Integrating custom FOC model from MATLAB® / Simulink® into PSOC™ Control C3 project

About this document

Scope and purpose

This application note explains how to export a custom Field-Oriented Control (FOC) model from Simulink® and integrate it into a Motor Control Driver Interface (MCDI) generated project template for use with the PSOC™ Control C3 microcontroller. Throughout the document, critical points regarding the use of these tools are highlighted, along with detailed instructions for the key steps necessary to ensure successful integration. This is demonstrated by providing an example project featuring a complete motor control setup targeted for the PSC3M5 MC1 Kit.

Intended audience

This document is intended for anyone who uses the PSOC™ Control C3 microcontroller.

Table of contents

About this document . . . 1
Table of contents . . . 2
1 Introduction . . . 3
1.1 Hardware setup . . . 5
1.2 Software setup and program files . . . 5
2 MATLAB®/Simulink® model design . . . 7
3 MATLAB®/Simulink® embedded coder - code generation . . . 14
3.1 Usage of Simulink® coder blocks . . . 14
3.2 Usage of Embedded Coder Application . . . 17
4 ModusToolbox™ Motor Control Driver Interface (MCDI) . . . 22
4.1 Creation of Empty Project in ModusToolbox™ . . . 22
4.2 Configuration of Motor Control Driver Interface (MCDI) . . . 24
5 ModusToolbox™ Motor Suite GUI . . . 27
6 Application . . . 30
6.1 Preparing the application . . . 30
6.2 Using the ready-to-flash ModusToolbox™ project . . . 31
7 Testing . . . 33
8 Simulation results . . . 34
References . . . 36
Revision history . . . 37
Trademarks . . . 38
Disclaimer . . . 39

1 Introduction

This document guides on how to export a custom Field-Oriented Control (FOC) model from Simulink® for integration into ModusToolbox™ to be used on PSOC™ Control C3 microcontroller.

Simulink® provides a model-based design (MBD) functionality, enabling graphical modeling through various built-in blocks [1]. Additionally, it also allows for generating C code from the simulation model for real-world applications [2].

The process involves the use of two automation tools: Embedded Coder and Motor Control Driver Interface (MCDI). These tools enable the generation of code for the microcontroller.

Conceptually, ModusToolbox™ Motor Control Driver Interface (MCDI) generates a code base that handles all the essential tasks, such as hardware initialization (clocks, peripherals, etc.), and software framework creation (interrupt service routine calls, variable definitions, etc.). This basically generates the skeleton code without the control algorithm. On top of this middleware layer, Simulink® generated code (using Embedded Coder) is added to implement the control algorithm. Figure 1 illustrates the conceptual structure.

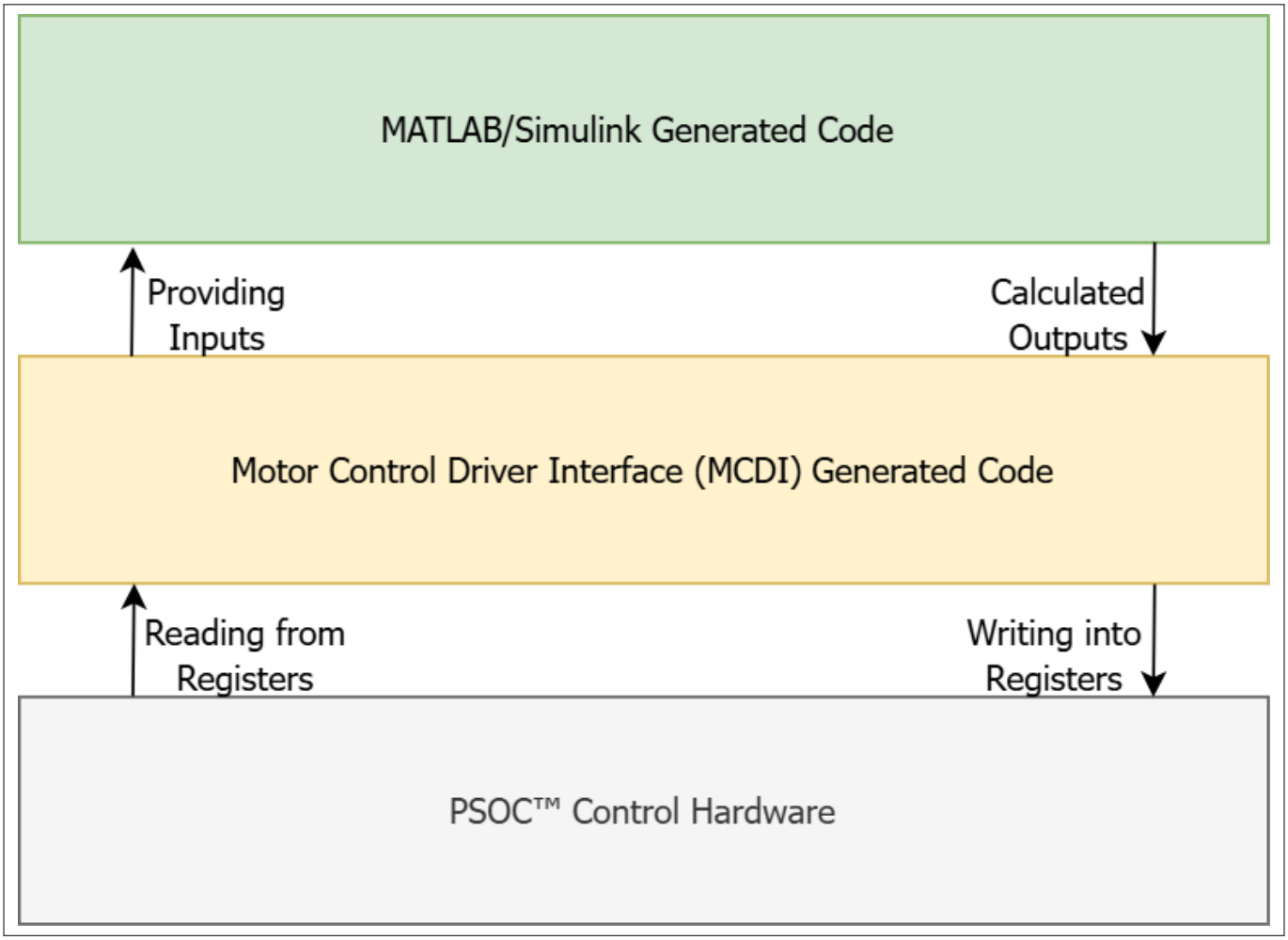

Figure 1 Application block diagram

The workflow is shown in Figure 2. The process starts with designing a custom Simulink® model for a PMSM drive from scratch. To save time, a pre-designed FOC Simulink® model is included with this document, allowing you to skip the design phase. This model serves as a practical reference for how the modeling should be performed. The provided model is specifically designed for the hardware setup described in Section Hardware setup.

The next step is simulating the designed model and analyze the results. The scope block in Simulink® can be used to monitor the signals and observe the system behavior during simulation. If the simulation results are

generated as expected, then the process proceeds with code generation using Embedded Coder. Otherwise, the design must be reviewed. However, the provided Simulink® model is pre-configured, and functions as intended, allowing it to be directly simulated without modifications. Detailed explanations of the key aspects of the Simulink® model design and the use of Embedded Coder are provided in Sections MATLAB®/Simulink® model design and MATLAB®/Simulink® embedded coder - code generation , respectively.

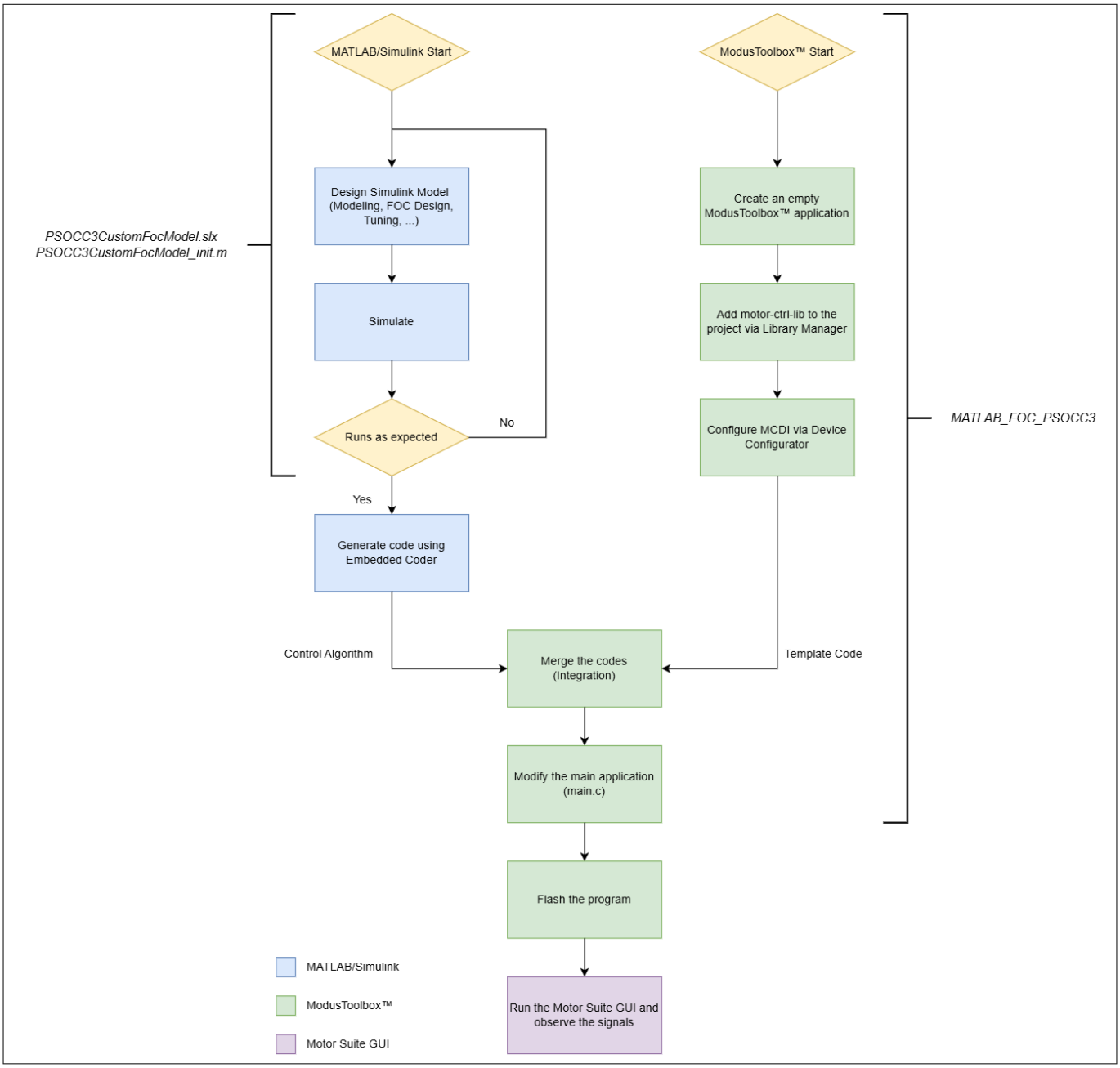

Figure 2 Workflow

To begin the code integration process,

1. Create an empty ModusToolbox™ application first
2. Enable the use of MCDI functionality by adding `motor-ctrl-lib` to the project
3. Configure the MCDI solution according to the designed controller within the simulation model and the hardware setup

A detailed instructions for performing these operations are provided in Section ModusToolbox™ Motor Control Driver Interface (MCDI). A ready-to-flash ModusToolbox™ project is also provided, which included an MCDI-configured solution for the simulation model and the hardware setup in Section Hardware setup.

1.1 Hardware setup

In this example project, the PSOC™ Control C3M5 Motor Drive Control Card [4] is used, with the adapter and power cards attached. The motor used in this demonstration is a DB42M03 Permanent Magnet Synchronous Motor (PMSM) model. Figure 3 illustrates the detailed hardware setup employed in this demonstration. The system is powered up with a current limited to a specific level using an external power supply.

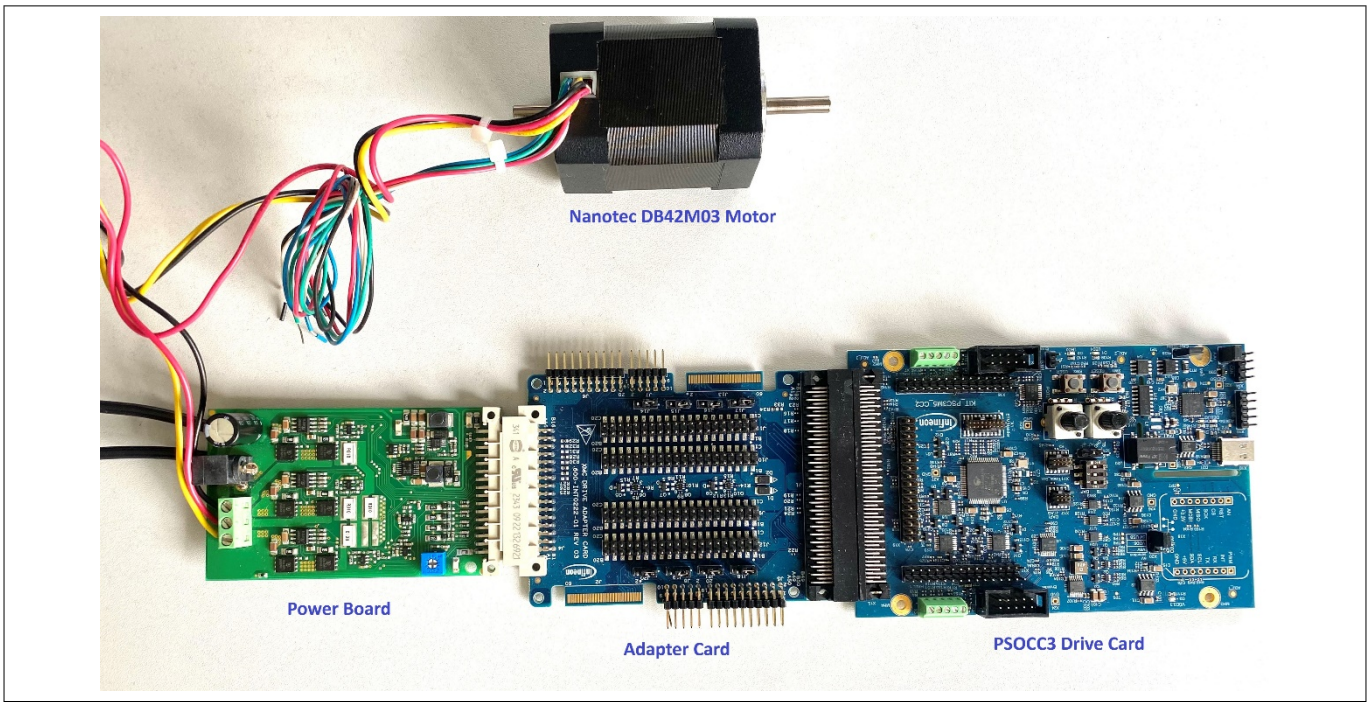

Figure 3 Hardware setup

1.2 Software setup and program files

This example project does not use any default Simulink® project examples provided by MathWorks. Instead, the document is provided with two files and a folder, as explained in Section MATLAB®/Simulink® model design.

The `PSOCC3CustomFocModel.slx` is a Simulink® file that contains the designed custom field-oriented controller (FOC) and the `PSOCC3CustomFocModel_init.m` script file is used to initialize the parameters within the simulation. The script file must be executed once before running the simulation to initialize the workspace. This `.m` file can also be used to change the values of the defined parameters. With the provided hardware setup, the `PSOCC3CustomFocModel.slx` file can directly be used for code generation without any modification, enabling seamless functioning. The Simulink® project has been tested using MATLAB® version R2023b.

On the embedded side, the project application is developed using the Eclipse IDE for ModusToolbox™ 3.4. Initially, an empty application is created, which is then followed by the configuration of MCDI. A detailed explanation of this process can be found in Section ModusToolbox™ Motor Control Driver Interface (MCDI).

The `MATLAB_FOC_PSOCC3` folder contains the ready-to-flash ModusToolbox™ project, representing the final version of the software to be flashed onto the hardware. This project allows users to bypass all the detailed steps outlined in this document and proceed directly to deployment and testing. Instructions for importing this project into ModusToolbox™ is provided in Using the ready-to-flash ModusToolbox™ project.

1 Introduction

These provided files (`PSOCC3CustomFocModel.slx`, `PSOCC3CustomFocModel_init.m`, `MATLAB_FOC_PSOCC3`) contain the work completed for the specified blocks in the flowchart as shown in Figure 2 and should be used to bypass the work associated with indicated blocks.

2 MATLAB®/Simulink® model design

MathWorks provides a Simulink® model example for PMSM Drive [4], as shown in Figure 4. This model is referred to as the *original Simulink®* model throughout this section to distinguish it from our *custom FOC model*. In this section, the *original Simulink®* model is compared with our *custom FOC model* to identify design requirements. Several key design considerations must be addressed to ensure proper abstraction during modeling.

The input signals for the controller subsystem (within the *original Simulink®* model) include current values (in amps), the reference speed value (in RPM), and position measurements obtained from the encoder subsystem. In real-world, the ADC hardware measures currents with 12-bit quantization. For this reason, our *custom FOC model* includes a simple ADC hardware model, enabling the model to accept inputs as 12-bit integers. The outputs of the controller subsystem are obtained directly from the PWM signals. However, this is abstracted in our *custom FOC model*, as in the real-world system, the PWM signals are generated by the TCPWM hardware rather than by the code itself. To account for this, a simple TCPWM block is added to the model , makes the outputs of the controller subsystem integer compare values that are written into the TCPWM registers.

The *original Simulink®* model implements an encoder-based control algorithm. In contrast, our *custom FOC model* uses a sensorless FOC algorithm. Therefore, the controller subsystem does not require position information as an input signal. Instead, the controller subsystem calculates the position using the measured 3-phase current. In our *custom FOC model*, the observer block within the controller subsystem is responsible for these position-based calculations.

A notable addition in our *custom FOC model* is the state machine block, which is absent in the *original Simulink®* model. This block manages the control states, starting with open-loop control followed by closed-loop control. The states are designed to allow the control to begin immediately after power-up. The transition from open-loop to closed-loop control occurs once the motor reaches 400 RPM. Afterward, the speed is adjusted using the potentiometer. Note that the minimum adjustable speed is 400 RPM.

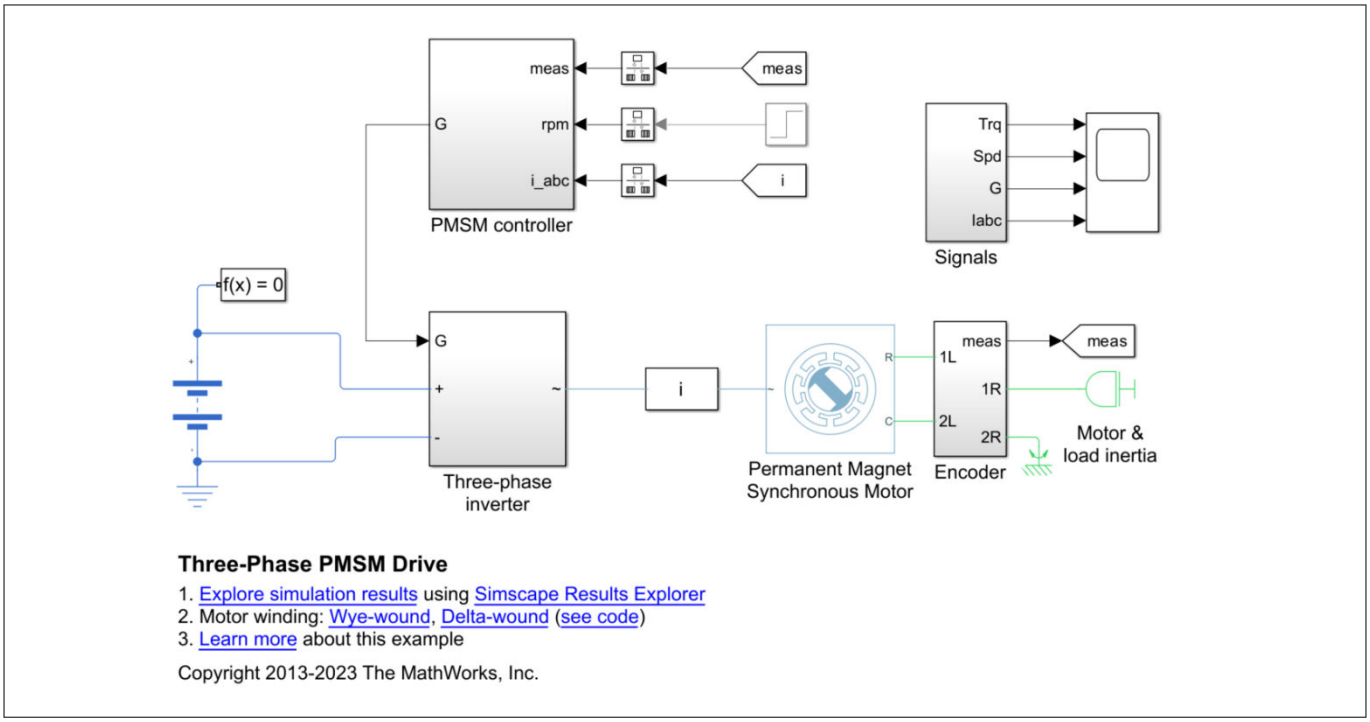

Figure 4 Original Simulink® model for PMSM drive

Our *custom FOC model* demonstrates the operation of a basic field-oriented controller that is used to drive a PMSM motor. A high-level representation of this model is shown in Figure 5.The blocks highlighted by blue correspond to components related to the microcontroller. Key details regarding these microcontroller-specific blocks are discussed in this section. Meanwhile, the remaining blocks in the model represent the inverter hardware, the PMSM motor, and the attached load. These blocks are pre-configured to match the hardware

setup described in Section Hardware setup. However, for users utilizing a custom hardware setup, it is essential to reconfigure these external blocks to align with the specific characteristics of their hardware system.

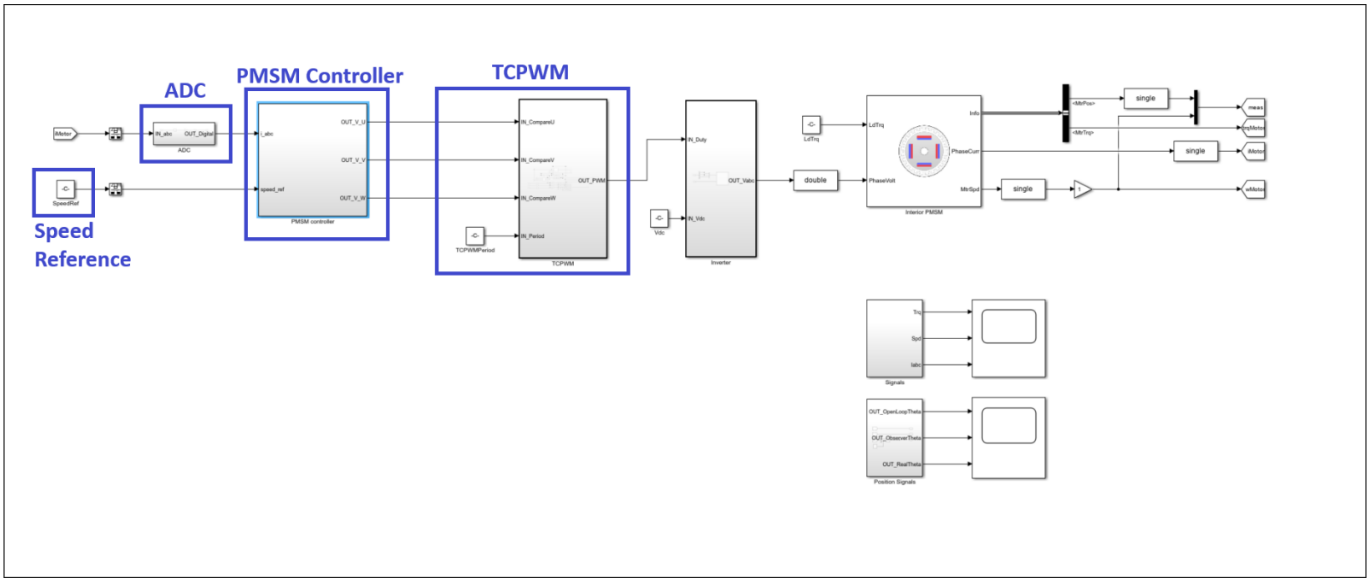

Figure 5 Custom Simulink® model (`PSOCC3CustomFocModel.slx`)

The ADC block roughly models the ADC hardware peripheral. It converts the three-phase current values in amperes to 12-bit quantized integers representing raw ADC measurements. There are two important parameters inside this block which need to be taken into account. The first parameter is the quantization interval of the current values. This represents the resolution of the ADC hardware and mostly depends on the external scaling hardware. The second parameter represents the 0 V bias of the scaling hardware. For a 12-bit ADC, the 0 V bias is ideally equal to 2048. However, it might also change depending on the hardware.

To simplify the configuration of the ADC block, two workspace variables `params.sys.adc.quant`, `params.sys.adc.bias` are defined in `PSOCC3CustomFocModel_init.m` script file. The `params.sys.adc.quant` variable initializes the quantization interval parameter in both quantizer and the gain block. The `params.sys.adc.bias` variable initializes the constant block that adds up the bias.

The constant block represents the ADC measurements from the potentiometer and its value must be between 0 and 4096. This value is used to adjust the reference speed, which can be set between 400 and 2000 RPM. The 12-bit integer range is scaled to match to this speed interval inside the controller subsystem.

The TCPWM block takes integer compare values and a period value as input and generates the corresponding PWM signals. This block is specifically designed to generate symmetric PWM signals for which the counter operates in *Up and Down* counting mode. This configuration should align with the MCDI configuration, which is described in Section Configuration of Motor Control Driver Interface (MCDI). Alternatively, MATLAB® offers built-in blocks that can serve as replacements for the TCPWM block. However, it is important to keep in mind that the algorithm's final outputs consist of integer compare values corresponding to the 3-phases, which are directly written into the TCPWM registers. As a result, additional conversion steps may be required when using alternative blocks to ensure proper compatibility with the system.

Sampling time of TCPWM block in the simulation must be consistent with the clock configuration in the MCDI tool. In other words, this value must match the actual clocking frequency of the hardware counter in the real world. This can be adjusted by right clicking the subsystem block and clicking on block parameters. If the "Treat as atomic unit" checkbox is checked, the sample time can be adjusted directly by entering a value in the textbox. For this reason, a workspace variable `params.sys.tcpwm.f_tcpwm` is created in the script file to keep the clocking frequency of the TCPWM counter. The "Sample time" of this block is assigned as `params.sys.tcpwm.t_tcpwm` variable, which is the reciprocal of `params.sys.tcpwm.f_tcpwm`. It should be noted that increasing this parameter slows down the simulation. Although reducing this parameter speeds up the simulation, it may result in undesired drawbacks.

One of the key inputs required by the TCPWM block is the period value, which defines the maximum count the counter reaches during its operation. This parameter also plays a critical role in determining the resolution of the generated PWM signals. In symmetric mode, the period value is calculated by the following formula:

$$Period = \frac{f_{TCPWM}}{2 \times f_{control}} \quad (1)$$

Period value

***Note**: It is crucial to note that the code is going to be generated only for the PMSM Controller subsystem block, since the ADC and TCPWM blocks represent the model of existing hardware.*

Within the simulation, this value is supplied to the TCPWM subsystem block through an external constant block. This constant block is initialized using the expression: `params.sys.tcpwm.f_tcpwm/(params.ctrl.f_c*2)`, ensuring the correct configuration of the TCPWM block for the desired operation.

Figure 6 shows the internal functional architecture of the PMSM Controller subsystem. There are a few important steps that need to be taken to ensure code compatibility during code integration. The most prominent steps are the passing of the measured ADC values to the control algorithm as inputs and writing the output compare values to the corresponding registers of TCPWM peripheral.

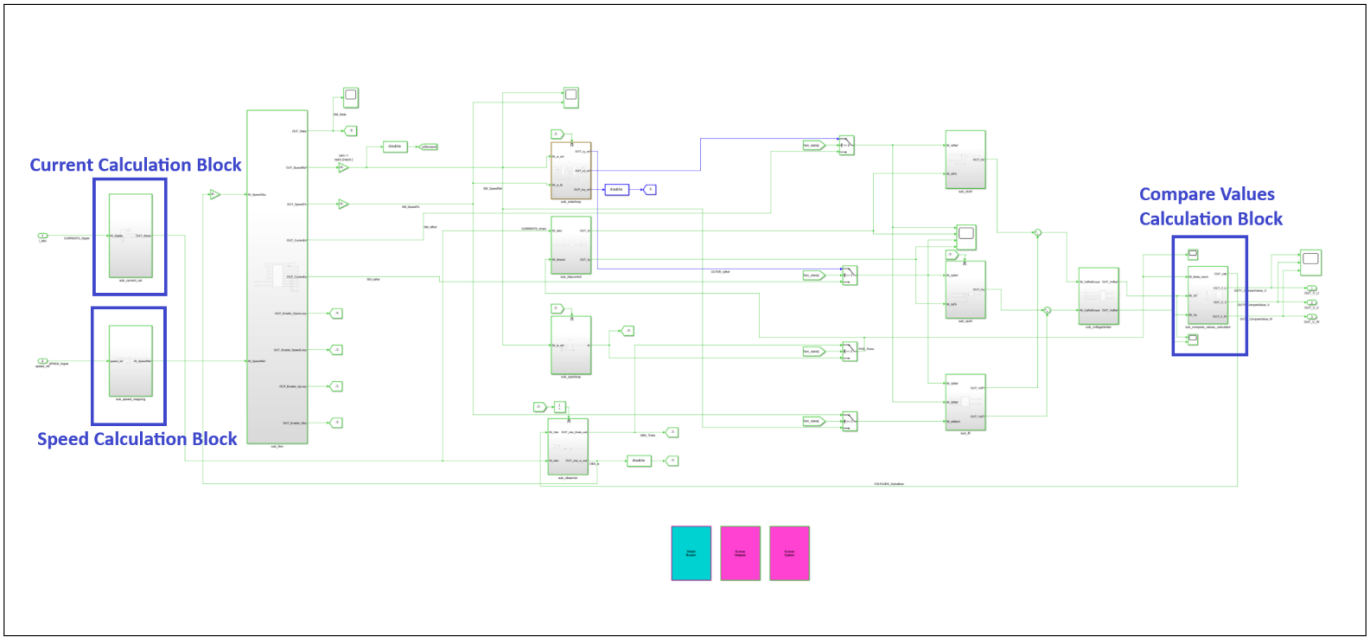

Figure 6 Block diagram of the PMSM Controller Subsystem (*PSOCC3CustomFocModel.slx*)

ModusToolbox™ MCDI provides the 3-phase current measurements in a signed 16-bit integer array. There must be one block within the controller subsystem to scale the raw ADC current values to amperes for further use. This block is indicated by the upper left blue box in Figure 6. This configuration of this scaling operation depends on the external hardware, and the given block is configured for the MC1 Motor Control Kit.

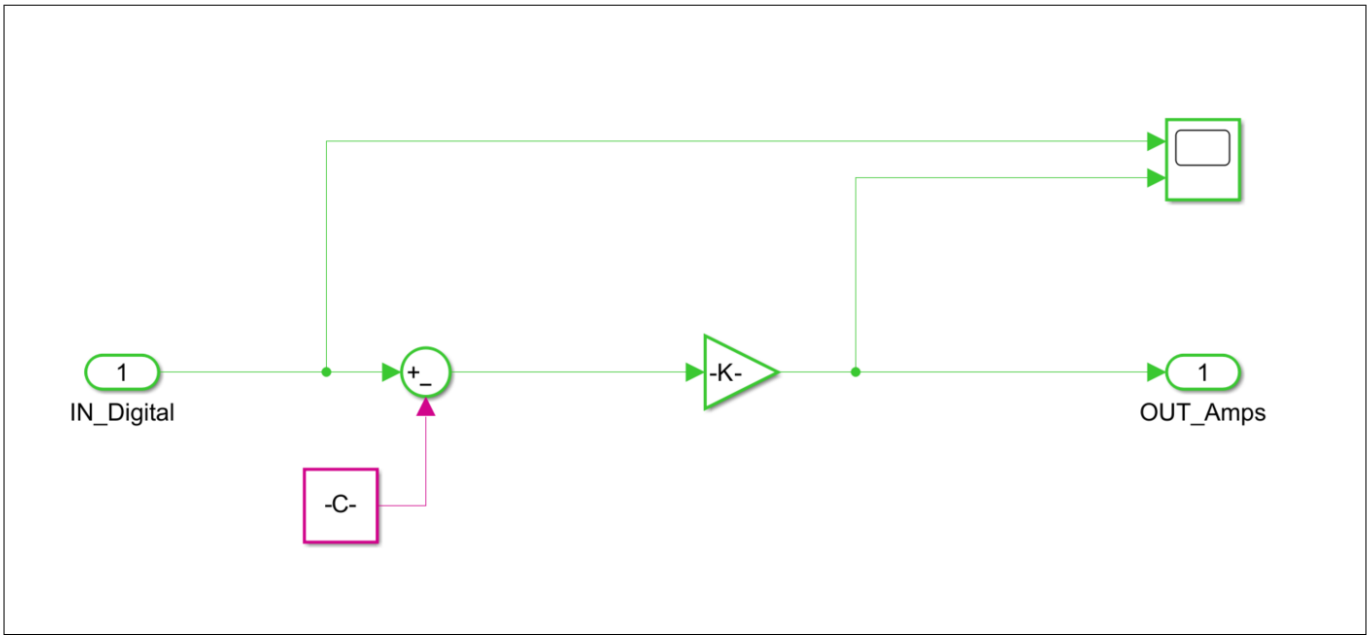

Figure 7 Internal structure of the current calculation block

Figure 7 illustrates the internal structure of the current calculation block. The input port **IN_Digital** takes 3 integer values corresponding to the ADC measurements of the 3-phase currents. Initially, the DC offset is removed by subtracting the value specified in `params.sys.adc.bias` from the measurements. Next, the resulting integer values are scaled by multiplying them with `params.sys.adc.quant`, converting the measurements into currents expressed in amperes.

ModusToolBox™ MCDI also provides some additional hardware measurements that can be used for algorithms, such as bus voltage, analog speed input, and power board temperature. The analog speed input is used to show an additional reference example of how the hardware measurements can be handled within the controller subsystem. In this subsystem, there is a speed calculation block, highlighted by a blue box in the bottom left corner in Figure 6. This block takes raw 12-bit ADC measurements as input and scales them to a range between 400 and 2000 RPM.

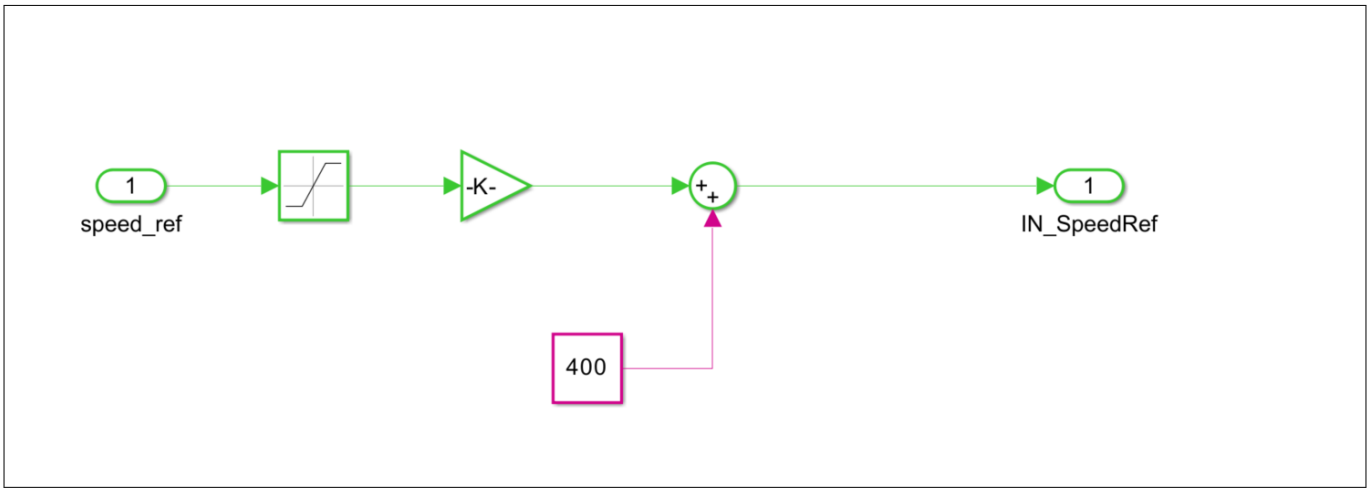

Figure 8 Internal structure of the speed calculation block

Figure 8 shows the internal structure of the speed calculation block. The input port **speed_ref** accepts a single integer value representing the ADC measurement from the potentiometer on the board. The input is passed through a saturation block, which constrains the values within the range of 0 to 4096. Subsequently, this range is linearly scaled to correspond to a speed interval of 400 to 2000 RPM. A constant block is included to ensure that the minimum speed is set to 400 RPM during closed-loop operation.

The controller subsystem block is responsible for outputting three integer values that will be written into the compare registers of the TCPWM peripheral. Therefore, it is also the task of the controller subsystem to calculate the integer values. The block indicated by the rightmost blue box in Figure 6 is dedicated for this purpose. The compare values calculation block contains inverse Park and inverse Clarke transforms, which are followed by a MATLAB® function block that translates the computed voltage values into compare values. The MATLAB® function block within the Compare Values Calculation Block relies on several predefined variables to compute the compare values. These are `params.sys.tcpwm.f_tcpwm`, `params.ctrl.f_c`, and `params.sys.vdc_nom` and defined in `.m` file. For this example, these are equal to 48 MHz, 20 kHz, and 24 (V) respectively.

All parameters related to both the control and simulation processes are defined in the `PSOCC3CustomFocModel_init.m` file. This initialization script contains all the necessary parameter definitions required for simulation. A part of this `.m` file where key parameters are defined is given in Code Listing 1.

Code listing 1

```
clc;
clear;

%% ADC HW Parameters
params.sys.adc.quant = (((5.0)/(3.3))*3300*1.0E-3) / (4096 * 10.0E-3 * 12); % ADC Quantization
Interval
params.sys.adc.bias = 2048 + 59;                                            % ADC 0V bias

%% TCPWM HW Parameters
% params.sys.tcpwm.f_tcpwm = (240.0e6); % TCPWM Frequency [Hz]
params.sys.tcpwm.f_tcpwm = (48.0e6); % TCPWM Frequency [Hz]
params.sys.tcpwm.t_tcpwm = 1/(params.sys.tcpwm.f_tcpwm); % TCPWM Period [s]

%% Battery
params.sys.vdc_nom = 24.0; % DC Supply Voltage [V]

%% Load Parameters
params.load.inertia = 0.000075; % Mechanical Load Inertia [kg*m^2]

%% Motor Parameters
params.motor.P          = 8; % Number of Poles (NOT pole pairs)
params.motor.lq         = 600.0E-6; % Quadrature Axis Inductance [H]
params.motor.ld         = 600.0E-6; % Direct Axis Inductance [H]
params.motor.lam        = 6.0E-3; % Magnetic Flux [Wb]
params.motor.r          = 400.0E-3; % Resistance [Ohm]
params.motor.T_max      = 0.390; % Maximum Torque [Nm]
params.motor.i_cont     = 3.50; % Rated Current [A]
params.motor.v_nom      = 24.0 * (1/sqrt(3)); % Nominal Voltage [V]
params.motor.rotorJ     = 48; % [g*cm^2]
params.mech.inertia    = 11.0E-6; % [kg.m^2]
params.mech.viscous    = 1.2E-5; % [kg.m^2/sec]
params.mech.friction   = 6.1E-3; % [N.m]

%% Control Parameters
params.ctrl.f_c                = 20e3; % Control Frequency = 20KHz
params.ctrl.t_c                = (1/params.ctrl.f_c); % Control Period = 1/Control Frequency
params.ctrl.P_max              = params.sys.vdc_nom * params.motor.i_cont; % Maximum Power
params.ctrl.T_max              = params.motor.T_max; % Maximum Torque
params.ctrl.rpm_transition     = 400; % RPM value at which the transition from OL to Cl occurs

%% Outer (Speed) Loop Parameters
params.ctrl.speed.kp = 0.0015;
params.ctrl.speed.ki = 0.0150;
params.ctrl.speed.kaw = 0;
params.ctrl.speed.rate_lim     = ((((2*pi)*(((4000.0 /
4.0)*(1.0/60.0)))))*(0.5)*(params.motor.P));
params.ctrl.speed.ratio = 10;

%% Id Loop Parameters
params.ctrl.current.id.kp = 2.0;
params.ctrl.current.id.ki = 0.5 * params.ctrl.f_c;
```

```
params.ctrl.current.id.kaw = 0;

%% Iq Loop Parameters
params.ctrl.current.iq.kp = 2.0;
params.ctrl.current.iq.ki = 0.5 * params.ctrl.f_c;
params.ctrl.current.iq.kaw = 0;

%% Feedforward Loop Parameters
params.ctrl.current.ff.idVect = [-200,200];
params.ctrl.current.ff.iqVect = [-200,200];
params.ctrl.current.ff.LdMatrix = params.motor.ld * ones(2, 2);
params.ctrl.current.ff.LqMatrix = params.motor.lq * ones(2, 2);
params.ctrl.current.ff.psimMatrix = params.motor.lam * ones(2, 2);
```

3 MATLAB®/Simulink® embedded coder - code generation

This section describes the code generation using the Embedded Coder. Figure 9 illustrates the flow followed in this section.

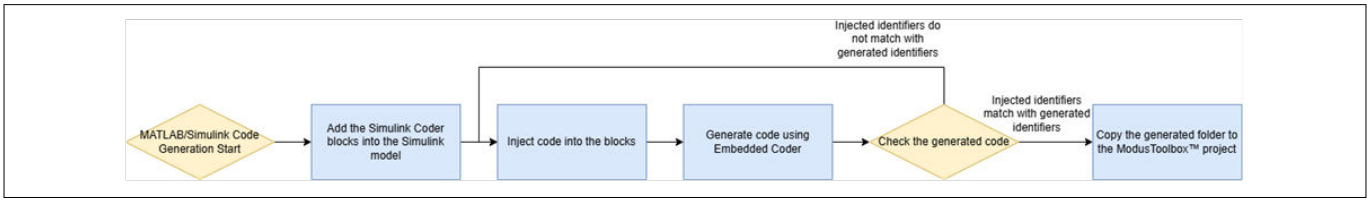

Figure 9 Embedded coder flowchart

3.1 Usage of Simulink® coder blocks

Simulink provides different blocks to manipulate code generation. These blocks can be found under **Library Browser> Simulink® Coder> Custom Code**. These blocks do not have any effect on simulation. Instead, they only affect the code generation.

There are three important blocks (Figure 10), which are used to inject code into different locations in the generated code. This injected code ensures the input and output transfer between the MATLAB® generated code and ModusToolbox™ MCDI generated code.

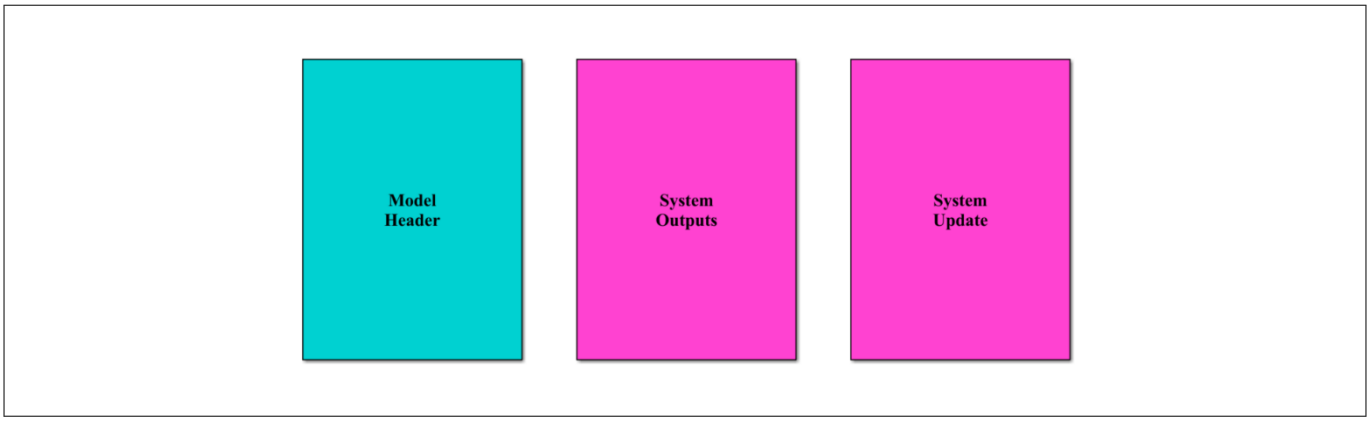

Figure 10 Simulink® coder custom code blocks

The generated source code contains two primary functions, each identified by the suffixes `<system>_initialize and <system>_step`. The initialization function is called only once within the main function to set up the system, while the step function must be called periodically within an interrupt service routine. The System Update and System Output blocks affect the code generated for the step function, although they insert code into different specific locations.

The “Model Header” block allows for the insertion of custom code into the generated header file. This feature can be utilized to add external file inclusions, declare variables or functions, and perform similar tasks. In our specific scenario, this block is used to include the MCDI header file into the automatically generated header file. This will allow us to access the array generated by the MCDI, which keeps the 3-phase current data. By double-clicking on this block, as illustrated in Figure 11, the injected code can be seen.

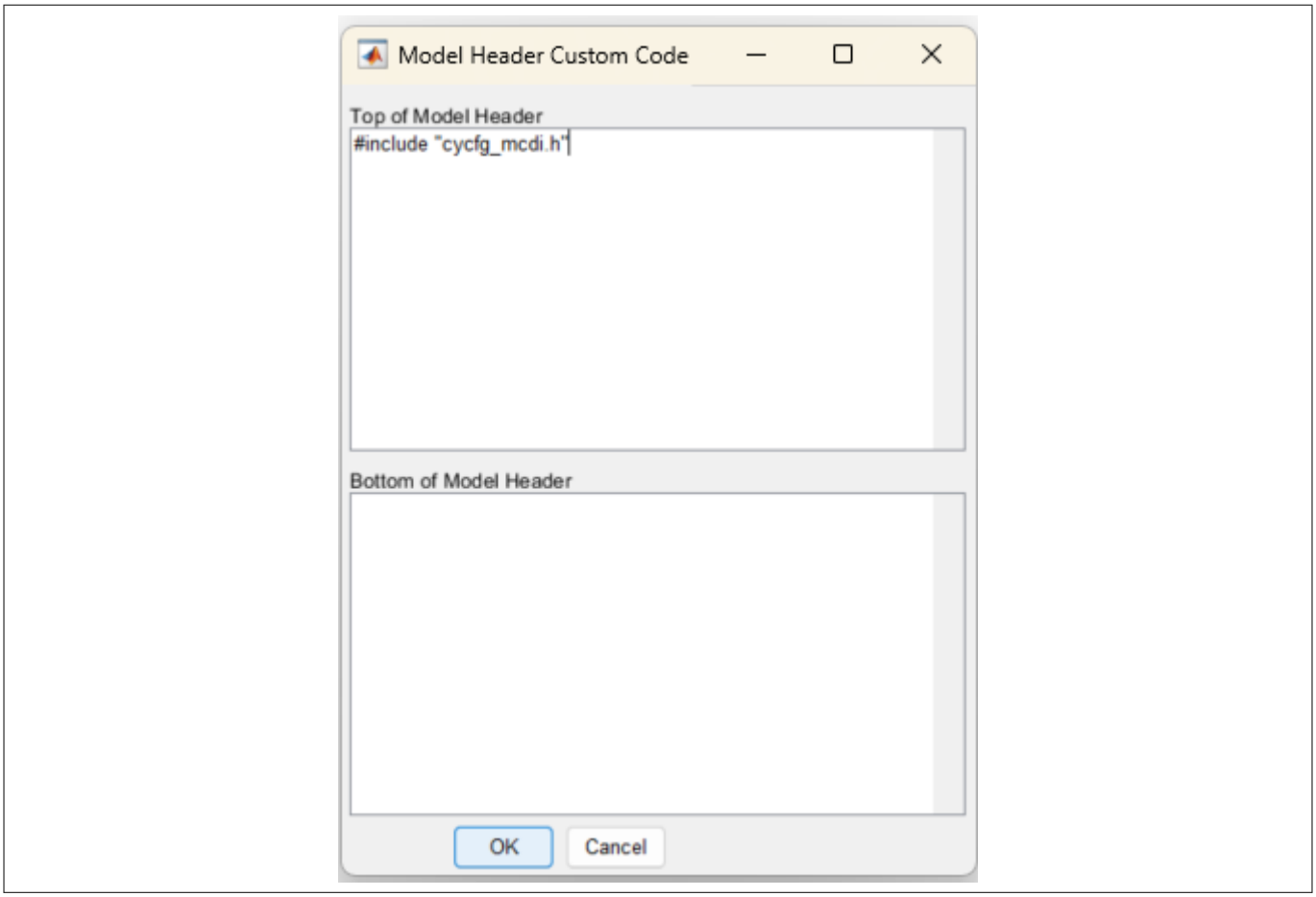

Figure 11 Model Header Custom Code block

The “System Outputs” block allows for the insertion of custom code into the beginning of the step function. This feature can be utilized to transfer the system inputs. In our case, this block is used to transfer the ADC measurements from MCDI code to the MATLAB® generated code. By double-clicking on this block, as illustrated in Figure 12, the injected code can be seen.

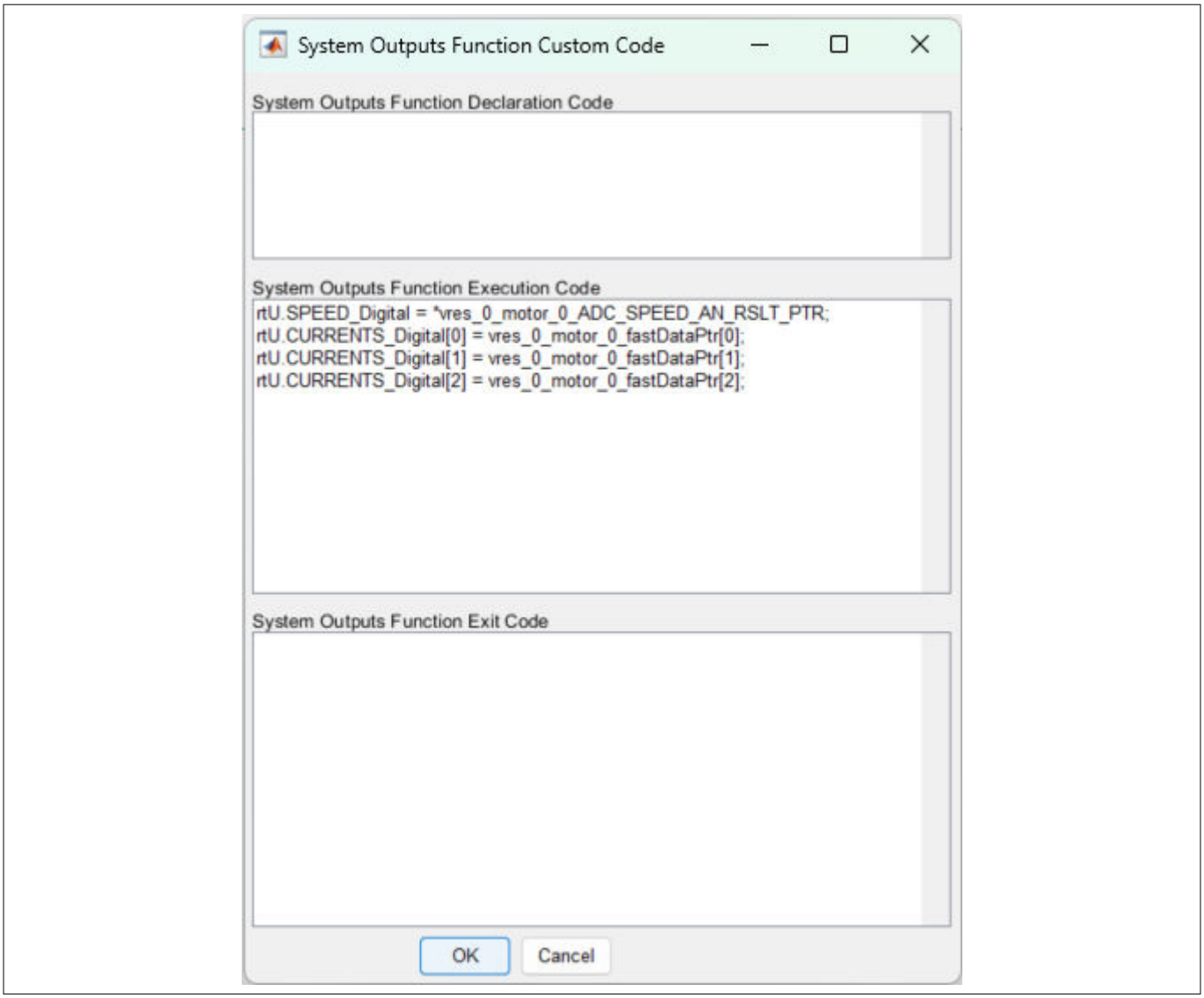

Figure 12 System Outputs Custom Code block

Simulink® signal within the controller subsystem, for the input 3-phase currents,s are named as *CURRENTS_Digital*. The embedded coder uses this signal name as an identifier in the generated code. `vres_0_motor_0_fastDataPtr`is the array defined by the MCDI code that will be shown in Section Configuration of Motor Control Driver Interface (MCDI). Using this array, the 3-phase inputs are transferred from MCDI code to the Embedded Coder generated code. Similarly, *SPEED_Digital* represents the Simulink® signal within the controller subsystem used to set the target speed. In the generated code, the value of this is assigned by `*vres_0_motor_0_ADC_SPEED_AN_RSLT_PTR`, which is a pointer to the memory address that stores the ADC measurements from the potentiometer. This will also be shown in Section Configuration of Motor Control Driver Interface (MCDI).

The "System Update" block allows for the insertion of custom code at the end of the step function. This functionality can be used to handle the transfer of system outputs. In our specific case, this block is employed to write the compare values, which is computed by the MATLAB®/Simulink®-generated code into the TCPWM registers. By double-clicking on this block, the injected code can be seen (see Figure 13).

The `Cy_TCPWM_PWM_SetCompare0BufVal` is a C function used to write a value into the compare registers. In `OUT_V_U, OUT_V_V, OUT_V_W` are the names of the outports of the controller subsystem in Simulink® model, which are directly used in the Simulink® generated code.

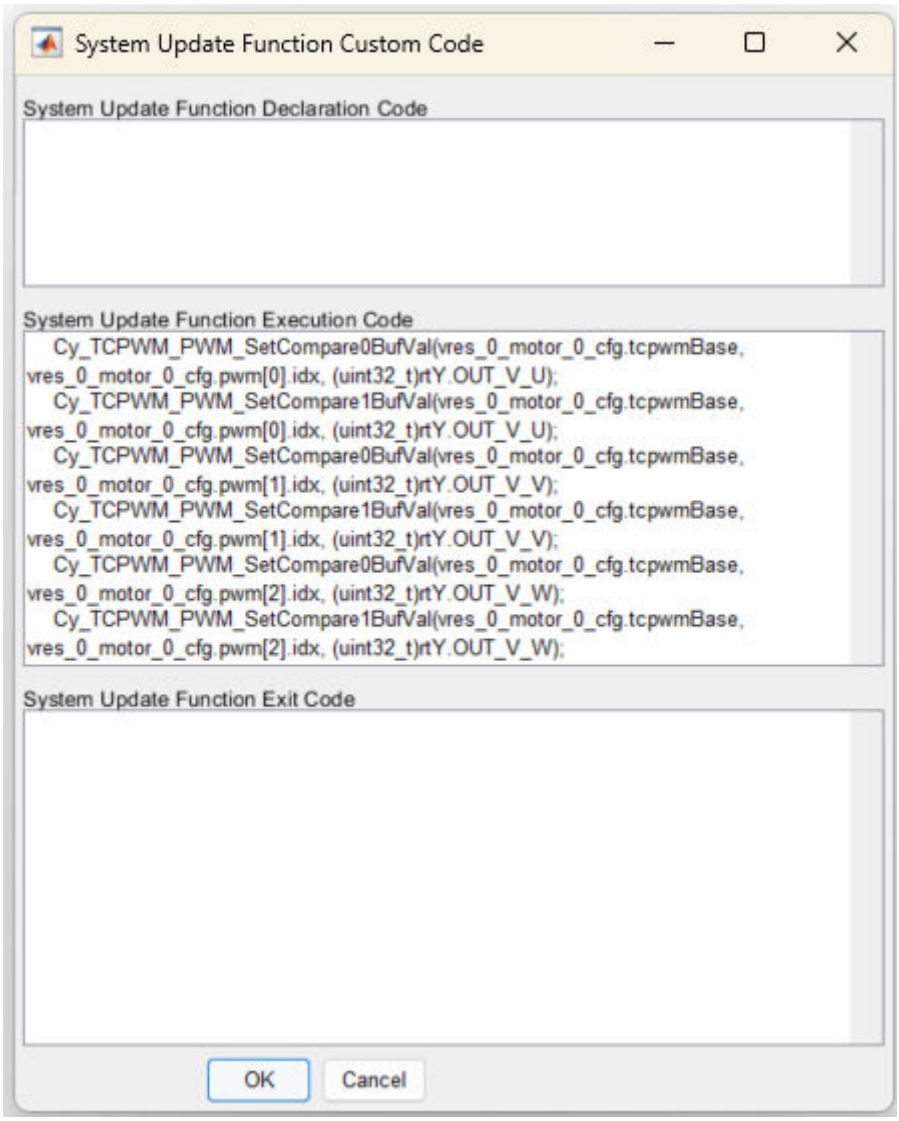

Figure 13 System Update Custom Code block

If the signals in the Simulink® model are unnamed or if a mismatch arises, an alternative method to determine the identifiers used in the code generation is as follows:

1. First, the code should be generated, as described later in Section 3.2
2. Open the generated code and observe the identifiers assigned for the inputs and outputs. Note down these identifiers
3. Use these identifiers to replace the corresponding ones within the injected code in the "System Outputs" and "System Update" blocks
4. Regenerate the code after making the necessary changes to the injected code

As long as the identifiers match and the signal names in model remain unchanged, there is no need to update/ modify the injected code.

3.2 Usage of Embedded Coder Application

The **Embedded Coder** application in Simulink® is used to generate the C code from the models. It can be accessed from the Apps tab in Simulink®. Upon opening, it provides a menu as illustrated in Figure 14. Before generating the code, several configurations need to be completed to ensure proper code generation. These

settings can be adjusted by clicking on the **Settings** option located within the **C CODE** tab of the Embedded Coder interface.

These configurations allow for customization of the code generation process, ensuring that the generated code meets the specific requirements of the project or target system.

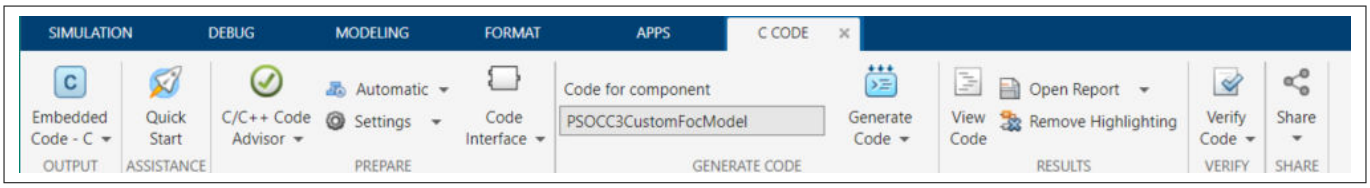

Figure 14 Embedded Coder menu

To configure the code generation settings,

Open the **Configuration Parameters** window by clicking on **Settings** in Embedded Coder menu (see Figure 14) or by using the shortcut CTRL+E

1. Set the system target file `ert.tlc` in the **Code Generation** section (see Figure 15), under the **Target Selection box**
2. Check and enable the options **Generate code** only and **Package code and artifacts** to ensure the generated code is compressed into a zip file and exclude unnecessary or redundant files from the package

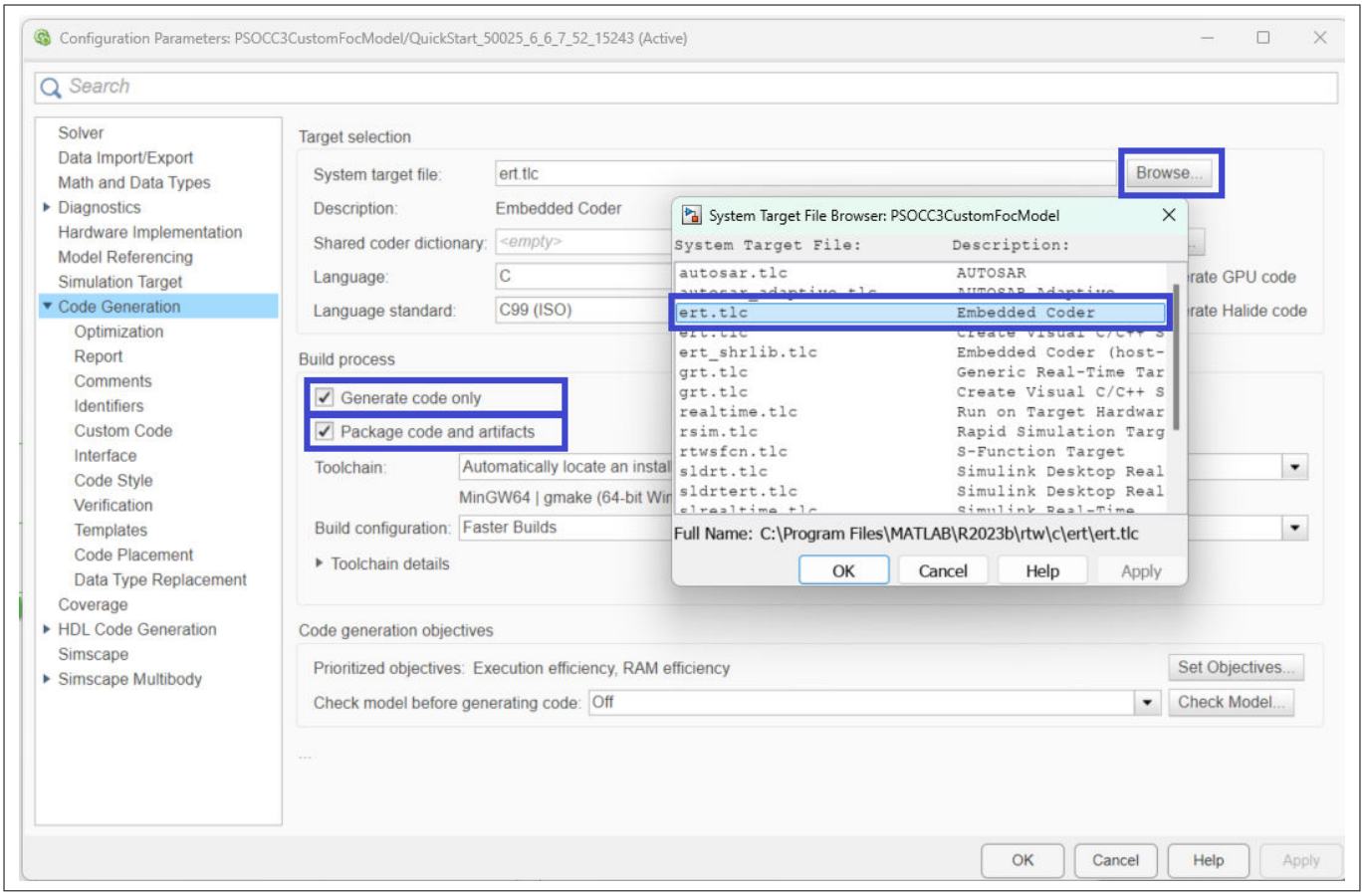

Figure 15 Configuration Parameters window Code Generation tab

An additional configuration to consider involves disabling the generation of the default `main.c` file. The following are the steps to generate the code and extract the relevant files:

1. Navigate to the **Templates** section under **Code Generation** in the **Configuration Parameters** window

2. Uncheck the **Generate an example main program** checkbox under the **Custom Templates** box as shown in Figure 16 to prevent the automatic generation of the `main.c` file

   ***Note**: Presence of `main.c` file can lead to compilation errors caused by multiple definitions of the main function, especially if a custom main function is already implemented in your project.*

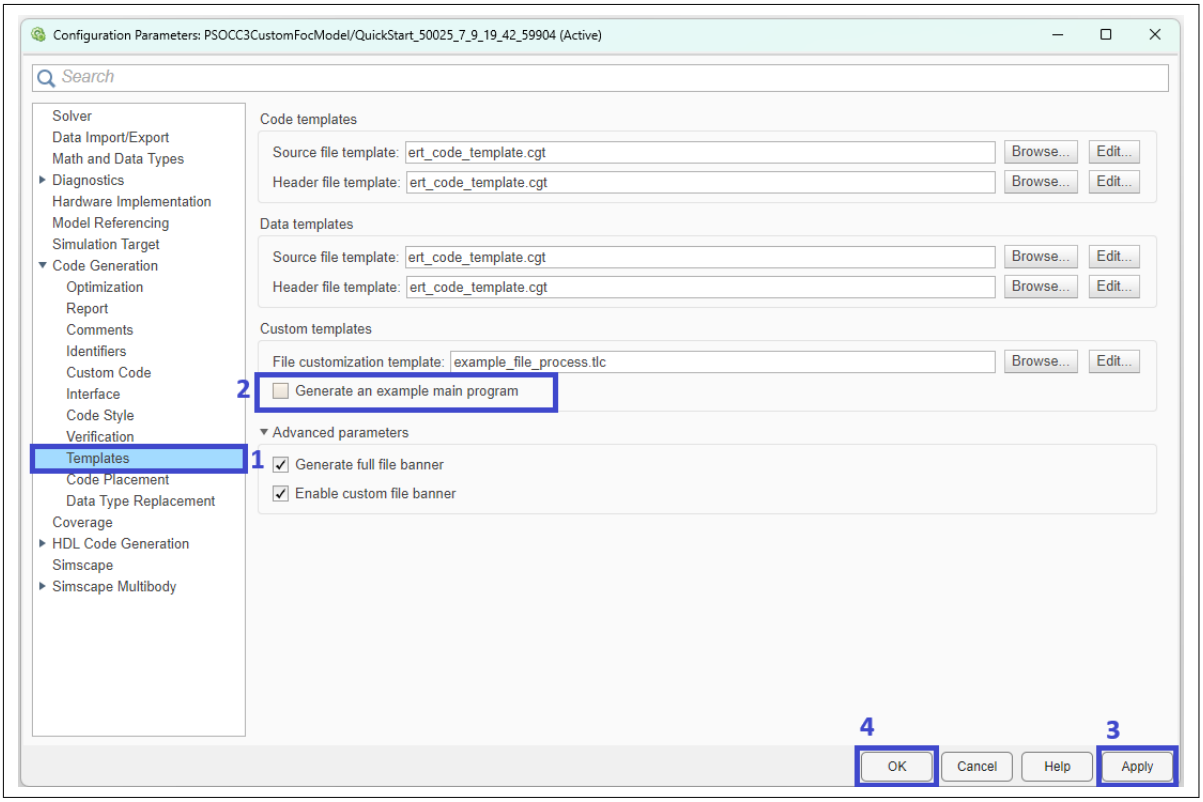

Figure 16 Configuration Parameters window Templates tab

3. Click on the **Apply** button to save the changes, and then click **OK** button to close the **Configuration Parameter** window (see Figure 16)

To proceed with code generation,

1. Click on **Quick Start** under the **C CODE** tab which is highlighted by a blue box. This opens the **Welcome** window

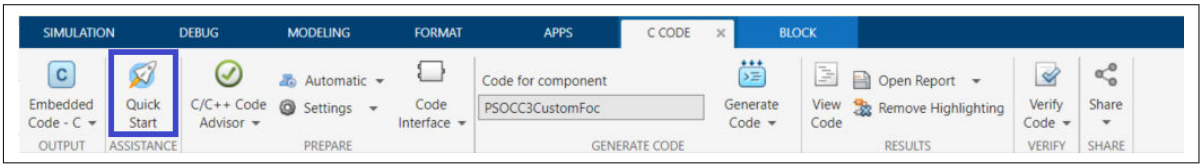

Figure 17 Quick Start button under embedded coder tab

2. Click **Next** to navigate and proceed to the **System** window
3. Select the **Subsystem** box and choose the appropriate subsystem for which the code will be generated. In this example, the subsystem is **"PMSM controller"**as depicted in Figure 18

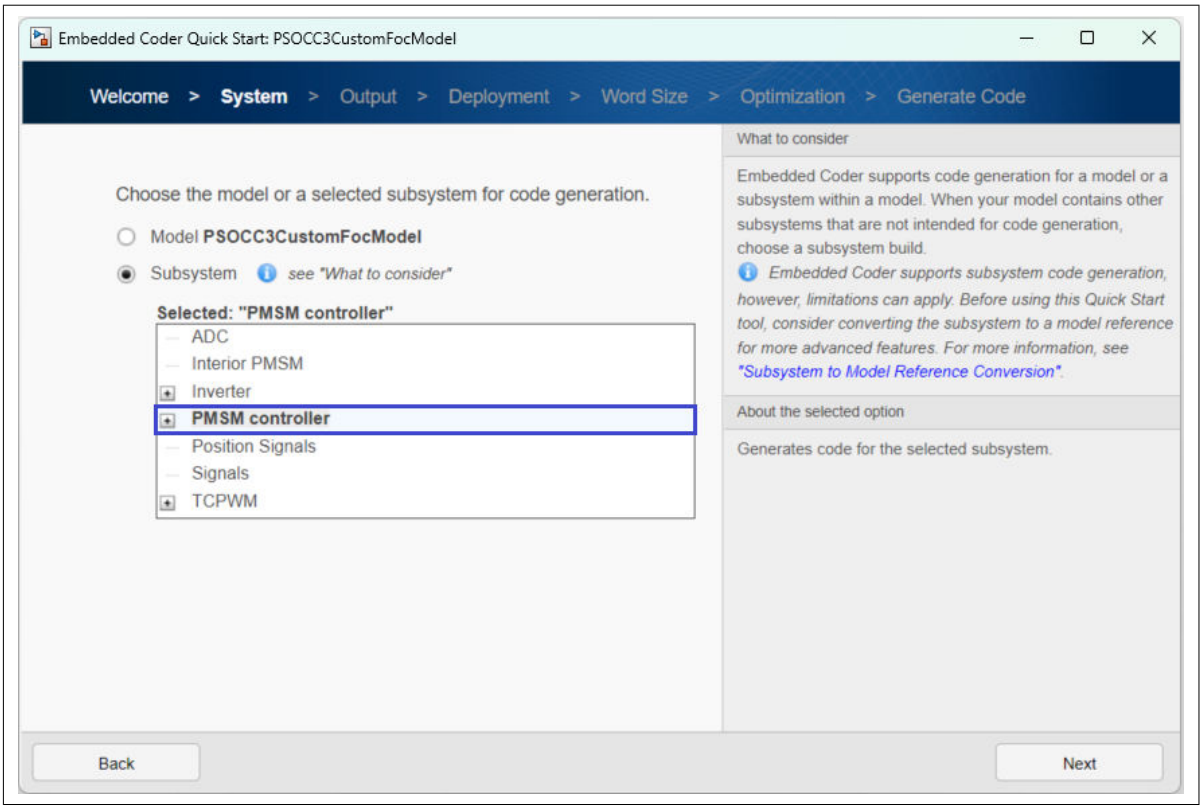

Figure 18 Quick Start menu

4. After selecting the correct subsystem, click the **Next** button repeatedly until you reach the **Word Size** window

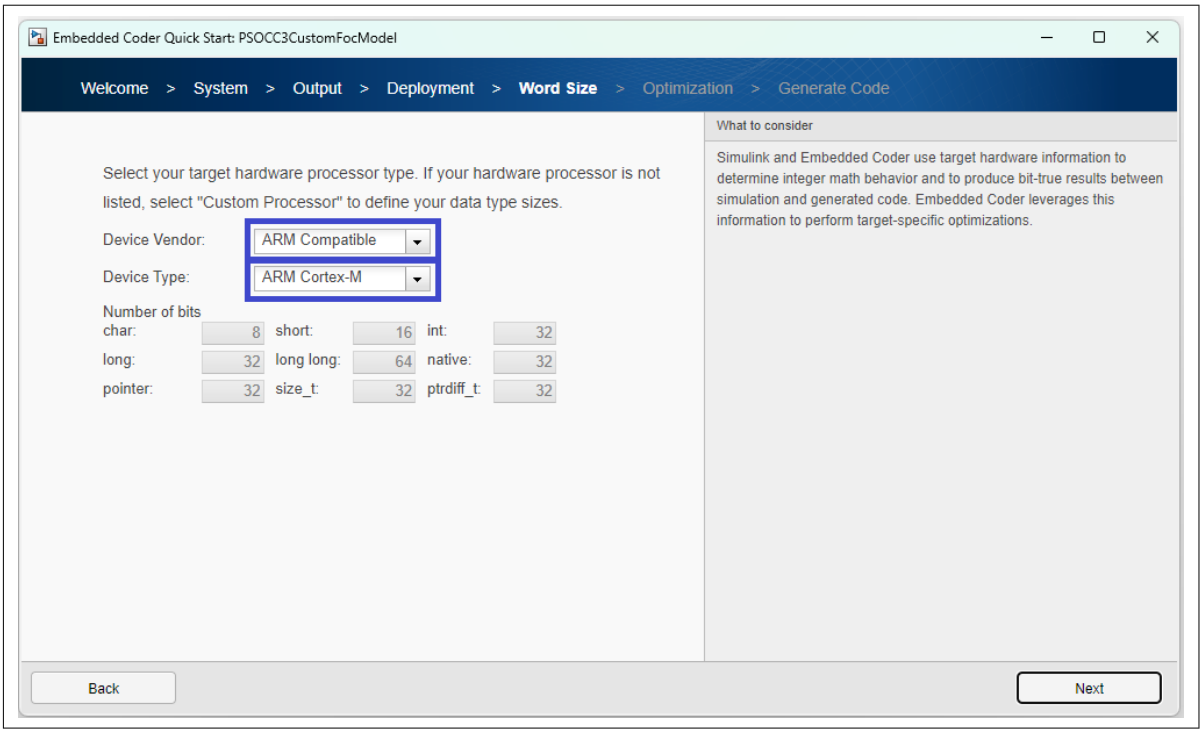

Figure 19 Word Size tab under Quick Start menu

5. As shown in Figure 19, configure the following settings in the **Word Size** window:

Device Vendor: Select **Arm® Compatible**

Device Type: Choose **Arm® Cortex®-M**

6. Continue clicking **Next** until the code generation process begins
7. After the code generation process is complete, simply click on the **Finish** button to close the **Quick Start** workflow
8. Open the **File Explorer** and navigate to the designated working directory, where the generated files are stored. Locate the generated zip folder and extract its contents

   ***Note**: Based on code generation settings, there are usually 2 folders: one named after the Simulink® model and the other named based on the MATLAB® version.*

9. Find the folder named after the Simulink® model (this folder will have the same name as your model), which contains the relevant code
10. Copy only the specific folder that contains the relevant code. This folder can later be pasted into the created ModusToolbox™ project, which will be further explained in Section Application

    ***Note**: It is important to note that there might be some cases where certain folders within the folder named after the MATLAB® version might also be required.*

4 ModusToolbox™ Motor Control Driver Interface (MCDI)

4.1 Creation of Empty Project in ModusToolbox™

To create a new empty ModusToolbox™ project for PSOC™ Control, follow these steps:

1. Open **ModusToolbox™** and navigate to the menu bar
2. Click on **File > New > ModusToolbox™ Application** to begin the project creation process
3. In the dialog box, locate and expand the **PSOC™ Control BSPs** section, then select **KIT_PSC3M5_CC2** as the target board. Click **Next** to proceed
4. In the next screen, under the **Getting Started** section, expand the options and select the **Empty App** template as the base for your application
5. Provide a name for your new application in the appropriate field. `MATLAB_FOC_PSOCC3` is used for this example
6. Finally, click on **Create** to generate the new project

Once the project has been successfully created, activate the **MCDI** within the Device Configurator. This requires adding the **Motor Control Library** to the project, even if the library itself is not actively used in the application. Follow these steps to add the **Motor Control Library:**

1. Open the Library Manager:

   Locate the **Library Manager** in the **Quick Panel** of ModusToolbox™ and open it (see Figure 20)
2. Add the Motor Control Library as depicted in Figure 20:
   a. In the Library Manager, click on the **Add Library** button
   b. Expand the **Middleware** section to find the **motor-ctrl-lib**
   c. Select `motor-ctrl-lib` and click **OK**
3. Update the Project:

   After adding the library, click on the **Update** button located at the bottom right of the Library Manager window

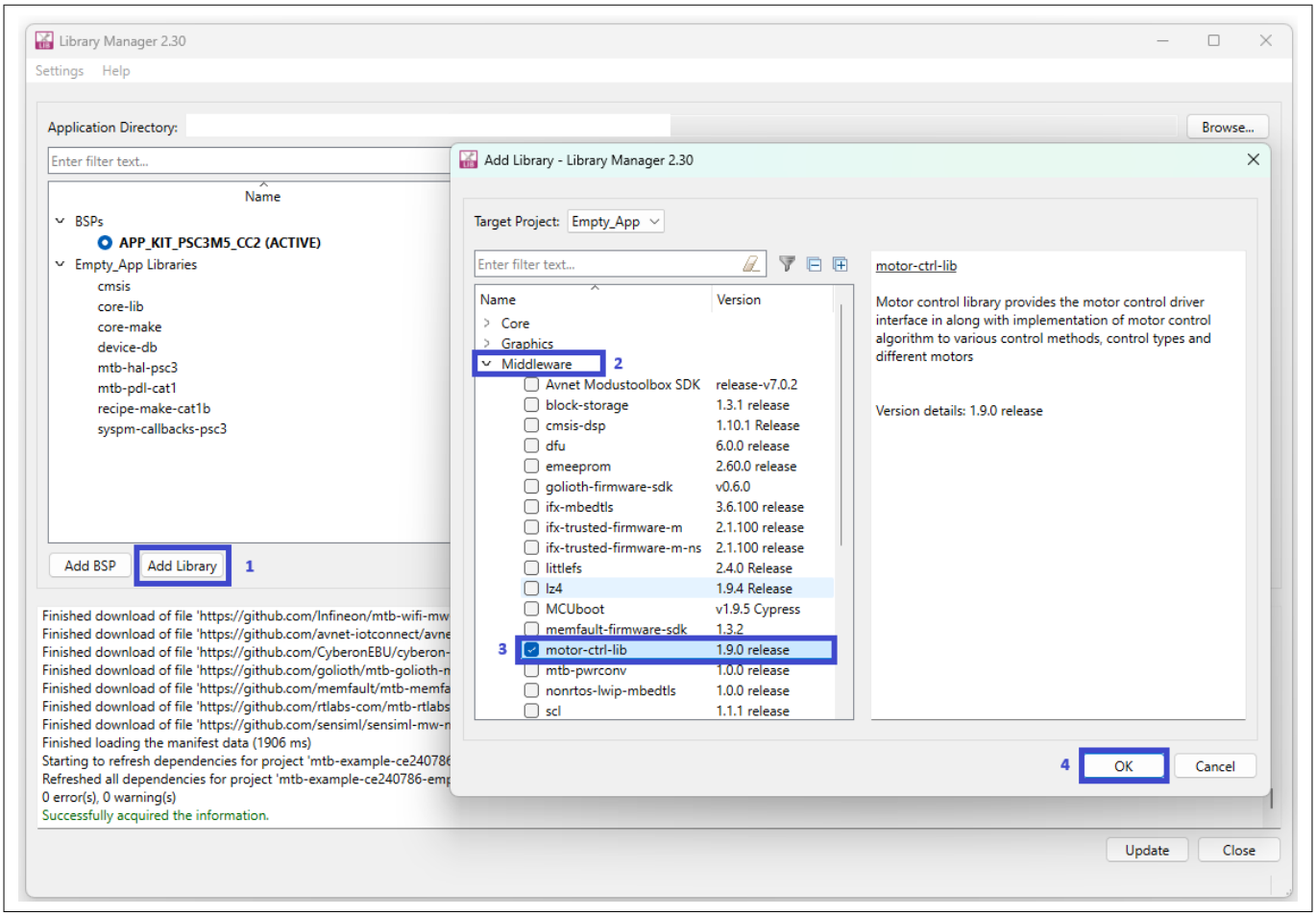

Figure 20 Library Manager

This process ensures that the Motor Control Library is included in the project, which is essential for activating the **MCDI** in the Device.

After adding the Motor Control Library, the project's ***Makefile*** must be modified by including few specific lines of command. These lines are necessary for ensuring that the project is correctly set up for building. Follow these steps:

1. **Edit the Makefile:**
   - **a.** Open the **Makefile** in your project directory
   - **b.** Add the command lines:

     ```
     CY_IGNORE=$(wildcard ../mtb_shared/motor-ctrl-lib/*/HwInterface*)
     CY_IGNORE+=$(wildcard ../mtb_shared/motor-ctrl-lib/*/OperationalCode*)
     CY_IGNORE+=$(wildcard ../mtb_shared/motor-ctrl-lib/*/ThirdPartyLib*)
     ```

   - **c.** Modify the makefile commands as:

     ```
     VFP_SELECT=hardfp
     CFLAGS=-Ofast
     ```

2. **Clean and compile the Project:**
   - **a.** Go to the **Quick Panel** in ModusToolbox™

     Click on **Clean Project** to remove any previous build artifacts and prepare the project for a fresh build

     After cleaning, click on **Build Project** to compile the project with the updated configurations.

     This process ensures that the project is properly configured and built.

4.2 Configuration of Motor Control Driver Interface (MCDI)

To open and configure the **MCDI Solution** in ModusToolbox™, follow these steps:

1. **Open the Device Configurator:**

   Navigate to the **Quick Panel** in ModusToolbox™ and click on **Device Configurator** to launch it.

2. **Access the Solutions Tab:**

   Once the Device Configurator opens, click on the **Solutions** tab located at the top of the window.

3. Add a Solution:

   a. In the **Solutions** tab, click on the **+** symbol to add a new solution

   b. From the list of available solutions in the appearing window, select `3-shunt-1.0` and click **OK** to confirm your selection

Upon selecting the solution, a new configuration window will appear on the right side of the screen. This window is where the parameters for the solution can be set and customized according to the project's requirements. Figure 21 depicts the configurations done for this example project.

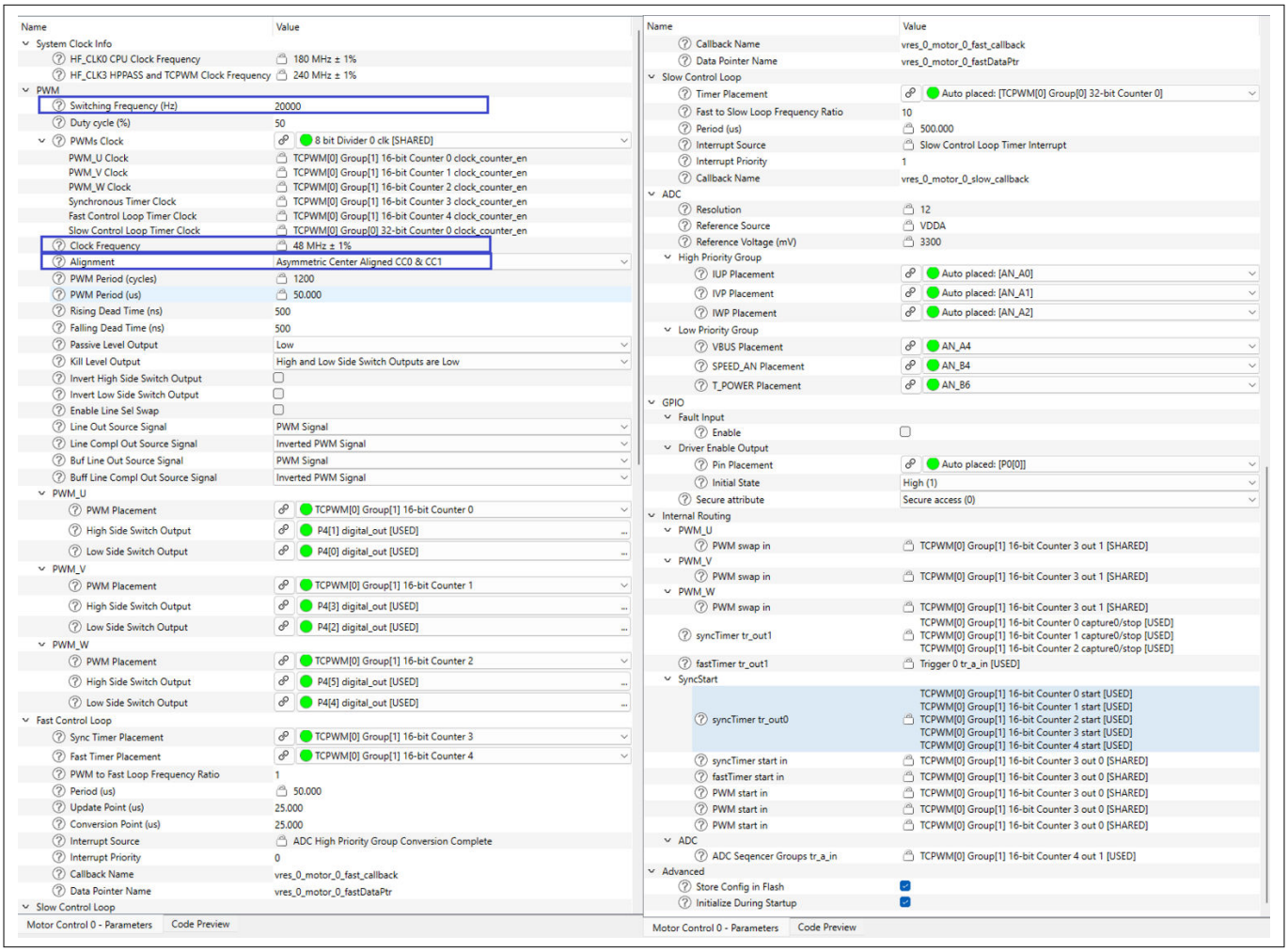

Figure 21 MCDI Solution Configuration

When configuring the project in the **MCDI solution**, it is crucial to ensure that the setup aligns with the Simulink® model as well as the hardware requirements. The following are the key points to consider during configuration:

1. **Switching Frequency:**

Enter the switching frequency as specified in the Simulink® model. The specific field for this parameter is usually highlighted in the interface (highlighted by the blue box in Figure 21). It is essential that this value matches the switching frequency defined in the Simulink® model to ensure synchronization.

2. **Clock Frequency:**
   - Ensure that the **Clock frequency** aligns with the **TCPWM frequency** used within the Simulink® model
   - To configure this, select the **PWM Clock** in the settings and click on the **link button** (located to the left of the clock configuration field). This action will direct you to the **Clock Configuration** window
   - In the **Clock Configuration** window, set the appropriate **divider value** for the clock source so that the final clock frequency matches the TCPWM frequency specified in the Simulink® model
3. **Verify Clock Frequency:**

   Confirm that the **Clock frequency** field reflects the correct and expected frequency necessary for the project. This step ensures that the system operates as intended.
4. **Alignment Mode:**

   Select the **Asymmetric Center Aligned CC0 & CC1** mode for the PWM alignment. This mode must match the alignment configuration set for TCPWM subsystem block in the Simulink® model, ensuring synchronization between the model and the hardware.
5. **Data Pointer Name:**

   The name specified in this field will serve as the identifier for the array that stores the 3-phase current data. It is critical that this name matches the identifier used in the injected code, as outlined in Section Usage of Simulink® coder blocks. It is essential for proper integration and functionality of the system and any mismatch could lead to compilation errors.
6. **Hardware-Specific Configurations:**
   - Complete the remaining configurations based on the specific hardware being used. The configurations in Figure 21 are done for MC1 Motor Control kit
   - The speed reference input (mentioned in Section Usage of Simulink® coder blocks)is located in the **Low Priority Group** under the **ADC** tab group. This functionality is mapped to the analog input from the potentiometer. The code generated by the MCDI already defines the pointer `vres_0_motor_0_ADC_SPEED_AN_RSLT_PTR`, which refers to the memory address where measured analog reference speed value is stored. The content of this memory address is updated periodically. The definitions for this and similar functionalities are in `cycfg_mcdi.h` file

After configuring the **Solutions** tab, the associated links and configurations must be set up correctly to ensure proper functionality. The following steps outline how to configure these settings:

1. **Enable Samplers in the Analog Tab:**
   - **a.** Navigate to the **Analog** tab in the Device Configurator
   - **b.** Expand the path **HRPASS > SAR > Samplers > Direct**
   - **c.** Check the checkboxes for **Sampler 0, Sampler 1, Sampler 2, and Sampler 4**
2. **Enable Muxed Samplers:**
   - **a.** Still under **HRPASS > SAR > Samplers**, switch to the **Muxed** section
   - **b.** Check the checkboxes for **Sampler 0** and **Sampler 2**
3. **Enable SAR:**

   Check the checkbox for **SAR** under the **HRPASS** hierarchy
4. **Enable HRPASS:**

   Check the checkbox for **HRPASS**
5. **Enable the Autonomous Controller:**

Check the checkbox for **Autonomous Controller** to enable this module

6. **Add State 0 in the Autonomous Controller:**
   a. Under **Autonomous Controller > STT**, click the **+** button to add a new state
   b. Name the new state **"State 0"**

After completing the configurations in the Device Configurator, press **CTRL+S** to save the settings and generate the corresponding code. Once the code is generated, close the Device Configurator.

5 ModusToolbox™ Motor Suite GUI

It is possible to monitor the variables defined in the auto-generated code during operation using the ModusToolbox™ Motor Suite GUI. It offers a software oscilloscope functionality that can be used to sample the global variables within the ModusToolbox™ project. This software is available for download through Infineon Developer Center. This section explains how to use the ModusToolbox™ Motor Suite application.

Before starting a project in the Motor Suite GUI, the probeScope library needs to be integrated into the project. This library is available within the `motor-ctrl-lib`, which is located under `mtb-shared`. Alternatively, it can be obtained from the motor-ctrl-lib repository on Infineon's GitHub. This probescope folder should be copied under the ModusToolbox™ project's root path as depicted in Figure 22.

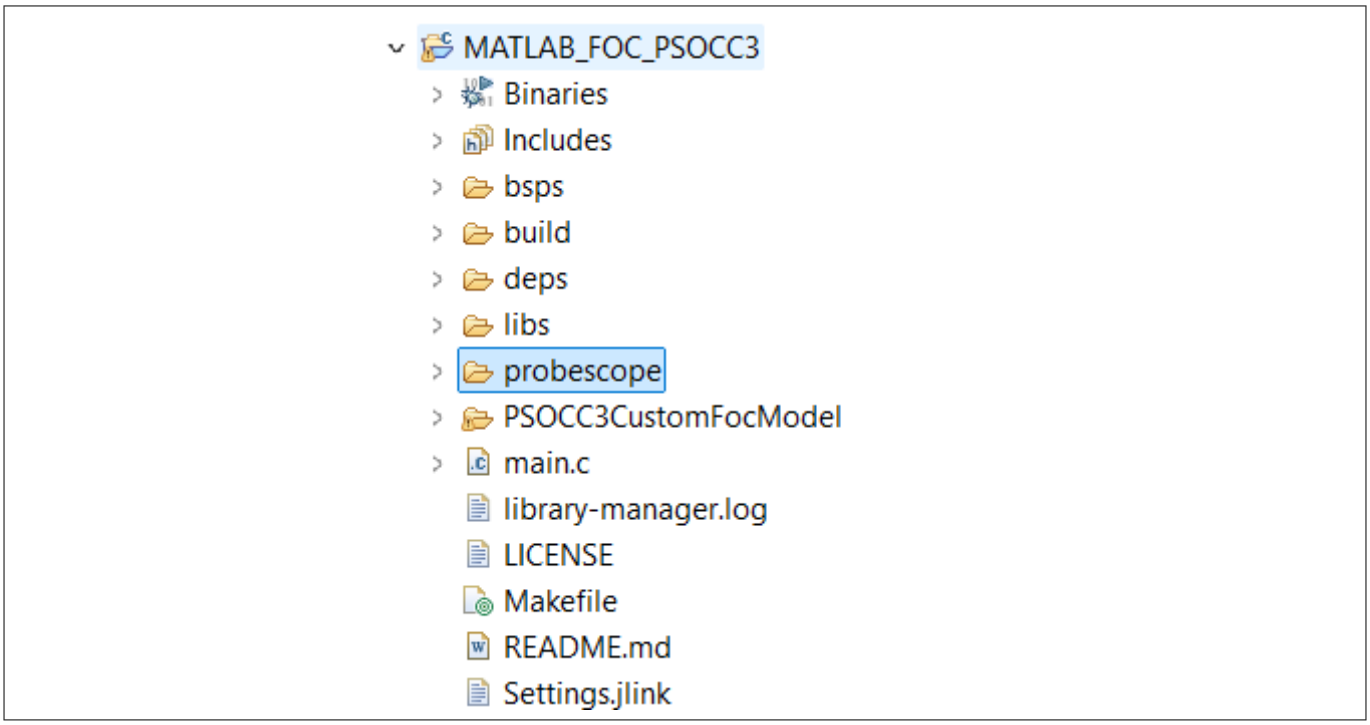

Figure 22 Probescope folder inside ModusToolbox™ project

This library provides us two high-level C functions, which are the `ProbeScope_Init` and `ProbeScope_Sampling`. The function `ProbeScope_Init` must be called once in main function to initialize the library. The function `ProbeScope_Sampling` must be called periodically in the interrupt service routine. The example code for the usage of these functions is provided in Code Listing 2 in Section Application.

It is assumed that the code provided in the next section has already been flashed onto the device. To proceed:

1. Open **ModusToolbox™ Motor Suite**, which can be accessed from the **Quick Panel**
2. In the **New Project** window, find and select **PSOC™ Control C3**. Choose the **RFO** option and click on **New Project** to create a new project
3. Once the project is created, locate the **ELF symbol** at the bottom-right corner of the GUI. Click on it to select the appropriate executable ELF file (highlighted by red-box 1 in Figure 23)
4. Navigate to the designated working directory, select the corresponding ELF file, and click **Open** as depicted in Figure 23 (highlighted by red-boxes 2 and 3)
5. Finally, click on the **Launch Oscilloscope** button to proceed (orange button highlighted by red-box 4 in the top-right corner of the GUI)

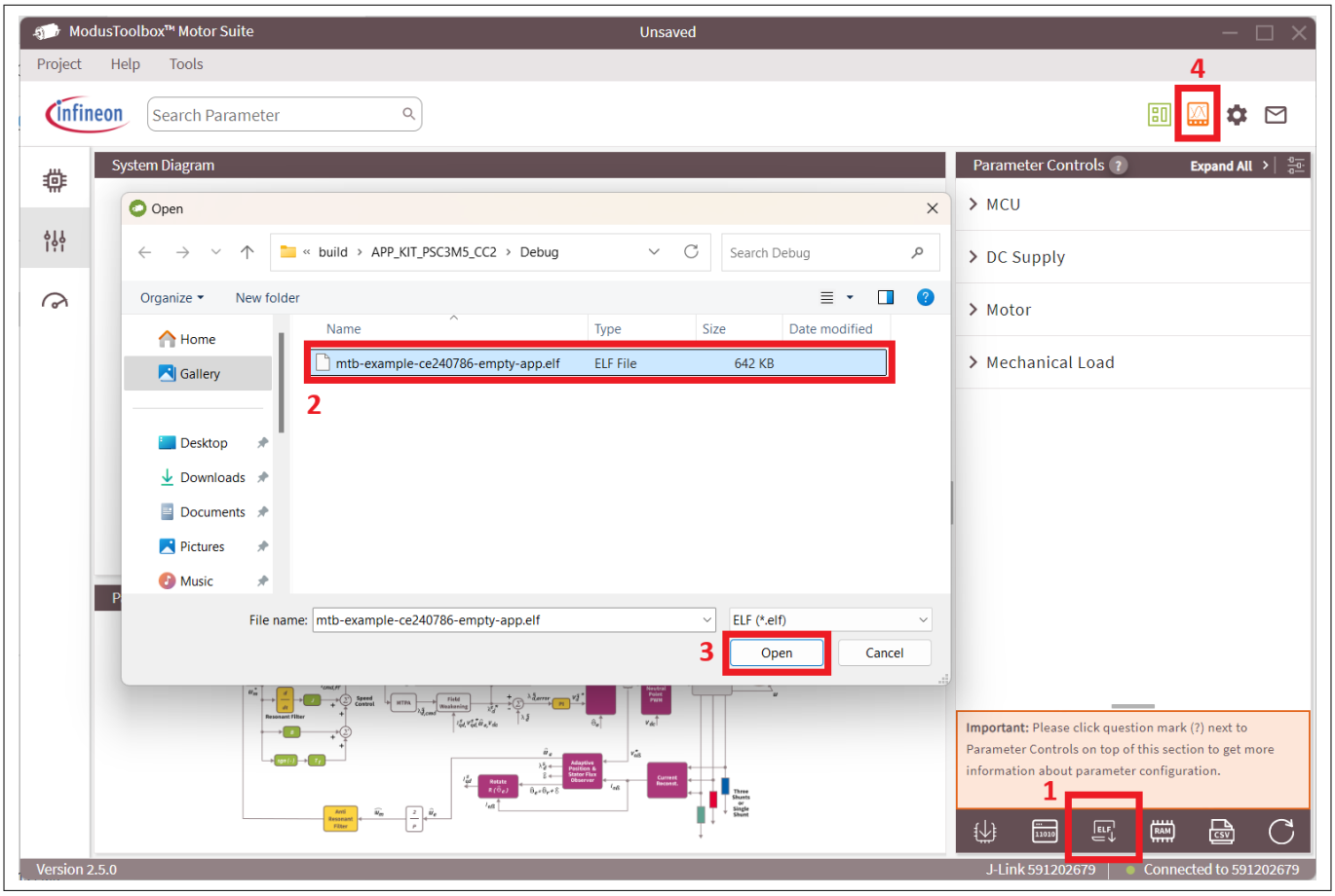

Figure 23 ELF file selection in Motor Suite GUI

The channels of the oscilloscope need to be configured first. The channels 1, 2, and 3 are configured with `rtu.CURRENTS_Digital[0]`, `rtu.CURRENTS_Digital[1]`, and `rtu.CURRENTS_Digital[2]`, respectively. The 4th channel can be disabled by unchecking the checkbox. Clicking on the top-left green connect button starts the oscilloscope. Figure 24 shows a screenshot of the oscilloscope.

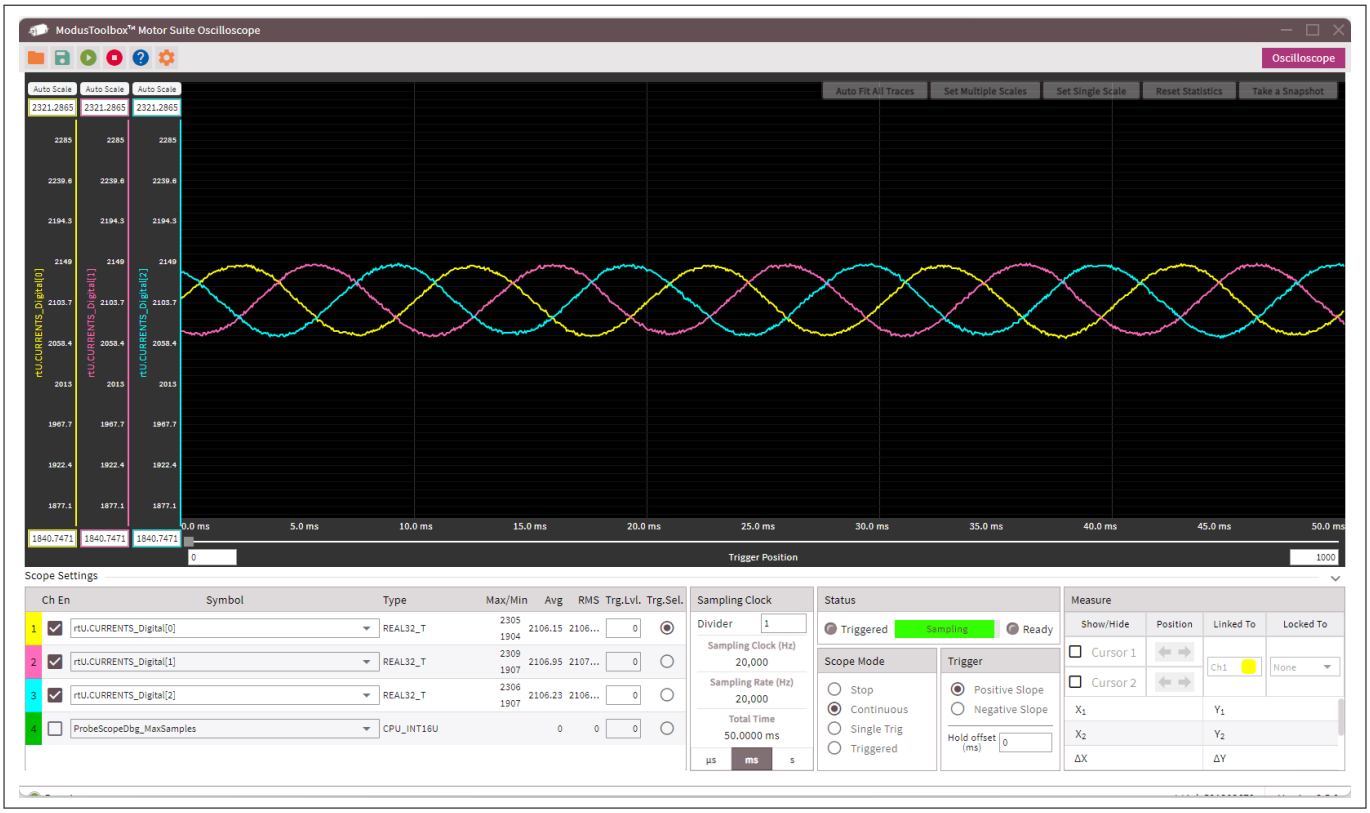

Figure 24 Screenshot of Motor Suite Oscilloscope

6 Application

6.1 Preparing the application

To finalize the code integration process, a few more additional steps are required. First, ensure that the Simulink® generated folder is copied under the ModusToolbox™ project's root as shown in Figure 25. The Embedded Coder generated files are highlighted by a blue box in the following figure.

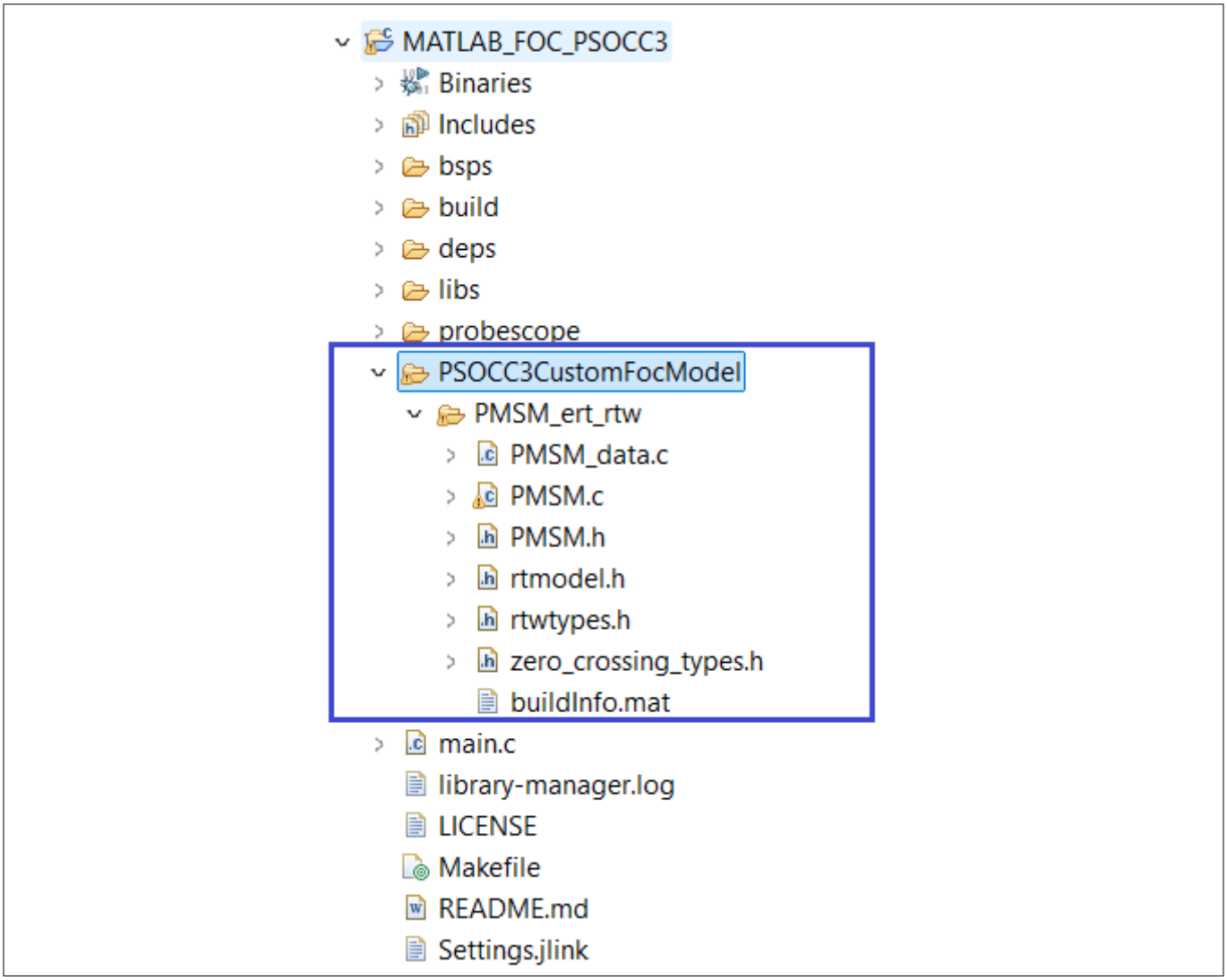

Figure 25 Example figure

***Note**: Ensure that each time the code is generated, it needs to be manually copied into the ModusToolbox™ project folder.*

Then, we need to specifically define the necessary callback functions in the `main.c` file and ensure that all the required initialization functions are called in the appropriate order to properly set up the system. An example code of `main.c` is given in the code listing 2 below.

Code listing 2

```
#include "mtb_hal.h"
#include "cybsp.h"
#include "pmsm.h"
#include "probe_scope.h"

void vres_0_motor_0_slow_callback(void)
{
}

void vres_0_motor_0_fast_callback(void)
{
    PMSM_step();
    ProbeScope_Sampling();
}

int main(void)
{
    cy_rslt_t result;
    result = cybsp_init();
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }
    vres_0_motor_0_init();
    mtb_mcdi_enable(&vres_0_motor_0_cfg);
    ProbeScope_Init((uint32_t)20e3);
    PMSM_initialize();
    vres_0_motor_0_start();
    __enable_irq();
    for (;;)
    {
    }
}
```

The `vres_0_motor_0_fast_callback` is the callback function periodically invoked by the MCDI code. This function serves as the execution point for the control algorithm. As explained in Section Usage of Simulink® coder blocks, the step function of the control algorithm is called within this callback.

Furthermore, all the necessary initialization functions are called within the main function. These initialization functions set up the hardware and software components required for the application.

6.2 Using the ready-to-flash ModusToolbox™ project

To bypass all the detailed steps, if desired, the `MATLAB_FOC_PSOCC3` project folder is provided. This folder can be directly imported into the Eclipse IDE for ModusToolbox™ and used. Follow these steps to import the project:

1. **Create a New Workspace Folder:**

   Create a new folder at your desired location on your system.

2. **Switch to the New Workspace:**
   a. In Eclipse IDE for ModusToolbox™, go to **File > Switch Workspace > Other**
   b. Click on **Browse**, select the newly created folder, and click **Launch** to switch to the new workspace
3. **Import the Project:**
   a. Go to **File > Import**, then expand the **General** category
   b. Select **Existing Projects into Workspace** and click **Next**
4. **Select the `MATLAB_FOC_PSOCC3` Folder:**
   Click **Browse** to select the root directory. Locate and select the `MATLAB_FOC_PSOCC3` folder, then click **Select Folder**
5. Copy the Project into the Workspace:
   a. Check the checkbox labeled **Copy projects into workspace**
   b. Click **Finish** to complete the import process
   c. If a pop-up appears asking to overwrite files, select **Yes to All**
6. **Retrieve Dependencies:**
   Open the terminal inside the IDE and run the following command to download all necessary libraries: **make getlibs.**

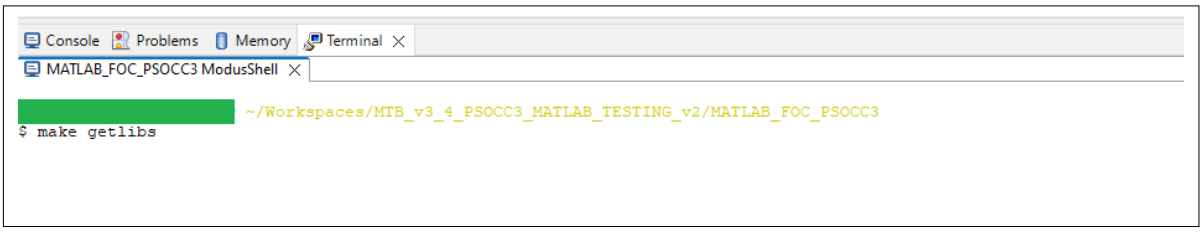

Figure 26 Terminal command

7. **Build the Project:**
   Click on the project folder in the **Project Explorer**, then **Build Project** to compile the code. It can then be flashed.

7 Testing

To ensure a smooth and correct process, follow these steps:

1. **Power Up the Hardware**
   - **a.** Ensure that boards are connected as shown in Figure 3
   - **b.** Configure the poiwer source of the drive card so that it is powered by the power board. To achieve this, move the jumper X20 on the drive card to the position that combines VDD5 and V5V.
   - **c.** Connect the drive card to PC via USB.
   - **d.** Power up the power board through a current limiting power source.
2. **Clean the Project:**

   Click on **Clean Project** in the **Quick Panel** to remove any previously compiled code or temporary files. This ensures a fresh build without issues from prior builds.
3. **Build the Project:**

   Click on **Build Project** to compile the source code. This step generates the executable code needed for the application.
4. **Start the Debug Session:**

   Click on the **Debug** button to initiate the debugging session and flash the compiled code onto the target hardware.
5. **Resume the Code**

8 Simulation results

The custom FOC algorithm is designed with a state machine that controls transitions between four distinct operational states. This section discusses the simulation results, with an emphasis on analyzing the scope outputs obtained from Simulink®.

State 1 *Rotor Alignment*: In the first state, a constant d-axis current (I_d) is supplied to the motor to align the rotor position with the stator's magnetic field. During this alignment phase, the currents remain constant and controlled by current controllers. The corresponding results are shown in the simulation plot in Figure 27 between the time interval of 0 to 200 ms.

State 2 *Open-Loop Operation*: After the alignment phase, the state machine transitions to the open-loop state. In this state, the motor begins to rotate, but the observer is not actively closed into the control loop; it runs passively in the background. During open-loop operation, both I_d and I_q currents are maintained at constant values. Open-loop state continues until the motor reaches a predefined speed of 400 RPM, as shown in Figure 27. Speed control is not active in this state.

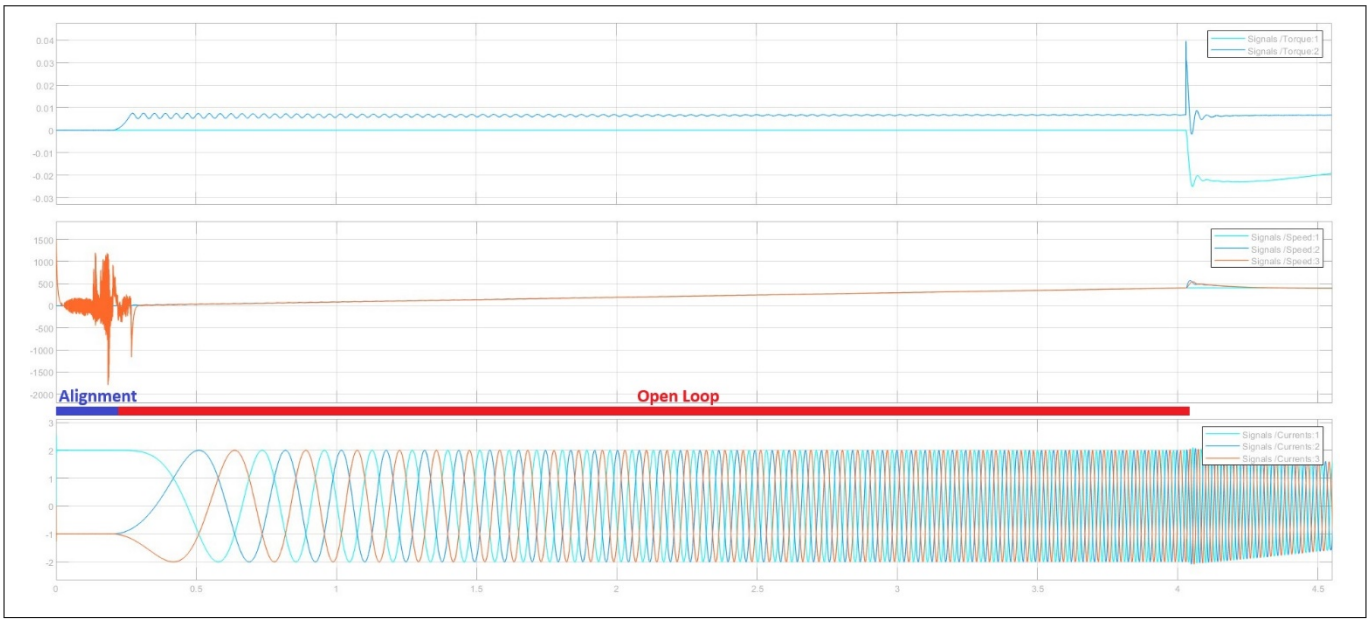

Figure 27 Alignment and Open Loop States

State 3 *Transition*: Once the motor achieves the target speed, the state machine enters an intermediate transition state, at which the speed loop is activated. During and after this stage, the speed feedback is obtained from the observer. The primary objective of this state is to gradually reduce the d-axis current (I_d) to zero in a controlled manner. This reduction in I_d results in visible decreases in the amplitudes of the sinusoidal current envelopes, as depicted in Figure 28. Throughout this transition, the motor speed is maintained at the predefined value by the state machine. The completion of this state occurs when I_d has reached zero.

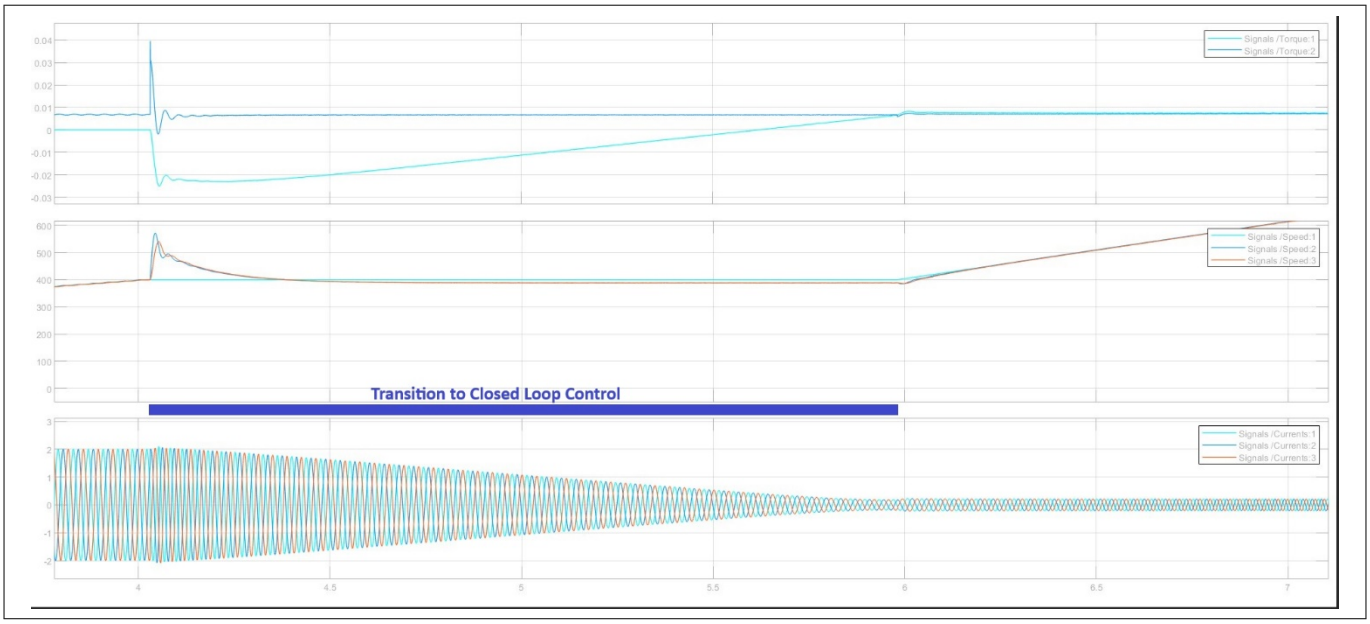

Figure 28 Transition State to Closed Loop Control

State 4 *Closed Loop Control*: In the final state, the FOC algorithm transitions to full closed-loop speed control. During this state, the state machine sets the motor's target speed to user provided value, which is 1000 RPM in the given example. In real-world hardware implementation, the target speed is dynamically adjusted by the user through a potentiometer during this state. The behavior of this state can be observed starting from the 6th second in Figure 29.

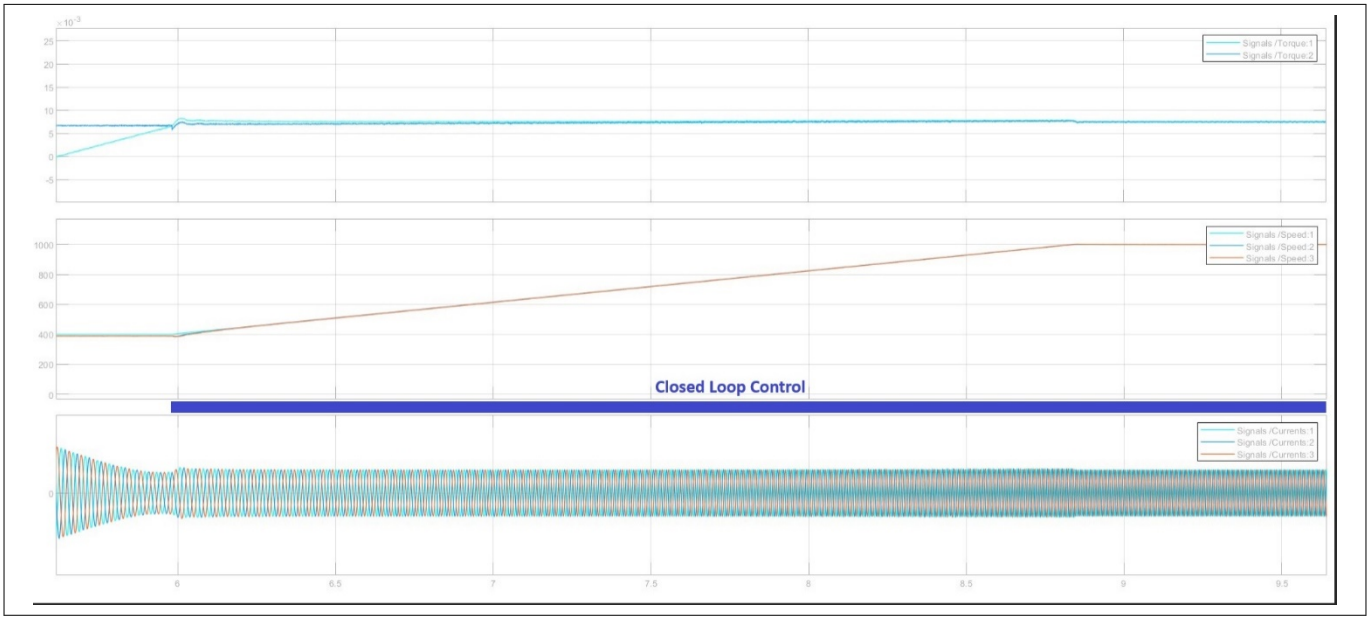

Figure 29 Closed loop state

References

[1] MathWorks: *Simulink®*; Available online
[2] MathWorks: *Code Generation*; Available online
[3] Infineon Technologies AG: *KIT_PSC3M5_MC1 PSOC™ Control C3 Motor Control Evaluation Kit guide*; Available online
[4] MathWorks: *Three-Phase PMSM Drive*; Available online

Revision history

Document revision	Date	Description of changes
**	2025-10-09	Initial release

Trademarks

PSOC™, formerly known as PSoC™, is a trademark of Infineon Technologies. Any references to PSoC™ in this document or others shall be deemed to refer to PSOC™.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2025-10-09

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2025 Infineon Technologies AG

All Rights Reserved.

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-pwq1754457598891

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.