UG10248

MCUXpresso SDK Field-Oriented Control of 3-Phase PMSM and BLDC Motors (FRDMMCXA346)

Rev. 2.0 — 6 February 2026 User guide

Document information

Information	Content
Keywords	FRDM-MCXA346 , PMSM, FOC, MCAT, MID, Motor control, Sensorless control, Speed control, Servo control, Position control
Abstract	This user guide describes the implementation of the motor-control software for 3-phase Permanent Magnet Synchronous Motors.

1 Introduction

This SDK motor-control example user guide describes the implementation of the motor-control software for three-phase Permanent Magnet Synchronous Motors (PMSM) using the following NXP platforms:

- FRDM-MCXA346
- Freedom Development Platform for Low-Voltage, Three-Phase PMSM Motor Control (FRDM-MC-LVPMSM)

The document is divided into several parts. The hardware setup, FRDM-EXT-PMSM processor features, and peripheral settings are described at the beginning of the document. The next part contains the PMSM project description and motor-control peripheral initialization. The last part describes the user interface and additional example features.

The available motor-control example types with supported motors and possible control methods are listed in Table 1.

Table 1. Available example type, supported motors, and control methods

Example type	Supported motor	Possible control methods in SDK example				
		Scalar and voltage	Current FOC (torque)	Sensorless speed FOC	Sensored speed FOC	Sensored position FOC
pmsm_enc	Linix 45ZWN24-40	✓	✓	✓	N/A	N/A
	Teknic M-2311P (with ENC)	✓	✓	✓	✓	✓

The SDK motor-control example is as follows:

- **pmsm_enc** - the pmsm_enc example uses float arithmetic. The example contains sensored and sensorless Field-Oriented Vector Control (FOC). This example can be used both for sensored and sensorless motor-control applications. The default motor configuration is tuned for the Teknic M-2311P motor.

The SDK motor-control example contains several additional features:

- The **FreeMASTER** pmsm_float_enc.pmpx project provides a simple and user-friendly way for algorithm tuning, software control, debugging, and diagnostics.
- The **MCAT** (Motor Control Application Tuning) page is based on the FreeMASTER runtime debugging tool.
- **MID** (Motor Parameter Identification).

The control software and the PMSM control theory are described in *Sensorless PMSM Field-Oriented Control (FOC)* (document DRM148).

2 Hardware setup

The following chapter describes the hardware and setup needed for the example to work properly.

2.1 Linix motor

The Linix 45ZWN24-40 motor is a low-voltage three-phase PMSM motor with hall sensors, used in PMSM applications. The motor parameters are listed below.

Table 2. Linix 45ZWN24-40 motor parameters

Characteristic	Symbol	Value	Units
Rated voltage	Vt	24	V
Rated speed	-	4000	RPM

Table 2. Linix 45ZWN24-40 motor parameters*...continued*

Characteristic	Symbol	Value	Units
Rated torque	T	0.0924	Nm
Rated power	P	40	W
Continuous current	Ics	2.34	A
Number of pole-pairs	pp	2	-

Figure 1. Linix 45ZWN24-40 PMSM

The motor has two types of connectors (cables). The first cable has three wires and powers the motor. The second cable has five wires and is designated for the hall sensors signal sensing. For the PMSM sensorless application, only the power input wires are needed.

2.2 Teknic motor

The Teknic M-2310P-LN-04K and M-2311P-LN-08D are low-voltage three-phase PMSM motors used in PMSM applications. The motors have two feedback sensors (hall and encoder). For information on the wiring of feedback sensors, see the data sheet on the manufacturer webpage. The motor parameters are listed below.

Table 3. Teknic M-2310P-LN-04K motor parameters

Characteristic	Symbol	Value	Units
Rated speed	-	6000	RPM
Continuous (RMS) torque	T	0.247	Nm
Continuous (RMS) power	P	170	W
Continuous (RMS) current	Ics	7.1	A
Number of pole-pairs	pp	4	-
Encoder density	-	4000 (1000)	Count/revolution (pulses)

Table 4. Teknic M-2311P-LN-08D motor parameters

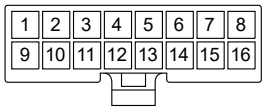
Characteristic	Symbol	Value	Units
Rated speed	-	6000	RPM

Table 4. Teknic M-2311P-LN-08D motor parameters*...continued*

Characteristic	Symbol	Value	Units
Continuous (RMS) torque	T	0.4	Nm
Continuous (RMS) power	P	206	W
Continuous (RMS) current	Ics	7.7	A
Number of pole-pairs	pp	4	-
Encoder density	-	8000 (2000)	Count/revolution (pulses)

Figure 2. Teknic PMSM

For the sensorless control mode, you only need the power input wires. If used with the hall or encoder sensors, connect the sensor wires to the NXP Freedom power stage.



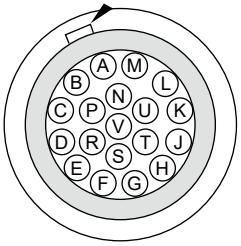
(Wire entry view)

Motor phases

Pin	Color	Signal	Pin	Color	Signal
1	DRAIN x3	P DRAIN	9	16AWG BLK	PHASE R
2	N/A	N/A	10	16AWG RED	PHASE S
3	GRN	COMM S-T	11	16AWG WHT	PHASE T
4	GRN/WHT	COMM R-S	12	RED	+5VDC IN
5	GRY/WHT	COMM T-R	13	BRN	ENC 1
6	DRAIN x1	E DRAIN	14	ORN	ENC B
7	BLK	GND	15	BLU	ENC A
8*	BLU/WHT	ENC A~	16*	ORN/WHT	ENC B~

Encoder wires

Figure 3. Teknic motor connector type 1



Motor phases (Mating face shown) Encoder wires

Pin	Color	Signal	Pin	Color	Signal
R	DRAIN x3	P DRAIN	L	GRY/WHT	COMM T-R
C	16AWG RED	PHASE S	U	BRN	ENC I
D	16AWG WHT	PHASE T	G	GRN	COMM S-T
B	16AWG BLK	PHASE R	T	RED	+5VDC IN
J	BLU	ENC A	F*	ORN/WHT	ENC B~
K*	BLU/WHT	ENC A~	V	ORN	ENC B
H	GRN/WHT	COMM R-S	M	DRAIN x1	E DRAIN
S	BLK	GND			

Figure 4. Teknic motor connector type 2

2.3 FRDM-MC-LVPMSM

In a shield form factor, this evaluation board effectively turns an NXP Freedom development board or an evaluation board into a complete motor-control reference design. It is compatible with existing NXP Freedom development boards and evaluation boards. The Freedom motor-control headers are compatible with the Arduino R3 pin layout.

The FRDM-MC-LVPMSM low-voltage three-phase PMSM Freedom development platform board has a power supply input voltage of 24 VDC - 48 VDC with reverse-polarity-protection circuitry. The auxiliary power supply of 5.5 VDC is created to supply the FRDM MCU boards. The output current is up to 5 A RMS. The inverter itself is realized by a three-phase bridge inverter (six MOSFETs) and a three-phase MOSFET gate driver. The analog quantities (such as the three-phase motor currents, DC-bus voltage, and DC-bus current) are sensed on this board. There is also an interface for speed and position sensors (encoder, hall).

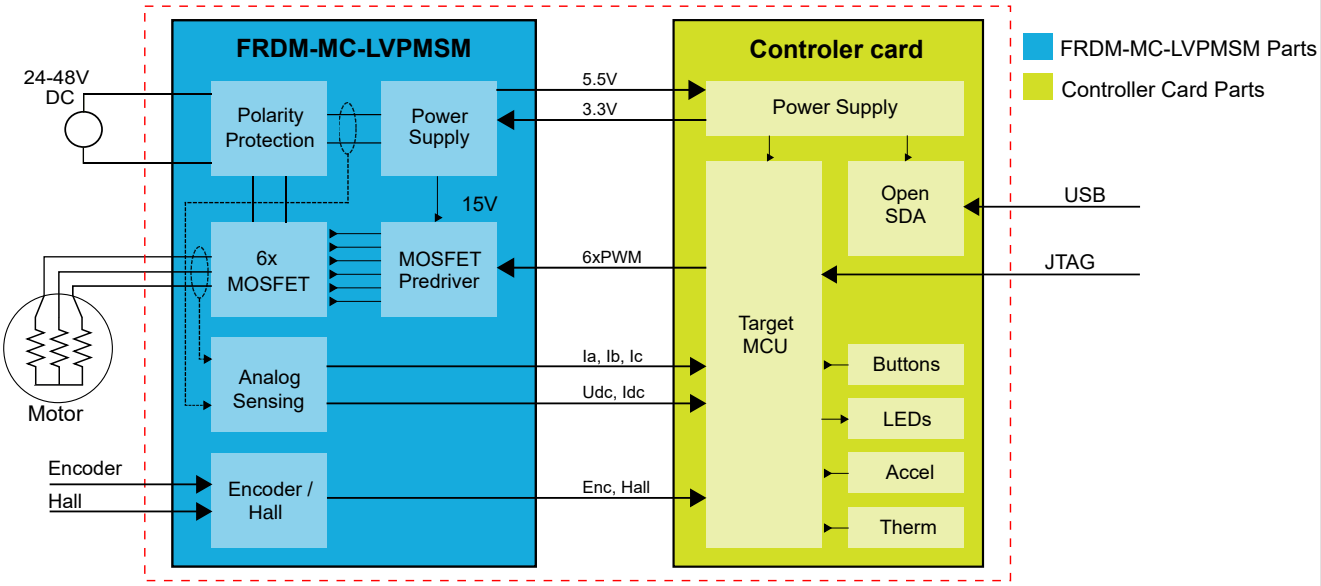


Figure 5. Motor-control development platform block diagram

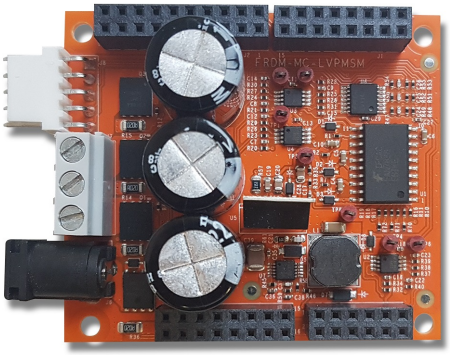


Figure 6. FRDM-MC-LVPMSM

The FRDM-MC-LVPMSM board does not require a complicated setup. For more information about the Freedom development platform, see www.nxp.com.

Note: *There might be a wrong FRDM-MC-LVPMSM series in the market (series VV19520XXX). This series is populated with 10-mΩ shunt resistors and noisy operational amplifiers which affect the phase-current measurement. The* `mc_pmsm` *example is tuned for the original FRDM-MC-LVPMSM board with 20-mΩ shunt resistors.*

Note: *On newer boards, transistor overheating may occur. Therefore, it may be necessary to increase the default dead-time from 500 ns to 1000 ns (M1_PWM_DEADTIME macro).*

2.4 FRDM-MCXA346

Table 5. FRDM-MCXA346 jumper settings (default)

Jumper	Setting	Jumper	Setting
JP1	1-2	JP3	1-2
JP2	1-2	JP7	1-2

All others jumpers are open.

2.4.1 Hardware assembling

1. Connect the FRDM-MC-LVPMSM shield on top of the FRDM-MCXA346 board (there is only one possible option).
2. Connect the 3-phase motor wires to the screw terminals (J7) on the Freedom PMSM power stage.
3. Plug the USB cable from the USB host to the Debug USB connector J15 (MCU Link) on the FRDM board.
4. Plug the 24-V DC power supply to the DC power connector on the Freedom PMSM power stage.

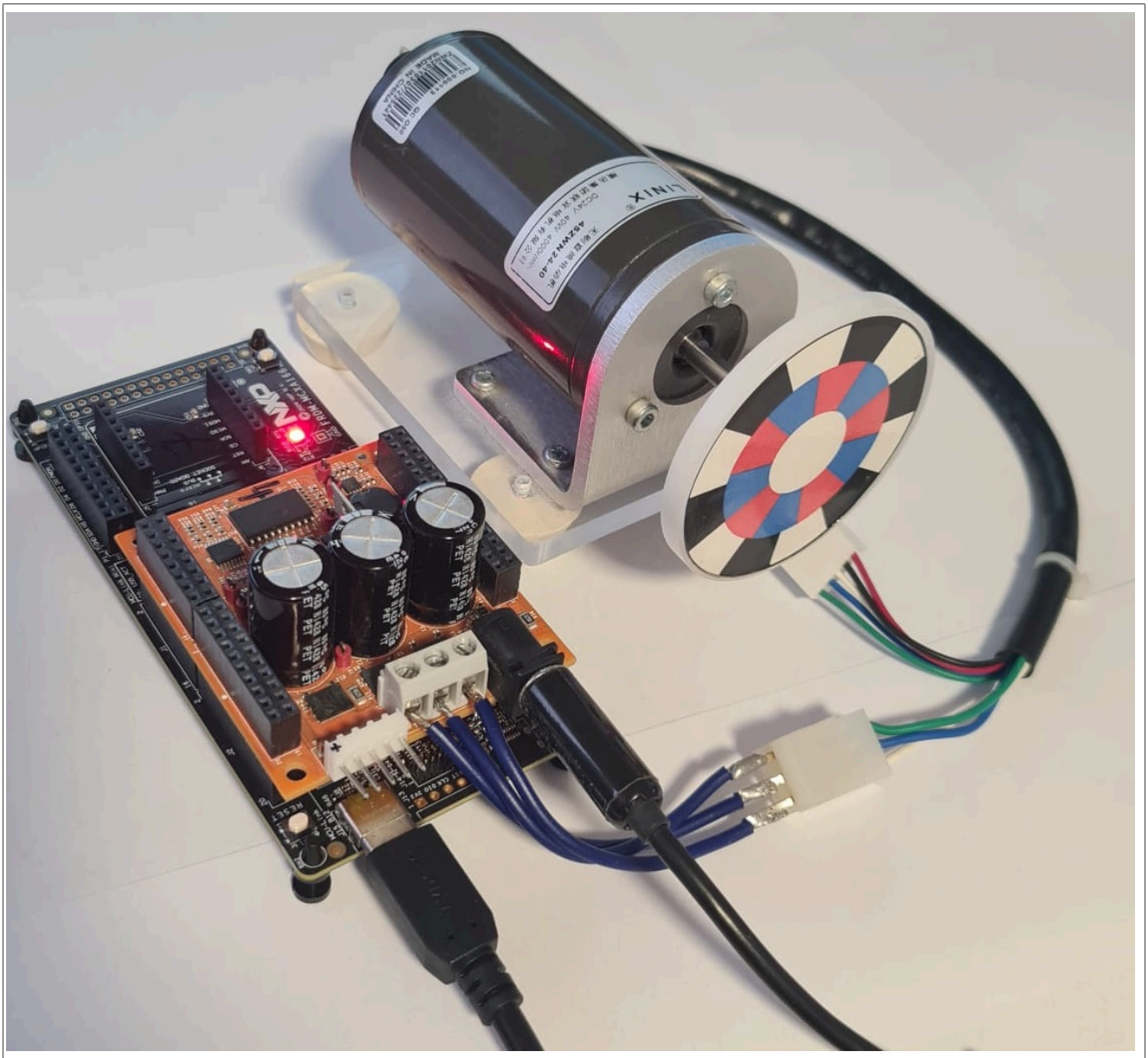


Figure 7. Assembled Freedom system

Note: *The example has been tested on the board with schematic number SCH-93951 REVB.*

3 Processors features and peripheral settings

This chapter describes the peripheral settings and application timing.

3.1 MCX Axxx

The MCX portfolio is a comprehensive selection of Arm Cortex-M based MCUs, offering expanded scalability with breakthrough product capabilities, simplified system design, and a developer-focused experience through the widely adopted MCUXpresso suite of software and tools. The new simplified system design offers optimal enablement and intelligent peripherals for the intelligent edge including machine learning, wireless, voice, motor control, analog, and more. The MCX portfolio is part of the NXP EdgeVerse edge computing platform.

The MCX A series MCUs expand the MCX Arm Cortex-M33 product offerings with multiple high-speed connectivity, serial peripherals, timers, analog, and low power consumption.

For more information, see the MCX General-Purpose MCUs web page.

3.1.1 Hardware timing and synchronization

Correct and precise timing is crucial for motor-control applications. Therefore, the motor-control-dedicated peripherals take care of the timing and synchronization on the hardware layer. In addition, you can set the PWM frequencies as a multiple of the ADC interrupt (ADC ISR) frequency, where the FOC algorithm is calculated. In this case, the PWM frequency is equal to the FOC frequency.

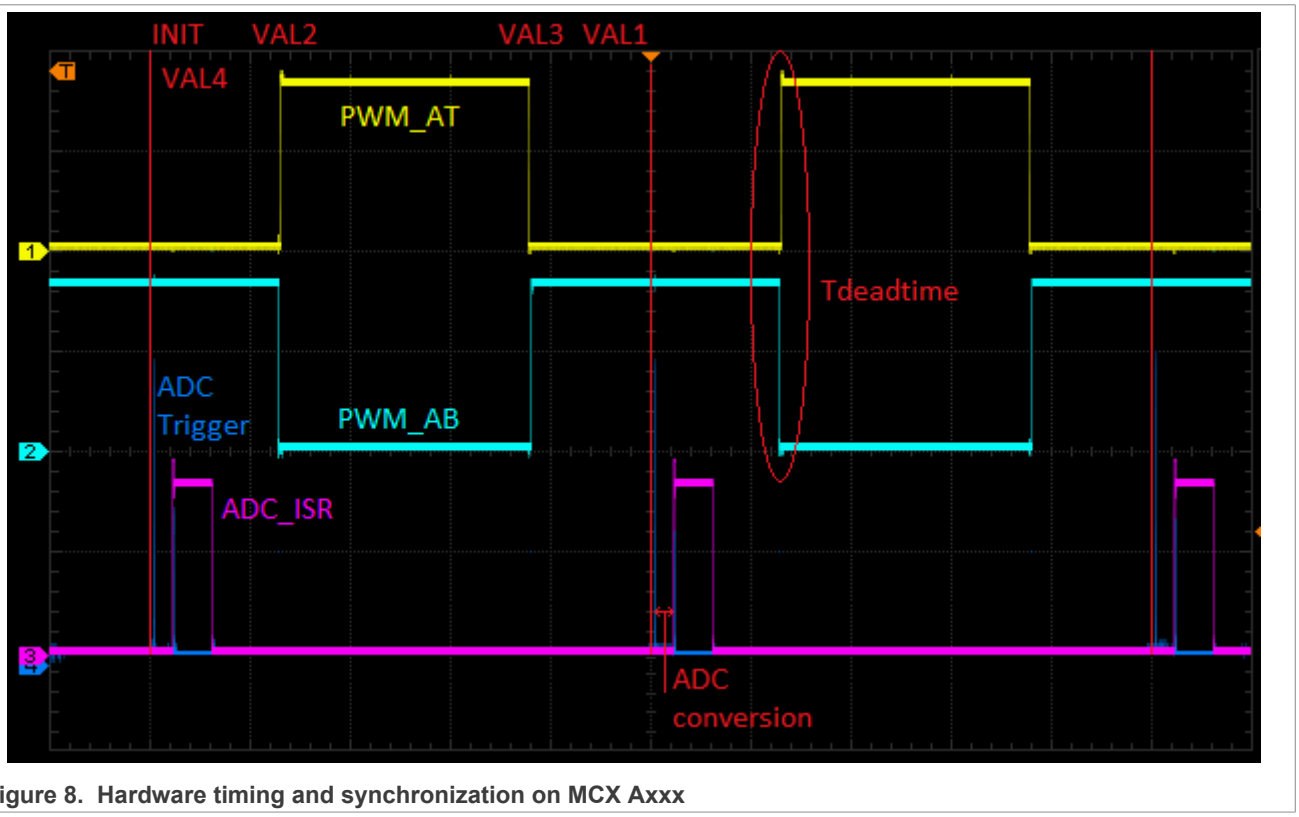


Figure 8. Hardware timing and synchronization on MCX Axxx

- The top signal shows the PWM_AT (PWM phase A - top) and PWM_AB (PWM phase A - bottom). The dead time is emphasized at the PWM top and PWM bottom signals.

- The eFlexPWM submodule SM0 generates trigger 0 (ADC trigger) when the counter counts to a value equal to the VAL4 value. The ADC trigger is delayed by approximately Tdeatime/2. This delay ensures correct current sampling at duty cycles close to 100 %.
- When the ADC conversion completes, the ADC_ISR (ADC interrupt) is entered. The FOC calculation is done in this interrupt.

3.2 CPU load and memory usage

The following information applies to the application built using one of the following IDEs: MCUXpresso IDE, IAR, Keil MDK, or CodeWarrior. The memory usage is calculated from the *.map linker file, including the FreeMASTER recorder buffer allocated in RAM. In the MCUXpresso IDE, you can see the memory usage after the project build in the Console window. The table below shows the maximum CPU load of the supported examples. The CPU load is measured using the SYSTICK timer. The CPU load is dependent on the fast-loop (FOC calculation) and slow-loop (speed-loop) frequencies. The total CPU load is calculated using the following equations:

$$CPU_{fast} = cycles_{fast} \frac{f_{fast}}{f_{CPU}} 100 \left[\%\right] \quad (1)$$

$$CPU_{slow} = cycles_{slow} \frac{f_{slow}}{f_{CPU}} 100 \left[\%\right] \quad (2)$$

$$CPU_{total} = CPU_{fast} + CPU_{slow} \left[\%\right] \quad (3)$$

Where:

CPU_{fast} = the CPU load taken by the fast loop

$cycles_{fast}$ = the number of cycles consumed by the fast loop

f_{fast} = the frequency of the fast-loop calculation

f_{CPU} = CPU frequency

CPU_{slow} = the CPU load taken by the slow loop

$cycles_{slow}$ = the number of cycles consumed by the slow loop

f_{slow} = the frequency of the slow-loop calculation

CPU_{total} = the total CPU load consumed by the motor control

Table 6. Maximum CPU load (fast loop)

	MCXA346
CPU load	24.36 % (16% if code is placed to RAM)
Conditions	IAR IDE, Debug configuration, RTCESL_MAU_ON, Optimization High-Balanced

Table 7. Memory usage

	MCX A346
Read-only code memory	41 680 B
Read-only data memory	15 344 B
Read/write data memory	5 852 B
Conditions	IAR IDE, debug configuration, RTCESL_MAU_ON, high-balanced optimization

Note: *Memory usage and maximum CPU load can differ depending on the used IDEs and settings.*

4 SDK package project file and IDE workspace structure

All the necessary files are included in one package, which simplifies the distribution and decreases the size of the final package. The directory structure of this package is simple, easy to use, and organized logically. The folder structure used in the IDE differs from the structure of the PMSM package installation, but it uses the same files. The different organization is chosen due to better manipulation of folders and files in workplaces and the possibility of adding or removing files and directories. The `pack_motor_<board_name>` project includes all the available functions and routines. This project serves for development and testing purposes.

4.1 PMSM project structure

The directory tree of the PMSM project is shown in the following figure.

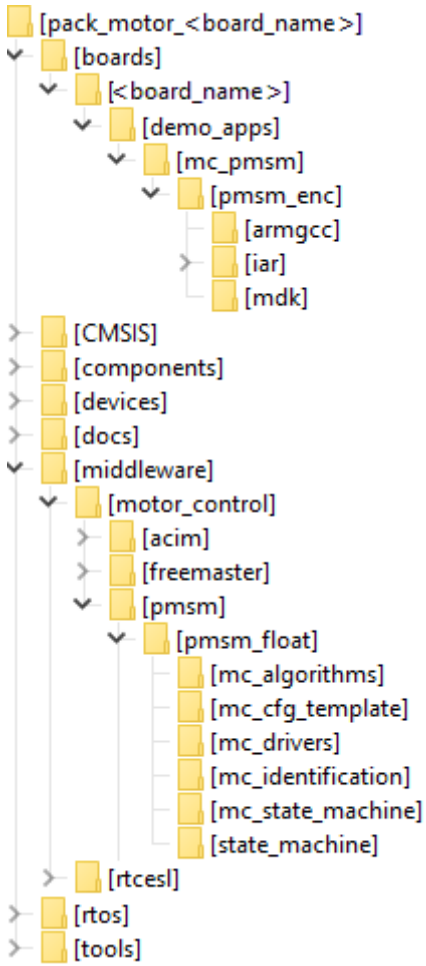


Figure 9. Directory tree

The main project folder `pack_motor_<board_name>\boards\<board_name>\demo_apps\mc_pmsm\pmsm_enc\` contains the following folders and files:

- `iar`: for the IAR Embedded Workbench IDE.
- `armgcc`: for the GNU Arm IDE.
- `mdk`: for the uVision Keil IDE.

- m1_pmsm_appconfig.h: contains the definitions of constants for the application control processes, parameters of the motor and regulators, and the constants for other vector-control-related algorithms. When you tailor the application for a different motor using the Motor Control Application Tuning (MCAT) tool, the tool generates this file at the end of the tuning process.
- main.c: contains the basic application initialization (enabling interrupts), subroutines for accessing the MCU peripherals, and interrupt service routines. The FreeMASTER communication is performed in the background infinite loop.
- board.c: contains the functions for the UART, GPIO, and SysTick initialization.
- board.h: contains the definitions of the board LEDs, buttons, UART instance used for FreeMASTER, and so on.
- clock_config.c and .h: contain the CPU clock setup functions. These files are going to be generated by the clock tool in the future.
- mc_periph_init.c: contains the motor-control driver peripherals initialization functions that are specific for the board and MCU used.
- mc_periph_init.h: header file for mc_periph_init.c. This file contains the macros for changing the PWM period and the ADC channels assigned to the phase currents and board voltage.
- freemaster_cfg.h: the FreeMASTER configuration file, containing the FreeMASTER communication and features setup.
- pin_mux.c and .h: port configuration files. Generate these files in the pin tool.
- peripherals.c and .h: MCUXpresso Config Tool configuration files.

The main motor-control folder pack_motor_<board_name>\middleware\motor_control\ contains these subfolders:

- pmsm: contains the main PMSM motor-control functions.
- freemaster: contains the FreeMASTER project file pmsm_float_enc.pmpx. Open this file in the FreeMASTER tool and use it to control the application. The folder also contains the auxiliary files for the MCAT tool.

The pack_motor_<board_name>\middleware\motor_control\pmsm\pmsm_float\ folder contains these subfolders, common to the other motor-control projects:

- mc_algorithms: contains the main control algorithms used to control the FOC and speed-control loop.
- mc_cfg_template: contains templates for MCUXpresso Config Tool components.
- mc_drivers: contains the source and header files used to initialize and run motor-control applications.
- mc_identification: contains the source code for the automated parameter-identification routines of the motor.
- mc_state_machine: contains the software routines that are executed when the application is in a particular state or state transition.
- state_machine: contains the state machine functions for the FAULT, INITIALIZATION, STOP, and RUN states.

5 Github project structure

More information about the examples released through NXP MCUXpresso Github are in the MCUXpresso SDK Documentation.

6 Motor-control peripheral initialization

The motor-control peripherals are initialized by calling the MCDRV_Init_M1() function during MCU startup and before the peripherals are used. All initialization functions are in the mc_periph_init.c source file and the mc_periph_init.h header file. The definitions specified by the user are also in these files. The features

provided by the functions are the 3-phase PWM generation and 3-phase current measurement, as well as the DC-bus voltage and auxiliary quantity measurement. The principles of both the 3-phase current measurement and the PWM generation using the Space Vector Modulation (SVM) technique are described in *Sensorless PMSM Field-Oriented Control* (document DRM148).

The `mc_periph_init.h` header file provides the following macros defined by the user:

- `M1_MCDRV_ADC_PERIPH_INIT`: this macro calls the ADC peripheral initialization.
- `M1_MCDRV_PWM_PERIPH_INIT`: this macro calls the PWM peripheral initialization.
- `M1_MCDRV_QD_ENC`: this macro calls the QD peripheral initialization.
- `M1_PWM_FREQ`: the value of this definition sets the PWM frequency.
- `M1_FOC_FREQ_VS_PWM_FREQ`: enables you to call the fast-loop interrupt at every first, second, third, or n^{th} PWM reload. This is convenient when the PWM frequency must be higher than the maximal fast-loop interrupt.
- `M1_SPEED_LOOP_FREQ`: the value of this definition sets the speed-loop frequency (TMR1 interrupt).
- `M1_PWM_DEADTIME`: the value of the PWM dead time in nanoseconds.
- `M1_PWM_PAIR_PH[A..C]`: these macros enable a simple assignment of the physical motor phases to the PWM periphery channels (or submodules). You can change the order of the motor phases this way.
- `M1_ADC[1,2]_PH_[A..C]`: these macros assign the ADC channels for the phase current measurement. The general rule is that at least one-phase current must be measurable on both ADC converters, and the two remaining phase currents must be measurable on different ADC converters. The reason for this is that the selection of the phase current pair to measure depends on the current SVM sector. If this rule is broken, a preprocessor error is issued. For more information about the 3-phase current measurement, see *Sensorless PMSM Field-Oriented Control* (document DRM148).
- `M1_ADC[1,2]_UDCB`: this define is used to select the ADC channel for the measurement of the DC-bus voltage.

In the motor-control software, the following API-serving ADC and PWM peripherals are available:

- The APIs available for the ADC are:
  - `mcdrv_adc_t`: MCDRV ADC structure data type.
  - `void M1_MCDRV_ADC_PERIPH_INIT()`: this function is called by default during the ADC peripheral initialization procedure invoked by the `MCDRV_Init_M1()` function and should not be called again after the peripheral initialization is done.
  - `void M1_MCDRV_CURR_3PH_CHAN_ASSIGN(mcdrv_adc_t*)`: calling this function assigns proper ADC channels for the next 3-phase current measurement based on the SVM sector.
  - `void M1_MCDRV_CURR_3PH_CALIB_INIT(mcdrv_adc_t*)`: this function initializes the phase-current channel-offset measurement.
  - `void M1_MCDRV_CURR_3PH_CALIB(mcdrv_adc_t*)`: this function reads the current information from the unpowered phases of a stand-still motor and filters them using moving average filters. The goal is to obtain the value of the measurement offset. The length of the window for moving the average filters is set to eight samples by default.
  - `void M1_MCDRV_CURR_3PH_CALIB_SET(mcdrv_adc_t*)`: this function asserts the phase-current measurement offset values to the internal registers. Call this function after a sufficient number of `M1_MCDRV_CURR_3PH_CALIB()` calls.
  - `void M1_MCDRV_CURR_3PH_VOLT_DCB_GET(mcdrv_adc_t*)`: this function reads and calculates the actual values of the 3-phase currents, DC-bus voltage, and auxiliary quantity.
- The APIs available for the PWM are:
  - `mcdrv_pwma_pwm3ph_t`: MCDRV PWM structure data type.
  - `void M1_MCDRV_PWM_PERIPH_INIT`: this function is called by default during the PWM periphery initialization procedure invoked by the `MCDRV_Init_M1()` function.
  - `void M1_MCDRV_PWM3PH_SET(mcdrv_pwma_pwm3ph_t*)`: this function updates the PWM phase duty cycles.

  - `void M1_MCDRV_PWM3PH_EN(mcdrv_pwma_pwm3ph_t*)`: this function enables all PWM channels.
  - `void M1_MCDRV_PWM3PH_DIS(mcdrv_pwma_pwm3ph_t*)`: this function disables all PWM channels.
  - `bool_t M1_MCDRV_PWM3PH_FLT_GET(mcdrv_pwma_pwm3ph_t*)`: this function returns the state of the overcurrent fault flags and automatically clears the flags (if set). This function returns true when an overcurrent event occurs. Otherwise, it returns false.
- The APIs available for the quadrature encoder are:
  - `mcdrv_qd_enc_t`: MCDRV QD structure data type.
  - `void M1_MCDRV_ENC_PERIPH_INIT()`: this function is called by default during the QD periphery initialization procedure, invoked by the `MCDRV_Init_M1()` function.
  - `void M1_MCDRV_ENC_GET(mcdrv_qd_enc_t*)`: this function returns the actual position and speed.
  - `void M1_MCDRV_ENC_SET_DIRECTION(mcdrv_qd_enc_t*)`: this function sets the direction of the quadrature encoder.
  - `void M1_MCDRV_ENC_SET_PULSES(mcdrv_qd_enc_t*)`: this function sets the number of pulses of the quadrature encoder.
  - `void M1_MCDRV_ENC_CLEAR(mcdrv_qd_enc_t*)`: this function clears the internal variables and decoder counter.

Note: *Not all macros are available for every motor-control example type. The structure data types may vary across platforms.*

7 User interface

The application contains the demo mode to demonstrate the motor rotation. You can operate it either using the user button or using FreeMASTER. The NXP development boards include a user button associated with a port interrupt (generated whenever one of the buttons is pressed). At the beginning of the ISR, a simple logic executes and the interrupt flag clears. When you press the button, the demo mode starts. When you press the same button again, the application stops and transitions back to the STOP state.

The other way to interact with the demo mode is to use the FreeMASTER tool. The FreeMASTER application consists of two parts: the PC application used for variable visualization and the set of software drivers running in the embedded application. The serial interface transfers data between the PC and the embedded application. This interface is provided by the debugger included in the boards.

The application can be controlled using the following two interfaces:

- The user button on the development board (controlling the demo mode):
  - FRDM-MCXA346 - SW2
- Remote control using FreeMASTER (Section 8):
  - Setting a variable in the FreeMASTER variable watch. See Section 8.5.

Identify all motor parameters if you are using your own motor (different from the default motors). The automated parameter identification is described in the following sections.

8 Remote control using FreeMASTER

This section provides information about the tools and recommended procedures to control the sensor/sensorless PMSM Field-Oriented Control (FOC) application using FreeMASTER. The application contains the embedded-side driver of the FreeMASTER real-time debug monitor and data visualization tool for communication with the PC. It supports non-intrusive monitoring, as well as the modification of target variables in real time, which is very useful for the algorithm tuning. Besides the target-side driver, the FreeMASTER tool requires the installation of the PC application as well. You can download the latest version of FreeMASTER at the FreeMASTER webpage. To run the FreeMASTER application including the MCAT tool, double-click

the **pmsm_float_enc.pmpx** file located in the middleware\motor_control\freemaster folder. The FreeMASTER application starts and the environment is created automatically, as defined in the *.pmpx file.

Note: *In MCUXpresso, the FreeMASTER application can run directly from an IDE in the motor_control/freemaster folder.*

8.1 Establishing FreeMASTER communication

The remote operation is provided by FreeMASTER via the USB interface. To control a PMSM motor using FreeMASTER, perform the following steps:

1. Download the project from your chosen IDE to the MCU and run it.
2. Open the pmsm_float_enc.pmpx FreeMASTER project. The PMSM project uses the TSA by default, so it is not necessary to select a symbol file for FreeMASTER.
3. To establish the communication, click the communication button (the green "GO" button in the top left-hand corner).

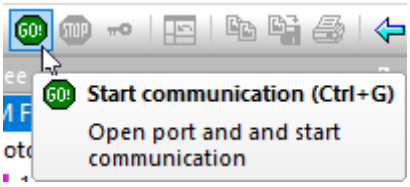


Figure 10. Green "GO" button in top left-hand corner

4. If the communication is established successfully, the FreeMASTER communication status in the bottom right-hand corner changes from "Not connected" to "RS-232 UART Communication; COMxx; speed=115200". Otherwise, the FreeMASTER warning pop-up window appears.

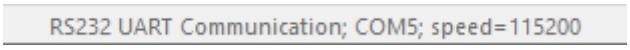


Figure 11. FreeMASTER - communication is established successfully

5. To reload the MCAT HTML page and check the App ID, press F5.
6. Control the PMSM motor by writing to a control variable in the variable watch.
7. If you rebuild and download the new code to the target, turn the FreeMASTER application off and on.

If the communication is not established successfully, perform the following steps:

1. Go to the **Project** > **Options** > **Comm** tab and make sure that the correct COM port is selected and the communication speed is set to 115200 bit/s.

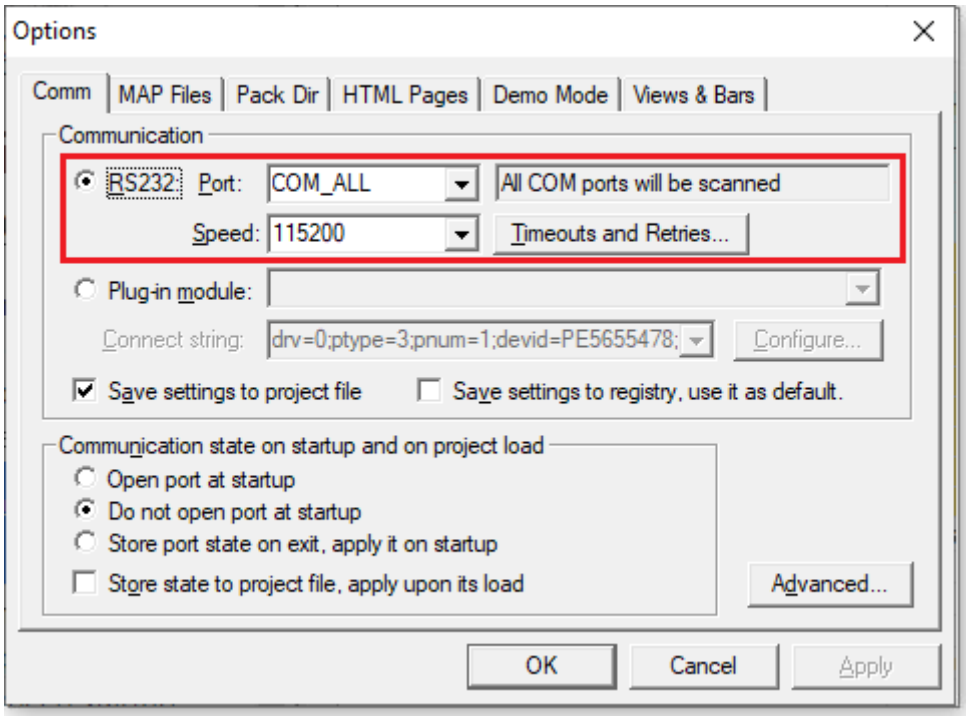


Figure 12. FreeMASTER communication setup window

2. Ensure that your computer is communicating with the plugged board. Unplug and then plug in the USB cable and reopen the FreeMASTER project.

8.2 TSA replacement with ELF file

The FreeMASTER project for the motor-control example uses the Target-Side Addressing (TSA) information about variable objects and types to be retrieved from the target application by default. With the TSA feature, you can describe the data types and variables directly in the application source code and make this information available to the FreeMASTER tool. The tool can then use this information instead of reading symbol data from the application ELF/Dwarf executable file.

FreeMASTER reads the TSA tables and uses the information automatically when an MCU board is connected. A great benefit of using the TSA is that there are no issues with the correct path to the ELF/Dwarf file. The variables described by the TSA tables may be read-only, so even if FreeMASTER attempts to write the variable, the target MCU side denies the value. The variables not described by any TSA tables may also become invisible and protected, even for read-only access.

The use of TSA means more memory requirements for the target. If you do not want to use the TSA feature, you must modify the example code and FreeMASTER project.

To modify the example code, perform the following steps:

1. Open the motor-control project and rewrite the `FMSTR_USE_TSA` macro from 1 to 0 in the `freemaster_cfg.h` file.
2. Build, download, and run the motor-control project.
3. Open the FreeMASTER project and click **Project** > **Options** (or press Ctrl+T).
4. Click the **MAP Files** tab and find the default symbol file (ELF/Dwarf executable file) located in the IDE output folder.

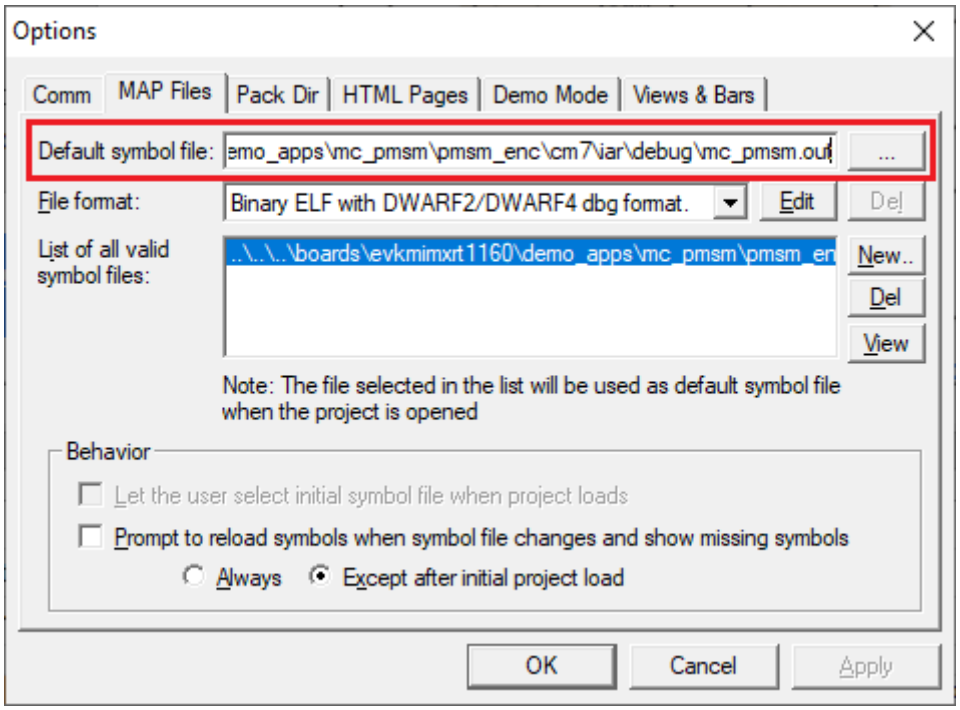


Figure 13. Default symbol file

5. Click **OK** and restart the FreeMASTER communication.

For more information, see *FreeMASTER for Embedded Applications* (document FMSTERUG).

8.3 Motor Control Aplication Tuning (MCAT) interface

The PMSM sensor/sensorless FOC application can be easily controlled and tuned using the Motor Control Application Tuning (MCAT) plug-in for PMSM. The MCAT for PMSM is a user-friendly page, which runs within FreeMASTER. The tool consists of the tab menu and workspace, as shown in Figure 14. Each tab from the tab menu (4) represents one submodule, which enables tuning or controlling different application aspects. Besides the MCAT page for PMSM, several scopes, recorders, and variables in the project tree (5) are predefined in the FreeMASTER project file to further simplify the motor parameter tuning and debugging.

When the FreeMASTER is not connected to the target, the "Board found" line (2) shows "Board ID not found". When the communication with the target MCU is established, the "Board found" line is read from the Board ID variable watch and displayed. If the connection is established and the board ID is not shown, press F5 to reload the MCAT HTML page.

There are three action buttons in MCAT (3):

- **Load data** - MCAT input fields (for example, motor parameters) are loaded from the `mX_pmsm_appconfig.h` file (JSON formatted comments). Only existing `mX_pmsm_appconfig.h` files can be selected for loading. The loaded `mX_pmsm_appcofig.h` file is displayed in a grey field (7).
- **Save data** - MCAT input fields (JSON formatted comments) and output macros are saved to the `mX_pmsm_appconfig.h` file. Up to nine files (`m1-9_pmsm_appconfig.h`) can be selected. A pop-up window with the user motor ID and description appears when a different `mX_pmsm_appcofig.h` file is selected. The motor ID and description are also saved in `mX_pmsm_appcofig.h` as a JSON comment. The embedded code includes `m1_pmsm_appcofig.h` only in the single-motor-control application. Therefore, saving to higher indexed `mX_pmsm_appconfig.h` files has no effect at the compilation stage.
- **Update target** - writes the MCAT-calculated tuning parameters to FreeMASTER variables, which effectively updates the values on the target MCU. These tuning parameters are updated in the MCU RAM. To write these

tuning parameters to the MCU flash memory, `m1_pmsm_appcofig.h` must be saved, code recompiled, and downloaded to the MCU.

Every time the communication with the board is established or MCAT is refreshed by pressing F5, MCAT looks for the `m1_pmsm_appconfig.h` file. A path to `m1_pmsm_appconfig` is relative to the *.pmpx file. The path is composed from the fixed defined folder name and from the following FreeMASTER variables: *User Path 1, User Path 2, Board ID, Example ID*. The *User Path 1* and *User Path 2* variables are intended for a custom path to the `m1_pmsm_appconfig.h` file. These variables are defined in the `main.c` example file, where you can modify them. There are several possible paths with a different priority (higher to lower) and a typical use case:

1. *User Path 1/*
2. *User Path 2/*
3. ../boards/*Board ID*/mc_pmsm/*Example ID/*
   - typical use case during the SDK example development
4. ../../../boards/*Board ID*/demo_apps/mc_pmsm/*Example ID/*
   - typical use case in the SDK package
5. ../../../boards/*Board ID*/demo_apps/mc_pmsm/*Example ID*/cm7/
   - typical use case in the SDK package
6. ../../../boards/*Board ID*/demo_apps/mc_pmsm/*Example ID*/cm33_core0/
   - typical use case in the SDK package
7. ../../source/
   - typical use case in the workspace

When the `m1_pmsm_appconfig.h` file is found at one of the above-defined paths, it is loaded into the MCAT. Then, MCAT uses this directory for further operations (load data, save data). When the `m1_pmsm_appconfig.h` file is not found even at the paths above, the default `m1_pmsm_appconfig.h` file is loaded. The default `m1_pmsm_appconfig.h` file is located in the `mcat` folder. When the default file is loaded to the MCAT, the newly saved `mX_pmsm_appconfig.h` files are placed next to the *.pmpx file.

Note: *Since the path to the `mX_pmsm_appconfig.h` file is composed from User Path 1, User Path 2, Board ID, and Example ID, FreeMASTER must be connected to the target. The FreeMASTER variable values are read before using the Save/Load buttons.*

Note: *Only the **Update target** button updates the values on the target in real time. The Load/Save buttons operate with the `mX_pmsm_appcofig.h` file only.*

Note: *The MCAT may require Internet connection. If no Internet connection is available, the CSS and icons may not be loaded properly.*

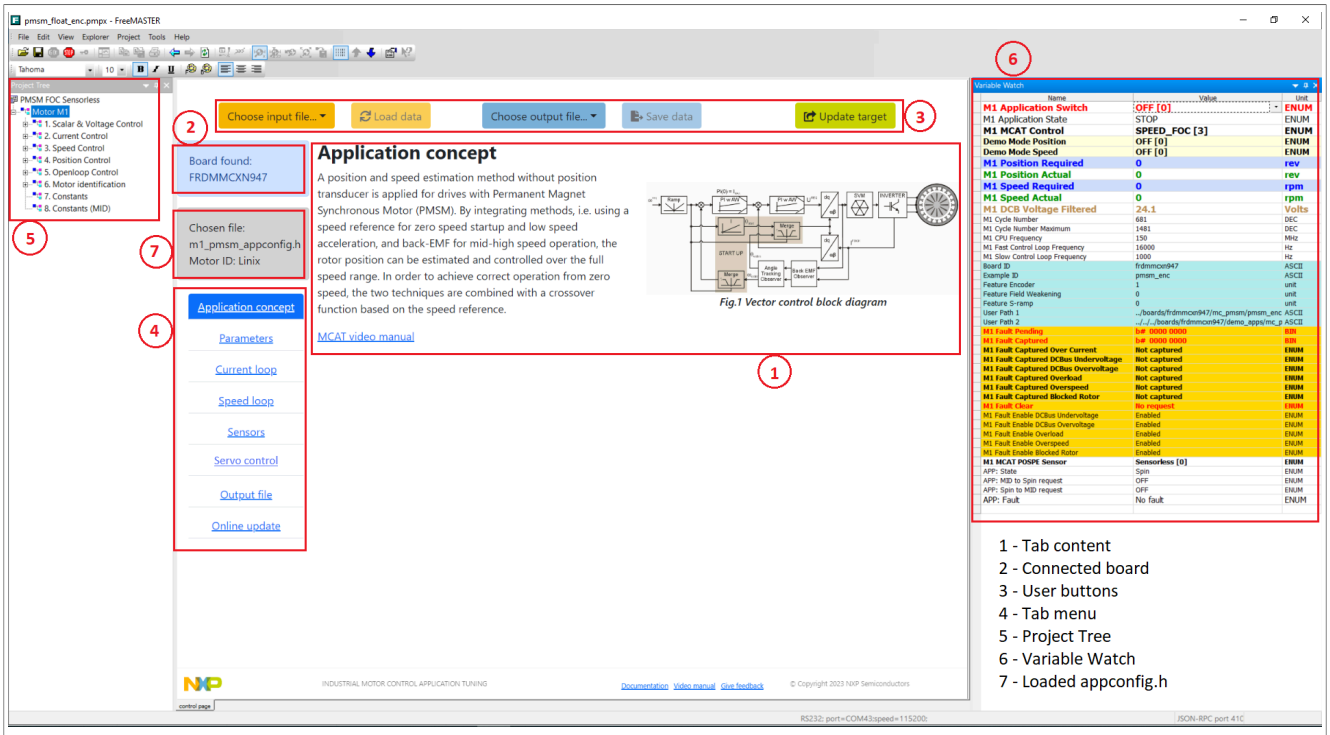


Figure 14. FreeMASTER and MCAT layout

In the default configuration, the following tabs (4) are available:

- Application concept: welcome page with the PMSM sensor/sensorless FOC diagram and a short application description.
- Motor parameters identification (MID): this page is used to control the motor parameters identification.
- Parameters: this page enables you to modify the motor parameters, hardware and application scales specification, alignment, and fault limits.
- Current loop: current loop PI controller gains and output limits.
- Speed loop: this tab contains fields for the specification of the speed controller proportional and integral gains, as well as the output limits and parameters of the speed ramp.
- Servo control: this tab contains fields for the specification of the position controller proportional gain, as well as the output limits.
- Sensorless: this page enables you to tune the parameters of the BEMF observer, tracking observer, and open-loop startup.
- Output file: this tab shows all the calculated constants that are required by the PMSM sensored/sensorless FOC application. It is also possible to generate the `mX_pmsm_appconfig.h` file, which is then used to preset all application parameters permanently at the project rebuild.
- Online update: this tab shows the actual values of variables on the target and new calculated values, which can be used to update the target variables.

Every subblock in the FreeMASTER project tree (5) has several variables defined in the variable watch (6).

The following sections provide simple instructions on how to identify the parameters of a connected PMSM motor and how to tune the application appropriately.

8.3.1 MCAT tabs description

This chapter describes the MCAT input parameters and equations used to calculate the MCAT output (generated) parameters. In the default configuration, the tabs described below are available. Some tabs

may be missing if they are not supported in the embedded code. There are general constants used in MCAT calculations, listed in the following table:

Table 8. Constants used in equations

Constant	Value	Unit
UmaxCoeff	1.73205	-
DiscMethodFactor	2	-
k_factor	100	-
pi	3.1416	-

8.3.1.1 Application concept

This tab is a welcome page with the PMSM sensored/sensorless FOC diagram and a short description of the application.

8.3.1.2 Motor parameters identification (MID)

This tab is used to control the MID (pole-pair assistant, electrical and mechanical parameters identification). The input measurement parameters and the estimated motor parameters are available in this tab. The buttons triggering the identification are in this tab as well. To start the selected measurement, click the "Start" button. You are notified about the identification progress via status messages. See Section 8.8.1 for more information.

Table 9. MID tab inputs

MCAT group	MCAT name	Equation name	Description	Unit
Motor parameters for measurements	I Nom	midInParamINom	Nominal motor current.	[A]
	N Nom	midInParamNNom	Nominal motor speed.	[RPM]
Pole-pair identification assistant	Pp	midPolePairIAPp	The motor number of the pole-pairs number determined by the user.	[-]

8.3.1.3 Parameters

This tab enables the modification of motor parameters, specification of hardware and application scales, alignment, and fault limits. All inputs are described in the following table. The MCAT group and MCAT name help to locate a parameter in the MCAT layout. The equation name represents the input parameter in the equations below.

Table 10. Parameters tab inputs

MCAT group	MCAT name	Equation name	Description	Unit
Motor parameters	PP	parametersPp	Motor number of pole-pairs. You can obtain it from the motor manufacturer or use the pole-pair assistant to determine and then fill them manually.	-
	Rs	parametersRs	Stator phase resistance. You can obtain it from the motor manufacturer or use the electrical parameters	[Ω]

Table 10. Parameters tab inputs*...continued*

MCAT group	MCAT name	Equation name	Description	Unit
			identification and then fill them manually.	
	Ld	parametersLd	Stator direct inductance. You can obtain it from the motor manufacturer or use the electrical parameters identification and then fill them manually.	[H]
	Lq	parametersLq	Stator quadrature inductance. You can obtain it from the motor manufacturer or use the electrical parameters identification and then fill them manually.	[H]
	Kt	parametersKt	Motor torque constant. You can obtain it from the motor manufacturer or use the Kt identification and then fill them manually.	[Nm/A]
	J	parametersJ	Drive inertia (motor + plant). Use the mechanical identification and then fill it manually.	[kg.m2]
	Iph nom	parametersIphNom	Nominal motor current. You can obtain it from the motor manufacturer.	[A]
	Uph nom	parametersUphNom	Nominal motor voltage. You can obtain it from the motor manufacturer.	[V]
	N nom	parametersNnom	Nominal motor speed. You can obtain it from the motor manufacturer.	[RPM]
Hardware scales	I max	parametersImax	Current-sensing HW scale. Keep it as it is in case of standard NXP HW or recalculate it according to your own schematic.	[A]
	U DCB max	parametersUdcbMax	DC-bus voltage-sensing HW scale. Keep it as it is in case of standard NXP HW or recalculate it according to your own schematic.	[V]
Fault limits	U DCB trip	parametersUdcbTrip	DC-bus braking resistor threshold. The braking resistor's transistor is turned on when the DC-bus voltage exceeds this threshold.	[V]
	U DCB under	parametersUdcbUnder	DC-bus undervoltage fault threshold.	[V]
	U DCB over	parametersUdcbOver	DC-bus overvoltage fault threshold.	[V]

Table 10. Parameters tab inputs*...continued*

MCAT group	MCAT name	Equation name	Description	Unit
	N over	parametersNover	Overspeed fault threshold.	[RPM]
	N min	parametersNmin	Minimal closed-loop speed. When the required speed ramps down under this threshold, the motor-control state machine goes to a freewheel state, where the top and bottom transistors are turned off and the motor speeds down freely. It applies only for sensorless operation.	[RPM]
	E block	parametersEblock	Blocked rotor detection. When the BEMF voltage drops under the *E block* threshold for more than *E block per* (fast-loop ticks), the blocked rotor fault is detected.	[V]
	E block per	parametersEblockPer		-
Application scales	N max	parametersNmax	Application speed scale. Keep it around 10 % of the margin above *N over*.	[RPM]
	U DCB IIR F0	parametersUdcbIIRf0	Cut-off frequency of the DC-bus IIR filter.	[Hz]
	Calibration duration	parametersCalibDuration	ADC (phase current offset) calibration duration. It is done every time when transitioning from STOP to RUN.	[sec]
	Fault duration	parametersFaultDuration	After a fault condition disappears, wait for the defined time to clear the pending faults bitfield and transition to the STOP state.	[sec]
	Freewheel duration	parametersFreewheelDuration	Freewheel state duration. The freewheel state is entered when the ramped speed drops under *N min*.	[sec]
	Scalar Uq min	parametersScalarUqMin	Scalar control voltage minimal value.	[V]
	Scalar V/Hz factor ratio	parametersScalarVHzRatio	Scalar V/Hz ratio gain.	[%]
Alignment	Align voltage	parametersAlignVoltage	Motor alignment voltage.	[V]
	Align duration	parametersAlignDuration	Motor alignment duration.	[sec]

The output equations are as follows (applies for saving to `mX_pmsm_appconfig.h` and also for updating a corresponding FreeMASTER variable):

- `M1_U_MAX` = parametersUdcbMax / UmaxCoeff
- `M1_MOTOR_PP` = parametersPp
- `M1_I_PH_NOM` = parametersIphNom
- `M1_I_MAX` = parametersImax
- `M1_U_DCB_MAX` = parametersUdcbMax

- M1_U_DCB_TRIP = parametersUdcbTrip
- M1_U_DCB_UNDERVOLTAGE = parametersUdcbUnder
- M1_U_DCB_OVERVOLTAGE = parametersUdcbOver
- M1_FREQ_MAX = parametersNmax / 60 * parametersPp
- M1_ALIGN_DURATION = parametersAlignDuration / speedLoopSampleTime
- M1_CALIB_DURATION = parametersCalibDuration / speedLoopSampleTime
- M1_FAULT_DURATION = parametersFaultDuration / speedLoopSampleTime
- M1_FREEWHEEL_DURATION = parametersFreewheelDuration / speedLoopSampleTime
- M1_E_BLOCK_TRH = parametersEblock
- M1_E_BLOCK_PER = parametersEblockPer
- M1_N_MIN = parametersNmin / 60 * (parametersPp * 2 * pi)
- M1_N_MAX = parametersNmax / 60 * (parametersPp * 2 * pi)
- M1_N_ANGULAR_MAX = (60 / (parametersPp * 2 * pi))
- M1_N_NOM = parametersNnom / 60 * (parametersPp * 2 * pi)
- M1_N_OVERSPEED = parametersNover / 60 * (parametersPp * 2 * pi)
- M1_UDCB_IIR_B0 = (2 * pi * parametersUdcbIIRf0 * currentLoopSampleTime) / (2 + (2 * pi * parametersUdcbIIRf0 * currentLoopSampleTime))
- M1_UDCB_IIR_B1 = (2 * pi * parametersUdcbIIRf0 * currentLoopSampleTime) / (2 + (2 * pi * parametersUdcbIIRf0 * currentLoopSampleTime))
- M1_UDCB_IIR_A1 = -(2 * pi * parametersUdcbIIRf0 * currentLoopSampleTime - 2) / (2 + (2 * pi * parametersUdcbIIRf0 * currentLoopSampleTime))
- M1_SCALAR_UQ_MIN = parametersScalarUqMin
- M1_ALIGN_VOLTAGE = parametersAlignVoltage
- M1_SCALAR_VHZ_FACTOR_GAIN = parametersUphNom * parametersScalarVHzRatio / 100 / (parametersNnom * parametersPp / 60)
- M1_SCALAR_INTEG_GAIN = 2 * pi * parametersPp * parametersNmax / 60 * currentLoopSampleTime / pi
- M1_SCALAR_RAMP_UP = speedLoopIncUp * currentLoopSampleTime / 60 * parametersPp
- M1_SCALAR_RAMP_DOWN = speedLoopIncDown * currentLoopSampleTime / 60 * parametersPp

8.3.1.4 Current loop

This tab enables the current loop PI controller gains and output limits tuning. All inputs are described in the following table. The MCAT group and MCAT name help to locate the parameter in the MCAT layout. The equation name represents the input parameter in the equations below.

Table 11. Current loop tab input

MCAT group	MCAT name	Equation name	Description	Unit
Loop parameters	Sample time	currentLoopSampleTime	Fast-control-loop period. This disabled value is read from the target via FreeMASTER because the application timing is set in the embedded code by the peripherals setting. This value is accessible only if the target is not connected and the value cannot be obtained from the target.	[sec]
	F0	currentLoopF0	Current controller bandwidth.	[Hz]
	ξ	currentLoopKsi	Current controller attenuation.	-

Table 11. Current loop tab input*...continued*

MCAT group	MCAT name	Equation name	Description	Unit
Current PI controller limits	Output limit	currentLoopOutputLimit	Current controller output voltage limit equals the duty cycle limit. Be careful when setting this limit above 95 %, because it affects the current sensing. Some minimal bottom-transistors on time is required.	[%]

The output equations are as follows (applies for saving to `mX_pmsm_appconfig.h` and also for updating a corresponding FreeMASTER variable):

- `M1_CLOOP_LIMIT` = currentLoopOutputLimit / UmaxCoeff / 100
- `M1_D_KP_GAIN` = (2 * currentLoopKsi * 2 * pi * currentLoopF0 * parametersLd) - parametersRs
- `M1_D_KI_GAIN` = (2 * pi * currentLoopF0)^2 * parametersLd * currentLoopSampleTime / DiscMethodFactor
- `M1_Q_KP_GAIN` = (2 * currentLoopKsi * 2 * pi * currentLoopF0 * parametersLq) - parametersRs
- `M1_Q_KI_GAIN` = (2 * pi * currentLoopF0)^2 * parametersLq * currentLoopSampleTime / DiscMethodFactor
- kp_q = (4 * currentLoopKsi * pi * currentLoopF0 * parametersLq - parametersRs)
- ki_q = (parametersLq * (2 * pi * currentLoopF0)^2)
- Ti = kp_q / ki_q
- `M1_Q_IIR_ZC_B0` = currentLoopSampleTime / (currentLoopSampleTime + 2 * Ti)
- `M1_Q_IIR_ZC_B1` = currentLoopSampleTime / (currentLoopSampleTime + 2 * Ti)
- `M1_Q_IIR_ZC_A1` = - (currentLoopSampleTime - 2 * Ti) / (currentLoopSampleTime + 2 * Ti)

8.3.1.5 Speed loop

This tab enables the speed-loop PI controller gains and output limits tuning and the required speed-ramp parameters and feedback-speed filter tuning. The MCAT group and MCAT name help to locate the parameter in the MCAT layout. The equation name represents the input parameter in the equations below.

Table 12. Speed loop tab input

MCAT group	MCAT name	Equation name	Description	Unit
Loop parameters	Sample time Ts	speedLoopSampleTime	Slow-control-loop period. This disabled value is read from the target via FreeMASTER, because the application timing is set in the embedded code by the peripherals setting. This value is accessible only if the target is not connected and the value cannot be obtained from the target.	[sec]
	Bandwidth F0	speedLoopF0	Speed controller bandwidth.	[Hz]
	Attenuation ξ	speedLoopKsi	Speed controller attenuation.	-
Speed ramp	Inc up	speedLoopIncUp	Required speed maximal acceleration.	[RPM/sec]
	Inc down	speedLoopIncDown	Required speed maximal acceleration.	[RPM/sec]

Table 12. Speed loop tab input*...continued*

MCAT group	MCAT name	Equation name	Description	Unit
Actual speed filter	Cut-off freq	speedLoopCutOffFreq	Speed feedback (before entering the PI subtraction) filter bandwidth.	[Hz]
Speed PI controller limits	Upper limit	speedLoopUpperLimit	Maximal required Q-axis current (speed controller output). The Q-axis current limitation equals the motor torque limitation.	[A]
	Lower limit	speedLoopLowerLimit	Minimal required Q-axis current (speed controller output). The Q-axis current limitation equals the motor torque limitation.	[A]

The output equations are as follows (applies for saving to `mX_pmsm_appconfig.h` and also for updating a corresponding FreeMASTER variable):

- `M1_SPEED_PI_PROP_GAIN` = (4 * speedLoopKsi * pi * speedLoopF0 * parametersJ) / (parametersKt * parametersPp))
- `M1_SPEED_PI_INTEG_GAIN` = ((2 * pi * speedLoopF0) * (2 * pi * speedLoopF0) * parametersJ) / (parametersKt * parametersPp) * (speedLoopSampleTime / DiscMethodFactor))
- `M1_SPEED_RAMP_UP` = (speedLoopIncUp * speedLoopSampleTime / (60 / (parametersPp * 2 * pi)))
- `M1_SPEED_RAMP_DOWN` = (speedLoopIncDown * speedLoopSampleTime / (60 / (parametersPp * 2 * pi)))
- `M1_SPEED_IIR_B0`= (2 * pi * speedLoopCutOffFreq * speedLoopSampleTime) / (2 + (2 * pi * speedLoopCutOffFreq * speedLoopSampleTime))
- `M1_SPEED_IIR_B1` = (2 * pi * speedLoopCutOffFreq * speedLoopSampleTime) / (2 + (2 * pi * speedLoopCutOffFreq * speedLoopSampleTime))
- `M1_SPEED_IIR_A1` = -(2 * pi * speedLoopCutOffFreq * speedLoopSampleTime - 2) / (2 + (2 * pi * speedLoopCutOffFreq * speedLoopSampleTime))
- `M1_SPEED_LOOP_HIGH_LIMIT` = speedLoopUpperLimit
- `M1_SPEED_LOOP_LOW_LIMIT` = speedLoopLowerLimit
- Tfilt = `M1_SPEED_PI_PROP_GAIN` / (`M1_SPEED_PI_INTEG_GAIN` * (2 / speedLoopSampleTime))
- `M1_SPEED_IIR_ZC_B0` = (speedLoopSampleTime / (speedLoopSampleTime + 2 * Tfilt))
- `M1_SPEED_IIR_ZC_B1` = (speedLoopSampleTime / (speedLoopSampleTime + 2 * Tfilt))
- `M1_SPEED_IIR_ZC_A1` = (-1) * (speedLoopSampleTime - 2 * Tfilt) / (speedLoopSampleTime + 2 * Tfilt)

8.3.1.6 Servo control

This tab enables the position loop P/PI controller gains and output limits tuning. The MCAT group and MCAT name help to locate the parameter in the MCAT layout. The equation name represents the input parameter in the equations below.

Table 13. Servo control tab input

MCAT group	MCAT name	Equation name	Description	Unit
Servo control parameters	Sample time Ts	positionLoopSampleTime	Slow-control-loop period. This disabled value is read from the target via FreeMASTER because the application timing is set in the embedded code by the peripherals setting. This value is accessible only if the target is not connected and the value	[sec]

Table 13. Servo control tab input*...continued*

MCAT group	MCAT name	Equation name	Description	Unit
			cannot be obtained from the target.	
	Bandwidth F0	positionLoopF0	Servo controller bandwidth.	[Hz]
	Attenuation ξ	positionLoopKsi	Servo controller attenuation.	-
Position loop - P controller limits	Upper limit	positionPControllerHighLimit	Maximal required speed (position controller output).	[RPM/sec]
	Lower limit	positionPControllerLowLimit	Minimal required speed (position controller output).	[RPM/sec]
Speed loop - PI controller limits	Upper limit	servo_speedLoopUpperLimit	Maximal required Q-axis current (speed controller output). The Q-axis current limitation equals the motor torque limitation.	[A]
	Lower limit	servo_speedLoopLowerLimit	Minimal required Q-axis current (speed controller output). The Q-axis current limitation equals the motor torque limitation.	[A]

Output equations (apply for saving to `mX_pmsm_appconfig.h` and also for updating a corresponding FreeMASTER variable):

- k10 = 1 / (2 * pi * parametersPp)
- `M1_SERVO_POSITION_P_PROP_GAIN` = (2 * pi * positionLoopF0) / (2 * positionLoopKsi + 1) * k10
- `M1_SERVO_POSITION_P_HIGH_LIMIT` = (2 * pi * positionPControllerHighLimit * parametersPp / 60.0)
- `M1_SERVO_POSITION_P_LOW_LIMIT` = (2 * pi * positionPControllerLowLimit * parametersPp / 60.0)
- `M1_SERVO_FEED_FRWD_K1` = (2 * positionLoopKsi * `M1_SERVO_POSITION_P_PROP_GAIN` / (2 * pi * positionLoopF0))
- `M1_SERVO_FEED_FRWD_K2` = (`M1_SERVO_POSITION_P_PROP_GAIN` / ((2 * pi * positionLoopF0) * (2 * pi * positionLoopF0)))
- Tfilt = (2 * positionLoopKsi + 1) / ((2 * positionLoopKsi + 1) * 2 * pi * positionLoopF0)
- `M1_SERVO_IIR_ZC_B0` = (positionLoopSampleTime / (positionLoopSampleTime + 2 * Tfilt))
- `M1_SERVO_IIR_ZC_B1` = (positionLoopSampleTime / (positionLoopSampleTime + 2 * Tfilt))
- `M1_SERVO_IIR_ZC_A1` = (-1) * (positionLoopSampleTime - 2 * Tfilt) / (positionLoopSampleTime + 2 * Tfilt)
- `M1_SERVO_SPEED_PI_PROP_GAIN` = (((2 * positionLoopKsi + 1) * (2 * pi * positionLoopF0) * parametersJ) / (parametersKt * parametersPp))
- `M1_SERVO_SPEED_PI_INTEG_GAIN` = (((parametersJ * (2 * positionLoopKsi + 1) * (2 * pi * positionLoopF0) * (2 * pi * positionLoopF0)) / (parametersKt * parametersPp)) * (positionLoopSampleTime / DiscMethodFactor)
- `M1_SERVO_SPEED_PI_HIGH_LIMIT` = servo_speedLoopUpperLimit
- `M1_SERVO_SPEED_PI_LOW_LIMIT` = servo_speedLoopLowerLimit

8.3.1.7 Sensorless

This tab enables the BEMF observer and tracking observer parameters tuning and open-loop startup tuning. The MCAT group and the MCAT name help to locate the parameter in the MCAT layout. The equation name represents the input parameter in the equations below.

Table 14. Sensorless tab input

MCAT group	MCAT name	Equation name	Description	Unit
BEMF observer parameters	F0	sensorlessBemfObsrvF0	BEMF observer bandwidth.	[Hz]
	ξ	sensorlessBemfObsrvKsi	BEMF observer attenuation.	-
Tracking observer parameters	F0	sensorlessTrackObsrvF0	Tracking observer bandwidth.	[Hz]
	ξ	sensorlessTrackObsrvKsi	Tracking observer attenuation.	-
Open-loop startup parameters	Startup ramp	sensorlessStartupRamp	Open-loop startup ramp.	[RPM/sec]
	Startup current	sensorlessStartupCurrent	Open-loop startup current.	[A]
	Merging Speed	sensorlessMergingSpeed	Merging speed.	[RPM]
	Merging Coefficient	sensorlessMergingCoeff	Merging coefficient (100 % = merging is done within one electrical revolution).	[%]

The output equations are as follows (applies for saving to `mX_pmsm_appconfig.h` and also for updating a corresponding FreeMASTER variable):

- `M1_I_SCALE` = (parametersLd / (parametersLd + currentLoopSampleTime * parametersRs))
- `M1_U_SCALE` = (currentLoopSampleTime / (parametersLd + currentLoopSampleTime * parametersRs))
- `M1_E_SCALE` = (currentLoopSampleTime / (parametersLd + currentLoopSampleTime * parametersRs))
- `M1_WI_SCALE` = (parametersLq * currentLoopSampleTime / (parametersLd + currentLoopSampleTime * parametersRs))
- `M1_BEMF_DQ_KP_GAIN` = ((2 * sensorlessBemfObsrvKsi * 2 * pi * sensorlessBemfObsrvF0 * parametersLd - parametersRs))
- `M1_BEMF_DQ_KI_GAIN` = (parametersLd * (2 * pi * sensorlessBemfObsrvF0)^2 * currentLoopSampleTime)
- `M1_TO_KP_GAIN` = 2 * sensorlessTrackObsrvKsi * 2 * pi * sensorlessTrackObsrvF0
- `M1_TO_KI_GAIN` = ((2 * pi * sensorlessTrackObsrvF0)^2) * currentLoopSampleTime
- `M1_TO_THETA_GAIN` = (currentLoopSampleTime / pi)
- `M1_OL_START_RAMP_INC` = (sensorlessStartupRamp * currentLoopSampleTime / (60 / (parametersPp * 2 * pi)))
- `M1_OL_START_I` = sensorlessStartupCurrent
- `M1_MERG_SPEED_TRH` = (sensorlessMergingSpeed / (60 / (parametersPp * 2 * pi)))
- `M1_MERG_COEFF` = ((sensorlessMergingCoeff / 100) * sensorlessMergingSpeed * parametersPp * currentLoopSampleTime) / 60
- `TO_IIR_cutoff_freq` = 1 / (2 * speedLoopSampleTime) * 0.8
- `M1_TO_SPEED_IIR_B0` = (2 * pi * TO_IIR_cutoff_freq * currentLoopSampleTime) / (2 + (2 * pi * TO_IIR_cutoff_freq * currentLoopSampleTime))
- `M1_TO_SPEED_IIR_B1` = (2 * pi * TO_IIR_cutoff_freq * currentLoopSampleTime) / (2 + (2 * pi * TO_IIR_cutoff_freq * currentLoopSampleTime))
- `M1_TO_SPEED_IIR_A1` = -(2 * pi * TO_IIR_cutoff_freq * currentLoopSampleTime - 2) / (2 + (2 * pi * TO_IIR_cutoff_freq * currentLoopSampleTime))

8.4 Aplication tuning in preprocessor

If you do not want to use the MCAT tool for control parameter calculation, the parameters can be computed using the preprocessor. The preprocessor uses the provided motor data (resistance, inductance, number of poles, nominal speed) and, based on the implemented algorithm, calculates the optimal control values. To use the preprocessor-based calculation, copy the contents of the `m1_pmsm_preprocessor.h` file into

`m1_pmsm_appconfig.h` and set the basic parameters. The `m1_pmsm_preprocessor.h` file contains the equations described in Section 8.3.1.

Note: *m1_pmsm_preprocessor is currently available only for floating-point applications.*

8.5 Motor control modes - how to run a motor

In the "Project Tree" option, you can choose between the scalar and FOC control using the appropriate FreeMASTER tabs. The FreeMASTER variables can control the application, corresponding to the control structure selected in the FreeMASTER project tree. This is useful for application tuning and debugging. The required control structure must be selected in the "M1 MCAT Control" variable. To turn the application on or off, use the "M1 Application Switch" variable. Setting/clearing the "M1 Application Switch" variable also enables/disables all PWM channels.

Before the motor starts, complete the following conditions:

1. Connect the power supply to the inverter with a correct voltage value.
2. If you want to use sensored control (encoder feedback), connect the encoder to the inverter.
3. No pending fault. Check the "M1 Fault Pending" variable in the "Motor M1" project tree subblock. If there is a value, remove the cause of the fault or disable fault checking (for example, in the "M1 Fault Enable Blocked Rotor" variable).

8.5.1 Scalar control

The following figure shows the scalar control diagram. It is the simplest type of motor-control techniques. The ratio between the magnitude of the stator voltage and the frequency must be kept at the nominal value. Therefore, the control method is sometimes called Volt per Hertz (or V/Hz). The position-estimation BEMF observer and tracking observer algorithms run in the background, even if the estimated position information is not used directly. This is useful for the BEMF observer tuning. For more information, see *Sensorless PMSM Field-Oriented Control* (document DRM148).

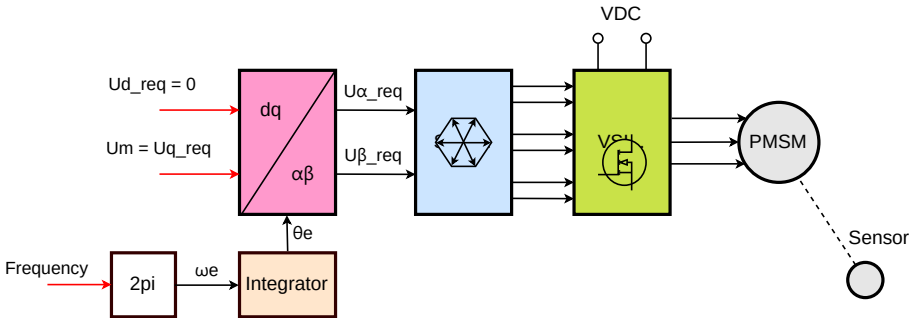


Figure 15. Scalar control mode

To run the motor in scalar control, follow these steps:

1. Switch the project tree subblock to the "Scalar & Voltage Control" option.
2. Switch the "M1 MCAT Control" variable to "SCALAR_CONTROL".
3. Set the required frequency (for example, 20 Hz) in the "M1 Scalar Freq Required" variable.
4. Set the "M1 Application Switch" variable to "1". The motor starts spinning.
5. Observe the motor speed, position, phase currents, and other graphs predefined in the subblock scopes and recorders.

8.5.2 Open-loop control mode

The open-loop mode (shown in the figure below) is similar to the scalar control mode in function. However, it provides more flexibility in specifying the required parameters. This mode allows you to set specific angle and frequency, according to the following equation:

$$\theta_{el} = \theta_{init} + \int_{t_0}^{t} 2\pi f \, dt \tag{4}$$

Besides setting the voltages in the D and Q axes, when using this mode, you can also enable the current controllers and specify the required currents for the D and Q axes. Therefore, you can use this function for the current controller parameter tuning. The current controllers cannot be enabled/disabled in the SPIN state. Turn the application switch off before enabling/disabling the current controllers.

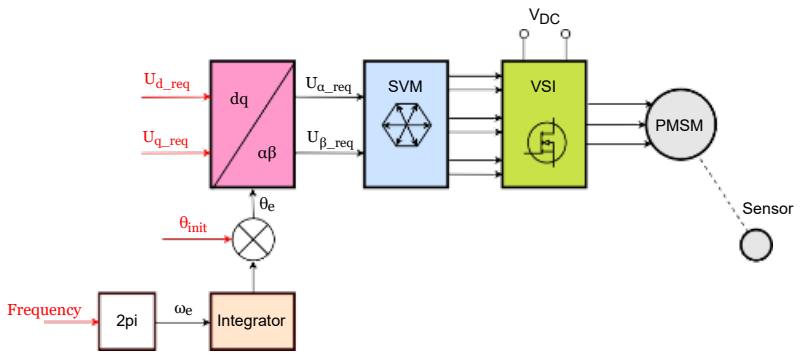


Figure 16. Voltage - open-loop control

To run the motor in voltage - open-loop control, follow these steps:

1. Switch the project tree subblock to "Openloop Control".
2. Switch the "M1 MCAT Control" variable to "OPEN_LOOP".
3. Set the required values in the "M1 Openloop Required Ud" and "M1 Openloop Required Uq" variables.
4. Set the required initial position in the "M1 Openloop Theta Electrical" variable.
5. Set the required frequency in the "M1 Openloop Required Frequency Electrical" variable.
6. Set the "M1 Application Switch" variable to "1". The motor starts spinning.
7. Observe the motor speed, position, phase currents, and other graphs predefined in the subblock scopes and recorders.

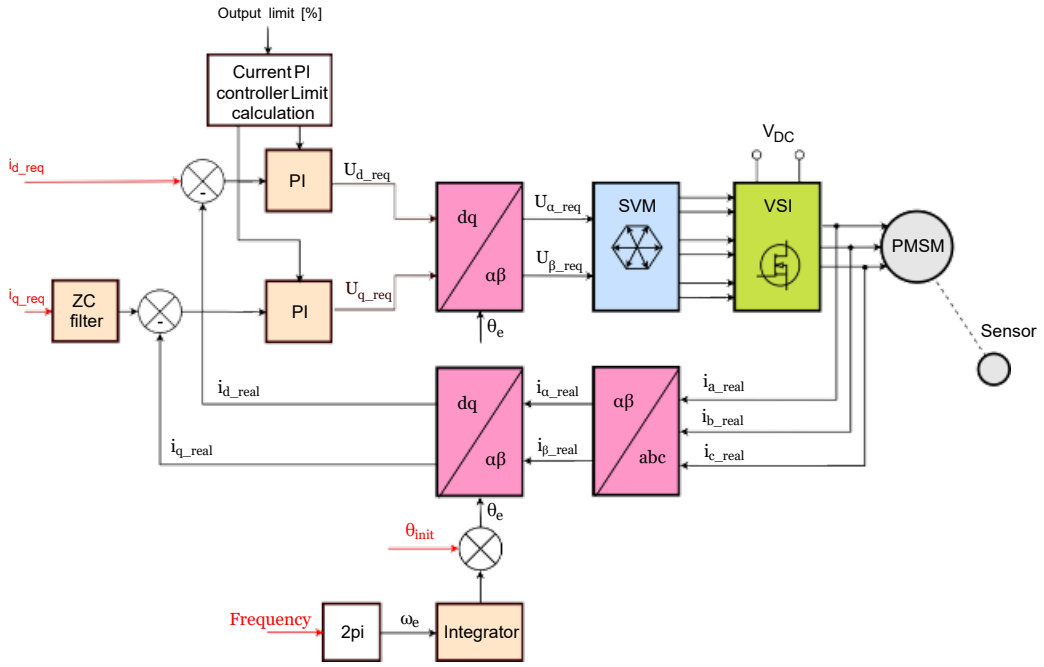


Figure 17. Current - open-loop control

To run the motor in current - open-loop control, follow these steps:

1. Switch the project tree subblock to "Openloop Control".
2. Switch the "M1 MCAT Control" variable to "OPEN_LOOP".
3. Set the "M1 Openloop Use I Control" variable to "1".
4. Set the required values in the "M1 Openloop Required Id" and "M1 Openloop Required Iq" variables.
5. Set the required initial position in the "M1 Openloop Theta Electrical" variable.
6. Set the required frequency in the "M1 Openloop Required Frequency Electrical" variable.
7. Set the "M1 Application Switch" variable to "1". The motor starts spinning.
8. Observe the motor speed, position, phase currents, and other graphs predefined in the subblock scopes and recorders.

8.5.3 Voltage control

Figure 18 shows the block diagram of the voltage FOC. Unlike the scalar control, the position feedback is closed using the BEMF observer and the stator-voltage magnitude is not dependent on the motor speed. You can specify both the D and Q axes stator voltages in the "M1 MCAT Ud Required" and "M1 MCAT Uq Required" fields. This control method is useful for the BEMF observer functionality check.

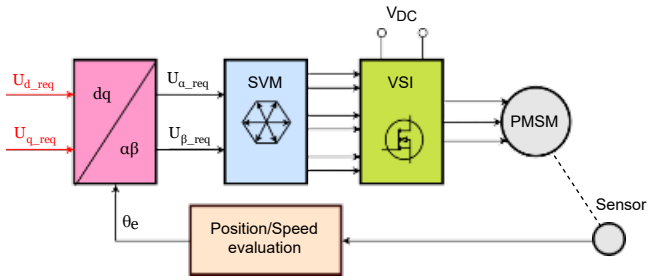


Figure 18. Voltage FOC control mode

To run the motor in voltage control, follow these steps:

1. Switch the project tree subblock to "Scalar & Voltage Control".
2. Switch the "M1 MCAT Control" variable to "VOLTAGE_FOC".
3. Set the required voltages in the "M1 MCAT Uq Required" and "M1 MCAT Ud Required" variables.
4. Set the "M1 Application Switch" variable to "1". The motor starts spinning.
5. Observe the motor speed, position, phase currents, and other graphs predefined in the subblock scopes and recorders.

8.5.4 Current/torque control

The current (or torque) FOC control requires the rotor position feedback and the currents transformed into a D-Q reference frame. There are two reference variables ("M1 MCAT Id Required" and "M1 MCAT Iq Required") available for the motor control, as shown in Figure 19. The D-axis current component "M1 MCAT Id Required" controls the rotor flux. The Q-axis current component of the "M1 MCAT Iq Required" current generates torque and, by its application, the motor starts spinning. By changing the polarity of the "M1 MCAT Iq Required" current, the motor changes the direction of rotation. Supposing the BEMF observer is tuned correctly, the current PI controllers can be tuned using the current FOC control structure.

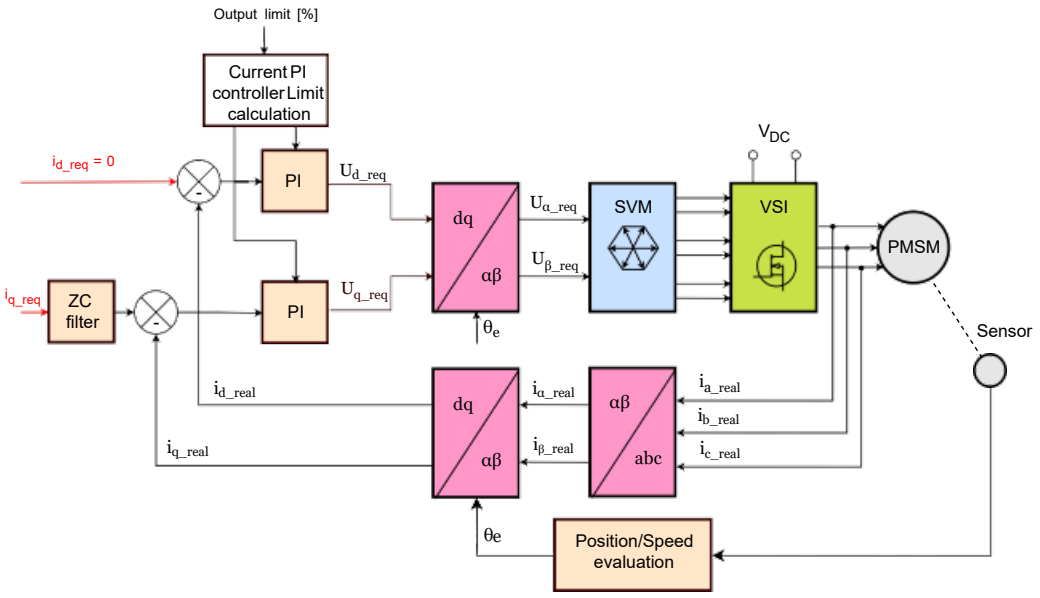


Figure 19. Current/torque control mode (sensored)

To run the motor in current control, follow these steps:

1. Switch the project tree subblock to "Current Control".
2. Switch the "M1 MCAT Control" variable to "CURRENT_FOC".
3. Set the required currents in the "M1 MCAT Iq Required" and "M1 MCAT Id Required" variables.
4. Set the "M1 Application Switch" variable to 1. The motor starts spinning.
5. Observe the motor speed, position, phase currents, and other graphs predefined in the subblock scopes and recorders.

8.5.5 Speed FOC control

As shown in Figure 20, the speed PMSM sensor/sensorless FOC is activated by enabling the speed FOC control structure. Enter the required speed into the "M1 Speed Required" field. The D-axis current reference is held at 0 during the entire FOC operation.

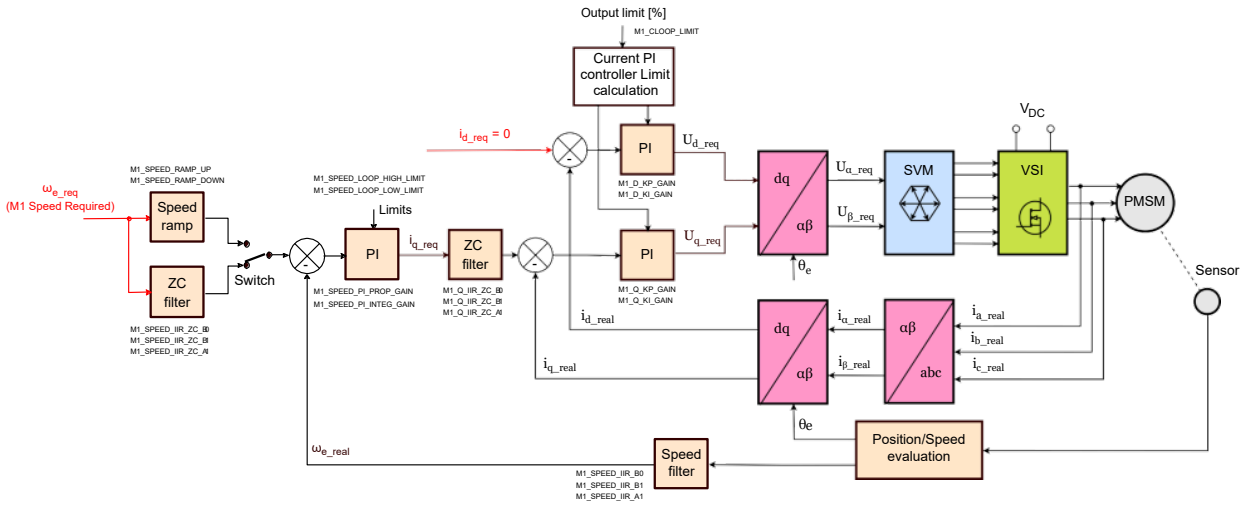


Figure 20. Speed FOC control mode (sensored)

To run the motor in the speed FOC control, follow these steps:

1. Switch the project tree subblock to "Speed Control".
2. Switch the "M1 MCAT Control" variable to "SPEED_FOC".
3. Choose between the sensored and sensorless control in the "M1 MCAT POSPE Sensor" variable.
4. Set the required speed (for example, 1000 RPM) in the "M1 Speed Required" variable. The motor starts spinning.
5. Observe the motor speed, position, phase currents, and other graphs predefined in the subblock scopes and recorders.

8.5.6 Position (servo) control

The position of the PMSM FOC sensor is shown in Figure 21 (available for sensored/encoder-based applications only). You can tune the position control using the P controller in the "Servo control" menu tab. An encoder sensor is required for the feedback. Without the sensor, the position control does not work. A braking resistor is missing on the FRDM-MC-LVPMSM board. Therefore, it is necessary to set soft parameters of the P controller because the voltage on the DC-bus can rise when braking the quickly spinning shaft. It may cause an overvoltage fault.

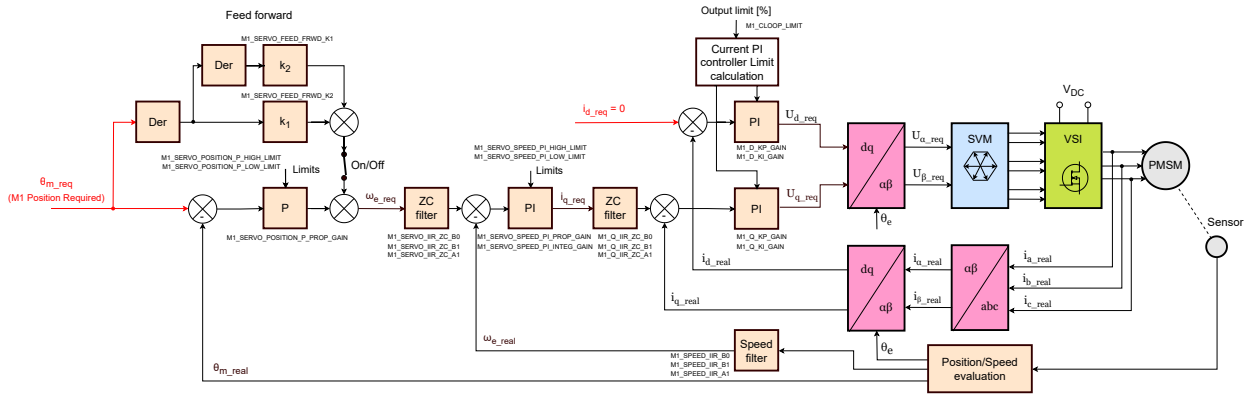


Figure 21. Position control mode

To run the motor in position (servo) control, follow these steps:

1. Switch the project tree subblock to "Position Control".
2. Switch the "M1 MCAT Control" variable to "POSITION_CNTRL".
3. Swich the "M1 MCAT POSPE Sensor" variable to "Encoder [1]".
4. Set the required position in the "M1 Position Required" variable (for example, 10 revolutions).
5. Set the "M1 Application Switch" variable to 1. The motor starts and automatically stops in the required position.
6. Change the "M1 Encoder Direction" variable if the motor does not spin. (See Section 8.12.1)
7. Observe the motor speed, position, phase currents, and other graphs predefined in the subblock scopes and recorders.

8.6 Faults explanation

When the motor is running or during the tuning process, there may be several fault conditions. Therefore, the motor-control example has an integrated fault indication located in the variable watch of the "Motor M1" FreeMASTER subblock. If a fault is indicated, the state machine enters the FAULT state.

Variable Watch

Name	Value	Unit
M1 Fault Pending	b# 0000 0000	BIN
M1 Fault Captured	b# 0000 0000	BIN
M1 Fault Captured Over Current	Not captured	ENUM
M1 Fault Captured DCBus Undervoltage	Not captured	ENUM
M1 Fault Captured DCBus Overvoltage	Not captured	ENUM
M1 Fault Captured Overload	Not captured	ENUM
M1 Fault Captured Overspeed	Not captured	ENUM
M1 Fault Captured Blocked Rotor	Not captured	ENUM
M1 Fault Clear	No request	ENUM
M1 Fault Enable DCBus Undervoltage	Enabled	ENUM
M1 Fault Enable DCBus Overvoltage	Enabled	ENUM
M1 Fault Enable Overload	Enabled	ENUM
M1 Fault Enable Overspeed	Enabled	ENUM
M1 Fault Enable Blocked Rotor	Enabled	ENUM

Figure 22. Faults in variable watch located in "Motor M1" subblock

8.6.1 Variable "M1 Fault Pending"

It shows the actually persisting faults, which means that the fault indicated during the fault conditions is accomplished. For example, if the source voltage is still under the undervoltage fault threshold, the undervoltage pending fault is shown. If the fault condition disappears, the pending fault is cleared automatically. "M1 Fault Pending" is shown in a binary format in the FreeMASTER variable watch. Each place in the variable denotes a different fault condition.

- b 0000 0001 - the overcurrent fault is indicated. If the overcurrent fault is present, the PWMs are automatically disabled. The fault occurs when the DC-bus current exceeds the **Imax** value (current-sensing HW scale).
- b 0000 0010 - the undervoltage fault is indicated. The undervoltage fault occurs when the UDCBus voltage (source voltage) is lower than the **U DCB under** threshold.
- b 0000 0100 - the overvoltage fault is indicated. The overvoltage fault occurs when the UDCBus voltage (source voltage) is higher than the **U DCB over** threshold.
- b 0000 1000 - the overload fault is indicated. The overload fault occurs when the rotor is overloaded.
- b 0001 0000 - the overspeed fault is indicated. The overspeed fault occurs when the rotor speed exceeds the **N over** threshold.
- b 0010 0000 - the blocked rotor fault is indicated. The blocked rotor fault occurs when the BEMF voltage is lower than the **E block** threshold and the duration of the drop is longer than **E block per**.
- b 0100 0000 - the ENC timeout fault is indicated. This fault occurs when the speed/position feedback sensor is not working properly (for example, the encoder has no power supply, the ENC ISR is not called, and so on).
  Note: *The ENC timeout fault is integrated only in servo applications that use EnDat/BiSS encoders.*

Variable Watch

Name	Value	Unit
M1 Fault Pending	b# 0000 0010	BIN
M1 Fault Captured	b# 0000 0010	BIN
M1 Fault Captured Over Current	Not captured	ENUM
M1 Fault Captured DCBus Undervoltage	Captured	ENUM
M1 Fault Captured DCBus Overvoltage	Not captured	ENUM
M1 Fault Captured Overload	Not captured	ENUM
M1 Fault Captured Overspeed	Not captured	ENUM
M1 Fault Captured Blocked Rotor	Not captured	ENUM
M1 Fault Clear	No request	ENUM
M1 Fault Enable DCBus Undervoltage	Enabled	ENUM
M1 Fault Enable DCBus Overvoltage	Enabled	ENUM
M1 Fault Enable Overload	Enabled	ENUM
M1 Fault Enable Overspeed	Enabled	ENUM
M1 Fault Enable Blocked Rotor	Enabled	ENUM

Figure 23. Undervoltage fault is indicated (pending)

8.6.2 Variable "M1 Fault Captured"

If any fault condition appears, the fault captured is indicated. Similar to "fault pending", "fault captured" is shown in the BIN format, but every fault type has its own variable ("M1 Fault Captured Over Curent" and others). For example, if the undervoltage fault condition is accomplished, the "fault captured" is indicated. The "fault captured" is also indicated after the undervoltage fault condition disappears. The captured faults are cleared manually by writing "Clear [1]" to "M1 Fault Clear".

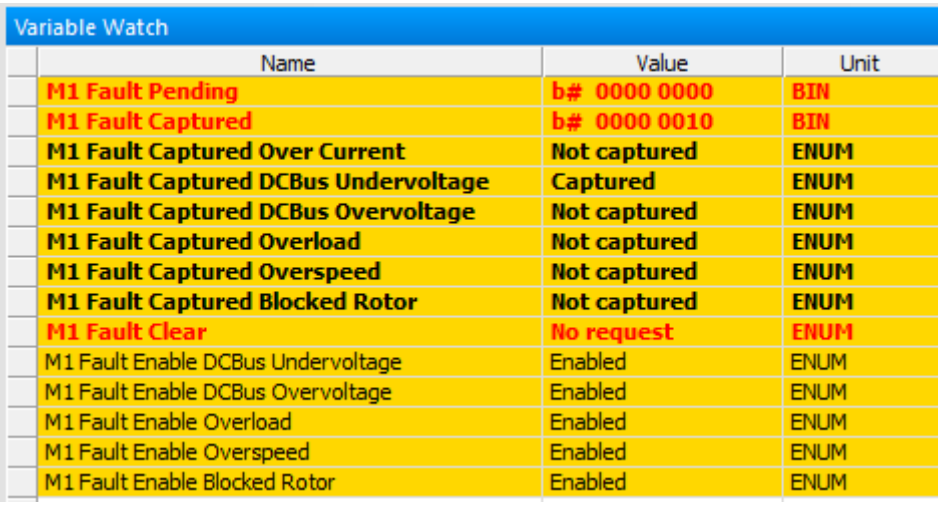


Figure 24. Undervoltage fault is captured

8.6.3 Variable "M1 Fault Enable"

The fault indication can be unwanted during the tuning process. Therefore, the fault indication can be disabled by writing "Disabled [0]" to the "M1 Fault Enable" variables.

Note: *The overcurrent fault cannot be disabled.*

Note: *The fault thresholds are in the "MCAT parameters" tab.*

8.7 Initial motor parameters and hardware configuration

The motor-control examples contain two or more configuration files: `m1_pmsm_appconfig.h`, `m2_pmsm_appconfig.h`, and so on. Each contains constants tuned for the selected motor (see the supported motors in Section 2). The initial motor parameters and the hardware configuration (inverter) are loaded to the MCAT from the `m1_pmsm_appconfig.h` configuration file. There are three ways to change the motor configuration corresponding to the connected motor.

1. The first way is to rename the configuration file:
   - In the project example folder, find the configuration file to use.
   - Rename this configuration file to `m1_pmsm_appconfig.h`.
   - Rebuild the project and load the code to the MCU.
2. The second way is to change the motor configuration, as described in Section 8.3.
3. The last way is to change the motor and hardware parameters manually:
   - Open the PMSM control application FreeMASTER project containing the dedicated MCAT plug-in module.
   - Select the "Parameters" tab.
   - Specify the parameters manually. You can obtain the motor parameters from the motor data sheet or using the PMSM parameters measurement procedure described in *PMSM Electrical Parameters Measurement* (document AN4680). All parameters provided in Table 15 are accessible. The motor inertia J expresses the overall system inertia and can be obtained using a mechanical measurement. The J parameter is used to calculate the speed controller constant. However, you can use also manual controller tuning to calculate this constant.

Table 15. MCAT motor parameters

Parameter	Units	Description	Typical range
pp	[-]	Motor pole-pairs	1-10

Table 15. MCAT motor parameters*...continued*

Parameter	Units	Description	Typical range
Rs	[Ω]	1-phase stator resistance	0.3-50
Ld	[H]	1-phase direct inductance	0.00001-0.1
Lq	[H]	1-phase quadrature inductance	0.00001-0.1
Ke	[V.sec/rad]	BEMF constant	0.001-1
J	[kg.m^2]	System inertia	0.00001-0.1
Iph nom	[A]	Motor nominal phase current	0.5-8
Uph nom	[V]	Motor nominal phase voltage	10-300
N nom	[RPM]	Motor nominal speed	1000-2000

- Set the hardware scales - the modification of these two fields is not required when a reference to the standard power stage board is used. These scales express the maximum measurable analog current and voltage quantities.
- Check the fault limits - these fields are calculated using the motor parameters and hardware scales (see Table 16).

Table 16. Fault limits

Parameter	Units	Description	Typical range
U DCB trip	[V]	Voltage value at which the external braking resistor switch turns on	U DCB Over ~ U DCB max
U DCB under	[V]	Trigger value at which the undervoltage fault is detected	0 ~ U DCB Over
U DCB over	[V]	Trigger value at which the overvoltage fault is detected	U DCB Under ~ U max
N over	[RPM]	Trigger value at which the overspeed fault is detected	N nom ~ N max
N min	[RPM]	Minimal actual speed value for the sensorless control	(0.05~0.2) *N max

- Check the application scales - these fields are calculated using the motor parameters and hardware scales (see Table 17).

Table 17. Application scales

Parameter	Units	Description	Typical range
N max	[RPM]	Speed scale	>1.1 * N nom
E max	[V]	BEMF scale	ke* Nmax
kt	[Nm/A]	Motor torque constant	-

- Check the alignment parameters - these fields are calculated using the motor parameters and hardware scales. The parameters express the required voltage value applied to the motor during the rotor alignment and its duration.
- To save the modified parameters into the inner file, click the "Store data" button.

8.8 Identifying parameters of user motor

Because the model-based control methods of the PMSM motors provide high performance (for example, dynamic response, efficiency), obtaining an accurate model of a motor is an important part of the drive design and control. For the implemented FOC algorithms, it is necessary to know the value of the stator resistance R_s, direct inductance L_d, quadrature inductance L_q, and BEMF constant K_e. Unless the default PMSM motor described above is used, the motor parameter identification is the first step in the application tuning. This section shows how to identify user motor parameters using MID. MID is written in floating-point arithmetics. Each MID algorithm is detailed in Section 8.9. MID can be controlled either via the MCAT control tab "Motor parameters identification (MID)" or through the FreeMASTER variable watch in the "Motor Identification" subblock (see Figure 26). To control MID using the FreeMASTER variable watch, the MID mode must be activated first (see Section 8.8.2).

8.8.1 Motor parameter identification using MCAT

Enter the nominal current (I Nom) and nominal speed (N Nom) of the motor into the MCAT input fields. Start the "Pole-pair assistant" by clicking the "START" button. After the number of pole pairs is determined, click "STOP" and enter this value into the MCAT input field "Pp". Continue with the "Electrical parameters measurement", which requires "I Nom" as an input parameter. Once the electrical parameters are estimated, proceed to the "Mechanical parameters measurement". The required inputs are "I Nom" and "N Nom". Estimated values are automatically copied to the input fields in the "Parameters" tab. Click the "Update target" button to update the new values on the target MCU. The measurement process can be stopped at any time by clicking the "STOP" button associated with the triggered measurement.

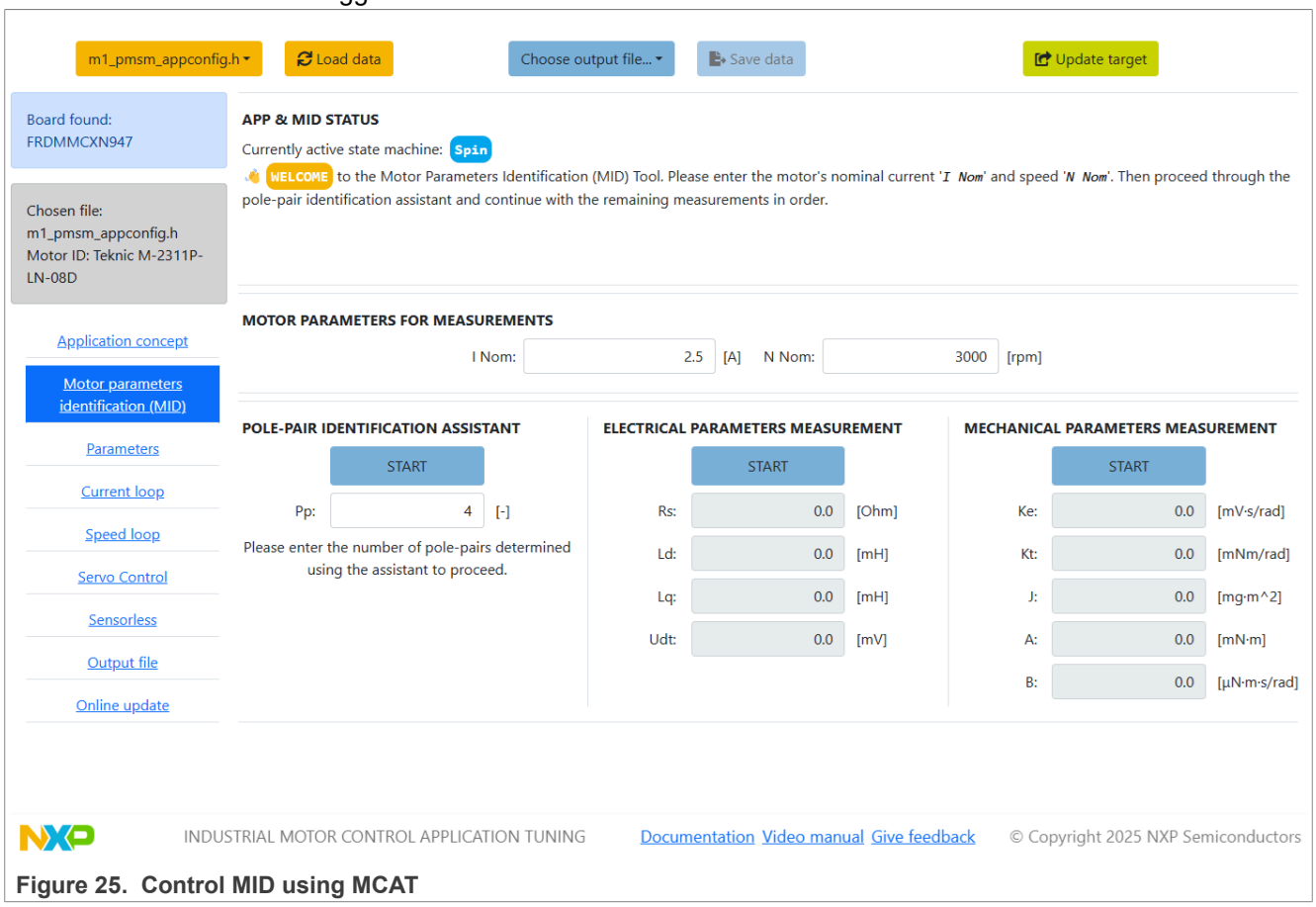


Figure 25. Control MID using MCAT

Input measurement parameters and corresponding FreeMASTER variables for each measurement type are listed in the table below. Additional FreeMASTER variables that require updates are managed internally by the MCAT background process and cannot be modified through the MCAT interface.

Table 18. MID measurement parameters configurable in MCAT

Measurement type	Updated FreeMASTER variable	Updated value
Pole-pair assistant	MID: Config Pp Id Meas	0.2 * I Nom
	MID: Config Pp Freq El. Required	0.1 * N Nom / 60
Electrical measurement	MID: Config El I DC Nominal	0.5 * I Nom
Mechanical measurement	MID: Config Mech Friction Estimation	1
	MID: Config Mech I Nominal	I Nom
	MID: Config Mech N Nominal	N Nom
	MID: Known Param Pp	Pp

When using MCAT for electrical parameter measurement, Mode 0 is used (see Section 8.10.1). For mechanical parameters measurement, the advanced mode (see Section 8.11.2) is utilized, where two measurement speeds derived from the nominal speed are internally calculated by the estimation algorithm.

8.8.2 Switch between Spin and MID

You can switch between two modes of application: *Spin* and *MID* (Motor Identification). The *Spin* mode is used to control PMSMs (see Section 8.3). The *MID* mode is used for motor parameter identification (see Section 8.8.3). Switching is controlled via the FreeMASTER variable watch in the "Motor M1" or "Motor identification" subblocks. The actual mode of application is shown in the *APP: State* variable. The mode is changed by writing "ON" to *APP: MID to Spin request* or *APP: Spin to MID request* variables. The transition between Spin and MID can be done only if the actual mode is in a defined stop state (for example, MID is not in progress or the motor is stopped). The result of the change mode request is shown in the *APP: Fault* variable. The MID fault occurs when parameter identification still runs, or the MID state machine is in the fault state. A spin fault occurs when the *M1 Application switch* variable watch is ON or when the *M1 Application state* variable watch is not "STOP".

8.8.3 Motor parameter identification using variable watch

The whole MID is controlled via the FreeMASTER variable watch. The MID subblock is shown in Figure 26. The motor parameter identification workflow is as follows:

1. Set the *MID: Command* variable to "STOP".
2. Select the measurement type that you want to perform via the *MID: Measurement Type* variable:
   - `PP_ASSIST` - Pole-pair identification assistant
   - `EL_PARAMS` - Electrical parameters measurement
   - `MECH_PARAMS` - Mechanical parameters measurement
3. Insert the known motor parameters via the *MID: Known Param* set of variables. All parameters with a non-zero known value are used instead of the measured parameters (if necessary).
4. Set the measurement configuration parameters in the *MID: Config* set of variables.
5. Start the measurement by setting *MID: Command* to "START".
6. Observe the actual *MID: State*, *MID: Faults*, and *MID: Finished measurements* variables to check the estimation progress.
7. If the measurement finishes successfully, the corresponding bit in the *MID: Finished measurements* variable is set. The measured motor parameters appear in the *MID: Measured* set of variables, and *MID: State* changes to "STOP".

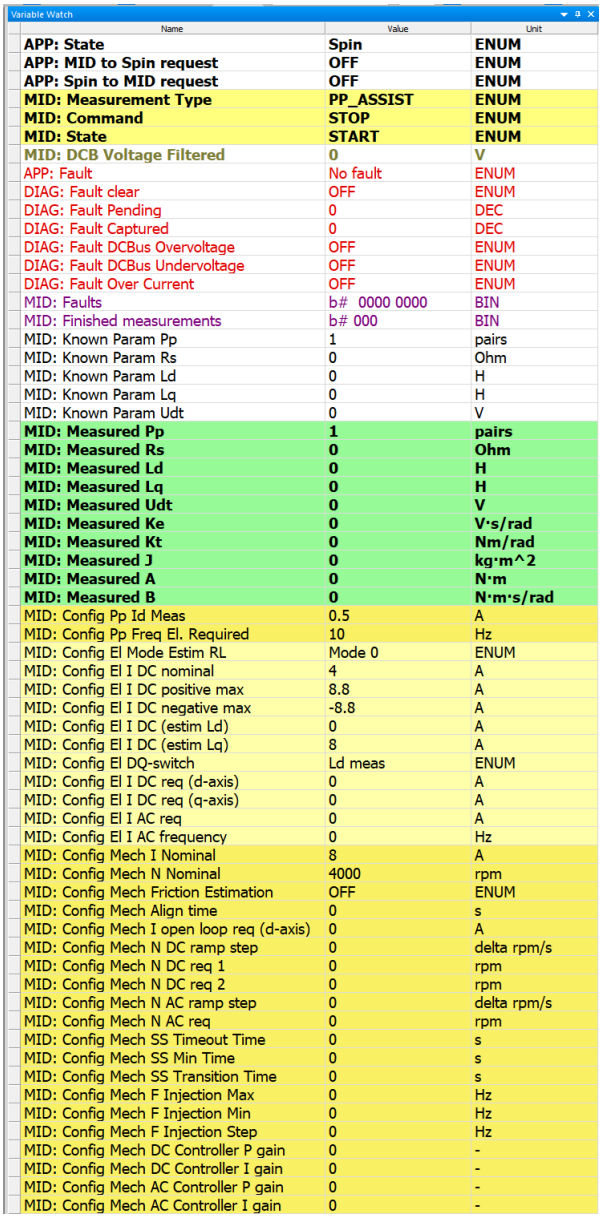


Figure 26. FreeMASTER MID variable watch

8.8.4 MID faults

The MID faults are saved in the format of masks in the *MID: Faults* variable. Faults are cleared automatically when starting a new measurement. If a MID fault appears, the measurement process stops and brings the MID state machine safely to the "STOP" state. For more details on individual faults, see Table 19.

.

Table 19. Measurement faults

Fault mask	Fault description	Fault reason	Troubleshooting
b#0000 0001	Align current fault	Align current is out of range.	Check the align current in the sUserMIDMeasConfig structure.

Table 19. Measurement faults*...continued*

Fault mask	Fault description	Fault reason	Troubleshooting
b#0000 0010	Pp measurement fault	Pole-pair assist measurement configuration is out of range.	Check the Pp measurement configuration.
b#0000 0100	Electrical parameters initialization fault	Electrical measurement configuration is out of range.	Check and update the measurement configuration regarding the used mode.
b#0000 1000	Electrical parameters measurement fault	The required value cannot be reached or the measurement configuration is wrong.	Check whether the measurement configuration is valid.
b#0001 0000	Pp number fault	The number of pole-pair is out of range.	Update the pole-pair number to fit the range.
b#0010 0000	Input electrical parameters fault	The required electrical parameters for a mechanical measurement are missing.	Fill the missing parameters.
b#0100 0000	Mechanical parameters initialization fault	The mechanical measurement configuration is out of range.	Check and update the measurement configuration of the used mode.
b#1000 0000	Electrical parameters measurement fault	The required value cannot be reached or the measurement configuration is wrong.	Check whether the measurement configuration is valid.

8.8.5 MID finished status

After the selected MID measurement comletes successfully, the corresponding bit in *MID: Finished measurements* is set. It is cleared when the measurement is triggered again.

Table 20. Measurement finished status

MID finished mask	Description
b#0001	Pole-pair assist completed
b#0010	Electrical parameters measurement completed
b#0100	Mechanical parameters measurement completed

8.9 MID algorithms

This section describes how each MID algorithm works.

8.9.1 Stator resistance measurement

The stator resistance R_s is averaged from the DC steps generated by the algorithm. The DC step levels are automatically derived from the currents inserted by the user. For more details, see the documentation of the `AMCLIB_EstimRL` function in AMMCLib.

8.9.2 Stator inductances measurement

Injection of the AC/DC currents is used for the inductances (L_d *and* L_q) estimation. The injected AC/DC currents are automatically derived from the currents inserted by the user. The default AC current frequency is 500 Hz. For more details, see the documentation of the `AMCLIB_EstimRL` function in AMMCLib.

8.9.3 Number of pole-pair assistant

The number of pole-pairs can be measured only with a position sensor. However, there is a simple assistant to determine the number of pole-pairs (`PP_ASSIST`). If *pp* is 1, the assistant performs one electrical revolution, stops for a few seconds, and then repeats. Because the *pp* value is the ratio between the electrical and mechanical speeds, it can be determined as the number of stops per one mechanical revolution. It is recommended to refrain from counting the stops during the first mechanical revolution because the alignment occurs during the first revolution and affects the number of stops. During the `PP_ASSIST` measurement, the current loop is enabled, and the I_d current is controlled to "MID: Config Pp Id Meas". The electrical position is generated by integrating the open-loop frequency "MID: Config Pp Freq El. Required". If the rotor does not move after the start of the `PP_ASSIST` assistant, stop the assistant, increase "MID: Config Pp Id Meas", and restart the assistant.

8.9.4 Mechanical parameters measurement

The mechanical parameters measurement includes the estimated BEMF constant K_e, torque constant K_t, moment of inertia J, and, optionally, static friction A and viscous friction B. The mechanical estimation procedure is implemented through an internal FOC controller that drives the motor and observes its response characteristics. The mechanical estimation requires the motor electrical parameters, which can be estimated using the *MCAA_EstimRL_FLT* function. For more details, see the documentation of the `AMCLIB_EstimBJ` function from AMMCLib.

8.10 Electrical parameters measurement control

This section describes how to control the electrical parameters measurement, which contains measuring stator resistance R_s, direct inductance L_d, and quadrature inductance L_q. There are four modes of measurement, selected by the "MID: Config El Mode Estim RL" variable.

The `MCAA_EstimRLInit_FLT` function must be called before the first use of the `MCAA_EstimRL_FLT` function. The `MCAA_EstimRL_FLT` function must be called periodically with the `F_SAMPLING` sampling period, which can be defined by the user. The maximum sampling frequency `F_SAMPLING` is 10 kHz. In the scopes under "Motor identification", the FreeMASTER subblock can be observed in measured currents, estimated parameters, and so on.

8.10.1 Mode 0

This mode is automatic. Inductances are measured at a single operating point. The rotor is not fixed. The user has to specify the nominal current ("MID: Config El I DC nominal" variable). The AC and DC currents are automatically derived from the nominal current. The frequency of the AC signal is set to 500 Hz.

The function outputs stator resistance R_s, direct inductance L_d, and quadrature inductance L_q.

8.10.2 Mode 1

DC stepping is automatic in this mode. The rotor is not fixed. Compared to Mode 0, an automatic measurement of the inductances for a defined number (*NUM_MEAS*) of different DC current levels is performed using positive values of the DC current. The L_{dq} dependency map is in the "Inductances (Ld, Lq)" recorder. Specify the following parameters before the parameters estimation:

- "MID: Config El I DC (estim Lq)" - current to determine L_q. In most cases, it is the nominal current.
- "MID: Config El I DC positive max" - the maximum positive DC current for the L_{dq} dependency map measurement.

The injected AC and DC currents are automatically derived from the "MID: Config El I DC (estim Lq)" and "MID: Config El I DC positive max" currents. The frequency of the AC signal is set to 500 Hz.

The function outputs stator resistance R_s, direct inductance L_d, quadrature inductance L_q, and L_{dq} dependency map.

8.10.3 Mode 2

DC stepping is automatic in this mode. The rotor must be mechanically fixed after the initial alignment with the first phase. Compared to Mode 1, an automatic measurement of the inductances for a defined number (*NUM_MEAS*) of different DC current levels is performed using both the positive and negative values of the DC current. The estimated inductances are in the "Inductances (Ld, Lq)" recorder. Specify the following parameters before the parameters estimation:

- "MID: Config El I DC (estim Ld)" - current to determine L_d. In most cases, it is 0 A.
- "MID: Config El I DC (estim Lq)" - current to determine L_q. In most cases, it is the nominal current.
- "MID: Config El I DC positive max" - the maximum positive DC current for the L_{dq} dependency map measurement. In most cases, it is the nominal current.
- "MID: Config El I DC negative max" - the maximum negative DC current for the L_{dq} dependency map measurement.

The injected AC and DC currents are automatically derived from the "MID: Config El I DC (estim Ld)", "MID: Config El I DC (estim Lq)", MID: Config El I DC positive max", and "MID: Config El I DC negative max" currents. The frequency of the AC signal is set to 500 Hz.

The function outputs stator resistance R_s, direct inductance L_d, quadrature inductance L_q, and L_{dq} dependency map.

8.10.4 Mode 3

This mode is manual. The rotor must be mechanically fixed after alignment with the first phase. R_s is not calculated in this mode. The estimated inductances can be observed in the "Ld" or "Lq" scopes. The following parameters can be changed during the runtime:

- "MID: Config El DQ-switch" - the axis switch for the AC signal injection (0 for injecting the AC signal to the D axis, 1 for injecting the AC signal to the Q axis).
- "MID: Config El I DC req (d-axis)" - the required DC current in the D axis.
- "MID: Config El I DC req (q-axis)" - the required DC current in the Q axis.
- "MID: Config El I AC req" - the required AC current injected to the D axis or the Q axis.
- "MID: Config El I AC frequency" - the required frequency of the AC current injected to the D axis or the Q axis.

Unless specified otherwise, all modes operate with the default parameters. To tune the advanced algorithm parameters, update the variables in the advanced tuning structure `g_sEstimRLAdvTune` as needed. Navigate to the `mid_sm_states.c` file to update this structure.

8.11 Mechanical parameters measurement control

This section describes how to control the mechanical parameters measurement, which contains the measuring BEMF constant K_e, torque constant K_t, moment of inertia J, static friction A, and viscous friction B. There are the basic and advanced modes.

The `MCAA_EstimBJInit_FLT` initialization function must be called before the first use of `MCAA_EstimBJ_FLT`. Function `MCAA_EstimBJ_FLT` must be called periodically with the sampling period specified during initialization. Additionally, the mechanical parameter estimation requires the pole-pair number, nominal current and speed, as well as the motor electrical parameters. Observe the measured currents, estimated parameters, and other values in the "Motor identification" FreeMASTER subblock.

8.11.1 Basic mode

In this mode, one speed measurement is used to estimate the BEMF constant K_e, torque constant K_t, and moment of Inertia *J*. This speed can be calculated internally by the algorithm from the nominal speed or edited manually by the user during initialization. To modify this speed, update the *MID: Config Mech N DC req 1* FreeMASTER variable.

In the basic mode, the *MID: Config Mech Friction Estimation* variable must be set to "OFF" and the *MID: Config Mech N DC req 2* FreeMASTER variable is ignored.

8.11.2 Advanced mode

When compared to the basic mode, two measurement speeds are used in the advanced mode. In addition to the K_e, K_t, and *J* parameters, the algorithm also estimates static friction *A* and viscous friction *B*. These two speeds can be calculated internally by the algorithm from the nominal speed or edited manually by the user during initialization. To modify these speeds, update the *MID: Config Mech N DC req 1* and *MID: Config Mech N DC req 2* FreeMASTER variables.

In the advanced mode, the *MID: Config Mech Friction Estimation* variable must be set to "ON".

8.12 Control parameters tuning

For a correct current measurement and proper functionality of the BEMF observer, follow these steps:

1. Select the scalar control in the "M1 MCAT Control" FreeMASTER variable watch.
2. Set the "M1 Application Switch" variable to "ON". The application state changes to "RUN".
3. Set the required frequency value in the "M1 Scalar Freq Required" variable; for example, 15 Hz in the "Scalar & Voltage Control" FreeMASTER project tree. The motor starts spinning.
4. Select the "Phase Currents" recorder from the "Scalar & Voltage Control" FreeMASTER project tree.
5. The optimal ratio for the V/Hz profile can be found by changing the V/Hz factor directly using the "Scalar V/Hz factor ratio" MCAT input. The shape of the motor currents should be close to a sinusoidal shape (Figure 27).

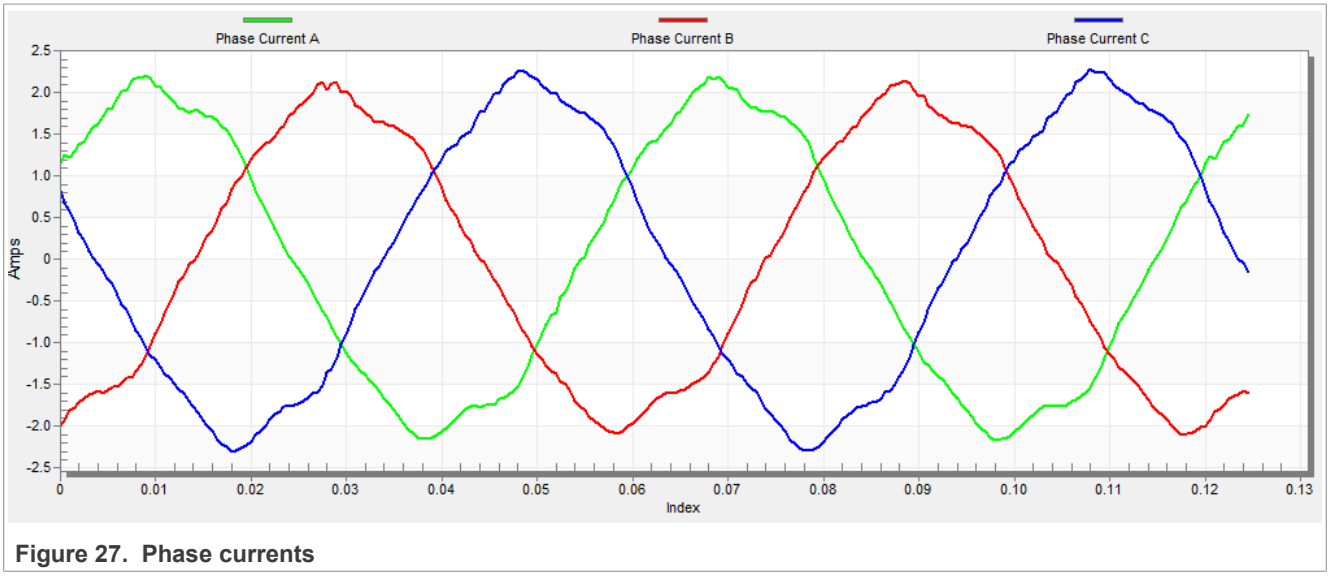


Figure 27. Phase currents

6. Select the "Position" recorder to check the observer functionality. The difference between the "Position Electrical Scalar" and the "Position Estimated" should be minimal (see Figure 28) for the BEMF position and speed observer to work properly. The position difference depends on the motor load. The higher the load, the bigger the difference between the positions due to the load angle.

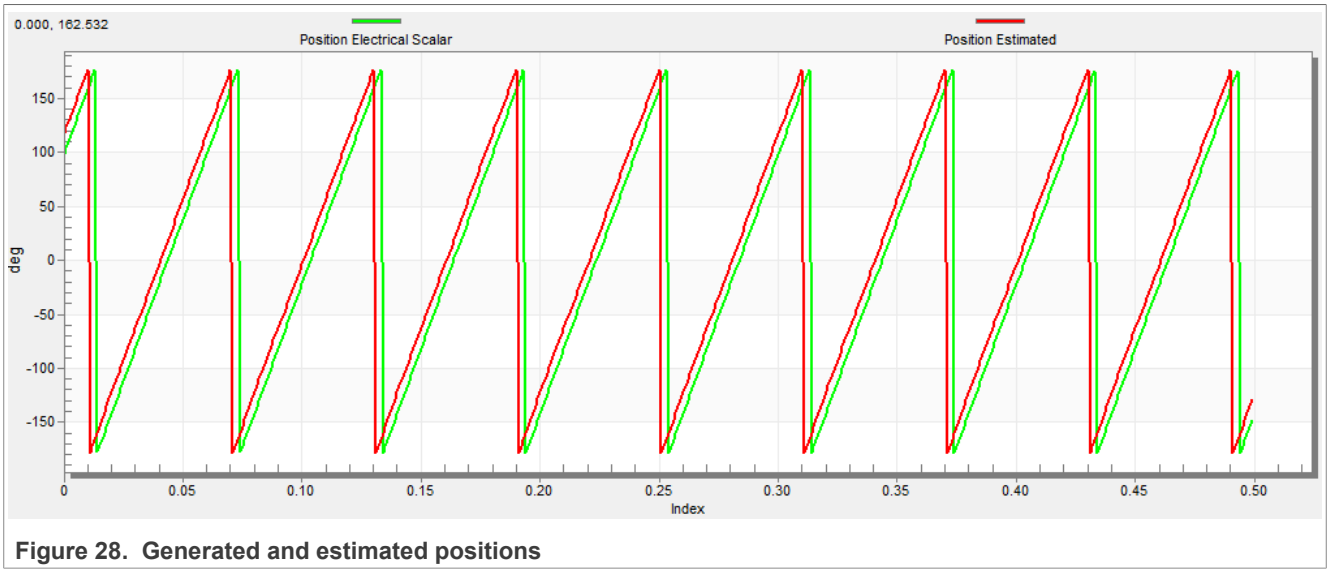


Figure 28. Generated and estimated positions

7. If an opposite speed direction is required, set a negative speed value into the "M1 Scalar Freq Required" variable.
8. The proper observer functionality and the measurement of analog quantities are expected at this step.
9. Enable the voltage FOC mode in the "M1 MCAT Control" variable while the main application switch "M1 Application Switch" is turned off.
10. Switch the main application switch on and set a non-zero value in the "M1 MCAT Uq Required" variable. The FOC algorithm uses the estimated position to run the motor.

8.12.1 Encoder sensor setting

The encoder sensor settings are in the "Sensors" tab. The encoder sensor enables you to compute the speed and position for the sensored speed. For a proper encoder counting, set the number of encoder pulses per one revolution and the proper counting direction. The number of encoder pulses is based on information about the encoder from its manufacturer. If the encoder sensor has more pulses per revolution, the speed and position computing is more accurate. The counting direction is provided by connecting the encoder signals to the NXP Freedom board and also by connecting the motor phases.

To determine the direction of rotation, follow these steps:

1. Navigate to the "Scalar & Voltage Control" tab in the project tree and select "SCALAR_CONTROL" in the "M1 MCAT Control" variable.
2. Turn the application switch on. The application state changes to "RUN".
3. Set the required frequency value in the "M1 Scalar Freq Required" variable (for example, 15 Hz). The motor starts spinning.
4. Check the encoder direction. Select the "Encoder Direction Scope" from the "Scalar & Voltage Control" project tree. If the encoder direction is right, the estimated speed is equal to the measured mechanical speed. If the measured mechanical speed is opposite to the estimated speed, the direction must be changed. The first way is to change the "M1 Encoder Direction" variable - only the 0 or 1 values are allowed. The second way is to invert the encoder wires - phase A and phase B (or the other way round).

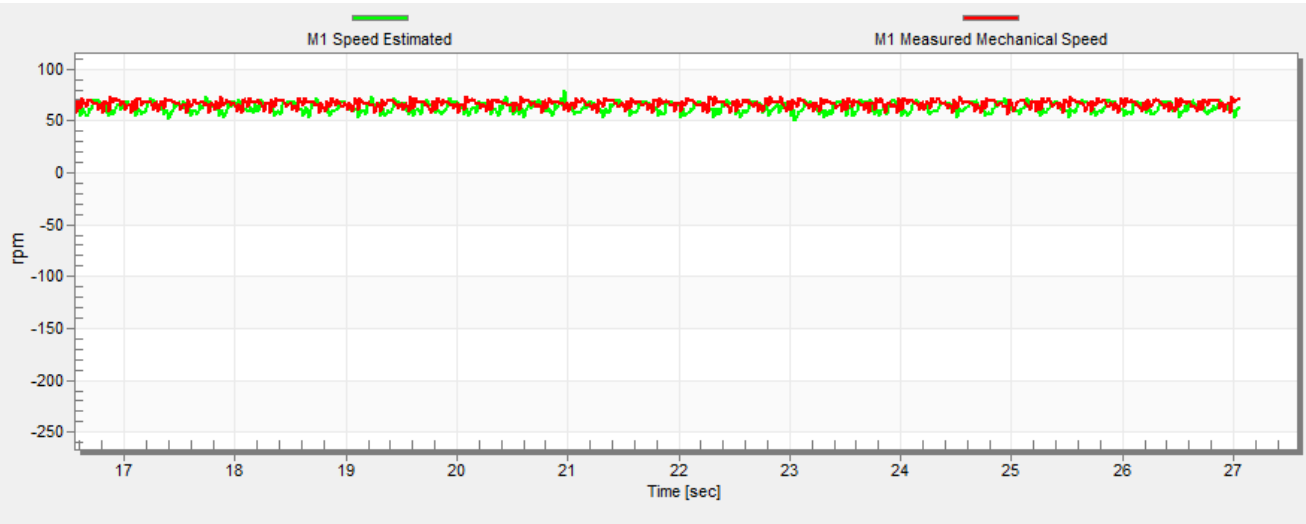


Figure 29. Encoder direction - right direction

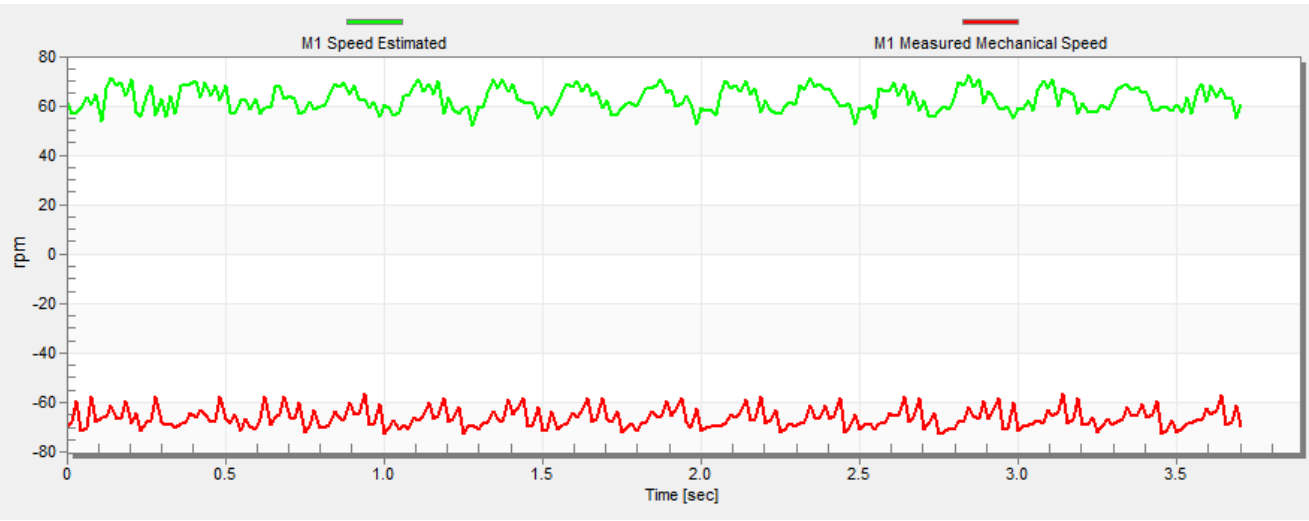


Figure 30. Encoder direction - wrong direction

8.12.2 Alignment tuning

For the alignment parameters, navigate to the "Parameters" MCAT tab. The alignment procedure sets the rotor to an accurate initial position and enables you to apply full startup torque to the motor. A correct initial position is needed mainly for high startup loads (compressors, washers, and so on). The alignment aims to have the rotor in a stable position, without any oscillations before the startup.

- The alignment voltage is the value applied to the D axis during the alignment. Increase this value for a higher shaft load.
- The alignment duration expresses the time when the alignment routine is called. Tune this parameter to eliminate rotor oscillations or movement at the end of the alignment process.

8.12.3 Current loop tuning

The parameters for the current D, Q, and PI controllers are fully calculated using the motor parameters and no action is required in this mode. If the calculated loop parameters do not correspond to the required response, the bandwidth and attenuation parameters can be tuned.

1. Select "Openloop Control" in the FreeMASTER project tree, set "M1 MCAT Control" to "OPENLOOP_CTRL", and switch "M1 Openloop Use I Control" on.
2. Turn the application on by switching "M1 Application Switch" on and then set "M1 Openloop Requred Id" for rotor alignment. The rotor alignment always uses Id, even when you are tuning the Q axis regulator.
3. Mechanically lock the motor shaft and turn the application off.
4. Set the required loop bandwidth and attenuation in the MCAT "Current loop" tab and then click the "Update target" button. The tuning loop bandwidth parameter defines how fast the loop response is while the tuning loop attenuation parameter defines the actual overshoot magnitude.
5. Select the "Current Controller Id" recorder in the project tree, turn the application on, and set the required step amplitude in "M1 Openloop Requred Id". Observe the step response in the recorder.
6. Tune the loop bandwidth and attenuation until you achieve the required response. The example waveforms show the correct and incorrect settings of the current loop parameters:
   - The loop bandwidth is low (100 Hz) and the settling time of the Id current is long (Figure 31).

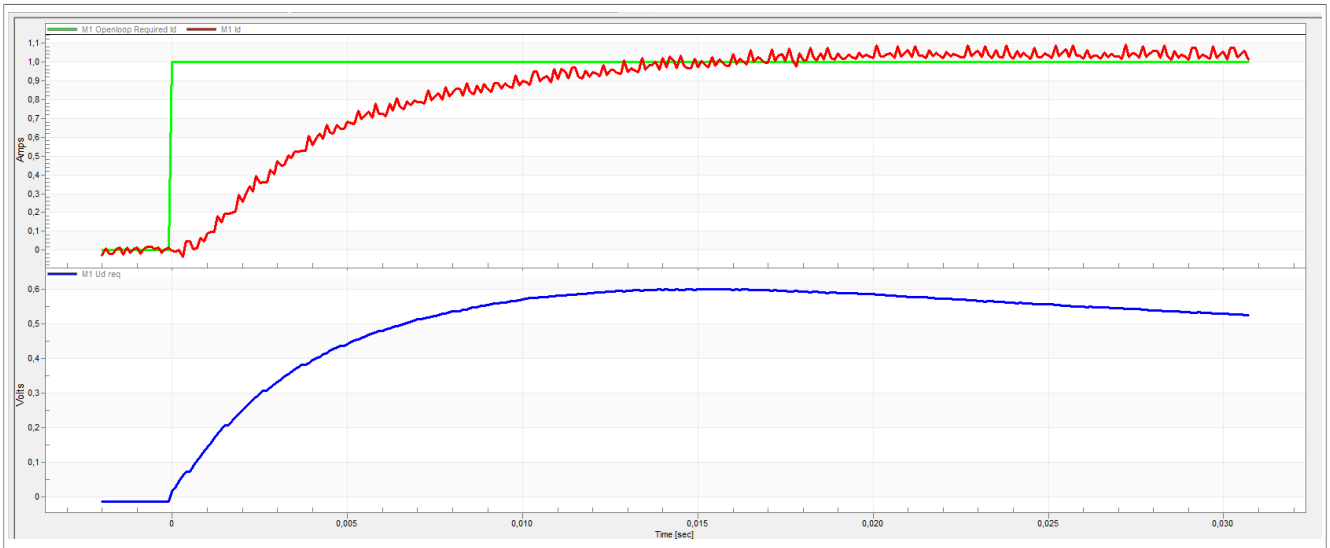


Figure 31. Slow step response of the Id current controller

   - The loop bandwidth (300 Hz) is optimal and the response time of the Id current is sufficient (Figure 32).

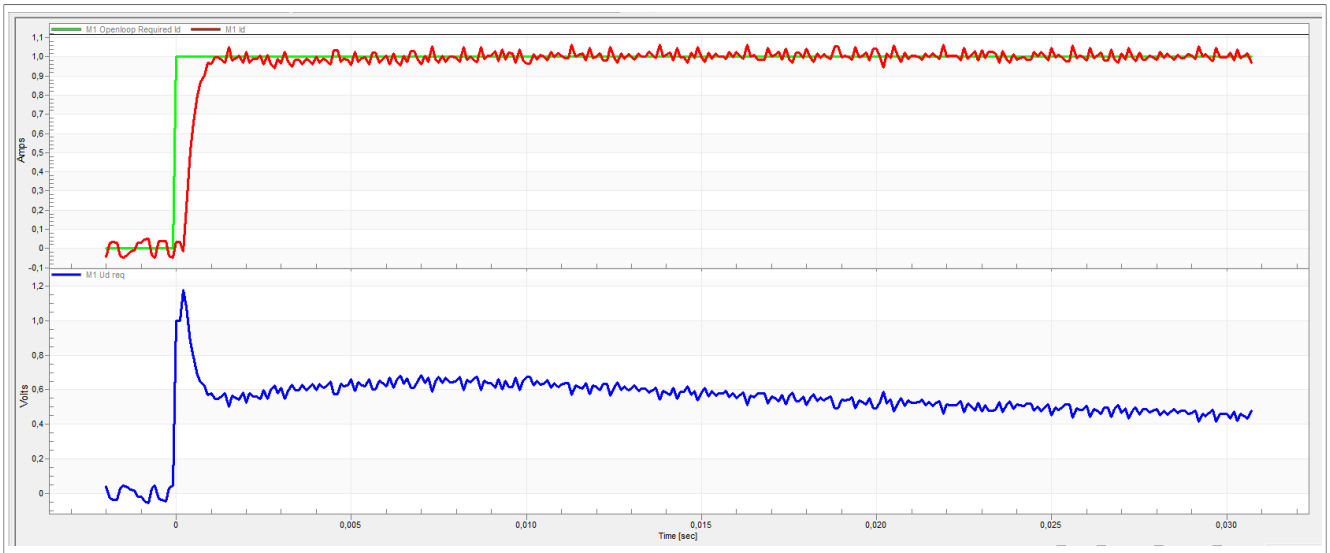


Figure 32. Optimal step response of the Id current controller

- The loop bandwidth is high (700 Hz) and the response time of the Id current is very fast, but with oscillations and overshoot (Figure 33).

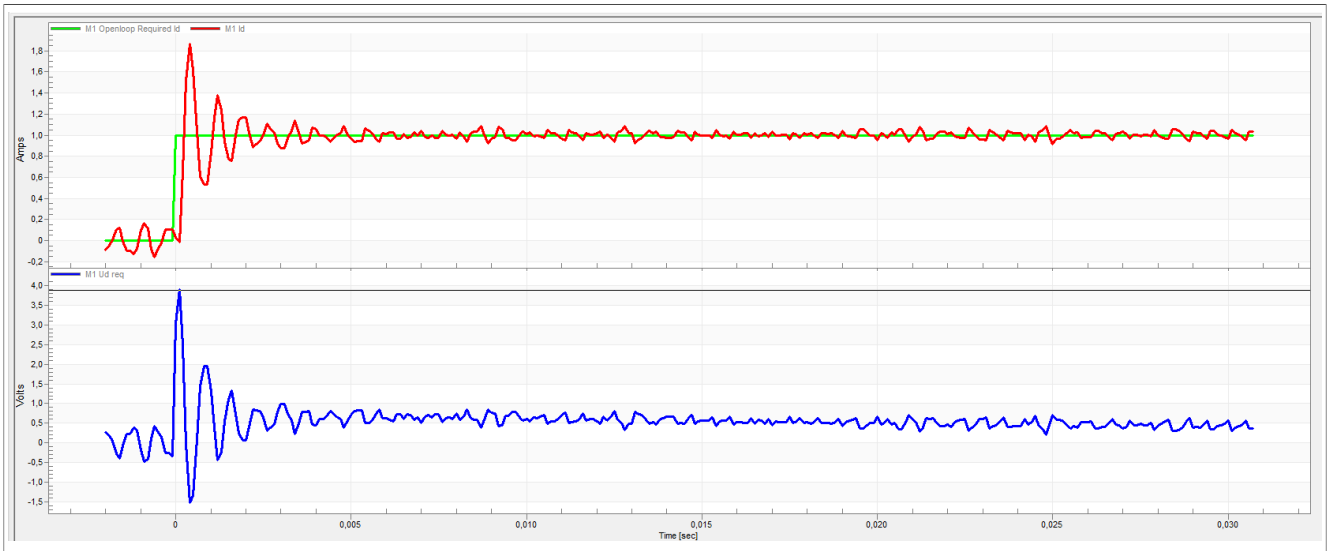


Figure 33. Fast step response of the Id current controller

8.12.4 Speed ramp tuning

To tune the speed ramp parameters, follow these steps:

1. The speed command is applied to the speed controller through a speed ramp. The ramp function contains two increments (up and down), which express the motor acceleration and deceleration per second. If the increments are very high, they can cause an overcurrent fault during acceleration and an overvoltage fault during deceleration. In the "Speed" scope, you can see whether the "Speed Actual Filtered" waveform shape equals the "Speed Ramp" profile.
2. The increments are common for the scalar and speed control. The increment fields are in the "Speed loop" tab and accessible in both tuning modes. Clicking the "Update target" button applies the changes to the MCU. An example speed profile is shown in Figure 34. The ramp increment down is set to 500 RPM/sec and the increment up is set to 3000 RPM/sec.

3. The startup ramp increment is in the "Sensorless" tab and its value is higher than the speed-loop ramp.

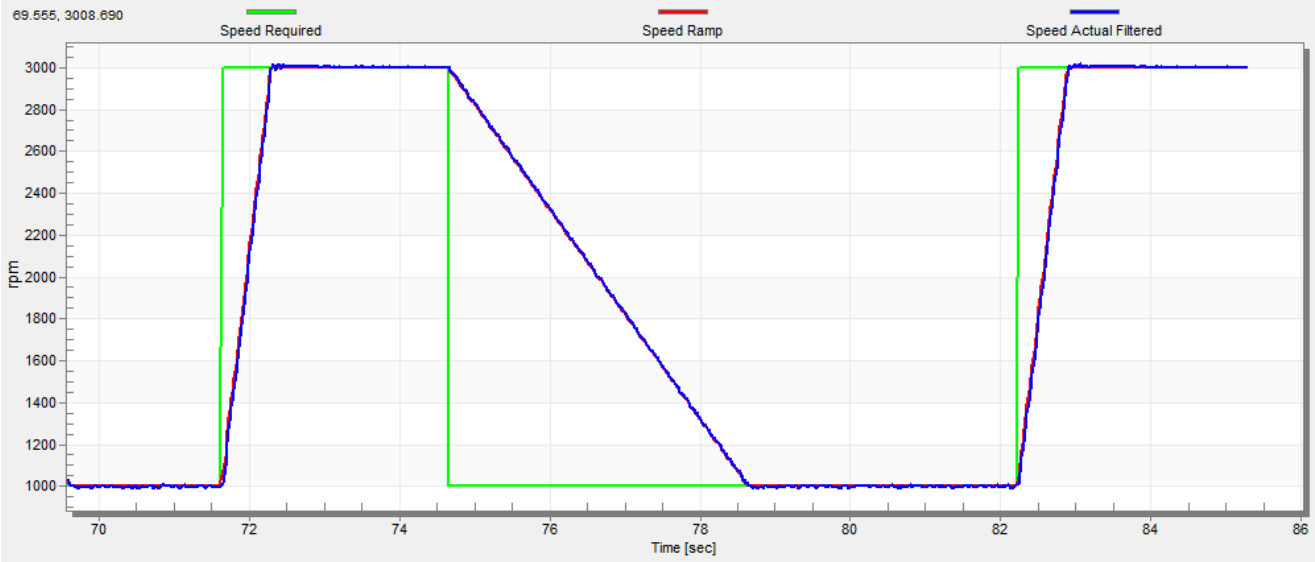


Figure 34. Speed profile

8.12.5 Open-loop startup

To tune the open-loop startup parameters, follow these steps:

1. The startup process can be tuned by a set of parameters located in the "Sensorless" tab. Two of them (ramp increment and current) are accessible in both tuning modes. The startup tuning can be processed in all control modes besides the scalar control. Setting the optimal values results in a proper motor startup. An example startup state of low-dynamic drives (fans, pumps) is shown in Figure 35.
2. Select the "Startup" recorder from the FreeMASTER project tree.
3. Set the startup ramp increment to a higher value than the speed-loop ramp increment.
4. Set the startup current according to the required startup torque. For drives such as fans or pumps, the startup torque is not very high and can be set to 15 % of the nominal current.
5. Set the required merging speed. When the open-loop and estimated position merging starts, the threshold is mostly set in the range of 5 % ~ 10 % of the nominal speed.
6. Set the merging coefficient. In the position-merging process duration, 100 % corresponds to one electrical revolution. The higher the value, the faster the merge. Values close to 1 % are set for drives where a high startup torque and smooth transitions between the open loop and the closed loop are required.
7. To apply the changes to the MCU, click the "Update Target" button.
8. Select "SPEED_FOC" in the "M1 MCAT Control" variable.
9. Set the required speed higher than the merging speed.
10. Check the startup response in the recorder.
11. Tune the startup parameters until you achieve an optimal response.
12. If the rotor does not start running, increase the startup current.
13. If the merging process fails (the rotor is stuck or stopped), decrease the startup ramp increment, increase the merging speed, and set the merging coefficient to 5 %.

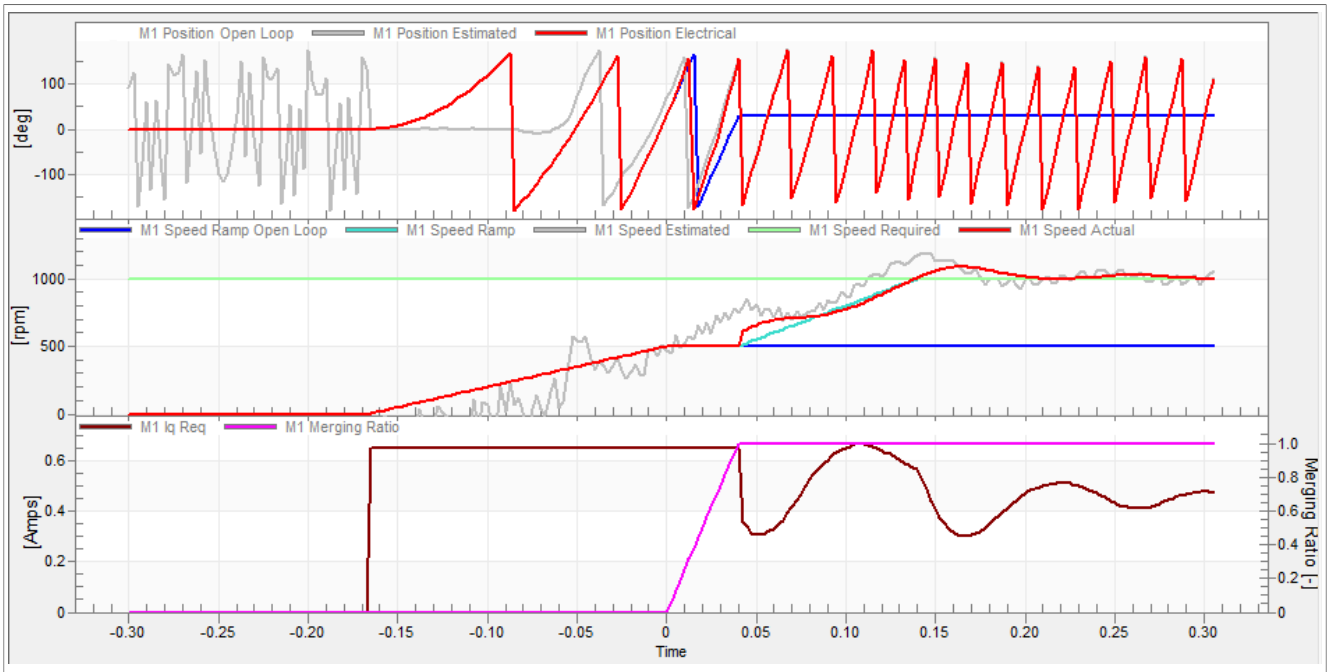


Figure 35. Motor startup

8.12.6 BEMF observer tuning

You can tune the bandwidth and attenuation parameters of the BEMF and tracking observers. To tune the bandwidth and attenuation parameters, follow these steps:

1. Navigate to the "Sensorless" MCAT tab.
2. Set the required bandwidth and attenuation of the BEMF observer. The bandwidth is typically set to a value close to the current loop bandwidth.
3. Set the required bandwidth and attenuation of the tracking observer. The bandwidth is typically set in the range of 10 Hz - 20 Hz for most low-dynamic drives (fans, pumps).
4. To apply the changes to the MCU, click the "Update target" button.
5. Select the "Observer" recorder from the FreeMASTER project tree and check the observer response in the "Observer" recorder.

8.12.7 Speed PI controller tuning

The motor speed control loop is a first-order function with a mechanical time constant that depends on the motor inertia and friction. If the mechanical constant is available, the PI controller constants can be tuned using the loop bandwidth and attenuation. Otherwise, the manual tuning of the P and I portions of the speed controllers is available to obtain the required speed response (see Figure 36). There are dozens of approaches to tune the PI controller constants. To set and tune the speed PI controller for a PMSM motor, follow these steps:

1. Select the "Speed Controller" option from the FreeMASTER project tree.
2. Select the "Speed loop" tab.
3. Check the "Manual Constant Tuning" option - that is, the "Bandwidth" and "Attenuation" fields are disabled and the "SL_Kp" and "SL_Ki" fields are enabled.
4. Tune the proportional gain:
   - Set the "SL_Ki" integral gain to 0.
   - Set the speed ramp to 1000 RPM/sec (or higher).

- Run the motor at a convenient speed (about 30 % of the nominal speed).
- Set a step in the required speed to 40 % of N_{nom}.
- Adjust the proportional gain "SL_Kp" until the system responds to the required value properly and without any oscillations or excessive overshoot:
  - If the "SL_Kp" field is set low, the system response is slow.
  - If the "SL_Kp" field is set high, the system response is tighter.
  - When the "SL_Ki" field is 0, the system most probably does not achieve the required speed.
  - To apply the changes to the MCU, click the "Update Target" button.

5. Tune the integral gain:
   - Increase the "SL_Ki" field slowly to minimize the difference between the required and actual speeds to 0.
   - Adjust the "SL_Ki" field such that you do not see any oscillation or large overshoot of the actual speed value while the required speed step is applied.
   - To apply the changes to the MCU, click the "Update target" button.
6. Tune the loop bandwidth and attenuation until the required response is received. The example waveforms with the correct and incorrect settings of the speed-loop parameters are shown in the following figures:
   - The "SL_Ki" value is low and the "Speed Actual Filtered" does not achieve the "Speed Ramp".

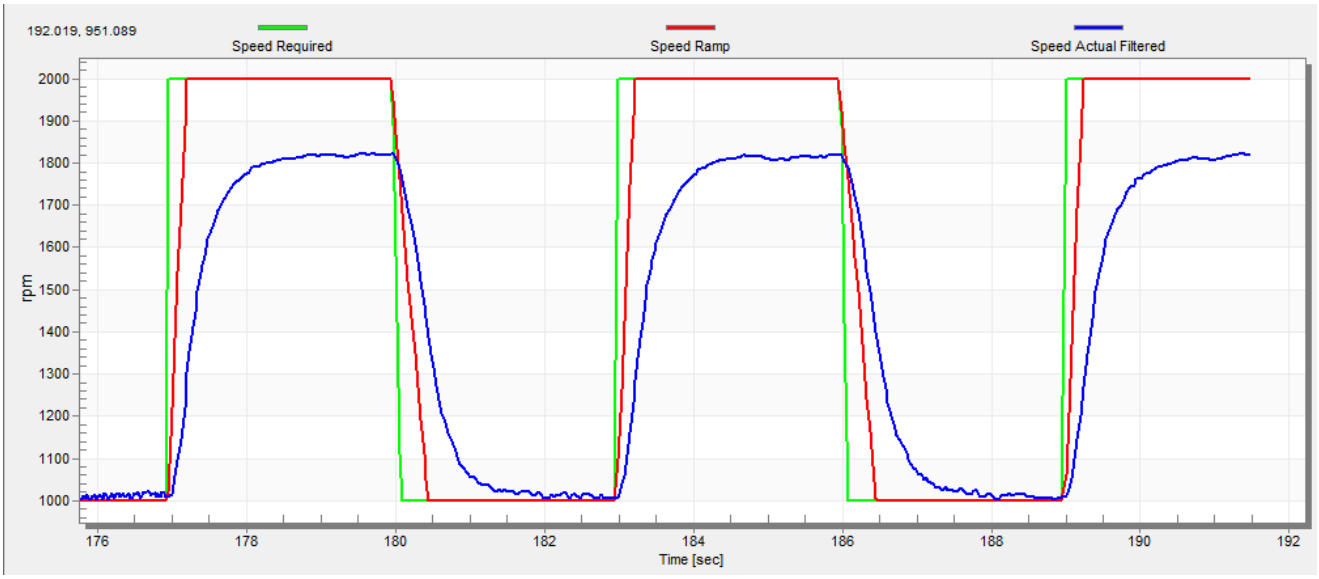


Figure 36. Speed controller response - "SL_Ki" value is low, "Speed Ramp" is not achieved

- The "SL_Kp" value is low, the "Speed Actual Filtered" greatly overshoots, and the long settling time is unwanted.

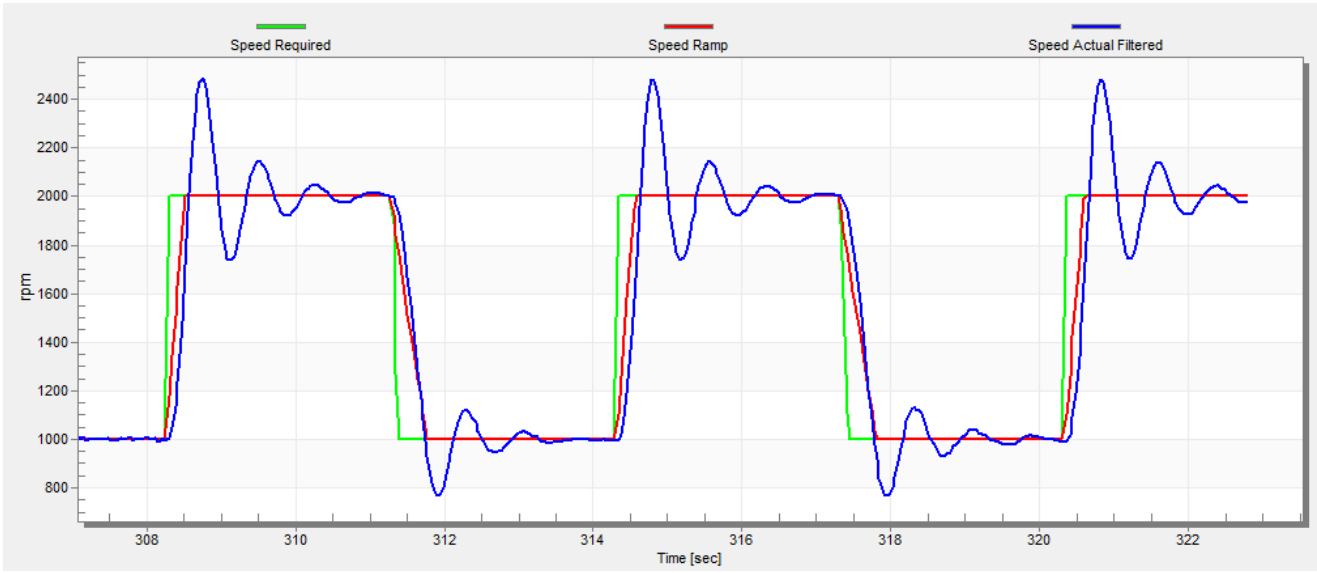


Figure 37. Speed controller response - "SL_Kp" value is low, "Speed Actual Filtered" greatly overshoots

- The speed-loop response has a small overshoot and the "Speed Actual Filtered" settling time is sufficient. Such response is considered optimal.

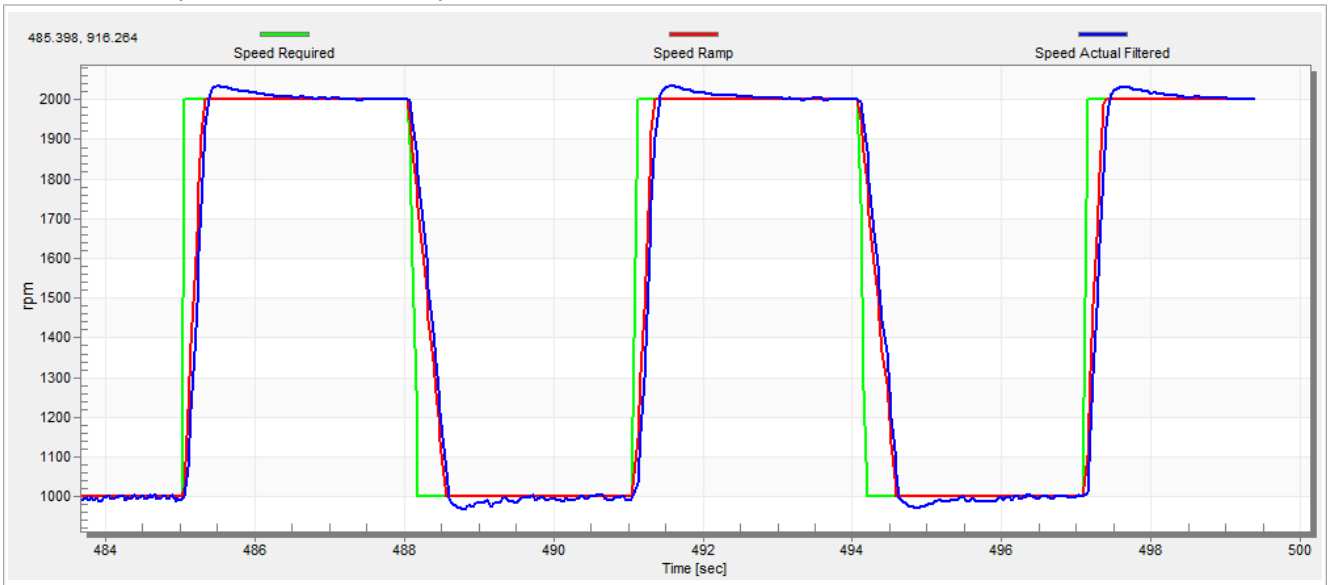


Figure 38. Speed controller response - speed loop response with a small overshoot

8.12.8 Position P controller tuning

You can tune the position-control loop using "Bandwidth F0" and "Attenuation ξ" in the "Servo control" tab. You can use a proportional controller to unpretend the position-control systems. The key for the optimal position response is a proper value of the controller, which multiplies the error by the proportional gain (Kp) to get the controller output. Use an encoder sensor for the position control to work. The following steps provide an example of how to set the position P controller for a PMSM:

1. Select the "Position Controller" scope in the "Position Control" tab in the FreeMASTER project tree.
2. Tune the proportional gain by "Bandwidth F0" and "Attenuation ξ":
   - Set the preferred value of "Bandwidth F0" and "Attenuation ξ". The MCAT automatically computes the position controller gain and feed-forward constants.
   - Select the position control, and set the required position in the "M1 Position Required" variable (for example, 10 revolutions).
   - Select the "Position Controller" scope and watch the actual position response.
3. Repeat the previous steps until you achieve the required position response.

The "PL_Kp" value is low and the actual position response on the required position is very slow.

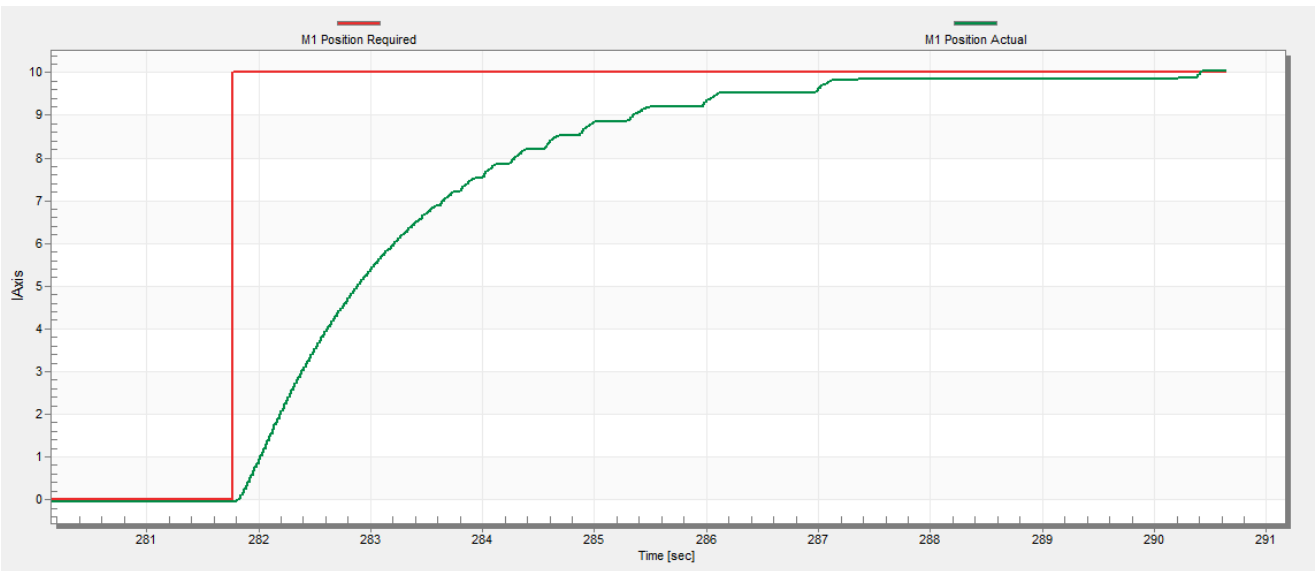


Figure 39. Position controller response - "PL_Kp" value is low, the actual position response is very slow

The "PL_Kp" value is too high and the actual position overshoots the required position.

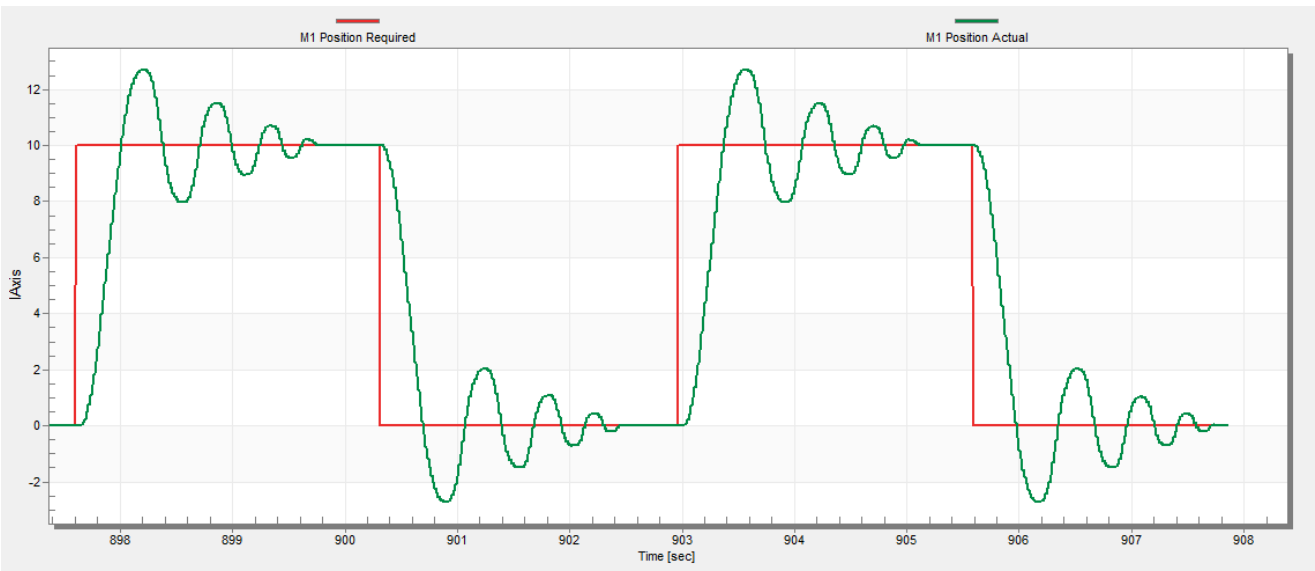


Figure 40. Position controller response - "PL_Kp" value is too high and the actual position overshoots

The "PL_Kp" value and the actual position response are optimal.

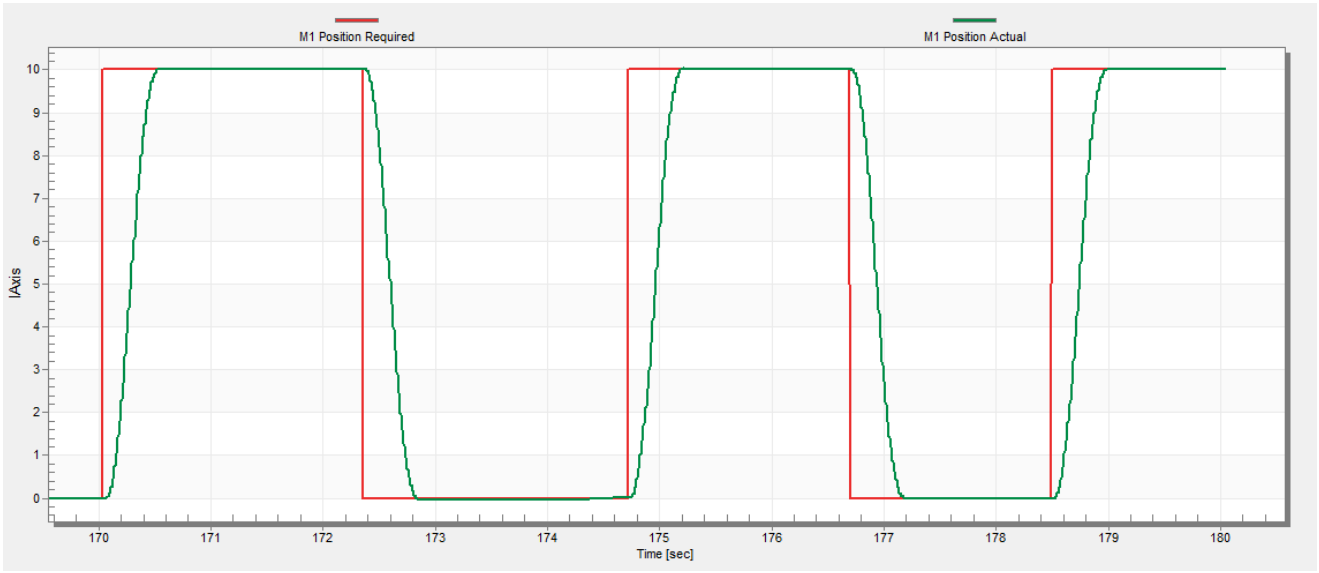


Figure 41. Position controller response - the actual position response is good

9 Conclusion

This application note describes the implementation of the sensored and sensorless field-oriented control of a three-phase PMSM. The motor control software is implemented on the NXP FRDM-MCXA346 board with the FRDM-MC-LVPMSM NXP Freedom development platform. The hardware-dependent part of the control software is described in Section 2. The motor-control application timing, and the peripheral initialization are described in Section 3. The motor user interface and remote control using FreeMASTER are described in Section 7. The motor parameters identification theory and the identification algorithms are described in Section 8.9.

10 Acronyms and abbreviations

Table 21 lists the acronyms and abbreviations used in this document.

Table 21. Acronyms and abbreviations

Acronym	Meaning
ADC	Analog-to-Digital Converter
ACIM	Asynchronous Induction Motor
ADC_ETC	ADC External Trigger Control
AN	Application Note
BLDC	Brushless DC motor
CCM	Clock Controller Module
CPU	Central Processing Unit
DC	Direct Current
DRM	Design Reference Manual
ENC	Encoder
FOC	Field-Oriented Control

Table 21. Acronyms and abbreviations...*continued*

Acronym	Meaning
GPIO	General-Purpose Input/Output
LPIT	Low-Power Periodic Interrupt Timer
LPUART	Low-Power Universal Asynchronous Receiver/Transmitter
MCAT	Motor Control Application Tuning tool
MCDRV	Motor Control Peripheral Drivers
MCU	Microcontroller Unit
PDB	Programmable Delay Block
PI	Proportional Integral Controller
PLL	Phase-Locked Loop
PMSM	Permanent Magnet Synchronous Motor
PWM	Pulse-Width Modulation
QD	Quadrature Decoder
TMR	Quad Timer
USB	Universal Serial Bus
XBAR	Inter-Peripheral Crossbar Switch
IOPAMP	Internal Operational Amplifier

11 References

These references are available on NXP webpage:

- *Sensorless PMSM Field-Oriented Control* (document DRM148)
- *Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM* (document AN4642)
- MCX General-Purpose MCUs

12 Useful links

- MCUXpresso SDK for Motor Control
- Motor Control Application Tuning (MCAT) Tool
- FRDM-MC-PMSM Freedom Development Platform
- MCUXpresso IDE - Importing MCUXpresso SDK
- MCUXpresso Config Tool
- MCUXpresso SDK Builder (SDK examples in several IDEs)
- Model-Based Design Toolbox (MBDT)

13 Revision history

This section summarizes the changes done to the document since the initial release.

Table 22. Revision history

Document ID	Release date	Description
UG10248 v.2.0	6 February 2026	Mechanical identification algorithm updated. MID can be controlled via MCAT tab.
UG10248 v.1.0	3 July 2025	Initial release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents