

PMSMMC56F82000EVK

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

Rev. 0 — 12 January 2023

User guide

1 Introduction

This user's guide describes the implementation of the sensorless Motor Control software for the 3-phase Permanent Magnet Synchronous Motor (PMSM), including the motor parameters identification algorithm, on the NXP MC56F81xxx and MC56F83xxx MCUs based on the 56800EX DSP architecture. The sensorless control software and PMSM control theory in general are described in *Sensorless PMSM Field-Oriented Control* (document [DRM148](#)). The NXP Freedom board ([FRDM-MC-LVPMSM](#)) is used as hardware platform for the PMSM control reference solution. The hardware-dependent part of the sensorless control software, including a detailed peripheral setup and the Motor Control (MC) peripheral drivers, is addressed as well. The motor parameters identification theory and algorithms are described 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 present a simple and user-friendly way for motor parameter identification, algorithm tuning, software control, debugging, and diagnostics.

This document provides instructions for running and controlling the PMSM project using the MC56F81000-EVK, MC56F83000-EVK boards with the FRDM-MC-LVPMSM power stage or the TWR-MC56F8200, TWR-MC56F8400 boards with the TWR-MC-LV3PH power stage. The software provides the sensorless field-oriented speed, torque, and scalar control. You can control the application using the board buttons or the FreeMASTER application. 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 is described in the following sections. For more information, visit www.nxp.com/motorcontrol_pmsm.

Available motor control examples, supported motors and possible control methods are listed in [Table 1](#).

Table 1. Available examples and control methods

Example	Supported motor	Possible control methods in SDK example				
		Scalar & Voltage	Current FOC (Torque)	Sensorless Speed FOC	Sensored Speed FOC	Sensored Position FOC
pmsm_snsless	Linux 45ZWN24-40 (default motor)	✓	✓	✓	N/A	N/A

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



2 Hardware setup

The PMSM Field-Oriented Control (FOC) application runs on two platforms: FRDM-MC-LVPMSM development platform with the MC56F81000-EVK or MC56F83000-EVK development board OR TWR-MC-LV3PH development platform with the TWR-MC56F8200 or TWR-MC56F8400 development board. Both platforms (Freedom and Tower) use Linux 45ZWN24-40 permanent magnet synchronous motor as a default motor.

2.1 TWR-MC-LV3PH

This evaluation board, in a Tower form factor, effectively turns a Tower development board into a complete motor control reference design, compatible with existing Tower DSC development boards. This board provides all necessary feedback signals for driving PMSM and BLDC motors.

The TWR-MC-LV3PH has power supply input voltage 12-24VDC, extended up to 50VDC with reverse polarity protection circuitry. Auxiliary power supply 5 and 3.3 VDC is created to provide power supply for Tower MCU boards. Output current is up to 5 amps RMS. The inverter itself is realized with 3-phase bridge inverter (6-MOSFET's) and 3-phase MOSFET gate driver. Analog quantities such as 3-phase motor currents, 3-phase motor back-EMF voltage, DC bus voltage and DC bus current are sensed on this board. There is an interface for speed/position sensors (Encoder Hall) and connector for braking resistor. There are also user LED, power-on LED and 6 PWM LED diodes for diagnostics. The block diagram of complete Tower motor control development kit is in [Figure 1](#)

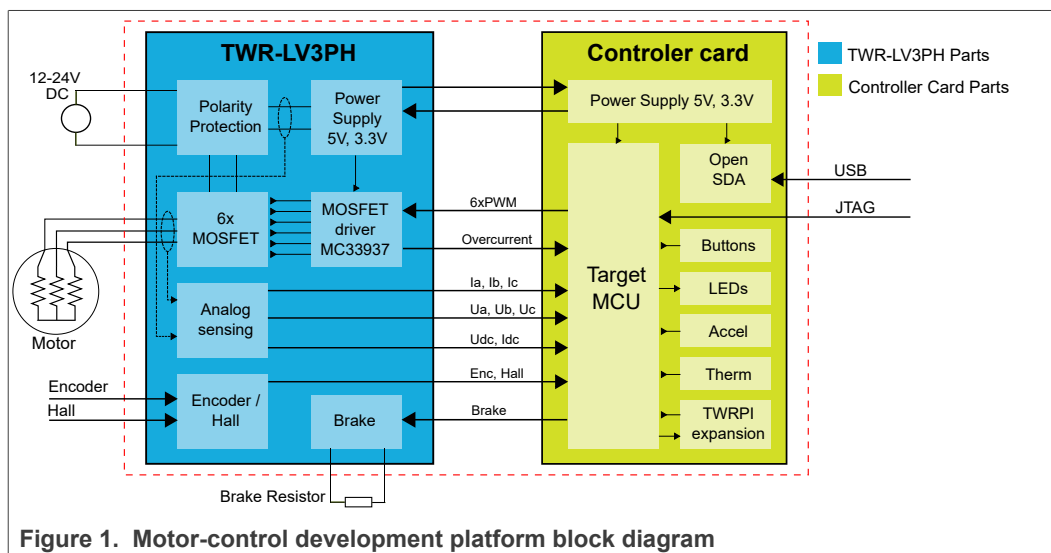
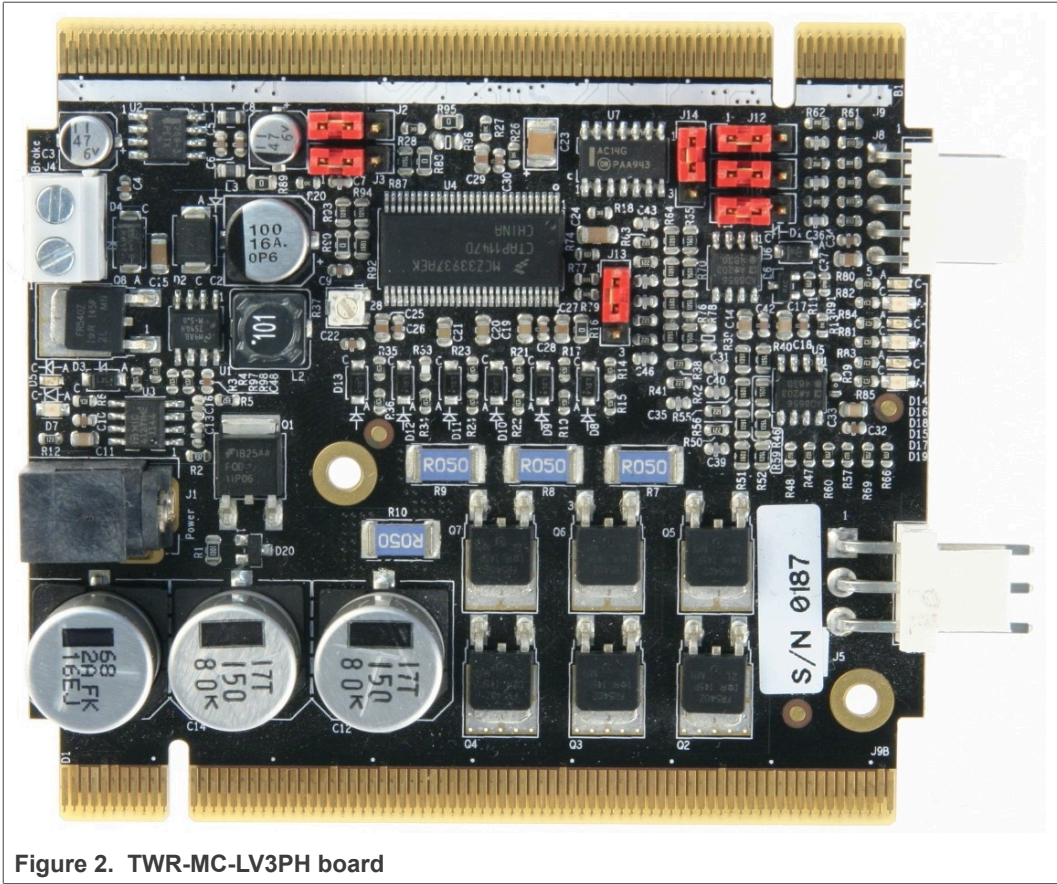


Figure 1. Motor-control development platform block diagram



The TWR-MC-LV3PH board requires a correct jumper setting for FOC control.

Table 2. TWR-MC-LV3PH jumper setting

Jumper	Setting
J2	1-2
J3	1-2
J10	1-2
J11	1-2
J12	1-2
J13	1-2
J14	1-2

For more information about the Tower 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 3](#).

Table 3. Linix 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	-



Figure 3. Linix 45ZWN24-40 permanent magnet synchronous motor

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 TWR-56F8200

The TWR-MC56F8200 development board is an ideal platform for evaluation and development with the MC56F82xxx MCU based on the 56800EX DSP architecture. The board includes the high-performance on-board debug probe. Configure the jumper settings according to [Table 4](#) for the motor-control application to work properly.

Table 4. TWR-56F8200 jumper settings

Jumper	Setting	Jumper	Setting	Jumper	Setting
J1	open	J9	3-4	J20	close
J2	open	J11	2-3	J21	1-2, 3-4, 5-6, 7-8
J4	2-3	J15	close	J23	open
J5	2-3	J16	1-2, 3-4		
J7	2-3	J17	open		
J8	3-4	J19	open		

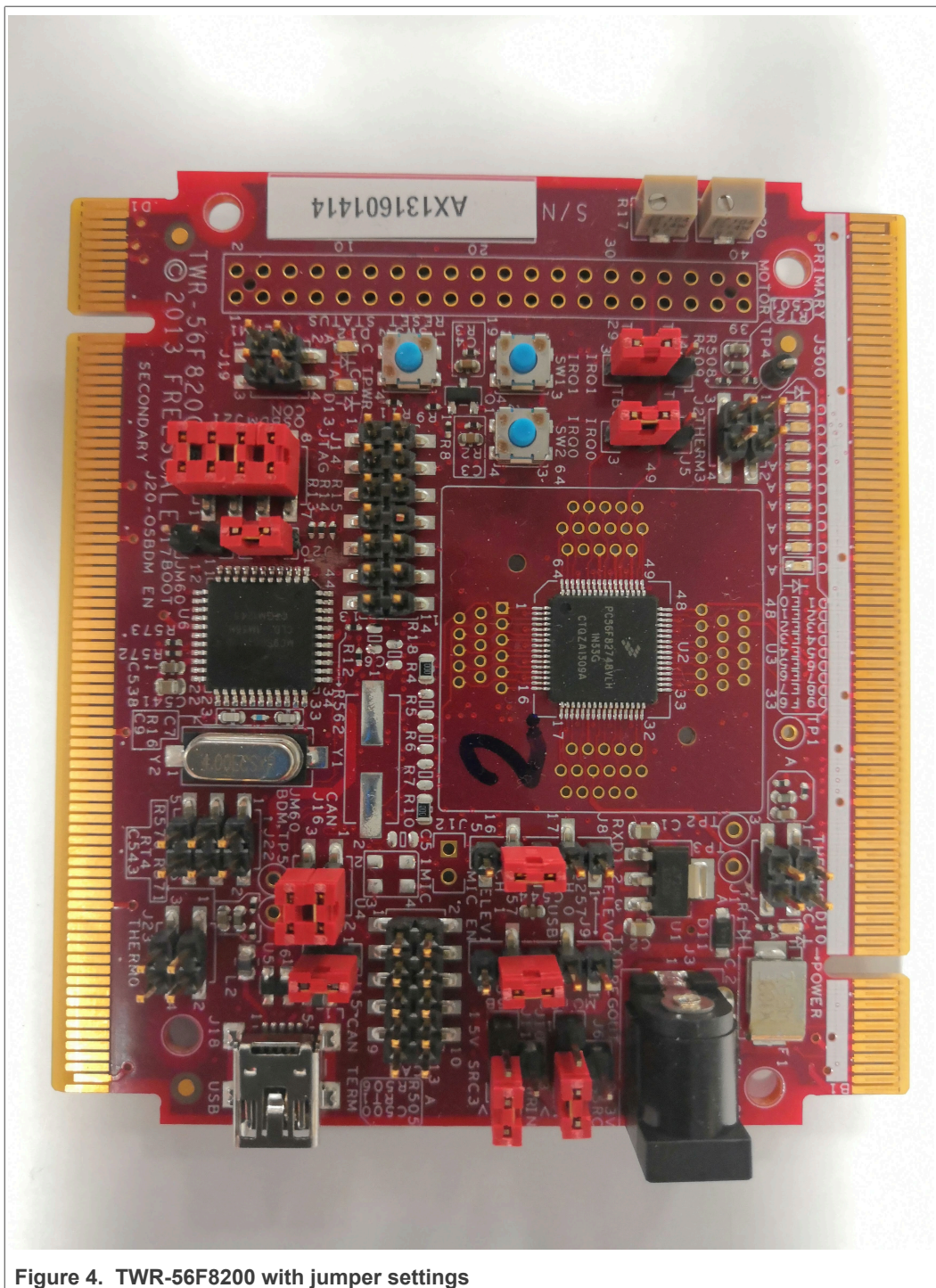


Figure 4. TWR-56F8200 with jumper settings

2.3.1 Hardware assembling

1. Plug the TWR-MC-LV3PH power stage to the TWR-MC56F8200 board.
2. Connect the 3-phase motor wires to the screw terminals (J5) on the TWR-MC-LV3PH power stage.
3. Plug the USB cable from the USB host to the USB connector (J18) on the TWR-MC56F8200 board.

4. Compile the project and program the TWR-MC56F8200 board prior to plugging 24V DC to TWR-MC-LV3PH. Plugging 24V DC to a non-programmed board could cause a shoot through top and bottom transistors.
5. Plug the 24-V DC power supply to the DC power connector on the TWR-MC-LV3PH power stage (J1).

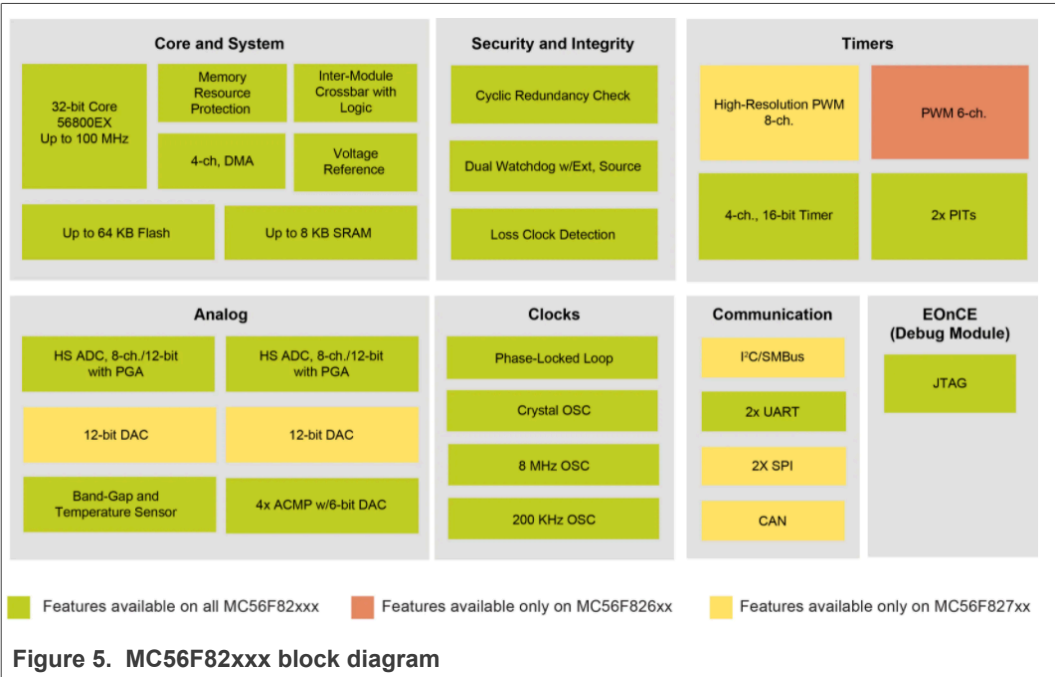
3 MC56F8xxxx series features and peripheral settings

This section describes the peripheral settings and application timing. The MC56F8xxxx MCU series contains a high-performance 56800EX DSP core. It combines the best features of Microcontrollers (MCUs) and powerful Digital Signal Processing (DSP) capabilities in a single chip. The MC56F8xxxx MCU series are optimized for digital signal processing and applications ranging from general embedded markets to motor control and power conversion. In addition, the MC56F8xxxx MCU series features four scalable families with broad package and memory options, as well as the comprehensive MCUXpresso software and tools ecosystem and low-cost development boards.

3.1 MC56F82xxx

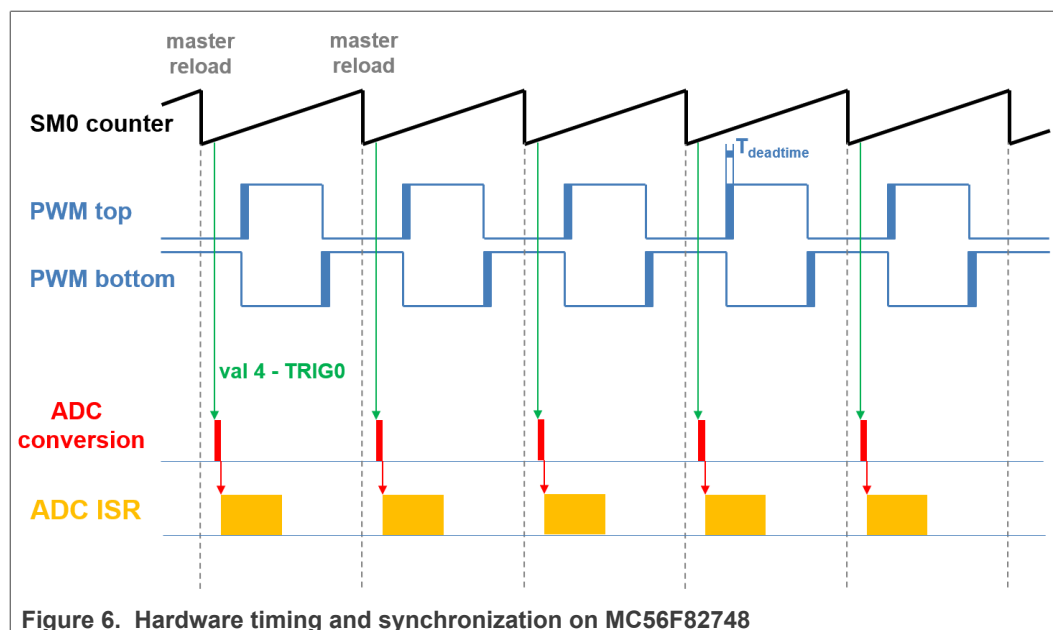
The MC56F82xxx DSP MCU family offers outstanding power consumption at runtime with exceptional performance, precision, and control. This high-efficiency family is based on the high-performance 56800EX DSP core with a frequency of up to 100 MHz. This family extends the DSC products' cost performance to the best.

The MC56F82xxx family offers extremely cost-effective solution for power conversion and motor control applications.



3.1.1 MC56F82748 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. The PWM frequency is equal to the FOC calculation frequency. The timing diagram is shown in [Figure 6](#).



- The top signal shows the eFlexPWM SM0 counter. The dead time is emphasized at the PWM top and PWM bottom signals.
- The eFlexPWM SM0 also generates a trigger for the ADC with a short delay. This delay ensures correct current sampling at duty cycles close to 100 %.
- When the ADC conversion is completed, the ADC ISR (ADC interrupt) is entered. The FOC calculation is done in this interrupt.

3.1.2 MC56F82748 peripheral settings

All peripherals are configured using the MCUXpresso Config Tools. The MCUXpresso Config Tools set pins, clocks, and peripherals by generating corresponding sources (*pin_mux.c.h*, *clock_config.c.h*, *pripherals.c.h*, *Flash_config.h*, *freemaster_cfg.h*, and *startup_clock_mode_config.h*). On MC56F82748, the eFlexPWM is used for 6-channel PWM generation with DC bus over-current protection. The 12-bit cyclic ADC is used for the phase currents and DC-bus voltage measurement. The PIT Timer is used for slow interrupt generation. The peripheral description is well described and depicted in the MCUXpresso Config Tools after opening *mc_pmsm_sdm.mex* or *mc_pmsm_ldm.mex*. For more information about the MCUXpresso Config Tools, see [MCUXpresso Config Tools](#).

3.2 CPU load and memory usage

The following information applies to the application built using the CodeWarrior v11.1 IDE in the Debug and Release configurations with the compiler updated to build 1151. The below tables show the memory usage and CPU load. The memory usage is calculated from the *.map* linker file, including the 4-KB FreeMASTER recorder buffer allocated in RAM. The CPU load is measured using the QuadTimer, which is driven directly by the CPU clock. 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 10 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 (10 KHz).

f_{CPU} - the 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 5. MC56F82748 load and memory usage

Configuration	flash_idm_lpm_release	flash_sdm_lpm_release
Flash memory	35 968 B	35 880 B
RAM memory	5 696 B	5 632 B
Maximum CPU load	48.1 %	

Note: None of configurations above features FreeMASTER TSA table. Both configurations feature FreeMASTER recorder which consumes **1024 B** of RAM and can be eliminated in the final code. The stack that consumes the RAM is **1024 B** and it can be decreased in the final code.

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 a cloned PMSM project is shown below.

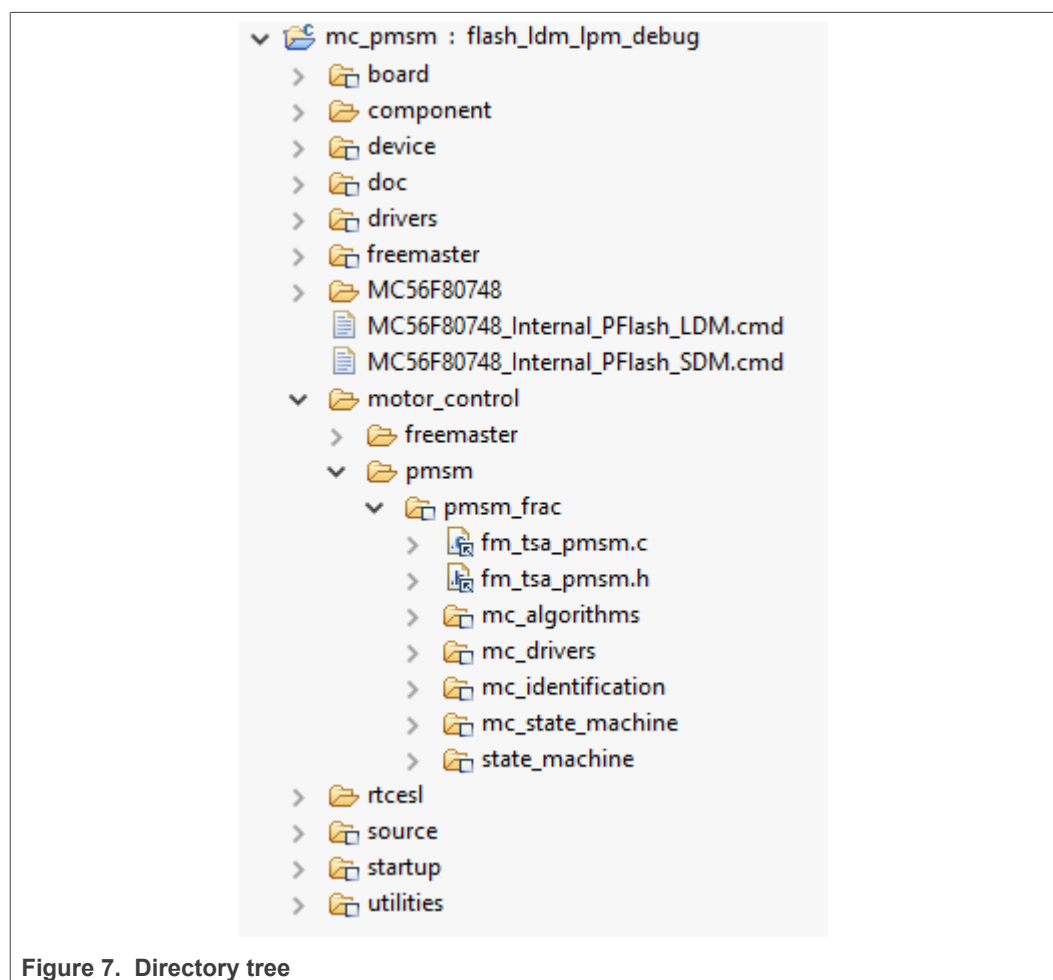


Figure 7. Directory tree

The main project folder contains the following folders and files:

- *.project* and *.cproject*—project files for CodeWarrior IDE.
- *board/board.c* and *.h*—contains the definitions and functions related to the board like LEDs and buttons.

- *board/clock_config.c and .h*—contains the MCU clock setup functions. These files are generated by the MCUXpresso Config Tools.
- *board/peripherals.c and .h*—contains the MCU peripherals' setup functions. These files are generated by the MCUXpresso Config Tools.
- *board/pin_mux.c and .h*—contains the MCU pins' setup functions. These files are generated by the MCUXpresso Config Tools.
- *device*—contains the MCU-related files like registers' definitions.
- *doc*—documentation folder.
- *drivers*—contains the necessary SDK drivers for MCU peripherals.
- *freemaster*—contains the FreeMASTER embedded part of code.
- *motor_control/pmsm/pmsm_frac*—folder containing common motor control code
 - *mc_algorithms*—contains the main control algorithms used to control the FOC and speed control loop.
 - *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.
 - *fm_tsa_pmsm.c and .h*—TSA tables with variables used in FreeMASTER.
- *middleware/motor_control/freemaster*—contains the FreeMASTER *.pmp file and all MCAT-related files.
- *rtcesl*—Real-Time Control Embedded Software Libraries with elementary algorithms used to build the FOC.
- *source/freemaster_cfg.h*—FreeMASTER configuration file containing FreeMASTER communication and features setup. This file is generated by the MCUXpresso Config Tools.
- *source/m1_pmsm_appconfig.h*—contains the definitions of constants for the application control processes, parameters of the motor and controllers, 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, this file is re-generated at the end of the tuning process.
- *source/main.c*—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.
- *source/mc_periph_init.c and .h*—contains the motor-control peripheral drivers' initialization functions and macros for changing ADC channels assigned to the phase currents and board voltage. These files are specific for the board and MCU used.

5 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.

6 User interface

The application contains the demo mode to demonstrate motor rotation. You can operate it using a button on the board or via FreeMASTER. 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 debugger or the virtual serial port circuit included in the boards.

The application can be controlled using these two interfaces:

- The user's button on the development board (controlling the demo mode). See *pin_mux.c/.h* which button is configured for this function.
- 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.

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

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

7.1 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.pmpx*. 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.

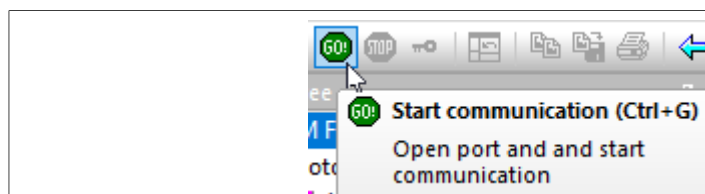


Figure 8. Green “GO” button placed in top left-hand corner

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.

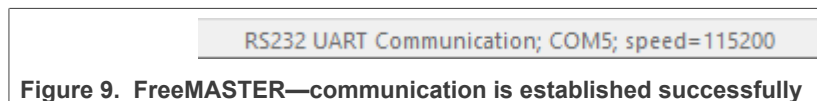


Figure 9. FreeMASTER—communication is established successfully

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.

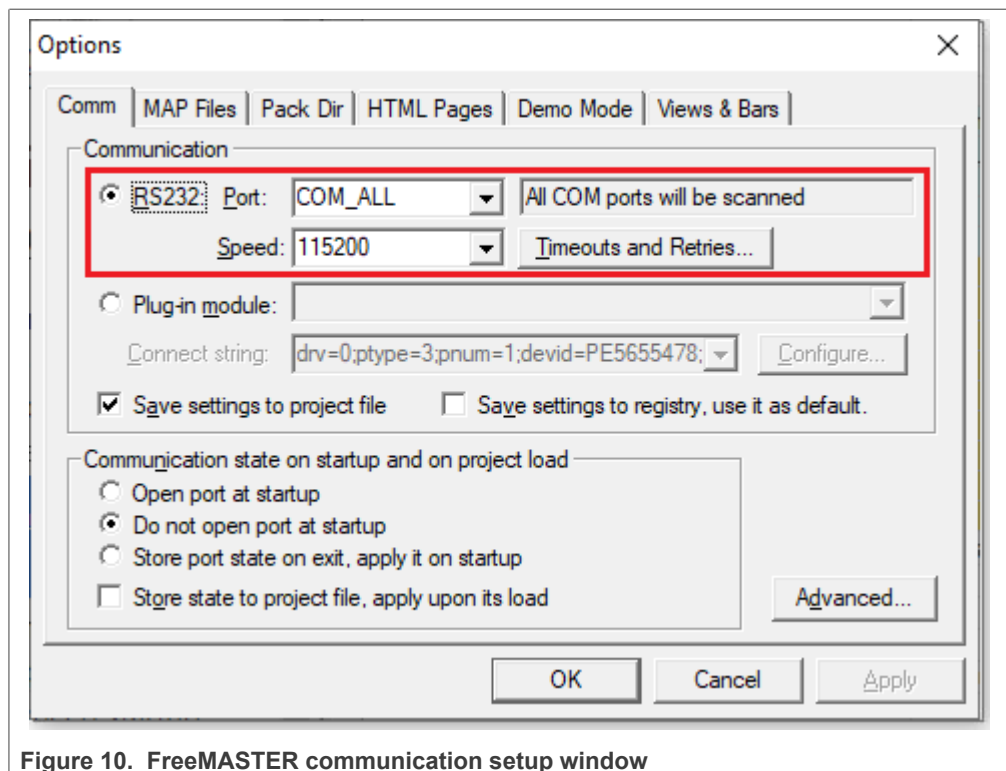


Figure 10. FreeMASTER communication setup window

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

7.2 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

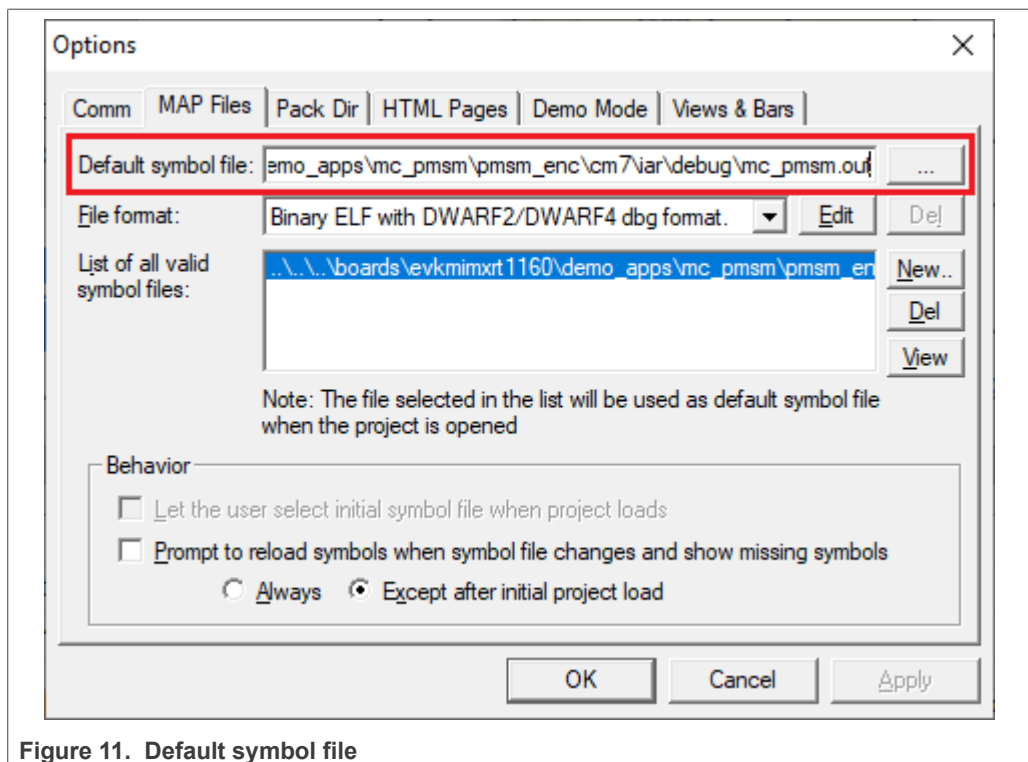


Figure 11. Default symbol file

- Click to OK and restart FreeMASTER communication.

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

7.3 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 12](#). 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

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

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.

Note: In the dual motor control application, MCAT interface is not available.

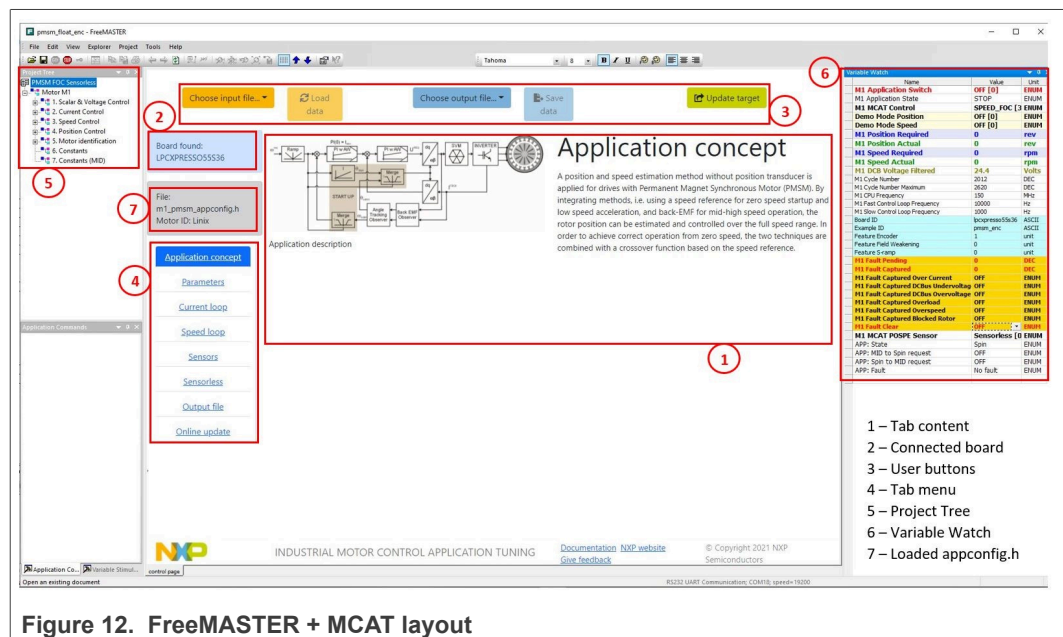


Figure 12. FreeMASTER + MCAT layout

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.

7.4 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.

7.4.1 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.

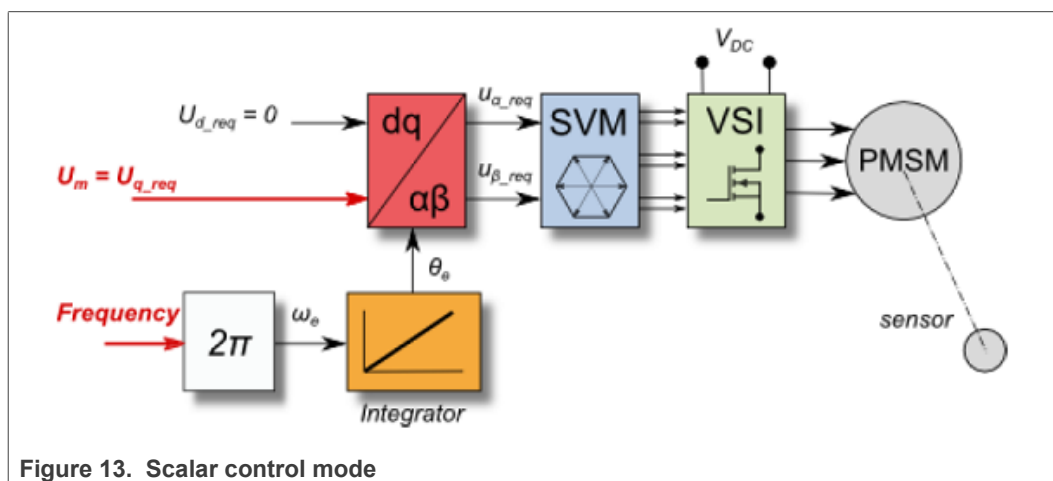
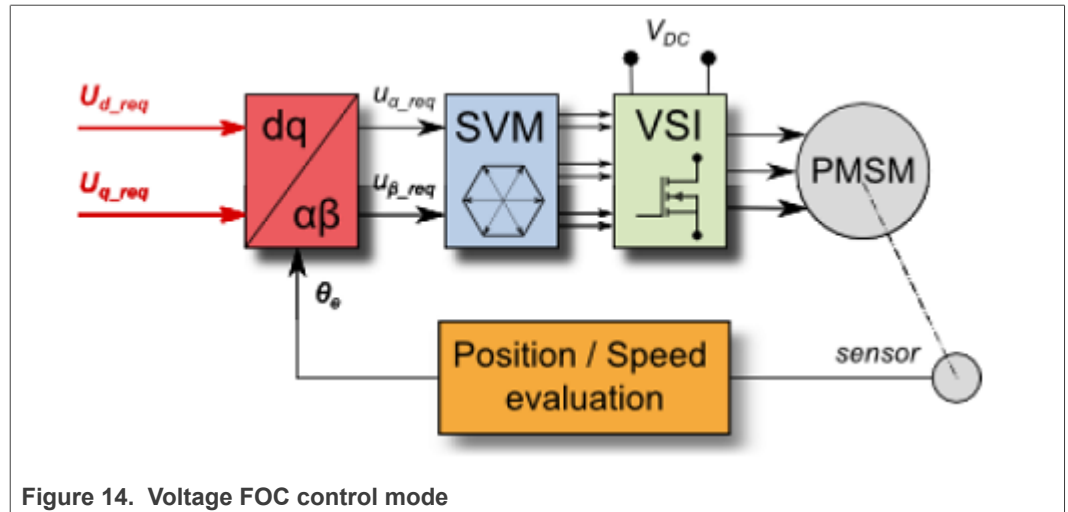
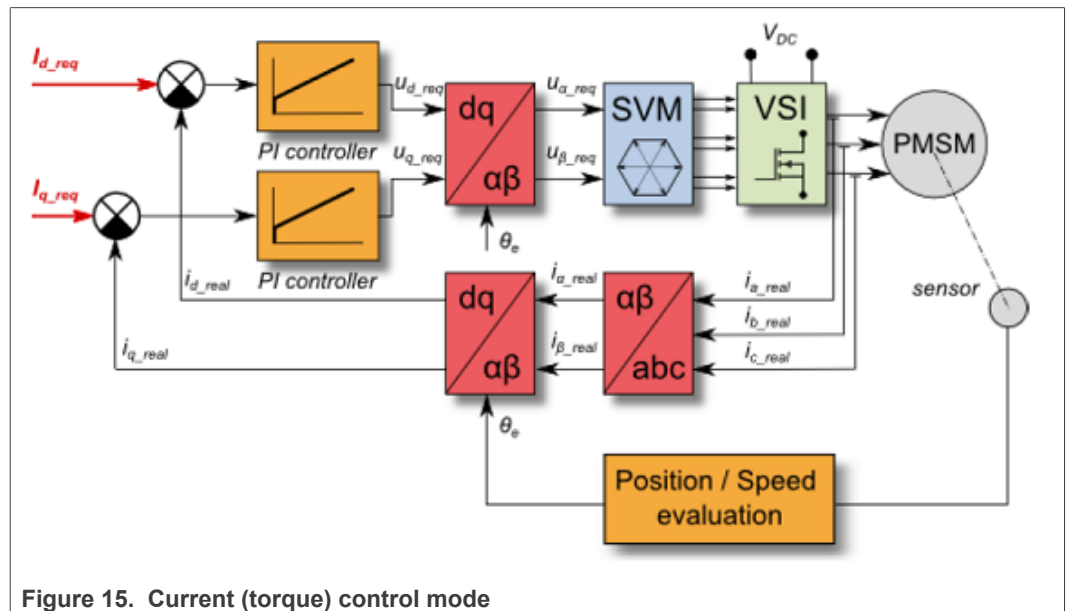


Figure 13. Scalar control mode

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.



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.

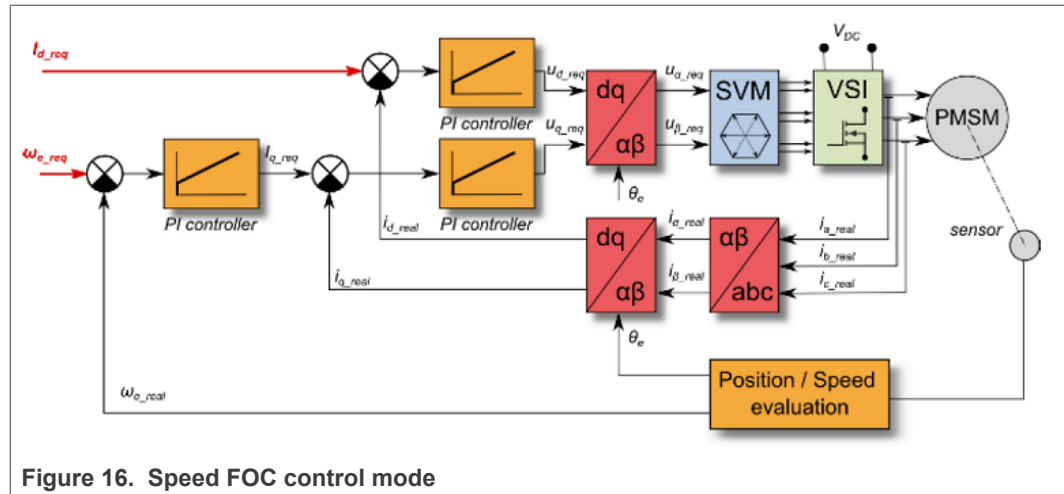


Figure 16. Speed FOC control mode

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.

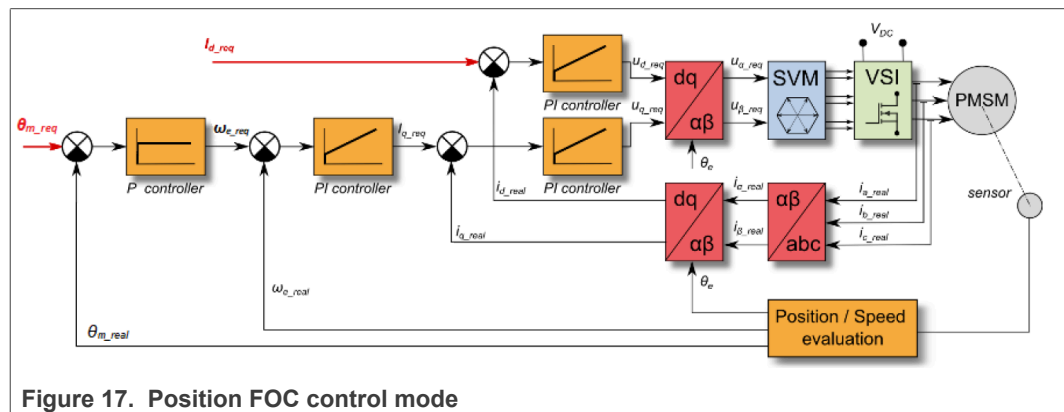


Figure 17. Position FOC control mode

7.5 Switch between Spin and MID

User can switch between two modes of application: Spin and MID (Motor identification). Spin mode is used for control PMSM (see [Section "Motor Control Modes"](#)). MID mode is used for motor parameters identification (see [Section "Identifying parameters of user motor"](#)). Navigate to *Motor Identification* subblock in the FreeMASTER project tree. Actual mode of application is shown in *APP: State* variable. The mode can be changed by *APP: Switch request Spin/MID* variable. The transition between Spin and MID can be done only if actual mode is in a defined stop state (e.g. MID is not in progress or motor is stopped). The result of the change mode request is shown in *APP: Fault* variable. *Fault MID to Spin* occurs when parameters identification still runs or MID state machine is in the fault state. *Fault Spin to MID* occurs when *M1 Application switch* variable watch is ON or *M1 Application state* variable watch is not STOP.

7.6 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 and quadrature inductance L_q . 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 fixed-point arithmetics. Each available MID algorithm is described in [Section "MID algorithms"](#). MID is controlled via the FreeMASTER "Motor Identification" page shown in [Figure 18](#).

Name	Value	Unit
MID: On/Off	OFF	ENUM
MID: State	STOP	ENUM
APP: Switch request Spin/MID	No request	ENUM
MID: Status	MID is stopped	ENUM
MID: Measurement Type	PP_ASSIST	ENUM
MID: Fault	No fault	ENUM
APP: State	MID	ENUM
APP: Fault	No fault	ENUM
DIAG: Fault Captured	0	DEC
DIAG: Fault clear	0	DEC
MID: Measured Rs	0	Ohm
MID: Measured Ld	0	H
MID: Measured Lq	0	H
MID Pp IdReqOpenLoop	0.824799	A
MID Pp SpeedElReq	439.893	rpm
MID: Config El Mode Estim RL	0	DEC
MID: Config El I DC nominal	5	A
MID: Config El I DC positive max	6	A
MID: Config El I DC negative max	-6	A
MID: Config El I DC (estim Ld)	0	A
MID: Config El I DC (estim Lq)	5	A
MID: Config El DQ-switch	Ld meas	ENUM
MID: Config El I DC req (d-axis)	0	A
MID: Config El I DC req (q-axis)	0	A
MID: Config El I AC req	0	A
MID: Config El I AC frequency	0	Hz

Figure 18. MID FreeMASTER control

7.6.1 Motor parameter identification using MID

The whole MID is controlled via the FreeMASTER "Variable Watch". Motor Identification (MID) sub-block shown in [Figure 18](#). The motor parameter identification workflow is following:

1. Set the *MID: On/Off* variable to OFF.
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.
3. Set the measurement configuration parameters in the *MID: Config* set of variables.
4. Start the measurement by setting *MID: On/Off* to ON.
5. Observe the *MID: Status* variable which indicates whether identification runs or not. Variable *MID: State* indicates actual state of the MID state machine. Variable *MID: Fault* indicates fault captured by estimation algorithm (e.g. incorrect measurement parameters). Variable is cleared automatically. Variable *DIAG: Fault Captured* indicates captured hardware faults (e.g. DC bus undervoltage). Variable is cleared by setting "On" to *DIAG: Fault clear* variable.
6. If the measurement finishes successfully, the measured motor parameters are shown in the *MID: Measured* set of variables and *MID: State* goes to STOP.

Table 6. MID: Fault variable

Fault mask	Description	Troubleshooting
b#0001	Error during initialization electrical parameters measurement.	Check whether inputs to the <i>MCAA_EstimRLInit_F16</i> are valid.
b#0010	Electrical parameters measurement fault. Some required value cannot be reached or wrong measurement configuration.	Check whether measurement configuration is valid.

Table 7. DIAG: Fault Captured variable

Fault mask	Description
b#0001	Overcurrent fault occurs.
b#0010	Undervoltage fault occurs.
b#0100	Overvoltage fault occurs.

7.7 MID algorithms

This section describes how each available MID algorithm works.

7.7.1 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_F32* function from [AMMCLib](#).

7.7.2 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_F32* function from [AMMCLib](#).

7.7.3 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 Pp IdReqOpenLoop*. The electrical position is generated by integrating the open-loop frequency *MID Pp SpeedEIReq*. If the rotor does not move after the start of PP_ASSIST assistant, stop the assistant, increase *MID Pp IdReqOpenLoop*, and restart the assistant.

7.8 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_F16* must be called before the first use of *MCAA_EstimRL_F16*. Function *MCAA_EstimRL_F16* 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.

7.8.1 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 .

7.8.2 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.

7.8.3 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 (L_d , L_q)" recorder. User has to specify following parameters before parameters estimation:

- *MID: Config EI I DC (estim L_d)* - Current to determine L_d . In most cases 0 A.
- *MID: Config EI I DC (estim L_q)* - 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 L_d)*, *MID: Config EI I DC (estim L_q)*, *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.

7.8.4 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 " L_d " or " L_q " scopes. The following parameters can be changed during the runtime:

- *MID: Config EI 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.

7.9 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 8](#) 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 8. 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
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 9](#)).

Table 9. 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 10. 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.

7.10 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 19). 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!

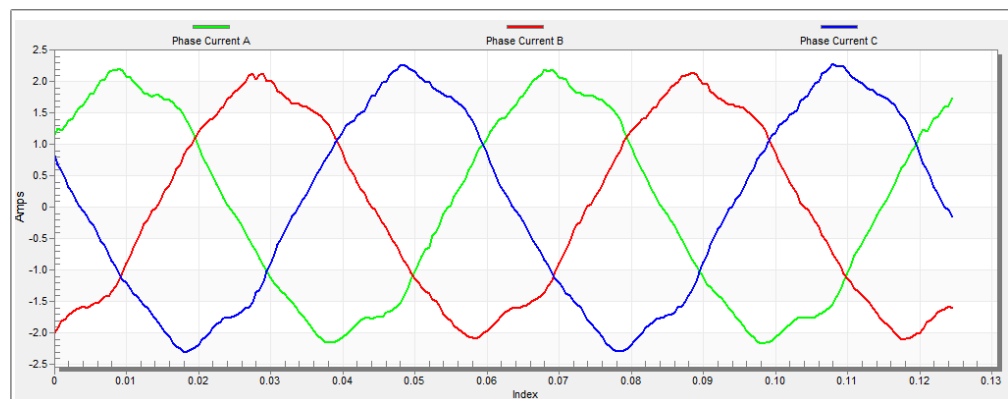


Figure 19. Phase currents

- 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 20) 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.

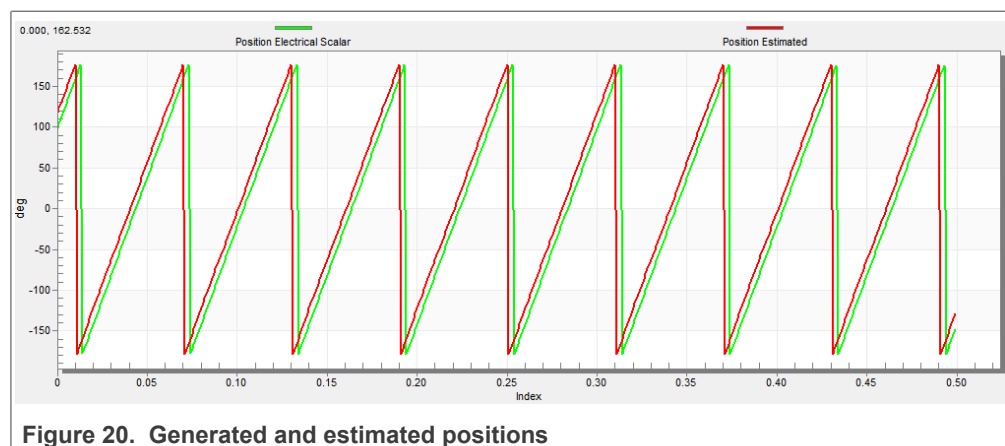


Figure 20. Generated and estimated positions

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.

7.11 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.

7.12 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 21](#)).

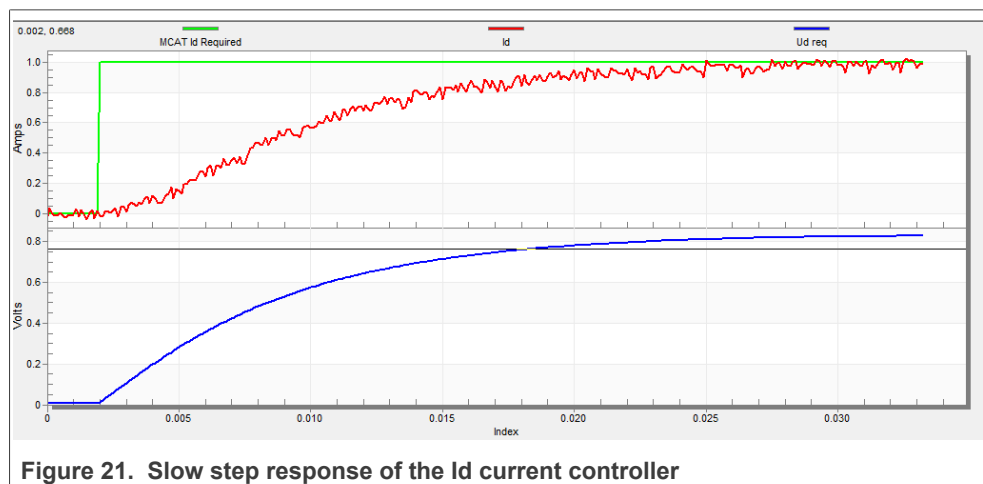


Figure 21. Slow step response of the Id current controller

- The loop bandwidth (400 Hz) is optimal and the response time of the Id current is sufficient (see [Figure 22](#)).

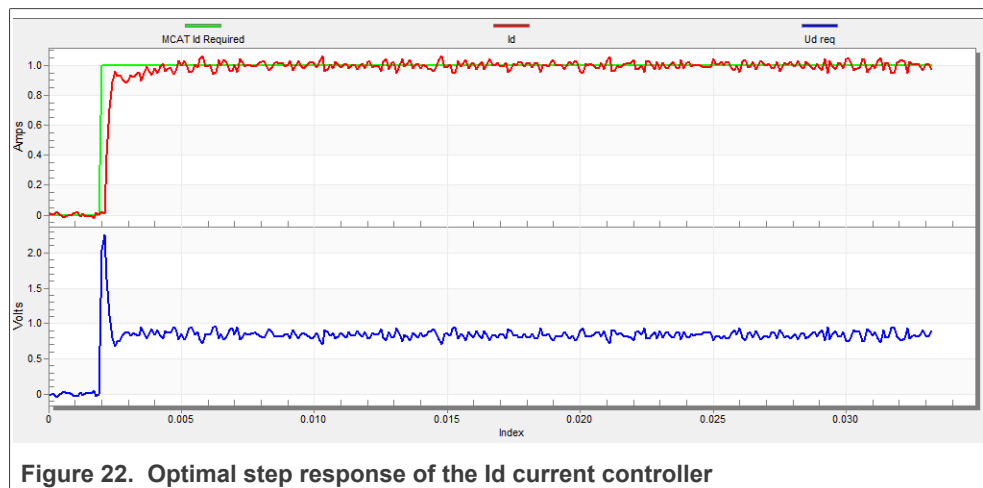


Figure 22. Optimal step response of the Id current controller

- The loop bandwidth is high (700 Hz) and the response time of the Id current is very fast, but with oscillation and overshoot (see [Figure 23](#)).

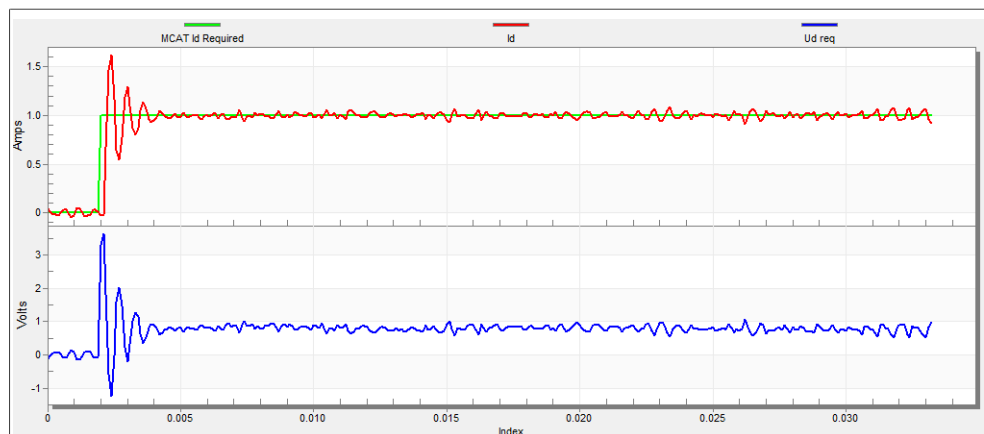


Figure 23. Fast step response of the Id current controller

7.13 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 24. 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.

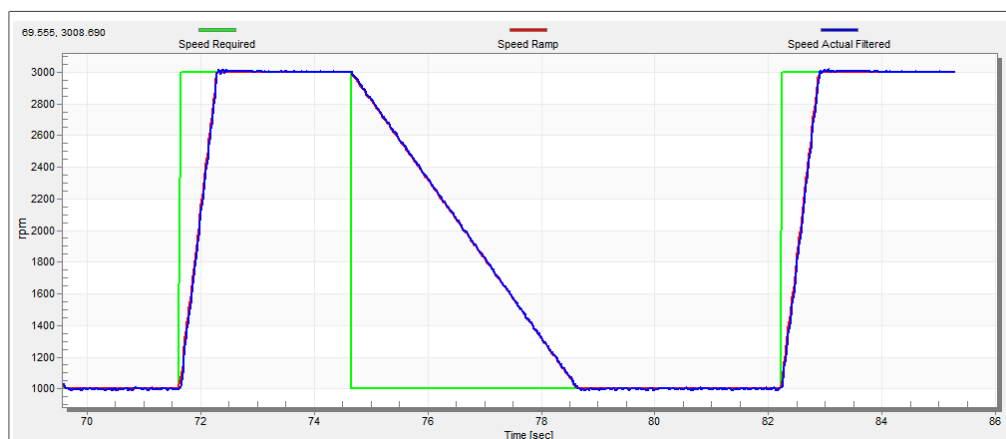


Figure 24. Speed profile

7.14 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 25](#).

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 %.

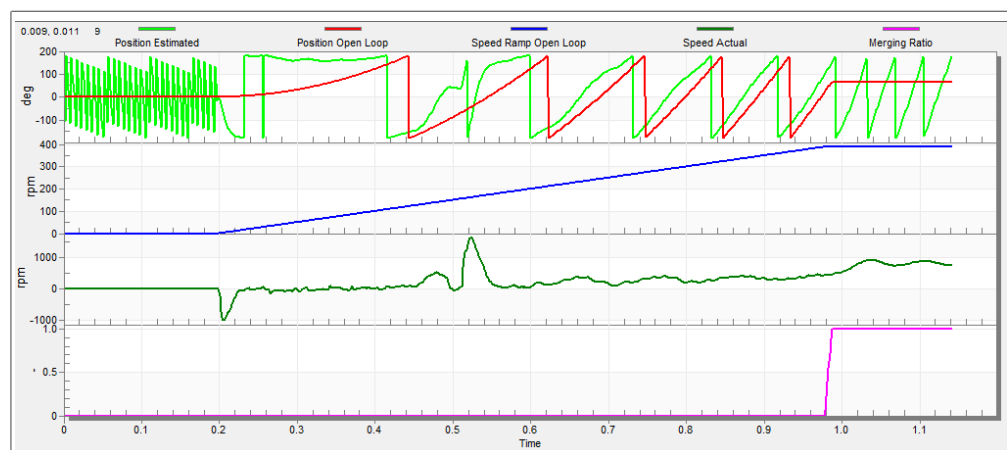


Figure 25. Motor startup

7.15 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.

7.16 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 26](#)). 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 26](#)).

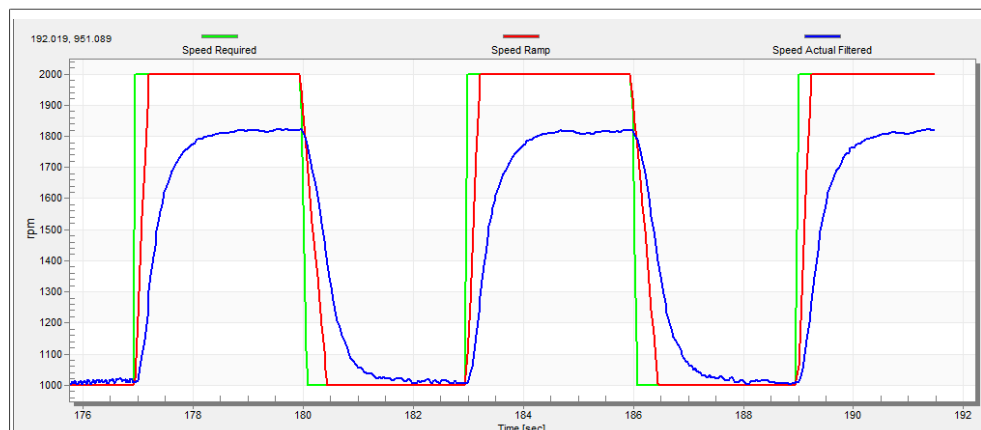


Figure 26. Speed controller response—SL_Ki value is low, Speed Ramp is not achieved

- The “SL_Kp” value is low, the “Speed Actual Filtered” greatly overshoots, and the long settling time is unwanted (see [Figure 27](#)).

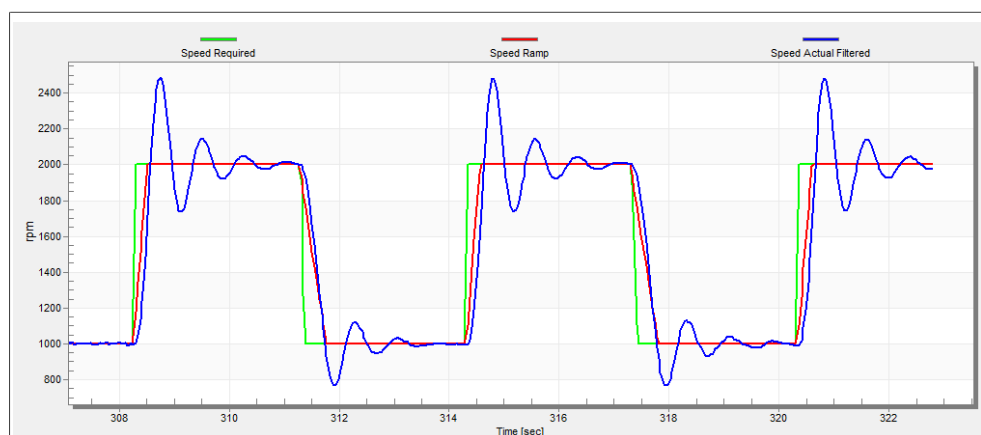


Figure 27. Speed controller response—SL_Kp value is low, Speed Actual Filtered greatly overshoots

- The speed loop response has a small overshoot and the “Speed Actual Filtered” settling time is sufficient. Such response can be considered optimal (see [Figure 28](#)).

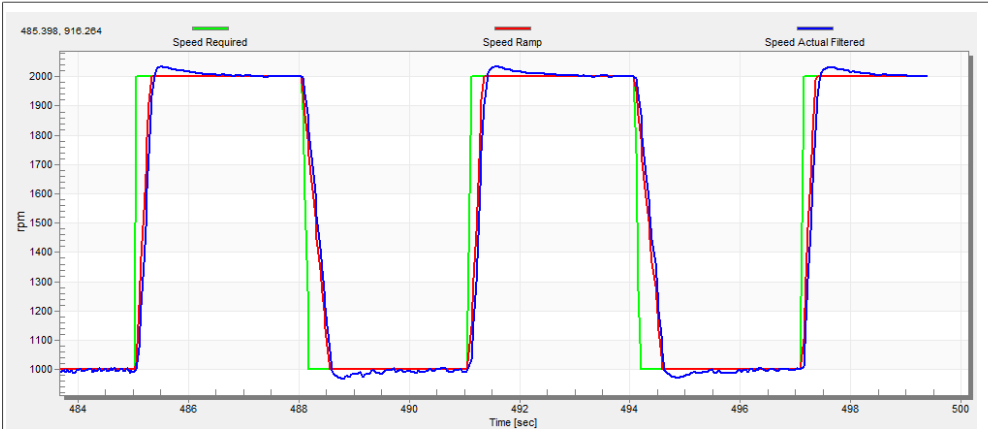


Figure 28. Speed controller response—speed loop response with a small overshoot

8 Conclusion

This application note describes the implementation of the sensor and sensorless Field-Oriented Control of a 3-phase PMSM on the NXP MC56F8xxx series device. The hardware-dependent part of the control software is described in [Section "Hardware setup"](#). The motor-control application timing and peripheral initialization were described in the chapter about MC56F8xxx series features. The motor user interface and remote control using FreeMASTER are as follows. The motor parameters identification theory and the identification algorithms are described in [Section "Identifying parameters of user motor"](#).

9 Acronyms and abbreviations

Table 11. 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

10 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](#))

11 Useful links

1. PMSM Control Reference Design www.nxp.com/motorcontrol_pmsm
2. BLDC Control Reference Design www.nxp.com/motorcontrol_bldc
3. ACIM Control Reference Design www.nxp.com/motorcontrol_acim
4. [FRDM-MC-PMSM Freedom Development Platform](#)
5. [Get Started with the MC56F81000-EVK](#)
6. [Get Started with the MC56F83000-EVK](#)
7. [CodeWarrior IDE](#)
8. MCUXpresso SDK Builder (SDK examples in several IDEs) <https://mcuxpresso.nxp.com/en/welcome>

12 Copyright and permission

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, Cold Fire+, 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, Design Start, DynamiQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μ Vision, 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.

13 Revision history

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

Table 12. Revision history

Revision number	Date	Substantive changes
0	01/2023	Initial release

Tables

Tab. 1.	Available examples and control methods	1	Tab. 7.	DIAG: Fault Captured variable	23
Tab. 2.	TWR-MC-LV3PH jumper setting	3	Tab. 8.	MCAT motor parameters	26
Tab. 3.	Linux 45ZWN24-40 motor parameters	4	Tab. 9.	Fault limits	26
Tab. 4.	TWR-56F8200 jumper settings	4	Tab. 10.	Application scales	26
Tab. 5.	MC56F82748 load and memory usage	9	Tab. 11.	Acronyms and abbreviations	36
Tab. 6.	MID: Fault variable	23	Tab. 12.	Revision history	40

Figures

Fig. 1.	Motor-control development platform block diagram	2	Fig. 16.	Speed FOC control mode	21
Fig. 2.	TWR-MC-LV3PH board	3	Fig. 17.	Position FOC control mode	21
Fig. 3.	Linux 45ZWN24-40 permanent magnet synchronous motor	4	Fig. 18.	MID FreeMASTER control	22
Fig. 4.	TWR-56F8200 with jumper settings	5	Fig. 19.	Phase currents	27
Fig. 5.	MC56F82xxx block diagram	7	Fig. 20.	Generated and estimated positions	28
Fig. 6.	Hardware timing and synchronization on MC56F82748	8	Fig. 21.	Slow step response of the Id current controller	29
Fig. 7.	Directory tree	10	Fig. 22.	Optimal step response of the Id current controller	29
Fig. 8.	Green “GO” button placed in top left-hand corner	15	Fig. 23.	Fast step response of the Id current controller	30
Fig. 9.	FreeMASTER—communication is established successfully	15	Fig. 24.	Speed profile	30
Fig. 10.	FreeMASTER communication setup window	16	Fig. 25.	Motor startup	31
Fig. 11.	Default symbol file	17	Fig. 26.	Speed controller response—SL_Ki value is low, Speed Ramp is not achieved	33
Fig. 12.	FreeMASTER + MCAT layout	18	Fig. 27.	Speed controller response—SL_Kp value is low, Speed Actual Filtered greatly overshoots	33
Fig. 13.	Scalar control mode	19	Fig. 28.	Speed controller response—speed loop response with a small overshoot	34
Fig. 14.	Voltage FOC control mode	20			
Fig. 15.	Current (torque) control mode	20			

Contents

1	Introduction	1
2	Hardware setup	2
2.1	TWR-MC-LV3PH	2
2.2	Linux 45ZWN24-40 motor	3
2.3	TWR-56F8200	4
2.3.1	Hardware assembling	5
3	MC56F8xxx series features and peripheral settings	7
3.1	MC56F82xxx	7
3.1.1	MC56F82748 hardware timing and synchronization	7
3.1.2	MC56F82748 peripheral settings	8
3.2	CPU load and memory usage	8
4	Project file and IDE workspace structure	10
4.1	PMSM project structure	10
5	Motor-control peripheral initialization	12
6	User interface	14
7	Remote control using FreeMASTER	15
7.1	Establishing FreeMASTER communication	15
7.2	TSA replacement with ELF file	16
7.3	MCAT FreeMASTER interface (Motor Control Application Tuning)	17
7.4	Motor Control Modes	19
7.4.1	Control structure	19
7.5	Switch between Spin and MID	21
7.6	Identifying parameters of user motor	22
7.6.1	Motor parameter identification using MID	22
7.7	MID algorithms	23
7.7.1	Stator resistance measurement	23
7.7.2	Stator inductances measurement	23
7.7.3	Number of pole-pair assistant	24
7.8	Electrical parameters measurement control	24
7.8.1	Mode 0	24
7.8.2	Mode 1	24
7.8.3	Mode 2	25
7.8.4	Mode 3	25
7.9	Initial configuration setting and update	25
7.10	Control structure modes	27
7.11	Alignment tuning	28
7.12	Current loop tuning	28
7.13	Speed ramp tuning	30
7.14	Open loop startup	30
7.15	BEMF observer tuning	31
7.16	Speed PI controller tuning	32
8	Conclusion	35
9	Acronyms and abbreviations	36
10	References	37
11	Useful links	38
12	Copyright and permission	39
13	Revision history	40