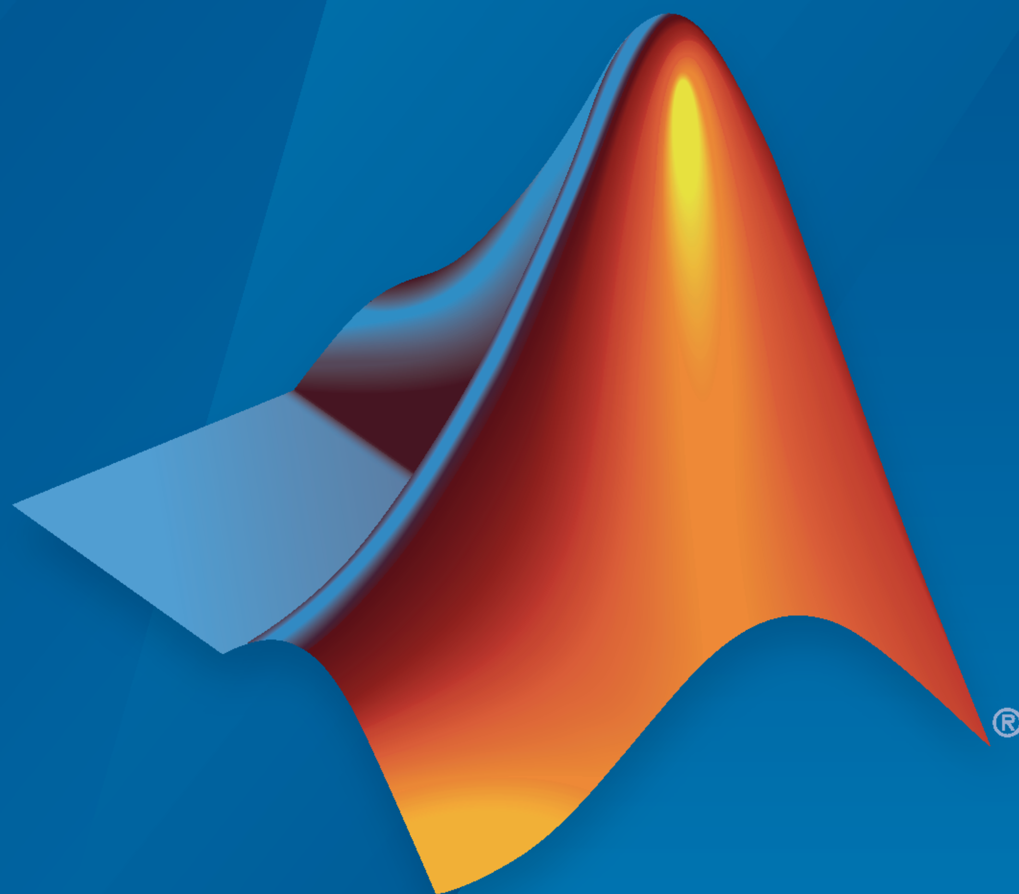
Motor Control Blockset™

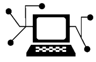
User's Guide



MATLAB® & SIMULINK®

R2020b

How to Contact MathWorks



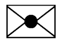
Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Motor Control Blockset™ User's Guide

© COPYRIGHT 2020 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2020	Online only	New for Version 1.0 (Release 2020a)
September 2020	Online only	Revised for Version 1.1 (Release R2020b)

Contents

1 Design the Controller

Design Field-Oriented Control Algorithm **1-2**

Design Current and Position Scaling Subsystems **1-3**

Design Current Controller Subsystem **1-6**

Perform Manual Gain-Tuning of Current Controller **1-10**

Design Speed Control Algorithm **1-13**

Perform Manual Gain-Tuning of Speed Controller **1-16**

Code Verification and Profiling Using Processor-In-the-Loop Testing . . **1-17**
- Required MathWorks Products **1-17**
- Supported Hardware **1-17**
- Prepare PIL Model **1-17**
- Verify Code by Using PIL **1-19**
- Analyze PIL Profiling Results **1-26**

2 Deploy and Validate System

Prepare Target Hardware **2-2**
- Verify Direction of Rotation of Motor **2-2**
- Calibrate Current Sensor **2-2**
- Calibrate Position Sensor **2-2**

Add Hardware Drivers to Simulation Model and Deploy to Target Hardware **2-4**

Task Scheduling in Target Hardware **2-6**

Adding ADC Driver Library Block **2-8**

Adding Quadrature Encoder Driver Block **2-10**

Add PWM Driver Block **2-12**

Add Hardware Interrupt Trigger Block for Current Control Loop **2-15**

Run in Open-Loop and Switch to Closed-Loop **2-17**

Model Configuration and Hardware Deployment **2-20**

Validate System **2-22**
Calculate Physical Motor Load in Target Hardware **2-23**
Compare Speed Controller Response in Simulation and in Target Hardware **2-24**
Compare Current Controller Response in Simulation and in Target Hardware **2-26**

3 Plant Modeling

Creating Plant Model Using Motor Control Blockset **3-2**

Use PMSM Block and Motor Parameters to Design Plant Model **3-3**

Add Average-Value Inverter Block **3-5**

Create Motor Phase Current Sensing and Signal Conditioning Subsystem **3-6**

Create Position Sensing Subsystem **3-7**

Add Delay in Plant Model **3-8**

Integrate the Blocks and Subsystems **3-9**

4 Hardware Troubleshooting

Check ADC Inputs **4-2**
Description **4-2**
Action **4-2**

Verify PWM Outputs **4-4**
Description **4-4**
Action **4-4**

Check Hardware Connections **4-6**
Description **4-6**
Action **4-6**

Test Algorithm Design **4-7**
Description **4-7**
Action **4-7**

Check Generated Code **4-8**
Description **4-8**
Action **4-8**

Design the Controller

- “Design Field-Oriented Control Algorithm” on page 1-2
- “Design Current and Position Scaling Subsystems” on page 1-3
- “Design Current Controller Subsystem” on page 1-6
- “Perform Manual Gain-Tuning of Current Controller” on page 1-10
- “Design Speed Control Algorithm” on page 1-13
- “Perform Manual Gain-Tuning of Speed Controller” on page 1-16
- “Code Verification and Profiling Using Processor-In-the-Loop Testing” on page 1-17

Design Field-Oriented Control Algorithm

To implement the speed control algorithm for a motor, perform these tasks:

- Current scaling — Convert current from ADC counts to PU
- Quadrature encoder position decoding — Read the quadrature encoder position counts and calculate the rotor electrical position
- Torque control — Current control in *d*-*q* axis
- Speed control

These steps help you implement the speed control algorithm for a PMSM using Motor Control Blockset and are related to the model `mcb_pmsm_foc_qep_f28379d` used in the example “Field-Oriented Control of PMSM Using Quadrature Encoder”. They explain the steps to tune the control parameters for *d*-axis and *q*-axis current controllers and the speed controller.

1. “Design Current and Position Scaling Subsystems” on page 1-3
2. “Design Current Controller Subsystem” on page 1-6
3. “Perform Manual Gain-Tuning of Current Controller” on page 1-10
4. “Design Speed Control Algorithm” on page 1-13
5. “Perform Manual Gain-Tuning of Speed Controller” on page 1-16

In these steps, variables are used to define datatypes and execution times of the current and speed controllers. See the initialization script linked to the example model `mcb_pmsm_foc_qep_f28379d` for details on the variables defined in these steps.

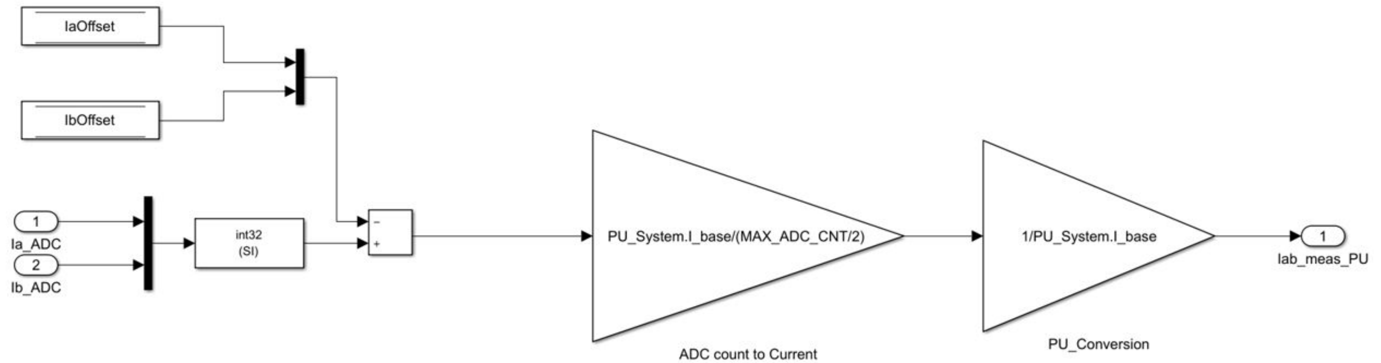
Tip A basic understanding of Simulink® is a prerequisite for following this workflow as these workflow steps do not provide details on tasks like defining a datatype in a constant block or using math operations blocks in Simulink.

See “Estimate Motor Parameters by Using Motor Control Blockset Parameter Estimation Tool” for estimating the motor parameters. Then, see “Creating Plant Model Using Motor Control Blockset” on page 3-2 to design a plant model. This helps you verify the control algorithm in simulation.

Design Current and Position Scaling Subsystems

Use these steps to design the current and position scaling subsystems:

1 Create the current scaling subsystem.



This subsystem reads the current in ADC counts and converts it to per-unit (PU) values.

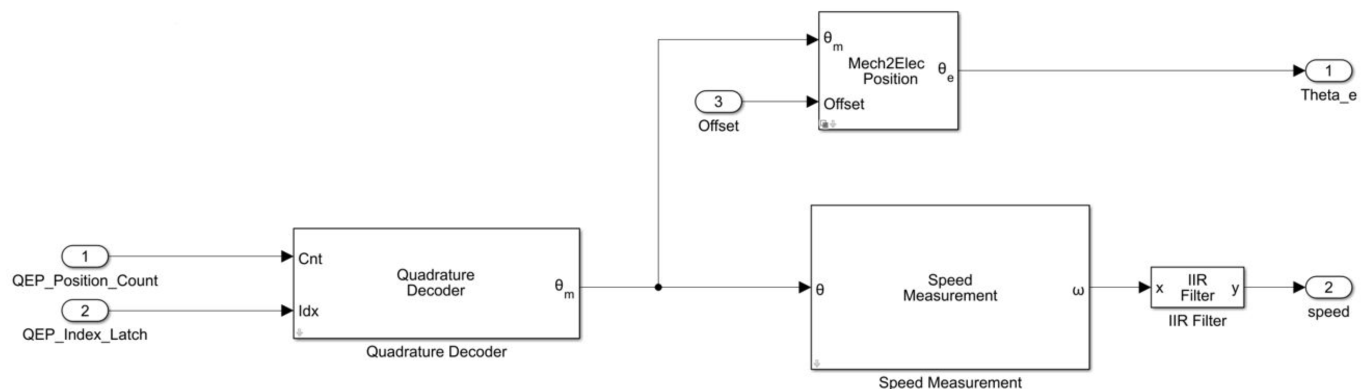
In this subsystem, the *IaOffset* and *IbOffset* Data Store Memory blocks are the ADC offsets for current measurement and they are hardware specific. The file `mcb_SetInverterParameters.m` contains the default ADC offset (*CtSensAOffset* and *CtSensBOffset*) for few commercially available inverters. For details about ADC offset calibration in hardware, see “Run 3-Phase AC Motors in Open-Loop Control and Calibrate ADC Offset”.

In this subsystem, the motor phase current measured in ADC counts is converted to current in PU. The `PU_System.I_base` value refers to the base current in this subsystem. For details about the PU system, see “Per-Unit System”. See the `mcb_SetPUSystem.m` file that computes the PU values for the system.

You can use the base values for computing the real-world values from per-unit. To implement the real-world or SI unit values, see the model `mcb_pmsm_foc_qep_f28379d_SIUnit` used in the example “Field Oriented Control of PMSM by Using SI Units”.

The *IaOffset* and *IbOffset* Data Store Memory blocks are used to share data between the current and position subsystems.

2 Create the position scaling subsystem.

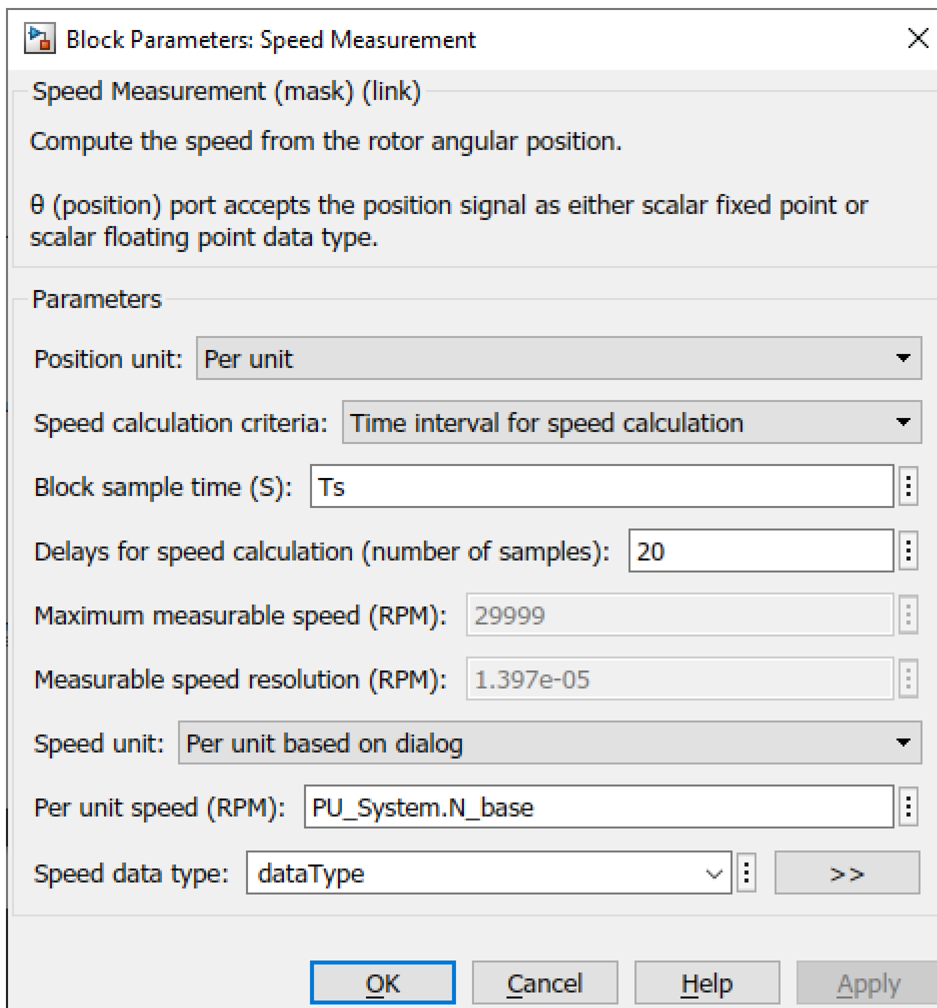


This subsystem reads the rotor position from the QEP pulse count.

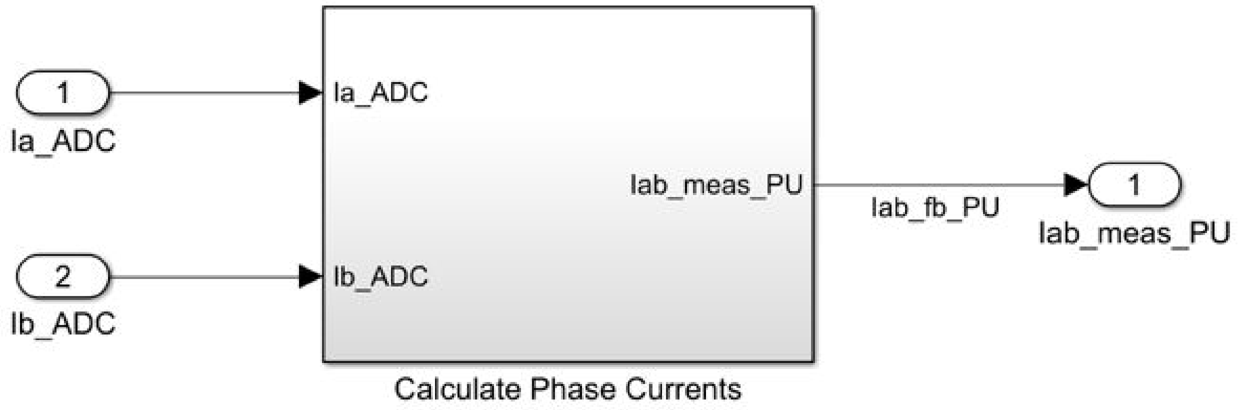
In this subsystem, the Quadrature Decoder block reads the position count from the plant model or hardware driver block. The block converts the rotor mechanical position in encoder position counts to rotor mechanical angle in PU (`0-1`).

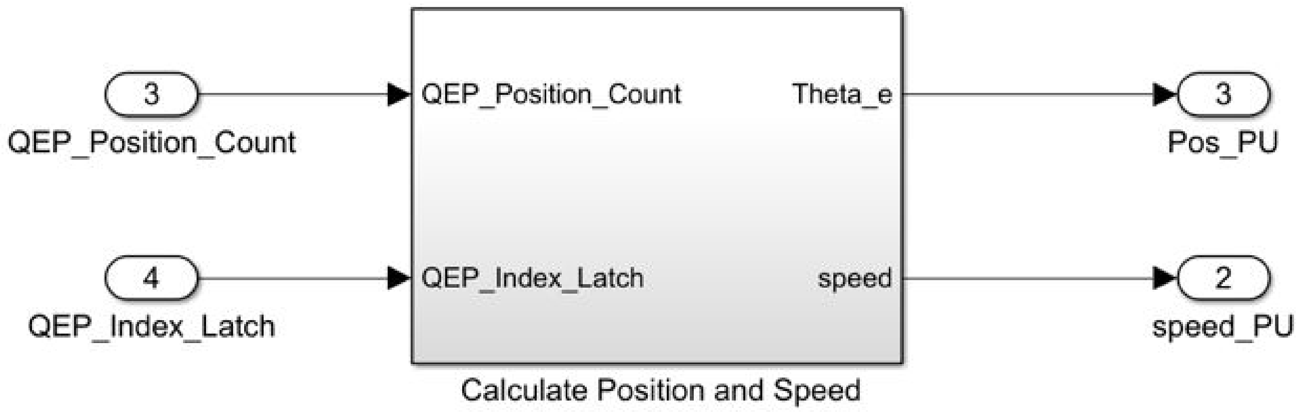
The Mechanical to Electrical Position (Mech2Elec Position) block adjusts the mechanical angle for QEP offset and converts it to electrical angle. This rotor electrical angle is required for the FOC algorithm to spin the motor. Refer to “Quadrature Encoder Offset Calibration for PMSM Motor” for calculating the QEP encoder offset.

The Speed Measurement block calculates speed from the rotor position. In the Speed Measurement block parameters dialog box, set the **Delays for speed calculation (number of samples)** parameter to `20`. We selected the value `20` in this workflow so that the block can measure the maximum speed of the motor that is under test. The Speed Measurement block outputs the speed in PU.



These subsystems that you create includes the current scaling and position decoding logic.

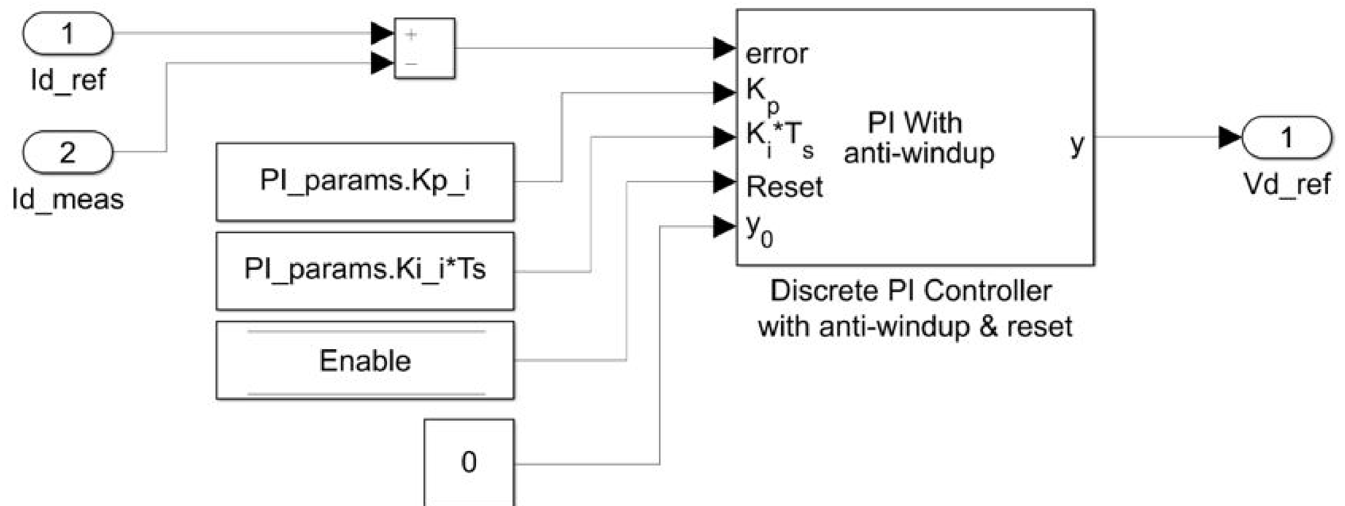

1
Ia_ADC
2
Ib_ADC
Ia_ADC
Ib_ADC
Iab_meas_PU
Iab_fb_PU
1
Iab_meas_PU
Calculate Phase Currents


3
QEP_Position_Count
4
QEP_Index_Latch
QEP_Position_Count
QEP_Index_Latch
Theta_e
speed
3
Pos_PU
2
speed_PU
Calculate Position and Speed

Design Current Controller Subsystem

Use these steps to design the current controller subsystem:

1. From the Motor Control Blockset library in the Simulink Library Browser, use the Discrete PI Controller with anti-windup & reset block (in the Controls/Controllers library) to design the *d*-axis and *q*-axis current control.



The MATLAB function `mcb.internal.SetControllerParameters` (in the model initialization script) calculates the PI control gains for the *d*-axis and *q*-axis current controller and speed controller. For details regarding the control parameter gain estimation, see “Estimate Control Gains from Motor Parameters”. See the model initialization script file `mcb_pmsm_foc_qep_f28379d_data.m` (used in the example “Field-Oriented Control of PMSM Using Quadrature Encoder”) for the sampling time (T_s) of `50`µs.

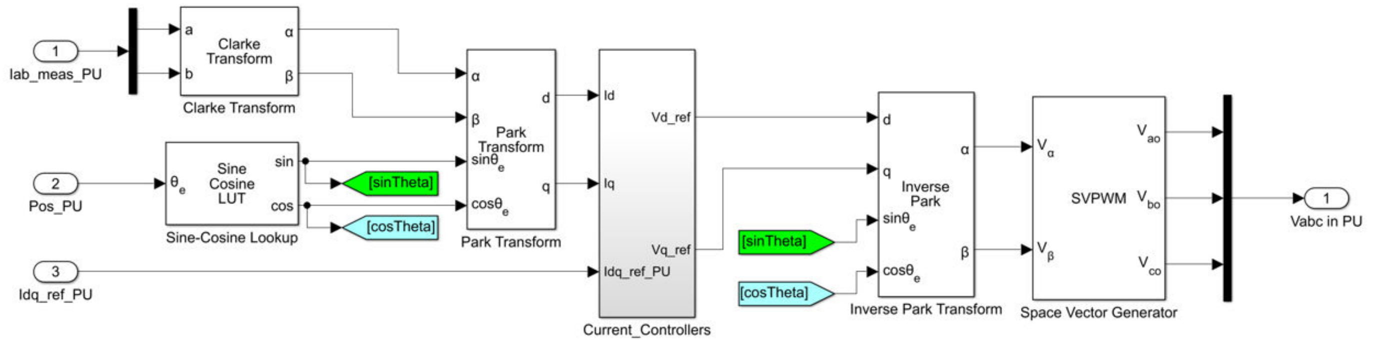
In the subsystem diagram, the *Enable* variable is a Data Store Memory used to reset the controller. Adding *Enable* variable is optional.

The subsystem also uses three constant blocks with these values:

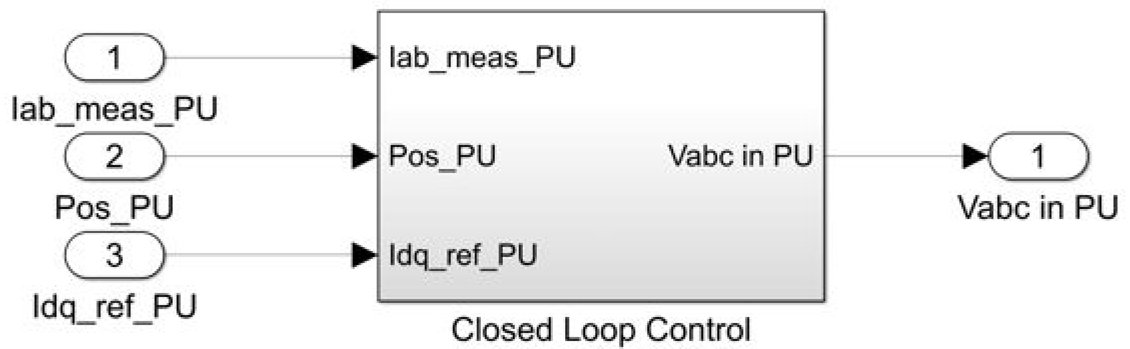
- *PI_params.Kp_i*
- *PI_params.Ki_i*Ts*
- *0*

Create a similar subsystem for the *q*-axis PI controller. Integrate the subsystems for *d*-axis and *q*-axis PI controllers into a single subsystem (Current_Controllers) that controls the *d*-axis and *q*-axis currents.

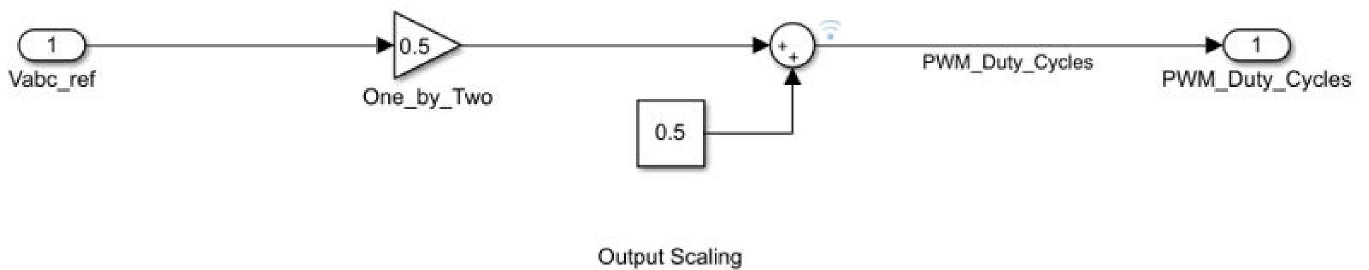
2. Add the Clarke Transform, Park Transform, Inverse Park Transform, and Space Vector Generator blocks from the `Motor Control Blockset/Controls/Math Transforms` library to the Current_controllers subsystem (that you created in step 1) as shown in this figure.



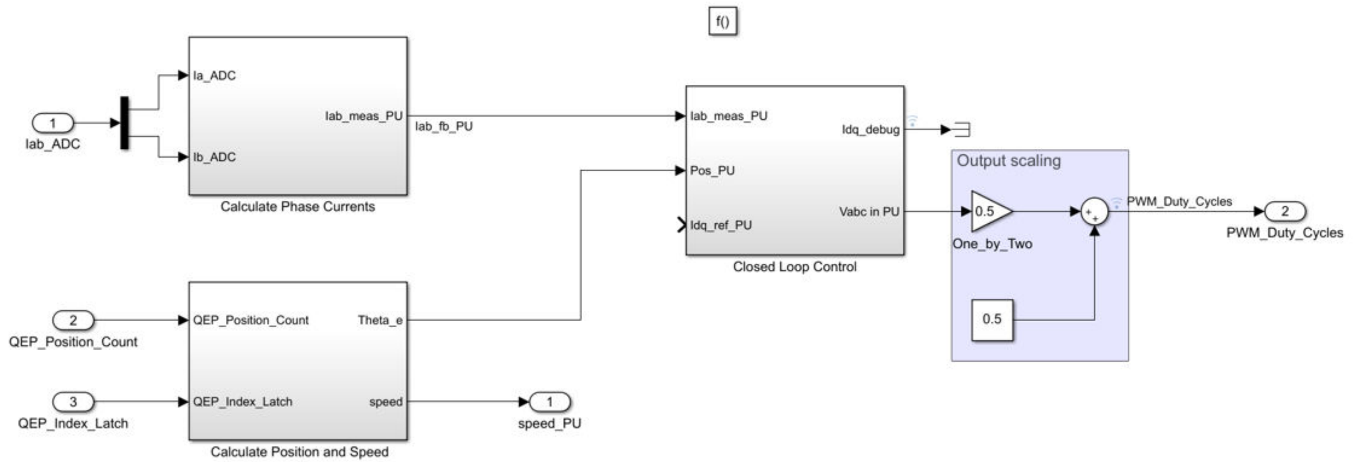
3 Integrate the components that you created in step 2 into a single subsystem (Closed Loop Control that implements closed loop field-oriented control) as shown in this figure.



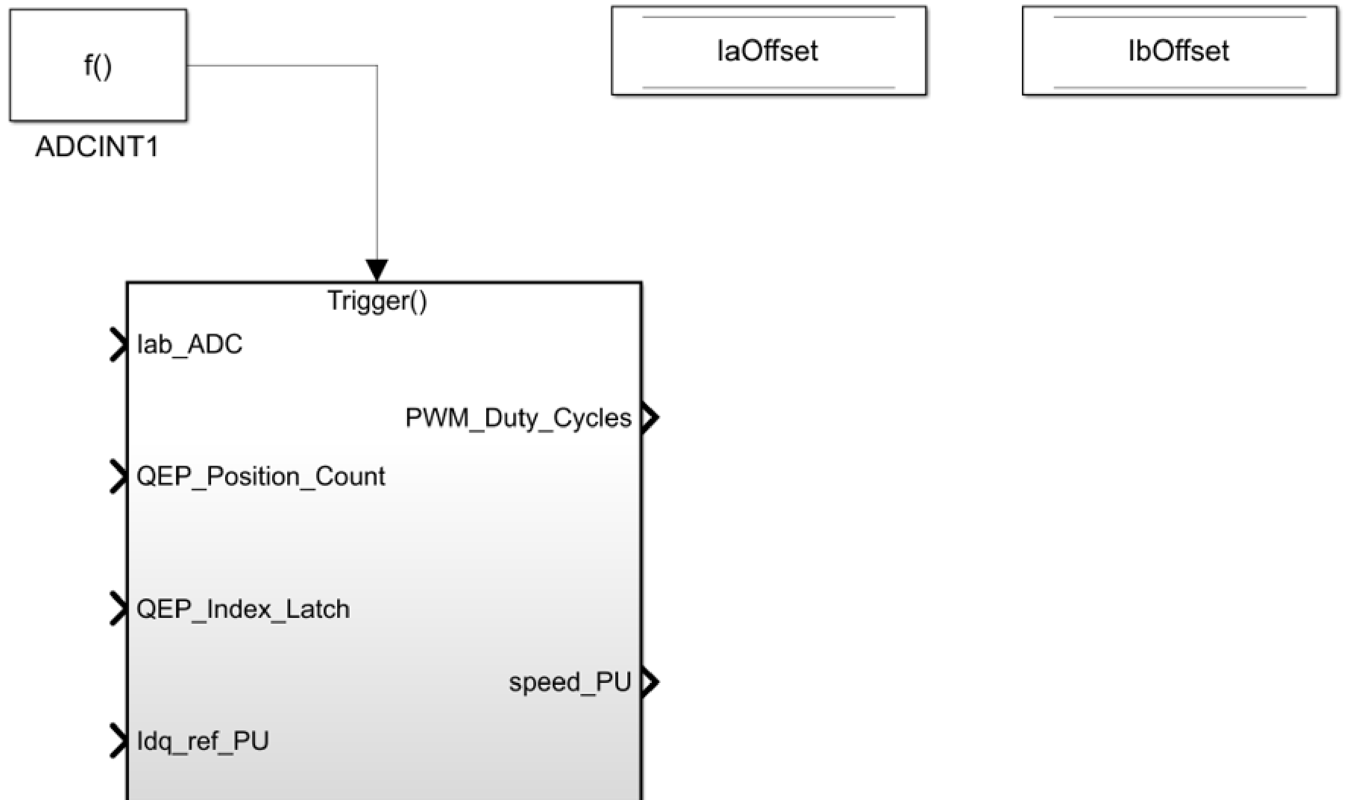
4 Create an Output scaling subsystem to scale the Pulse Width Modulation (PWM) outputs. This subsystem outputs the normalized PWM duty cycles (`0-1`) for the plant model.



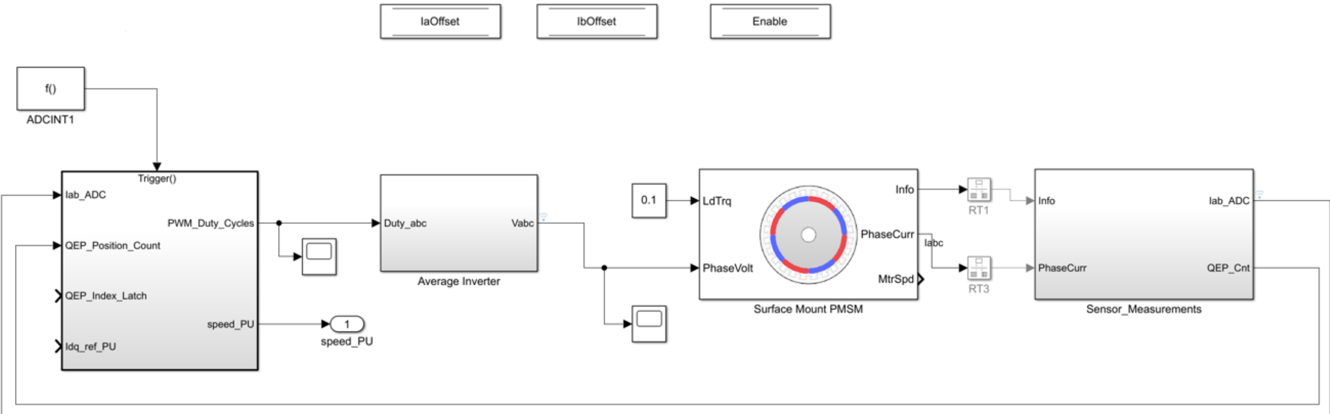
5 Create a new subsystem by integrating the current scaling, QEP position decoding, Closed Loop Control, and Output scaling subsystems. Add the trigger block from the `Simulink/Ports & Subsystems` library to this subsystem and set the **Trigger type** parameter to `function-call`.



6 Add a Function-Call Generator block from the `Simulink/Ports & Subsystems` library to the subsystem created in step 5. Set the **Sample time** parameter of the block to equal the control-loop sample time, T_s (that has a default value of `50e-6` s).



7 Integrate the plant model and the controller subsystem that you created in step 6. For detailed steps on how to create a plant model for a motor control system, see “Creating Plant Model Using Motor Control Blockset” on page 3-2.


IaOffset
IbOffset
Enable
f()
ADCINT1
Trigger()
Iab_ADC
PWM_Duty_Cycles
QEP_Position_Count
QEP_Index_Latch
speed_PU
Idq_ref_PU
Duty_abc
Vabc
Average Inverter
1
speed_PU
0.1
LdTrq
PhaseVolt
Info
PhaseCurr
MtrSpd
Iabc
Surface Mount PMSM
RT1
RT3
Info
PhaseCurr
Iab_ADC
QEP_Cnt
Sensor_Measurements

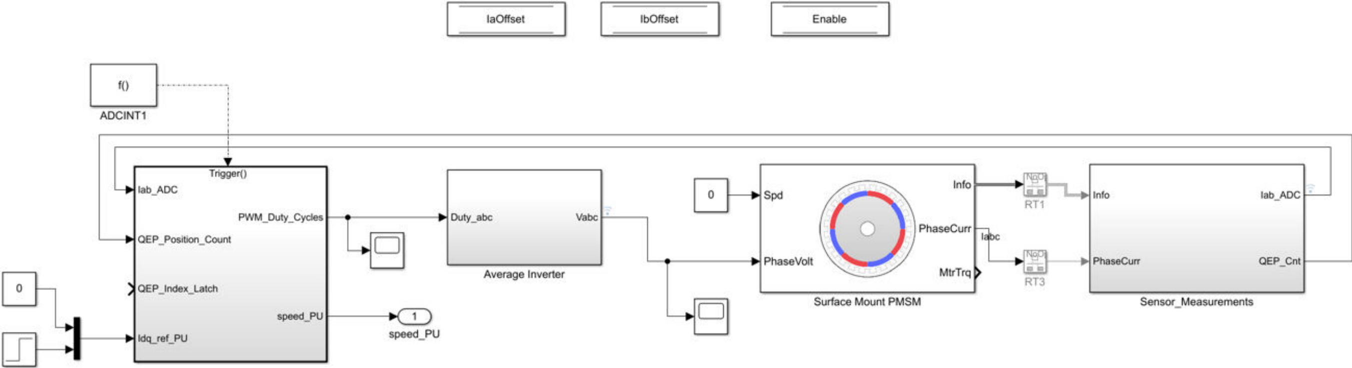
Perform Manual Gain-Tuning of Current Controller

In this step, you perform gain tuning for the *d*-axis and *q*-axis current controller manually.

This step is optional, but helps you to tune the control gain parameters for the current controller. Provide step change for `Id_ref` and analyze the current controller performance from the step response of `Id_meas`. Repeat the same process for `Iq_ref` to tune the *q*-axis current controller. In the plant model, lock the rotor to ensure that the motor does not spin when you provide a step change for `Id_ref` or `Iq_ref`. In the Surface Mount PMSM block parameters dialog box, set the **Mechanical input configuration** parameter to `Speed`. Provide the **Spd** input (of the Surface Mount PMSM block) as `0` to ensure that the rotor is locked.

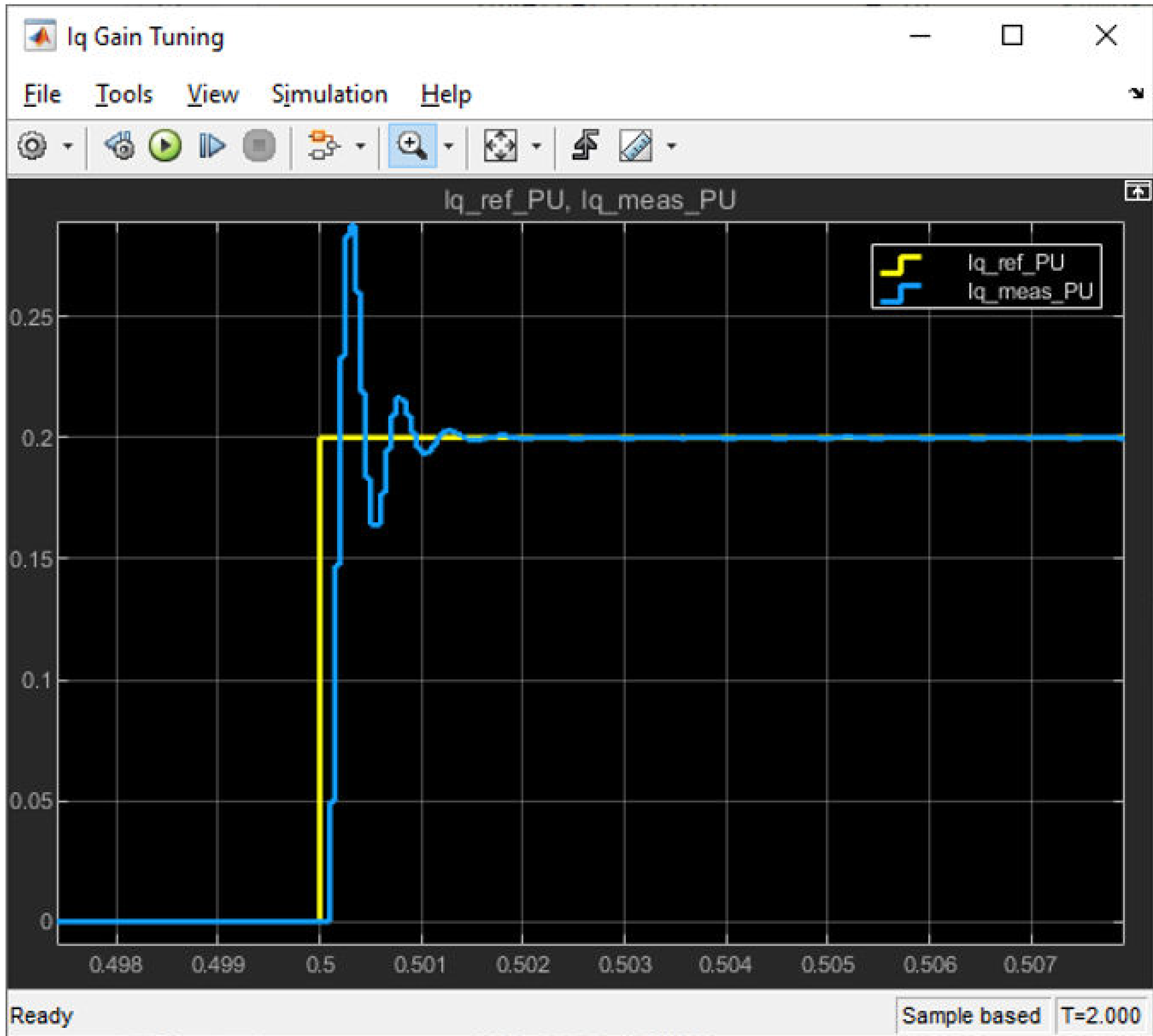
Block Parameters: Surface Mount PMSM
Surface Mount PMSM (mask) (link)
Model the dynamics of a three-phase surface mount permanent magnet synchronous motor (PMSM) with sinusoidal back electromotive force.
Block Options
Mechanical input configuration: Speed
Simulation type: Discrete
Sample Time (Ts): Ts_motor
Load Parameters:
File:
Browse
Load from file
Save to file
Parameters
Initial Values
Number of pole pairs (P): pmsm.p
Stator resistance per phase (Rs): pmsm.Rs [Ohm]
Stator d-axis inductance (Ldq_): pmsm.Ld [H]
Permanent flux linkage constant (lambda_pm): pmsm.FluxPM [Wb]
OK
Cancel
Help
Apply

The integrated plant and controller subsystem simulation model allows you to manually tune the gains of the current controller. Provide a step input to `Iq_ref` in the range (`0` to `0.2`) PU and observe the measured feedback. Adjust the control parameters of the current controller to meet your control objectives.



Simulate the model and plot the `Iq_ref_PU` and `Iq_meas_PU` and analyze the step response. This allows you to tune the control parameters for the *q*-axis controller to meet the control objectives.

Iq_ref_PU, Iq_meas_PU



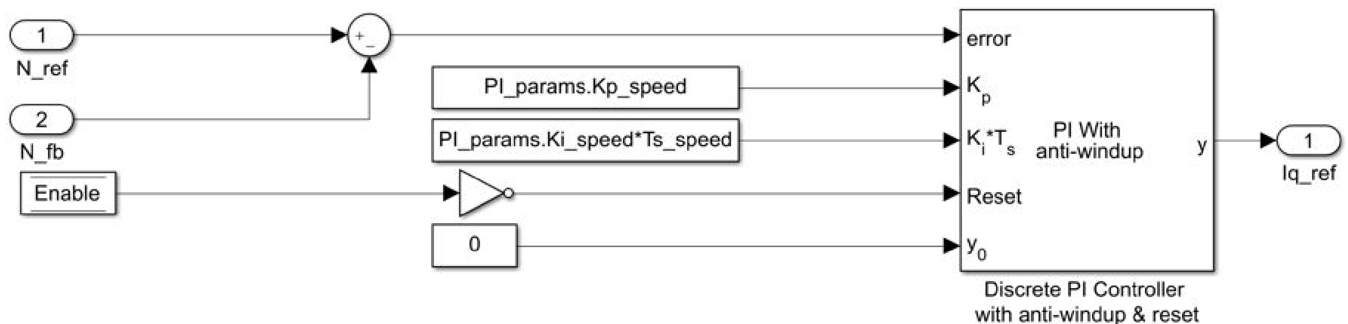
You can follow the same method for tuning the *d*-axis current controller. In the Surface Mount PMSM block parameters dialog box, set the **Mechanical input configuration** parameter to `Torque`.

Design Speed Control Algorithm

To design a speed control algorithm:

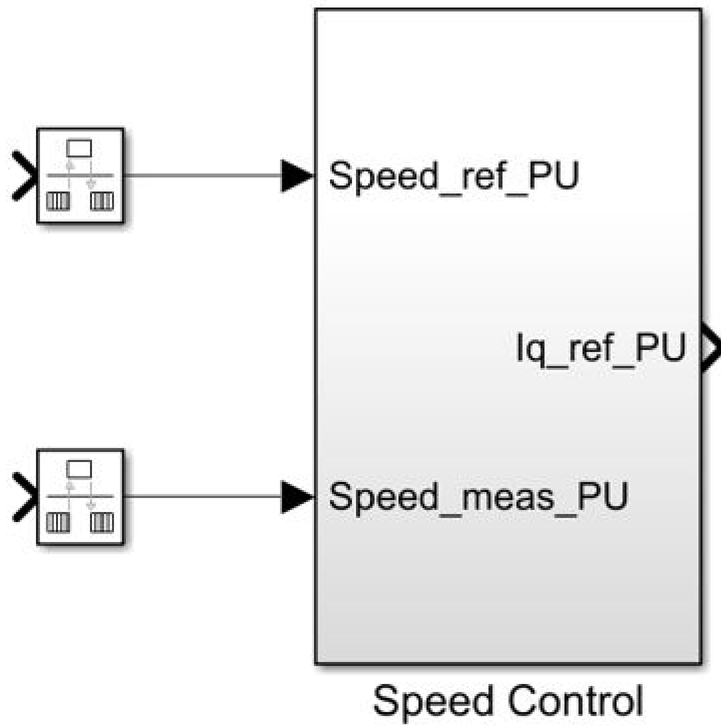
1. Create a speed controller subsystem. The `Iq_ref` current output of the speed controller subsystem is used as an input to the current controller subsystem that you created earlier.

   To create a speed controller subsystem, from the Simulink Library Browser, select the Discrete PI Controller with anti-windup & reset block from the `Motor Control Blockset/Controls/Controllers` library.

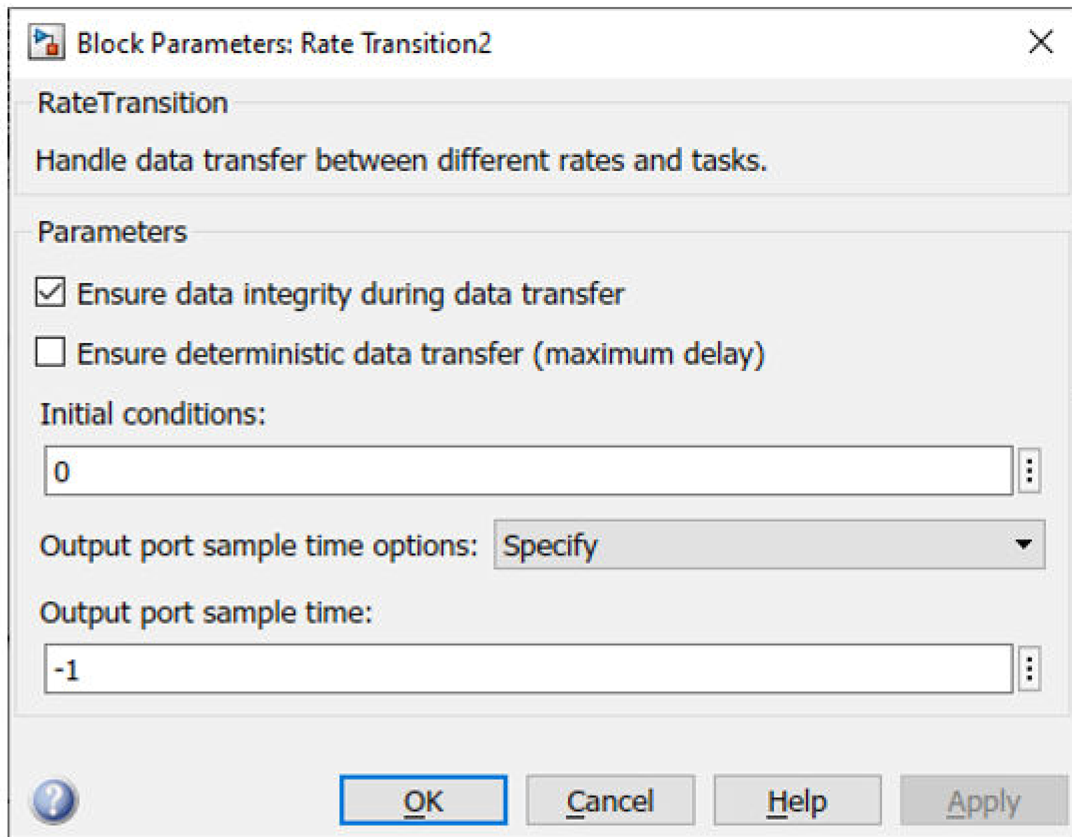


   The MATLAB® function `mcb.internal.SetControllerParameters` (in the model initialization script) calculates the PI control gains for the *d*-axis and *q*-axis current controller and speed controller. For details regarding the control parameter gain estimation, see “Estimate Control Gains from Motor Parameters”. See the model initialization script file `mcb_pmsm_foc_qep_f28379d_data.m` (used in the example “Field-Oriented Control of PMSM Using Quadrature Encoder”) for the sampling time (*Ts_speed*) of `500`µs. Use **Enable** Data-Store Memory block to reset the controller but this is optional.

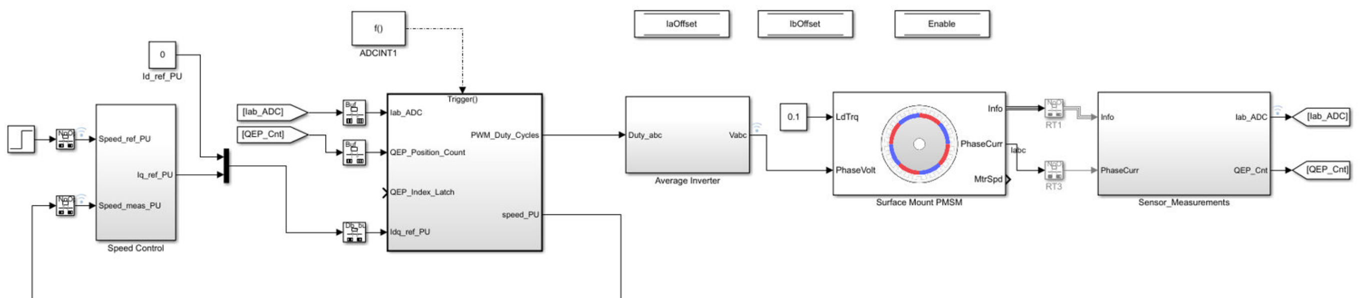
2. Create a subsystem for the speed controller and add a Rate Transition block (from the `Simulink/Signal Attributes` library) to the inputs with the sample time as *Ts_speed* (execution time of the speed control loop).



3. Integrate the Speed Controller subsystem with the integrated Current Controller and plant model subsystems. Connect the **Iq_ref_PU** output port of the Speed Controller subsystem to the Current Controller subsystem input port through a Rate Transition block as both ports execute in different sample rates. This figure shows an example of settings of the Rate Transition block connected to the Speed Controller and Current Controller subsystems.

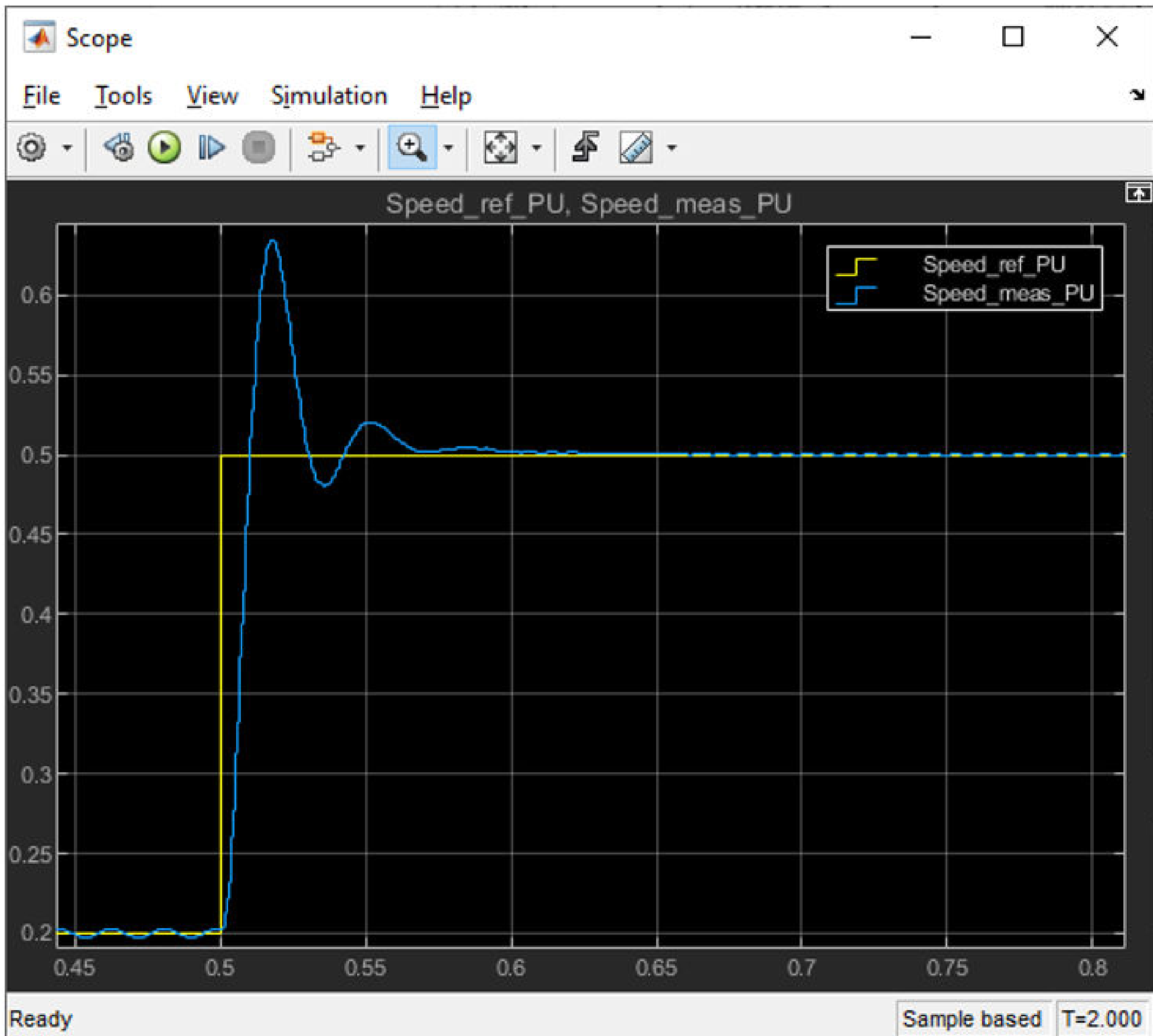


This figure shows the integrated Speed Controller, Current Controller, and plant model subsystems.



Perform Manual Gain-Tuning of Speed Controller

To manually tune the Speed Controller subsystem, add a step input (in the range `0.2` to `0.5` PU) to the **Speed_ref_PU** input in Speed Controller subsystem. Monitor the speed step response and tune the Speed Controller subsystem parameters. This figure shows the step response of the speed controller.



The preceding procedure provides an approach to implement speed control for a PMSM motor in simulation. Run the simulation and analyze the controller performance.

You can generate the C code from this control algorithm using Embedded Coder®. You can deploy this code and the hardware drivers to the target hardware.

Code Verification and Profiling Using Processor-In-the-Loop Testing

In the processor-in-the-loop (PIL) simulation, the control algorithm executes in the target hardware, but the plant model runs on the host machine. The plant model (running on the host machine) simulates the input and output signals for the controller (running on the target hardware) and communicates with the controller by using the serial communication interface. This functionality allows you to use PIL simulation to determine the execution time on the target hardware, which you can then compare with the execution time for simulating the model on the host machine.

The execution time or the performance metrics of an algorithm that you obtain from PIL simulation, helps you to detect algorithm overrun on the target hardware. The PIL profiling report indicates the average and maximum execution times of an algorithm on the target hardware. This example explains PIL profiling on the Texas Instruments™ LAUNCHXL-F28379D hardware board.

We use an example model `mcb_pmsm_foc_sim.slx` to demonstrate code verification in PIL simulation. This example shows PIL profiling for the Current Control subsystem in the example model. This subsystem includes the Field-Oriented Control (FOC), current scaling (per-unit conversion), speed measurement, and rotor position scaling (computation of angle from the encoder position counts) algorithms. The PIL profiling report shows the average execution and maximum execution times of the control algorithm in the target hardware.

This section addresses these tasks:

- Verify code execution by using PIL testing by comparing the algorithm in the simulation and target hardware operating modes.
- Perform PIL profiling by measuring the algorithm execution time in the target hardware and generate the PIL profiling report.

Required MathWorks Products

- Embedded Coder
- Embedded Coder Support Package for Texas Instruments C2000™ Processors

Supported Hardware

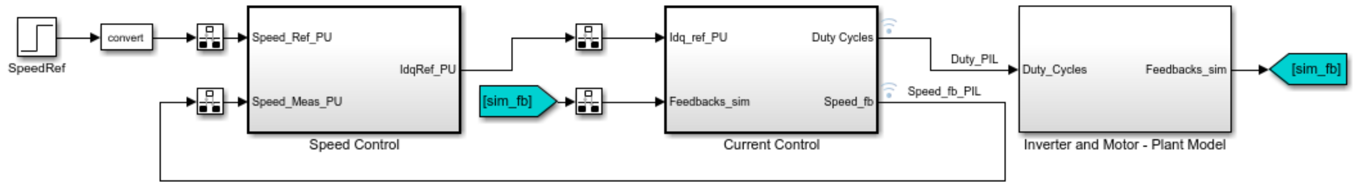
- LAUNCHXL-F28379D controller hardware board

Prepare PIL Model

Use these steps to prepare the PIL model for profiling:

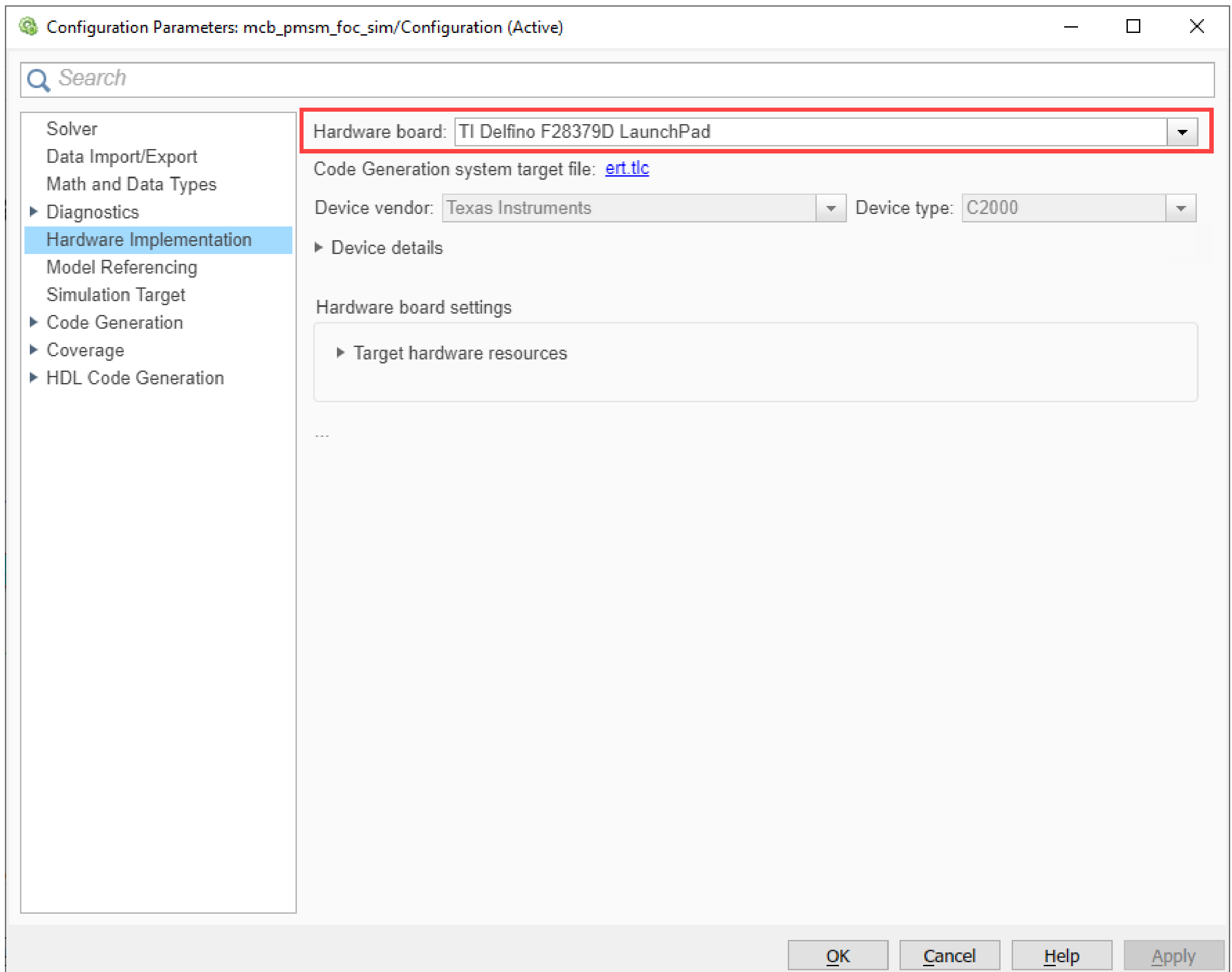
1 Open the model `mcb_pmsm_foc_sim.slx` by using this command:

```
open_system('mcb_pmsm_foc_sim.slx');
```



This model simulates the PMSM motor and FOC algorithm for closed-loop speed control.

2. On the **Hardware** tab of the Simulink toolstrip, click **Hardware Settings**.
3. In the **Configuration Parameters** window, under **Hardware Implementation**, set the **Hardware board** field to `TI Delfino F28379D LaunchPad`.

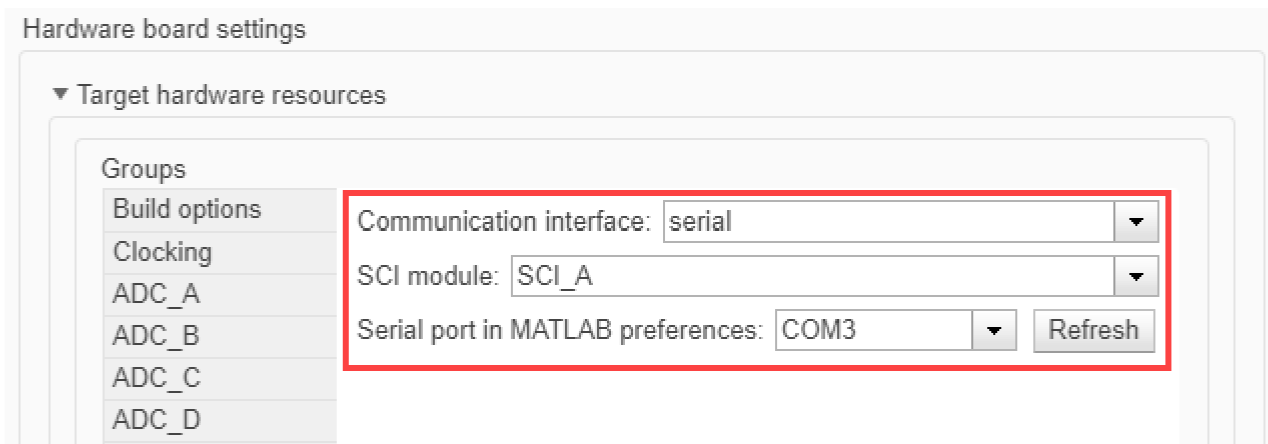


Verify Code by Using PIL

Use these steps to verify the code in PIL:

1 In the Configuration Parameters dialog box, select these configuration settings under **Hardware Implementation** > **Hardware board settings** > **Target hardware resources** > **PIL**:

- **a** **Communication Interface** - Select `serial`.
- **b** **SCI module** - Select `SCI_A`.
- **c** **Serial port in MATLAB preferences** - Model automatically detects the communication port to which you have connected the hardware. This parameter remains unchanged for the rest of the currently active MATLAB session. Click the **Refresh** button to detect the communication port again.



2 Open the script file `mcb_PIL_config_TI.m` to set the configuration parameters:

```
edit('mcb_PIL_config_TI.m');
```

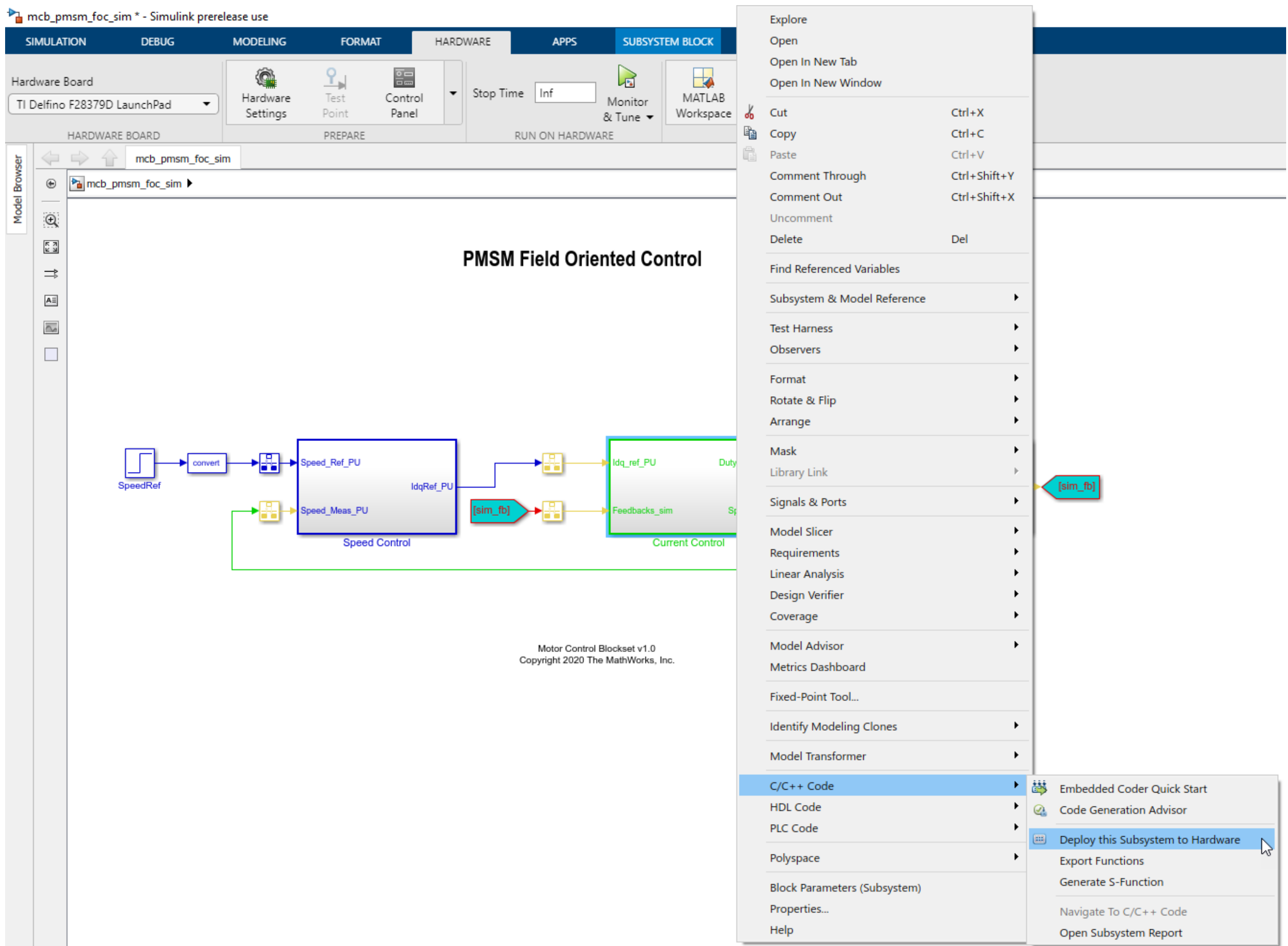
3 Update the model name and stop time in the script.

mcb_PIL_config_TI.m

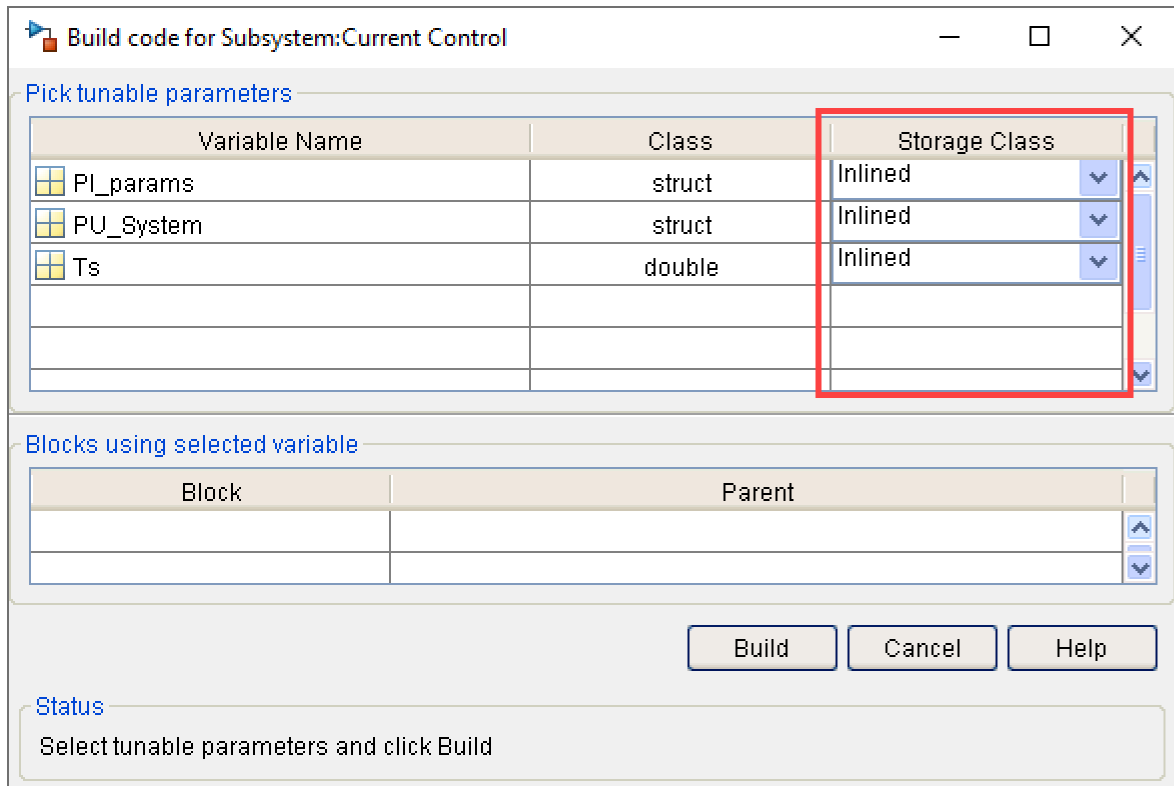
```
1      % mcb_PIL_config_TI initializes the congiuration parameter for PIL profiling
2      %
3      % For PIL profiling, update the below in MATLAB script
4      % model - name of the model identified for PIL profing
5      % StopTime - time required for profiling. Ensure algorithm reaches steady
6      % state within this specified time.
7      %
8      % This code is tested for TI LAUNCHXL-F28379D (TMS320F28379d)
9      %
10     % Note: Before running the script, ensure COM port is updated in
11     % config set->hardware implementation->Target hardware resources->PIL
12     %
13     % Copyright 2020 The MathWorks, Inc.
14
15 -   model = 'mcb_pmsm_foc_sim';
16 -   set_param(model,'StopTime','0.5');
17 -   set_param(model,'SimulationMode','normal');
18 -   set_param(model,'ReturnWorkspaceOutputs','on');
19 -   set_param(model,'CodeExecutionProfiling','on');
20 -   set_param(model,'CodeProfilingInstrumentation','coarse');
21 -   set_param(model,'CodeProfilingSaveOptions','SummaryOnly');
22 -   set_param(model,'CreateSILPILBlock','PIL');
23 -   set_param(model,'DefaultParameterBehavior','Inlined');
24
```

4 Run the script to update the configuration parameters of the simulation model and the PIL preferences.

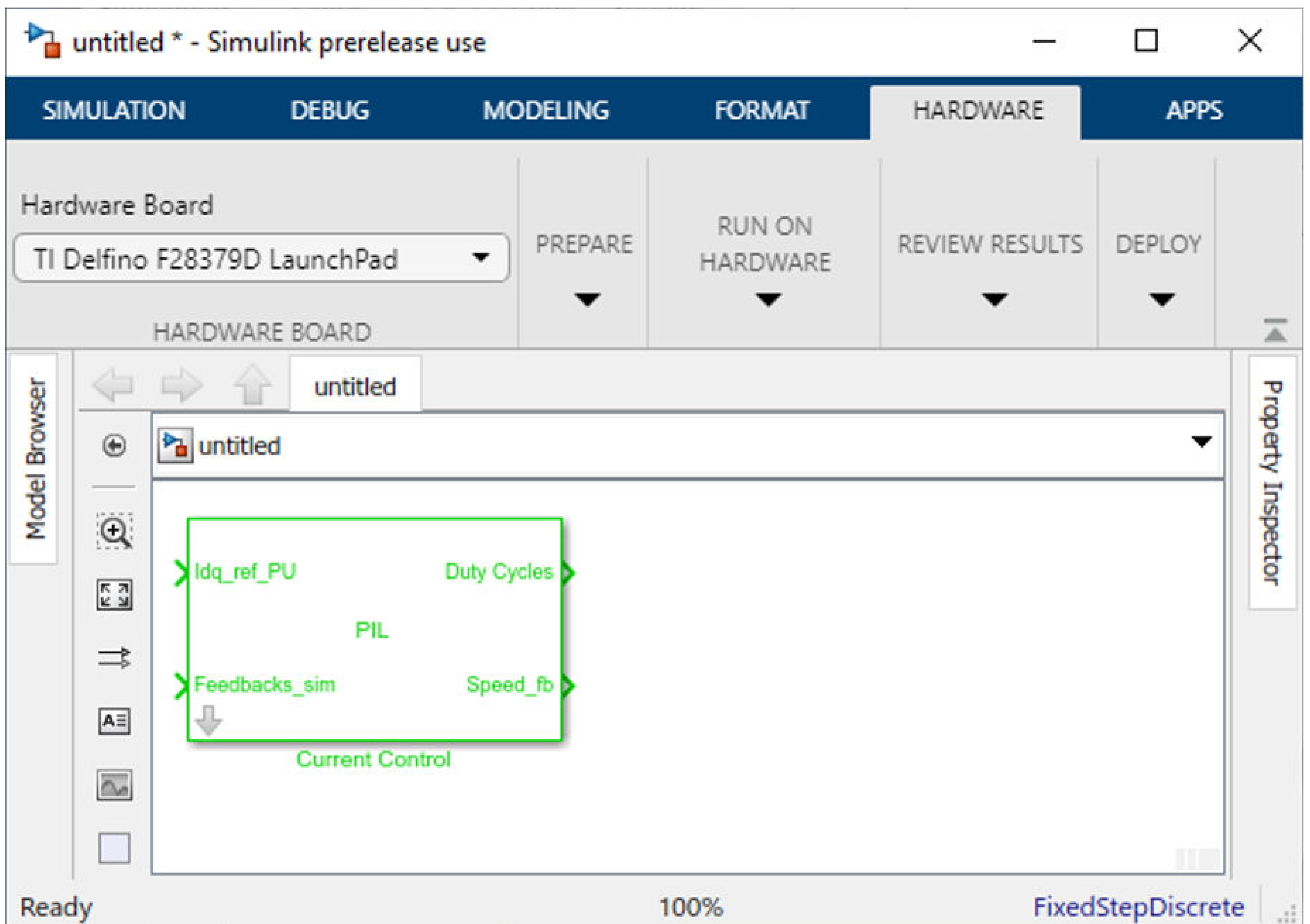
5 Right-click the Current Control subsystem in the `mcb_pmsm_foc_sim.slx` example model. Under the **C/C++ Code** menu, select **Deploy this Subsystem to Hardware**.



The system displays the **Build code for Subsystem** dialog box. Set the Storage Class to `Inlined` for all parameters.



6 Click **Build** to create a model named `untitiled` that includes a PIL subsystem called Current Control.



7 Rename the Current Control subsystem to Current Control (PIL).

untitled * - Simulink prerelease use

SIMULATION DEBUG MODELING FORMAT HARDWARE APPS BLOCK

Hardware Board

TI Delfino F28379D LaunchPad

HARDWARE BOARD

PREPARE

RUN ON HARDWARE

REVIEW RESULTS

DEPLOY

Model Browser

untitled

untitled

Idq_ref_PU

Duty Cycles

PIL

Feedbacks_sim

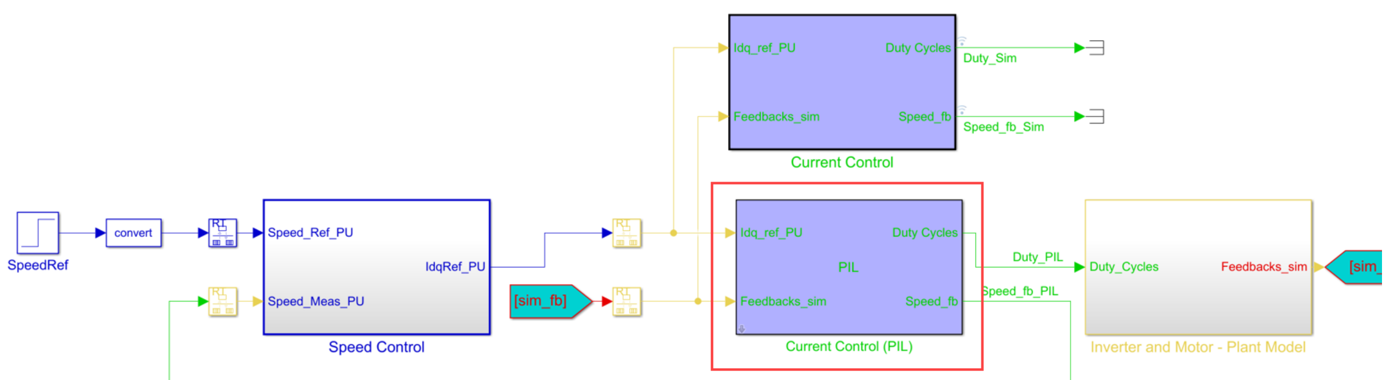
Speed_fb

Current Control (PIL)

Property Inspector

8 Copy the Current Control (PIL) subsystem and replace the Current Control subsystem in the `mcb_pmsm_foc_sim.slx` example model.

PMSM Field Oriented Control

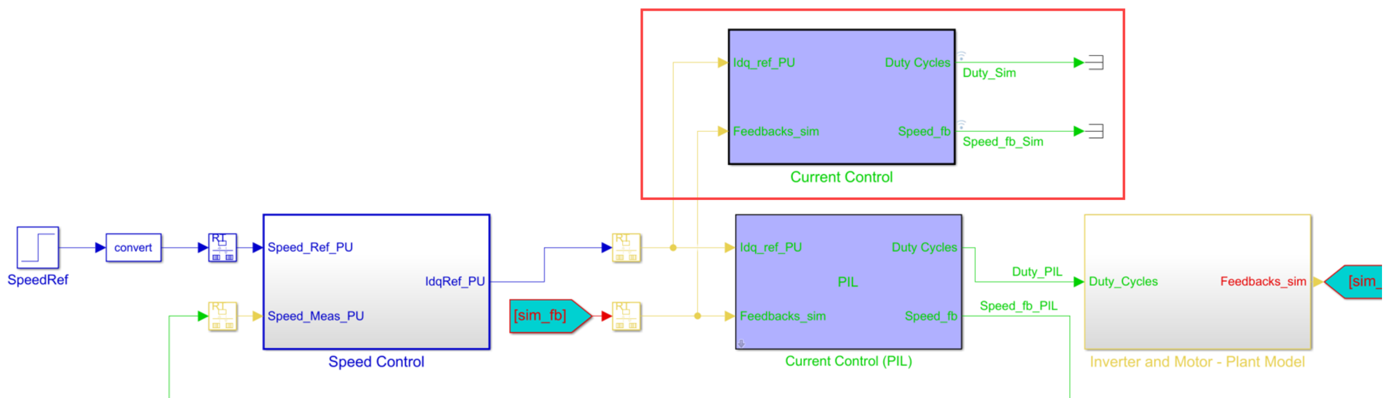


Motor Control Blockset v1.0
Copyright 2020 The MathWorks, Inc.

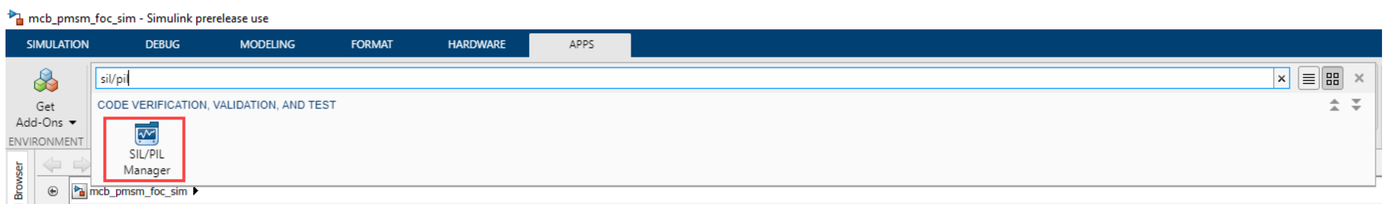
Explore more:
1. Edit motor & inverter parameters
2. Simulate this model

In the PIL mode, the system deploys the Current Control (PIL) subsystem to the target and executes the subsystem in the target hardware.

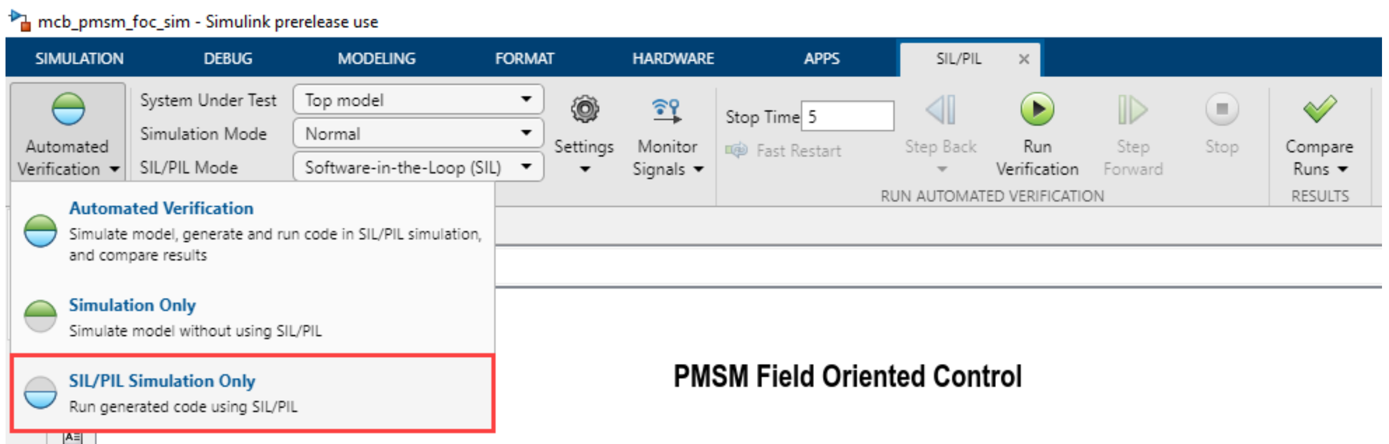
9 To compare the algorithm execution on the host machine simulation and in the PIL simulation, connect the Current Control subsystem parallely to the Current Control (PIL) subsystem. In addition, enable signal logging in the subsystem outputs.



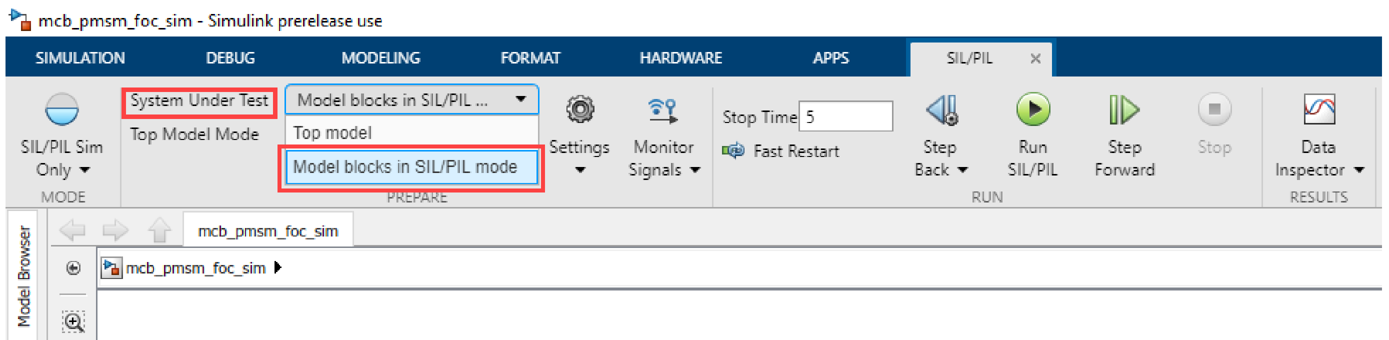
10 On the Simulink toolstrip, select the **SIL/PIL Manager** app from the **Apps** tab.



11 On the **SIL/PIL** toolstrip, select `SIL/PIL Sim Only`.



12 Select `Model blocks in SIL/PIL mode` in the **System Under Test** field.



13 Click **Run SIL/PIL** on the **SIL/PIL** toolstrip to build the Current Control (PIL) subsystem and deploy it to the target.

   After the system deploys the subsystem, the Current Control (PIL) subsystem executes on the target hardware processor, while the plant model runs on the host machine.

Analyze PIL Profiling Results

When PIL simulation ends, the system generates a profiling report.

Note PIL simulation takes more time than the host-machine-based simulation. This is because of the serial communication (related to inputs and outputs of the Current Control (PIL)) between the host machine and subsystem that runs on the target hardware.

Code Execution Profiling Report

Code Execution Profiling Report for mcb_pmsm_foc_sim_v2/Current Control1

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See Code Execution Profiling for more information.

1. Summary

Total time	50681790
Unit of time	ns
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f');
Timer frequency (ticks per second)	2e+08
Profiling data created	16-Jan-2020 18:09:48

2. Profiled Sections of Code

Section	Maximum Execution Time in ns	Average Execution Time in ns	Maximum Self Time in ns	Average Self Time in ns	Calls	
[+] Current_initialize	2260	2260	1365	1365	1	
Current_step [5e-05 0]	5135	5067	5135	5067	10001	
Current_terminate	540	540	540	540	1	

3. CPU Utilization

Task	Average CPU Utilization	Maximum CPU Utilization
Current_step [5e-05 0]	10.13%	10.27%
Overall CPU Utilization	10.13%	10.27%

4. Definitions

CPU Utilization: Percentage of CPU time assigned to a task. Computed by dividing task execution time by sample time.

Execution Time: Time between start and end of code section.

OK Help

The preceding profiling report, which is for fixed-point datatype, shows the maximum and average execution times of the Current Control (PIL) subsystem running on the target hardware.

You can use the **Data Inspector** button on the **Simulation** tab to compare the signals logged during host-machine-based simulation and PIL simulation (executed on the target). This helps you verify the accuracy of host-machine-based simulation and PIL simulation.

This plot compares the speed feedback signals from the Current Control (PIL) and Current Control subsystems.

Note If the execution time exceeds 60% of the budgeted time, we recommend you optimize the algorithm using one of these techniques:

- Execute from RAM.
- Offload some functionalities to CLA or other CPUs.
- Scale the algorithm to run at every alternate cycle.
- Move the less critical functionalities like speed calculation to a slower rate.

For more details on SIL/PIL code verification, see:

- Code Verification and Validation with PIL
- Code Execution Profiling with SIL and PIL
- SIL/PIL Manager Verification Workflow

Deploy and Validate System

- “Prepare Target Hardware” on page 2-2
- “Add Hardware Drivers to Simulation Model and Deploy to Target Hardware” on page 2-4
- “Task Scheduling in Target Hardware” on page 2-6
- “Adding ADC Driver Library Block” on page 2-8
- “Adding Quadrature Encoder Driver Block” on page 2-10
- “Add PWM Driver Block” on page 2-12
- “Add Hardware Interrupt Trigger Block for Current Control Loop” on page 2-15
- “Run in Open-Loop and Switch to Closed-Loop” on page 2-17
- “Model Configuration and Hardware Deployment” on page 2-20
- “Validate System” on page 2-22

Prepare Target Hardware

Follow these steps to prepare the target hardware before you deploy the control algorithm developed using Motor Control Blockset to it.

Verify Direction of Rotation of Motor

The phase sequence of the motor connection in the target hardware determines the direction of rotation of the motor. The Motor Control Blockset example models consider the direction of rotation during the position ramp-up as a positive direction and the corresponding measured speed as positive. It is recommended that you run the motor in open-loop control with a position ramp from `0` to `1` and ensure that the position feedback is positive. The example models in Motor Control Blockset use this convention for the motor's direction of rotation.

For the supported hardware, the algorithm in the example “Quadrature Encoder Offset Calibration for PMSM Motor”, runs the motor and finds the offset between the *d*-axis of the rotor and the encoder index pulse (when the rotor is aligned to the *d*-axis of the stator). The red LED in the host model for this example turns on when the direction is opposite. When this happens, you should change the phase sequence of the motor wiring (swap any two motor wires).

See the example “Hall Offset Calibration for PMSM Motor” to identify the direction of rotation of a motor that uses Hall sensors.

Note When using a Hall sensor, ensure that the Hall sequence updated in the Hall Speed and Position and Hall Validity blocks matches the sequence of the actual Hall signals. If you update an incorrect Hall sequence, the direction read by the target hardware is the opposite of the actual direction.

Calibrate Current Sensor

Signal conditioning circuits for the current sensor introduces an offset voltage in the ADC input when measuring both positive and negative current. For example, an ADC with a voltage reference of `3.3` V can have an offset of `1.65` V (when using Texas Instruments BOOSTXL-DRV8305 hardware). This offset value varies due to tolerances of the passive components in the signal conditioning circuit. It is recommended that you measure the ADC offset of the board during initialization.

In the hardware initialization block used in most Motor Control Blockset example models, the system computes the average current sensor ADC values and uses them as ADC offset values for measuring the current. ADC offset values are represented in ADC counts.

See the example “Run 3-Phase AC Motors in Open-Loop Control and Calibrate ADC Offset” to manually calibrate the ADC offset and update the offset in the script file.

See the Hardware Init subsystem in the example “Field-Oriented Control of PMSM Using Quadrature Encoder” to calculate the ADC offset before starting closed-loop control.

Calibrate Position Sensor

For PMSM, the position used in the current control algorithm should align with the *d*-axis position of the rotor. By default, the quadrature encoder position sensor reads the mechanical position of the

rotor with reference to its index pulse. The position offset is the position read by the quadrature encoder when *d*-axis of the rotor is aligned to phase A. To obtain the accurate motor position, correct the position read by the quadrature encoder by using this offset value. Then provide the corrected motor position value as an input to the current control algorithm.

Any delay between the actual rotor position and the position provided to the current controller affects the motor functionality and performance.

For more details, see the examples “Quadrature Encoder Offset Calibration for PMSM Motor” and “Hall Offset Calibration for PMSM Motor”.

Add Hardware Drivers to Simulation Model and Deploy to Target Hardware

This topic explains the steps to add hardware drivers to a simulation model and deploy the model to the target hardware.

This topic uses the model `mcb_pmsm_foc_sim` as a reference to explain the steps involved in the hardware deployment. The model `mcb_pmsm_foc_sim` simulates the field-oriented control (FOC) algorithm for implementing speed control for a PMSM.

The speed control algorithm is deployed to the target hardware Texas Instruments LAUNCHXL-F28379D (connected to Texas Instruments BOOSTXL-DRV8305). These are the hardware interface details:

Interface	Pin on LAUNCHXL-F28379D
Phase-A input of the motor	ADCINC2
Phase-B input of the motor	ADCINB2
PWM A output from the motor	EPWM1A
PWM B output from the motor	EPWM2A
PWM C output from the motor	EPWM3A
Enable Driver BOOSTXL-DRV8305	GPIO124

These steps explain how to add the hardware driver blocks from the Embedded Coder Support Package for Texas Instruments C2000 Processors so as to deploy the control algorithm to the target hardware LAUNCHXL-F28379D (connected to BOOSTXL-DRV8305).

1. "Task Scheduling in Target Hardware" on page 2-6
2. "Adding ADC Driver Library Block" on page 2-8
3. "Adding Quadrature Encoder Driver Block" on page 2-10
4. "Add PWM Driver Block" on page 2-12
5. "Add Hardware Interrupt Trigger Block for Current Control Loop" on page 2-15
6. "Run in Open-Loop and Switch to Closed-Loop" on page 2-17
7. "Model Configuration and Hardware Deployment" on page 2-20

These steps use variables to define the datatypes, the execution time of the current controller, and the execution time of the speed controller. See the model initialization script of the example model `mcb_pmsm_foc_sim` for details on the variables defined in these steps.

To understand the prerequisites needed to deploy the control algorithm to any target hardware, see "Prepare Target Hardware" on page 2-2. For details related to the hardware connections, see "Hardware Connections".

To implement a simulation model with a PMSM motor FOC control algorithm, see "Design Field-Oriented Control Algorithm" on page 1-2.

A basic understanding of Simulink is a prerequisite to follow these steps. For details about the ADC driver, the quadrature encoder driver, and the hardware interrupt block, see the example model `mcb_pmsm_foc_qep_f28379d`, which uses an architecture similar to what we describe.

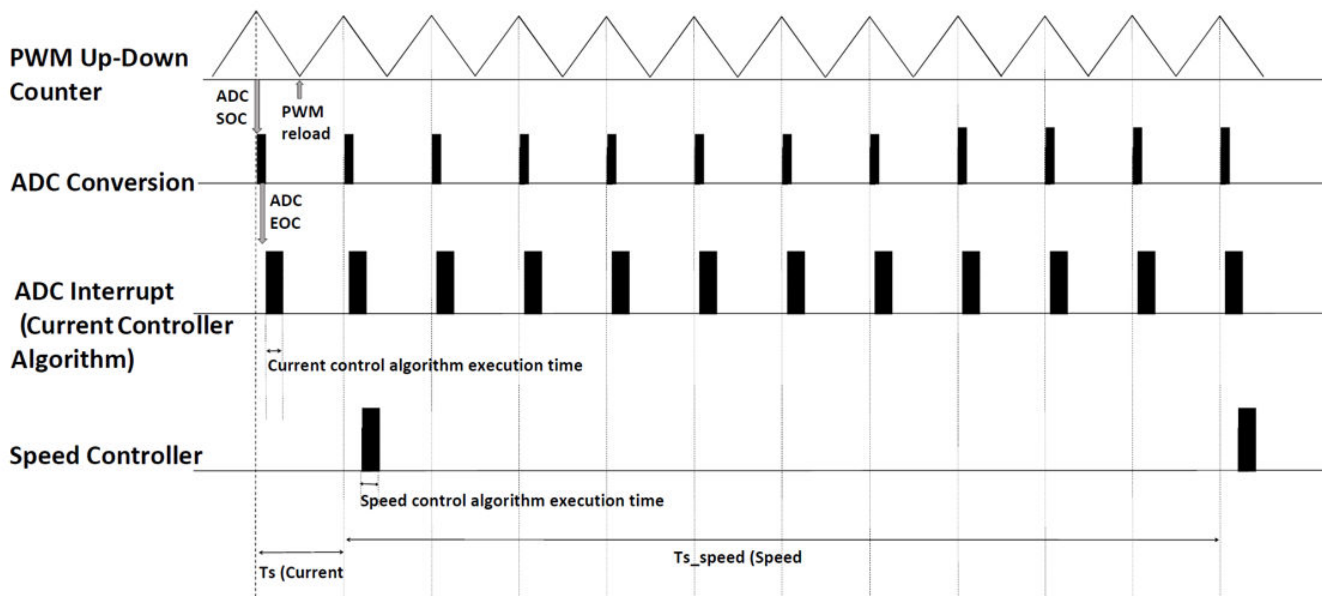
Note For target hardware other than LAUNCHXL-F28379D (connected to BOOSTXL-DRV8305), you can follow these steps, but select the driver blocks (ADC, PWM, Interrupt) from the appropriate supported hardware library.

Task Scheduling in Target Hardware

In the example model `mcb_pmsm_foc_sim`, configuring the current controller and the speed controller are the two important tasks. The current controller is scheduled to run after every T_s (50 µsec for a 20 kHz switching frequency) and the speed controller runs after every T_{s_speed} ($10*T_s$). The current controller reads the motor phase currents and position and computes the PWM duty cycle to run the motor. The speed controller runs the control loop, calculates I_q reference for the current controller, and controls the motor speed in the closed-loop.

In the target hardware, the current controller is synchronized with the ADC interrupt (for every T_s) and the speed controller is triggered after every T_{s_speed} ($10*T_s$).

This figure shows the event sequence, interrupt trigger, and software execution time for the control algorithm running in the target hardware.



In this figure, the execution times for the current controller and speed controller are not in scale. See the processor datasheet to better understand the functionality of the processor peripherals such as the ADC (analog-to-digital converter) and the PWM (pulse-width modulation).

The event sequence followed in the model is:

1. The processor peripheral PWM, which is center-aligned (Up-Down Counter), triggers the start-of-conversion (SOC) event for the ADC module when the PWM counter value equals the PWM period.
2. The ADC module converts the sampled analog signal into digital counts and triggers the end-of-conversion (EOC) event.
3. The EOC triggers the ADC interrupt.
4. The current controller is scheduled to execute with the ADC interrupt.
5. The speed controller is scheduled to run after every T_{s_speed}.

You can also use SoC Blockset™ for task scheduling, profiling, and addressing challenges related to ADC-PWM synchronization, controller response, and studying different PWM settings. For details, see “Integrate MCU Scheduling and Peripherals in Motor Control Application” (SoC Blockset).

Adding ADC Driver Library Block

In the example model `mcb_pmsm_foc_sim`, the current controller block receives the motor phase current in ADC counts from the plant model that converts the motor phase current from Amperes to ADC counts. In the target hardware, the current controller reads the motor phase current from the ADC block. Follow this workflow to add the ADC driver block.

In the Texas Instruments BOOSTXL-DRV8305 inverter hardware, the phase-A current of the motor is read from ADC C2 channel and phase-B current is read from ADC B2 channel. Select **ADC module** C and channel 2 to obtain the phase-A current of the motor. Select **ADC module** B and channel 2 to obtain the phase-B current of the motor. For other target hardware, select the **ADC module** and channel where the motor phase currents are interfaced.

Select `ePWM1_ADCSOCA` as the trigger source in the ADC block because the PWM library block triggers the start-of-conversion event SOC0 when the PWM counter equals the PWM period register.

Select `ADCINT1` in the ADC B module. This triggers an ADC interrupt at the end-of-conversion (EOC) event. When the ADC interrupt occurs, the FOC current control algorithm executes.

In the Simulink library browser, select and add the ADC block from the F2837xD library in Embedded Coder Support Package for Texas Instruments C2000 Processors. Configure the ADC block to read the phase-A and phase-B currents of the motor.

In the ADC block parameters dialog box, configure the ADC C module and Channel 2 to read the phase-A current of the motor, as shown in this table.

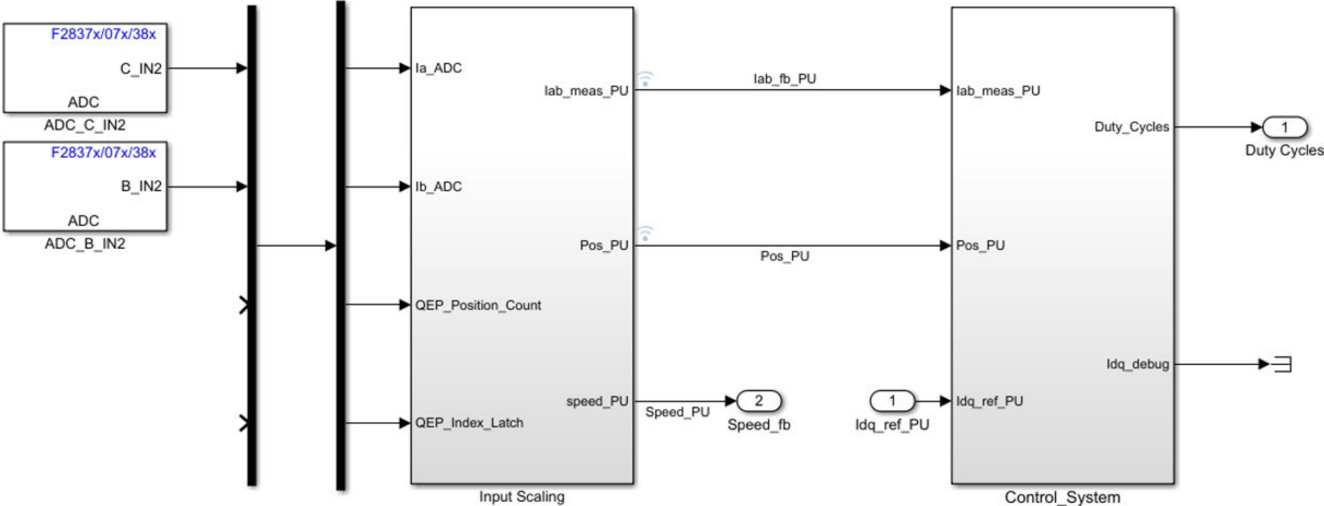
Tab and Parameter in ADC Block	Settings
SOC Trigger > ADC Module	**C**
SOC Trigger > SOC trigger number	**SOC0**
SOC Trigger > SOC trigger source	**ePWM1_ADCSOCA**
Input Channels > Conversion channel	**ADCIN2**

Rename the block as ADC_C_IN2.

In ADC block parameters dialog box, configure the ADC B module and channel 2 (to read Motor Phase-B current) and the ADC interrupt (ADCINT1), as shown in this table.

Tab and Parameter in ADC Block	Settings
SOC Trigger > ADC Module	**B**
SOC Trigger > SOC trigger number	**SOC0**
SOC Trigger > SOC trigger source	**ePWM1_ADCSOCA**
SOC Trigger > Post interrupt at AOC trigger	`on`
SOC Trigger > Interrupt selection	**ADCINT1**
SOC Trigger > ADCINT1 continuous mode	`on`
Input Channels > Conversion channel	**ADCIN2**

Rename the block as ADC_B_IN2.


F2837x/07x/38x
C_IN2
ADC
ADC_C_IN2
F2837x/07x/38x
B_IN2
ADC
ADC_B_IN2
Ia_ADC
Ib_ADC
QEP_Position_Count
QEP_Index_Latch
Iab_meas_PU
Pos_PU
speed_PU
Input Scaling
Iab_fb_PU
Pos_PU
Speed_PU
2
Speed_fb
1
Idq_ref_PU
Iab_meas_PU
Pos_PU
Idq_ref_PU
Duty_Cycles
Idq_debug
Control_System
1
Duty Cycles

Adding Quadrature Encoder Driver Block

In the Simulink Library Browser, add the eQEP block from Embedded Coder Support Package for Texas Instruments C2000 Processors > F2837xD.

The eQEP block reads the quadrature encoder pulses and increments the position count. This block outputs the quadrature encoder pulse for the mechanical rotor position wraparound when the quadrature encoder index pulse is read.

See the section Quadrature Encoder Interface Configuration in “Model Configuration Parameters” for configurations related to the quadrature encoder.

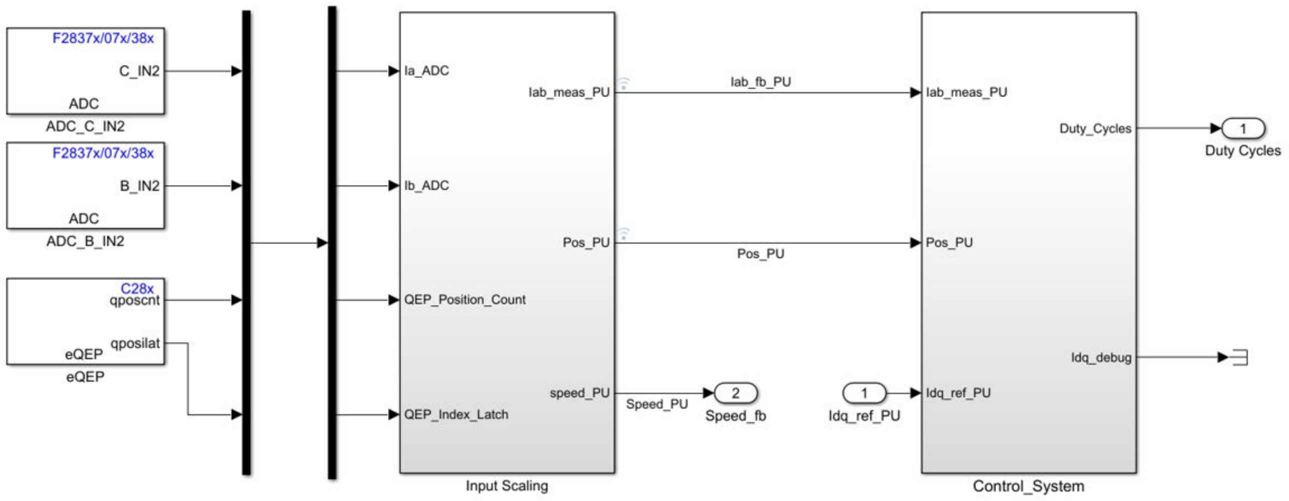
In c28x eQEP block parameters dialog box, configure the quadrature encoder to read the quadrature encoder pulse count in the TI processor and wrap the pulse counter output when index pulse is found as shown in this table.

Tab and Parameter in eQEP Block	Settings
General > Module	**eQEP1**
General > Sample time	**-1**
Position counter > Output position counter	`on`
Position counter > Maximum position counter value (0~4294967295)	**2^16-1**
Position counter > Position counter reset mode	**Reset on the first index event**
Position counter > Output latch position counter on index event	`on`
Position counter > Index event latch of position counter	**Falling edge**

Rename the block as eQEP.

eQEP1 module is selected because the quadrature encoder is connected to the QEP_A interface in the LaunchPadXL28379d hardware board. Select the sample time as `-1` because the library block is function-call triggered by the ADC interrupt synchronously. The maximum position counter value is `2^16-1` because the position counter uses a 16-bit architecture in the library driver block. The position counter reset mode on the index pulse wraps the position count on the index pulse.

Add the eQEP driver block module to the `mcb_pmsm_foc_sim/Current control` subsystem as shown in this figure.



F2837x/07x/38x
C_IN2
ADC
ADC_C_IN2
F2837x/07x/38x
B_IN2
ADC
ADC_B_IN2
C28x
qposcnt
qposilat
eQEP
eQEP
Ia_ADC
Ib_ADC
QEP_Position_Count
QEP_Index_Latch
Iab_meas_PU
Pos_PU
speed_PU
Input Scaling
Iab_fb_PU
Pos_PU
Speed_PU
2
Speed_fb
1
Idq_ref_PU
Iab_meas_PU
Pos_PU
Idq_ref_PU
Duty_Cycles
Idq_debug
Control_System
1
Duty Cycles

Add PWM Driver Block

In the Simulink Library Browser, add the ePWM block from Embedded Coder Support Package for Texas Instruments C2000 Processors > F2837xD.

Configure the ePWM1, ePWM2, and ePWM3 blocks for generating the PWM pulse. In the ePWM block parameters dialog box, calculate the pulse width modulation (PWM) counter period register value from the CPU frequency and the PWM frequency. For center-aligned PWM, divide the value by 2.

PWM counter period = CPU clock frequency / PWM frequency / 2

See the TMS320f28379d processor ePWM peripheral for more details.

In the F2837x/07x/004x/38x ePWM block parameters dialog box, update the settings to configure PWM1 to generate PWM pulses in the target hardware as shown in this table.

Tab and Parameter in ePWM Block	Settings
General > **Module**	**ePWM1**
General > **Timer Period**	Enter the PWM period value in the CPU clock cycle • PWM counter period = CPU clock frequency / PWM frequency / 2 • For LaunchPad 28379D, clock frequency is 200 MHz. For PWM frequency of 20 kHz, PWM counter period = `200e6 / 20e3 / 2`; PWM counter period = `5000`
Counter Compare > **Specify CMPA via**	**Input port**
Counter Compare > **CMPA initial value**	Enter the PWM counter period/ 2 (2500)
Counter Compare > **Specify CMPB via**	**Input port**
Counter Compare > **CMPB initial value**	Enter the PWM counter period/ 2 (2500)
Deadband unit > **Use deadband for ePWM1A**	`on`
Deadband unit > **Use deadband for ePWM1B**	`on`
Deadband unit > **Deadband polarity**	**Active high complementary (AHC)**
Deadband unit > **Deadband Rising edge (RED) period (0~16383)**	15
Deadband unit > **Deadband Falling edge (FED) period (0~16383)**	15
Event Trigger > **Enable ADC start of conversion for module A** check box (only for PWM1)	`on`
Event Trigger > **Start of conversion for module A event selection** (only for PWM1)	**Counter equals to period (CTR=PRD)**

Rename the block as ePWM1.

In the F2837x/07x/004x/38x ePWM block paramters dialog box, update the settings to configure PWM2 and PWM3 to generate PWM pulses in the target hardware. PWM2 and PWM3 are synchronized with PWM1. Follow ePWM1 configurations (other than Event Trigger) and add the configurations as shown in this table.

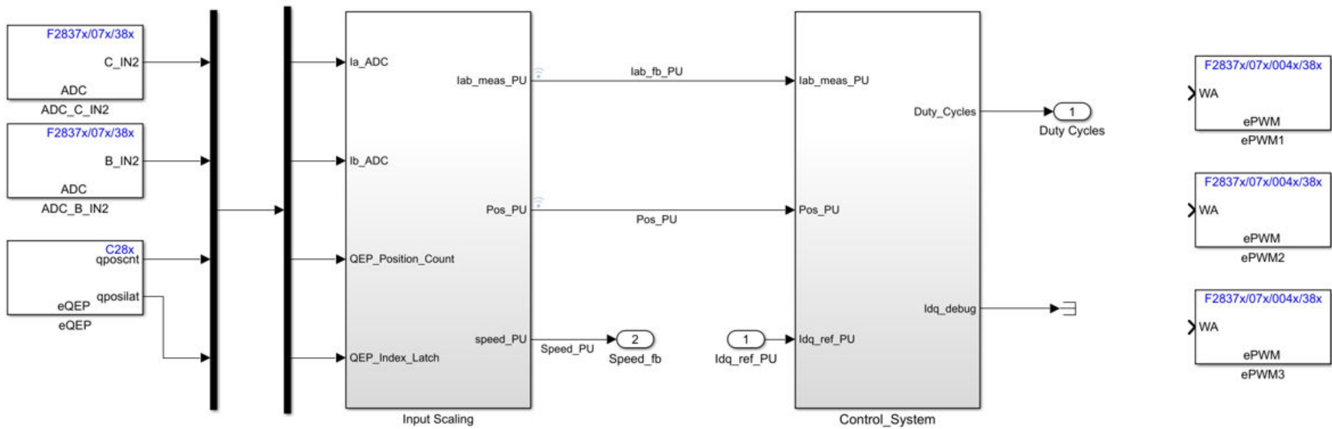
Tab and Parameter in ePWM Block	Settings
General > Module	**ePWM2**
General > Timer Period	Enter the PWM period value in the CPU clock cycle • PWM counter period = CPU clock frequency / PWM frequency / 2 • For LaunchPad 28379D, clock frequency is 200 MHz. For PWM frequency of 20 kHz, PWM counter period = `200e6 / 20e3 / 2`; PWM counter period = `5000`
General > Synchronization action	**Set counter to phase value specified via dialog**
General > Counting direction after phase synchronization	**Count up after sync**
General > Phase offset value (TBPHS)	`0`
Counter Compare > Specify CMPA via	**Input port**
Counter Compare > CMPA initial value	Enter the PWM counter period/ 2 (`2500`)
Counter Compare > Specify CMPB via	**Input port**
Counter Compare > CMPB initial value	Enter the PWM counter period/ 2 (`2500`)
Deadband unit > Use deadband for ePWM1A	`on`
Deadband unit > Use deadband for ePWM1B	`on`
Deadband unit > Deadband polarity	**Active high complementary (AHC)**
Deadband unit > Deadband Rising edge (RED) period (0~16383)	`15`
Deadband unit > Deadband Falling edge (FED) period (0~16383)	`15`

Rename the blocks as ePWM2 and ePWM3.

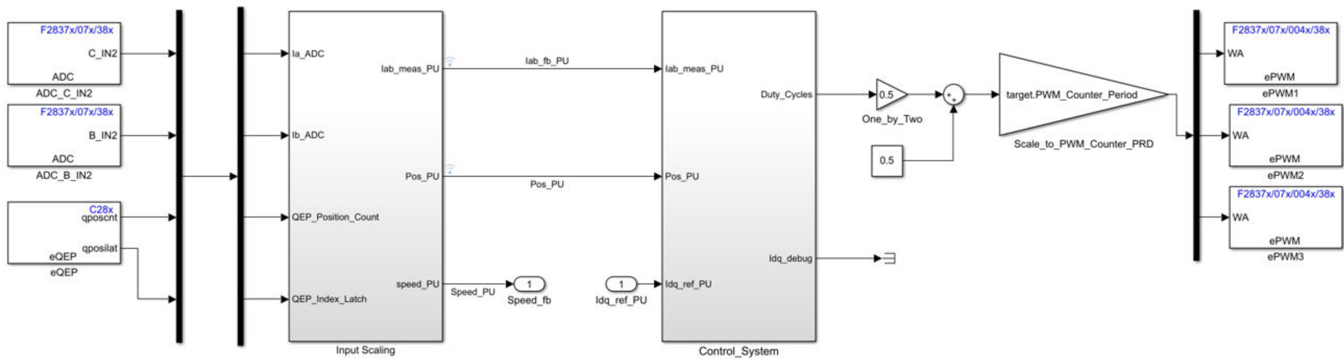
Select CMPA and CMPB as input ports where PWM duty is given as input. The range varies from `0` to *PWM_counter_period*. PWM outputs when PWM up-counter matches CMPA and PWM down-counter matches CMPB. By default, the system inputs a duty cycle of 50% by selecting PWM counter period / 2.

Enable dead time in the ePWM configuration. On the Event Trigger tab of PWM1 module, select ADC start of conversion event when the PWM counter equals the PWM period. In the SOC Trigger tab of the ADC block, select **SOCx trigger source** as `ePWM1_ADCSOCA`.

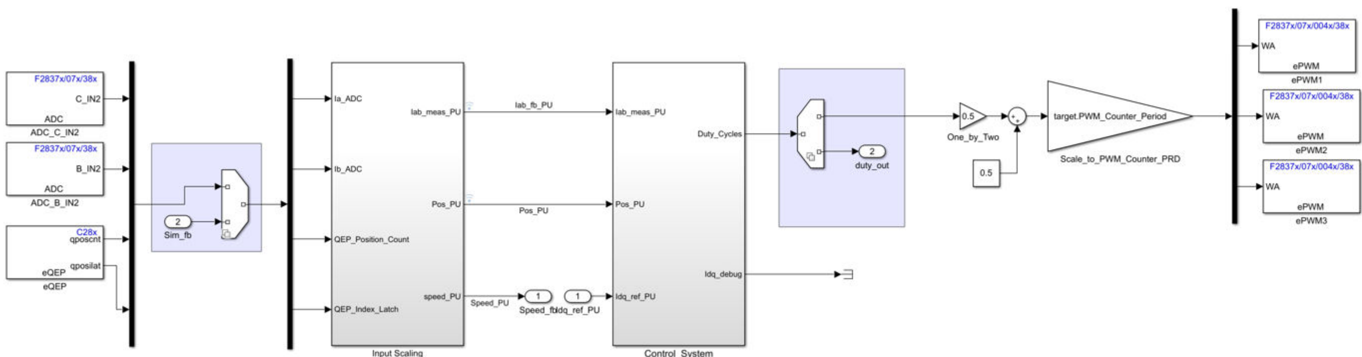
Synchronize the ePWM2 and ePWM3 blocks with the ePWM1 block by setting the synchronization timing to the moment when the PWM counter equals to zero in the ePWM2 and ePWM3 blocks.



The ePWM blocks expect the duty cycle value to range from `0` to period counter register (`5000`). Control_System subsystem outputs the PWM in the range `-1` to `1`. This model scales the output to `0` to `5000` (period counter value).



For simulation, add a variant source/sink to the hardware driver block for simulation and code generation.



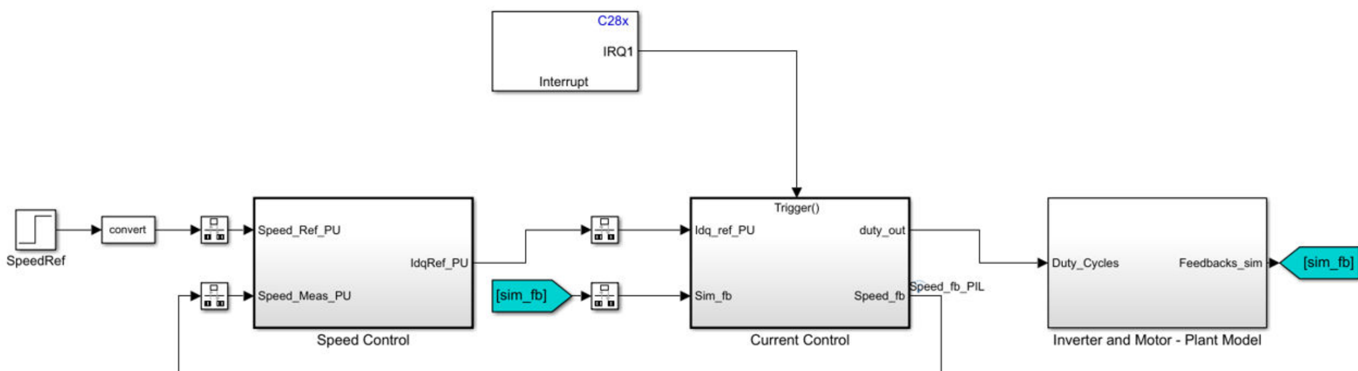
Add Hardware Interrupt Trigger Block for Current Control Loop

In the Simulink Library Browser, select and add the C28x Hardware Interrupt block from Embedded Coder Support Package for Texas Instruments C2000 Processors > Scheduling.

In the block parameters dialog box, update the settings to configure the hardware interrupt ADCINT1. In the block parameters dialog box, identify and update the PIE and CPU interrupts for the hardware interrupt ADCINT1.

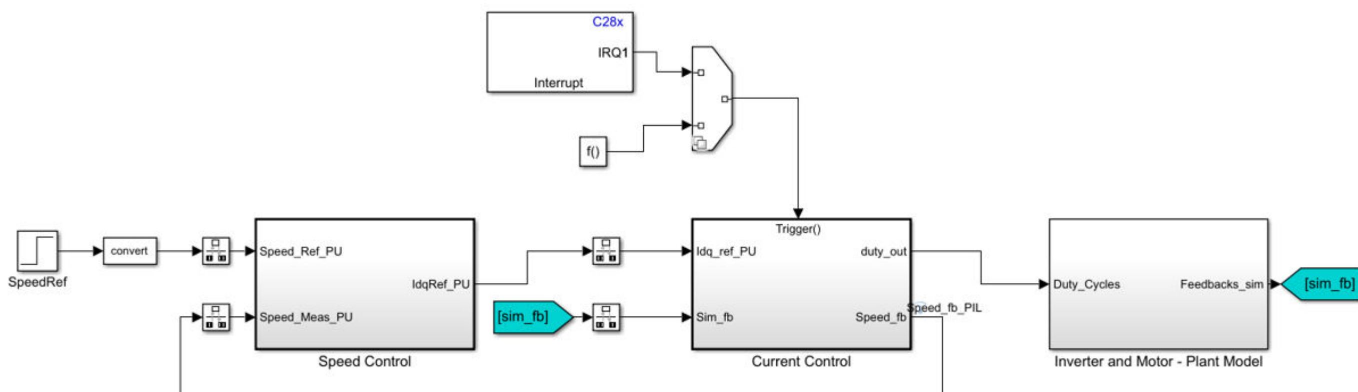
Parameter in C28x Hardware Interrupt Block	Settings
CPU interrupt numbers	**[1]**
PIE interrupt numbers	**[2]**

In the current control subsystem, add a Trigger block and set the **Trigger type** parameter to `function-call`. Connect this subsystem trigger input to the Hardware Interrupt block as shown in this figure.



In the Rate Transition block input to Current Control subsystem, change the **Output port sample time** to `-1`.

Add a Function-Call Generator block in variant source to support the model simulation. In the Function-Call Generator block, set the **Sample time** parameter as T_s (`50e-6`).



Simulate the model with the updated driver blocks and check the simulation results in the simulation Data Inspector. Variants ensure that ADC, PWM drivers, and interrupts are not active for simulation.

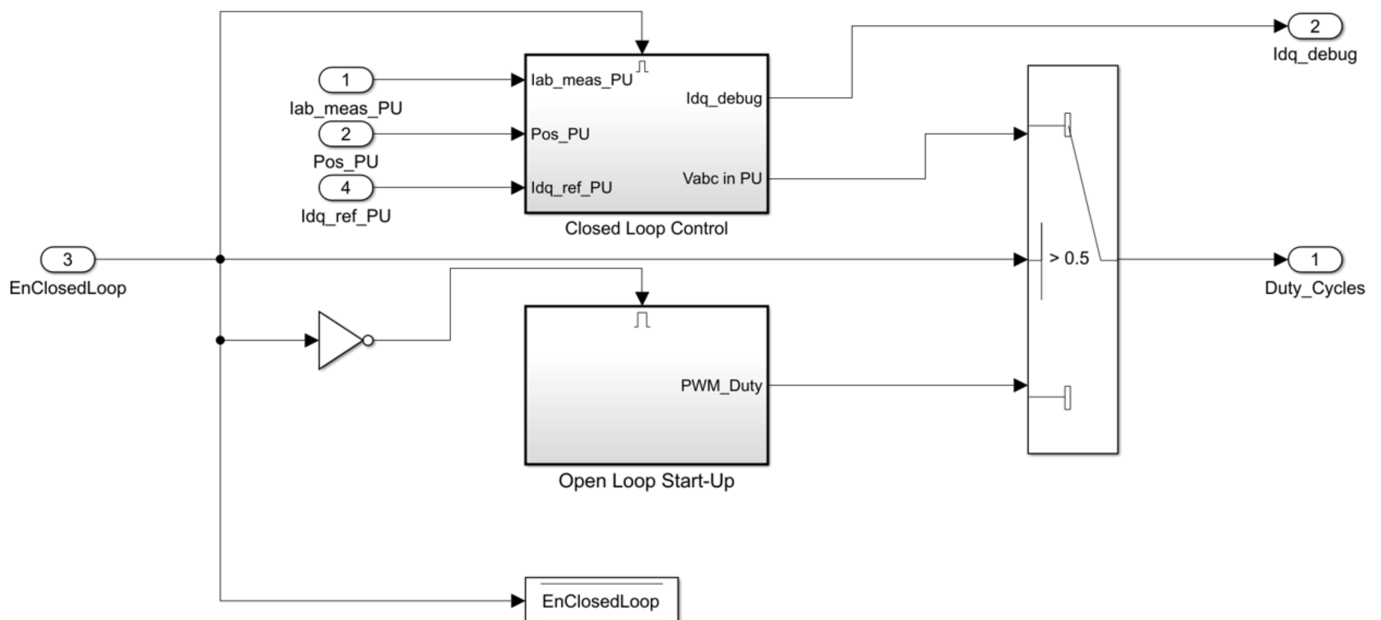
Run in Open-Loop and Switch to Closed-Loop

A permanent magnet synchronous motor (PMSM) with a quadrature encoder sensor requires an initial position to start the motor. As you do not have a method to determine the initial position at the beginning (before starting the motor), run the motor in open-loop and ensure that the quadrature encoder index pulse is read at least once. At the quadrature encoder index pulse, the quadrature encoder sensor resets its position to align with the mechanical angle of the motor. The motor switches from an open-loop run to closed-loop speed control to maintain the reference speed. This step is only applicable for a quadrature encoder sensor and not for a Hall position sensor. A Hall sensor outputs the initial position of the rotor segment from Hall signal port inputs.

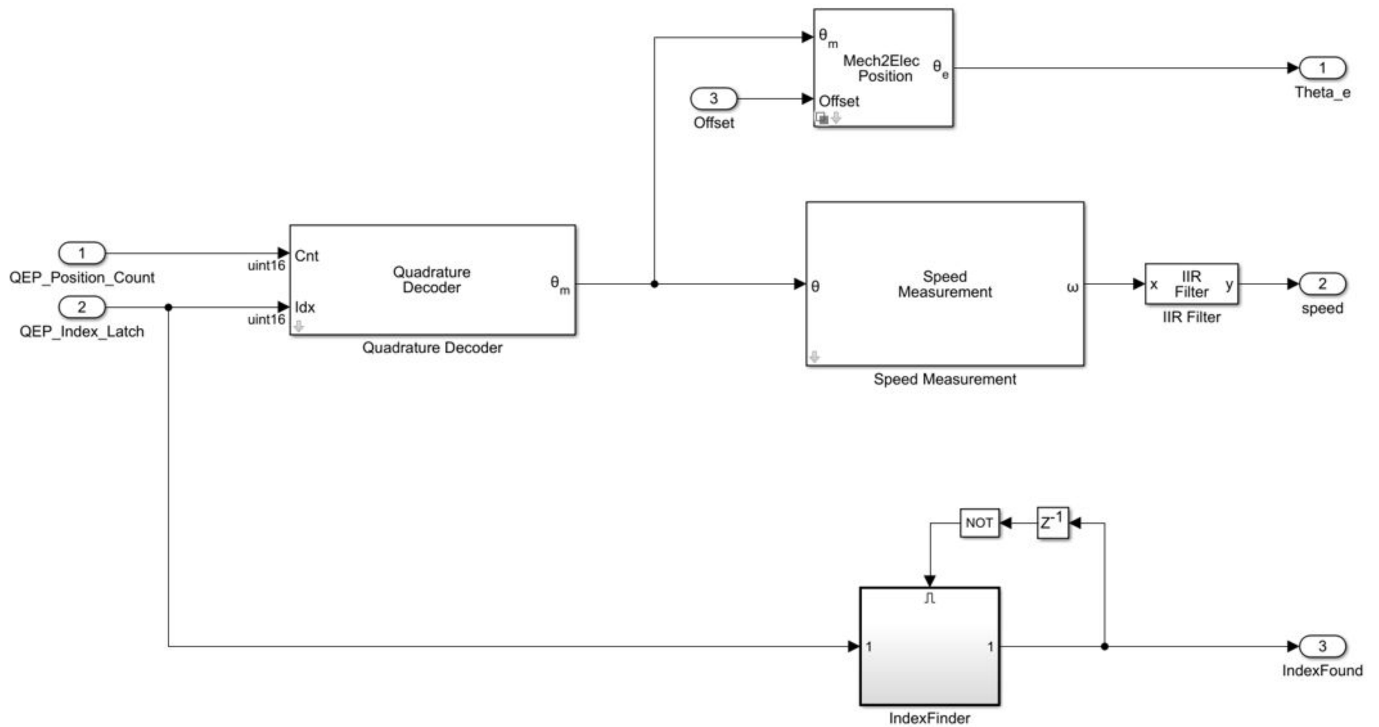
Follow these steps to implement an open-loop motor run with a transition to closed-loop control:

1. Copy the `mcb_pmsm_foc_qep_f28379d/Current Control/Control_system` subsystem to your model. This adds the logic to run the motor in open-loop. This subsystem switches the control from open-loop to closed-loop if *EnClosedLoop* input is `1`. Add an input port **EnClosedLoop**. This adds the Data Store Read blocks for *Enable* and *SpeedRef*. Add the Data Store Memory blocks *Enable*, *EnClosedLoop*, and *SpeedRef* in the topmost level of the model.

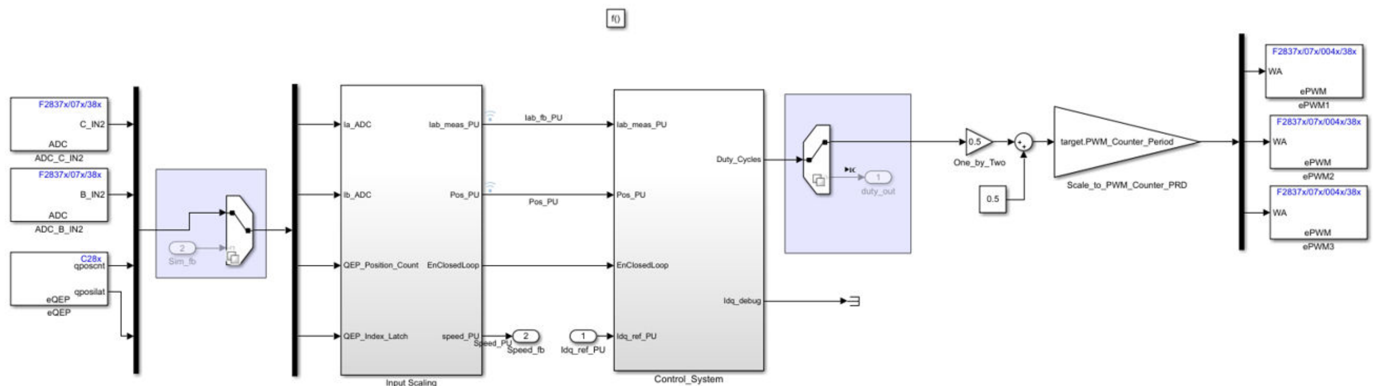
   When the open-loop run begins, the sign of *SpeedRef* decides the direction of the initial motor run. If *SpeedRef* is negative, the motor spins in the opposite direction during the open-loop run.



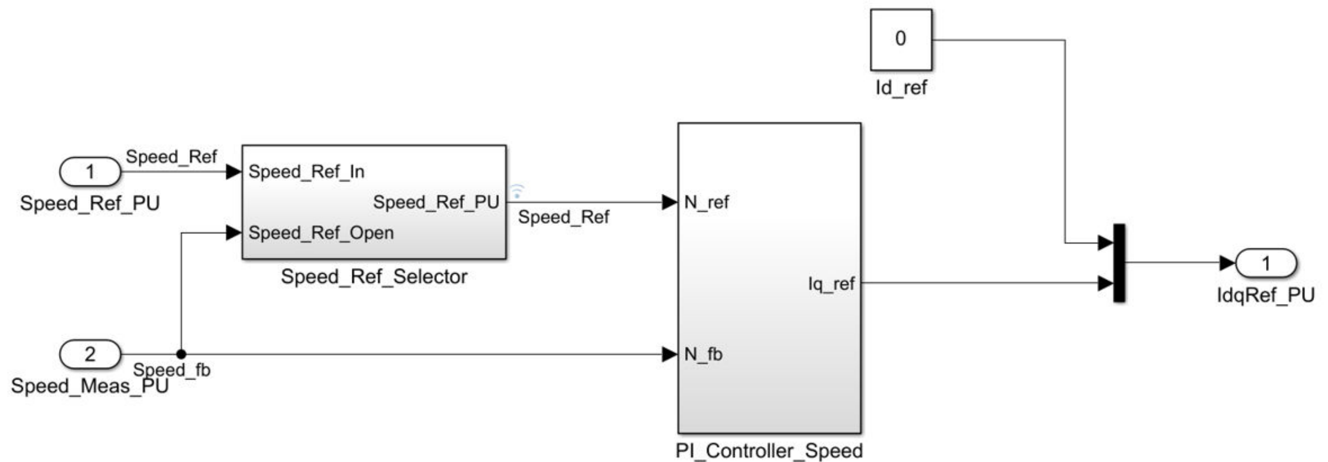
2. Copy the `mcb_pmsm_foc_qep_f28379d/Current Control/Input Scaling/Calculate Position and Speed` subsystem to your model. This adds the `IndexFinder` block. When quadrature encoder index pulse is detected for the first time, `IndexFound` port is set to `1`. Add an output port `IndexFound` and rename it to `EnClosedLoop`.



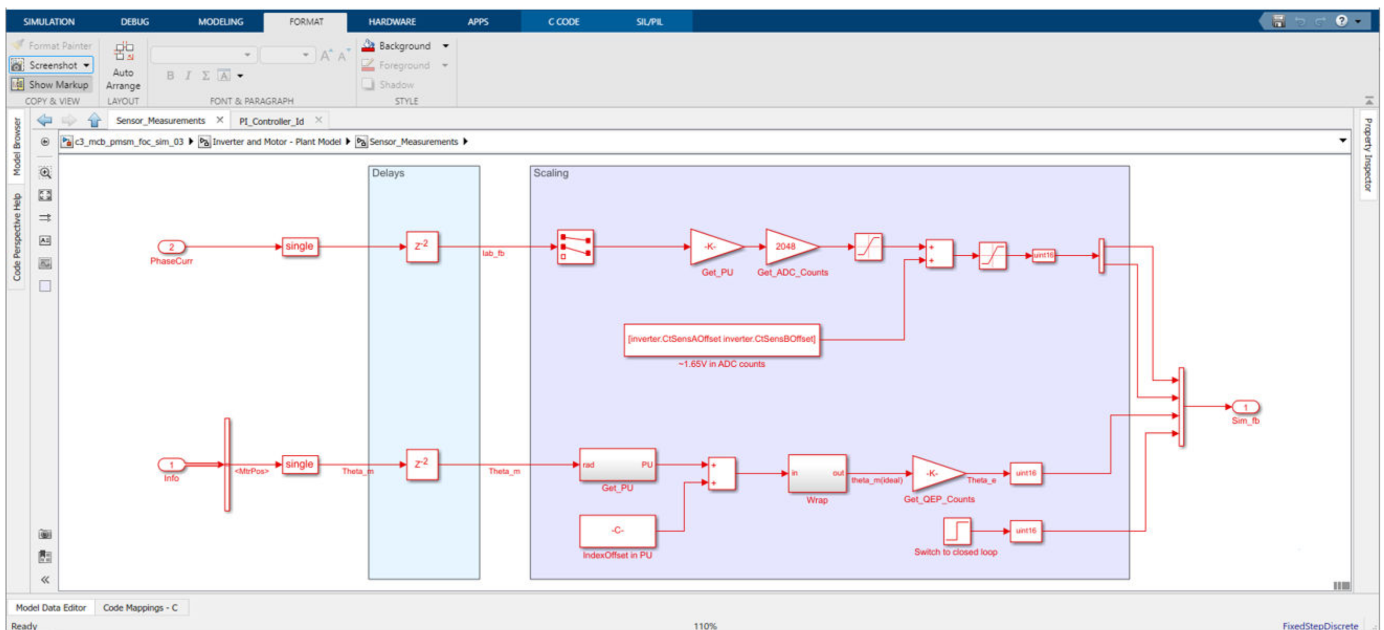
3 Connect the output port EnClosedLoop from the Input Scaling subsystem to the input port EnClosedLoop in the Control_System subsystem as shown in this figure.



4 Copy the `mcb_pmsm_foc_qep_f28379d/Speed Control/Speed_Ref_Selector` subsystem to your model. This block uses the `speed_ref` block when closed-loop control begins. For a smooth transition from open-loop to closed-loop, the speed measured is used as the speed reference during the open-loop run. Add a Data Store Write block *SpeedRef* to the `PI_Controller_Speed` input port.



5 In the plant model, a step input is added to simulate the IndexFinder block for simulation. Rename the step input to `Switch to closed loop`. See the `mcb_pmsm_foc_qep_f28379d/ Inverter and Motor - Plant Model/Sensor_Measurments` subsystem for the step input for switching to closed-loop. Select the step time of `0.1` and sample time of T_{s_motor}.



6 Create Data Store Memory blocks for *EnClosedLoop*, *Enable*, and *SpeedRef*. *Enable* block is used to reset the PI integrator before running the motor. Add these default values in the Data Store Memory blocks: *Enable* = `1`, *EnClosedLoop* = `0`, and *SpeedRef* = `0.25`.

The Data Store Memory blocks are used to share data across the subsystem.

7 Run the simulation and observe the speed reference and the speed feedback.

Model Configuration and Hardware Deployment

Use these steps to select the target hardware in the model configuration.

1. In the Simulink model, click **Hardware** > **Hardware Settings** to open the Configuration Parameters dialog box.
2. Open the **Hardware Implementation** tab and set **Hardware board** to **TI Delfino F28379D LaunchPad**.

For any other custom board, navigate to the **Hardware Implementation** tab of the Configuration Parameters dialog box and select the appropriate processor and edit the peripheral details in **Hardware board settings** > **Target hardware resources**.

For the solver and quadrature encoder interface configuration details, see “Model Configuration Parameters”.

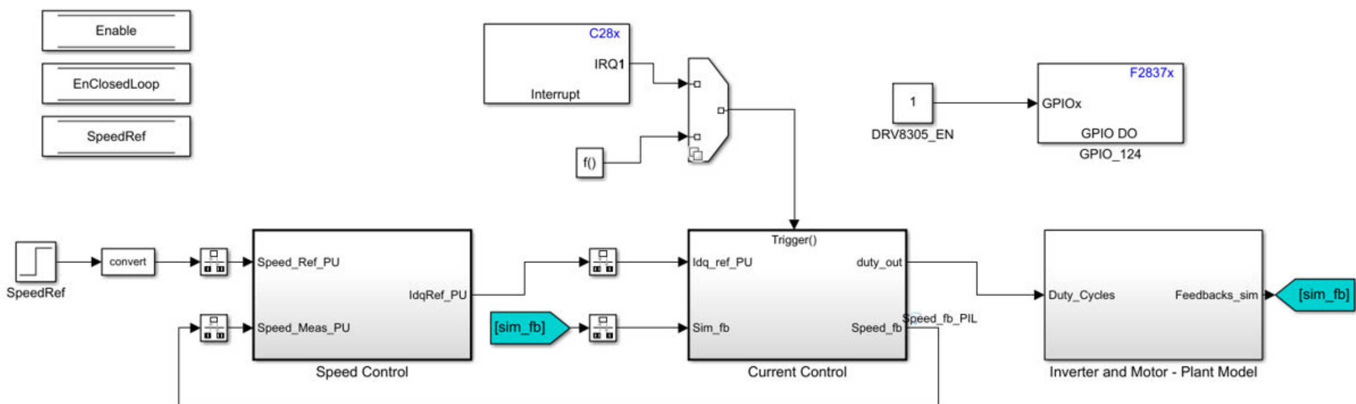
Connect the Texas Instruments BOOSTXL-DRV8305 board and QEP connector to the Texas Instruments LaunchPad XL hardware board. For hardware connection details related to Texas Instruments C2000 LaunchPadXL, see “Hardware Connections”. The BOOSTXL-DRV8305 (attached to the LaunchPadXL board) requires an enable signal. This signal is connected to the GPIO124 pin of the processor.

In the Simulink Library Browser, add Embedded Coder Support Package for Texas Instruments C2000 Processors > F2837xD > Digital Output. In the block parameters dialog box of the Digital Output block, change these settings:

Parameter in Digital Output Block	Settings
GPIO Group	**GPIO120~GPIO127**
GPIO124	on

Rename the block as GPIO_124.

Add a constant block with the value 1 as an input to the GPIO124 block as shown in this figure.



On the **Hardware** tab of the Simulink model, select **Build, Deploy & Start**. This generates the C code, CCS project, and a target-specific out-file. The system uses serial communication to download this target specific out-file to the target hardware and runs the downloaded algorithm in the hardware.

When the model is deployed to the target, the motor runs in open-loop and then runs in closed-loop speed control. This example recommends that you use serial communication to monitor and debug the signals. See the example model `mcb_pmsm_foc_qep_f28379d` for details on implementing serial receive and transmit communications (between the host and target models). From the Serial Receive block, update the Data Store Memory block to start and stop the motor.

Validate System

In this section...

"Calculate Physical Motor Load in Target Hardware" on page 2-23
"Compare Speed Controller Response in Simulation and in Target Hardware" on page 2-24
"Compare Current Controller Response in Simulation and in Target Hardware" on page 2-26

This section explains how to evaluate the accuracy of the plant (motor and inverter) model of the physical motor and load connected to the motor. You should validate the plant model and verify that the results are close to the physical system measurements before using the plant model for implementing advanced algorithms. You can validate the system by comparing the step response of speed control and current control in simulation and in target hardware connected to the motor.

Use the example "Tune Control Parameter Gains in Hardware and Validate Plant" to measure the step response of the current and speed controllers. The host model in this example communicates the current reference to the target hardware and measures the step response of the current controller.

- You can use any speed control example from Motor Control Blockset to validate the system.
- Validate speed control by comparing the step response in simulation and the hardware test values.
- Validate the *d*-axis current control by electrically or mechanically locking the rotor and comparing the step response in simulation and the hardware test results.

  You can use another method to validate the *d*-axis current control. Run the motor at a constant speed and provide a step change in the reference *d*-axis current. This requires two modifications in the speed control subsystem of the target model. Set a constant speed reference input. Command I_d reference from the host model. Compare the step response of the *d*-axis current in simulation and in the hardware tests.
- Validate the *q*-axis current control by mechanically coupling the motor with an external dynamometer running in speed control. This requires two modifications in the speed control subsystem of the target model. Discard the I_d and I_q reference from the speed PI controller output. Command I_d reference from the host model. Compare the step response of the *q*-axis current in simulation and in the hardware tests.

Warning When capturing the step response in *d*-axis current control, always use a positive step. Negative values of I_d can damage the permanent magnet in the PMSM.

Host model for Control Parameter Gain Tuning (Manual) in Hardware and Plant Validation

Control

Select Motor operating mode
- Stop
- Open loop run
- Torque control
- Speed control

Operating Mode Variables

Motor torque control mode
0.1 Id Reference
0 Iq Reference
Unlock Pos lock

Motor speed control mode
0.2 Speed Reference

Control loop gains

d-axis current controller
2.2383 Kp Gain
7423.0020000000004 Ki Gain

q-axis current controller
2.2383 Kp Gain_
7423.0020000000004 Ki Gain_

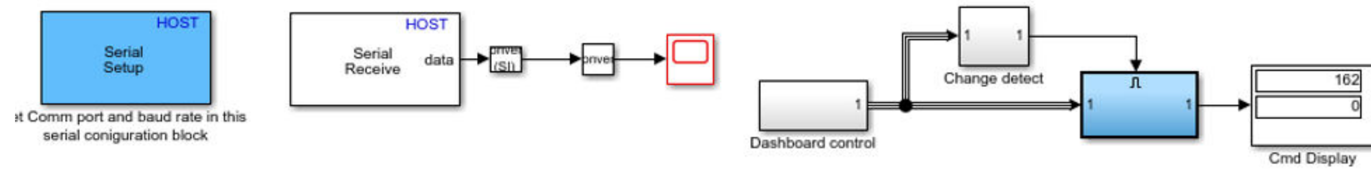
Speed controller
1.2102 Kp Gain__
32.0152 Ki Gain__

Monitor

Monitor Signal #1: V_alpha, V_beta, I_alpha, I_beta, Va_out, Vb_out, Vc_out, Ia_meas, Ib_meas, Id_ref, Id_meas, Vd_ctrl_out, Iq_ref, Iq_meas, Vq_ctrl_out, Position_meas, Speed_ref, Speed_meas

Monitor Signal #2: V_alpha, V_beta, I_alpha, I_beta, Va_out, Vb_out, Vc_out, Ia_meas, Ib_meas, Id_ref, Id_meas, Vd_ctrl_out, Iq_ref, Iq_meas, Vq_ctrl_out, Position_meas, Speed_ref, Speed_meas

1. Change COMM port in serial blocks
2. Caution: Stop the motor when swithcing between the modes



Motor Control Blockset v1.0
Copyright 2020 The MathWorks, Inc.

See the example “Tune Control Parameter Gains in Hardware and Validate Plant” to deploy the model to the hardware. Perform motor parameter estimation because accuracy in the plant model is important so as to match the simulation results with the hardware measurements.

Calculate Physical Motor Load in Target Hardware

Before comparing the controller responses in simulation and in the target hardware, the load torque in plant simulation must match the motor load in the physical system. Follow these steps to calculate the load torque in the physical system and update the calculated load torque in the plant model.

1. Run the host model and connect the target hardware through serial communication.
2. In Select Motor operating mode, select **Speed control**.

   The motor spins in speed control.
3. Select **Id_meas** in **Monitor Signal #1** and **Iq_meas** in **Monitor signal #2**. Read the `Id_meas` and `Iq_meas` values from the scope.
4. Convert the per-unit (PU) current to Amperes by multiplying it with `PU_System.I_base`.
5. Calculate the load torque in Nm using this equation:

   $$T_{\mathrm{load}} = 1.5 \times pole_pair \times [(flux_pm \cdot I_{\mathrm{q}}) + (L_{\mathrm{d}} - L_{\mathrm{q}})I_{\mathrm{d}} \cdot I_{\mathrm{q}}]$$

   where,

   $flux_pm$ = Permanment magnet flux linkage (pmsm.Flux_PM)

   $L_{\mathrm{d}},\ L_{\mathrm{q}}$ = Inductance in Henry (pmsm.Ld, pmsm.Lq)

   $I_{\mathrm{d}},\ I_{\mathrm{q}}$ = Current measured in Amperes

   $\mathtt{I}_{\mathtt{d_meas}}$, the measured I_{d} current (in PU), equals `0`.
6. In the `mcb_pmsm_operating_mode_f28379d/Motor and Inverter/Plant Model (sim)` sub system, provide the calculated load torque value as an input to the **LdTrq** port of the PMSM motor block.

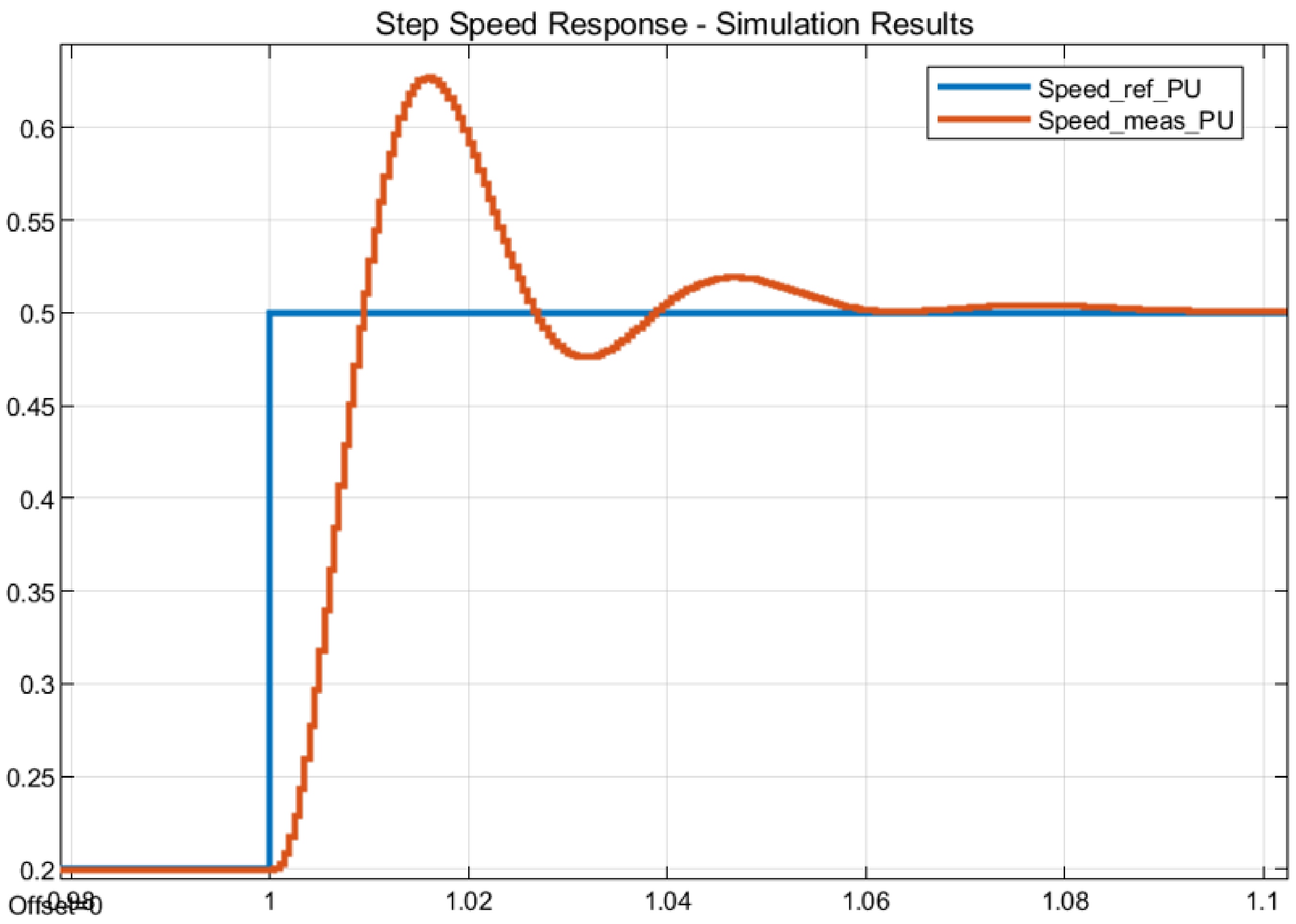
Compare Speed Controller Response in Simulation and in Target Hardware

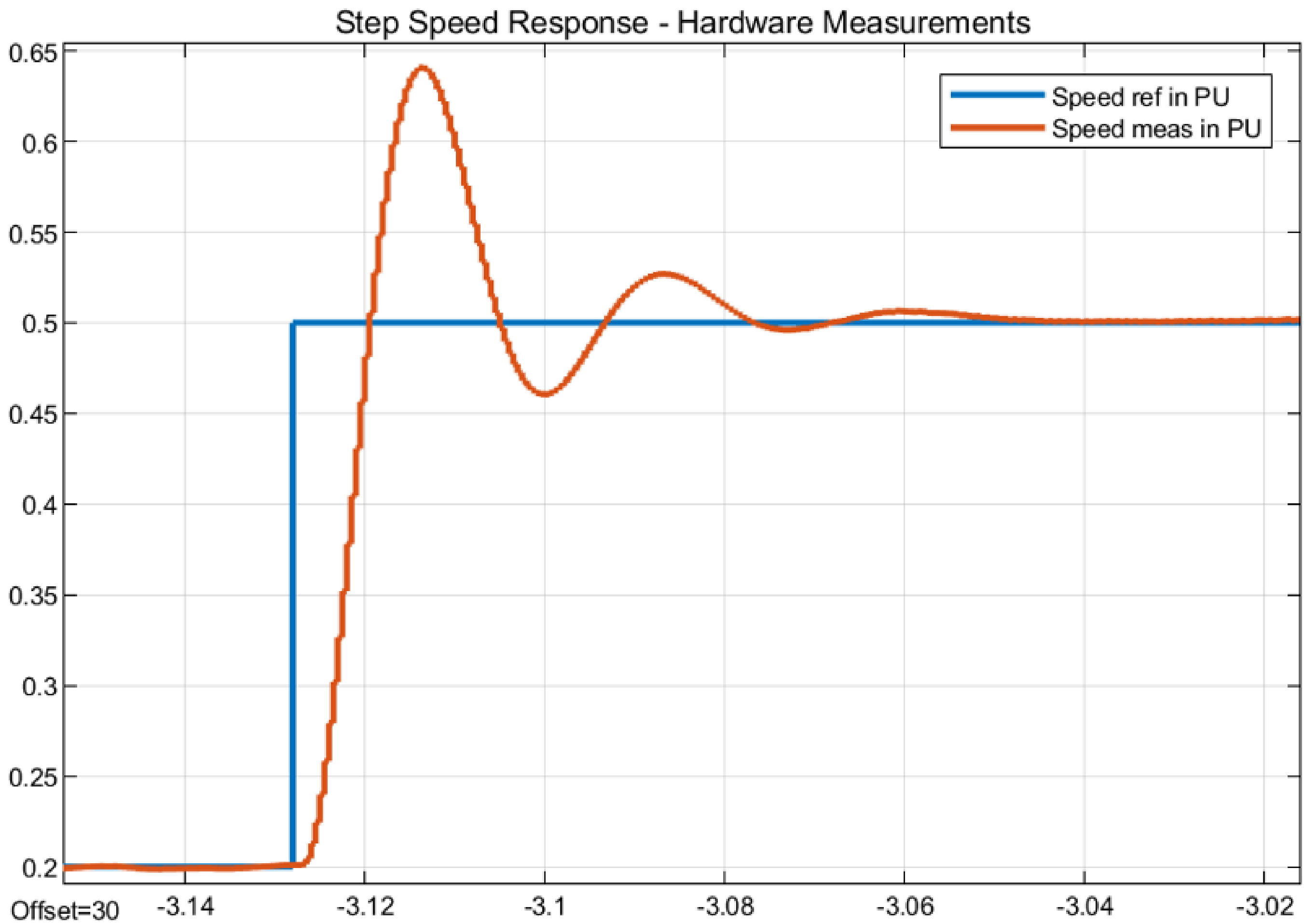
In simulation, provide a speed step input and note the speed response. On the target hardware, command the speed reference step input and observe the speed feedback. Compare the resulting step response in simulation and in target hardware to determine accuracy of the plant model.

1. Simulate the model `mcb_pmsm_operating_mode_f28379d`. Plot the reference speed and measured speed signals. By default, this example provides a step input of `0.2` to `0.5` to the simulation model.
2. Run the host model to communicate with the target hardware.
3. In Select Motor operating mode, change the mode from **Stop** to **Speed control**.
4. In the host model, select **Speed_ref** in **Monitor Signal#1** and **Speed_meas** in **Monitor Signal#2**.
5. Open the scope in the host model.
6. In host model interface, change the `speed_ref` from `0.2` to `0.5` and observe the step change in the scope.
7. Compare the step response of the hardware results with the simulation results.

Step Response Analysis for Speed Controller

Compare the step response obtained from simulation with the measurements obtained from the target hardware. The results may vary depending on the tolerances in the plant model. Generally, simulation results are close to the values measured on the target hardware.


Step Speed Response - Simulation Results
Speed_ref_PU
Speed_meas_PU
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
Offset=0
0.98
1
1.02
1.04
1.06
1.08
1.1



	Peak overshoot (%)	Peak time (ms)	Rise time (ms)	Settling time (ms)
Simulation results	20.13%	16.023	5.561	61.027
Hardware results	22 %	14.324	5.041	51.148

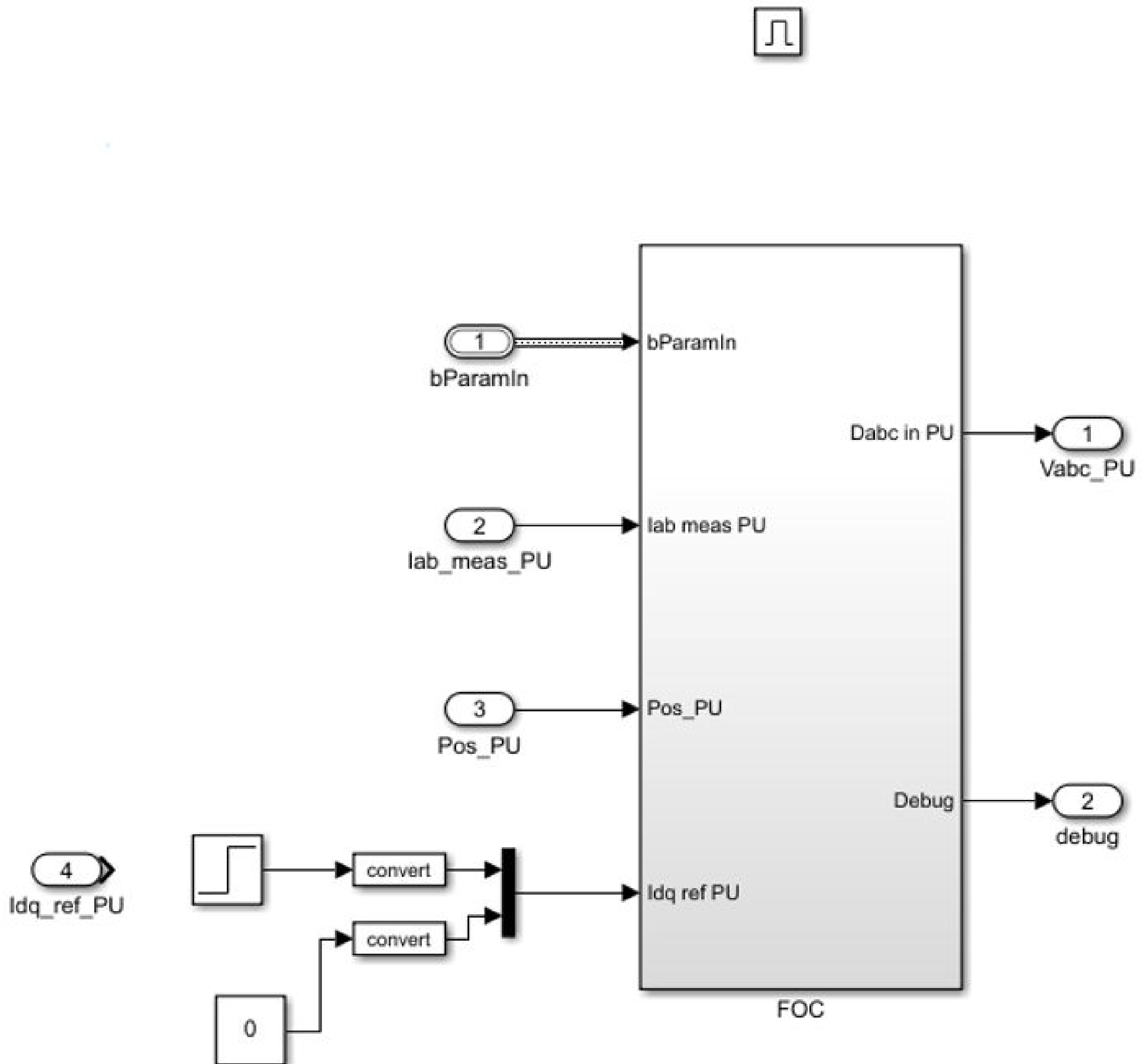
Compare Current Controller Response in Simulation and in Target Hardware

In simulation, provide a step current reference and note the current response. This example needs some changes to simulate the current reference step input. See these steps for the model changes. This applies only to simulation. In the target hardware, command the current reference step input and observe the current feedback. Compare the resulting step response in simulation and in the target hardware to determine the accuracy of the plant model.

1 For hardware measurements, run the host model.
2 In Select Motor operating mode, change the mode from **Stop** to **Torque control**.
3 Select **Id_ref** in **Monitor Signal#1** and **Id_meas** in **Monitor Signal#2** in the host model.
4 Open the scope in the host model.

5 Change `Id_ref` from `0.02` to `0.22` and observe the step change in the scope. Ensure that the motor is not running. The scope displays the step response for the `Id_ref` input.

6 For simulation, two changes are needed in the simulation model. In the `mcb_pmsm_operating_mode_f28379d/TorqueControl/Control Modes/torque_control` subsystem add a step input for the *d*-axis current controller. Choose a step input of `0.02` to `0.22` at `1` second. Select Time sample as `-1`. In the data-type conversion block, select the output datatype as `fixdt(1,32,17)`.



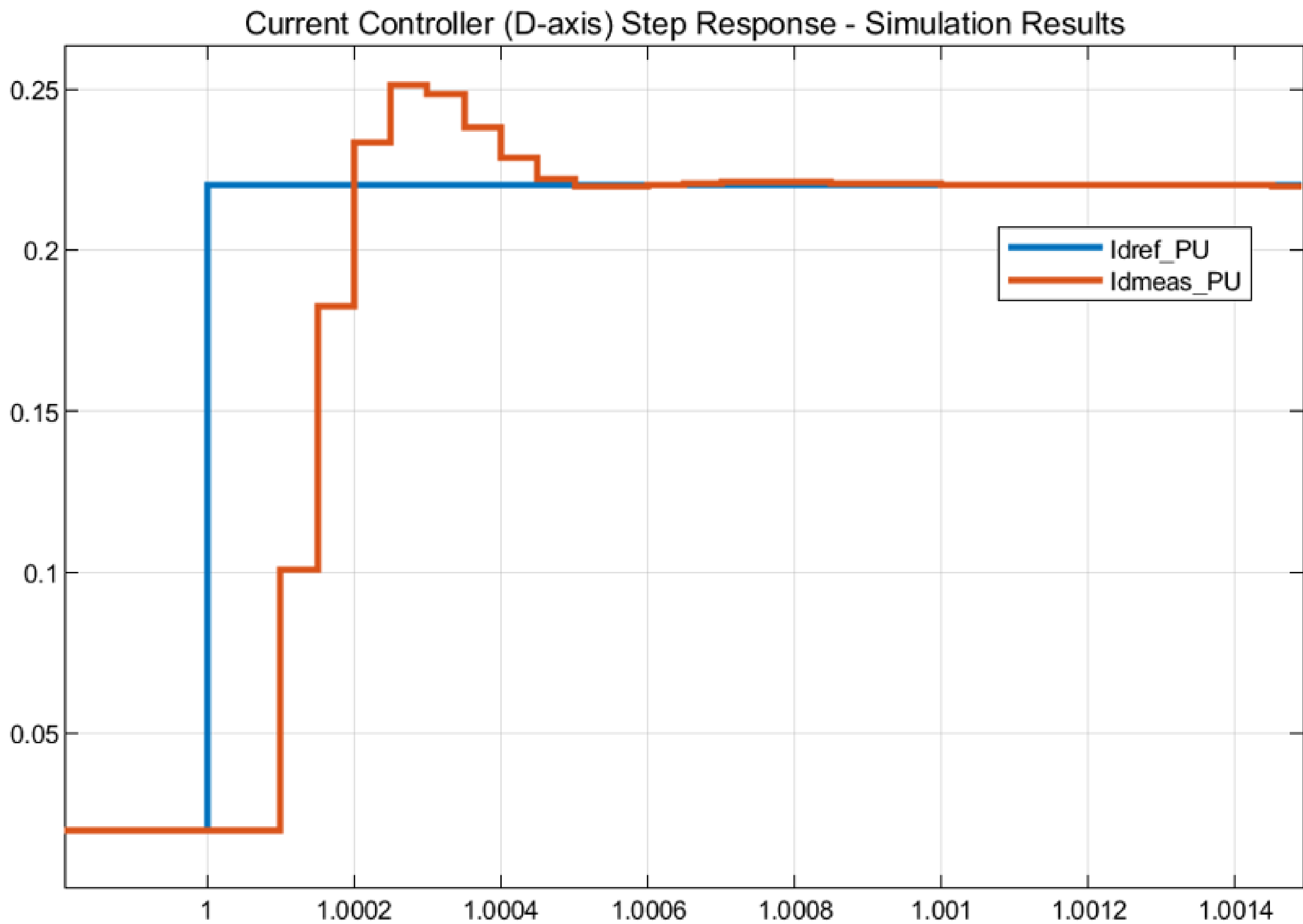
7 In the PMSM motor block available in the `mcb_pmsm_operating_mode_f28379d/Motor and Inverter/Plant Model (sim)` subsystem, change the Mechanical Input Configuration to **Speed** and input `0` to the **Spd** input port.

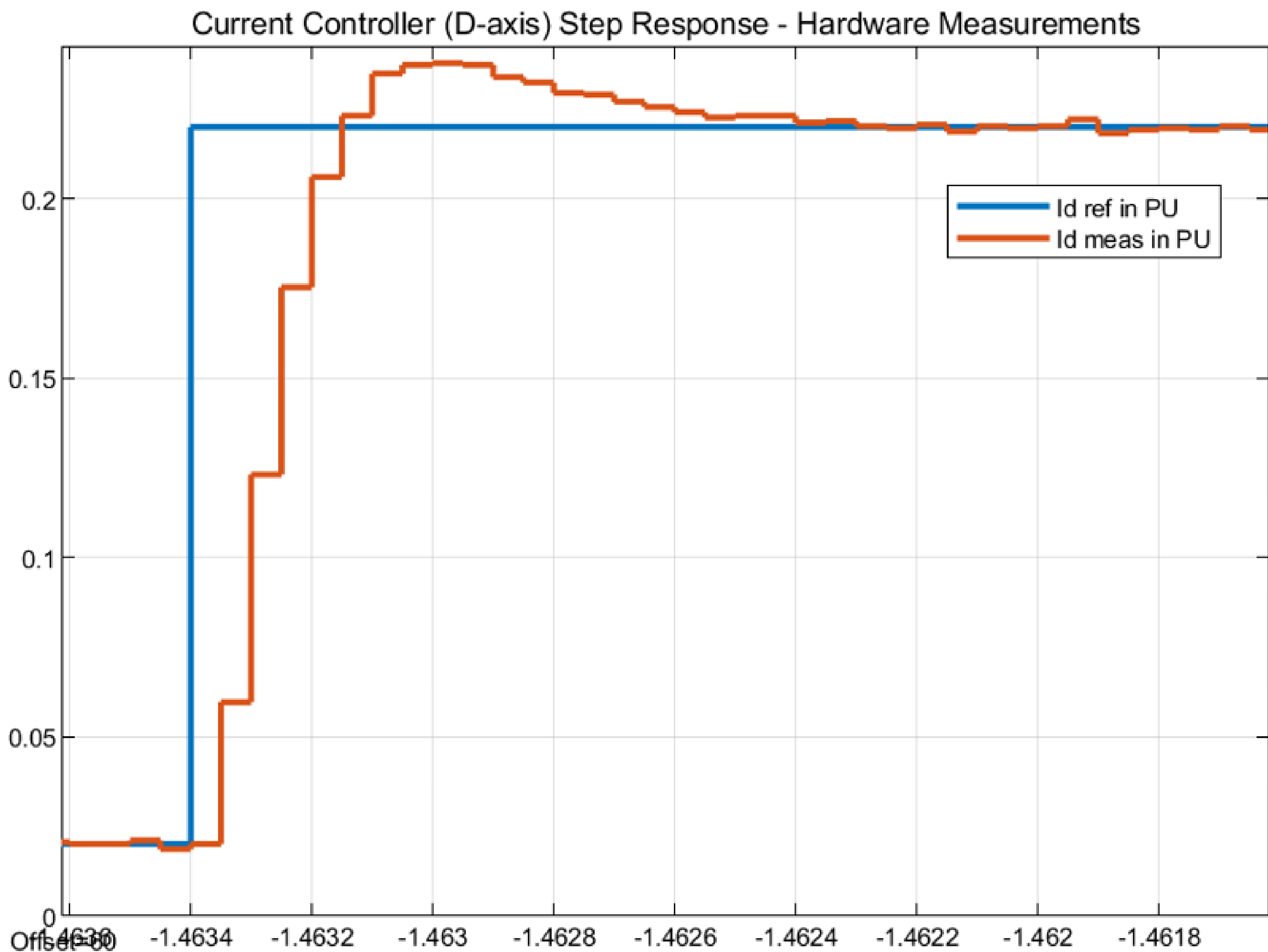
8 Run the simulation and measure the `Idref_PU` and `Idmeas_PU` values in the simulation Data Inspector.

9 Compare the step response in the hardware with the simulation results.

Step Response Analysis for d-axis Current Controller

Compare the scope image from simulation with the measurements from the target hardware. The results may vary depending on the tolerances in the plant model. With an accurate plant model, the simulation results are closer to the measured results from the target hardware.





	Peak overshoot (%)	**Peak time (µs)**	**Rise time (µs)**	**Settling time (µs)**
Simulation results	14 %	300	150	500
Hardware results	8.18 %	400	150	800

The accuracy of the plant model improves the accuracy of simulation and matches the test hardware results.

Tip If the simulation results differ considerably from the hardware measurements, verify the delay and scaling factor in the plant model.

Note For the *q*-axis current controller, align the motor to the *d*-axis and mechanically lock the rotor. Follow this for the *d*-axis current controller for comparative analysis. You can achieve external mechanical locking through the mechanical braking system or by coupling with a dynamometer motor running in speed control.

Plant Modeling

- "Creating Plant Model Using Motor Control Blockset" on page 3-2
- "Use PMSM Block and Motor Parameters to Design Plant Model" on page 3-3
- "Add Average-Value Inverter Block" on page 3-5
- "Create Motor Phase Current Sensing and Signal Conditioning Subsystem" on page 3-6
- "Create Position Sensing Subsystem" on page 3-7
- "Add Delay in Plant Model" on page 3-8
- "Integrate the Blocks and Subsystems" on page 3-9

Creating Plant Model Using Motor Control Blockset

An accurate plant model is a vital part of control system development using Motor Control Blockset. After creating an accurate plant model, you can verify the functionality of the control system, conduct closed-loop model-in-the-loop tests, tune gains using simulation, and optimize the design before you deploy the model in the actual plant.

When you create a plant model using Motor Control Blockset, you model these components to simulate functional behavior in a simulation environment:

- Permanent Magnet Synchronous Motor (PMSM)
- Average-value Inverter
- Sensors and signal conditioning circuits
- Processor peripherals: Analog-to-Digital converter (ADC) and Pulse-width-modulator (PWM)

You can verify the functionality of the plant model you create by:

1. Reading the normalized PWM duty cycle from the control algorithm
2. Simulating the motor for the connected load
3. Obtaining the output motor phase current (in terms of ADC counts) and the output motor position (in terms of encoder pulse counts) from the simulation

The workflow to create a plant model involves these steps.

Note See the plant model in `mcb_pmsm_foc_qep_f28379d` used in the example "Field-Oriented Control of PMSM Using Quadrature Encoder".

---

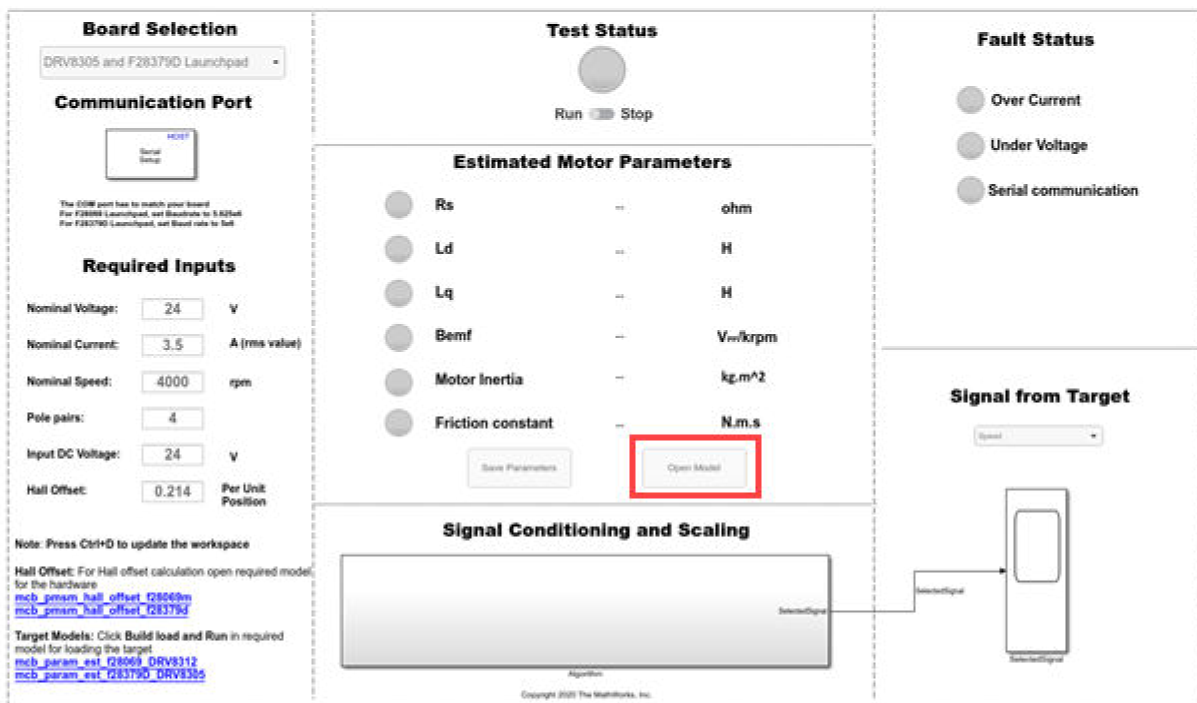
1. "Use PMSM Block and Motor Parameters to Design Plant Model" on page 3-3
2. "Add Average-Value Inverter Block" on page 3-5
3. "Create Motor Phase Current Sensing and Signal Conditioning Subsystem" on page 3-6
4. "Create Position Sensing Subsystem" on page 3-7
5. "Add Delay in Plant Model" on page 3-8
6. "Integrate the Blocks and Subsystems" on page 3-9

Use PMSM Block and Motor Parameters to Design Plant Model

You can use the surface mount or interior PMSM blocks from Motor Control Blockset in two ways to create a plant model.

- Estimate motor parameters by using Motor Control Blockset and open a Simulink model with PMSM motor block (auto-populated with estimated parameters):

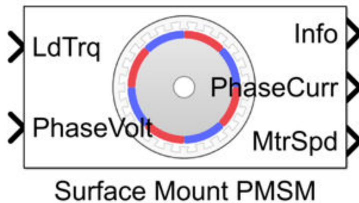
  The Motor Control Blockset parameter estimation workflow helps you to determine the motor parameters by performing a series of tests on the motor. For details, see "Estimate Motor Parameters by Using Motor Control Blockset Parameter Estimation Tool". After successfully estimating the motor parameters, click **Open Model** in the parameter estimation host model. A new model opens with the Interior PMSM block updated with the estimated motor parameters.



- Create a new model and add PMSM motor block manually from the Motor Control Blockset library:

  Create a new Simulink model and add the Surface Mount PMSM block from the Motor Control Blockset library in the Simulink library browser. Open the block mask and enter the motor parameters manually. You can obtain these parameters by using:

  - The Motor Control Blockset parameter estimation workflow. For details, see "Estimate Motor Parameters by Using Motor Control Blockset Parameter Estimation Tool".
  - The motor datasheet or from other known sources.



Surface Mount PMSM

Block Parameters: Surface Mount PMSM

Surface Mount PMSM (mask) (link)

Model the dynamics of a three-phase surface mount permanent magnet synchronous motor (PMSM) with sinusoidal back electromotive force.

Block Options

Mechanical input configuration: Torque

Simulation type: Discrete

Sample Time (Ts): 25e-6

Load Parameters:

File:

Browse

Load from file

Save to file

Parameters | Initial Values

Number of pole pairs (P): pmsm.p

Stator resistance per phase (Rs): pmsm.Rs [Ohm]

Stator d-axis inductance (Ldq_): pmsm.Ld [H]

Permanent flux linkage constant (lambda_pm): pmsm.FluxPM [Wb]

Physical inertia, viscous damping, static friction (mechanical): [pmsm.J, pmsm.B, 0] [Kgm^2, Nm/rad/s, Nm]

OK Cancel Help Apply

In Surface Mount PMSM block, set the **Simulation type** parameter to `Discrete` and the **Sample Time (Ts)** parameter to `25e-6` (half of the control frequency). Discrete simulation improves the simulation speed.

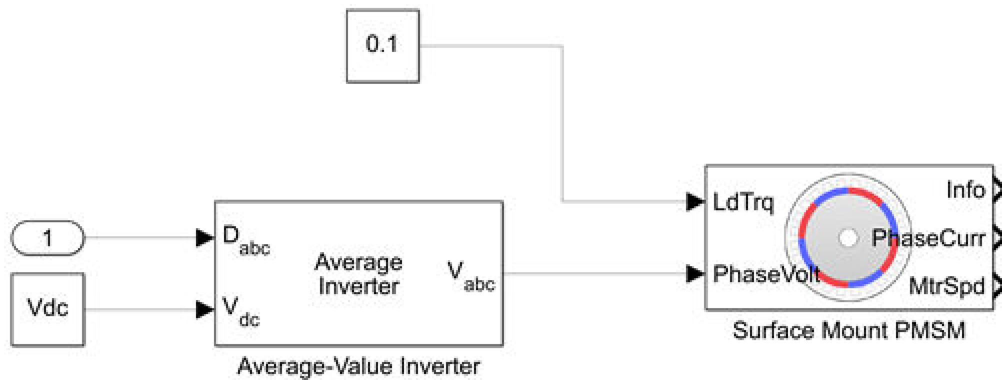
If parameters are available in a MAT-file, click the **Browse** button on the block mask to locate the MAT-file and then click **Load from file** to load the parameters.

The files containing the default motor parameters are available in the location `<matlabroot>\toolbox\autoblks\autoblksshared\mcbtemplates` for your reference.

In the Surface Mount PMSM block mask, you can also represent the motor parameters as workspace variables and use the m-script to automatically update these variables using the model initialization callback. For reference parameters of some commercially available motors, see the file `mcb_SetPMSMMotorParameter.m`. For details about this m-script file, see “Estimate Control Gains from Motor Parameters”)

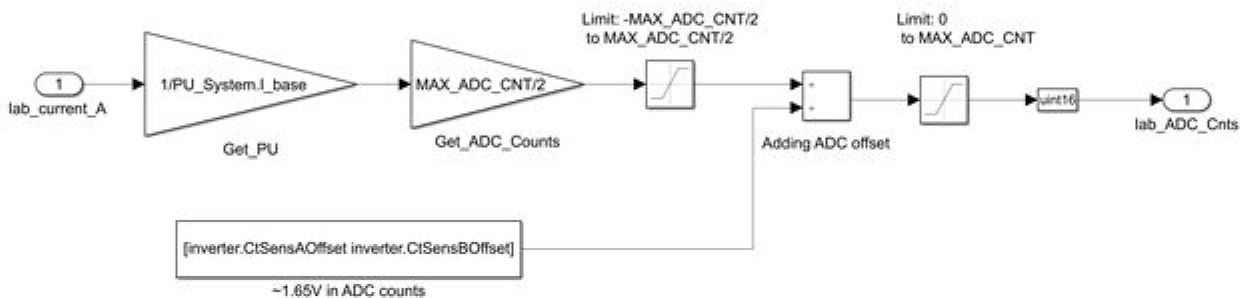
Add Average-Value Inverter Block

In the Simulink model that contains the Surface Mount PMSM block, add an Average-Value Inverter block from the Motor Control Blockset library. The Average-Value Inverter block reads the normalized PWM duty-cycle and DC voltage input (in volts) and outputs the phase voltages. Connect the $\mathbf{V_{abc}}$ output port of the Average-Value Inverter block to the **PhaseVolt** input port of the Surface Mount PMSM block.



Create Motor Phase Current Sensing and Signal Conditioning Subsystem

In the physical hardware, the motor current read by the current sensors is filtered and scaled to an ADC measurable range. The ADC peripheral in the processor reads the current signals and outputs the ADC counts for the current control algorithm. This figure shows an example of how you can model the motor phase current sensing and signal conditioning algorithms.



The maximum measurable peak current is considered as the base current. The ADC counts can be calculated from the base current, full-scale ADC values, along with the ADC offset, by using this equation:

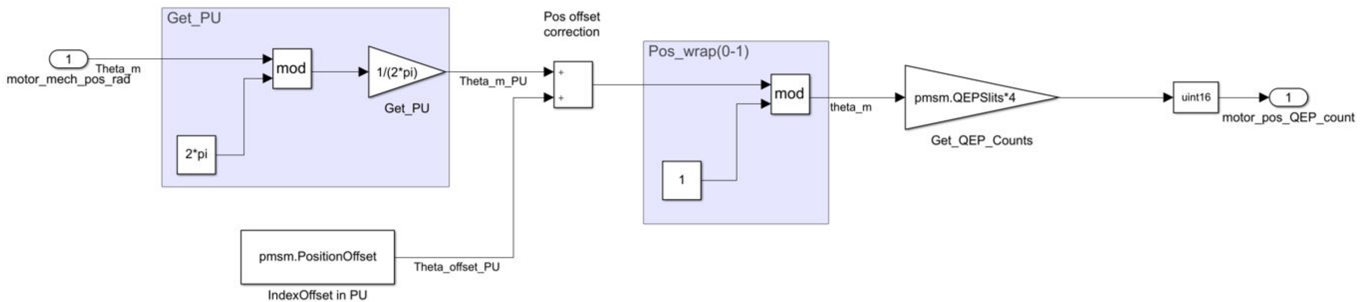
$$\text{ADC counts} = \frac{\left(\text{Full scale ADC counts}/2\right)}{\text{Base current (in amperes)}} + \text{ADC offset}$$

For the default inverter and signal conditioning circuit parameters for commercially available inverters, see the `mcb_SetInverterParameters.m` file. To add a new inverter configuration, create an inverter type in this file and use this in the model initialization script for parameter initialization. If you are using low-pass filters for measuring the current, add an average model to filter the current.

Create Position Sensing Subsystem

The position sensing subsystem reads the motor position from the Surface Mount PMSM block and simulates the QEP encoder pulse counts. The Surface Mount PMSM block outputs the mechanical position of the motor in rad/s.

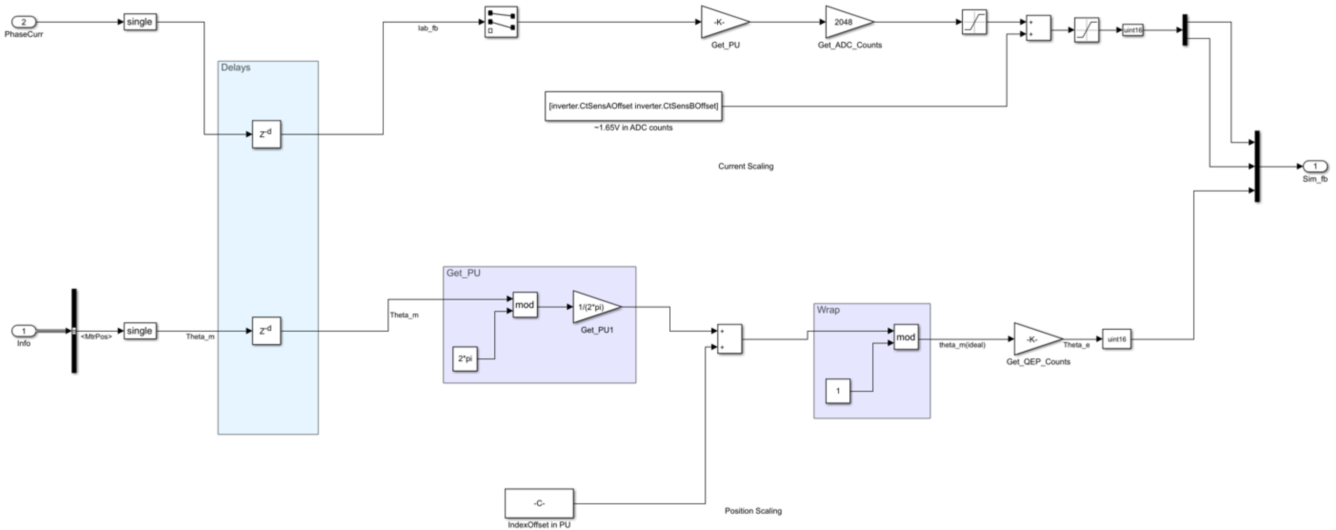
Convert the position in the range 0 to 2π rad/s to QEP encoder counts as shown in this figure.



For details about detecting the QEP index position offset with respect to the rotor *d*-axis, see "Quadrature Encoder Offset Calibration for PMSM Motor".

Add Delay in Plant Model

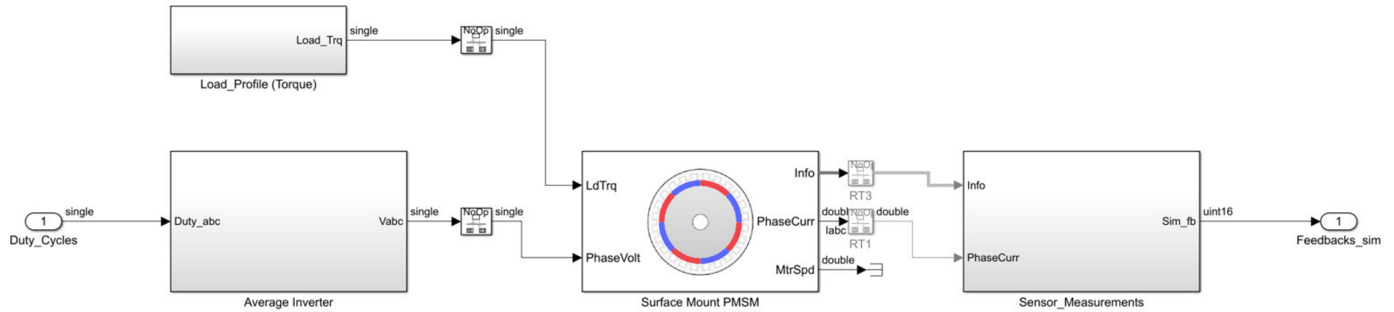
You can add delays in the plant model to simulate the control algorithm processing delays in the hardware and the PWM switching delays. The algorithm processing delay in the processor is the time taken to update the PWM. PWM switching delay is usually half the switching time period.



For adding delays in the discrete time solver with a sample rate of $T_s/2$ (half the switching time period), the processor computation delay and PWM switching delay are factored as Z^{-1} ($T_s/2$).

Integrate the Blocks and Subsystems

The final step of designing the plant model using Motor Control Blockset is to integrate the blocks and subsystems that you created earlier. The completed plant model accepts the normalized PWM from the controller and outputs the motor phase currents and position.



Hardware Troubleshooting

- “Check ADC Inputs” on page 4-2
- “Verify PWM Outputs” on page 4-4
- “Check Hardware Connections” on page 4-6
- “Test Algorithm Design” on page 4-7
- “Check Generated Code” on page 4-8

Check ADC Inputs

Description

The analog to digital converter (ADC) can measure incorrect values. For example, in custom-designed analog circuits, currents measured by ADC can be incorrect due to noise, out-of-phase measurements, or sampling issues. This results in faulty feedback to the control system that leads to instability.

Action

Verify ADC Pin

See the hardware schematics and verify that you identified and configured the correct ADC pins for a given measurement (a-phase, b-phase).

Verify ADC Block Configuration

Open the ADC block and verify that the **Input Channels**, **ADC module**, **SOC trigger**, **SOCx acquisition window** parameters are configured correctly.

ADC sampling begins with the SOC event. In some cases, for example, when sensing the current through the shunt resistors, ADC sampling requires synchronization with the bottom leg switches. In this case, verify that the SOC event is configured correctly with ADC-PWM interrupt synchronization. This also results in reduced EMI/EMC noise in the sampling because ADC conversion happens outside the PWM transition. For more information, see “Task Scheduling in Target Hardware” on page 2-6.

Reduce Noise in ADC Sampling

You may notice noise in the ADC samples. This may happen either if there is EMI/EMC or if sampling is faster than what the device can support. EMI/EMC can be reduced by improving the hardware design.

To avoid problems due to faster sampling, see the device datasheet and determine the maximum supported clock frequency of the ADC. For example, if you are using a Texas Instruments TMS320F28379D series microcontroller, it can support a CPU clock frequency of 200 MHz, but the maximum clock frequency supported by the ADC module is 50 MHz. Use this value to set **ADC clock prescaler (ADCCLK)** parameter on the **Hardware Implementation** tab in the **Configuration Parameter** dialog box of your model.

Check VDD of Current Measurement Device

Many current measurement devices derive VDD from the DC power supply (V_{DC}). In addition, the device enable pin also determines the supply voltage to the internal current measurement circuit (for example, Texas Instruments BOOSTXL-DRV8305). Absence of VDD (or the device enable pin) results in 0 V at the ADC of the target hardware. Ensure that these conditions are not present in your hardware.

Check ADC Current Conventions

Check if you are using the correct conventions for ADC current sensing. Motor Control Blockset considers the current entering the motor (or leaving the inverter) as positive. This convention changes with the hardware because of the differences in the inverting or noninverting op-amp and the analog current sensing circuit.

Test Readability of Unipolar and Bipolar Signals

Check if the measurement circuit is designed to read unipolar and bipolar signals.

DC signal measurement circuits are usually unipolar. For example, BoostXL-DRV8305 has a DC voltage measurement circuit that converts the voltage range of 0 – 44.3 V to 0 – 3.3 V at the ADC. Voltage ADCs cannot measure negative voltages.

AC signal measurement circuits are usually bipolar. For example, BoostXL-DRV8305 has an AC current measurement circuit that converts the current range of –23.57 to +23.57 A to 0 – 3.3 V at the ADC with an offset of 1.65V.

Check ADC Offset and Gain computation

Verify the ADC offset values before deploying and executing the code on the target hardware. For more information, see “Current Sensor ADC Offset and Position Sensor Calibration”.

Check the accuracy of the computed gain for conversion of the ADC counts to signal value in the real world as described in the previous section.

Check ADC Resolution

Check the ADC resolution to determine the minimum value of the signal that it can measure. For example, a 3.3 V 12-Bit ADC that can measure ±16.5 A has a resolution of 0.1 Volts/Ampere. The minimum current that the ADC can measure (excluding EMI/EMC and noise) is approximately 8 mA.

Determine the minimum measurable current by the ADC. Verify that this current is greater than the ADC signal-to-noise ratio, tolerance, and errors. Ensure that you simulate and check the model before deploying it to the target hardware.

Low ADC resolution can result in difficulties when implementing sensorless algorithms to control motors that consume very small currents (for example, 50 mA AC) on no load. In addition, EMI/EMC and noise affects ADC measurements. It is a good practice to simulate the model and verify if the ADC resolution is appropriate. Increase the gain of the sensor amplifier on the hardware to increase the ADC resolution.

Verify PWM Outputs

Description

The motor control algorithm generates the pulse width modulation (PWM) signals to control the motor through inverter. In some cases, the PWM signals can be incorrect due to improper switching frequency, wrong interrupt and PWM generation configurations, or error in the duty cycles. Incorrect PWM signals result in improper switching of the inverter.

Action

Verify PWM Frequency

Use an oscilloscope to verify that the generated PWM signals has the expected switching frequency. In embedded targets, configuration of the PWM module depends on factors such as target hardware and clock frequency. For example, you can use these equations to calculate PWM_Counter_Period for Texas Instruments C2000 targets that have the ePWM module configured to work with the Up-Down counting mode:

CPU_frequency (Hz) = 200e6

PWM_frequency (Hz) = 20e3

PWM_Counter_Period (PWM timer counts) = CPU_frequency/ PWM_frequency/ 2

Verify PWM Generation

Ensure that you feed a correct PWM duty cycle to the switching device (for example, MOSFET or IGBT). PWM generation depends on these active-high and active-low configurations:

- Active high — 25% duty results in 25% on-time for upper leg MOSFET or IGBT (recommended).
- Active low — 25% duty results in 75% on-time for upper leg MOSFET or IGBT.

In addition, check if there is any inversion of the PWM signal between the target and MOSFET due to the gate driver or isolator circuit (25% gate pulse must be 25% on-time by the driver chip).

Verify Interrupt Configuration

Majority of the controller algorithms are designed to work with the ADC-PWM synchronization for advantages like current sensing, reduced EMI/EMC interference.

ADC sampling begins with the SOC event. In some cases, for example, when sensing the current through the shunt resistors, ADC sampling requires synchronization with the bottom leg switches. In this case, verify that the SOC event is configured correctly with the ADC-PWM interrupt synchronization. This also results in reduced EMI/EMC noise in the sampling because ADC conversion happens outside the PWM transition. For more information, see “Task Scheduling in Target Hardware” on page 2-6.

Verify Updates to PWM Duty

Verify if the PWM duty is updated or refreshed in synchronization with the PWM module. To implement a robust control, it is a good practice to timely refresh the PWM duty (for example, once in T_{pwm}, preferably before T_{pwm}/ 2).

Check Behavior at PWM Generation Limits

Check the datasheet of the PWM driver circuit for support at the 0% duty and 100% duty limits. For functional safety, it is a good practice to limit the maximum duty cycle somewhere between 95 and 98% by setting the corresponding value in the DQ Limiter block.

Check for Incorrect PWM Generation Configuration

Verify that the hardware uses the correct PWM generation configuration. For example, BoostXL-DRV8305 supports 3-PWM mode, 6-PWM mode, and 1-PWM mode.

Check for Default Dead Bands

Check if there are dead bands introduced by the motor driver board. Consider this while generating dead bands from the PWM module.

Confirm Maximum Switching Frequency

Determine the maximum possible switching frequency for the inverter and driver from the device datasheets. Ensure that the model does not exceed this value.

Check Hardware Connections

Description

When you try to run the motor, you may face problems due to incorrect hardware connections. This may result in rise in temperature of the motor, inverter, hardware board or an abnormal behavior such as uncontrolled motor speed.

Action

Verify Hardware Connections

Check the wiring and connections before getting started. For details, see “Hardware Connections”.

Manually Check Rotation of Shaft

Verify that the shaft of your motor is rotating freely with minimal rotational friction. A mechanical failure in the bearings may result in thermal overloads, which can damage the motor windings.

Verify Rated Currents for Motor and Inverter

Determine the rated currents of the motor and inverter from the manufacturer datasheet. Ensure that you do not overload the motor for durations longer than what the original equipment manufacturer (OEM) has specified.

Check Motor and Inverter Temperature

Ensure that the temperature of the motor windings and inverter heat sink are within the expected temperature range. Overloading the hardware results in excessive heat that can damage the hardware.

Verify Measurements from Analog Circuits

Verify the range of the signals that you measure from the analog circuits (for example, the maximum current of the inverter).

Check for Additional Resistors

After you complete the process of estimating the motor parameters, you should not change the motor connections because this leads to differences in the contact and cable resistances. In addition, verify that the initialization script of the model takes into consideration any additional resistors present in the power circuit.

Verify Fault Pin and Enable Pin Connections

Check and verify that the fault pins and enable pins are connected correctly on the target hardware board.

Test Algorithm Design

Description

When simulating or running a model on the target hardware, you can face problems because of defects in the implementation of the control algorithm. This can lead to an uncontrolled motor speed, differences in the current waveforms or mismatch in PI controller gains between simulation and target hardware.

Action

Verify Parameters and Other Input Data

Verify that you identified and entered the inputs (for example, motor and inverter parameters, clock speed, and switching frequency) correctly. If the input data is incorrect, the motor control algorithm will not work. Use the Motor Control Blockset parameter estimation tool to compute the motor parameters. For more details, see “Estimate Motor Parameters by Using Motor Control Blockset Parameter Estimation Tool”.

Verify Waveforms of Measured Currents

After you load the motor shaft, verify that the waveforms for the measured signals match the shape visible in the simulations. For example, field-oriented control ensures perfect sinusoidal waveforms for currents. For exceptions, see “Check ADC Inputs” on page 4-2.

Verify Control System Design

Verify that all the controllers used in the model (for example, PI controllers and sliding mode observer) are designed correctly.

You can start by simulating the model by using the estimated motor parameters before deploying the model to the target hardware. Observe and verify the step responses for the current and speed by using both simulation and deployment on the target hardware.

Model-Based Design ensures that correct simulation of the model results in identical outcomes on the target hardware with identical gains (that match the gain values computed during simulation) for all the controllers.

Verify Signal Representation

Check if you can represent the signals correctly for a selected data type. For example, it is not possible to store the value 1024 in the 8-bit data-type. Similarly, it may not be possible to represent some gain values in the selected fixed-point resolution.

Verify Base Values for PU Representation

If you are working with the Per-Unit system, please check that the base value of a quantity (for example, base current), is selected correctly. For more details, see “Per-Unit System”.

Check Generated Code

Description

When simulating or running a model on the target hardware, you may face problems due to errors in the software architecture of a model. These errors can affect the performance of control algorithm and increase the code execution time on the hardware.

Action

Check Sample Times

Verify the base rates and other execution rates of the model by using **Debug** > **Information Overlays** > **Sample Time** > **Colors**. The different sample times of the model decide the execution of different tasks in the simulation and in the generated code.

Check for Overruns

Verify that there are no overruns beyond the available sample time. Algorithms with overruns affect the control system stability. If required, optimize the model for code execution. For more details, see “Code Verification and Profiling Using Processor-In-the-Loop Testing” on page 1-17.

Verify Low-Priority Interrupt Service Routines (ISR)

Verify that the low-priority interrupt service routines (ISR) (for example, speed control loop and communication service routines) are executed according to the design and are not ignored by any overruns in the high-priority ISRs.

Check Execution Order Priority

Check that the model uses a correct execution order priority. Verify that all the interrupts are configured correctly.

Verify Software Initialization

To allow the analog circuits to get ready, check that the software initialization delay (for example, ADC blanking time, PWM driver, and charge pump) is greater than the required value specified by the manufacturer (for example, 2μs).

Check Hardware Initialization

Verify that you initialized the target hardware and inverter correctly. Generally, the driver is disabled, which brings all the switches to a high impedance state and initializes the important variables to the default values.

Verify Third-Party Tool Version

Verify that you are using the recommended versions of the third-party tools. Check that bugs in the third-party software do not cause regressions.