

MCUXpresso SDK Field-Oriented Control (FOC) of 3-Phase PMSM and BLDC motors



Contents

Chapter 1 Introduction.....	3
Chapter 2 Hardware setup.....	4
Chapter 3 RT crossover processors features and peripheral settings.....	12
Chapter 4 Project file and IDE workspace structure.....	16
Chapter 5 Tools.....	18
Chapter 6 Motor-control peripheral initialization.....	19
Chapter 7 User interface.....	21
Chapter 8 Remote control using FreeMASTER.....	22
Chapter 9 Conclusion.....	54
Chapter 10 Acronyms and abbreviations.....	55
Chapter 11 References.....	56
Chapter 12 Useful links.....	57
Chapter 13 Revision history.....	58

Chapter 1

Introduction

This user's guide describes the implementation of the sensorless motor-control software for 3-phase Permanent Magnet Synchronous Motors (PMSM), including the motor parameters identification algorithm, on the NXP i.MX RT series crossover processors. The sensorless control software and the PMSM control theory in general are described in design reference manual [DRM148 Sensorless PMSM Field-Oriented Control \(FOC\)](#). The NXP Freedom ([FRDM-MC-LVPMSM](#)) is used as the hardware platform for the PMSM control reference solution. The hardware-dependent part of the sensorless and sensed control software, including a detailed peripheral setup and the Motor Control (MC) peripheral drivers, are addressed as well. The motor parameters identification theory and algorithms are presented in this document. The last part of the document introduces and explains the user interface represented by the Motor Control Application Tuning ([MCAT](#)) page based on the FreeMASTER run-time debugging tool. These tools provide a simple and user-friendly way for the motor parameter identification, algorithm tuning, software control, debugging, and diagnostics.

This document describes how to run and control the Permanent Magnet Synchronous Motor (PMSM) project using [i.MX RT Series Crossover Processors](#) with the Freedom development board. The software provides sensorless/sensed field-oriented vector position, speed, torque, and scalar control. You can control the application using the board buttons or via FreeMASTER. The motor identification and application tuning is done using the MCAT tool integrated in the FreeMASTER page. The required software, hardware setup, jumper settings, project arrangement, and user interface are described in the following sections.

Available motor control examples, supported motors and possible control methods are listed in [Table 1](#). More detailed description of the examples will be discussed in [Available motor control examples](#) chapter.

Table 1. Available examples and control methods

Example	Supported motor	Possible control methods in SDK example				
		Scalar & Voltage	Current FOC (Torque)	Sensorless Speed FOC	Sensed Speed FOC	Sensed Position FOC
pmsm_enc	Linux 45ZWN24-40 (default motor)	✓	✓	✓	N/A	N/A
	Teknic M-2310P (with ENC)	✓	✓	✓	✓	✓

NOTE

The latest documentation for the motor control SDK is available on http://www.nxp.com/motorcontrol_pmsm.

Chapter 2

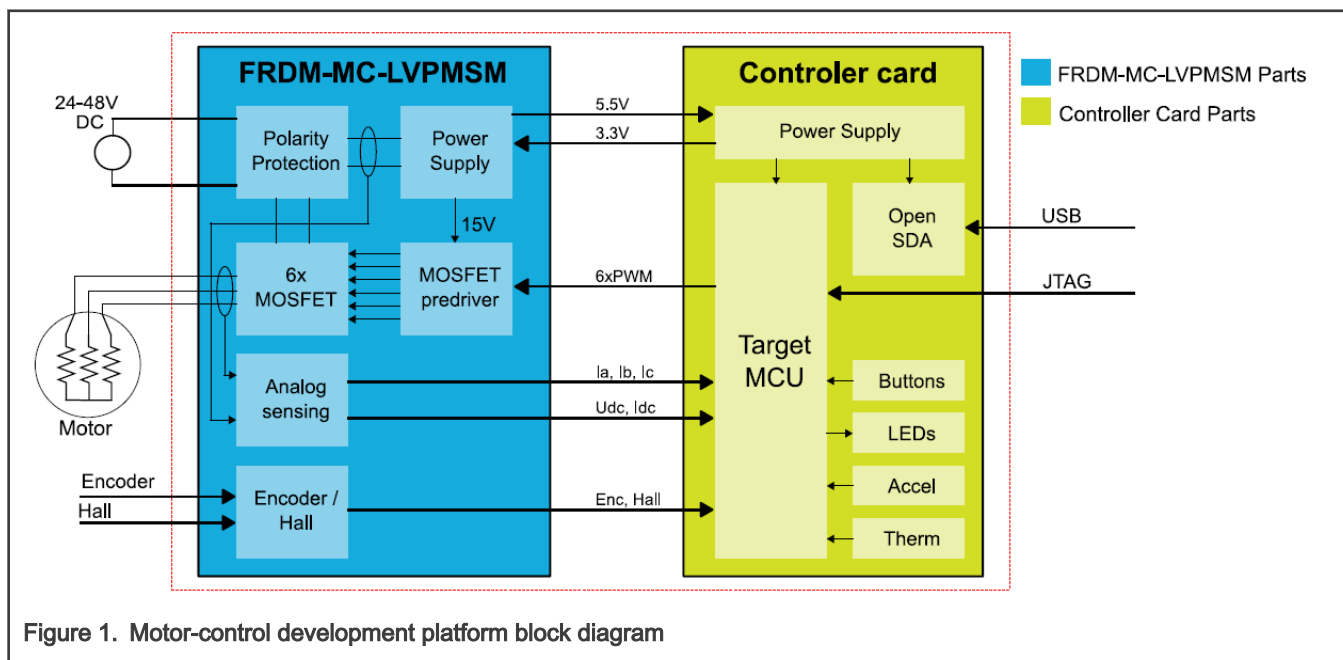
Hardware setup

The PMSM Field-Oriented Control (FOC) application runs on the FRDM-MC-LVPMSM development platform with the i.MX RT1170-EVK development tools, in combination with the Teknic M-2310P or Linux 45ZWN24-40 permanent magnet synchronous motors.

2.1 FRDM-MC-LVPMSM

This evaluation board, in a shield form factor, effectively turns an NXP Freedom development board or an evaluation board into a complete motor-control reference design, 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, 3-phase Permanent Magnet Synchronous Motor (PMSM) Freedom development platform board has the power supply input voltage of 24-48 VDC with a 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 3-phase bridge inverter (six MOSFETs) and a 3-phase MOSFET gate driver. The analog quantities (such as the 3-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). The block diagram of this complete NXP motor-control development kit is shown in [Figure 1](#).



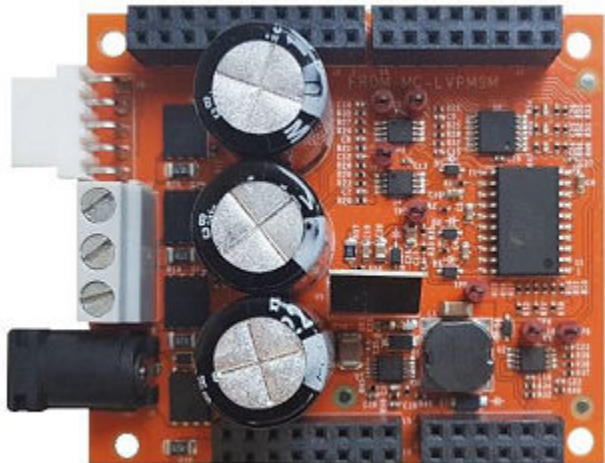


Figure 2. FRDM-MC-LVPMSM

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

2.2 Linux 45ZWN24-40 motor

The Linux 45ZWN24-40 motor is a low-voltage 3-phase permanent-magnet motor with hall sensor used in PMSM applications. The motor parameters are listed in Table 2.

Table 2. Linux 45ZWN24-40 motor parameters

Characteristic	Symbol	Value	Units
Rated voltage	Vt	24	V
Rated speed	-	4000	RPM
Rated torque	T	0.0924	Nm
Rated power	P	40	W
Continuous current	Ics	2.34	A
Number of pole-pairs	pp	2	-



The motor has two types of connectors (cables). The first cable has three wires and is designated to power 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.

2.3 Teknic M-2310P motor

The Teknic M-2310P-LN-04K motor is a low-voltage 3-phase permanent-magnet motor used in PMSM applications. The motor has two feedback sensors (hall and encoder). For information on the wiring of feedback sensors, see the datasheet on the manufacturer web page. The motor parameters are listed in [Table 3](#).

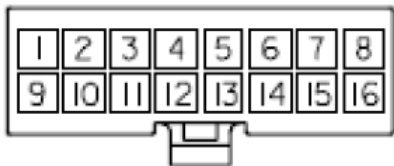
Table 3. Teknic M-2310P motor parameters

Characteristic	Symbol	Value	Units
Rated voltage	Vt	40	V
Rated speed	-	6000	RPM
Rated torque	T	0.247	Nm
Rated power	P	170	W
Continuous current	Ics	7.1	A
Number of pole-pairs	pp	4	-



Figure 4. Teknic M-2310P permanent magnet synchronous motor

For the sensorless control mode, you need only the power input wires. If used with the hall or encoder sensors, connect also 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 I
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 5. Teknic motor connector type 1

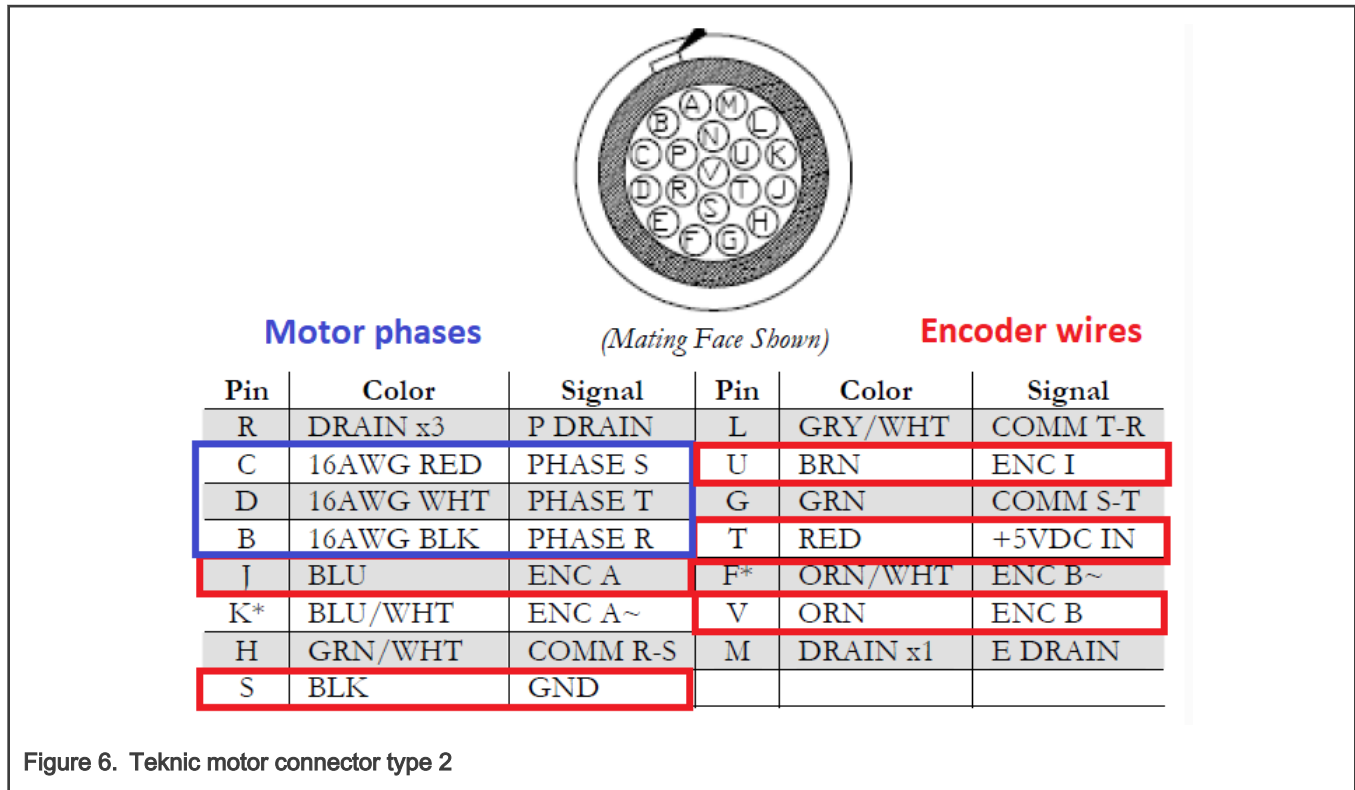


Figure 6. Teknic motor connector type 2

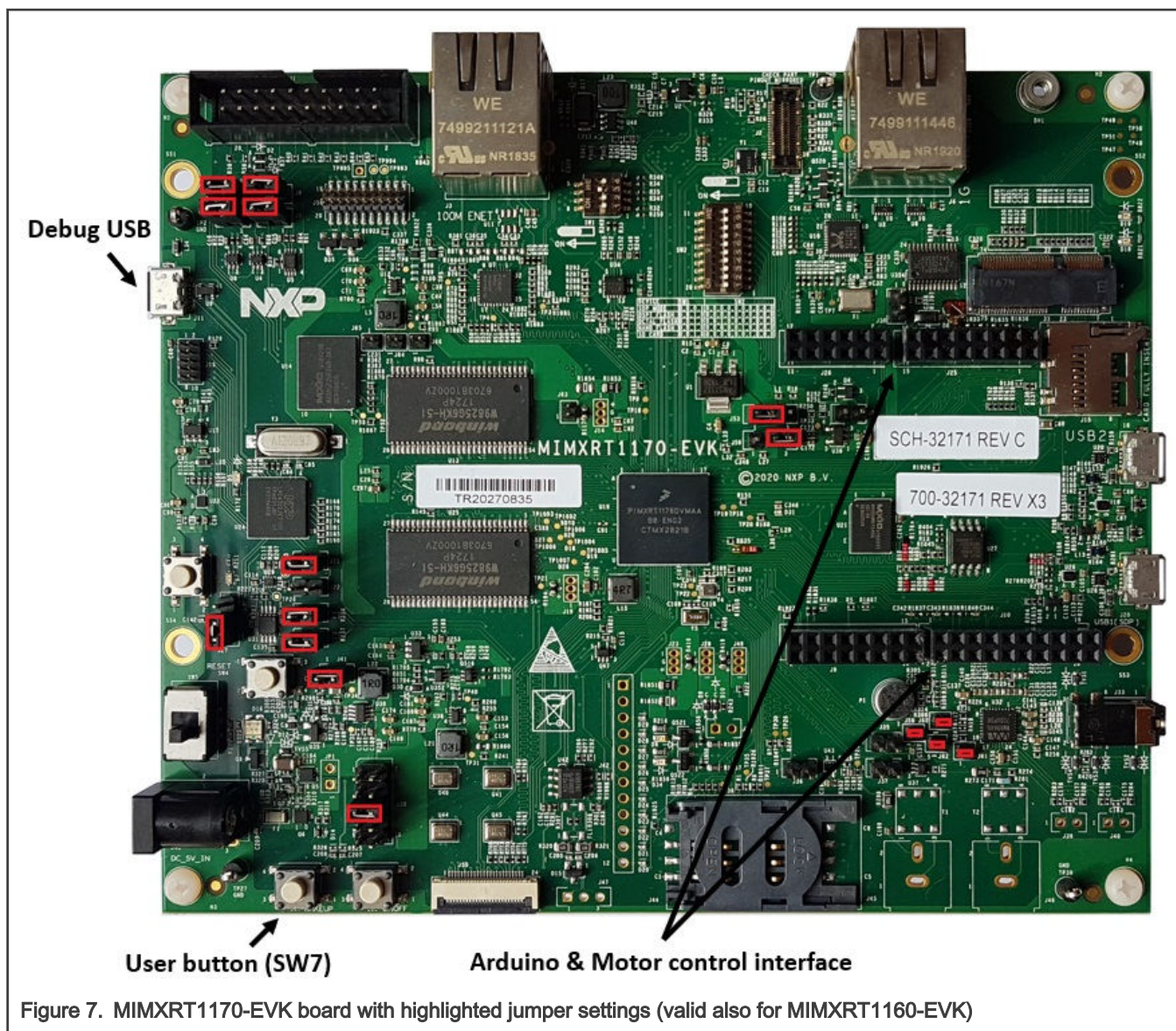
2.4 i.MX RT1160/1170 Evaluation Kit

The [MIMXRT1160-EVK](#) and [MIMXRT1170-EVK](#) are two-layer low-cost through-hole USB-powered PCBs. At its heart lies the i.MX RT11xx crossover MCU. The dual core i.MX RT1160 runs on the Cortex-M7 core at 600 MHz and Arm Cortex-M4 at 240 MHz and i.MX RT1170 runs on the Cortex-M7 core at 1 GHz and Arm Cortex-M4 at 400 MHz, while providing best-in-class security.

Table 4. MIMXRT1160-EVK and MIMXRT1170-EVK jumper settings

Jumper	Setting	Jumper	Setting	Jumper	Setting
J5	1-2	J31	1-2	J59	1-2
J6	1-2	J32	1-2	J60	1-2
J7	1-2	J38	5-6	J61	1-2
J8	1-2	J41	1-2	J62	1-2
J21	1-2	J53	1-2		
J27	2-3	J56	2-3		

All others jumpers are open.



For a correct connection, the motor-control application requires remove and solder some resistors. Please, solder resistor R1841 and R1842 for encoder and remove resistors R188 and R193 for right ADC measuring. These resistors are located on the top of the EVK board. For more details, see the schematic.

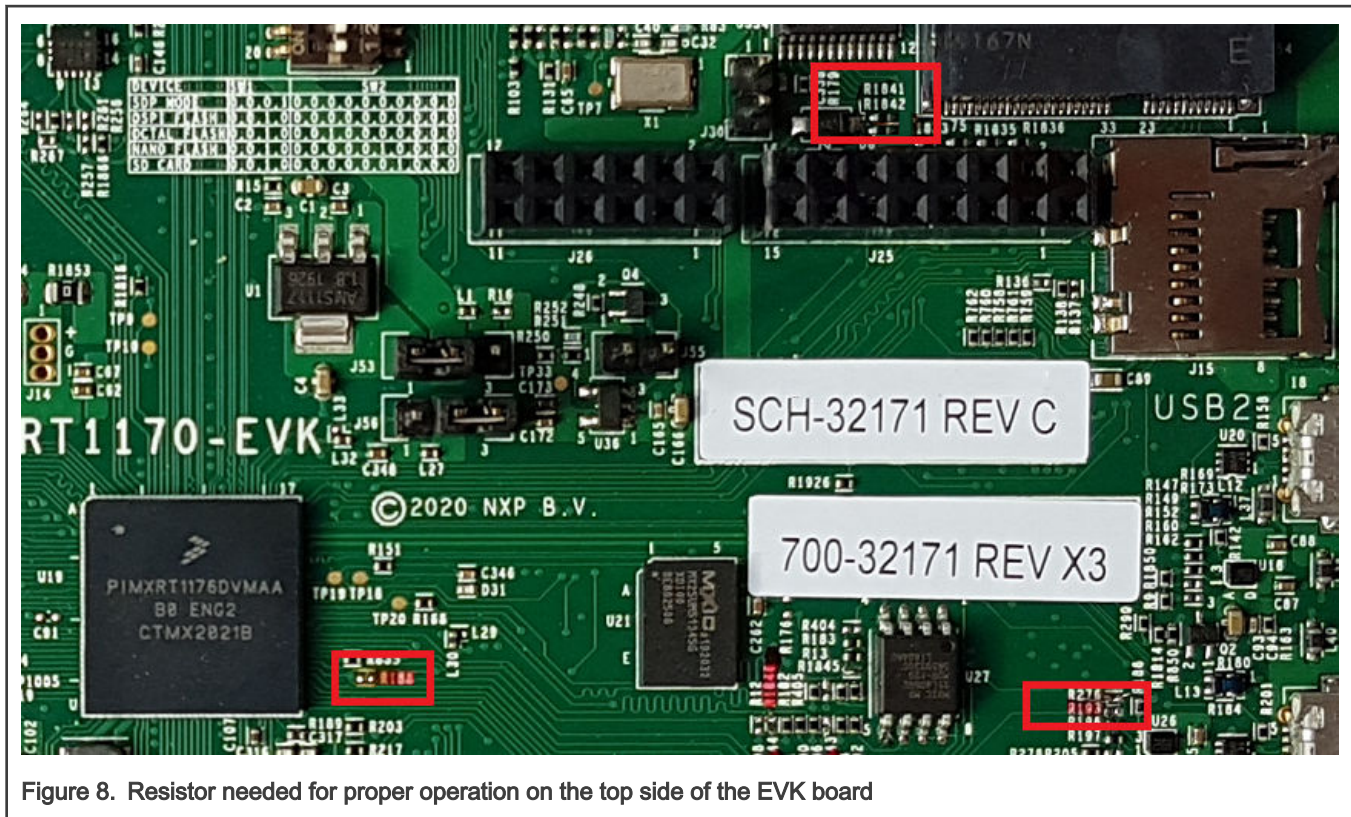


Figure 8. Resistor needed for proper operation on the top side of the EVK board

For more information about the MIMXRT1160-EVK or MIMXRT1170-EVK hardware (processor, peripherals, and so on), see the [MIMXRT1160 EVK Board Hardware User's Guide](#) or [MIMXRT1170 EVK Board Hardware User's Guide](#).

Hardware assembling

1. Connect the FRDM-MC-LVPMSM shield on top of the MIMXRT1160-EVK or MIMXRT1170-EVK board (there is only one possible option).

NOTE

Watch out for unwanted connections between bottom of FRDM-MC-PMSM and jumpers on top of MIMXRT11xx-EVK.

2. Connect the 3-phase motor wires to the screw terminals (J7) on the Freedom PMSM power stage.
3. On the top of FRDM-MC-PMSM shield connect by wires following pins:

Table 5. MIMXRT1170-EVK pin assignment

FRDM-MC-LVPMSM	Connection	On the MIMXRT1160/1170-EVK is pin connected to
CUR_A	J2, 1 <-> J4, 2	GPIO_AD_10
CUR_B	J2, 3 <-> J4, 6	GPIO_AD_12
CUR_C	J2, 5 <-> J4, 8	GPIO_AD_13
VOLT_DCB	J2, 7 <-> J4, 4	GPIO_AD_11
CUR_DCB	J2, 9 <-> J2, 8	GPIO_AD_30

4. Plug the USB cable from the USB host to the Debug USB connector (J11) on the EVK board.
5. Plug the 24-V DC power supply to the DC power connector on the Freedom PMSM power stage.

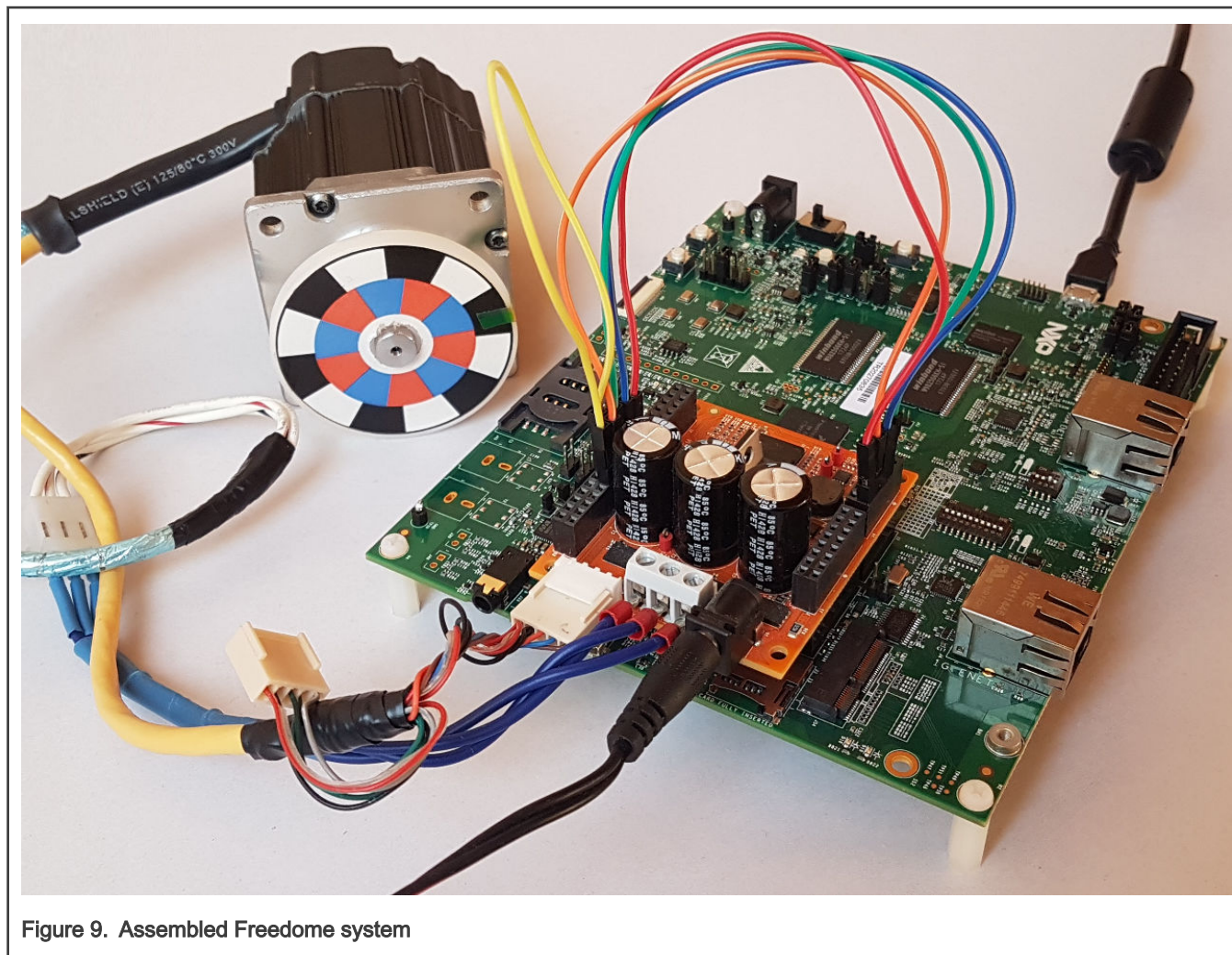


Figure 9. Assembled Freedom system

Chapter 3

RT crossover processors features and peripheral settings

This chapter describes the peripheral settings and application timing. The i.MX RT1xxx is a new processor family featuring NXP's advanced implementation of the Arm Cortex-M7 core.

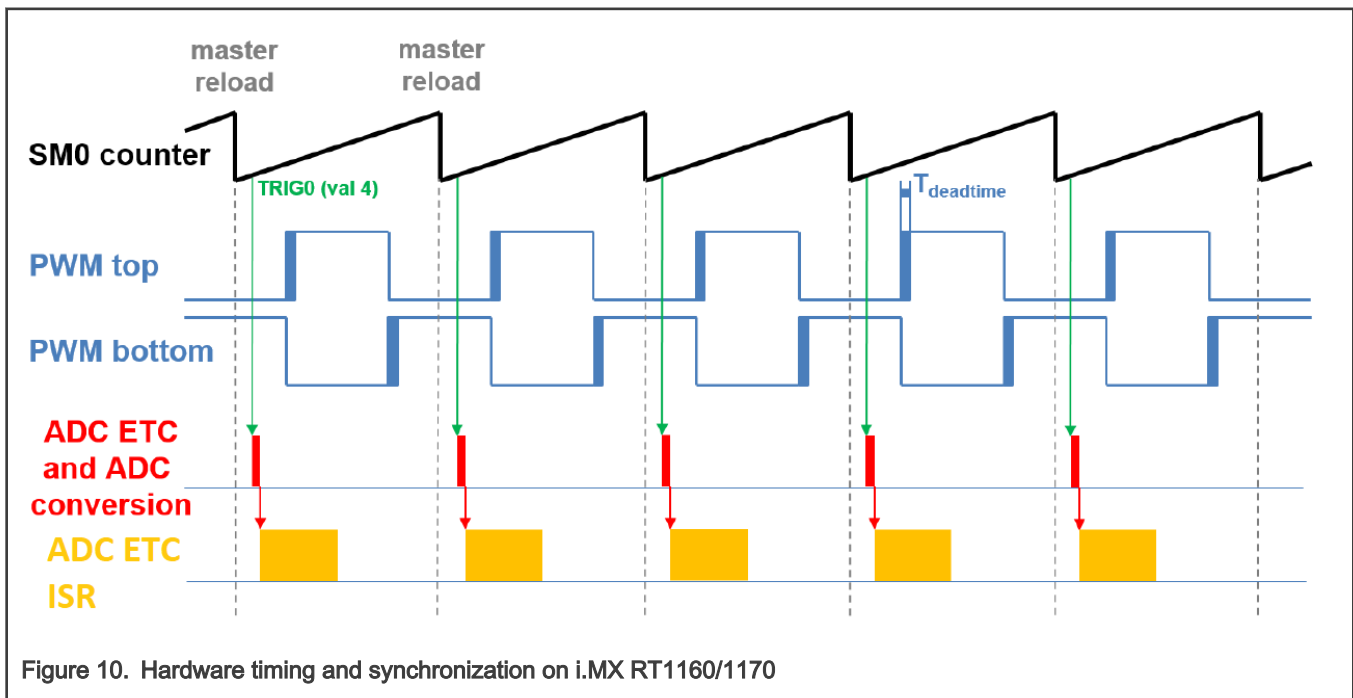
3.1 i.MX RT1160/1170

The **i.MX RT1160/1170** crossover MCUs are setting speed records at 1 GHz. This ground-breaking family combines superior computing power and multiple media capabilities with ease of use and real-time functionality. The i.MX RT1160/1170 MCU offers support over a wide temperature range and is qualified for consumer, industrial and automotive markets.

For more information see [i.MX RT1160](#) or [i.MX RT1170](#) Crossover MCU Family web pages.

3.1.1 RT1160/1170 - 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 frequency 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.



- The top signal shows the eFlexPWM counter (SM0 counter). The dead time is emphasized at the PWM top and PWM bottom signals. The SM0 submodule generates the master reload at every opportunity.
- The SM0 generates trigger 0 (when the counter counts to a value equal to the TRIG4 value) for the ADC_ETC (ADC External Trigger Control) with a delay of approximately $T_{\text{deadtime}}/2$. This delay ensures correct current sampling at the duty cycles close to 100 %.
- ADC_ETC starts the ADC conversion.
- When the ADC conversion is completed, the ADC_ETC ISR (ADC_ETC interrupt) is entered. The FOC calculation is done in this interrupt.

3.1.2 RT1160/1170 - Peripheral settings

This section describes the peripherals used for the motor control. On i.MX RT1160/1070, there are three submodules from the enhanced FlexPWM (eFlexPWM) used for 6-channel PWM generation and two 12-bit ADCs for the phase currents and DC-bus voltage measurement. The eFlexPWM and ADC are synchronized via submodule 0 from the eFlexPWM. The following settings are located in the *mc_periph_init.c* and *peripherals.c* files and their header files.

PWM generation - PWM1

- Six channels from three submodules are used for the 3-phase PWM generation. Submodule 0 generates the master reload at event every n^{th} opportunity, depending on the user-defined macro `M1_FOC_FREQ_VS_PWM_FREQ`.
- Submodules 1 and 2 get their clocks from submodule 0.
- The counters at submodules 1 and 2 are synchronized with the master reload signal from submodule 0.
- Submodule 0 is used for synchronization with ADC_ETC. The submodule generates the output trigger after the PWM reload, when the counter counts to VAL4.
- Fault mode is enabled for channels A and B at submodules 0, 1, and 2 with automatic fault clearing (the PWM outputs are re-enabled at the first PWM reload after the fault input returns to zero).
- The PWM period (frequency) is determined by how long it takes the counter to count from INIT to VAL1. By default, $\text{INIT} = -\text{MODULO}/2 = -12000$ and $\text{VAL1} = \text{MODULO}/2 - 1 = 11999$. The eFlexPWM clock is 240 MHz so it takes 0.0001 s (10 KHz).
- Dead-time insertion is enabled. Define the dead-time length in the `M1_PWM_DEADTIME` macro.

ADC external trigger control - ADC_ETC

The ADC_ETC module enables multiple users to share the ADC modules in the Time Division Multiplexing (TDM) way. The external triggers can be brought from the Cross BAR (XBAR) or other sources. The ADC scan is started via ADC_ETC.

- Both ADCs have set their own trigger chains.
- The trigger chain length is set to 2. The back-to-back ADC trigger mode is enabled.
- The SyncMode is on. In the SyncMode, ADC1 and ADC2 are controlled by the same trigger source. The trigger source is the PWM submodule 0.
- After both ADCs conversion is completed, ADC_ETC interrupt is enabled and serves the FOC fast-loop algorithm.

Analog sensing - ADC1 and ADC2

ADC1 and ADC2 are used for the MC analog sensing of currents and DC-bus voltage.

- The ADCs operate as 12-bit with the single-ended conversion and hardware trigger selected. The ADCs are triggered from ADC_ETC by the trigger generated by the eFlexPWM.

Quadrature Decoder (QD) module

The QD module is used to sense the position and speed from the encoder sensor.

- The direction of counting is set in the `M1_POSPE_ENC_DIRECTION` macro.
- The modulo counting and the modulus counting roll-over/under to increment/decrement revolution counter are enabled.

Peripheral interconnection for - XBARA1

The crossbar is used to interconnect the trigger from the PWM to the ADC_ETC and to connect the encoder (connected to GPIO) to the QD.

- The `FLEXPWM2_PWM1_OUT_TRIG0_1` output trigger (generated by submodule 0) is connected to `ADC_ETC_XBAR0_TRIG0`.
- The encoder signal Phase A and Phase B are configured in *pinmux.c*.

Slow-loop interrupt generation - TMR1

The QuadTimer module TMR1 is used to generate the slow-loop interrupt.

- The slow loop is usually ten times slower than the fast loop. Therefore, the interrupt is generated after the counter counts from CNTR0 = 0 to COMP1 = $IPG\ CLK\ ROOT / (16U * Speed\ Loop\ Freq)$. The speed loop frequency is set in the M1_SPEED_LOOP_FREQ macro and equals 1000 Hz.
- An interrupt (which serves the slow-loop period) is enabled and generated at the reload event.

FreeMASTER communication—LPUART1

LPUART1 (Low-Power Universal Asynchronous Receiver and Transmitter) is used for the FreeMASTER communication between the MCU board and the PC.

- The baud rate is set to 115200 bit/s.
- The receiver and transmitter are both enabled.
- The other settings are set to default.

3.2 Available motor control examples

Following examples are available for the device specified in this document:

- pmsm_enc

Supported motors and possible control methods for each example are mentioned in the Introduction of this document.

pmsm_enc example

This example can be used for the sensor and sensorless motor control application both. Default motor configuration is tuned for the Linux 45ZWN24-40 motor. Motor identification is implemented in this example. Changing motor configuration is described below.

Change motor configuration

Motor control examples contains two or more configuration files: *m1_pmsm_appconfig.h*, *m2_pmsm_appconfig.h* etc. Each of them contains constants tuned for the selected motor (Linux 45ZWN24-40 or Teknic M-2310P in case of the Freedom development platform, Mige 60CST-MO1330 in case of the High-voltage platform). There are two ways, how to change motor configuration corresponding to the connected motor. The first way is following:

- In the project example folder, find configuration file which will be used for.
- Rename this configuration file to *m1_pmsm_appconfig.h*.
- Rebuild project and load the code to the MCU.

The second way how to change motor configuration is described in [MCAT FreeMASTER interface \(Motor Control Application Tuning\)](#).

3.3 RT family - CPU load and memory usage

The following information apply to the application built using one of the following IDE: MCUXpresso IDE, IAR or Keil MDK. The memory usage is calculated from the *.map* linker file, including the 4-KB FreeMASTER recorder buffer allocated in RAM. In the MCUXpresso IDE, the memory usage can be also seen after 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. In this case, it applies to the fast-loop frequency of 16 KHz and the slow-loop frequency of 1 KHz. The total CPU load is calculated using the following equations:

$$CPU_{fast} = cycles_{fast} \frac{f_{fast}}{f_{CPU}} 100 [\%]$$

$$CPU_{slow} = cycles_{slow} \frac{f_{slow}}{f_{CPU}} 100 [\%]$$

$$CPU_{total} = CPU_{fast} + CPU_{slow} [\%]$$

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 (16 KHz).

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 (1 KHz).

CPU_{total} - the total CPU load consumed by the motor control.

Table 6. Maximum CPU load (fast loop)

Device	Example	debug configuration	
		Speed Control	Position Control
i.MX RT1170	pmsm_enc	7.3%	6.8%

CPU load measured without defined RAM_RELOCATION macro. Measured CPU load apply to the application built using IAR IDE.

NOTE

The maximum CPU load is depending on executing functions from RAM or FLASH memory. Executing functions can be speeding up in RTCESL_cfg.h header file by using macro RAM_RELOCATION.

NOTE

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

Chapter 4

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 in a logical manner. The folder structure used in the IDE is different from the structure of the PMSM package installation, but it uses the same files. The different organization is chosen due to a better manipulation with folders and files in workplaces and due to the possibility to add or remove files and directories. The “*pack_motor_board*” project includes all the available functions and routines, MID functions, scalar and vector control of the motor, FOC control, and FreeMASTER MCAT project. This project serves for development and testing purposes.

4.1 PMSM project structure

The directory tree of the PMSM project is shown in below.

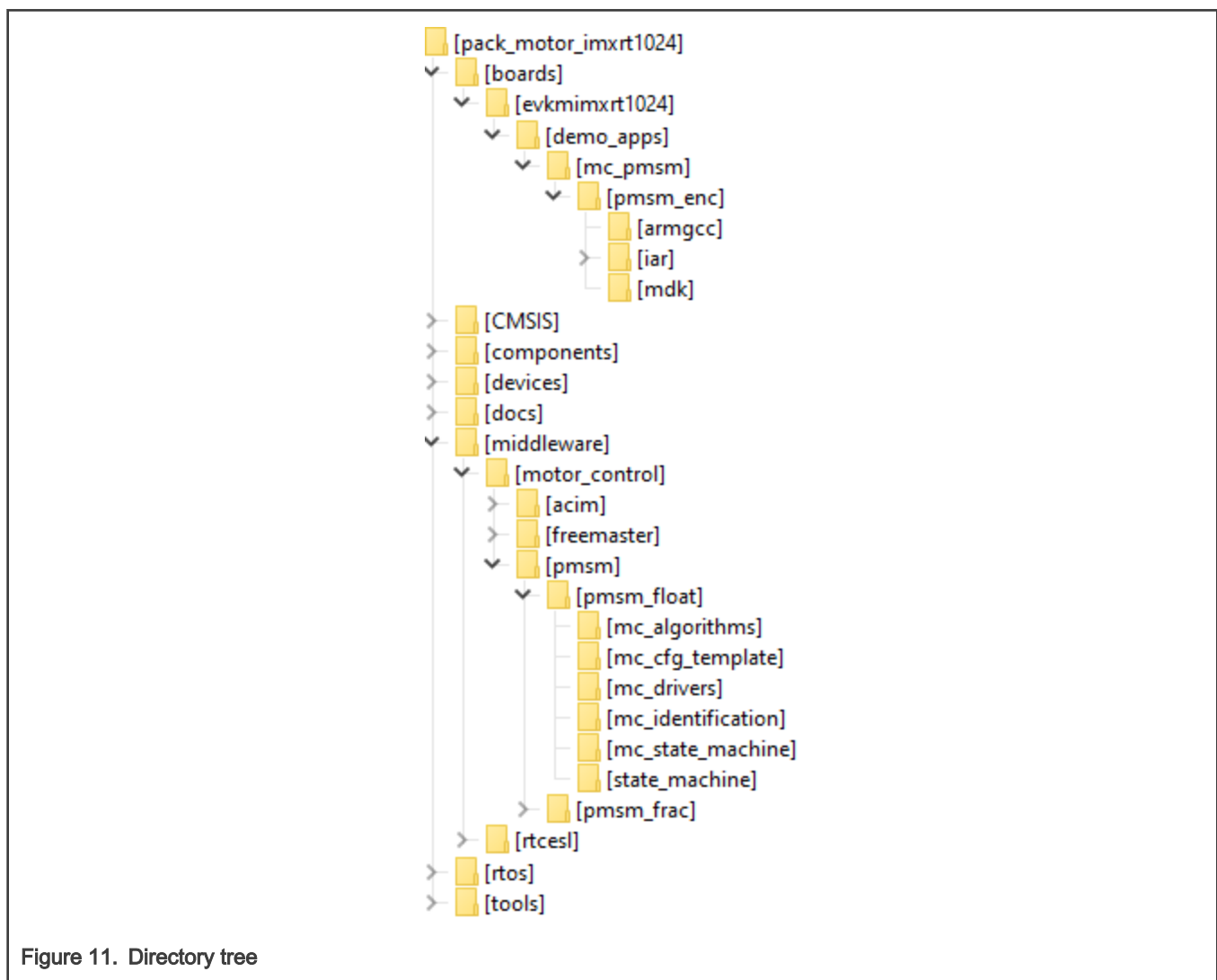


Figure 11. Directory tree

The main project folder *pack_motor_imxrt1xxx\boards\evkbimxrt1xxx\demo_apps\mc_pmsm\pmsm_enc* contains these 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*—contains 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 and .h*—port configuration files. It is recommended to generate these files in the pin tool.
- *peripherals.c and .h*—MCUXpresso Config Tool configuration files.

The main motor-control folder *pack_motor_imxrt10xx\middleware\motor_control* contains these subfolders:

- *pmsm*—contains main PMSM motor-control functions
- *freemaster*—contains the FreeMASTER project file *pmsm_float_enc.pmp*. 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_imxrt1xxx\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.

Chapter 5

Tools

Install the [FreeMASTER Run-Time Debugging Tool 3.1.2](#) and one of the following IDEs on your PC to run and control the PMSM application properly:

- [IAR Embedded Workbench IDE v9.20.2](#) or higher
- [MCUXpresso v11.5.0](#)
- [ARM-MDK - Keil µVision version 5.36](#)

For pin_mux.c, clock_config.c or peripherals.c modifications is recommended use [MCUXpresso Configuration Tool v11](#) or higher.

NOTE

For information on how to build and run the application in your IDE, see the *Getting Started with MCUXpresso SDK* document located in the `pack_motor_<board>/docs` folder or find the related documentation at [MCUXpresso SDK builder](#).

5.1 Compiler warnings

Warnings are diagnostic messages that report constructions that are not inherently erroneous and warn about potential runtime, logic, and performance errors. In some cases, warnings can be suspended and these warnings do not show during the compiling process. One of such special cases is the “unused function” warning, where the function is implemented in the source code with its body, but this function is not used. This case occurs when you implement the function as a supporting function for better usability, but you do not use the function for any special purposes for a while.

The IAR Embedded Workbench IDE suppresses these warnings:

- Pa082 - undefined behavior; the order of volatile accesses is not defined in this statement.
- Pa050 - non-native end of line sequence detected.

The Arm-MDK Keil µVision IDE suppresses these warnings:

- 6314 - No section matches pattern xxx.o (yy).

By default, there are no other warnings shown during the compiling process.

Chapter 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 several macros, which can be defined by the user:

- *M1_MCDRV_ADC_PERIPH_INIT*—this macro calls ADC peripheral initialization.
- *M1_MCDRV_PWM_PERIPH_INIT*—this macro calls PWM peripheral initialization.
- *M1_MCDRV_QD_ENC*—this macro calls 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 are used to 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, these API-serving ADC and PWM peripherals are available:

- The available APIs for the ADC are:
 - *mcdrv_adc_t*—MCDRV ADC structure data type.
 - *void M1_MCDRV_ADC_PERIPH_INIT()*—this function is by default called 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_ADC_GET(mcdrv_adc_t*)*—this function reads and calculates the actual values of the 3-phase currents, DC-bus voltage, and auxiliary quantity.

- The available APIs for the PWM are:
 - *mcdrv_pwm3ph_t*—MCDRV PWM structure data type.
 - *void M1_MCDRV_PWM_PERIPH_INIT*—this function is by default called during the PWM peripheral initialization procedure invoked by the *MCDRV_Init_M1()* function.
 - *void M1_MCDRV_PWM3PH_SET(mcdrv_pwm3ph_t*)*—this function updates the PWM phase duty cycles.
 - *void M1_MCDRV_PWM3PH_EN(mcdrv_pwm3ph_t*)*—this function enables all PWM channels.
 - *void M1_MCDRV_PWM3PH_DIS (mcdrv_pwm3ph_t*)*—this function disables all PWM channels.
 - *bool_t M1_MCDRV_PWM3PH_FLT_GET(mcdrv_pwm3ph_t*)*—this function returns the state of the over-current fault flags and automatically clears the flags (if set). This function returns true when an over-current event occurs. Otherwise, it returns false.
- The available APIs for the quadrature encoder are:
 - *mcdrv_qd_enc_t*—MCDRV QD structure data type.
 - *void M1_MCDRV_QD_PERIPH_INIT()*—this function is by default called during the QD peripheral initialization procedure invoked by the *MCDRV_Init_M1()* function.
 - *void M1_MCDRV_QD_GET(mcdrv_qd_enc_t*)*—this function returns the actual position and speed.
 - *void M1_MCDRV_QD_SET_DIRECTION(mcdrv_qd_enc_t*)*—this function sets the direction of the quadrature encoder.
 - *void M1_MCDRV_QD_CLEAR(mcdrv_qd_enc_t*)*—this function clears the internal variables and decoder counter.

Chapter 7

User interface

The application contains the demo mode to demonstrate motor rotation. You can operate it either using the user button, or using FreeMASTER. The NXP EVK 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. Data is transferred between the PC and the embedded application via the serial interface. This interface is provided by the CMSIS-DAP debugger included in the boards.

The application can be controlled using these two interfaces:

- The button on the MIMXRT1xxx-EVK development board (controlling the demo mode):
 - MIMXRT1170-EVK - SW7
- Remote control using FreeMASTER (chapter [Remote control using FreeMASTER](#)):
 - Using the Motor Control Application Tuning (MCAT) interface.
 - Setting a variable in the FreeMASTER Variable Watch.

If you are using your own motor (different from the default motors), make sure to identify all motor parameters. The automated parameter identification is described in the following sections.

Chapter 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 FreeMASTER 3.0 at www.nxp.com/freemaster. To run the FreeMASTER application including the MCAT tool, double-click the *pmsm_float_enc.pmp* file located in the *pack_motor_imxrt1xxx\middleware\motor_control\freemaster* folder. The FreeMASTER application starts and the environment is created automatically, as defined in the *.pmp file.

NOTE

In MCUXpresso can be FreeMASTER application run directly from IDE in *motor_control/freemaster* folder

8.1 Establishing FreeMASTER communication

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

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

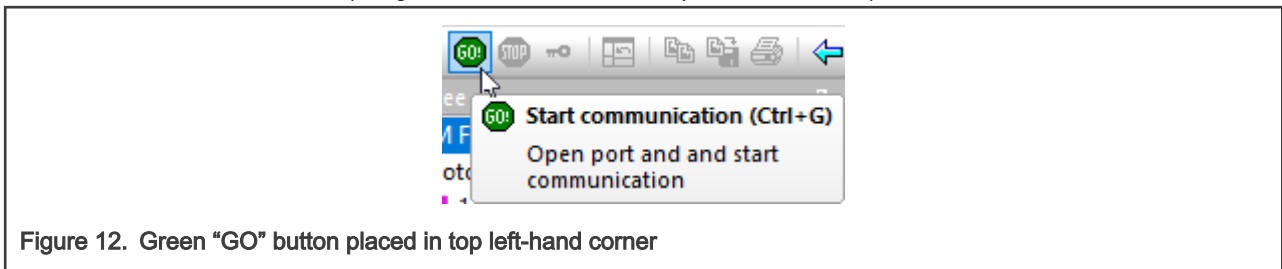


Figure 12. Green “GO” button placed in top left-hand corner

4. If the communication is established successfully, the FreeMASTER communication status in the bottom right-hand corner changes from “Not connected” to “RS232 UART Communication; COMxx; speed=115200”. Otherwise, the FreeMASTER warning popup window appears.



Figure 13. FreeMASTER—communication is established successfully

5. Press *F5* to reload the MCAT HTML page and check the App ID.
6. Control the PMSM motor by writing to a control variables in a 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 bps.

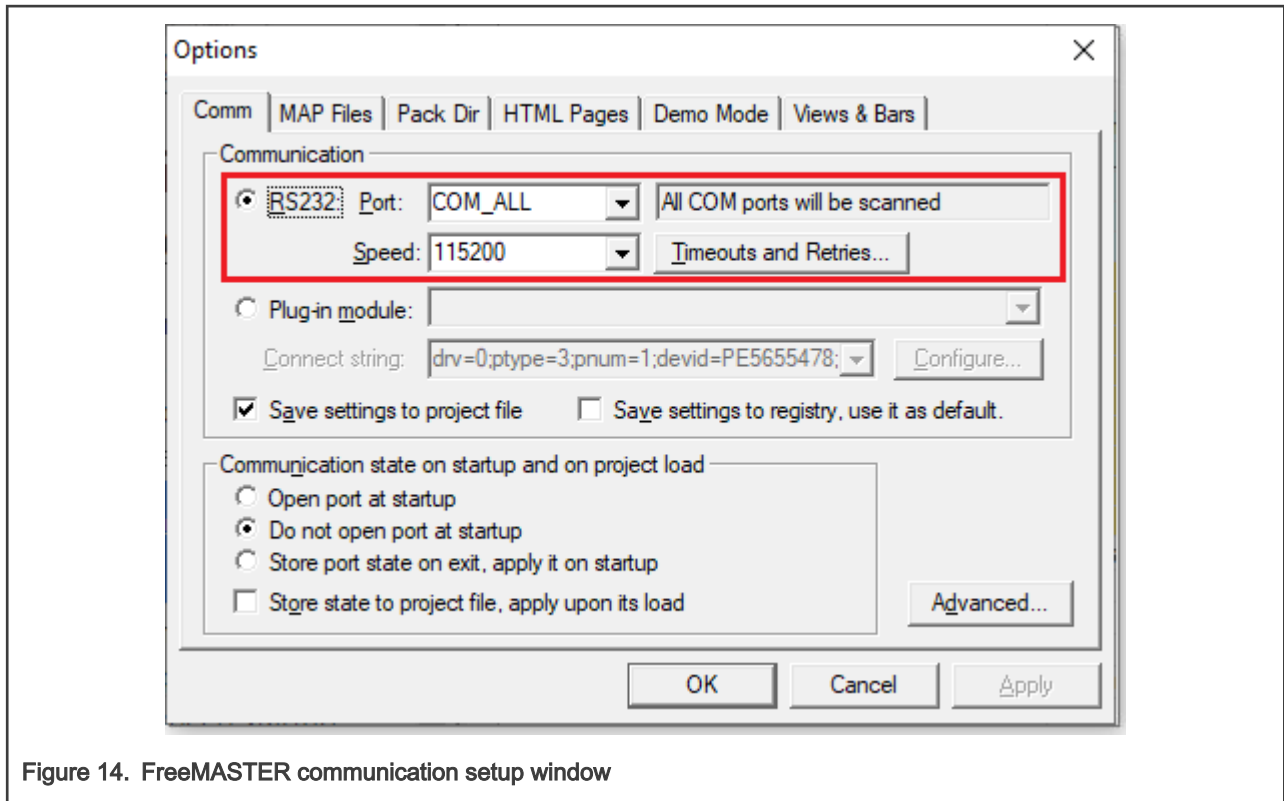


Figure 14. FreeMASTER communication setup window

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 motor control example uses 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's 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 are no issues with correct path to ELF/Dwarf file. The variables described by TSA tables may be read-only, so even if FreeMASTER attempts to write the variable, the value is actively denied by the target MCU side. 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 don't want to use the TSA feature, you need to modify the example code and Freemaster project. Follow these steps:

- Open motor control project and rewrite macro FMSTR_USE_TSA from 1 to 0 in freemaster_cfg.h file.
- Build, download and run motor control project
- Open FreeMASTER project and click to Project → Options (or use shortcut Ctrl+T)
- Click to MAP Files tab and find Default symbol file (ELF/Dwarf executable file) located in IDE Output folder

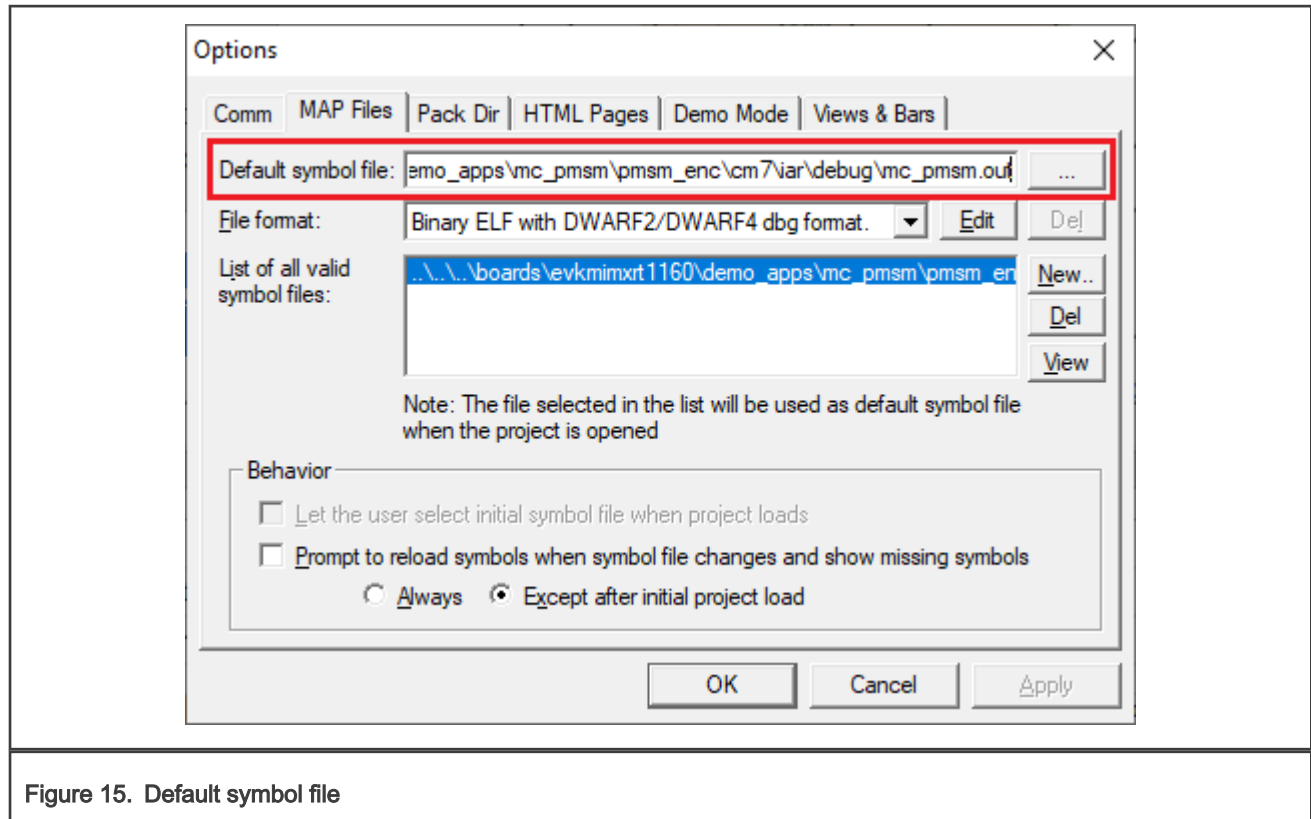


Figure 15. Default symbol file

- Click to OK and restart FreeMASTER communication.

For more information check [FreeMASTER User Guide](#)

8.3 MCAT FreeMASTER interface (Motor Control Application Tuning)

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 shown in [Figure 16](#). Each tab from the tab menu represents one sub-module which enables tuning or control different aspects of the application. Besides the MCAT page for PMSM, several scopes, recorders, and variables in the project tree 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 *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 (e.g. motor parameters) are loaded from *mX_pmsm_appconfig.h* file (JSON formatted comments). Only existing *mX_pmsm_appconfig.h* files can be selected for loading. Actually loaded *mX_pmsm_appconfig.h* file is displayed in grey field (7).
- **Save data** - MCAT input fields (JSON formatted comments) and output macros are saved to *mX_pmsm_appconfig.h* file. Up to 9 files (*m1-9_pmsm_appconfig.h*) can be selected. A pop up window with user motor ID and description appears when a different *mX_pmsm_appconfig.h* file is selected. The motor ID and description is also saved in *mX_pmsm_appconfig.h* in form of JSON comment. At single motor control application the embedded code `#includes m1_pmsm_appconfig.h` only. Therefore, saving to higher indexed *mX_pmsm_appconfig.h* files has no effect at compilation stage.
- **Update target** - writes the MCAT calculated tuning parameters to FreeMASTER Variables which effectively updates the values on target MCU. These tuning parameters are updated in MCU's RAM memory. To write these tuning parameters to MCU's flash memory, *m1_pmsm_appconfig.h* must be saved, code re-compiled and downloaded to MCU.

NOTE

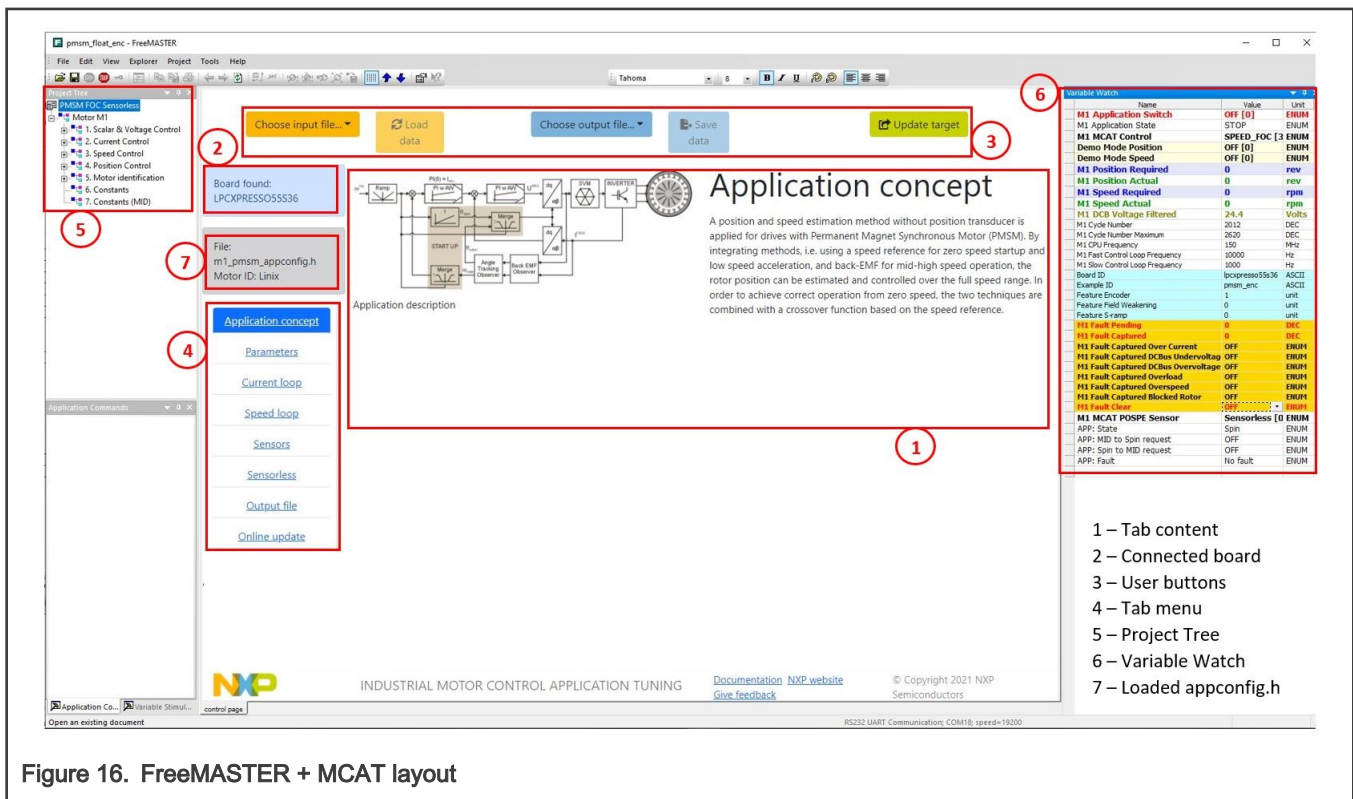
Path to `mX_pmsm_appconfig.h` file composes also from *Board ID* value. Therefore, FreeMASTER must be connected to target and *Board ID* value read prior using Save/Load buttons.

NOTE

Only **Update target** button updates values on target in real-time. Load/Save buttons operate with `mX_pmsm_appconfig.h` file only.

NOTE

MCAT may require internet connection. If no internet connection is available, CSS and icons may not be properly loaded.



- 1 – Tab content
- 2 – Connected board
- 3 – User buttons
- 4 – Tab menu
- 5 – Project Tree
- 6 – Variable Watch
- 7 – Loaded appconfig.h

Figure 16. FreeMASTER + MCAT layout

In the default configuration, the following tabs are available:

- “Application concept”—welcome page with the PMSM sensor/sensorless FOC diagram and a short description of the application.
- “Parameters”—this page enables you to modify the motor parameters, specification of hardware and application scales, 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. The position proportional controller constant is also set here.
- “Sensors”—this page contains the encoder parameters and position observer parameters. Not available for all devices.
- “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 sensor/sensorless FOC application. It is also possible to generate the `m1_pmsm_appconfig.h` file, which is then used to preset all application parameters permanently at the project rebuild.

- "Online update" — this tab shows actual values of variables on target and new calculated values, which can be used for update variables on the target.

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

8.3.1 MCAT tabs' description

This chapter describes MCAT input parameters and equations used to calculate MCAT output (generated) parameters. In the default configuration, the below described tabs are available. Some tabs may be missing if not supported in the embedded code. There are general constants used at MCAT calculations listed in the following table:

Table 7. Constants used in equations

Constant	Value	Unit
UmaxCoeff	1.73205	-
DiscMethodFactor	1	-
k_factor	100	-
pi	3.1416	-

Application concept

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

Parameters

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

Table 8. Parameters tab inputs

MCAT group	MCAT name	Equation name	Description	Unit
Motor parameters	PP	Pp	Motor number of pole-pairs. Obtain from motor manufacturer or use the pole-pair assistant to determine and then fill manually.	-
	Rs	Rs	Stator phase resistance. Obtain from motor manufacturer or use the electrical parameters identification and then fill manually.	[Ω]
	Ld	Ld	Stator direct inductance. Obtain from motor manufacturer or use the electrical parameters identification and then fill manually.	[H]
	Lq	Lq	Stator quadrature inductance. Obtain from motor manufacturer or use the electrical parameters identification and then fill manually.	[H]

Table continues on the next page...

Table 8. Parameters tab inputs (continued)

MCAT group	MCAT name	Equation name	Description	Unit
	Ke	Ke	Motor electrical constant. Obtain from motor manufacturer or use the Ke identification and then fill manually.	[V.sec/rad]
	J	J	Drive inertia (motor + plant). Use the mechanical identification and then fill manually.	[kg.m2]
	Iph nom	IphNom	Nominal motor current. Obtain from motor manufacturer.	[A]
	Uph nom	UphNom	Nominal motor voltage. Obtain from motor manufacturer.	[V]
	N nom	Nnom	Nominal motor speed. Obtain from motor manufacturer.	[rpm]
Hardware scales	I max	Imax	Current sensing HW scale. Keep as-is in case of standard NXP HW or recalculate according to own schematic.	[A]
	U DCB max	UdcbMax	DCBus voltage sensing HW scale. Keep as-is in case of standard NXP HW or recalculate according to own schematic.	[V]
Fault limits	U DCB trip	UdcbTrip	DCBus braking resistor threshold. Braking resistor's transistor is turned on when DCbus voltage exceeds this threshold.	[V]
	U DCB under	UdcbUnder	DCBus under voltage fault threshold	[V]
	U DCB over	UdcbOver	DCBus over voltage fault threshold	[V]
	N over	Nover	Over speed fault threshold	[rpm]
	N min	Nmin	Minimal closed loop speed. When the required speed ramps down under this threshold the motor control state machine goes to freewheel state where top and bottom transistors are turned off and motor speeds down freely. Applies only for sensorless operation.	[rpm]

Table continues on the next page...

Table 8. Parameters tab inputs (continued)

MCAT group	MCAT name	Equation name	Description	Unit
	E block	Eblock	Blocked rotor detection. When Bemf voltage drops under <i>E block</i> threshold for more than <i>E block per</i> (fast loop ticks), the blocked rotor fault is detected.	[V]
	E block per	EblockPer		-
Application scales	N max	Nmax	Application speed scale. Keep about 10% margin above <i>N over</i> .	[rpm]
	U DCB IIR F0	UdcbIIRf0	Cut-off frequency of DCBus IIR filter	[Hz]
	Calibration duration	CalibDuration	ADC (phase current offset) calibration duration. Done every time transitioning from STOP to RUN.	[sec]
	Fault duration	FaultDuration	After fault condition disappears wait defined time to clear pending faults bitfield and transition to STOP state.	[sec]
	Freewheel duration	FreewheelDuration	Free-wheel state duration. Freewheel state is entered when ramped speed drops under <i>N min</i> .	[sec]
	Scalar Uq min	ScalarUqMin	Scalar control voltage minimal value.	[V]
Alignment	Align voltage	AlignVoltage	Motor alignment voltage.	[V]
	Align duration	AlignDuration	Motor alignment duration.	[sec]

Output equations (applies for saving to mX_pmsm_appconfig.h and also for updating a corresponding FreeMASTER variables):

M1_U_MAX = UdcbMax / UmaxCoeff;

M1_FREQ_MAX = Nmax / 60 * Pp;

M1_ALIGN_DURATION = AlignDuration / speedLoopSampleTime;

M1_CALIB_DURATION = CalibDuration / speedLoopSampleTime;

M1_FAULT_DURATION = FaultDuration / speedLoopSampleTime;

M1_FREEWHEEL_DURATION = FreewheelDuration / speedLoopSampleTime;

M1_E_BLOCK_PER = EblockPer;

M1_SPEED_ANGULAR_SCALE = 60 / (Pp * 2 * pi);

M1_N_MIN = Nmin / 60 * (Pp * 2 * pi);

M1_N_MAX = Nmax / 60 * (Pp * 2 * pi);

M1_N_ANGULAR_MAX = (60 / (Pp * 2 * pi));

M1_N_NOM = Nnom / 60 * (Pp * 2 * pi);

M1_N_OVERSPEED = Nover / 60 * (Pp * 2 * pi);

```

M1_UDCB_IIR_B0 = (2 * pi * UdcblIRf0 * currentLoopSampleTime) / (2 + (2 * pi * UdcblIRf0 * currentLoopSampleTime));
M1_UDCB_IIR_B1 = (2 * pi * UdcblIRf0 * currentLoopSampleTime) / (2 + (2 * pi * UdcblIRf0 * currentLoopSampleTime));
M1_UDCB_IIR_A1 = -(2 * pi * UdcblIRf0 * currentLoopSampleTime - 2) / (2 + (2 * pi * UdcblIRf0 * currentLoopSampleTime));
M1_SCALAR_VHZ_FACTOR_GAIN = UphNom*k_factor/100/(Nnom*Pp/60);
M1_SCALAR_INTEG_GAIN = 2*pi*Pp*Nmax/60*currentLoopSampleTime/pi;
M1_SCALAR_RAMP_UP = speedLoopIncUp*currentLoopSampleTime/60*Pp;
M1_SCALAR_RAMP_DOWN = speedLoopIncDown*currentLoopSampleTime/60*Pp;

```

Current loop

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

Table 9. 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 target via FreeMASTER because application timing is set in embedded code by peripherals setting. This value is accesible only if target is not connected and value cannot be obtained from target.	[sec]
	F0	currentLoopF0	Current controller's bandwidth	[Hz]
	ξ	currentLoopKsi	Current controller's attenuation	-
Current PI controller limits	Output limit	currentLoopOutputLimit	Current controllers' output voltage limit = Duty cycle limit. Be careful setting this limit above 95% because it affects current sensing (Some minimal bottom transistors on time is required).	[%]

Output equations (applies for saving to mX_pmsm_appconfig.h and also for updating a corresponding FreeMASTER variables):

```

M1_CLOOP_LIMIT = currentLoopOutputLimit / UmaxCoeff / 100;
M1_D_KP_GAIN = (2 * currentLoopKsi * 2 * pi * currentLoopF0 * Ld) - Rs;
M1_D_KI_GAIN = (2 * pi * currentLoopF0)^2 * Ld * currentLoopSampleTime / DiscMethodFactor;
M1_Q_KP_GAIN = (2 * currentLoopKsi * 2 * pi * currentLoopF0 * Lq) - Rs;
M1_Q_KI_GAIN = (2 * pi * currentLoopF0)^2 * Lq * currentLoopSampleTime / DiscMethodFactor;

```

Speed loop

This tab enables speed loop PI controller gains and output limits tuning, required speed ramp parameters, feedback speed filter tuning, and position P controller gain tuning (available at sensed/encoder applications only). *MCAT group* and *MCAT name* helps to locate the parameter in MCAT layout. *Equation name* represents the input parameter in equations below.

Table 10. Speed loop tab input

MCAT group	MCAT name	Equation name	Description	Unit
Loop parameters	Sample time	speedLoopSampleTime	Slow control loop period. This disabled value is read from target via FreeMASTER because application timing is set in embedded code by peripherals setting. This value is accesible only if target is not connected and value cannot be obtained from target.	[sec]
	F0	speedLoopF0	Speed controller's bandwidth	[Hz]
	ξ	speedLoopKsi	Speed controller's attenuation	-
Speed ramp	Inc up	speedLoopIncUp	Required speed maximal acceleration	[rpm/sec]
	Inc down	speedLoopIncDown	Required speed maximal deceleration	[rpm/sec]
Actual speed filter	Cut-off freq	speedLoopCutOffFreq	Speed feedback (before entering PI subtraction) filter bandwidth.	[Hz]
Speed PI controller limits	Upper limit	speedLoopUpperLimit	Maximal required Q-axis current (Speed controller's output). Q-axis current limitation equals to motor torque limitation.	[A]
	Lower limit	speedLoopLowerLimit	Minimal required Q-axis current (Speed controller's output). Q-axis current limitation equals to motor torque limitation.	[A]
Position P controller constans	PL_Kp	speedLoopPLKp	Position controller proportional constant in time domain.	

Output equations (applies for saving to mX_pmsm_appconfig.h and also for updating a corresponding FreeMASTER variables):

$\text{varKt} = 3 * \text{Ke} / (\text{sqrt}(3));$

$\text{M1_SPEED_PI_PROP_GAIN} = (2 * \text{pi} / 60 * (4 * \text{speedLoopKsi} * \text{pi} * \text{speedLoopF0}) * \text{J} / \text{varKt});$

$\text{M1_SPEED_PI_INTEG_GAIN} = (2 * \text{pi} / 60 * ((2 * \text{pi} * \text{speedLoopF0}) * (2 * \text{pi} * \text{speedLoopF0}) * \text{J}) / (\text{varKt} * 10) * \text{speedLoopSampleTime});$

$\text{M1_SPEED_RAMP_UP} = (\text{speedLoopIncUp} * \text{speedLoopSampleTime} / (60 / (\text{Pp} * 2 * \text{pi})));$

$\text{M1_SPEED_RAMP_DOWN} = (\text{speedLoopIncDown} * \text{speedLoopSampleTime} / (60 / (\text{Pp} * 2 * \text{pi})));$

$\text{M1_SPEED_IIR_B0} = (2 * \text{pi} * \text{speedLoopCutOffFreq} * \text{currentLoopSampleTime}) / (2 + (2 * \text{pi} * \text{speedLoopCutOffFreq} * \text{currentLoopSampleTime}));$

$\text{M1_SPEED_IIR_B1} = (2 * \text{pi} * \text{speedLoopCutOffFreq} * \text{currentLoopSampleTime}) / (2 + (2 * \text{pi} * \text{speedLoopCutOffFreq} * \text{currentLoopSampleTime}));$

$\text{M1_SPEED_IIR_A1} = -(2 * \text{pi} * \text{speedLoopCutOffFreq} * \text{currentLoopSampleTime} - 2) / (2 + (2 * \text{pi} * \text{speedLoopCutOffFreq} * \text{currentLoopSampleTime}));$

Sensors

Available at sensed (encoder) applications only. This tab enables setting the encoder properties and tuning encoder's tracking observer. *MCAT group* and *MCAT name* helps to locate the parameter in MCAT layout. *Equation name* represents the input parameter in equations below.

Table 11. Sensors tab input

MCAT group	MCAT name	Equation name	Description	Unit
Quadrature encoder	Pulse number	sensorEncPulseNumber	Number of quadrature encoder pulses. Obtain this value from encoder manufacturer OR estimate based on speed/ position comparison of Scalar controlled application with encoder processing running on background.	[pulses]
	Direction	sensorEncDir	Encoder direction / Phase A&B order.	-
	Minimal speed	sensorEncNmin	Encoder minimal speed.	[rpm]
Position observer parameters	Sample time	sensorObsrvParSampleTime	Current control loop sampling period. This disabled value is read from target via FreeMASTER because application timing is set in embedded code by peripherals setting. This value is accesible only if target is not connected and value cannot be obtained from target.	[sec]
	F0	sensorObsrvParF0	Position observer bandwidth	[Hz]
	ξ	sensorObsrvParKsi	Position observer attenuation	-

Output equations (applies for saving to mX_pmsm_appconfig.h and also for updating a corresponding FreeMASTER variables):

```

M1_POSPE_KP_GAIN = (4.0 * pi * sensorObsrvParKsi * sensorObsrvParF0);
M1_POSPE_KI_GAIN = ((2*pi*sensorObsrvParF0)^2 * sensorObsrvParSampleTime);
M1_POSPE_INTEG_GAIN = (sensorObsrvParSampleTime / pi / DiscMethodFactor);
M1_POSPE_ENC_N_MIN = sensorEncNmin;
M1_POSPE_MECH_POS_GAIN = (32768/((sensorEncPulseNumber*4)/2));

```

Sensorless

This tab enables Bemf observer and Tracking observer parameters tuning and open-loop startup tuning. *MCAT group* and *MCAT name* helps to locate the parameter in MCAT layout. *Equation name* represents the input parameter in equations below.

Table 12. 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 start-up parameters	Start-up ramp	sensorlessStartupRamp	Open loop startup ramp	[rpm/sec]
	Start-up 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)	[%]

Output equations (applies for saving to mX_pmsm_appconfig.h and also for updating a corresponding FreeMASTER variables):

M1_I_SCALE = (Ld / (Ld + currentLoopSampleTime * Rs));

M1_U_SCALE = (currentLoopSampleTime / (Ld + currentLoopSampleTime * Rs));

M1_E_SCALE = (currentLoopSampleTime / (Ld + currentLoopSampleTime * Rs)) ;

M1_WI_SCALE = (Lq * currentLoopSampleTime / (Ld + currentLoopSampleTime * Rs));

M1_BEMF_DQ_KP_GAIN = ((2 * sensorlessBemfObsrvKsi * 2 * pi * sensorlessBemfObsrvF0 * Ld - Rs));

M1_BEMF_DQ_KI_GAIN = (Ld * (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 / (Pp * 2 * pi)));

M1_MERG_SPEED_TRH = (sensorlessMergingSpeed / (60 / (Pp * 2 * pi)));

M1_MERG_COEFF = ((sensorlessMergingCoeff / 100) * (60 / (Pp * sensorlessMergingSpeed))) / currentLoopSampleTime / 2 / 32768);

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));

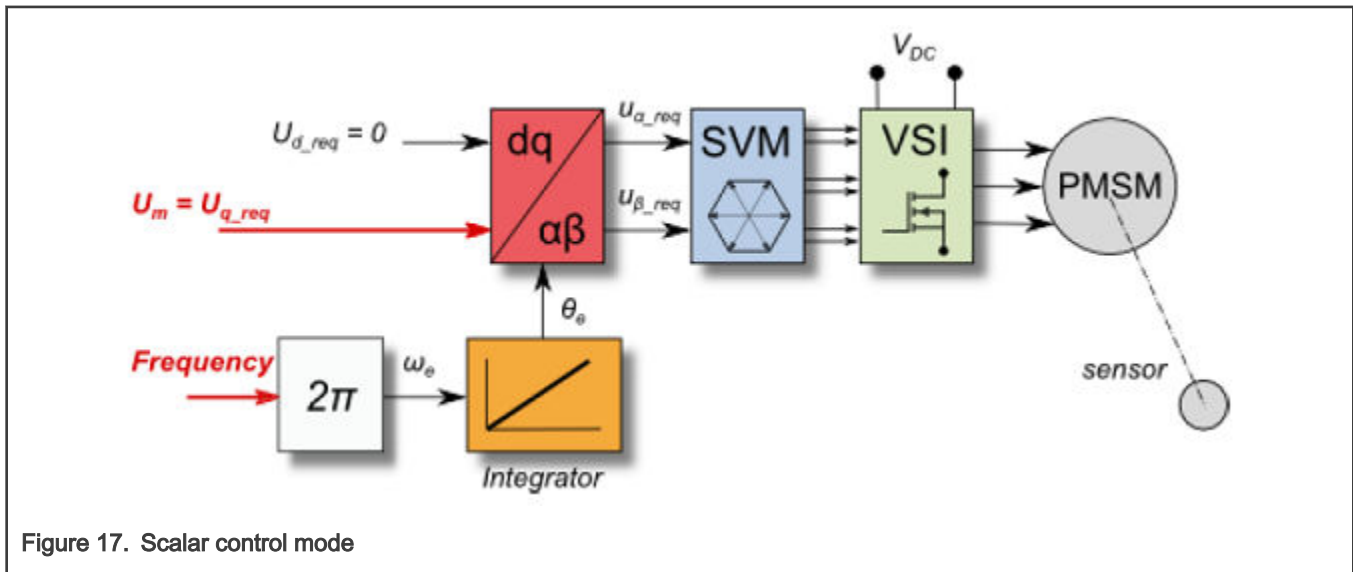
8.4 Motor Control Modes

In the "Project Tree" you can choose between the scalar control and the FOC control using the appropriate FreeMASTER tabs. The application can be controlled through the FreeMASTER variables watch which correspond to the control structure selected in FreeMASTER project tree. This is useful for application tuning and debugging. Required control structure must be selected in the

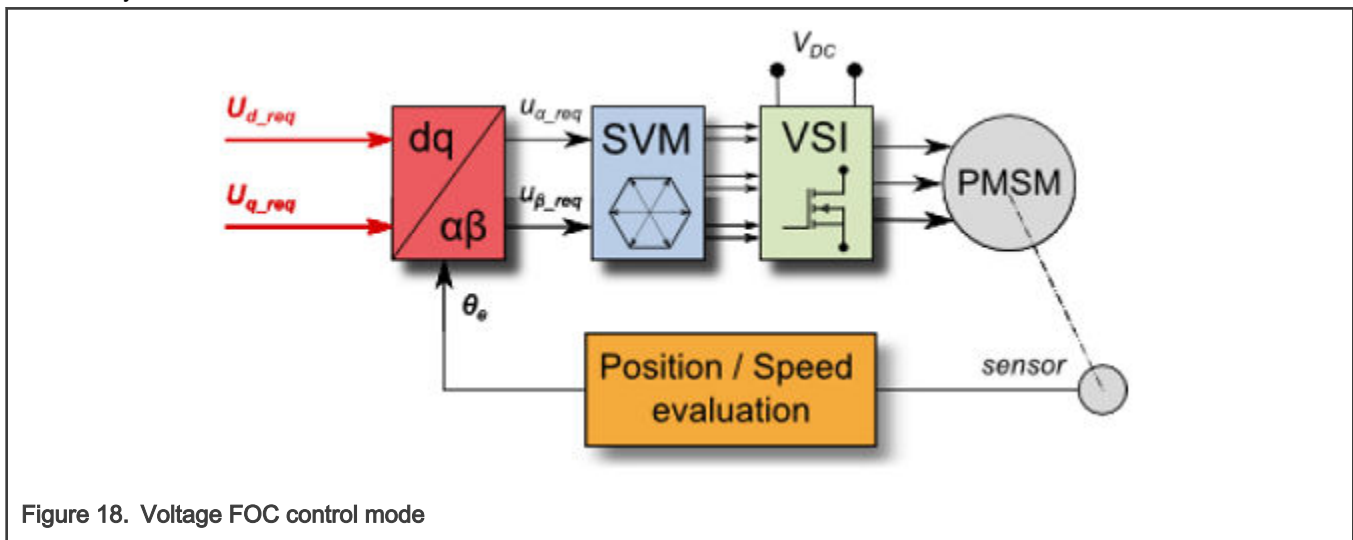
"M1 MCAT Control" variable. Then use "M1 Application Switch" variable to turn on or off the application. Set/clear "M1 Application Switch" variable also enables/disables all PWM channels.

Control structure

The scalar control diagram is shown in figure below. 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. Hence, the control method is sometimes called Volt per Hertz (or V/Hz). The position estimation BEMF observer and tracking observer algorithms (see Sensorless PMSM Field-Oriented Control ([document DRM148](#)) for more information) run in the background, even if the estimated position information is not directly used. This is useful for the BEMF observer tuning.



The block diagram of the voltage FOC is in figure below. 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. Both the d-axis and q-axis stator voltages can be specified in the "M1 MCAT Ud Required" and "M1 MCAT Uq Required" fields. This control method is useful for the BEMF observer functionality check.



The current FOC (or torque) 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 the block diagram in figure below. The d-axis current component "M1 MCAT Id Required" is responsible for the rotor flux control. The q-axis current component of the current "M1 MCAT Iq Required" generates torque and, by its application, the motor starts

running. By changing the polarity of the current "M1 MCAT Iq Required", the motor changes the direction of rotation. Supposing that the BEMF observer is tuned correctly, the current PI controllers can be tuned using the current FOC control structure.

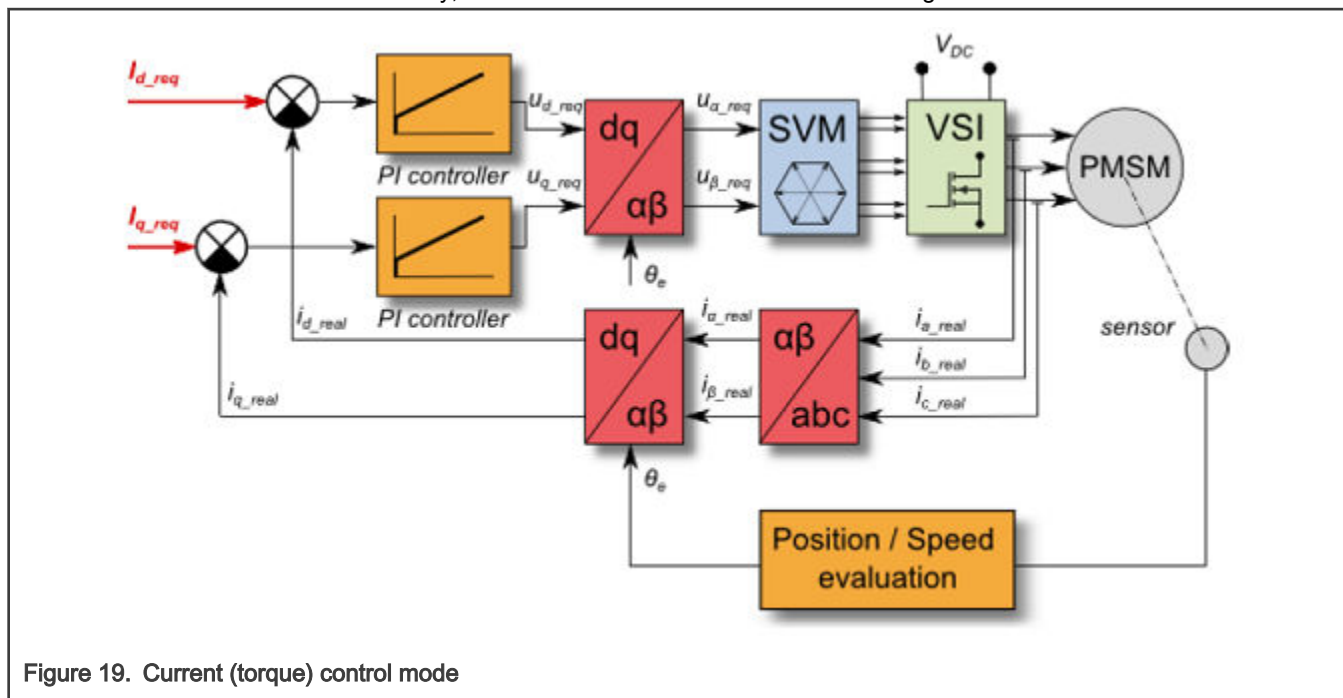


Figure 19. Current (torque) control mode

The speed PMSM sensor/sensorless FOC (its diagram is shown in figure below) 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.

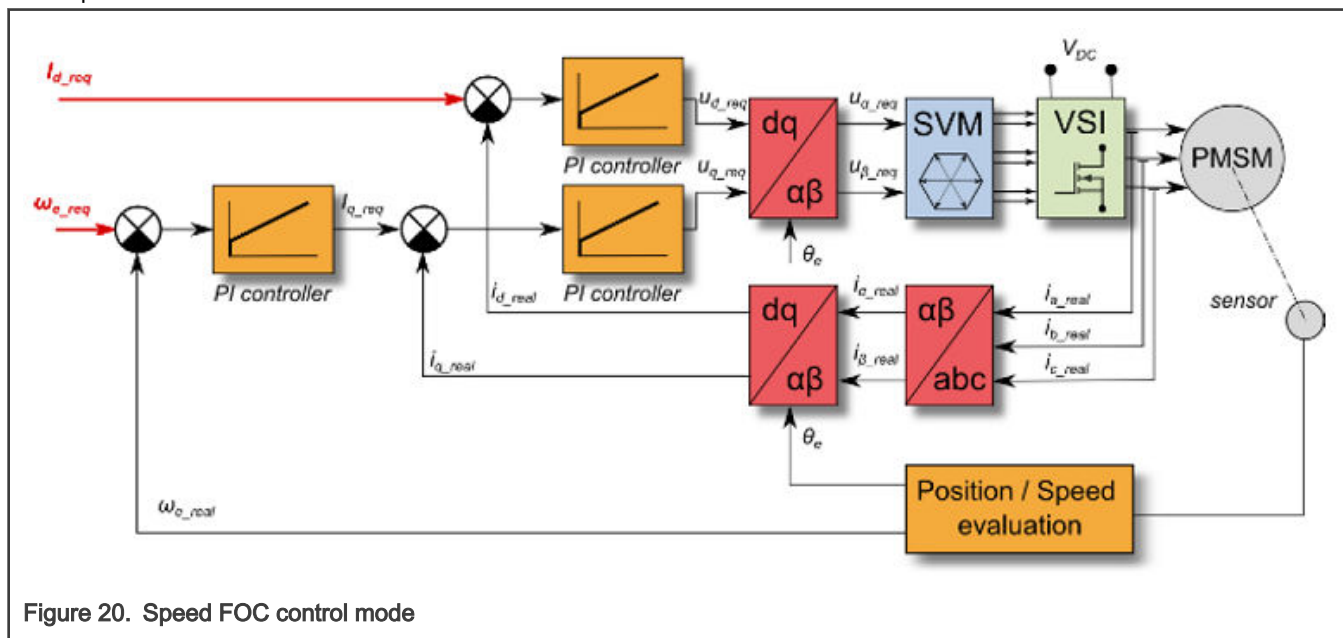


Figure 20. Speed FOC control mode

The position PMSM sensor FOC is shown in figure below (available for sensed/encoder based applications only). The position control using the P controller can be tuned in the "Speed loop" 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 needed to set a soft speed ramp (in the "Speed loop" menu tab) because the voltage on the DC-bus can rise when braking the quickly spinning shaft. It may cause the overvoltage fault.

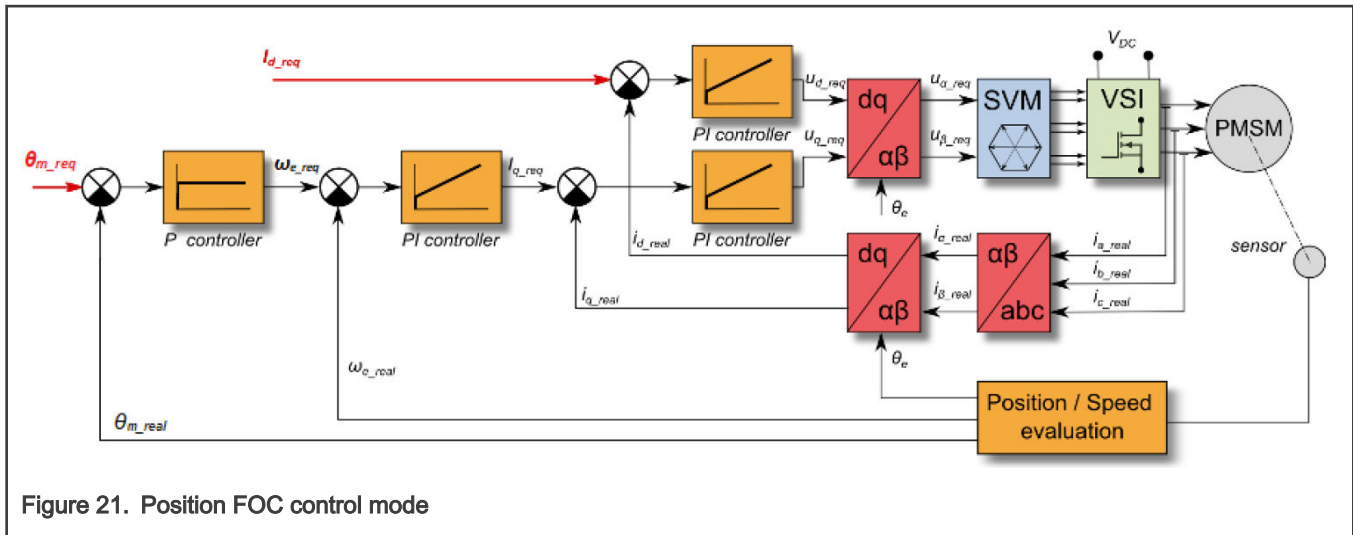


Figure 21. Position FOC control mode

8.5 Switch between Spin and MID

User can switch between two main modes of application: *Spin* and *MID* (Motor identification). *Spin* is for control PMSM (see [MCAT FreeMASTER interface \(Motor Control Application Tuning\)](#)). *MID* is for motor parameters identification (see [Motor parameter identification using MID](#)). Actual mode of application is shown in *APP: State* variable watch. The mode change can be made by *APP: MID to Spin request* or *APP: Spin to MID request* variables watch. The result of the change mode request shows *APP: Fault* variable watch. *MID fault* occurs when parameters identification still runs or MID state machine is in the fault state. *Spin fault* occurs when *M1 Application switch* variable watch is ON or *M1 Application state* variable watch is not STOP.

8.6 Identifying parameters of user motor

Because the model-based control methods of the PMSM drives provide high performance (e.g. 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 described in detail in [MID algorithms](#). MID is controlled via the FreeMASTER "Motor Identification" page shown in [Figure 22](#).

Name	Value	Unit	Period [ms]	
MID: Command	STOP	ENUM	1000	MID Control (set Run to trigger MID)
MID: State	STOP	ENUM	1000	Actual MID SM state
APP: State	MID	ENUM	1000	Actual application state (Spin or MID)
APP: MID to Spin request	OFF	ENUM	1000	Request to change application state
APP: Spin to MID request	OFF	ENUM	1000	
APP: Fault	No fault	ENUM	1000	Application fault
DIAG: Fault Pending	0	DEC	1000	General MC fault status
DIAG: Fault Captured	0	DEC	1000	
DIAG: Fault DCBus Overvoltage	OFF	ENUM	1000	General MC faults
DIAG: Fault DCBus Undervoltage	OFF	ENUM	1000	
DIAG: Fault Over Current	OFF	ENUM	1000	
DIAG: Fault clear	OFF	ENUM	1000	
MID: Faults	b# 0000	BIN	1000	MID faults and warnings
MID: Warnings	b# 0000	BIN	1000	
MID: Measurement Type	PP_ASSIST	ENUM	1000	Measurement type (pp assist, electrical or mechanical parameters, Ke constant)
MID: Known Param Pp	1	pairs	1000	Known motor parameters
MID: Known Param Rs	0	Ohm	1000	
MID: Known Param Ld	0	H	1000	
MID: Known Param Lq	0	H	1000	
MID: Known Param Ke	0	Vs/rad	1000	
MID: Known Param J	0	kgm ²	1000	
MID: Known Param B	0	Nms	1000	
MID: Start Result	b# 00 0000	BIN	1000	MID start result (equals to zero when all parameters are OK)
MID: Measured Pp	1	pairs	1000	Measured motor parameters
MID: Measured Rs	0	Ohm	1000	
MID: Measured Ld	0	H	1000	
MID: Measured Lq	0	H	1000	
MID: Measured Ke	0	Vs/rad	1000	
MID: Measured J	0	kgm ²	1000	
MID: Measured B	0	Nms	1000	
MID: Config Pp Id Meas	0.5	A	1000	Measurement configuration
MID: Config Pp Freq El. Required	10	Hz	1000	
MID: Config El Mode Estim RL	Mode 0	ENUM	1000	
MID: Config El I DC nominal	5	A	1000	
MID: Config El I DC positive max	6	A	1000	
MID: Config El I DC negative max	-6	A	1000	
MID: Config El I DC (estim Ld)	0	A	1000	
MID: Config El I DC (estim Lq)	5	A	1000	
MID: Config El DQ-switch	Ld meas	ENUM	1000	
MID: Config El I DC req (d-axis)	0	A	1000	
MID: Config El I DC req (q-axis)	0	A	1000	
MID: Config El I AC req	0	A	1000	
MID: Config El I AC frequency	0	Hz	1000	
MID: Config Ke Id Required	0.8	A	1000	
MID: Config Ke Freq El. Required	20	Hz	1000	
MID: Config Mech Kt	0.5	Nm/A	1000	
MID: Config Mech Iq Startup	0.3	A	1000	
MID: Config Mech Merging Coeff.	100	%	1000	
MID: Config Mech Iq Accelerate	0.3	A	1000	
MID: Config Mech Iq Decelerate	0.05	A	1000	
MID: Config Mech Speed Accel. start	251.327	rad/s	1000	
MID: Config Mech Speed Integ. start	251.327	rad/s	1000	
MID: Config Mech Speed Decel. start	345.575	rad/s	1000	

Figure 22. MID FreeMASTER control

Motor parameter identification using MID

The whole MID is controlled via the FreeMASTER "Variable Watch". Motor Identification (MID) sub-block shown in [Figure 22](#). The motor parameter identification workflow is following:

1. Set the *MID: Command* variable to STOP.
2. Select the measurement type you want to perform via the *MID: Measurement Type* variable:
 - PP_ASSIST - Pole-pair identification assistant.
 - EL_PARAMS - Electrical parameters measurement.
 - Ke - BEMF constant 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 measured parameters (if necessary).
4. Set the measurement configuration paramers in the *MID: Config* set of variables.
5. Start the measurement by setting *MID: Command* to RUN.
6. Observe the *MID Start Result* variable for the MID measurement plan validity (see [Table 15](#)) and the actual *MID: State*, *MID: Faults* (see [Table 13](#)), and *MID: Warnings* (see [Table 14](#)) variables.
7. If the measurement finishes successfully, the measured motor parameters are shown in the *MID: Measured* set of variables and *MID: State* goes to STOP.

MID faults and warnings

The MID faults and warnings are saved in the format of masks in the *MID: Faults* and *MID: Warnings* variables. Faults and warnings are cleared automatically when starting a new measurement. If a MID fault appears, the measurement process immediately stops and brings the MID state machine safely to the STOP state. If a MID warning appears, the measurement process continues. Warnings report minor issues during the measurement process. See [Table 13](#) and [Table 14](#) for more details on individual faults and warnings.

Table 13. Measurement faults

Fault mask	Fault description	Fault reason	Troubleshooting
b#0001	Electrical parameters measurement fault.	Some required value cannot be reached or wrong measurement configuration.	Check whether measurement configuration is valid.
b#0010	Mechanical measurement timeout.	Some part of the mechanical measurement (acceleration, deceleration) took too long and exceeded 10 seconds.	Raise the <i>MID: Config Mech Iq Accelerate</i> or lower the <i>MID: Config Mech Iq Decelerate</i> .

Table 14. Measurement warnings

Warning mask	Warning description	Warning reason	Troubleshooting
b#0001	K_e is out of range.	The measured K_e is negative.	Visually check whether the motor was spinning properly during the K_e measurement.

The MID measurement plan is checked after starting the measurement process. If a necessary parameter is not scheduled for the measurement and not set manually, the MID is not started and an error is reported via the *MID: Start Result* variable.

Table 15. MID Start Result variable

MID Start Result mask	Description	Troubleshooting
b#00 0001	Error during initialization electrical parameters measurement.	Check whether inputs to the <i>MCAA_EstimRLInit_FLT</i> are valid.
b#00 0010	The R_s value is missing.	Schedule electrical measurement or enter R_s value manually.
b#00 0100	The L_d value is missing.	Schedule electrical measurement or enter L_d value manually.
b#00 1000	The L_q value is missing.	Schedule electrical measurement or enter L_q value manually.
b#01 0000	The K_e value is missing.	Schedule K_e for measurement or enter its value manually.
b#10 0000	The Pp value is missing.	Enter the Pp value manually.

8.7 MID algorithms

This section describes how each MID algorithm works.

Stator resistance measurement

The stator resistance R_s is averaged from the DC steps, which are generated by the algorithm. The DC step levels are automatically derived from the currents inserted by user. For more details, please, refer to the documentation of *AMCLIB_EstimRL* function from [AMMCLib](#).

Stator inductances measurement

Injection of the AC/DC currents is used for the inductances (L_d , L_q) estimation. Injected AC/DC currents are automatically derived from the currents inserted by user. The default AC current frequency is 500 Hz. For more detail, please, refer to the documentation of *AMCLIB_EstimRL* function from [AMMCLib](#).

BEMF constant measurement

Before the actual BEMF constant K_e measurement, the BEMF and Tracking observers parameters are recalculated from the previously measured or manually set R_s , L_d , and L_q parameters. To measure K_e , the motor must spin. During the measurement, the motor is open-loop driven at the user-defined frequency *MID: Config Ke Freq El. Required* with the user-defined current *MID: Config Ke Id Required* value. When the motor reaches the required speed, the BEMF voltages obtained by the BEMF observer are filtered and K_e is calculated:

$$K_e = \frac{U_{BEMF}}{\omega_{el}} [\Omega]$$

When K_e is being measured, you have to visually check to determine whether the motor is spinning properly. If the motor is not spinning properly, perform these steps:

- Ensure that the number of pp is correct. The required speed for the K_e measurement is also calculated from pp . Therefore, inaccuracy in pp causes inaccuracy in the resulting K_e .
- Increase *MID: Config Ke Id Required* variable to produce higher torque when spinning during the open loop.
- Decrease *MID: Config Ke Freq El. Required* variable to decrease the required speed for the K_e measurement.

Number of pole-pair assistant

The number of pole-pairs cannot be measured without a position sensor. However, there is a simple assistant to determine the number of pole-pairs (PP_ASSIST). The number of the *pp* 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 not to count 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 PP_ASSIST assistant, stop the assistant, increase *MID: Config Pp Id Meas*, and restart the assistant.

Mechanical parameters measurement

The moment of inertia J and the viscous friction B can be identified using a test with the known generated torque T and the loading torque T_{load} .

$$\frac{d\omega_m}{dt} = \frac{1}{J} (T - T_{load} - B\omega_m) \text{ [rad/s}^2\text{]}$$

The ω_m character in the equation is the mechanical speed. The mechanical parameter identification software uses the torque profile. The loading torque is (for simplicity reasons) said to be 0 during the whole measurement. Only the friction and the motor-generated torque are considered. During the first phase of measurement, the constant torque T_{meas} is applied and the motor accelerates to 50 % of its nominal speed in time t_f . These integrals are calculated during the period from t_0 (the speed estimation is accurate enough) to t_f :

$$T_{int} = \int_{t_0}^{t_1} T dt \text{ [Nms]} \quad \omega_{int} = \int_{t_0}^{t_1} \omega_m dt \text{ [rad/s]}$$

During the second phase, the rotor decelerates freely with no generated torque, only by friction. This enables you to simply measure the mechanical time constant $\tau_m = J/B$ as the time in which the rotor decelerates from its original value by 63 %.

The final mechanical parameter estimation can be calculated by integrating:

$$\omega_m(t_1) = \frac{1}{J} T_{int} - \frac{B}{J} \omega_{int} + \omega_m(t_0) \text{ [rad/s]}$$

The moment of inertia is:

$$J = \frac{\tau_m T_{int}}{\tau_m [\omega_m(t_1) - \omega_m(t_0)] + \omega_{int}} \text{ [kgm}^2\text{]}$$

The viscous friction is then derived from the relation between the mechanical time constant and the moment of inertia. To use the mechanical parameters measurement, the current control loop bandwidth $f_{0,Current}$, the speed control loop bandwidth $f_{0,Speed}$, and the mechanical parameters measurement torque Trq_m must be set.

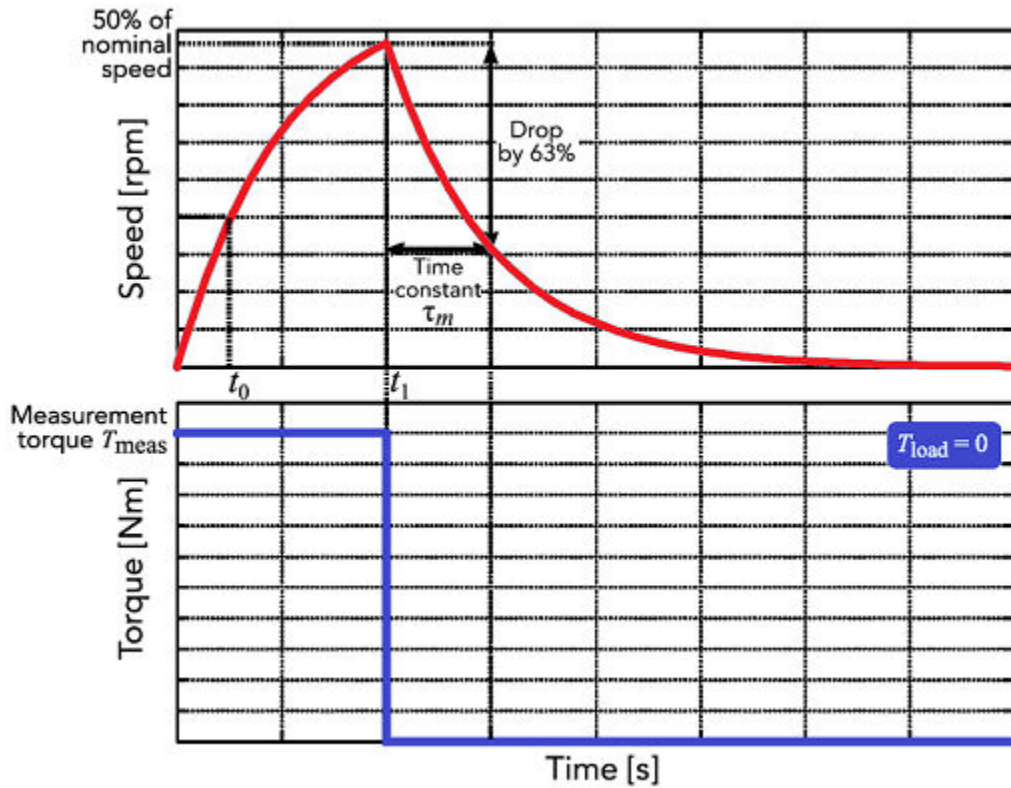


Figure 23. PMSM identification tab

8.8 Electrical parameters measurement control

This section describes how to control electrical parameters measurement, which contains measuring stator resistance R_s , direct inductance L_d and quadrature inductance L_q . There are available 4 modes of measurement which can be selected by *MID: Config EI Mode Estim RL* variable.

Function *MCAA_EstimRLInit_FLT* must be called before the first use of *MCAA_EstimRL_FLT*. Function *MCAA_EstimRL_FLT* must be called periodically with sampling period $F_SAMPLING$, which can be defined by user. Maximum sampling frequency $F_SAMPLING$ is 10 kHz. In the scopes under "Motor identification" FreeMASTER sub-block can be observed measured currents, estimated parameters etc.

Mode 0

This mode is automatic, inductances are measured at a single operating point. Rotor is not fixed. User has to specify nominal current (*MID: Config EI I DC nominal* variable). The AC and DC currents are automatically derived from the nominal current. Frequency of the AC signal set to default 500 Hz.

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

Mode 1

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

- *MID: Config EI I DC (estim Lq)* - Current to determine L_q . In most cases nominal current.
- *MID: Config EI I DC positive max* - Maximum positive DC current for the L_{dq} dependency map measurement.

Injected AC and DC currents are automatically derived from the *MID: Config EI I DC (estim Lq)* and *MID: Config EI I DC positive max* currents. Frequency of the AC signal set to default 500 Hz.

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

Mode 2

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

- *MID: Config EI I DC (estim Ld)* - Current to determine L_d . In most cases 0 A.
- *MID: Config EI I DC (estim Lq)* - Current to determine L_q . In most cases nominal current.
- *MID: Config EI I DC positive max* - Maximum positive DC current for the L_{dq} dependency map measurement. In most cases nominal current.
- *MID: Config EI I DC negative max* - Maximum negative DC current for the L_{dq} dependency map measurement.

Injected AC and DC currents are automatically derived from the *MID: Config EI I DC (estim Ld)*, *MID: Config EI I DC (estim Lq)*, *MID: Config EI I DC positive max* and *MID: Config EI I DC negative max* currents. Frequency of the AC signal set to default 500 Hz.

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

Mode 3

This mode is manual. Rotor must be mechanically fixed after alignment with the first phase. R_s is not calculated at 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 EI I DQ-switch* - Axis switch for AC signal injection (0 for injection AC signal to d-axis, 1 for injection AC signal to q-axis).
- *MID: Config EI I DC req (d-axis)* - Required DC current in d-axis.
- *MID: Config EI I DC req (q-axis)* - Required DC current in q-axis.
- *MID: Config EI I AC req* - Required AC current injected to the d-axis or q-axis.
- *MID: Config EI I AC frequency* - Required frequency of the AC current injected to the d-axis or q-axis.

8.9 Initial configuration setting and update

1. Open the PMSM control application FreeMASTER project containing the dedicated MCAT plug-in module.
2. Select the "Parameters" tab.
3. Leave the measured motor parameters or specify the parameters manually. The motor parameters can be obtained 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 16](#) 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, the manual controller tuning can also be used to calculate this constant.

Table 16. MCAT motor parameters

Parameter	Units	Description	Typical range
pp	[-]	Motor pole pairs	1-10
Rs	[Ω]	1-phase stator resistance	0.3-50
Ld	[H]	1-phase direct inductance	0.00001-0.1

Table continues on the next page...

Table 16. MCAT motor parameters (continued)

Parameter	Units	Description	Typical range
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 current and voltage analog quantities.
- Check the fault limits—these fields are calculated using the motor parameters and hardware scales (see Table 17).

Table 17. 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.

Table 18. Application scales

Parameter	Units	Description	Typical range
N max	[rpm]	Speed scale	>1.1 * N nom
E block	[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.
- Click the “Store data” button to save the modified parameters into the inner file.

8.10 Control structure modes

- Select the scalar control in the “M1 MCAT Control” FreeMASTER variable watch.
- Set the “M1 Application Switch” variable to “ON”. The application state changes to “RUN”.

- 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 running.
- Select the “Phase Currents” recorder from the “Scalar & Voltage Control” FreeMASTER project tree.
- The optimal ratio for the V/Hz profile can be found by changing the V/Hz factor directly using the “M1 V/Hz factor” variable. The shape of the motor currents should be close to a sinusoidal shape (Figure 24). Use the following equation for calculation V/Hz factor:

$$VHz_{factor} = \frac{U_{phnom} \cdot k_{factor}}{\frac{pp \cdot N_{nom}}{60} \cdot 100} [V / Hz]$$

where U_{phnom} is the nominal voltage, k_{factor} is ratio within range 0-100%, pp is the number of pole-pairs and N_{nom} are the nominal revolutions. Changes V/Hz factor won't be propagated to the m1_pmsm_appconfig.h!

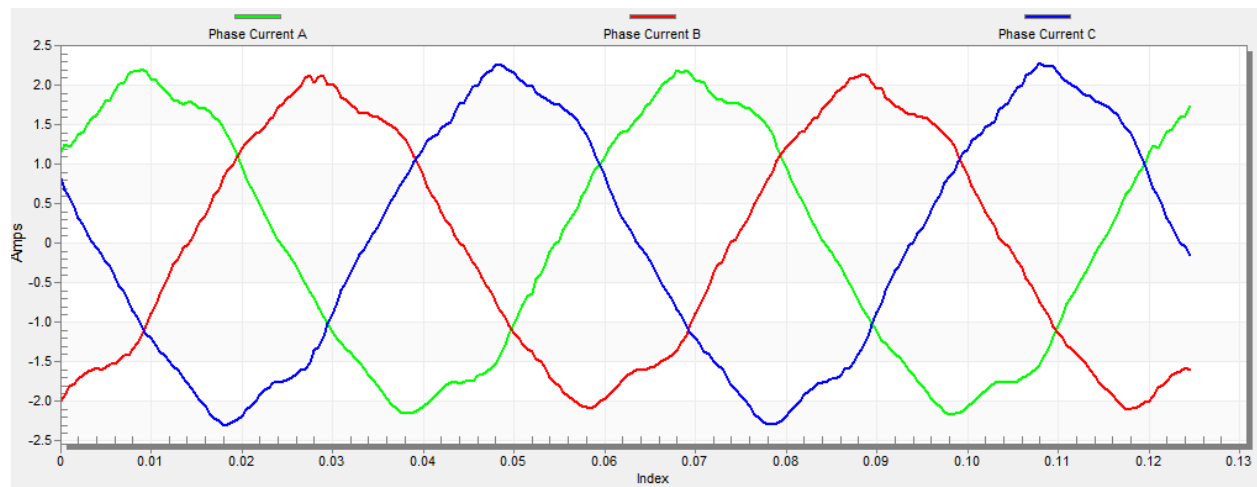


Figure 24. Phase currents

- 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 25) for the Back-EMF 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.

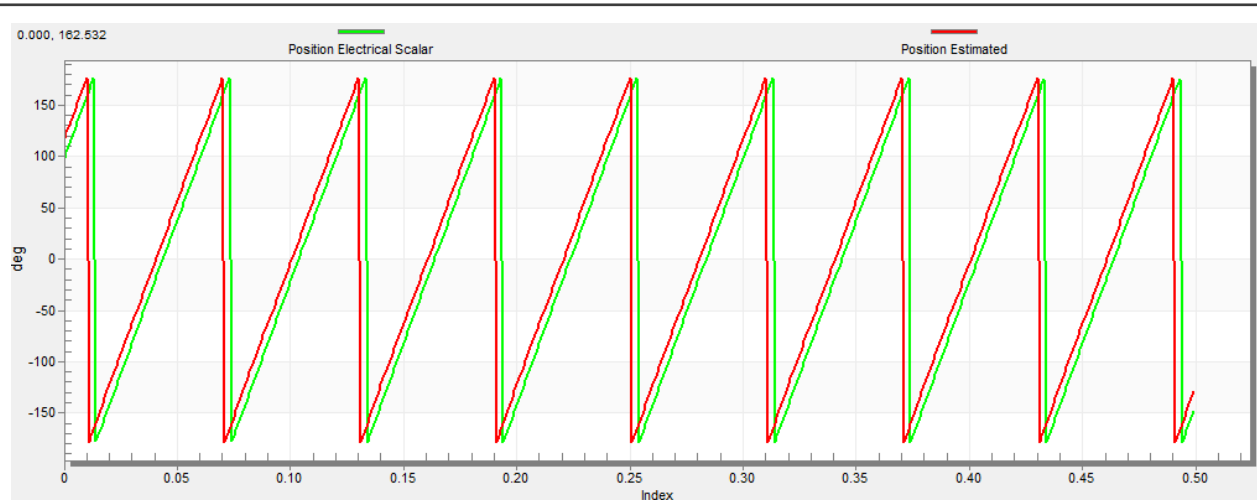


Figure 25. Generated and estimated positions

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 is 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.

8.11 Encoder sensor setting

The encoder sensor settings are in the "Sensors" tab. The encoder sensor enables you to compute speed and position for the sensed 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. The direction of rotation can be determined as follows:

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 running.
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 change "M1 Encoder Direction" variable - only 0 or 1 values is allowed. The second way is invert the encoder wires—phase A and phase B (or the other way round).

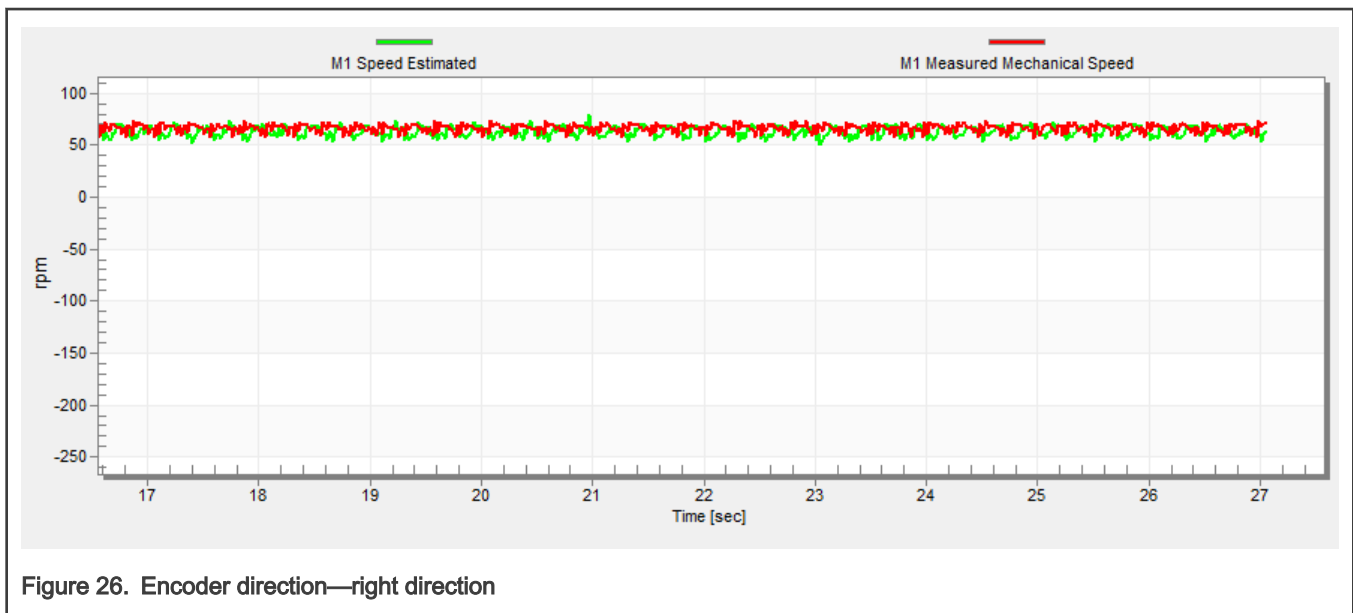
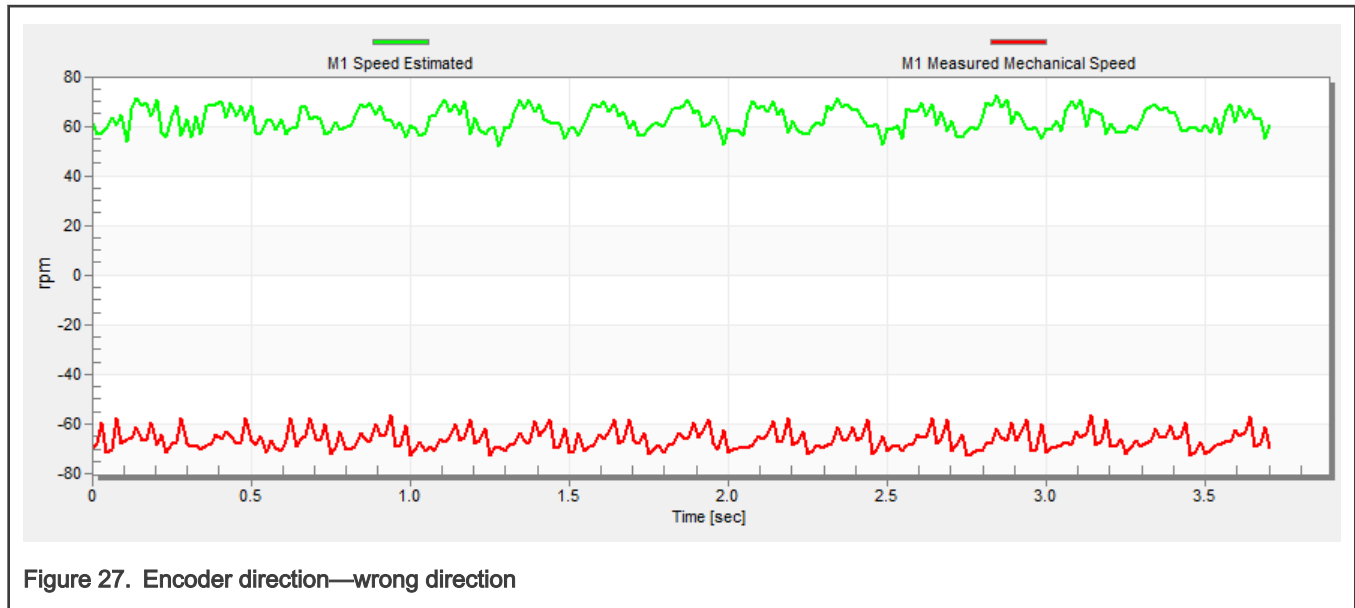


Figure 26. Encoder direction—right direction



8.12 Alignment tuning

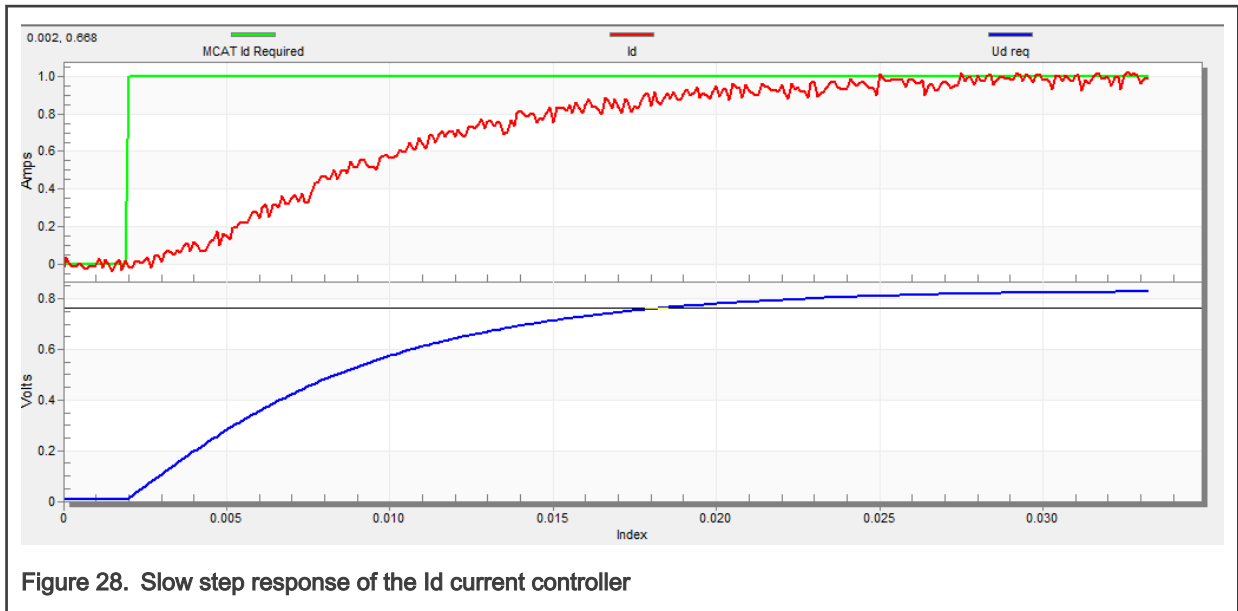
For the alignment parameters, navigate to the “Parameters” MCAT tab. The alignment procedure sets the rotor to an accurate initial position and enables you to apply full start-up torque to the motor. A correct initial position is needed mainly for high start-up loads (compressors, washers, and so on).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

1. The alignment voltage is the value applied to the d-axis during the alignment. Increase this value for a higher shaft load.
2. 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.

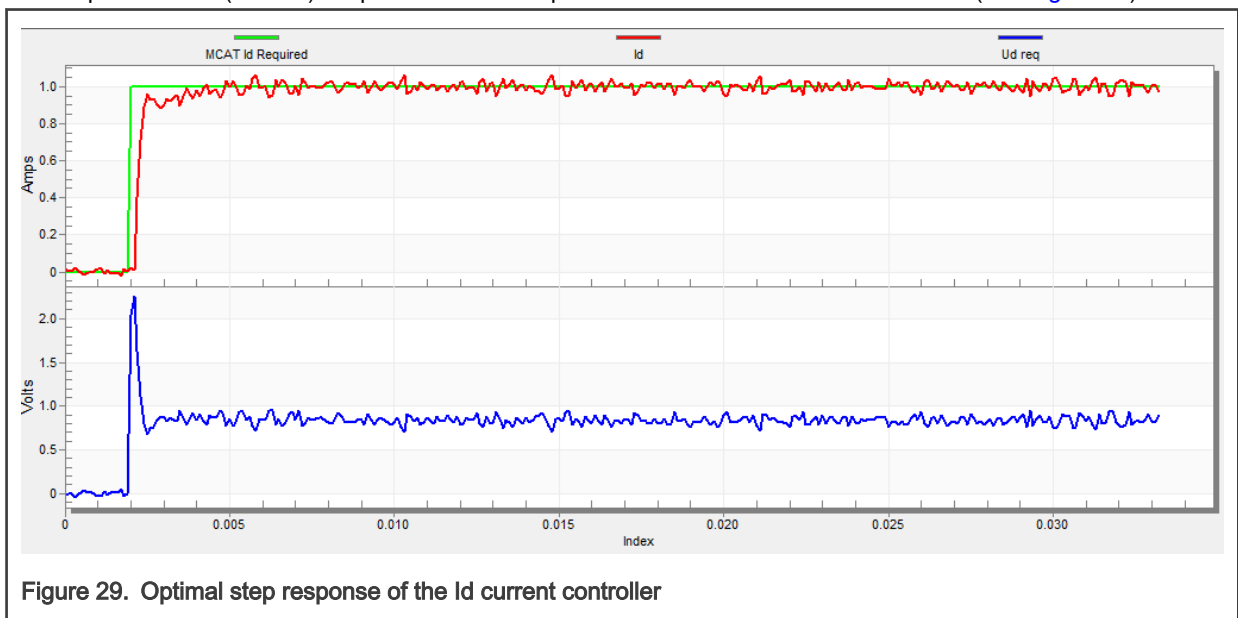
8.13 Current loop tuning

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.

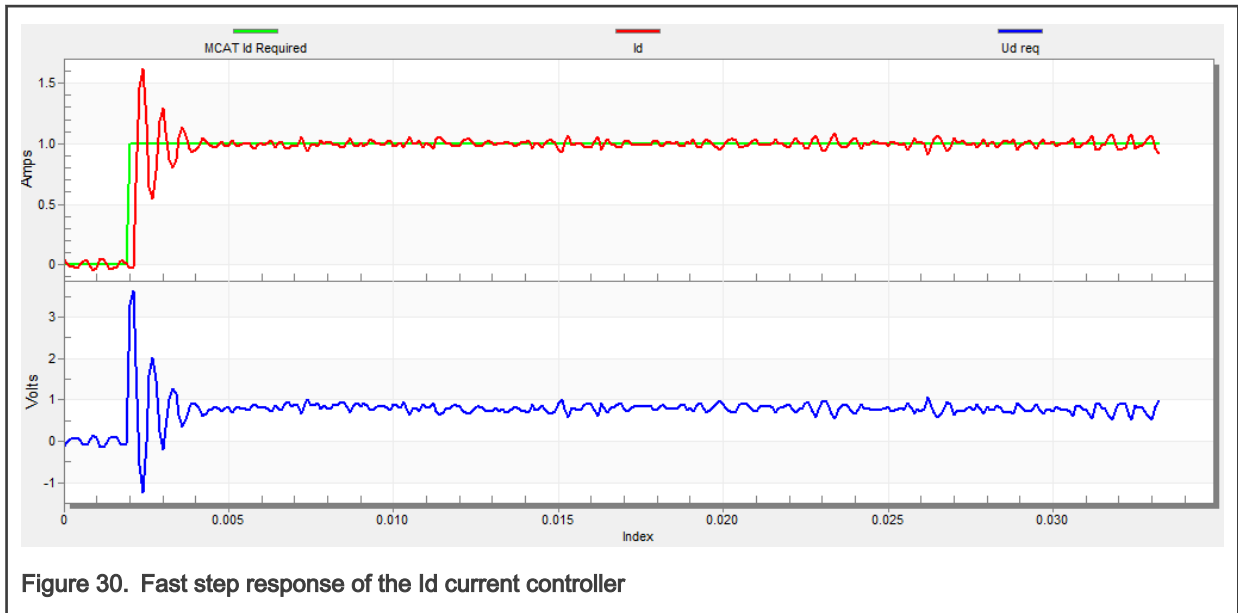
1. Lock the motor shaft.
2. Set the required loop bandwidth and attenuation and click the “Update target” button in the “Current loop” tab. The tuning loop bandwidth parameter defines how fast the loop response is whilst the tuning loop attenuation parameter defines the actual quantity overshoot magnitude.
3. Select the “Current Controller Id” recorder.
4. Select the “Current Control” in the FreeMASTER project tree, select “CURRENT_FOC” in “M1 MCAT Control” variable. Set the “M1 MCAT Iq required” variable to a very low value (for example 0.01), and set the required step in “M1 MCAT Id required”. The control loop response is shown in the recorder.
5. 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 (110 Hz) and the settling time of the Id current is long ([Figure 28](#)).



- The loop bandwidth (400 Hz) is optimal and the response time of the Id current is sufficient (see [Figure 29](#)).

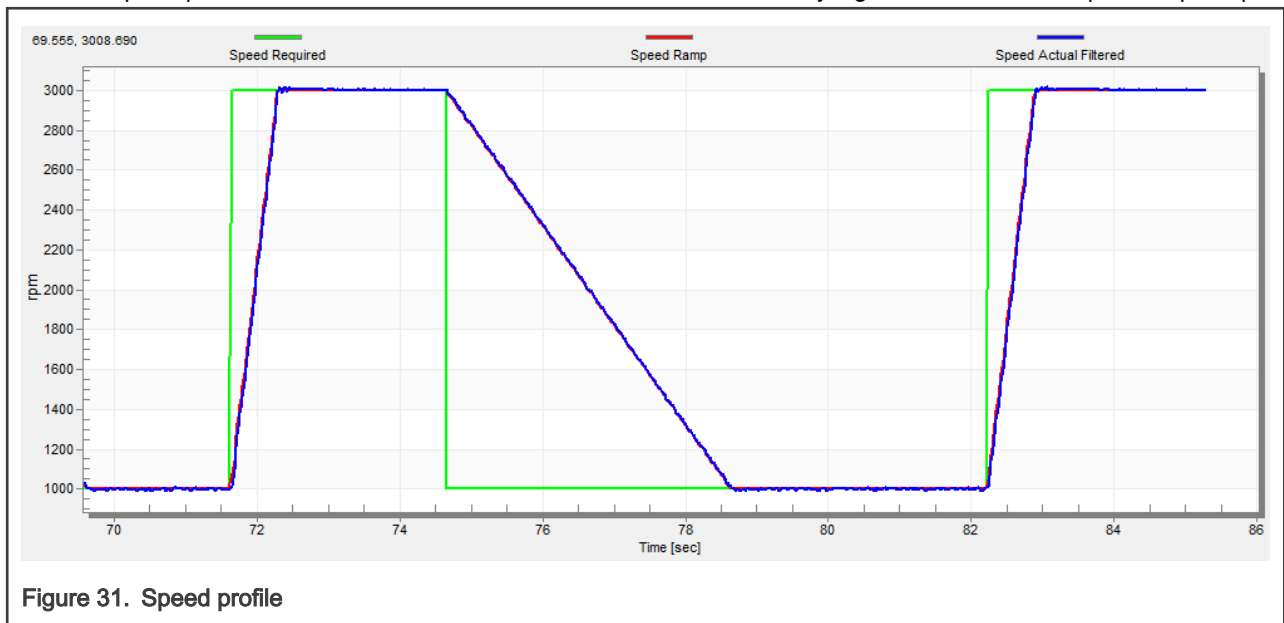


- The loop bandwidth is high (700 Hz) and the response time of the Id current is very fast, but with oscillation and overshoot (see [Figure 30](#)).



8.14 Speed ramp tuning

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 31. The ramp increment down is set to 500 rpm/sec and the increment up is set to 3000 rpm/sec.
3. The start-up ramp increment is in the “Sensorless” tab and its value is usually higher than that of the speed loop ramp.



8.15 Open loop startup

1. The start-up 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 start-up tuning can be processed in all control modes besides the scalar control. Setting the optimal values results in a proper motor startup. An example start-up state of low-dynamic drives (fans, pumps) is shown in [Figure 32](#).
2. Select the "Startup" recorder from the FreeMASTER project tree.
3. Set the start-up ramp increment typically to a higher value than the speed-loop ramp increment.
4. Set the start-up current according to the required start-up torque. For drives such as fans or pumps, the start-up 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 a half of an electrical revolution. The higher the value, the faster the merge. Values close to 1 % are set for the drives where a high start-up torque and smooth transitions between the open loop and the closed loop are required.
7. Click the "Update Target" button to apply the changes to the MCU.
8. Select "SPEED_FOC" in the "M1 MCAT Control" variable.
9. Set the required speed higher than the merging speed.
10. Check the start-up response in the recorder.
11. Tune the start-up parameters until you achieve an optimal response.
12. If the rotor does not start running, increase the start-up current.
13. If the merging process fails (the rotor is stuck or stopped), decrease the start-up ramp increment, increase the merging speed, and set the merging coefficient to 5 %.

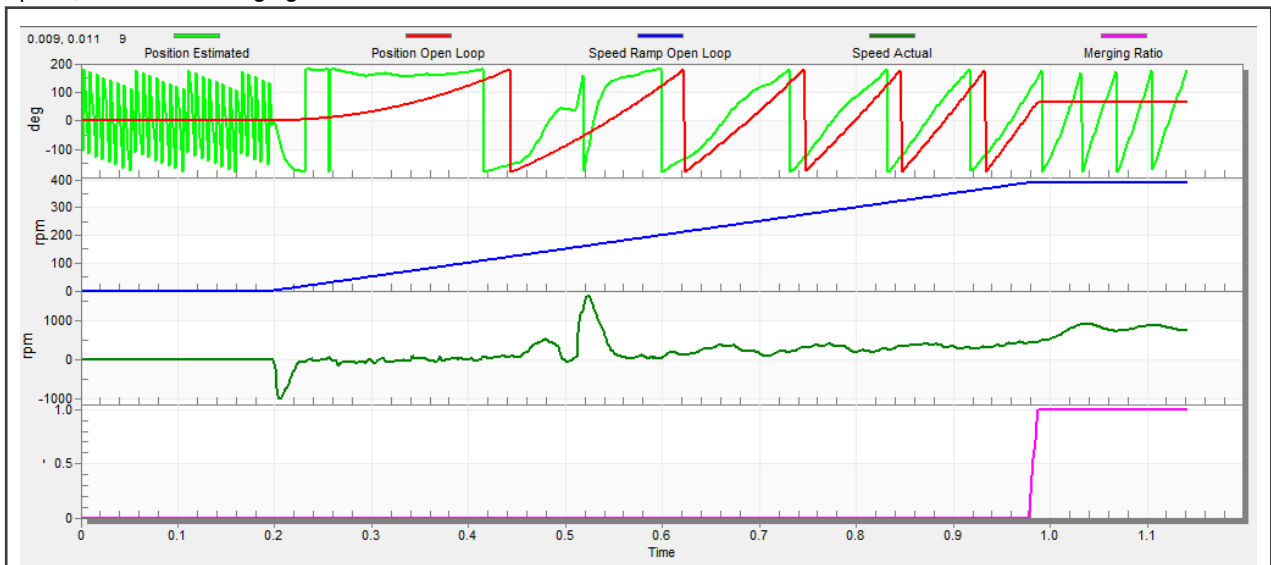


Figure 32. Motor startup

8.16 BEMF observer tuning

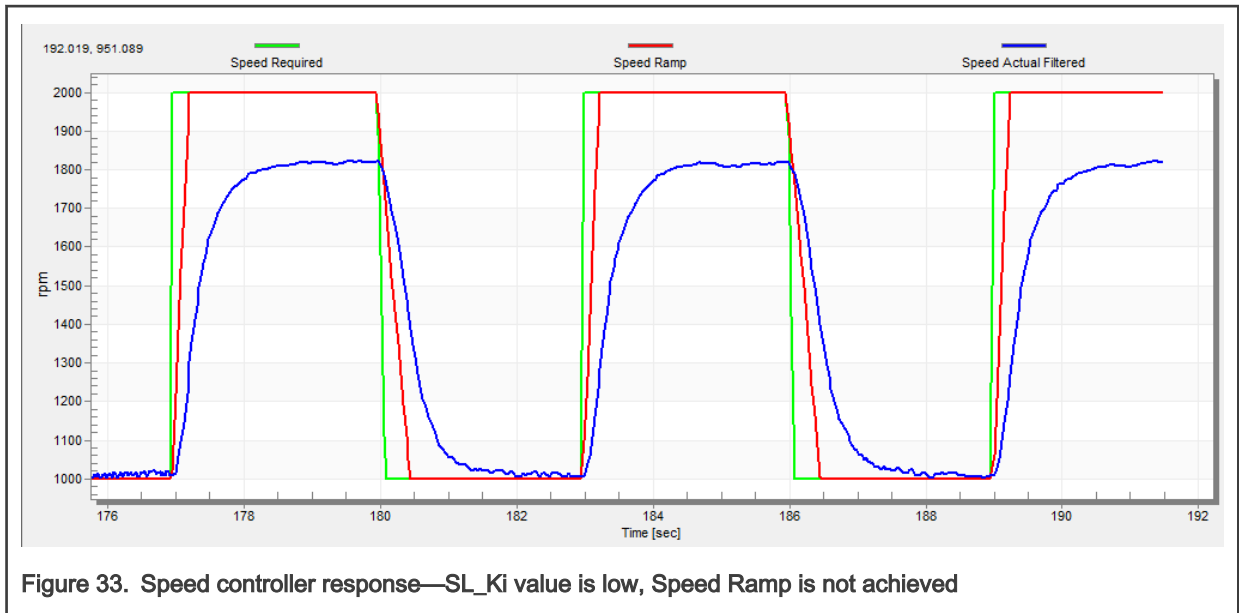
1. The bandwidth and attenuation parameters of the BEMF observer and the tracking observer can be tuned. Navigate to the "Sensorless" MCAT tab.
2. Set the required bandwidth and attenuation of the BEMF observer—the bandwidth is typically set to a value close to the current loop bandwidth.

3. Set the required bandwidth and attenuation of the tracking observer—the bandwidth is typically set in the range of 10 – 20 Hz for most low-dynamic drives (fans, pumps).
4. Click the “Update target” button to apply the changes to the MCU.
5. Select the “Observer” recorder from the FreeMASTER project tree and check the observer response in the “Observer” recorder.

8.17 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 the example response in [Figure 33](#)). There are dozens of approaches to tune the PI controller constants. The following steps provide an approach to set and tune the speed PI controller for a PM synchronous motor:

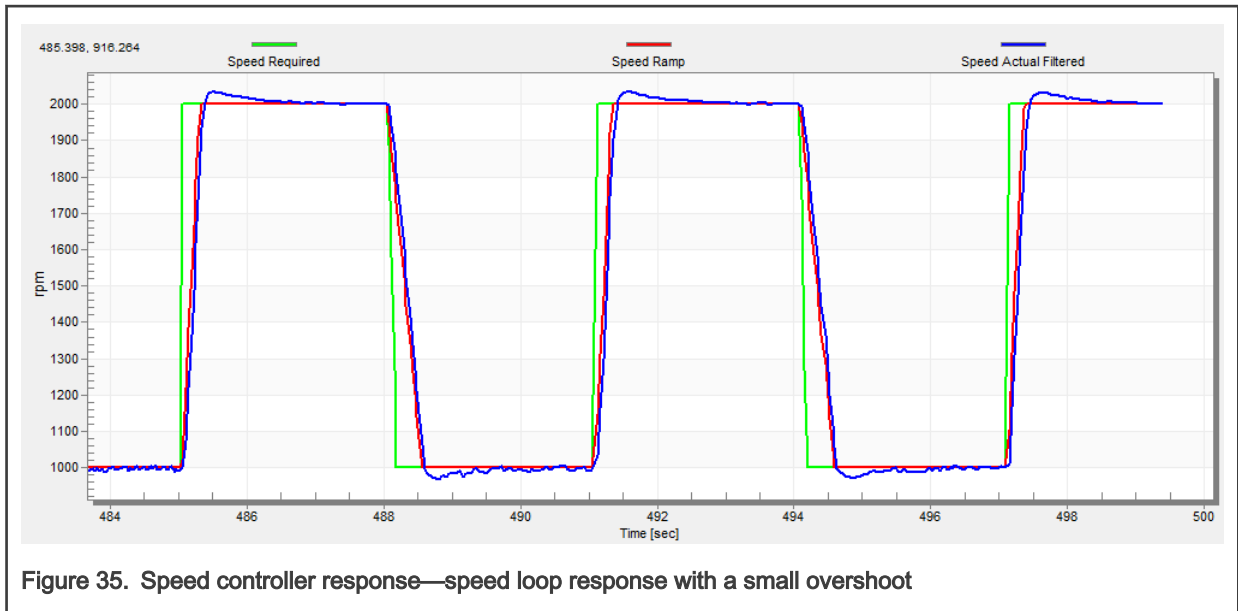
1. Select the “Speed Controller” option from the FreeMASTER project tree.
2. Select the “Speed loop” tab.
3. Check the “Manual Constant Tuning” option—that is, the “Bandwidth” and “Attenuation” fields are disabled and the “SL_Kp” and “SL_Ki” fields are enabled.
4. Tune the proportional gain:
 - Set the “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.
 - Click the “Update Target” button to apply the changes to the MCU.
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.
 - Click the “Update target” button to apply the changes to the MCU.
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” (see [Figure 33](#)).



- The “SL_Kp” value is low, the “Speed Actual Filtered” greatly overshoots, and the long settling time is unwanted (see [Figure 34](#)).



- The speed loop response has a small overshoot and the “Speed Actual Filtered” settling time is sufficient. Such response can be considered optimal (see [Figure 35](#)).



8.18 Position P controller tuning

The position control loop can be tuned using the proportional gain “M1 Position Loop Kp Gain” variable. It is a proportional controller that can be used to unpretend the position-control systems. The key for the the optimal position response is a proper value of the controller, which simply multiplies the error by the proportional gain (K_p) to get the controller output. The predefined base value can be manually changed. An encoder sensor must be used for a working position control. The following steps provide an example of how to set the position P controller for a PM synchronous motor:

1. Select the “Position Controller” scope in "Position Control" tab in the FreeMASTER project tree.
2. Tune the proportional gain in the position P controller constant:
 - Set a small value of “PL_Kp” (M1 Position Loop Kp Gain).
 - Select the position control, and set the required position in "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.

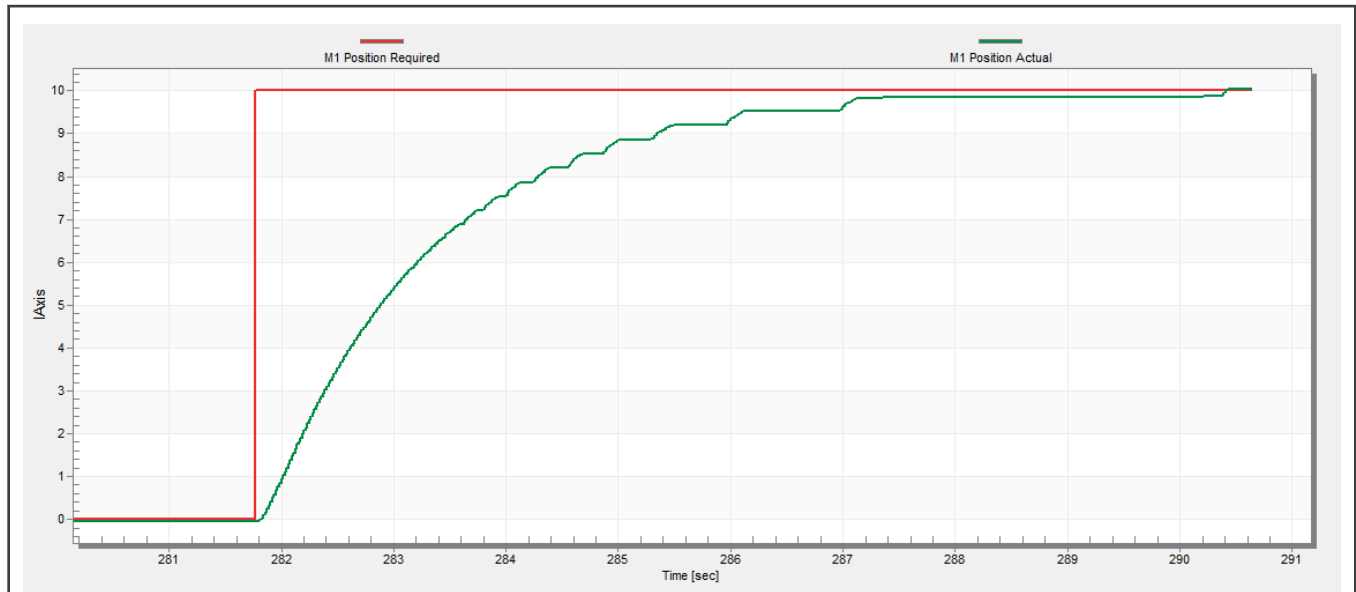


Figure 36. Position controller response—PL_Kp value is low, the actual position response is very slow

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

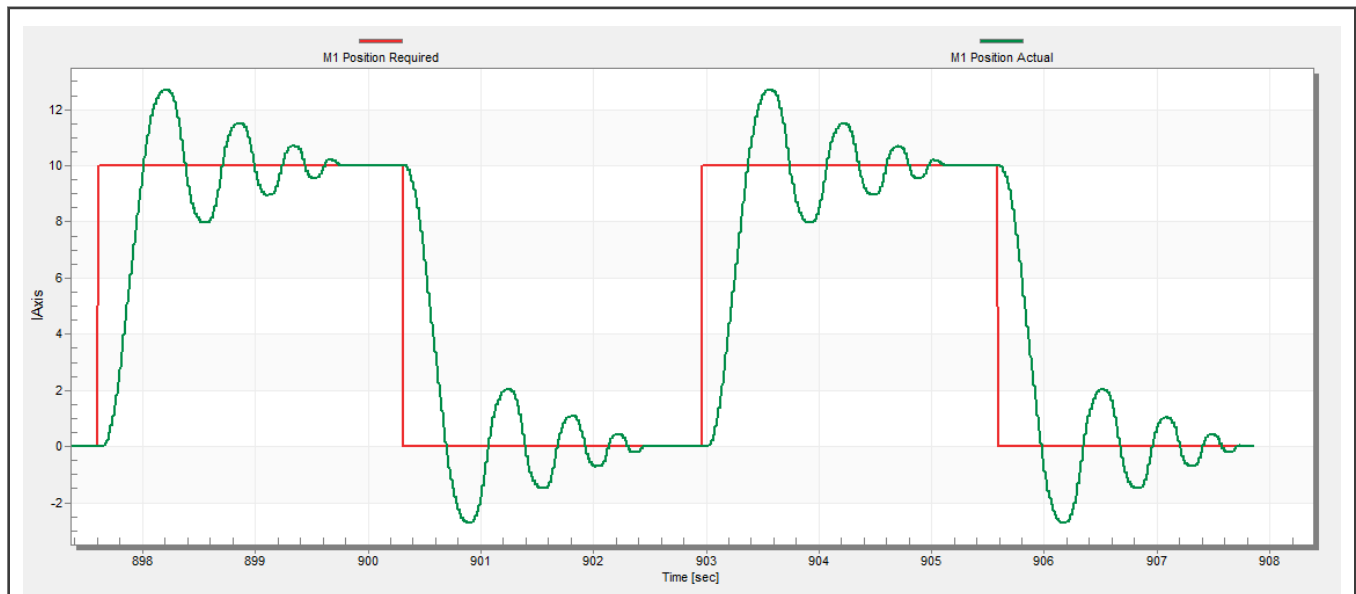


Figure 37. Position controller response—PL_Kp value is too high and the actual position overshoots

The “PL_Kp” value and the actual position response are optimal.

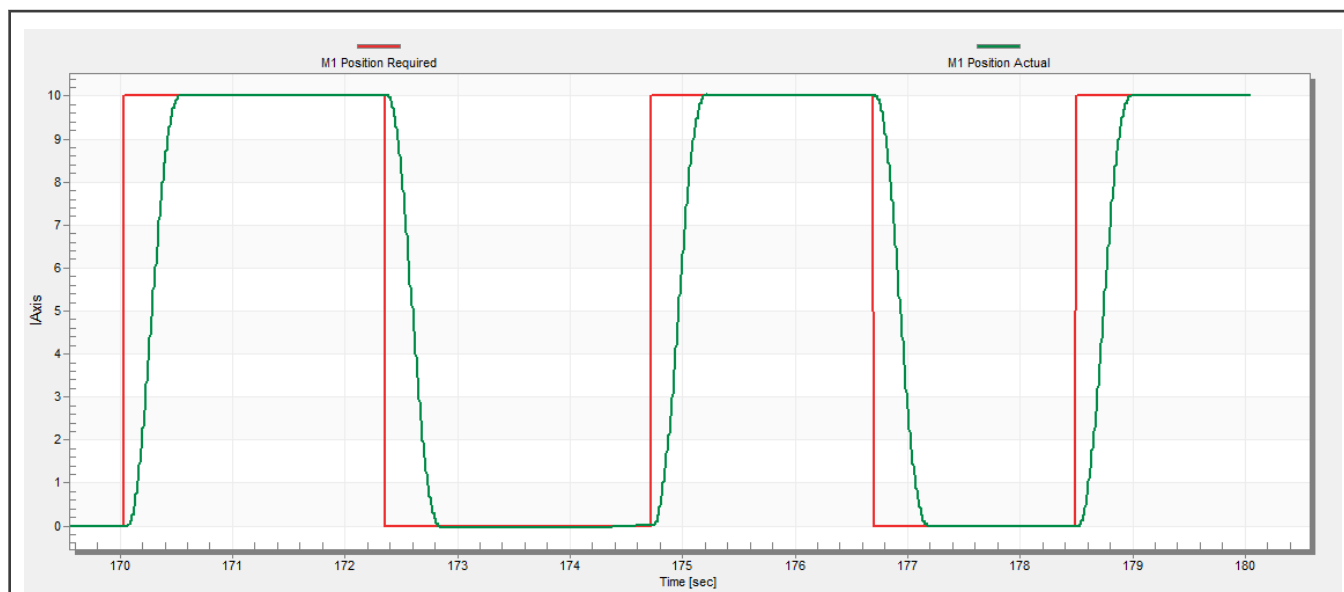


Figure 38. Position controller response—the actual position response is good

Chapter 9

Conclusion

This application note describes the implementation of the sensor and sensorless Field-Oriented Control of a 3-phase PMSM on the NXP MIMXRT1xxx EVK board with the FRDM-MC-LVPMSM NXP Freedom Development Platform. The hardware-dependent part of the control software is described in [Hardware setup](#). The motor-control application timing is described in [RT crossover processors features and peripheral settings](#) and the peripheral initialization is described in [Motor-control peripheral initialization](#). The motor user interface and remote control using FreeMASTER are as follows. The motor parameters identification theory and the identification algorithms are described in [Identifying parameters of user motor](#).

Chapter 10

Acronyms and abbreviations

Table 19. 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
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
PDB	Programmable Delay Block
PI	Proportional Integral controller
PLL	Phase-Locked Loop
PMSM	Permanent Magnet Synchronous Machine
PWM	Pulse-Width Modulation
QD	Quadrature Decoder
TMR	Quad Timer
USB	Universal Serial Bus
XBAR	Inter-Peripheral Crossbar Switch
IOPAMP	Internal operational amplifier

Chapter 11

References

These references are available on www.nxp.com:

1. *Sensorless PMSM Field-Oriented Control* (document [DRM148](#))
2. *Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM* (document [AN4642](#))
3. *PMSM Field-Oriented Control on MIMXRT10xx EVK User's Guide* (document [PMSMFOCRT10xxUG](#))
4. *PMSM Field-Oriented Control on MIMXRT10xx EVK* (document [AN12214](#))

Chapter 12

Useful links

1. MCUXpresso SDK for Motor Control www.nxp.com/motorcontrol
2. [i.MX RT1024-EVK board](#)
3. [i.MX RT1060-EVK board](#)
4. [i.MX RT1064-EVK board](#)
5. [i.MX RT1160-EVK board](#)
6. [i.MX RT1170-EVK board](#)
7. [i.MX RT Crossover MCUs](#)
8. [FRDM-MC-PMSM Freedom Development Platform](#)
9. [MCUXpresso IDE - Importing MCUXpresso SDK](#)
10. [MCUXpresso Config Tool](#)
11. MCUXpresso SDK Builder (SDK examples in several IDEs) <https://mcuxpresso.nxp.com/en/welcome>

Chapter 13

Revision history

[Revision history](#) summarizes the changes done to the document since the initial release.

Table 20. Revision history

Revision number	Date	Substantive changes
0	01/2022	Initial release
1	07/2022	MCAT with file management

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, UMEMS, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamiQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

The ARM logo consists of the lowercase letters "arm" in a bold, blue, sans-serif font.

© NXP B.V. 2022.

All rights reserved.

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 07/2022

Document identifier: PMSMRT1170