual values of the currents by the *iabcs* port of the control algorithm.
- **TIM4**: Called from the Timers block. It provides the counted pulses given by the encoder. The counter value is scaled to get the angle value of the rotor in radians.
- **GPIOC**: Called by the GPIO_Read block. It reads the binary value of a push button to carry out the *Go* command.
- **GPIOA**: Called by the GPIO_Write block. Turns ON or OFF a LED depending on the state of the *Go* command. The LED is ON only when the push button is pressed, otherwise it is OFF. Its purpose is just for debugging.
- **TIM1**: Called from the Timers block. It receives the duty cycle values between 0 and 100, so the PWM signal is produced in the corresponding physical ports of the MCU.
- **PWM_STOP**: Depending on the value of its input, the PWM channels are enabled or disabled using the if-else statement blocks. The enabling or disabling of these channels are done in the REGISTER Access block as shown in figure 34.

[7]The detailed information of the blocks configuration is in annex A3

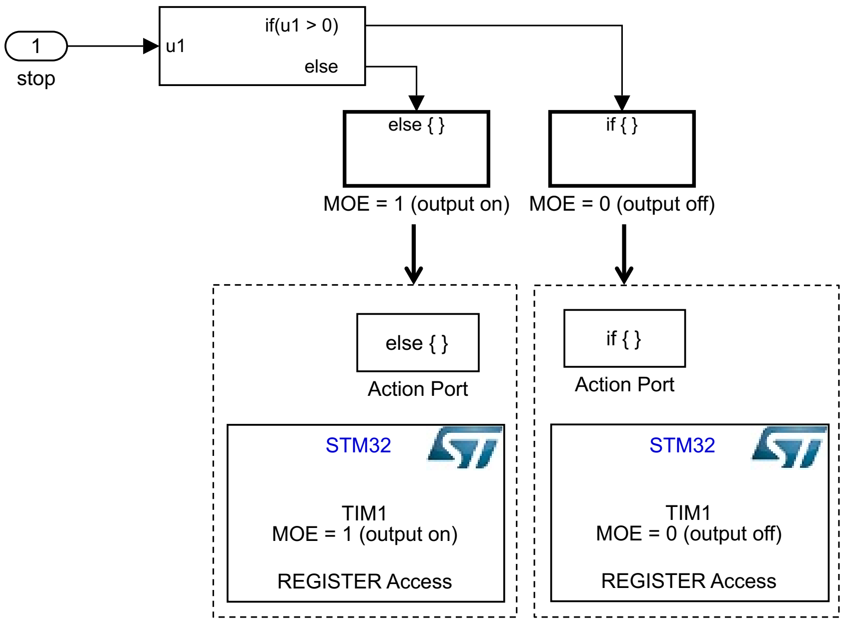


Figure 34: if-else statement

- **LabView**: It contains a routine for communicating with a virtual oscilloscope designed in the LabView software. The used protocol is USART.
- The last step for the Code Generation is to press Build Model in Simulink. When the project is built, it will be asked to open the C project within the IDE (see annex A4). There is no need no modify the code, so the Compile option has to be selected and then is time to load the compiled files onto the MCU by the ST-LINK protocol[8].

[8]The ST-LINK debugger/programmer interface is included the STM32 Nucleo boards and its protocol is compatible with the KEIL uVision IDE.

4.2.4 Bottom unit

The layers of the full executable project and their provenance are summarised in figure 35. It is also shown in the centre that the C project contains the control algorithm and their signals linked to the MCU's peripherals, and the latter interface the hardware through the physical ports of the MCU on the sides.

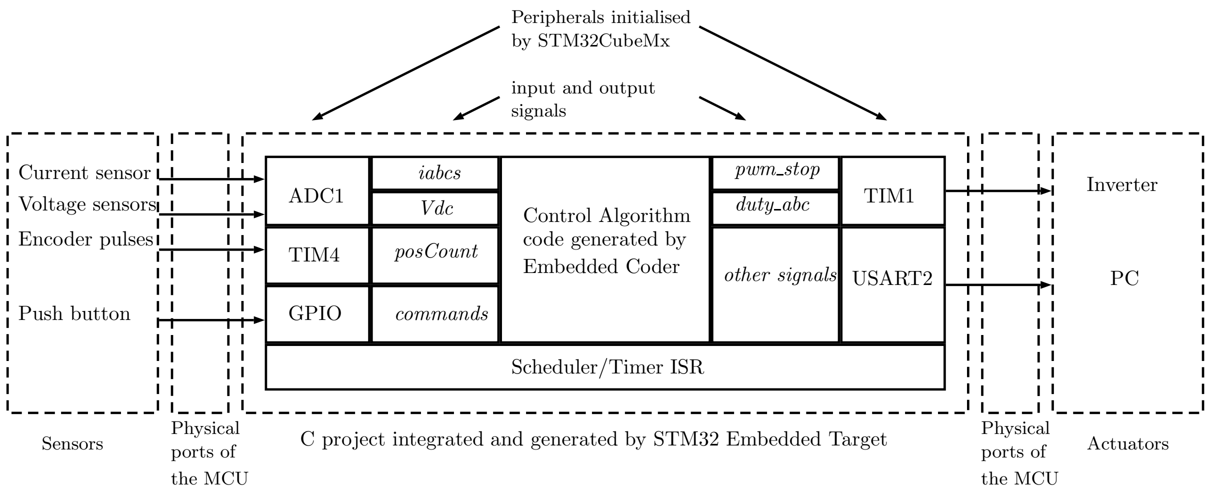


Figure 35: The project outline

5 Results

5.1 The generated C project

The C project and its components are named after Code Generation platforms[9]. As shown in annex A4, it contains a set of sub-folders with the C and header files. Only the most relevant files will be explained below:

- ***main.c***: This file contains the peripherals initialisation and calls the *initialize* and *step* functions of the *PROJECT_NAME.c* file. The *terminate* function does not exist because the MCU clears its memory when it is reset or turned off.
- ***PROJECT_NAME.c***: It contains the *initialize* function, where the initial values of the signals that connects the peripherals and the control algorithm are given, the interrupt service routines are enabled and the control algorithm is started. The *step* function is empty and instead the execution is written in the corresponding ISR.
- ***PROJECT_NAME_EXTI.c***: It contains the ISR that resets the counter value of the encoder pulses.
- ***PROJECT_NAME_TIM.c***: Here, the *step* function is run inside the timer ISR. In this instance, the *step* function is called *IHz_model* (declared in the *IHz_model.c* file) and its inputs and outputs are the scaled values of the corresponding registers with the readings/writings of the physical ports.
- ***IHz_model.c***: It contains the *IHz_model* function with the logic of the control algorithm.

5.2 Experimental set-up

The experimental set-up consists of interfacing the MCU board, the inverter, the current sensors, the voltage sensor, the power supply and the AC motor. The inverter and the sensors are included in the X-NUCLEO-IHM08M1 shield, which is compatible with the STM32F303RETx Nucleo board. The mentioned shield has a port for connecting the inverter's DC-link to a power supply, and three output ports corresponding to the three phases that supply the AC motor. The AC motor corresponds to the Microphase S140-2B353 permanent magnet synchronous motor that also includes an encoder, whose channels (A, B and Z) are connected to the Nucleo board for reading the rotor position. Figure 36 shows the experimental set-up.

[9]The whole project is named after the selected ioc file. The names of the files generated by Matlab/Simulink starts with the Simulink model's name (PROJECT_NAME).

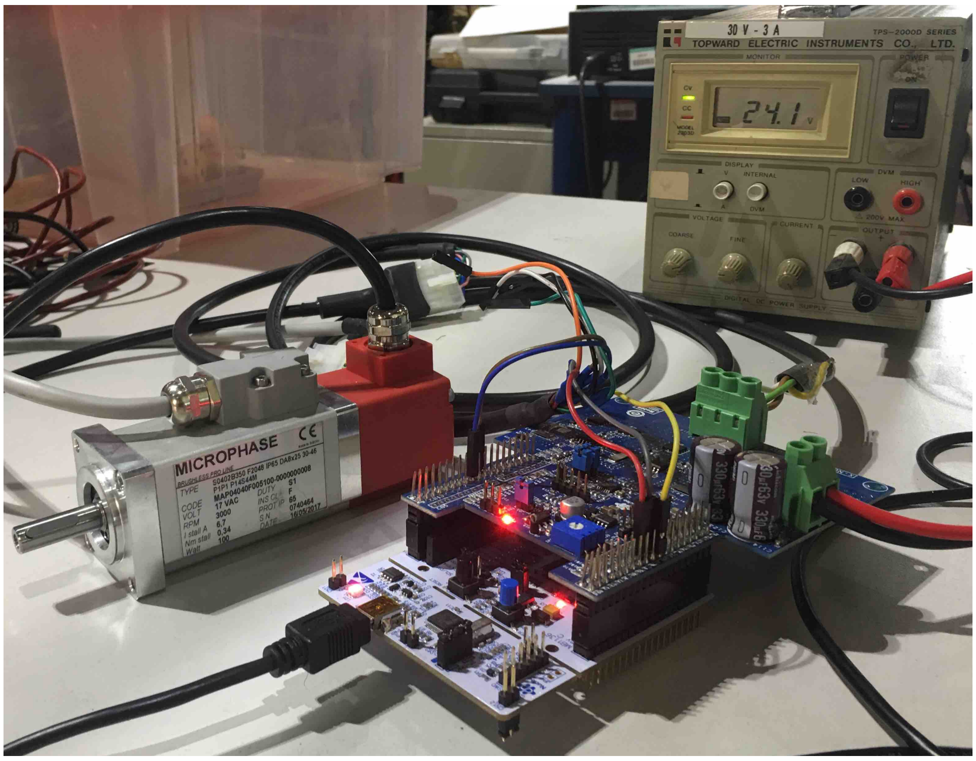


Figure 36: Experimental set-up

The advantage of using a shield-based inverter is that no wires are required for the connection between the inverter drivers and the sensors to the MCU board. The shield also lets the user access to the debugger and the reset buttons of the MCU board. On the other hand, for interfacing the encoder to the MCU, the A, B and Z channels and the 5V supply are connected using wires. The detailed pin connection is shown in the figure below.

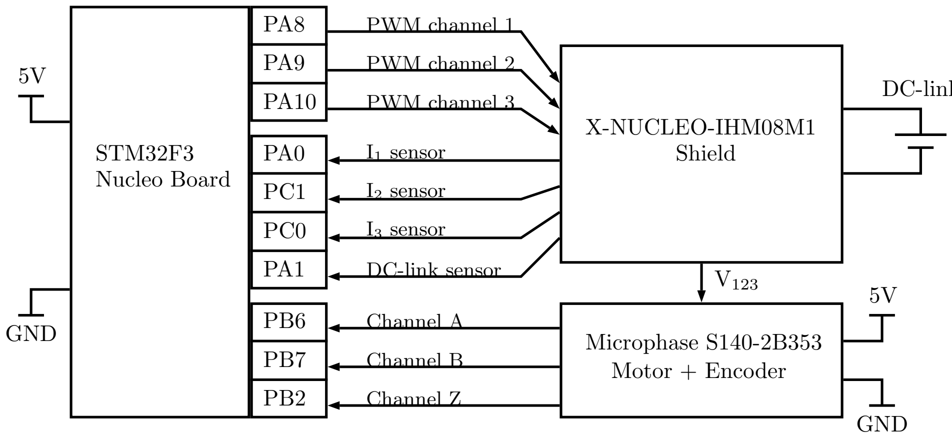


Figure 37: Detailed pin connection

Finally, the MCU board communicates to a computer using the USART protocol (with an USB cable) for doing the debugging. A serial communication is established in order to collect data from the MCU and visualise it in real-time while the control system is operating. For this work, a virtual oscilloscope software created in LabView is used to visualise the sensor measurements and other internal values of the MCU. This software receives a map file[10], so the name of a certain variable can be sent to the MCU as its address pointer in order to receive or modify the stored value. Figures 38 and 39 show the set-up for the data acquisition.

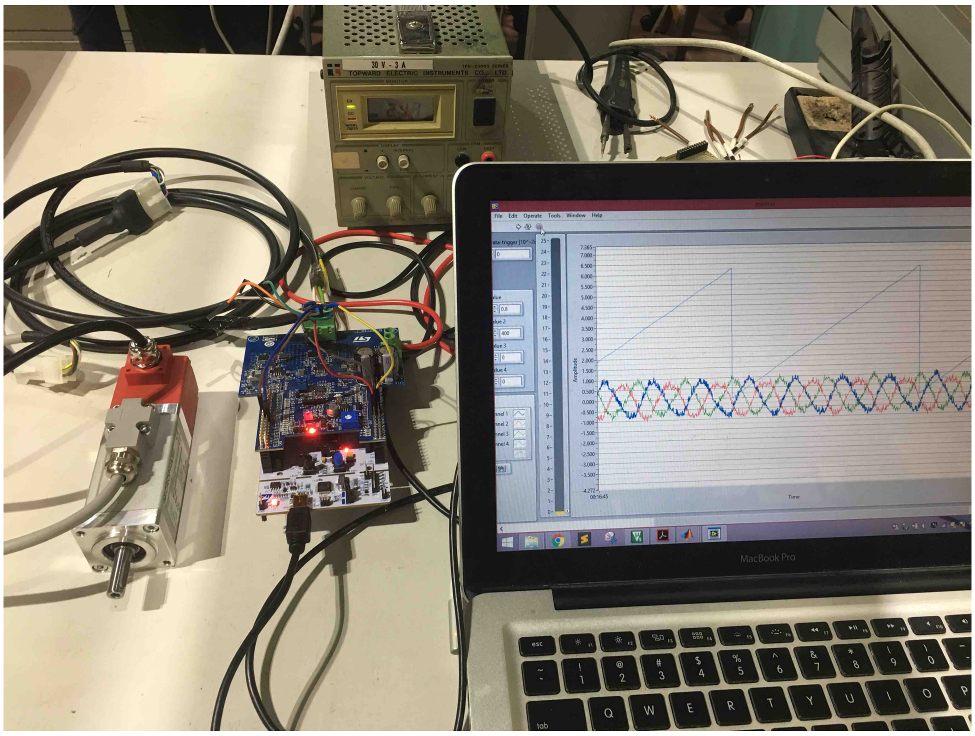
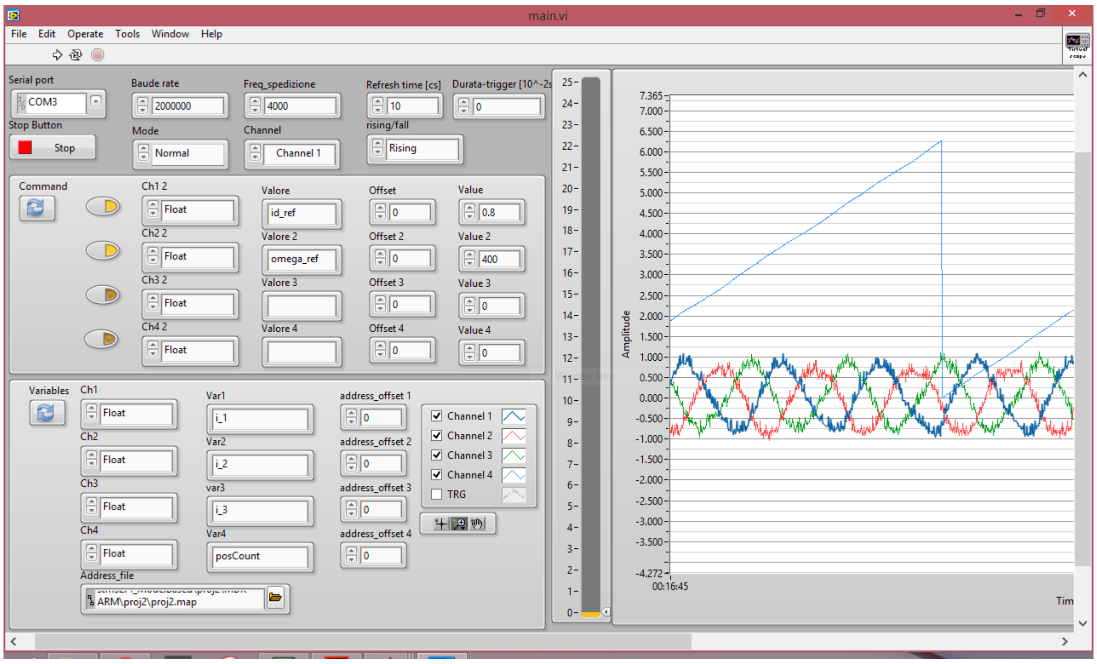


Figure 38: Set-up with data acquisition



main.vi

File Edit Operate Tools Window Help

Serial port: COM3
Stop Button: Stop
Baude rate: 2000000
Mode: Normal
Freq_spedizione: 4000
Channel: Channel 1
Refresh time [cs]: 10
rising/fall: Rising
Durata-trigger [10^-2s: 0

Command

Ch	Type	Valore	Offset	Value
Ch1 2	Float	id_ref	0	0.8
Ch2 2	Float	omega_ref	0	400
Ch3 2	Float		0	0
Ch4 2	Float		0	0

Variables

Ch	Type	Var	address_offset
Ch1	Float	i_1	0
Ch2	Float	i_2	0
Ch3	Float	i_3	0
Ch4	Float	posCount	0

Address_file: ARM\proj2\proj2.map

[✓] Channel 1
[✓] Channel 2
[✓] Channel 3
[✓] Channel 4
[ ] TRG

Figure 39: The virtual oscilloscope

[10]When the C project is built from the IDE, a map file is generated. The map file contains the memory address of each of the MCU's variables.

5.3 Experimental operation and debug

As the simulation of section 3.2 was successful, it is expected to have similar results in the real implementation of the control system. The modelled control system was based on the hardware ratings, whose main values are detailed in table 2.

PMSM	3-phase Inverter shield
• $V_{\text{supply}} = 17$ [VAC]	• 2-level output signal per leg
• $I_{\text{stall}} = 6.7$ [A]	• DC-link from 8 to 48 [VDC]
• $P_{\text{nom}} = 100$ [W]	• $I_{\text{out}}^{\text{max}} = 15$ [ARMS] in each phase
• $T_{\text{nom}} = 0.32$ [Nm]	• 3.3 [V] compatible NMOS gate driver
• $\omega_{\text{nom}}^{\text{mec}} = 3000$ [rpm]	• Power NMOS with $R_{DS} = 0.0014$ [Ω]
• $K_T = 0.05$ [Nm/A]	
• $R_{u-v} = 0.5$ [Ω]	
• $L_{u-v} = 0.53$ [mH]	

Table 2: Hardware ratings

Two tests with different control references are carried out to check the behaviour of the AC motor. Relevant values such as stator currents, rotor position, and PWM duty cycle are acquired and plotted for being confronted with the simulations of section 3.2. Some differences might be noticed between both responses due to the nonidealities ignored in the simulation, however, those differences should be minimal to validate a correct implementation. Also, readings of the transient behaviour of the motor are acquired for a dynamical analysis, to be sure that the plant operates within the safety limits.

5.3.1 Test 1

The first implementation to test uses the control references of $i_d^* = 0.8$ [A] and $\omega_{\text{mec}}^* = 400$ [rpm]. The current measurements and duty cycle data in steady state are collected and compared to the simulation results (fig. 40).

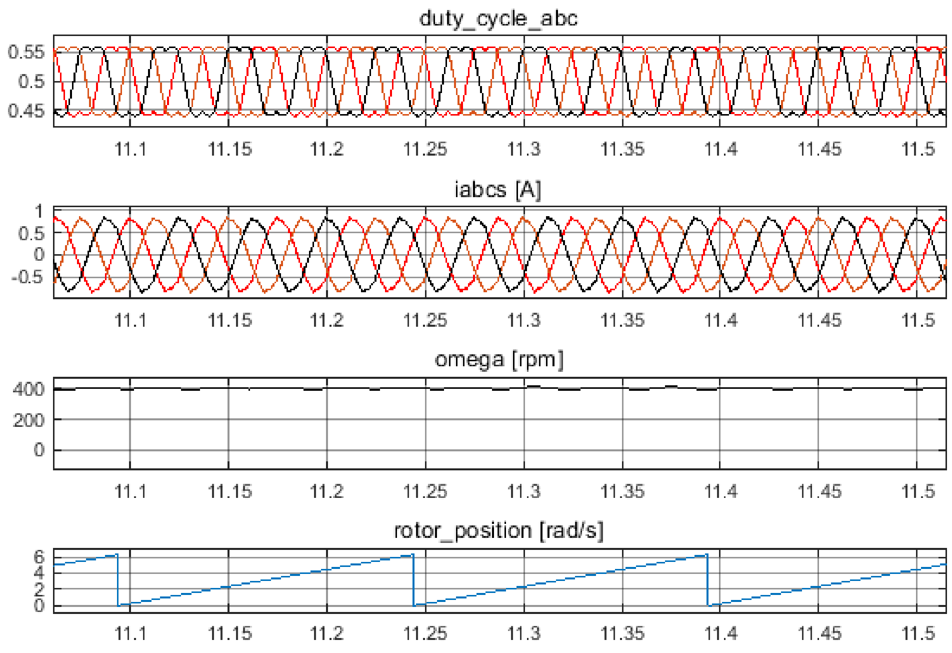


Figure 40: Simulation with $i_d^* = 0.8$ [A] and $\omega_{\text{mec}}^* = 400$ [rpm]

The data acquired from the real implementation is shown in figures 41, 42 and 43. It can be appreciated that the duty cycles in both simulation and implementation are very similar. The current amplitudes are 0.8 [A] as it was set in the reference, but it also can be noticed an oscillation of the current offset. This phenomenon is caused by the nonidealities that were not considered in the simulation, such as the high-frequency noise of the inverter, the eccentricity and the motor harmonics. The first nonideality is the noise introduced by the switching of the transistors, generating current peaks that are read by the shunt sensor. The eccentricity is a condition of unequal air gap that exists between the stator and the rotor, which produces different currents in each phase due to the reluctance irregularity. Finally, the saturation and non-sinusoidal flux distribution in the stator introduces harmonics in the currents.

The noise produces an error that is integrated and reduced by the PI regulator, causing an oscillating component in the output current but keeping the main value as the control reference. In contrast, the rotation speed of the rotor is less affected by the noise propagation due to its inertia. From the acquired data, it can be appreciated the frequency of the rotor position signal is exactly the same as the speed reference value.

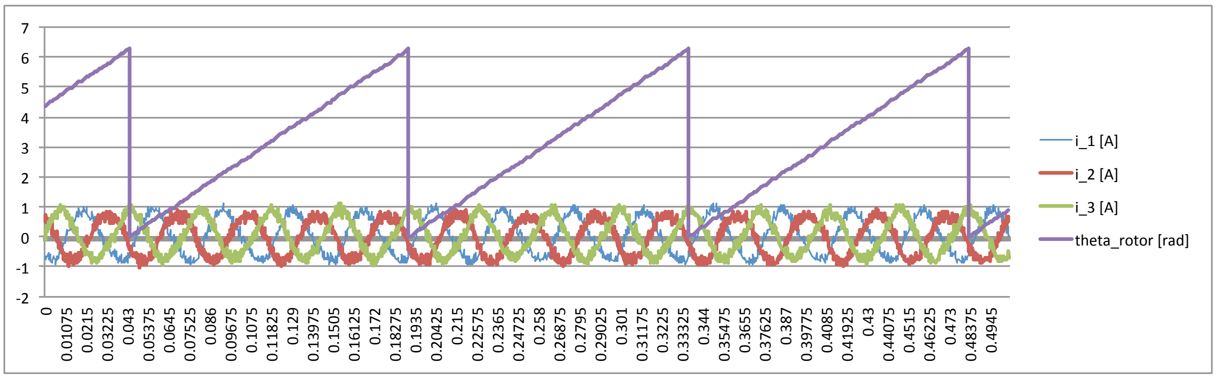


Figure 41: Stator currents and rotor position

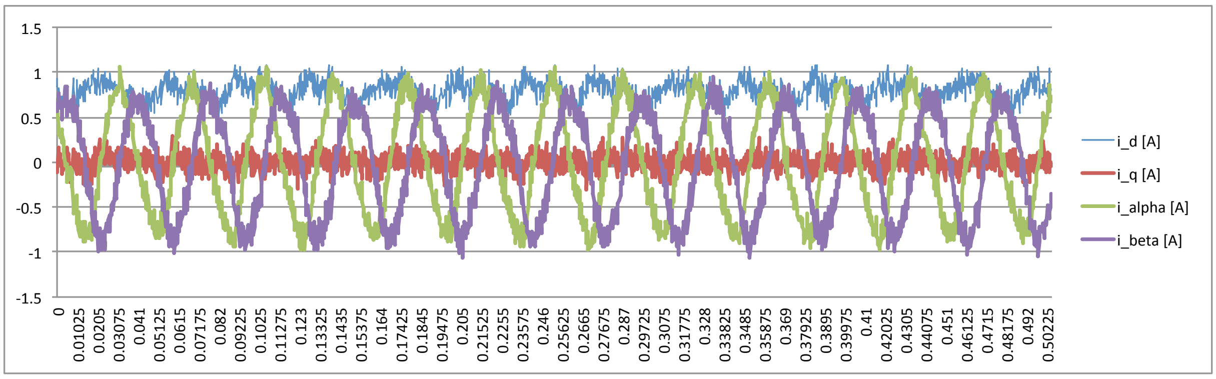


Figure 42: Stator currents $i_{\alpha\beta}$ and i_{dq}

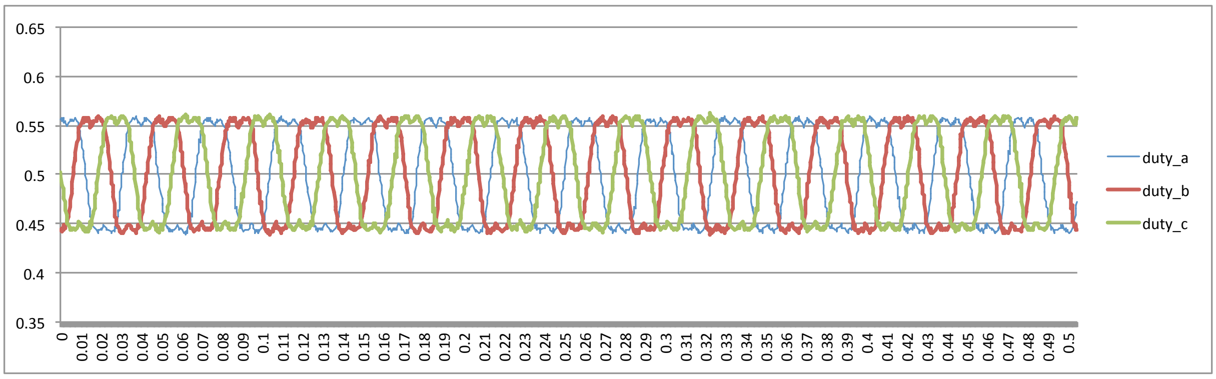


Figure 43: PWM duty cycles

5.3.2 Test 2

The results of the simulation with the control references of $i_d^* = 1.0$ [A] and $\omega_{\text{mec}}^* = 500$ [rpm] are shown in figure 44.

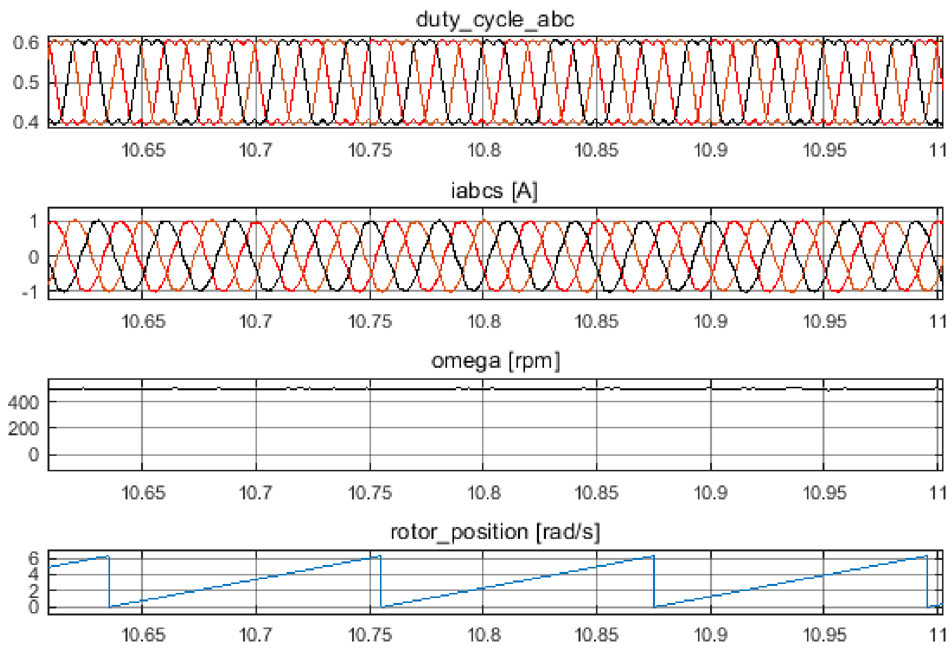


Figure 44: Simulation with $i_d^* = 1.0$ [A] and $\omega_{\text{mec}}^* = 500$ [rpm]

The same event happens with those references, in which the measured i_d current oscillates around 1.0 [A] due to the introduced noise (see 45, 46 and 47). Nevertheless, the designed control system has been capable to make the plant track the reference values and reduce the error around zero, even with these nonidealities.

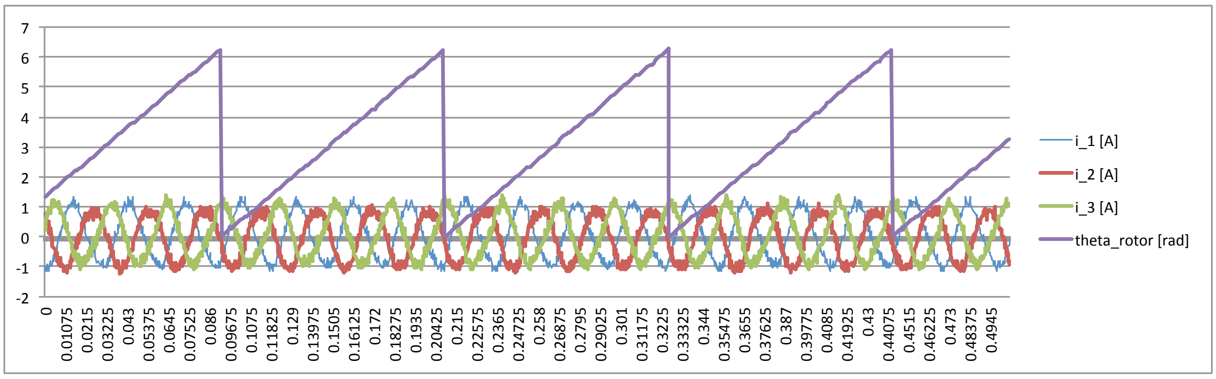


Figure 45: Stator currents and rotor position

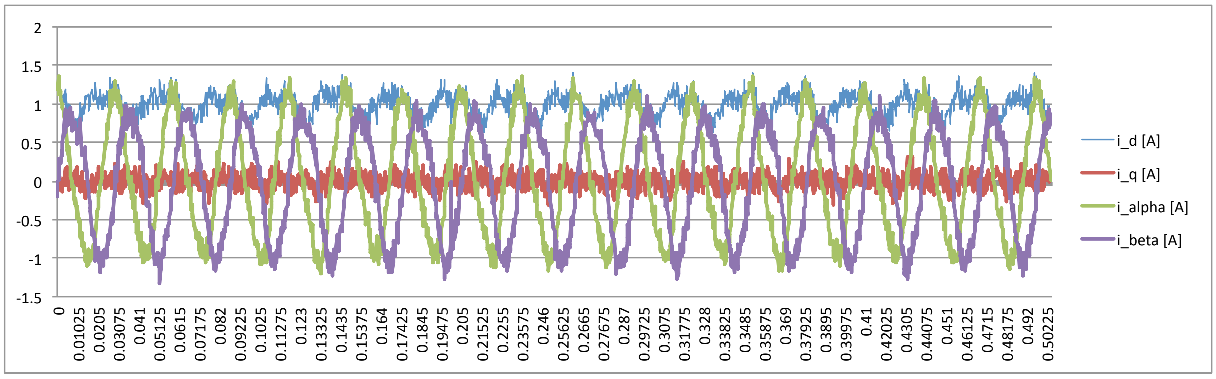


Figure 46: Stator currents $i_{\alpha\beta}$ and i_{dq}

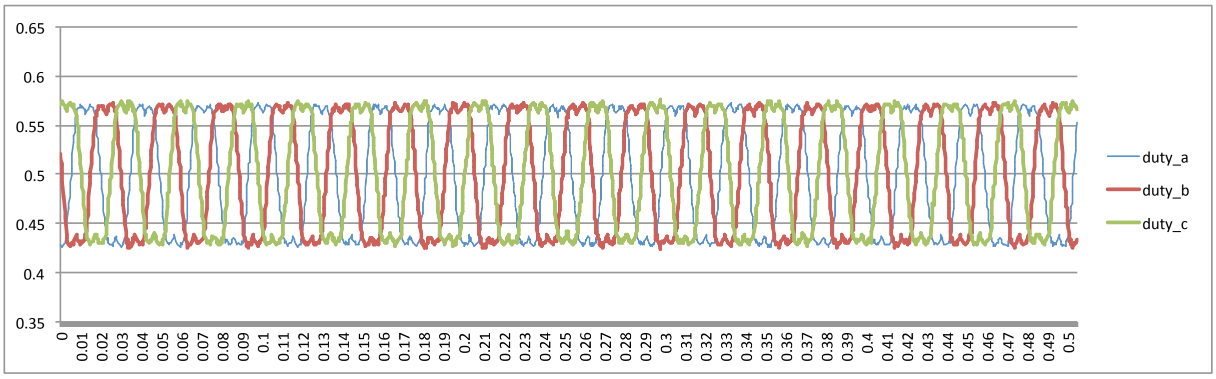


Figure 47: PWM duty cycles

To make sure the system is operating within the safety limits, the data is acquired during the start of the motor (figures 48, 49, 50 and 51). The fact of using a slow ramp in the speed reference limits the speed increase under a given acceleration. This avoided an abrupt response of the system, having its current in the desired range without a significant overshoot.

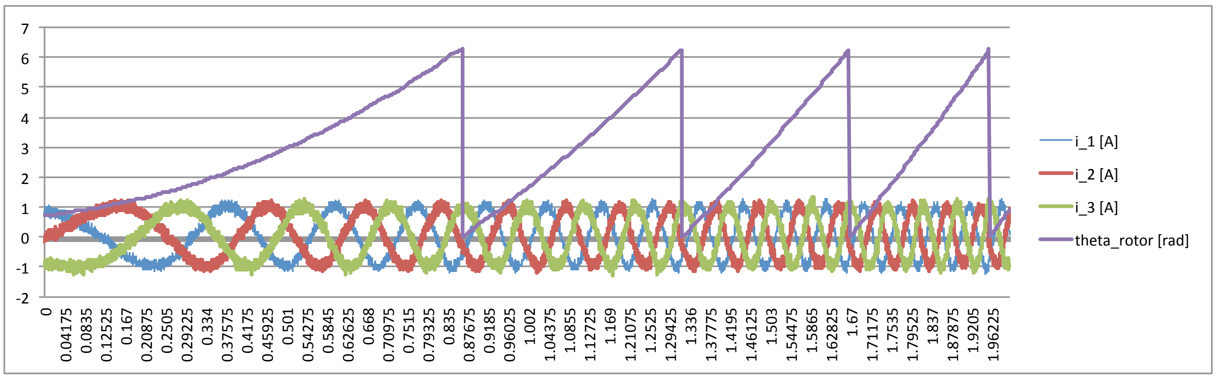


Figure 48: Stator currents and rotor position

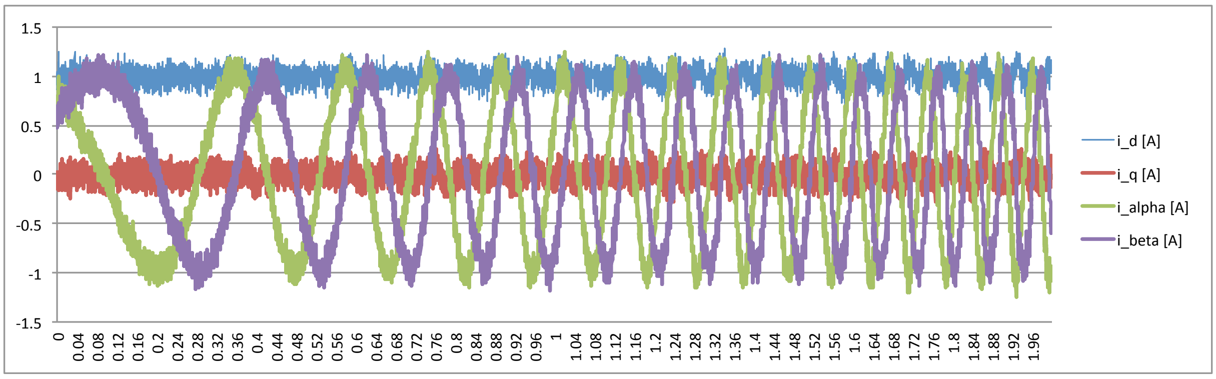


Figure 49: Stator currents $i_{\alpha\beta}$ and i_{dq}

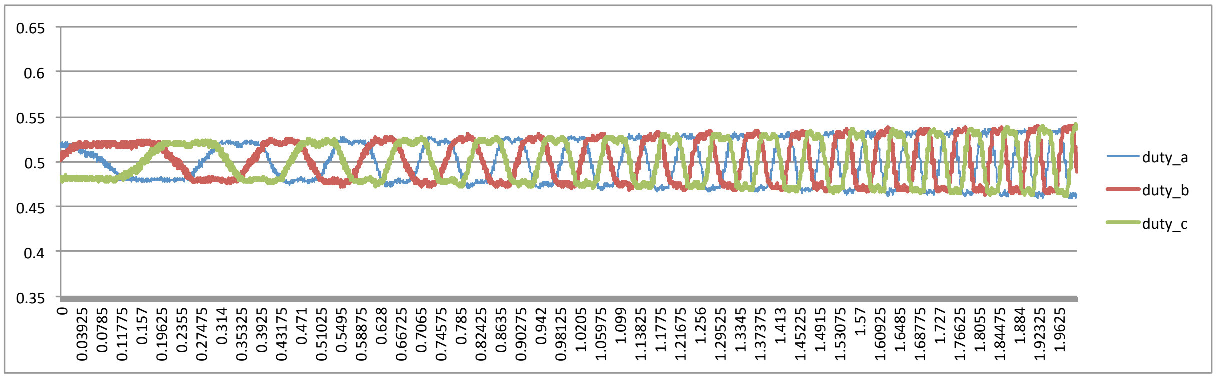


Figure 50: PWM duty cycles

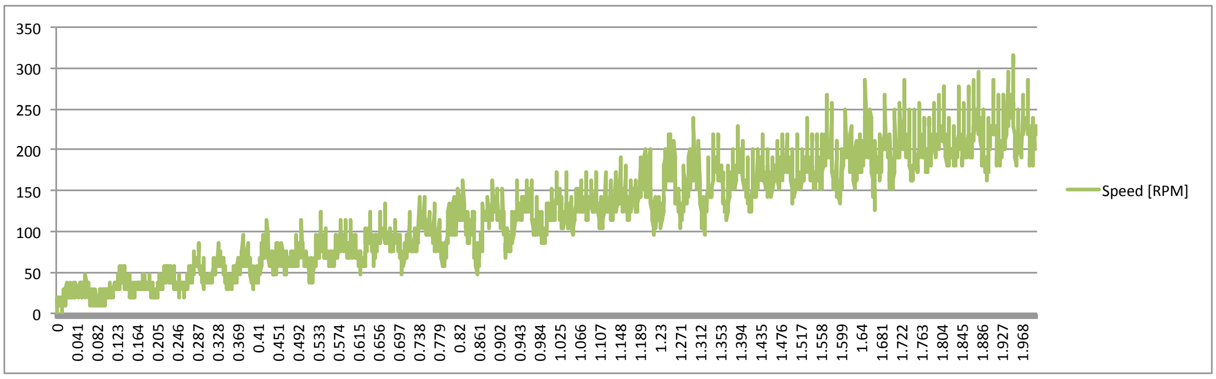


Figure 51: Rotor speed

6 Conclusions

In this work was illustrated a method to implement a motor control system with an STM32 Nucleo board using the Model-Based Design approach. Many steps that are commonly used were skipped or simplified, specially what regards to the mathematical modelling and code programming. Most of the project was focused in the modelling and simulation of the control system, while the implementation was practically automated using the proper software. A complex mathematical model was simply implemented by linking blocks and no C code was ever written.

For creating the I-Hz controller, a control algorithm was programmed with blocks using the tools provided by Matlab/Simulink. Complex mathematical representations for differential equations were simplified with discretisation and high level function blocks, while different tasks such as variables adaptation, control execution and over-current protection were scheduled in parallel states machines, whose tools are provided by the Stateflow toolbox.

Organising the control logic graphically in block diagrams and states machines facilitated the understanding of the algorithm and made easier the error detection from a high level perspective. The performance of the control algorithm was successfully tested with a modelled inverter and motor in a simulation, from which a similar behaviour is being expected in the experimental implementation of the control system.

The STM32CubeMX application provides an easy way to enable the MCU's peripherals and configure their registers. The configuration is done from a Graphic User Interface, where the pinout of the MCU and the available peripherals' options are displayed. The STM32 Embedded Target toolbox links a Simulink model to an ioc file (from the STM32CubeMX application) in order to make available the enabled peripheral blocks in the Simulink environment. Interfacing the control algorithm with the peripheral blocks from a Simulink environment is an easy and fast task if compared to the modelling and simulating stages.

The Simulink model that links the control algorithm with the peripherals contains all the control logic integrated to the target MCU, so the Embedded Coder toolbox was capable of carrying out the Code Generation of whole C project. The main advantage of having used the STM32 Embedded Target toolbox is that no manipulation is required in the C project for adding the HAL library and integrating the generated functions to the MCU's peripherals and routines because all this steps are automated. Other advantage is that the project is generated in a readable way, so it can be revised to trace the matching between the code and the graphical model.

From the KEIL uVision IDE is a C project editor, compiler and target linker, however, only the two latter options were required. The purpose of the IDE is just to load the control logic onto the STM32 Nucleo board, which is then ready to be connected to the hardware of the control system and start operating. A virtual oscilloscope was also included in the real implementation in order to acquire and to visualise the data, corresponding to the sensor readings, encoder readings and other internal variables of the MCU during the real operation.

Regarding the results, it has been seen that the control algorithm works similar to the simulated model, but it also has to deal with the noise introduced due to the high-frequency switching of the inverter, the eccentricity of the rotor and the motor harmonics that are propagated into the feedback loop. These nonidealities are not normally included in the default Simulink models of electric motors, and in the real implementation they produce some oscillation in the response due to the error integration by the PI regulator. Anyway, the error presence can be tolerated if it remains within certain limits and does not affect the performance of the control system significantly.

The Model-Based Design method was carried out with the proper software in order to implement a motor control system, both in simulation and in hardware from a single platform. The transition from the modelling to the real implementation becomes faster and carries exactly the same logic, so the same involved variables could be monitored in both cases for comparing them and validate the results. Using this method, time and effort reduction in the control system development process was achieved.

References

[1] STMicroelectronics, “STM32-MAT/TARGET Hands On”, Rev 2.1, 2014.

[2] STMicroelectronics, “STM32F303xB/C/D/E, STM32F303x6/8, STM32F328x8, STM32F358xC, STM32F398xE advanced ARM®-based MCUs”, Reference manual, Rev 8, 2017.

[3] STMicroelectronics, “STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM®-based 32-bit MCUs”, Reference manual, Rev 15, 2017.

[4] STMicroelectronics, “Getting started with X-NUCLEO-IHM08M1 low-voltage BLDC motor driver expansion board based on STL220N6F7 for STM32 Nucleo”, User manual, Rev 2, 2016.

[5] STMicroelectronics, “STM32Cube initialization code generator”, [Online]. Available on: `http://www.st.com/en/development-tools/stm32cubemx.html`. Accessed on 2018.

[6] MathWorks, “S-Function Basics”, [Online]. Available on: `https://www.mathworks.com/help/simulink/s-function-basics.html`. Accessed on 2018.

[7] MathWorks, “Legacy Code Integration”, [Online]. Available on: `https://www.mathworks.com/help/simulink/legacy-code-integration.html`. Accessed on 2018.

[8] MathWorks, “Model-Based Design”, [Online]. Available on: `https://www.mathworks.com/help/simulink/gs/model-based-design.html`. Accessed on 2018.

[9] MathWorks, “Stateflow: Model and simulate decision logic using state machines and flow charts”, [Online]. Available on: `https://uk.mathworks.com/products/simpower.html`. Accessed on 2018.

[10] MathWorks, “Model and simulate electrical power systems”, [Online]. Available on: `https://www.mathworks.com/help/physmod/sps/powersys/ref/permanentmagnetsynchronousmachine.html`. Accessed on 2018.

[11] MathWorks, “Permanent Magnet Synchronous Machine”, [Online]. Available on: `https://www.mathworks.com/help/physmod/sps/powersys/ref/permanentmagnetsynchronousmachine.html`. Accessed on 2018.

[12] MathWorks, “Embedded Coder: Generate C and C++ code optimized for embedded systems”, [Online]. Available on: `https://www.mathworks.com/products/embedded-coder.html`. Accessed on 2018.

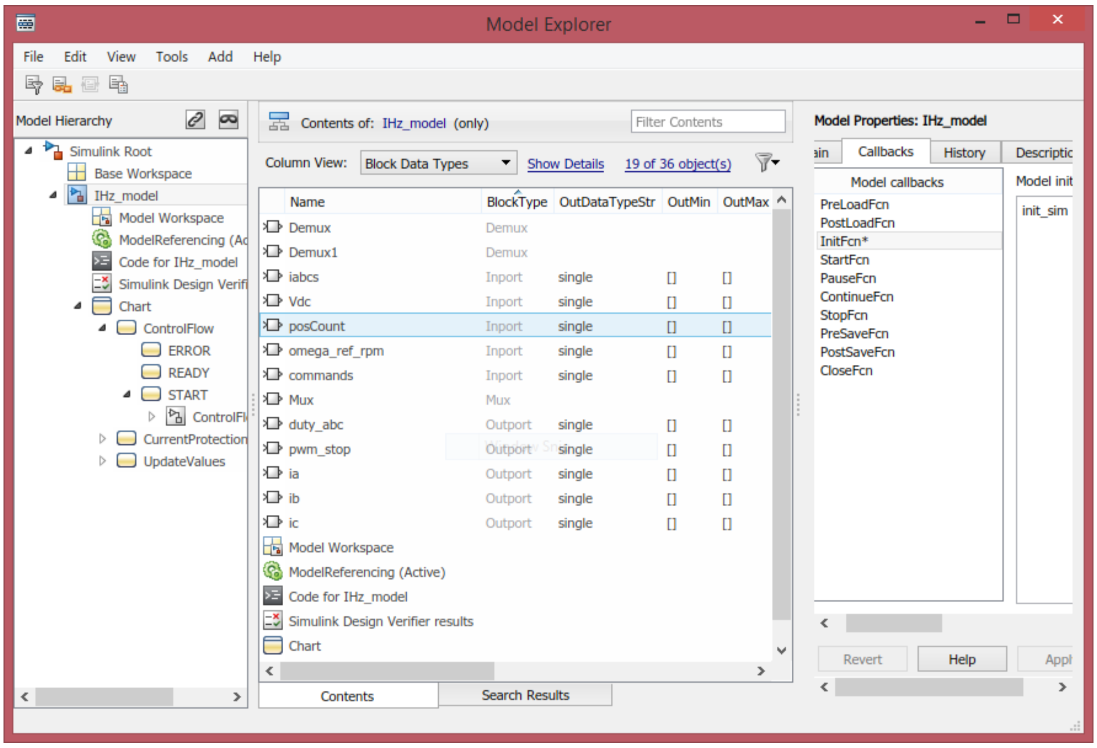
[13] Keil, “μVision IDE”, [Online]. Available on: `http://www2.keil.com/mdk5/uvision/`. Accessed on 2018.

[14] Microphase, “S140 series brushless servomotors”, Datasheet.

[15] O. Moreira, “Rapid Control Prototyping Using an STM32 Microcontroller”, Research Bachelor Thesis, Institut fur Elektrische Informationstechnik, Technische Universitat Clausthal, Clausthal-Zellerfeld, Germany, 2015.

7 Annexes

A1 Model Explorer configurations for the control algorithm



Contents of: IHz_model (only)

Column View: Block Data Types

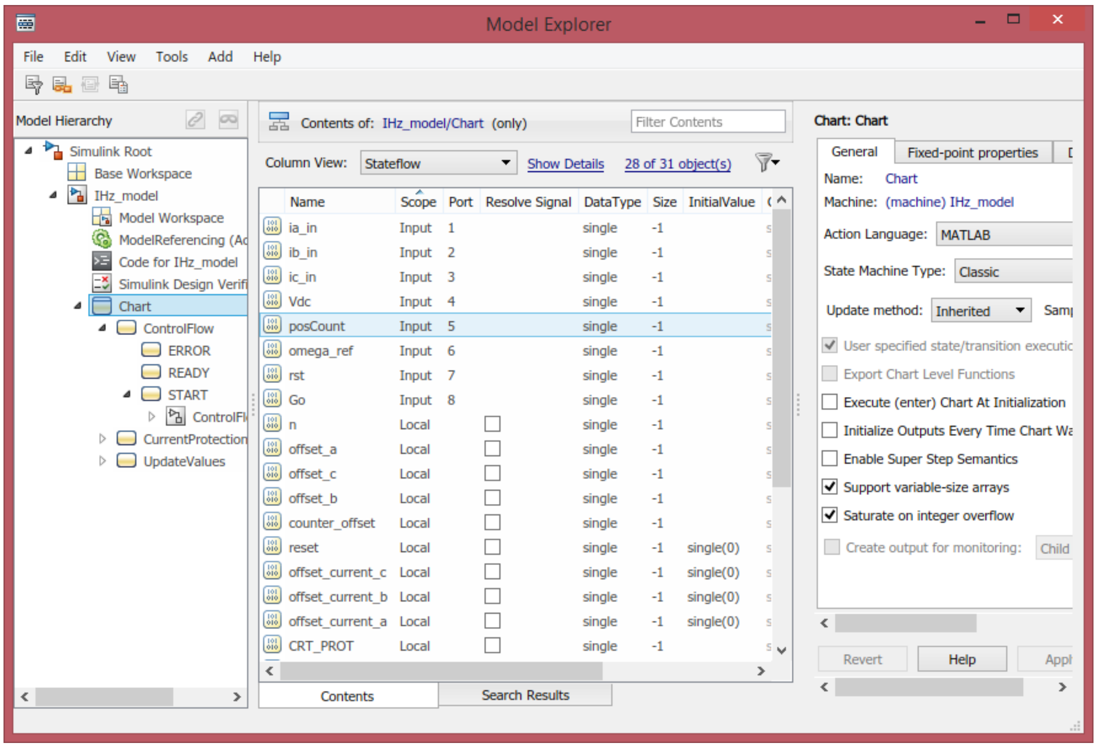
19 of 36 object(s)

Name	BlockType	OutDataTypeStr	OutMin	OutMax
Demux	Demux			
Demux1	Demux			
iabcs	Inport	single	[]	[]
Vdc	Inport	single	[]	[]
posCount	Inport	single	[]	[]
omega_ref_rpm	Inport	single	[]	[]
commands	Inport	single	[]	[]
Mux	Mux			
duty_abc	Outport	single	[]	[]
pwm_stop	Outport	single	[]	[]
ia	Outport	single	[]	[]
ib	Outport	single	[]	[]
ic	Outport	single	[]	[]
Model Workspace				
ModelReferencing (Active)				
Code for IHz_model				
Simulink Design Verifier results				
Chart				

Model callbacks: PreLoadFcn, PostLoadFcn, InitFcn*, StartFcn, PauseFcn, ContinueFcn, StopFcn, PreSaveFcn, PostSaveFcn, CloseFcn

Model init: init_sim

Figure 52: Model Explorer: IHz_model



Contents of: IHz_model/Chart (only)

Column View: Stateflow

28 of 31 object(s)

Name	Scope	Port	Resolve Signal	DataType	Size	InitialValue
ia_in	Input	1		single	-1	
ib_in	Input	2		single	-1	
ic_in	Input	3		single	-1	
Vdc	Input	4		single	-1	
posCount	Input	5		single	-1	
omega_ref	Input	6		single	-1	
rst	Input	7		single	-1	
Go	Input	8		single	-1	
n	Local		☐	single	-1	
offset_a	Local		☐	single	-1	
offset_c	Local		☐	single	-1	
offset_b	Local		☐	single	-1	
counter_offset	Local		☐	single	-1	
reset	Local		☐	single	-1	single(0)
offset_current_c	Local		☐	single	-1	single(0)
offset_current_b	Local		☐	single	-1	single(0)
offset_current_a	Local		☐	single	-1	single(0)
CRT_PROT	Local		☐	single	-1	

Chart: Chart

Name: Chart

Machine: (machine) IHz_model

Action Language: MATLAB

State Machine Type: Classic

Update method: Inherited

☑ User specified state/transition executio

☐ Export Chart Level Functions

☐ Execute (enter) Chart At Initialization

☐ Initialize Outputs Every Time Chart Wa

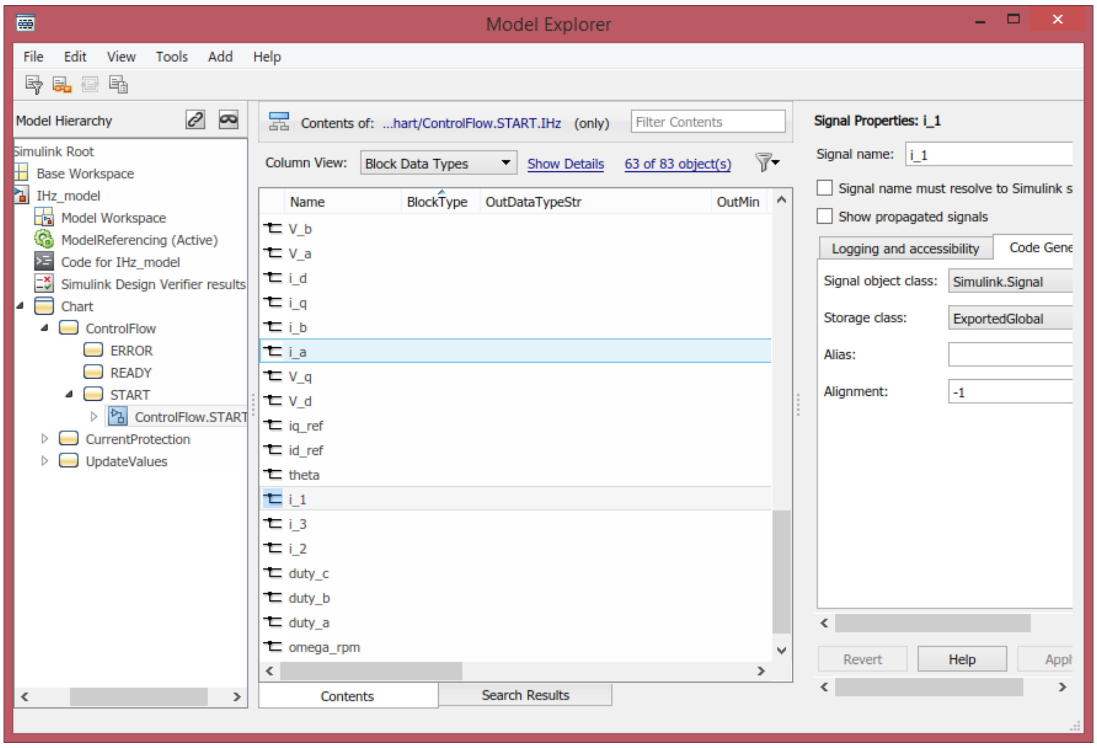
☐ Enable Super Step Semantics

☑ Support variable-size arrays

☑ Saturate on integer overflow

☐ Create output for monitoring: Child

Figure 53: Model Explorer: Chart



Model Hierarchy

- Simulink Root
  - Base Workspace
  - IHz_model
    - Model Workspace
    - ModelReferencing (Active)
    - Code for IHz_model
    - Simulink Design Verifier results
    - Chart
      - ControlFlow
        - ERROR
        - READY
        - START
          - ControlFlow.START
      - CurrentProtection
      - UpdateValues

Contents of: ...hart/ControlFlow.START.IHz (only)

Column View: Block Data Types   Show Details   63 of 83 object(s)

Name	BlockType	OutDataTypeStr	OutMin
V_b			
V_a			
i_d			
i_q			
i_b			
i_a			
V_q			
V_d			
iq_ref			
id_ref			
theta			
i_1			
i_3			
i_2			
duty_c			
duty_b			
duty_a			
omega_rpm			

Contents   Search Results

Signal Properties: i_1

Signal name: i_1

☐ Signal name must resolve to Simulink s

☐ Show propagated signals

Logging and accessibility   Code Gene

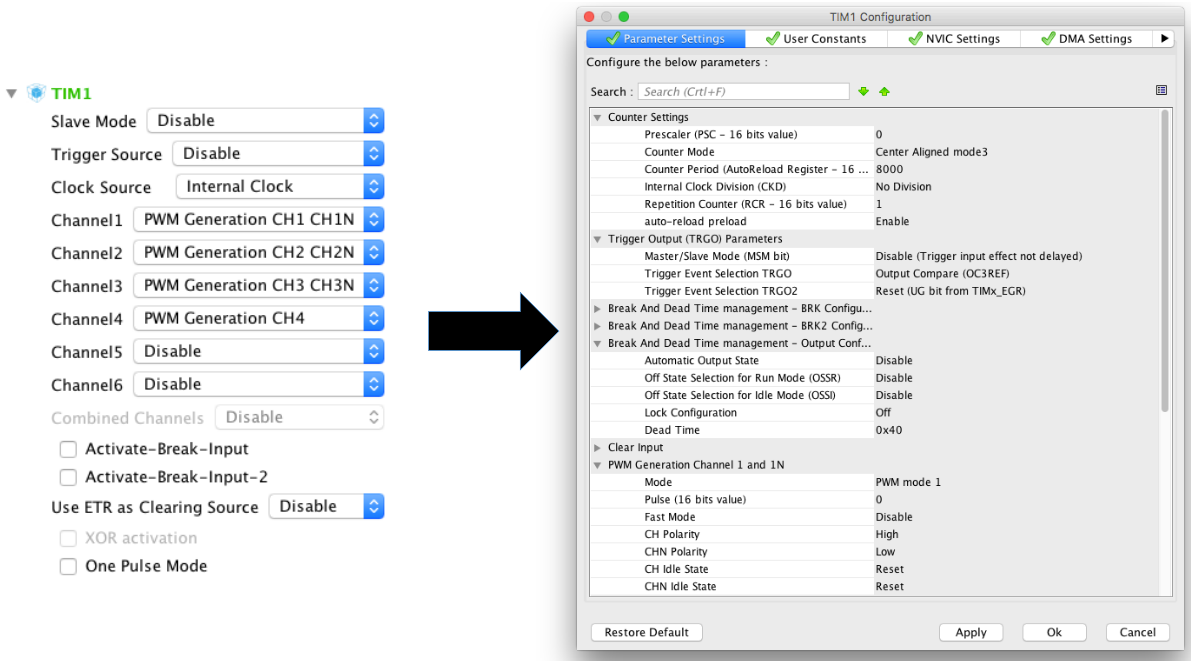
Signal object class:	Simulink.Signal
Storage class:	ExportedGlobal
Alias:	
Alignment:	-1

Revert   Help   Appl

Figure 54: Model Explorer: Simulink Function

A2 STM32CubeMX GUI for the peripherals configuration

Timer TIM1: Three channels and their negated channels (CHx and CHxN) are enabled for generating a PWM output. A fourth channel is only enabled to act as a trigger for the ADC conversion. The Counter Mode is Center Aligned mode 3 to generate an up-down counter. The Counter Period is set to 8000, so the timer frequency for the ISR is 64[Mhz]/2/8000 = 4 [kHz]. The Repetition Counter is set to 1 so the ISR only happens at the beginning of the up-counting. A dead time between the main channels and their negated channels is 0x40 = 64 pulses = 1 [μs]. The polarities of the CHx and CHxN channels are set to High and Low, respectively, to work properly with the gates of the inverter.



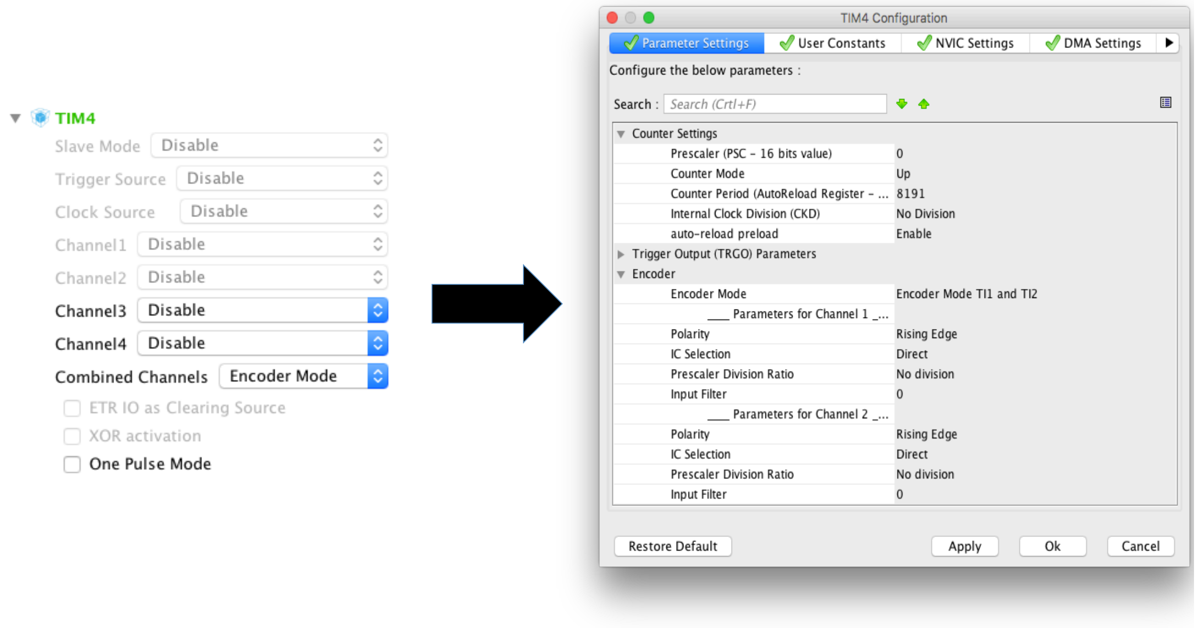
Slave Mode	Disable
Trigger Source	Disable
Clock Source	Internal Clock
Channel1	PWM Generation CH1 CH1N
Channel2	PWM Generation CH2 CH2N
Channel3	PWM Generation CH3 CH3N
Channel4	PWM Generation CH4
Channel5	Disable
Channel6	Disable
Combined Channels	Disable
☐ Activate-Break-Input	
☐ Activate-Break-Input-2	
Use ETR as Clearing Source	Disable
☐ XOR activation	
☐ One Pulse Mode	

Configure the below parameters :

Counter Settings	
Prescaler (PSC - 16 bits value)	0
Counter Mode	Center Aligned mode3
Counter Period (AutoReload Register - 16 ...	8000
Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - 16 bits value)	1
auto-reload preload	Enable
Trigger Output (TRGO) Parameters	
Master/Slave Mode (MSM bit)	Disable (Trigger input effect not delayed)
Trigger Event Selection TRGO	Output Compare (OC3REF)
Trigger Event Selection TRGO2	Reset (UG bit from TIMx_EGR)
Break And Dead Time management - BRK Configu...	
Break And Dead Time management - BRK2 Config...	
Break And Dead Time management - Output Conf...	
Automatic Output State	Disable
Off State Selection for Run Mode (OSSR)	Disable
Off State Selection for Idle Mode (OSSI)	Disable
Lock Configuration	Off
Dead Time	0x40
Clear Input	
PWM Generation Channel 1 and 1N	
Mode	PWM mode 1
Pulse (16 bits value)	0
Fast Mode	Disable
CH Polarity	High
CHN Polarity	Low
CH Idle State	Reset
CHN Idle State	Reset

Figure 55: TIM1 configuration

Timer TIM4: This timer operates in encoder mode, so it counts the encoder pulses from 0 to 8191 (Counter Period is set to 8191).


TIM4

Slave Mode	Disable
Trigger Source	Disable
Clock Source	Disable
Channel1	Disable
Channel2	Disable
Channel3	Disable
Channel4	Disable
Combined Channels	Encoder Mode

- [ ] ETR IO as Clearing Source
- [ ] XOR activation
- [ ] One Pulse Mode

TIM4 Configuration

Parameter Settings | User Constants | NVIC Settings | DMA Settings

Configure the below parameters :

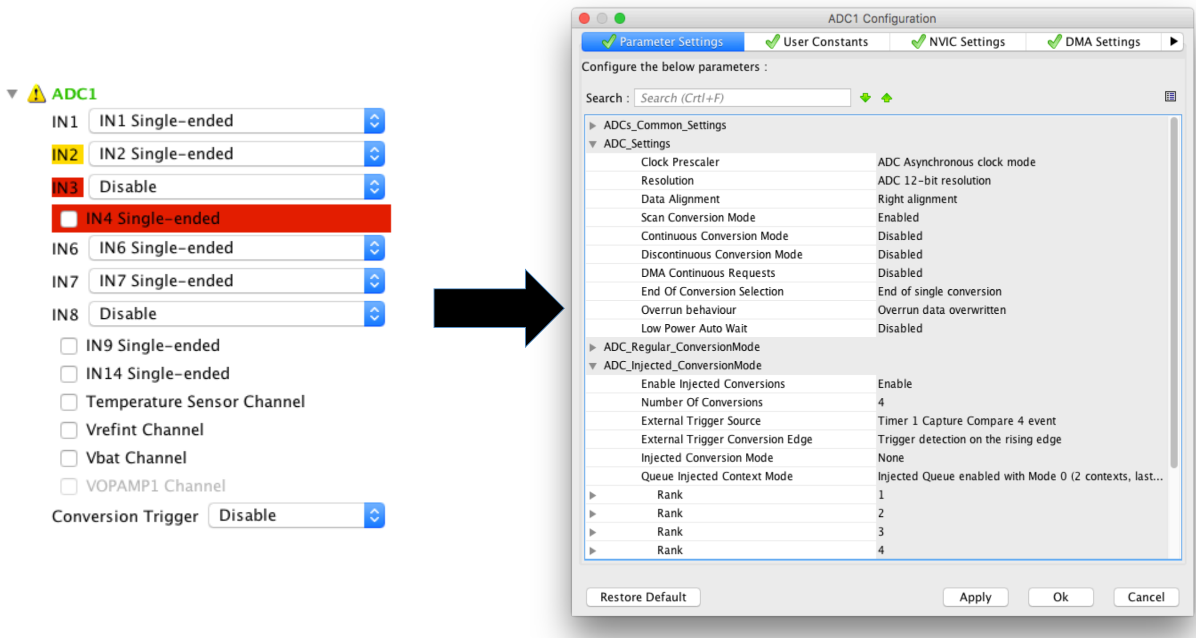
Search : Search (Crtl+F)

Counter Settings	
Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register - ...	8191
Internal Clock Division (CKD)	No Division
auto-reload preload	Enable
Trigger Output (TRGO) Parameters	
Encoder	
Encoder Mode	Encoder Mode TI1 and TI2
___ Parameters for Channel 1 _...	
Polarity	Rising Edge
IC Selection	Direct
Prescaler Division Ratio	No division
Input Filter	0
___ Parameters for Channel 2 _...	
Polarity	Rising Edge
IC Selection	Direct
Prescaler Division Ratio	No division
Input Filter	0

Restore Default | Apply | Ok | Cancel

Figure 56: TIM4 configuration

Analogue to Digital Converter ADC1: Only Injected Conversions are enabled and triggered by the Timer 1 Capture Compare 4 event (channel 4 of TIM1). According to the used inverter the ranking order is: channel 1 in rank 1, channel 7 in rank 2, channel 6 in rank 3 and channel 2 in rank 4.


ADC1

IN1	IN1 Single-ended
IN2	IN2 Single-ended
IN3	Disable
	IN4 Single-ended
IN6	IN6 Single-ended
IN7	IN7 Single-ended
IN8	Disable

- [ ] IN9 Single-ended
- [ ] IN14 Single-ended
- [ ] Temperature Sensor Channel
- [ ] Vrefint Channel
- [ ] Vbat Channel
- [ ] VOPAMP1 Channel

Conversion Trigger: Disable

ADC1 Configuration

Parameter Settings | User Constants | NVIC Settings | DMA Settings

Configure the below parameters :

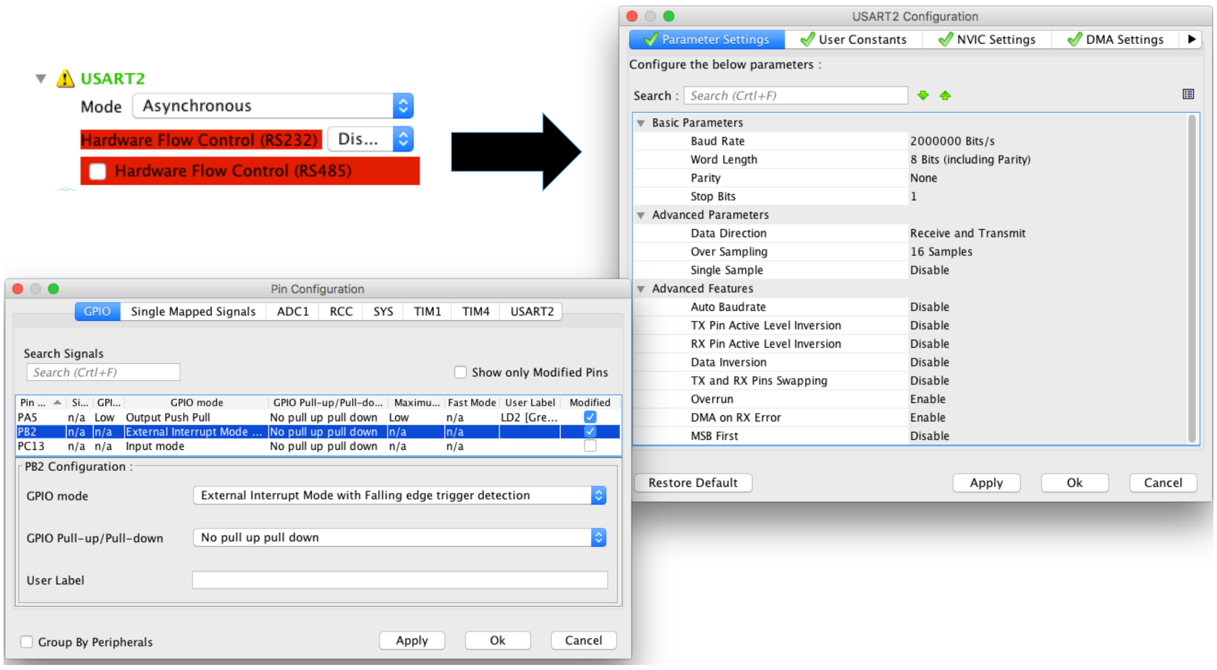
Search : Search (Crtl+F)

ADCs_Common_Settings	
ADC_Settings	
Clock Prescaler	ADC Asynchronous clock mode
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Scan Conversion Mode	Enabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	End of single conversion
Overrun behaviour	Overrun data overwritten
Low Power Auto Wait	Disabled
ADC_Regular_ConversionMode	
ADC_Injected_ConversionMode	
Enable Injected Conversions	Enable
Number Of Conversions	4
External Trigger Source	Timer 1 Capture Compare 4 event
External Trigger Conversion Edge	Trigger detection on the rising edge
Injected Conversion Mode	None
Queue Injected Context Mode	Injected Queue enabled with Mode 0 (2 contexts, last...
Rank	1
Rank	2
Rank	3
Rank	4

Restore Default | Apply | Ok | Cancel

Figure 57: ADC1 configuration

USART2 and pins: A 2,000,000 baud rate for asynchronous mode is set for the USART2. In the Pin configuration three pins are selected: an input for the *Go* push button, an output for a LED, and an External interrupt for the Z channel of the encoder to reset the counting of TIM4.



Basic Parameters	
Baud Rate	2000000 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples
Single Sample	Disable
Advanced Features	
Auto Baudrate	Disable
TX Pin Active Level Inversion	Disable
RX Pin Active Level Inversion	Disable
Data Inversion	Disable
TX and RX Pins Swapping	Disable
Overrun	Enable
DMA on RX Error	Enable
MSB First	Disable

Pin ...	Si...	GPI...	GPIO mode	GPIO Pull-up/Pull-do...	Maximu...	Fast Mode	User Label	Modified
PA5	n/a	Low	Output Push Pull	No pull up pull down	Low	n/a	LD2 [Gre...	✓
PB2	n/a	n/a	External Interrupt Mode ...	No pull up pull down	n/a	n/a		✓
PC13	n/a	n/a	Input mode	No pull up pull down	n/a	n/a		☐

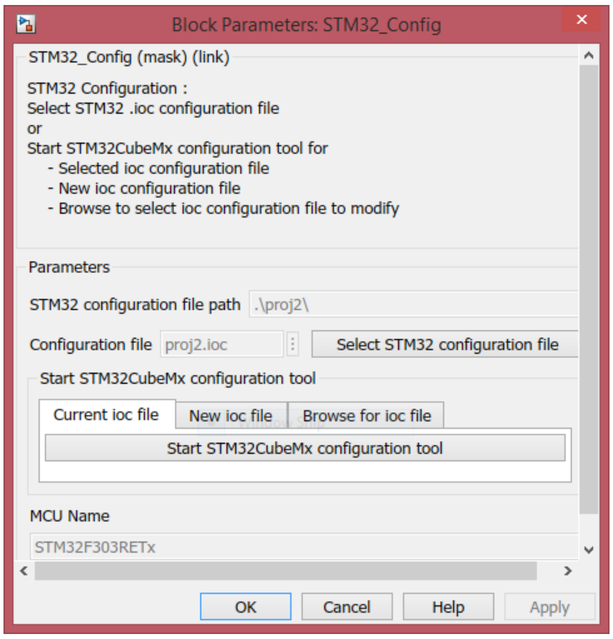
PB2 Configuration :

GPIO mode	External Interrupt Mode with Falling edge trigger detection
GPIO Pull-up/Pull-down	No pull up pull down
User Label	

Figure 58: USART2 and pin configuration

A3 Blocks configuration for the STM32 Embedded Target

STM32_Config block: The ioc file with the peripherals initialisation is located to enable their corresponding blocks in the Simulink Library Browser.


Block Parameters: STM32_Config
Select STM32 configuration file
Current ioc file
New ioc file
Browse for ioc file
Start STM32CubeMx configuration tool
OK
Cancel
Help
Apply

STM32_Config (mask) (link)

STM32 Configuration :
Select STM32 .ioc configuration file
or
Start STM32CubeMx configuration tool for
- Selected ioc configuration file
- New ioc configuration file
- Browse to select ioc configuration file to modify

Parameters

STM32 configuration file path .\proj2\

Configuration file proj2.ioc

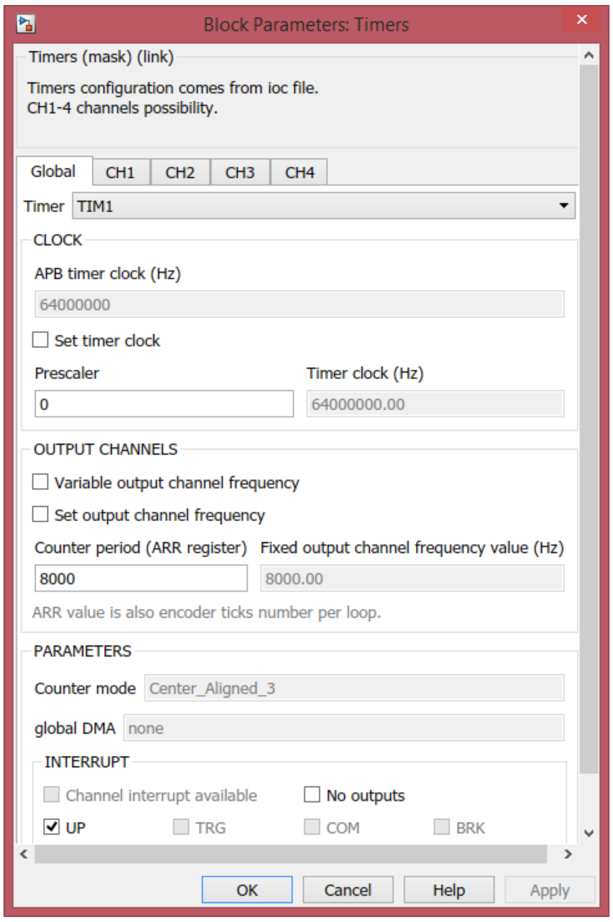
Start STM32CubeMx configuration tool

MCU Name

STM32F303RETx

Figure 59: STM32_Config

Timers block for the TIM1 interrupt signal: The Prescaler is set to 0 to avoid the source clock division and the Counter period is set to 8000 to compute a 4 [kHz] timer frequency, as explained in A2. The UP interrupt is enabled in order to trigger the timer ISR.


Block Parameters: Timers

Timers (mask) (link)

Timers configuration comes from ioc file.
CH1-4 channels possibility.

Global | CH1 | CH2 | CH3 | CH4

Timer: TIM1

CLOCK

APB timer clock (Hz): 64000000

☐ Set timer clock

Prescaler: 0
Timer clock (Hz): 64000000.00

OUTPUT CHANNELS

☐ Variable output channel frequency

☐ Set output channel frequency

Counter period (ARR register): 8000
Fixed output channel frequency value (Hz): 8000.00

ARR value is also encoder ticks number per loop.

PARAMETERS

Counter mode: Center_Aligned_3

global DMA: none

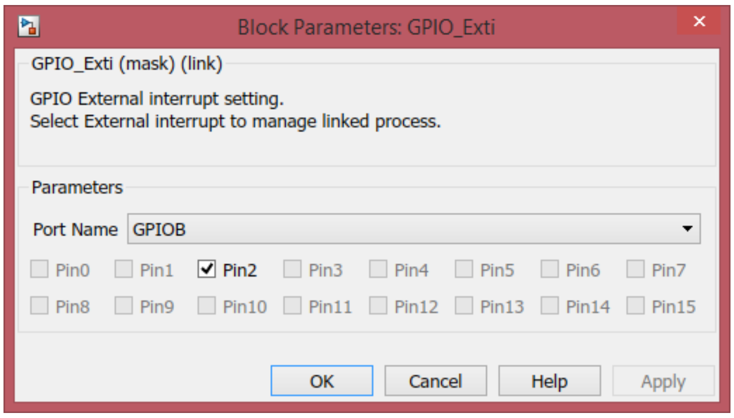
INTERRUPT

☐ Channel interrupt available ☐ No outputs

☑ UP ☐ TRG ☐ COM ☐ BRK

Figure 60: TIM1 interrupt signal

GPIO_Exti block: The Z channel port is selected, so the interrupt signal is generated as soon as it is detected in the corresponding physical port.


Block Parameters: GPIO_Exti

GPIO_Exti (mask) (link)

GPIO External interrupt setting.
Select External interrupt to manage linked process.

Parameters

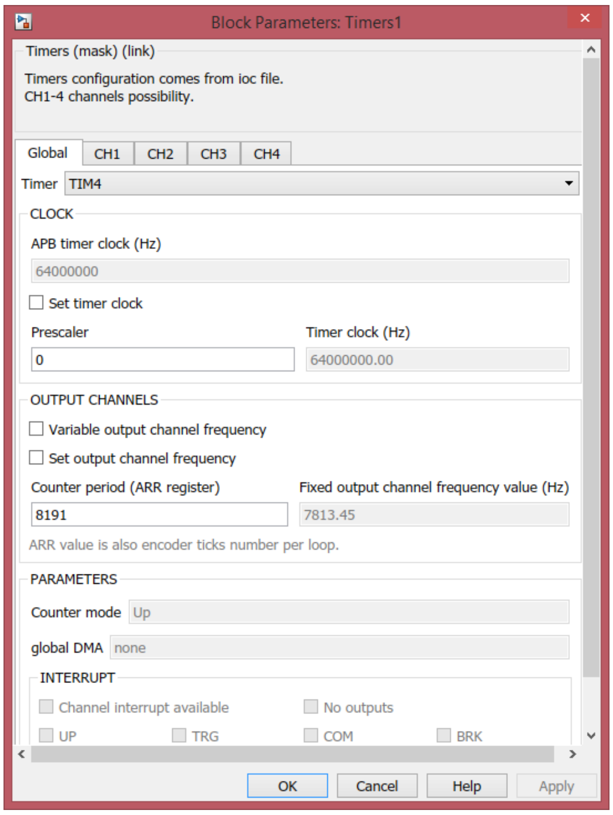
Port Name: GPIOB

☐ Pin0 ☐ Pin1 ☑ Pin2 ☐ Pin3 ☐ Pin4 ☐ Pin5 ☐ Pin6 ☐ Pin7

☐ Pin8 ☐ Pin9 ☐ Pin10 ☐ Pin11 ☐ Pin12 ☐ Pin13 ☐ Pin14 ☐ Pin15

Figure 61: GPIO_Exti block

Timers block for the TIM4: The Counter period corresponds to the number of pulses of the encoder in one revolution of the motor. In this case, 0 to 8191 pulses are counted.


Block Parameters: Timers1

Timers (mask) (link)

Timers configuration comes from ioc file.
CH1-4 channels possibility.

Global | CH1 | CH2 | CH3 | CH4

Timer: TIM4

CLOCK

APB timer clock (Hz): 64000000

[ ] Set timer clock

Prescaler: 0

Timer clock (Hz): 64000000.00

OUTPUT CHANNELS

[ ] Variable output channel frequency

[ ] Set output channel frequency

Counter period (ARR register): 8191

Fixed output channel frequency value (Hz): 7813.45

ARR value is also encoder ticks number per loop.

PARAMETERS

Counter mode: Up

global DMA: none

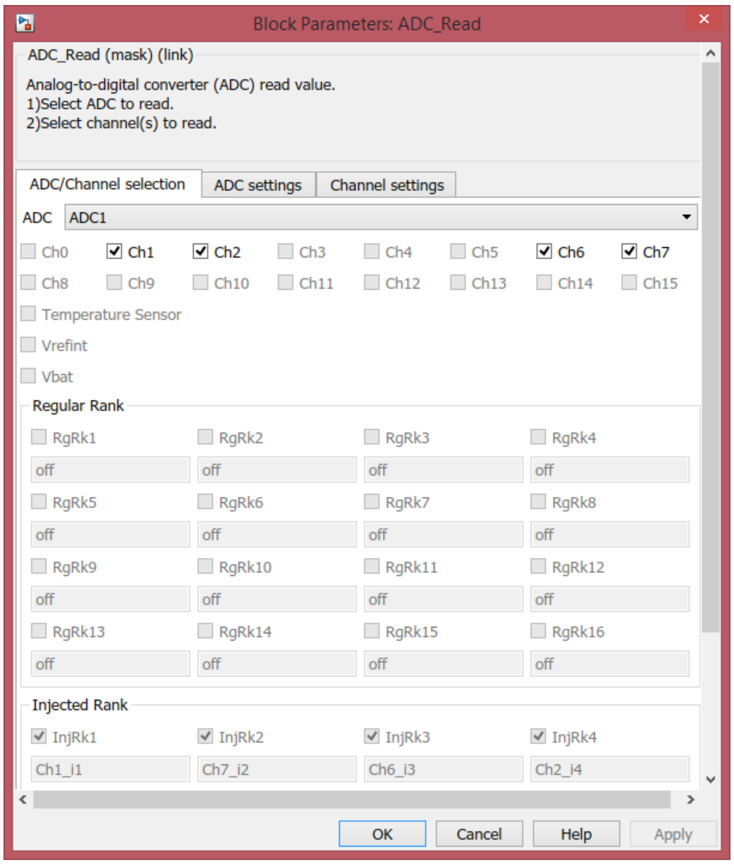
INTERRUPT

[ ] Channel interrupt available
[ ] No outputs
[ ] UP
[ ] TRG
[ ] COM
[ ] BRK

OK | Cancel | Help | Apply

Figure 62: TIM4 encoder mode

ADC_Read block: The injected channels are selected to be the outports of the block.


Block Parameters: ADC_Read

ADC_Read (mask) (link)

Analog-to-digital converter (ADC) read value.
1)Select ADC to read.
2)Select channel(s) to read.

ADC/Channel selection | ADC settings | Channel settings

ADC: ADC1

Ch0	✓ Ch1	✓ Ch2	Ch3	Ch4	Ch5	✓ Ch6	✓ Ch7
Ch8	Ch9	Ch10	Ch11	Ch12	Ch13	Ch14	Ch15

Temperature Sensor

Vrefint

Vbat

Regular Rank

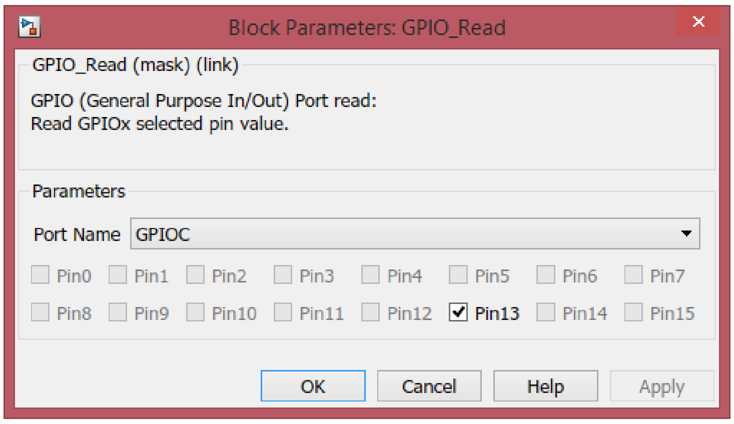
RgRk1	RgRk2	RgRk3	RgRk4
off	off	off	off
RgRk5	RgRk6	RgRk7	RgRk8
off	off	off	off
RgRk9	RgRk10	RgRk11	RgRk12
off	off	off	off
RgRk13	RgRk14	RgRk15	RgRk16
off	off	off	off

Injected Rank

✓ InjRk1	✓ InjRk2	✓ InjRk3	✓ InjRk4
Ch1_i1	Ch7_i2	Ch6_i3	Ch2_i4

Figure 63: ADC1

GPIO_Read block: The push button port is selected to detect the *Go* command.


Block Parameters: GPIO_Read

GPIO_Read (mask) (link)

GPIO (General Purpose In/Out) Port read:
Read GPIOx selected pin value.

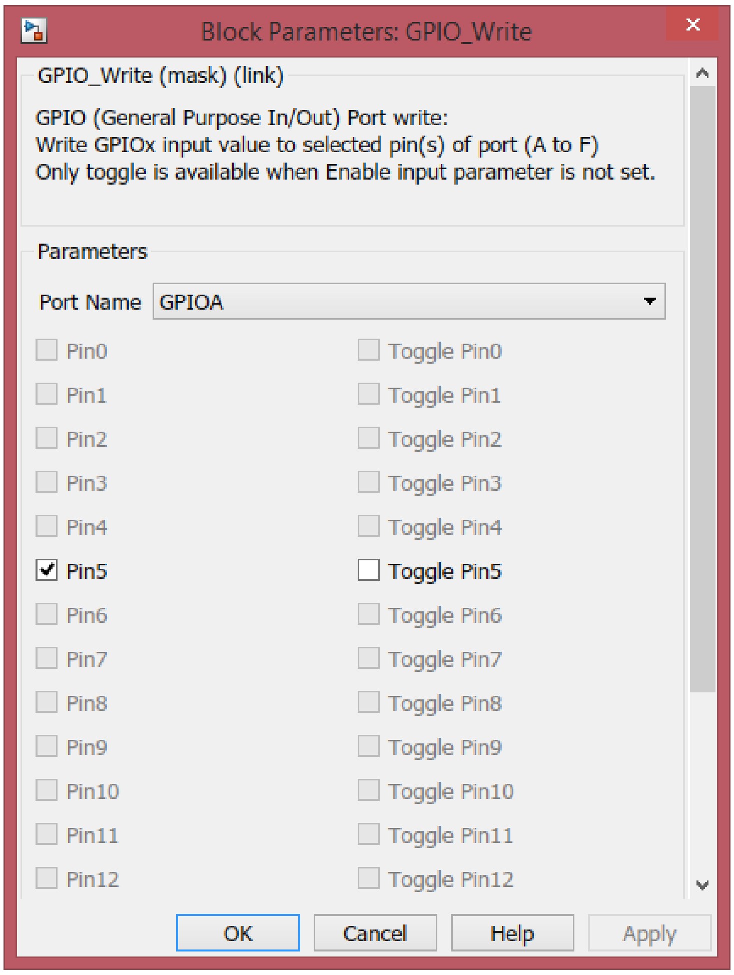
Parameters

Port Name: GPIOC

Pin0	Pin1	Pin2	Pin3	Pin4	Pin5	Pin6	Pin7
Pin8	Pin9	Pin10	Pin11	Pin12	✓ Pin13	Pin14	Pin15

Figure 64: GPIO_Read block

GPIO_Write block: The LED port is selected to visualise the binary state of the *Go* command.


Block Parameters: GPIO_Write

GPIO_Write (mask) (link)

GPIO (General Purpose In/Out) Port write:
Write GPIOx input value to selected pin(s) of port (A to F)
Only toggle is available when Enable input parameter is not set.

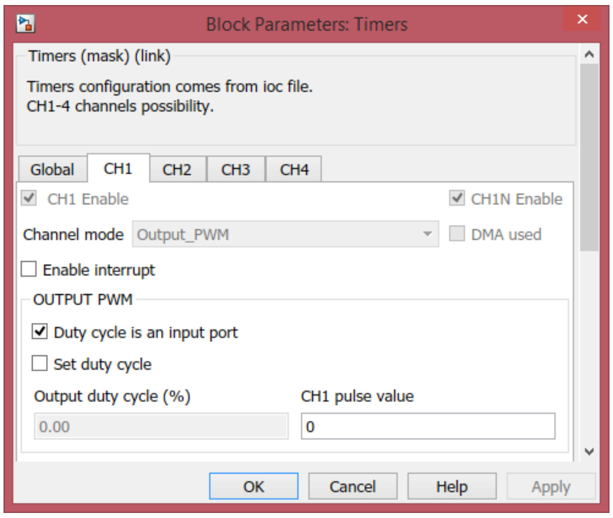
Parameters

Port Name: GPIOA

☐ Pin0	☐ Toggle Pin0
☐ Pin1	☐ Toggle Pin1
☐ Pin2	☐ Toggle Pin2
☐ Pin3	☐ Toggle Pin3
☐ Pin4	☐ Toggle Pin4
☑ Pin5	☐ Toggle Pin5
☐ Pin6	☐ Toggle Pin6
☐ Pin7	☐ Toggle Pin7
☐ Pin8	☐ Toggle Pin8
☐ Pin9	☐ Toggle Pin9
☐ Pin10	☐ Toggle Pin10
☐ Pin11	☐ Toggle Pin11
☐ Pin12	☐ Toggle Pin12

Figure 65: GPIO_Write block

Timers block for the TIM1 PWM ports: It is configured to receive the duty cycles in each of its channels and generate the PWM signals in the corresponding physical ports.


Block Parameters: Timers

Timers (mask) (link)

Timers configuration comes from ioc file.
CH1-4 channels possibility.

☑ CH1 Enable ☑ CH1N Enable

Channel mode: Output_PWM ☐ DMA used

☐ Enable interrupt

OUTPUT PWM

☑ Duty cycle is an input port

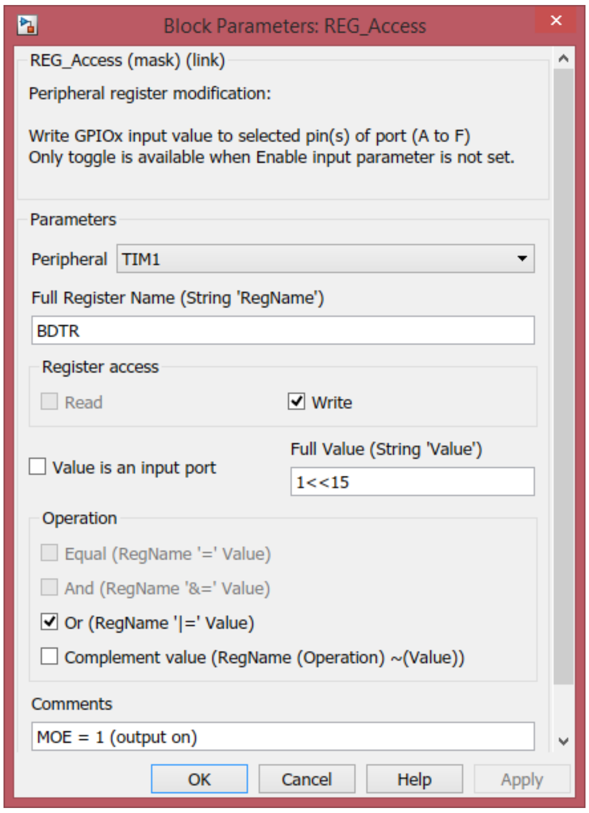
☐ Set duty cycle

Output duty cycle (%): 0.00

CH1 pulse value: 0

Figure 66: TIM1 PWM ports

REGISTER_Access blocks: Certain registers can be modified. For this case, the BDTR register of the TIM1 is accessed to turn ON or OFF the PWM ports.



REG_Access (mask) (link)

Peripheral register modification:

Write GPIOx input value to selected pin(s) of port (A to F)
Only toggle is available when Enable input parameter is not set.

Parameters

Peripheral: TIM1

Full Register Name (String 'RegName'): BDTR

Register access: Read [ ] Write [✓]

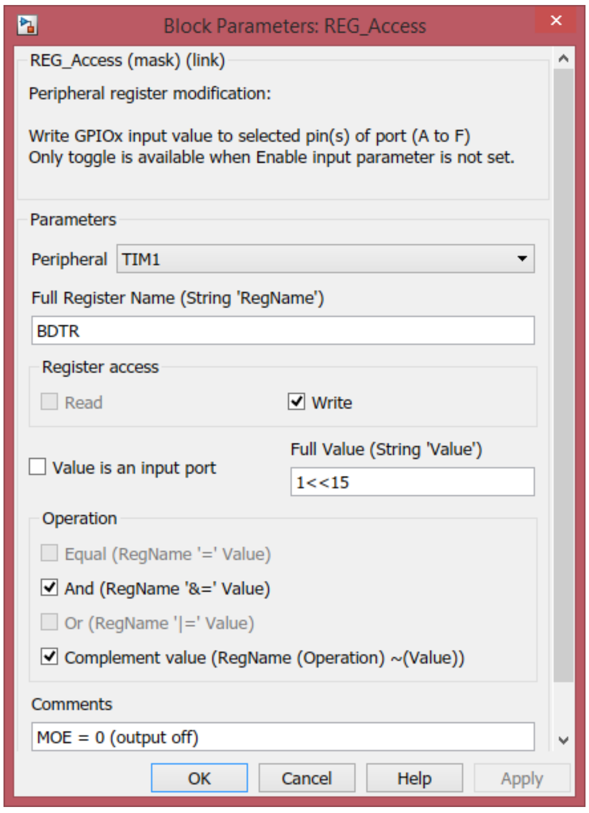
Value is an input port [ ]

Full Value (String 'Value'): 1<<15

Operation:
- [ ] Equal (RegName '=' Value)
- [ ] And (RegName '&=' Value)
- [✓] Or (RegName '|=' Value)
- [ ] Complement value (RegName (Operation) ~(Value))

Comments: MOE = 1 (output on)

Figure 67: TIM1 PWM ON



REG_Access (mask) (link)

Peripheral register modification:

Write GPIOx input value to selected pin(s) of port (A to F)
Only toggle is available when Enable input parameter is not set.

Parameters

Peripheral: TIM1

Full Register Name (String 'RegName'): BDTR

Register access: Read [ ] Write [✓]

Value is an input port [ ]

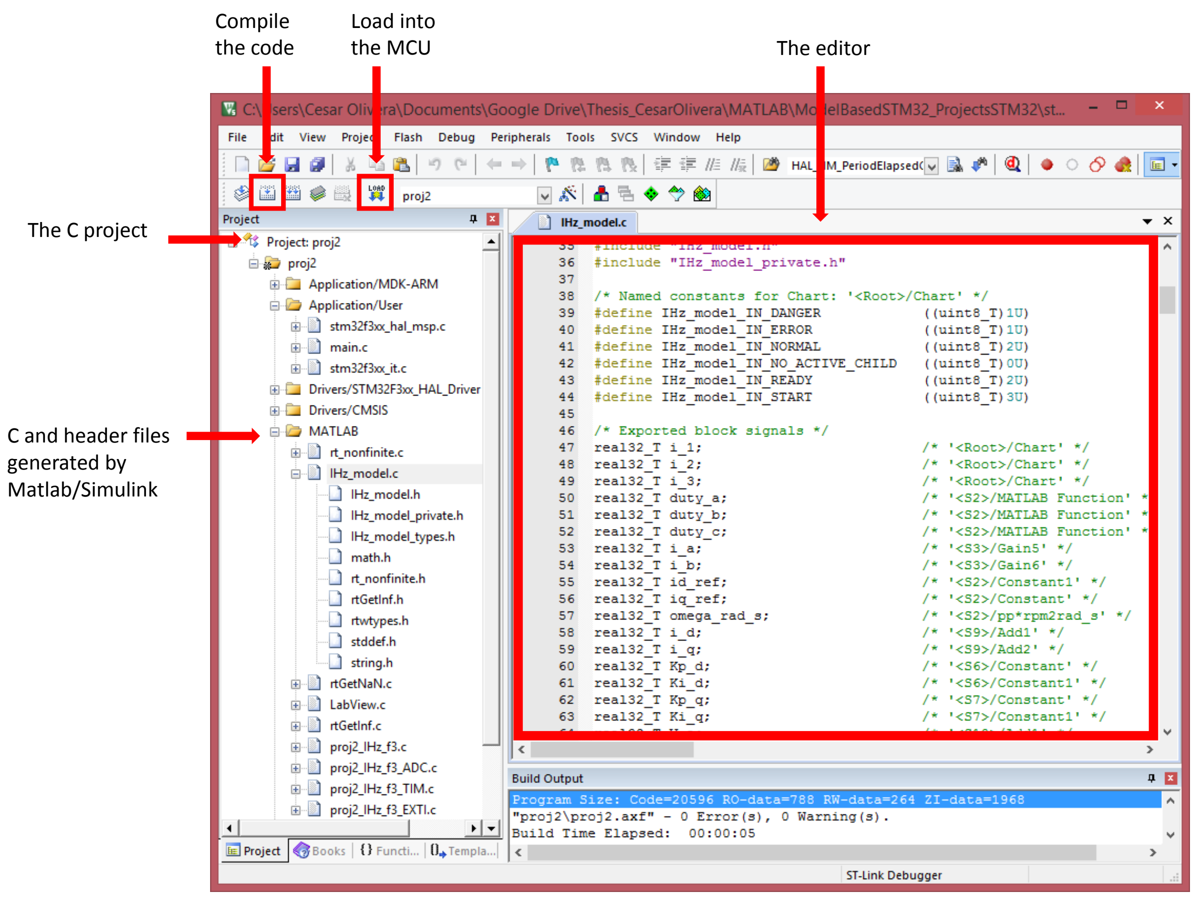
Full Value (String 'Value'): 1<<15

Operation:
- [ ] Equal (RegName '=' Value)
- [✓] And (RegName '&=' Value)
- [ ] Or (RegName '|=' Value)
- [✓] Complement value (RegName (Operation) ~(Value))

Comments: MOE = 0 (output off)

Figure 68: TIM1 PWM OFF

A4 KEIL uVision IDE



```
#include "IHz_model_private.h"

/* Named constants for Chart: '<Root>/Chart' */
#define IHz_model_IN_DANGER                 ((uint8_T)1U)
#define IHz_model_IN_ERROR                  ((uint8_T)1U)
#define IHz_model_IN_NORMAL                 ((uint8_T)2U)
#define IHz_model_IN_NO_ACTIVE_CHILD        ((uint8_T)0U)
#define IHz_model_IN_READY                  ((uint8_T)2U)
#define IHz_model_IN_START                  ((uint8_T)3U)

/* Exported block signals */
real32_T i_1;                              /* '<Root>/Chart' */
real32_T i_2;                              /* '<Root>/Chart' */
real32_T i_3;                              /* '<Root>/Chart' */
real32_T duty_a;                           /* '<S2>/MATLAB Function' *
real32_T duty_b;                           /* '<S2>/MATLAB Function' *
real32_T duty_c;                           /* '<S2>/MATLAB Function' *
real32_T i_a;                              /* '<S3>/Gain5' */
real32_T i_b;                              /* '<S3>/Gain6' */
real32_T id_ref;                           /* '<S2>/Constant1' */
real32_T iq_ref;                           /* '<S2>/Constant' */
real32_T omega_rad_s;                      /* '<S2>/pp*rpm2rad_s' */
real32_T i_d;                              /* '<S9>/Add1' */
real32_T i_q;                              /* '<S9>/Add2' */
real32_T Kp_d;                             /* '<S6>/Constant' */
real32_T Ki_d;                             /* '<S6>/Constant1' */
real32_T Kp_q;                             /* '<S7>/Constant' */
real32_T Ki_q;                             /* '<S7>/Constant1' */
```

```
Program Size: Code=20596 RO-data=788 RW-data=264 ZI-data=1968
"proj2\proj2.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:05
```

Figure 69: C project in KEIL uVision IDE