PMSMMKV11

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

Rev. 1 — 12 January 2023 User guide

1 Introduction

This user's guide describes the implementation of the sensorless motor-control software for a 3-phase Permanent Magnet Synchronous Motor (PMSM). The software is intended for PMSM with sinusoidal Back Electromotive Force (back-EMF) but is also very well usable for brushless motors (BLDC) with trapezoidal back-EMF.

The software also includes the motor parameters identification algorithm, on NXP 32-bit Kinetis V and Kinetis E series MCUs. The sensorless control software itself and the PMSM control theory, in general, are described in *DRM148: Sensorless PMSM Field-Oriented Control*.

The Freedom (FRDM-MC-LVPMSM) and High Voltage (HVP-MC3PH) power stages are used as hardware platforms 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 described as well.

Available motor control examples, supported motors and possible control methods are listed in Table 1.

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.

Table 1. Available examples and control methods

		Possible control methods in SDK example				
Example	**Supported motor**	Scalar & Voltage	Current FOC (Torque)	Sensorless Speed FOC	Sensored Speed FOC	Sensored Position FOC
pmsm_snsless	Linix 45ZWN24-40 (default motor)	✓	✓	✓	N/A	N/A
pmsm_snsless_reg_init	Linix 45ZWN24-40 - motor M2	✓	✓	✓	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 sensorless application runs on Tower, Freedom, or EVK development platforms with the 24-V Linix Motor in the default configuration. The HVP platform runs with the default configuration for the MIGE 60CST-MO1330 motor.

2.1 Linix 45ZWN24-40 motor

The Linix 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. 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 1. 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.

2.2 Running PMSM application on Freedom development platform

To run the PMSM application using the NXP Freedom development platform, you need these Freedom boards:

- Freedom board with a Kinetis V series MCU (FRDM-KV11Z or FRDM-KV31F).

- 3-phase low-voltage power Freedom shield (FRDM-MC-LVPMSM) with included Linix motor.

You can order all Freedom modules from www.nxp.com or from distributors to easily build the hardware platform for the target application.

2.2.1 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 2.

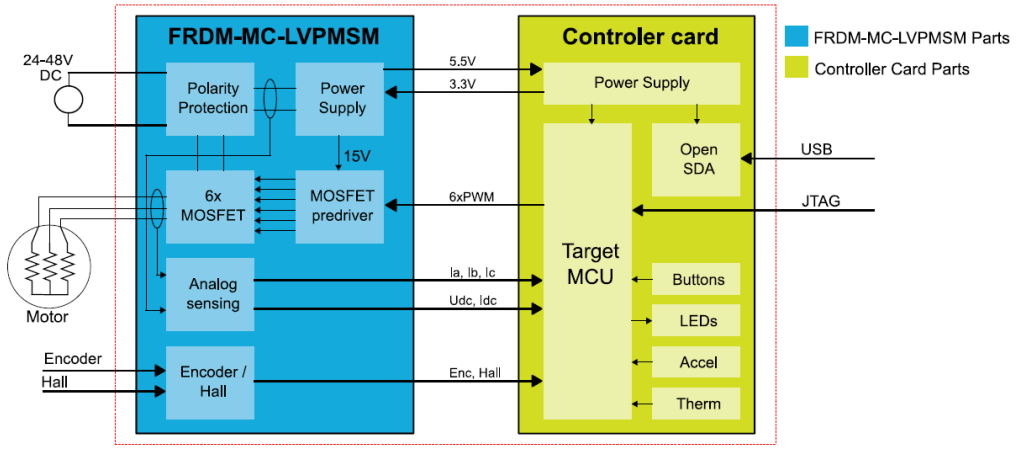

Figure 2. Motor-control development platform block diagram

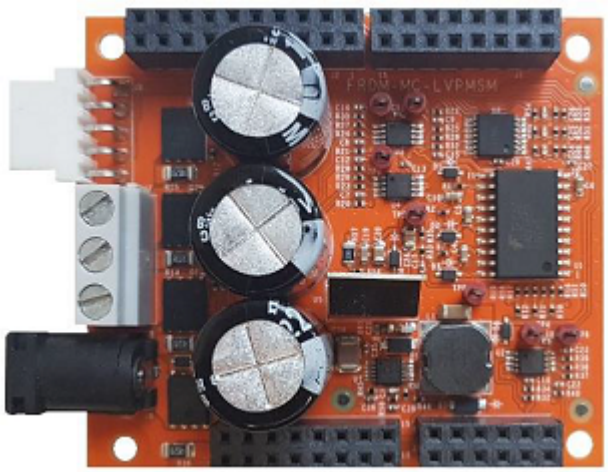

Figure 3. FRDM-MC-LVPMSM

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

2.2.2 FRDM-KV11Z board

The FRDM-KV11Z board is a low-cost development tool for the Kinetis V series KV1x MCU family built upon the Arm Cortex-M0+ processor. The FRDM-KV11Z board hardware is form-factor compatible with the Arduino R3 pin layout, providing a broad range of expansion board options. The FRDM-KV11Z platform features OpenSDA, the open-source hardware embedded serial and debug adapter running an open-source bootloader.

To begin, configure the jumpers on the FRDM-KV11Z Freedom System module properly. Table 3 lists the specific jumpers and their settings for the FRDM-KV11Z Freedom System module.

Table 3. FRDM-K11Z jumper settings

Jumper	Setting
J10	1-2

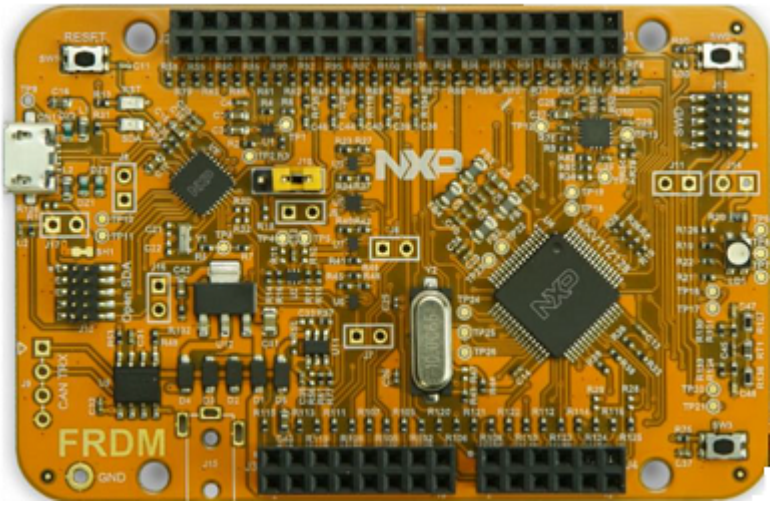

Figure 4. FRDM-KV11Z Freedom development board

2.2.3 Freedom system assembling

1. Connect the FRDM-MC-LVPMSM shield on top of the FRDM-Kxxxx board (there is only one possible option).
2. Connect the Linix motor 3-phase wires to the screw terminals on the board.
3. Plug the USB cable from the USB host to the OpenSDA micro USB connector.
4. Plug the 24-V DC power supply to the DC power connector.

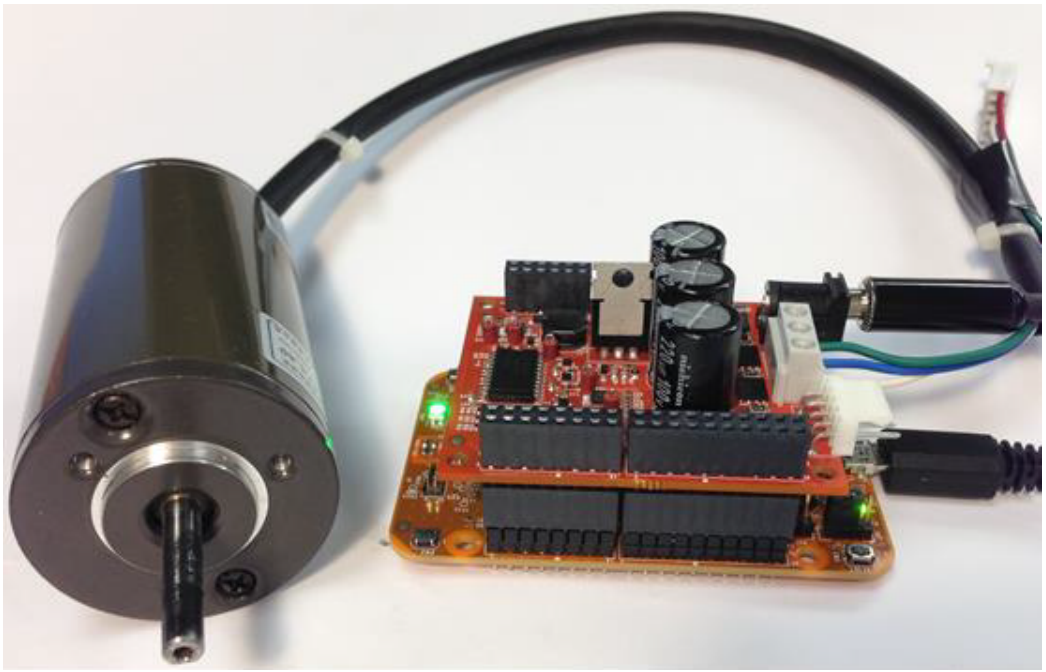

Figure 5. Assembled Freedom system

3 MCU features and peripheral settings

The peripherals used for motor control differ among different Kinetis MCUs. The peripheral settings and application timings for each MCU are described in the following sections.

3.1 KV1x family

The KV10Z and KV11Z MCU families are highly scalable members of the Kinetis V series and provide a cost-competitive motor-control solution. Built on the Arm Cortex-M0 core running at 75 MHz with up to 128 KB of flash and up to 16 KB of RAM, it delivers a platform enabling customers to build a scalable solution portfolio. Additional features include dual 16-bit ADCs sampling at up to 1.2 MS/s in a 12-bit mode and 20 channels of flexible motor-control timers (PWMs) across six independent time bases. For more information, see *KV11F Sub-Family Reference Manual* (document KV11P64M75RM).

3.1.1 Hardware timing and synchronization

Correct and precise timing is crucial for motor-control applications. Therefore, the motor-control-dedicated peripherals take care about the timing and synchronization on the hardware layer. It is also possible to set the PWM frequency as a multiple of the ADC interrupt (FOC calculation) frequency, in this case **FOCfreq = PWMfreq/2**. The timing diagram is in Figure 6.

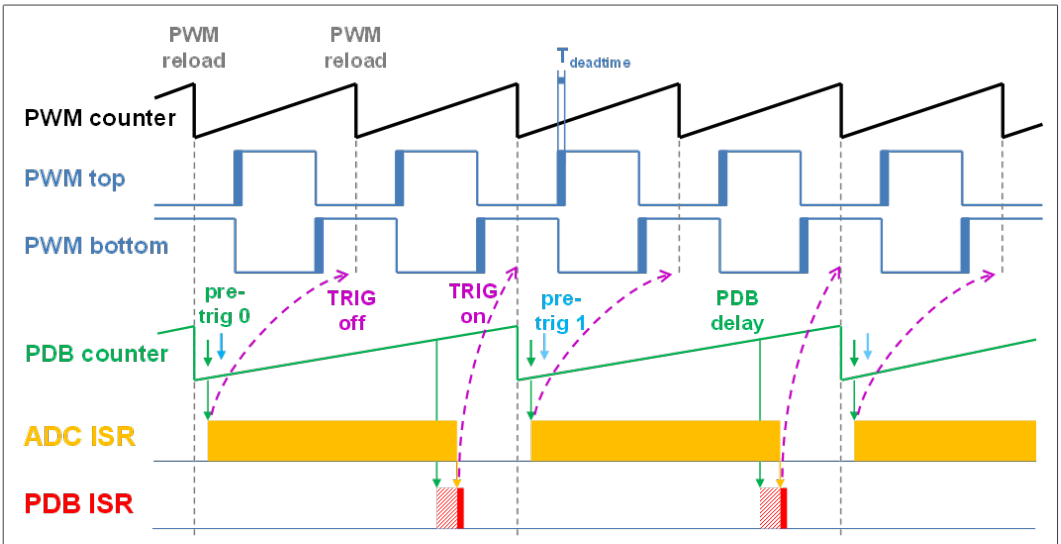

Figure 6. Hardware timing and synchronization on KV11Z

- The top signal (**PWM counter**) shows the FTM counter reloads. At the **PWM top** and **PWM bottom** signals, the dead time is emphasized. The **FTM_TRIG** is generated on the **PWM reload**, which triggers the PDB (resets the **PDB counter**).
- The PDB generates the first pre-trigger for the first ADC (phase current) sample with approximately **Tdeatime/2** delay. This delay ensures correct current sampling at duty cycles close to 100 %.
- When the conversion of the first ADC sample (phase current) is completed, the **ADC ISR** is entered. Firstly, the next **FTM_TRIG** is disabled (**TRIG off**). This ensures that the **PDB counter** does not reset at the next **PWM reload**. Then the FOC is calculated.

- In the middle of the next PWM period (**PDB delay**), the **PDB ISR** is called. This interrupt only enables the **FTM_TRIG** (**TRIG on**) in the next **PWM reload**. The **PDB ISR** has lower priority than the **ADC ISR**. The **PDB delay** length determines the ratio between the PWM and FOC frequencies.
- The PDB uses the back-to-back mode to automatically generate the **pre-trig 1** (for the DC-bus voltage measurement) immediately after the first conversion is completed.

3.1.2 Peripheral settings

This chapter describes only the peripherals used for motor control. On KV11Z, a 6-channel FlexTimer (FTM) is used for 6-channel PWM generation and two 16-bit SAR ADCs are used for the phase currents and DC-bus voltage measurement. The FTM and ADC are synchronized via the Programmable Delay Block (PDB). There is also one channel from another independent FTM used for the slow loop interrupt generation.

3.1.3 PWM generation - FTM0

- The FTM is clocked from the 75-MHz System clock.
- Only six channels are used, the other two are masked in the OUTMASK register.
- Channels 0+1, 2+3, and 4+5 are combined in pairs running in a complementary mode (each).
- The fault mode is enabled at each combined pair with automatic fault clearing (PWM outputs are re-enabled the first PWM reload after the fault input returns to zero).
- The PWM period (frequency) is determined as how long it takes the FTM to count from CNTIN to MOD. By default CNTIN = -MODULO/2 = -3750 and MOD = MODULO/2 -1 = 3749. The FTM is clocked from the System clock (75 MHz) so it takes 0.0001 s (10 KHz).
- Dead-time insertion is enabled at each combined pair. The dead-time length is calculated as System clock 75 MHz * Tdeadtime. The dead-time varies among platforms.
- The FTM generates a trigger to the PDB on counter initialization.
- The FTM fault input is enabled but its polarity and source varies among platforms.

3.1.4 Analog sensing – ADC0, ADC1

- The ADCs operate as 12-bit, single-ended converters.
- The clock source for both ADCs is the 25-MHz Bus clock divided by 2 = 12.5 MHz.
- For ADC calibration purposes, the ADC clock is 3.125 MHz and continues the conversion and averaging with 32 samples enabled in the SC3 register. After the calibration is done, the SC register is filled with its default values and the clock is set back to 12.5 MHz.
- Both ADCs are triggered from the PDB pre-triggers.
- There is an interrupt that serves the FOC fast-loop algorithm generated after the first conversion is completed.

3.1.5 PWM and ADC synchronization – PDB0

- Unlike the FTM, the PDB is clocked from the Bus clock which is three times slower than the System clock (used for FTM). Therefore, the modulo value at PDB is divided by three.
- The PDB is triggered from the FTM0_TRIG.

- The pre-trigger 0 at each channel is generated 0.5 * $T_{deadtime}$ after the FTM0_TRIG.
- The pre-trigger 1 at each channel is generated immediately after the first conversion is completed using the back-to-back mode.
- The PDB Sequence Error interrupt is enabled. This interrupt is generated if a certain result register is not read and the same pre-trigger occurs at this ADC.
- The PDB Delay interrupt is enabled. This interrupt is generated when the PDB_IDLY is reached. This interrupt enables the FTM_TRIG.
- The PDB Sequence Error and PDB Delay interrupts have a common interrupt vector. Which event generated the interrupt is determined at the beginning of the interrupt according to the ERR flag.

3.1.6 Over-current detection at FRDM platform – CMP1

- The plus input to the CMP is taken from the analog pin.
- The minus input to the CMP is taken from the 6-bit DAC0 reference. The DAC reference is set to 3.197 V (62/64*VDD) which corresponds to 7.73 A (for the 8.25 A scale).
- The CMP filter is enabled and four consecutive samples must agree.

3.1.7 Slow loop interrupt generation – FTM2

- The FTM2 is clocked from the System clock / 16, because the slow loop is usually ten times slower than the fast loop, so its modulo value can be kept reasonably low.
- The FTM counts from CNTIN = 0 to MOD = MODULO/16 * 10.
- An interrupt is enabled and generated at the counter reload and it serves the slow loop.

3.1.8 Communication with MC33937 MOSFET driver – SPI

- The SPI runs in the master mode.
- The SPI chip select 1 signal is active in logic high.
- The baud rate is 3.12 MHz.

3.1.9 Peripheral settings differences among platforms

There are some differences in peripheral settings among different platforms. Table 4 summarizes these differences.

Table 4. KV11 platform differences

Peripheral	Feature	Platform		
		FRDM	Tower	HVP
FTM0	PWM polarity	high sides active high low sides active high	high sides active low low sides active high	high sides active high low sides active high
	Fault source	FLT0, CMP1 out	FLT1, input pin	FLT0, input pin
	Fault polarity	Active high	Active high	Active low
	Dead-time	0.5 us	0.5 us	1.5 us
SPI	Driver on SPI	No	Yes	No
PDB	Pre-trigger 0 delay	0.25 us	0.25 us	0.75 us

3.1.10 CPU load and memory usage

The following information apply to the *demo* application built with IAR IDE. Table 5 shows the memory usage and the CPU load. The memory usage is calculated from the *.map* linker file, including the 2-KB FreeMASTER recorder buffer (allocated in RAM). 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 for the fast loop of 10 KHz and the slow loop of 1 KHz. The total CPU load is calculated according to 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. KV11 CPU load and memory usage (pmsm_snsless example debug configuration)

	MKV11Z
CPU load [%]	59.3
Flash usage [B]	73 660
RAM usage [B]	9 504

Table 6. KV11 CPU load and memory usage (pmsm_snsless_reg_init example debug configuration)

	MKV11Z
CPU load [%]	59.3
Flash usage [B]	44 692
RAM usage [B]	9 484

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

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 Figure 7.

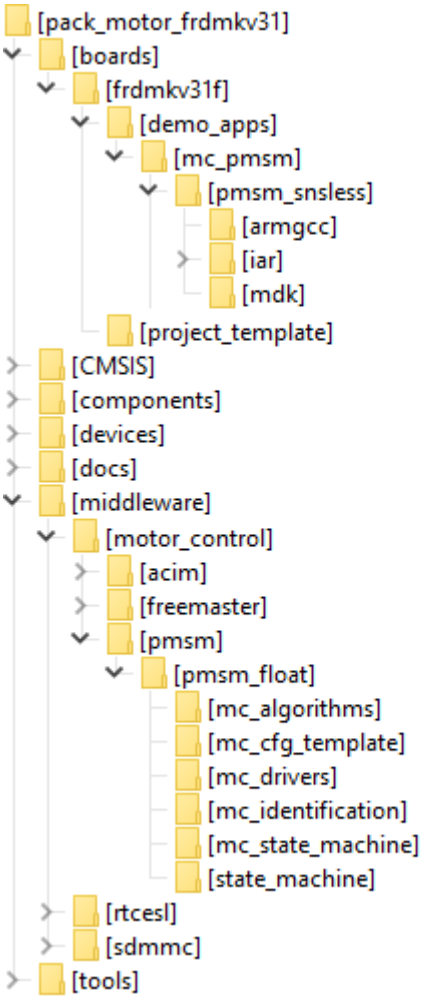

Figure 7. Directory tree

In the motor control project are located two examples. The main project folder in the first example is located in *pack_motor_xkxxx\boards\xkxxx\demo_apps\mc_pmsm\pmsm_snsless*\. The example has motor control peripherals set using MC_PMSM middleware

component in MCUXpresso Config Tool. The main project folder in the second example is located in *pack_motor_xkxxx\boards\xkxxx\demo_apps\mc_pmsm\pmsm_snsless_reg_init*\. The example has motor control peripherals set using Register Initialization component in MCUXpresso Config Tool. In both examples the main project folder 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.c 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_xkxxx\middleware\motor_control*\ contains these subfolders:

- *pmsm*—contains main pmsm motor-control functions.
- *freemaster*—contains the FreeMASTER project file *pmsm_float.pmp* (*pmsm_frac.pmp* for the fraction version of the MCU). 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_xkxxx\middleware\motor_control\pmsm\pmsm_float* folder contains these subfolders common to the other motor-control projects:

(*pack_motor_xkxxx\middleware\motor_control\pmsm\pmsm_frac* for the fraction version of the MCU)

- *mc_algorithms*—contains the main control algorithms used to control the FOC and speed control loop.
- *mc_cfg_template*—contains templates for MCUXpresso Config Tool components.
- *mc_drivers*—contains the source and header files used to initialize and run motor-control applications.
- *mc_identification*—contains the source code for the automated parameter-identification routines of the motor.

- *mc_state_machine*—contains the software routines that are executed when the application is in a particular state or state transition.
- *state_machine*—contains the state machine functions for the FAULT, INITIALIZATION, STOP, and RUN states.

5 Tools

Install the FreeMASTER Run-Time Debugging Tool 3.1.4 and one of the following IDEs on your PC to run and control the PMSM application properly:

- IAR Embedded Workbench IDE v9.32.1 or higher
- MCUXpresso v11.7.0
- ARM-MDK - Keil µVision version 5.37

For pin_mux.c, clock_config.c or peripherals.c modifications is recommended use MCUXpresso Configuration Tool v13 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_<booard>/ 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.

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 motor control project includes two types of examples. The first example **mc_pmsm** has motor control peripherals initialized using MC_PMSM middleware component in MCUXpresso Config Tool. The second example **mc_pmsm_reg_init** has motor control peripherals initialized using Register Initialization component in MCUXpresso Config Tool. Therefore, motor control peripheral initialization files *mc_periph_init.c and h* differ for both examples.

6.1 mc_pmsm example:

The *mc_periph_init.h* header file provides several macros that configure motor control low-level driver. It is recommended to modify this file using MCUXpresso Config Tools and MC_PMSM component. Manual modification is possible but some of the driver rules are checked only when file is generated using MCUXpresso Config Tools.

- *M1_PWM_TIMER*, *M1_PWM_TIMER_FTM0*—PWM generation timer instance.
- *M1_PWM_FREQ*—the value of this definition sets the PWM frequency.
- *M1_PWM_MODULO*—the value of PWM modulo must correspond with *M1_PWM_FREQ*.
- *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_FAST_LOOP_FREQ* —the value of this definition sets the speed loop frequency.
- *M1_SLOW_LOOP_FREQ* —the value of this definition sets the slow loop frequency.
- *M1_PWM_MODULO*—the value of slow loop modulo must correspond with *M1_SLOW_LOOP_FREQ*.
- *M1_FAST_LOOP_TS*—the value of fast loop period must correspond with *M1_FAST_LOOP_FREQ*.
- *M1_SLOW_LOOP_TS*—the value of slow loop period must correspond with *M1_SLOW_LOOP_FREQ*.
- *M1_PWM_PAIR_PH[A..C]*—these macros enable a simple assignment of the physical motor phases to the PWM periphery channels (or submodules). Change the order of the motor phases this way.
- *M1_PWM_DEADTIME_ENABLE*—enables PWM dead time insertion.
- *M1_PWM_DEADTIME_LENGTH_DTPS*—PWM dead time length (prescaler part).
- *M1_PWM_DEADTIME_LENGTH_DTVAL*—PWM dead time length (value part).
- *M1_FAULT_ENABLE*—enables PWM fault input.
- *M1_FAULT_NUM*—PWM fault input number.
- *M1_FAULT_POL*—PWM fault input polarity (0 = active high).
- *M1_FAULT_CMP_ENABLE*—PWM fault input taken from CMP output.
- *M1_FAULT_CMP_INSTANCE*—CMP instance used for fault detection.

- *M1_FAULT_CMP_THRESHOLD*—CMP instance used for fault detection.
- *M1_BRAKE_SET*, *M1_BRAKE_CLEAR*—macros that control the braking resistor GPIO.
- *M1_PWM_POL_TOP*, *M1_PWM_POL_BOTTOM*—inverter high-side and low-side polarity.
- *M1_SEC[1-6]_PH_[A..C]_[BASE, CHANNEL]*—these macros are used to assign the ADC channels for the phase current measurement. The general rule is that at least one of the phase currents 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. For more information about the 3-phase current measurement, see *Sensorless PMSM Field-Oriented Control* (document DRM148).
- *M1_[UDCB, AUX]_[BASE, CHANNEL]*—these macros are used to assign the ADC channels for the DC bus voltage and Auxiliary channel assignment. The general rule is that each quantity must be measured on different ADC instance.
- *ADC0_MUXSEL*, *ADC1_MUXSEL*—switches ADC muxed channels.
- *ADC_OFFSET_WINDOW*—ADC filter window during phase current offset calibration.
- *PDB_PRETRIG_DELAY*—PDB pre-trigger delay, should be set to half of the PWM dead time value.
- *M1_INRUSH_ENABLE*—enables inrush relay .
- *M1_INRUSH_DELAY*—inrush relay switch delay.
- *M1_INRUSH_SET()*,*M1_INRUSH_CLEAR()*—macros that control the inrush relay GPIO.

In the mc_pmsm example, these API-serving ADC and PWM peripherals are available:

- The available APIs for the ADC are:
  - *void M1_MCDRV_CURR_3PH_CHAN_ASSIGN(mcdrv_adc_t*)*—this function assigns ADC instances and channels to the phase-currents and prepares for next measurement.
  - *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_pwma_pwm3ph_t*—MCDRV PWM structure data type.
  - *void M1_MCDRV_PWM3PH_SET(mcdrv_pwma_pwm3ph_t*)*—this function updates the PWM phase duty cycles.
  - *void M1_MCDRV_PWM3PH_EN(mcdrv_pwma_pwm3ph_t*)*—calling this function enables all PWM channels.
  - *void M1_MCDRV_PWM3PH_DIS (mcdrv_pwma_pwm3ph_t*)*—calling this function disables all PWM channels.
  - *void M1_MCDRV_PWM3PH_FLT_GET(mcdrv_pwma_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.

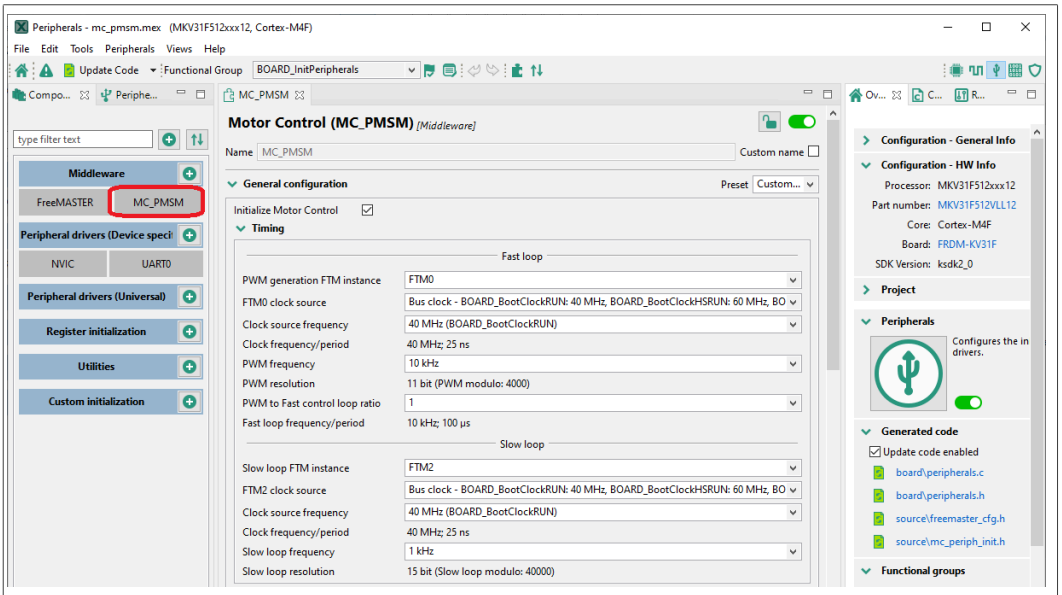

Figure 8. MCUXpresso Config Tool - MC_PMSM middleware component

6.2 mc_pmsm_reg_init example:

The *mc_periph_init.h* header file provides several macros that must be modified by user according to actual MC peripheral setting:

- *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 nth PWM reload. This is convenient when the PWM frequency must be higher than the maximal fast loop interrupt.
- *M1_FAST_LOOP_FREQ* —the value of this definition sets the speed loop frequency.
- *M1_SLOW_LOOP_FREQ* —the value of this definition sets the slow loop frequency.
- *M1_PWM_PAIR_PH[A..C]*—these macros enable a simple assignment of the physical motor phases to the PWM periphery channels (or submodules). Change the order of the motor phases this way.
- *M1_INRUSH_SET()*,*M1_INRUSH_CLEAR()*—macros that control the inrush relay GPIO.
- *M1_FAULT_NUM*—PWM fault input number.
- *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 of the phase currents 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.
- *ADC_OFFSET_WINDOW*—ADC filter window during phase current offset calibration.

In the mc_pmsm example, 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_CALIB_INIT(mcdrv_adc_t*)*—this function initializes the phase-current channel-offset measurement. This function always returns true.
  - *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. This function always returns true.
  - *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. This function always returns true.
  - *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. This function always returns true.
- The available APIs for the PWM are:
  - *mcdrv_pwma_pwm3ph_t*—MCDRV PWM structure data type.
  - *void M1_MCDRV_PWM3PH_SET(mcdrv_pwma_pwm3ph_t*)*—this function updates the PWM phase duty cycles. This function always returns true.
  - *void M1_MCDRV_PWM3PH_EN(mcdrv_pwma_pwm3ph_t*)*—calling this function enables all PWM channels. This function always returns true.
  - *void M1_MCDRV_PWM3PH_DIS (mcdrv_pwma_pwm3ph_t*)*—calling this function disables all PWM channels. This function always returns true.
  - *void M1_MCDRV_PWM3PH_FLT_GET(mcdrv_pwma_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.

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 development boards include a user button associated with a port interrupt (generated whenever one of the buttons is pressed). At the beginning of the ISR, a simple logic executes and the interrupt flag clears. When you press the button, the demo mode starts. When you press the same button again, the application stops and transitions back to the STOP state.

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

The application can be controlled the using these two interfaces:

- The button on the development board (controlling the demo mode):
  - FRDM-KV11Z - SW2
- 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.

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.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*

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.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.

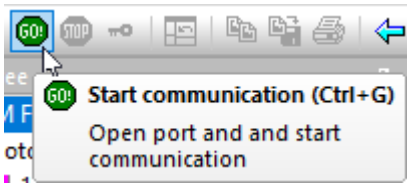

Figure 9. Green “GO” button placed in top left-hand corner

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.

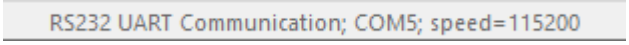

Figure 10. FreeMASTER—communication is established successfully

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.

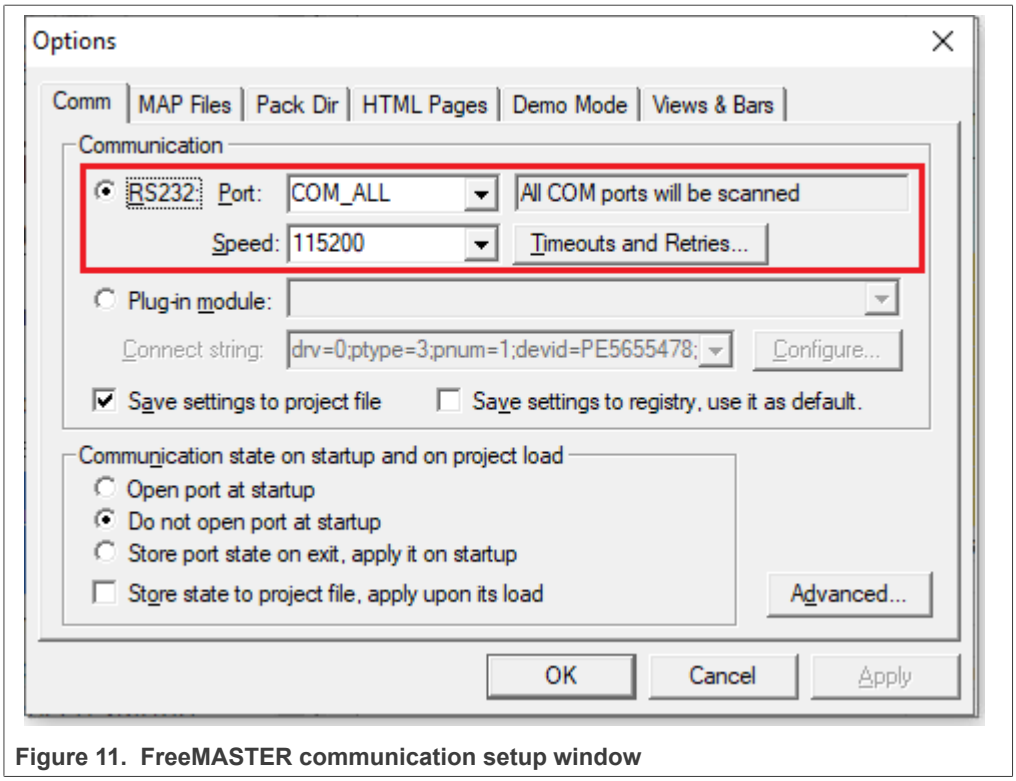

Figure 11. FreeMASTER communication setup window

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

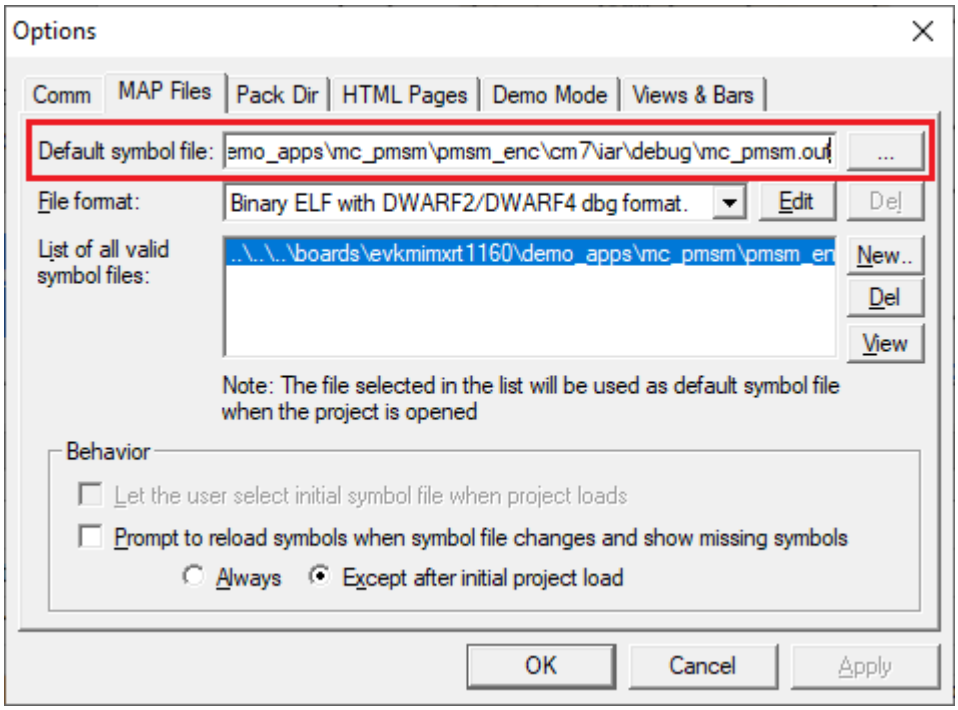

Figure 12. Default symbol file

- 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 13. 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_appcofig.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_appcofig.h file is selected. The motor ID and description is also saved in

mX_pmsm_appcofig.h in form of JSON comment. At single motor control application the embedded code #includes m**1**_pmsm_appcofig.h only. Therefore, saving to higher indexed m**X**_pmsm_appcofig.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_appcofig.h must be saved, code re-compiled and downloaded to MCU.

Note: *Path to mX_pmsm_appcofig.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_appcofig.h file only.*

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

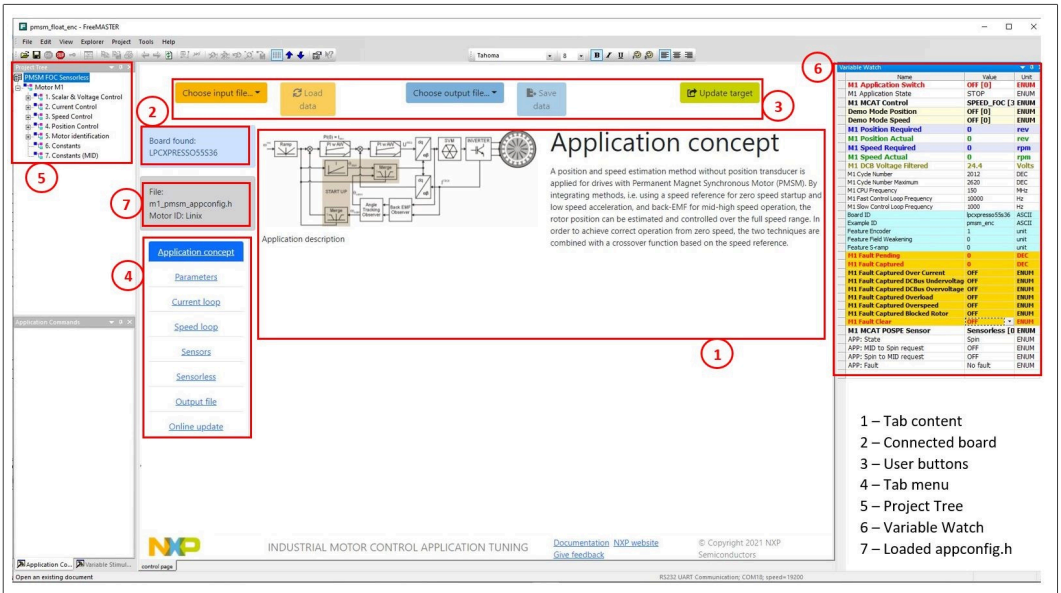

Figure 13. FreeMASTER + MCAT layout

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.

8.4 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.

8.4.1 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.

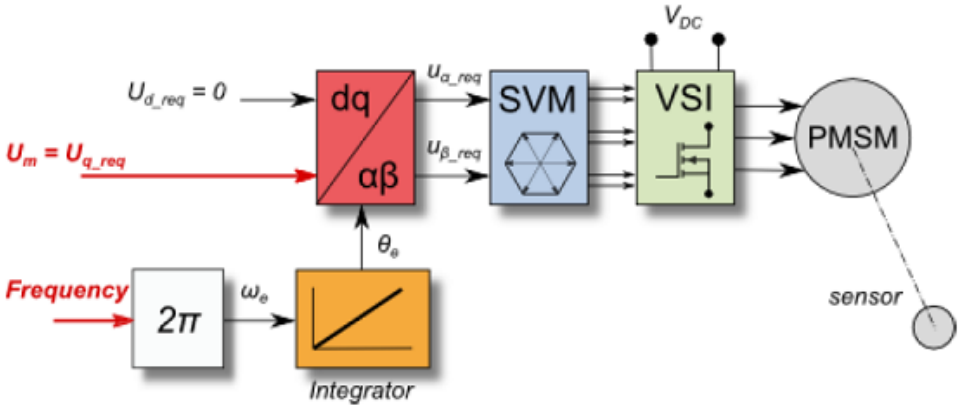

Figure 14. Scalar control mode

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.

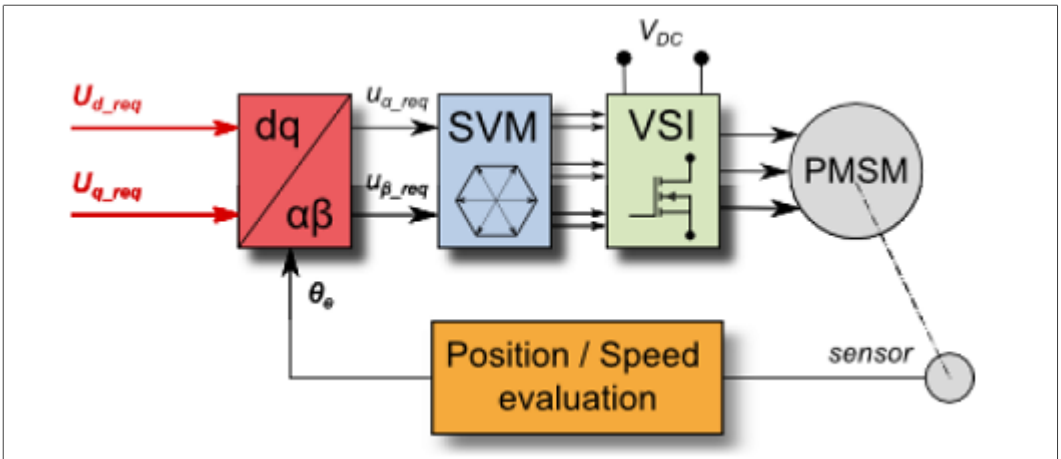

Figure 15. Voltage FOC control mode

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.

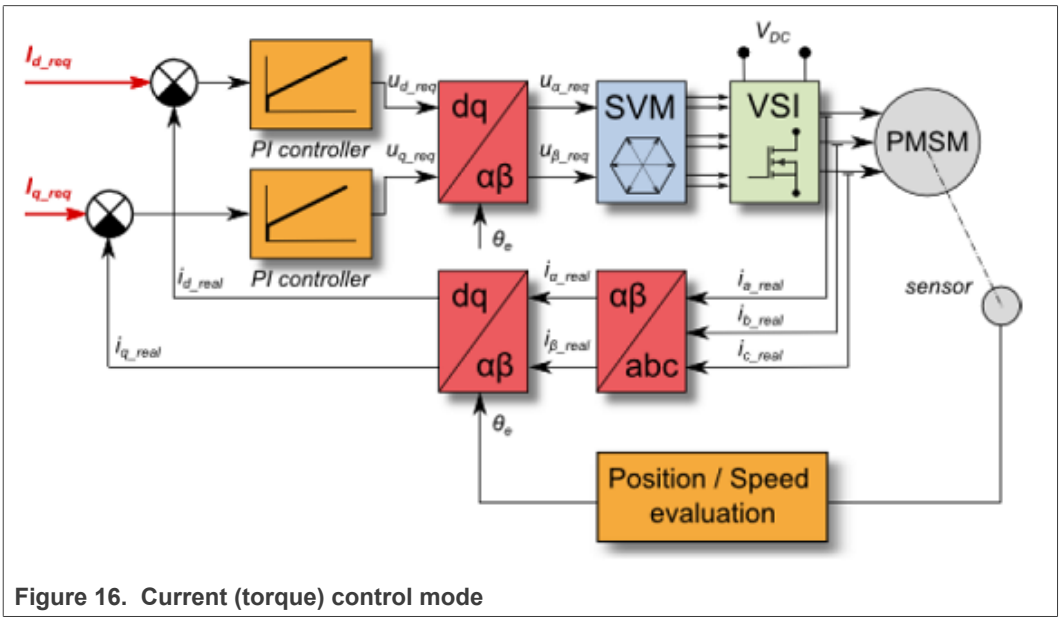

Figure 16. Current (torque) control mode

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.

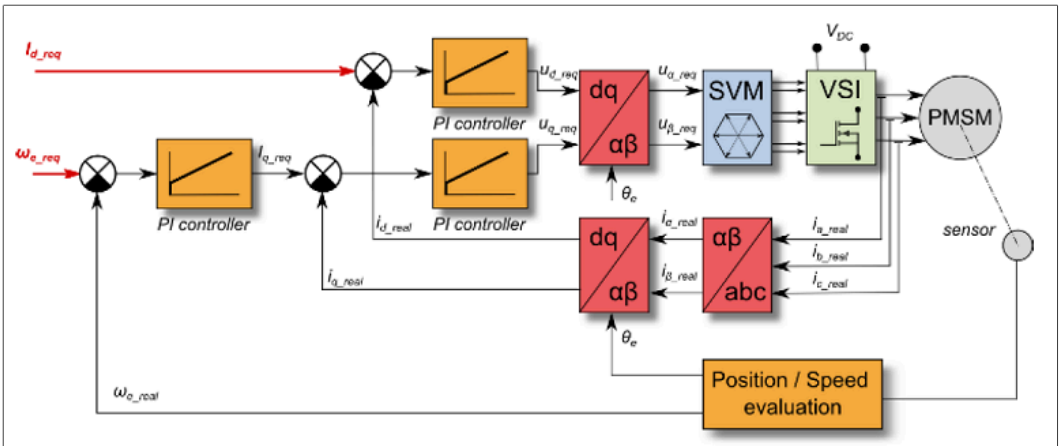

Figure 17. Speed FOC control mode

The position PMSM sensor FOC is shown in figure below (available for sensored/ 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.

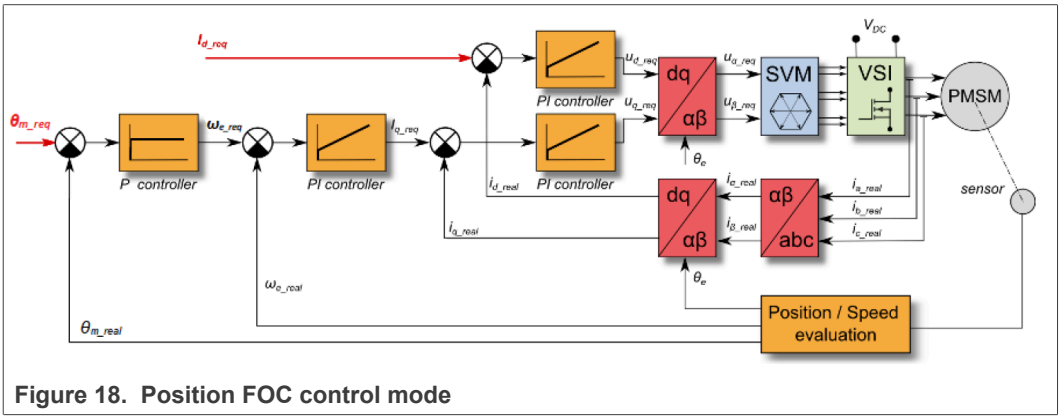

Figure 18. Position FOC control mode

8.5 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.

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 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 19.

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 19. MID FreeMASTER control

8.6.1 Motor parameter identification using MID

The whole MID is controlled via the FreeMASTER "Variable Watch". Motor Identification (MID) sub-block shown in Figure 19. 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 paramers 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 7. MID: Fault variable

Fault mask	Description	Troubleshooting
b#0001	Error during initialization electrical parameters measurement.	Check whether inputs to the *MCAA_EstimRLInit_F16* 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 8. DIAG: Fault Captured variable

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

8.7 MID algorithms

This section describes how each available MID algorithm works.

8.7.1 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.

8.7.2 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.

8.7.3 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 SpeedElReq*. If the rotor does not move after the start of PP_ASSIST assistant, stop the assistant, increase *MID Pp IdReqOpenLoop*, and restart the assistant.

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 El 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 definied be 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.

8.8.1 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 El 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.

8.8.2 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 definied 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 El I DC (estim Lq)* - Current to determine L_q. In most cases nominal current.
- *MID: Config El 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 El I DC (estim Lq)* and *MID: Config El 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.

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

8.8.4 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 "Ld" or "Lq" scopes. The following parameters can be changed during the runtime:

- *MID: Config El 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 El I DC req (d-axis)* - Required DC current in d-axis.
- *MID: Config El I DC req (q-axis)* - Required DC current in q-axis.
- *MID: Config El I AC req* - Required AC current injected to the d-axis or q-axis.
- *MID: Config El 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 9 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 9. 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

4. 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.
5. Check the fault limits—these fields are calculated using the motor parameters and hardware scales (see Table 10).

Table 10. 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

6. Check the application scales—these fields are calculated using the motor parameters and hardware scales.

Table 11. 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	-

7. 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.

8. Click the “Store data” button to save the modified parameters into the inner file.

8.10 Control structure modes

1. Select the scalar control in the "M1 MCAT Control" FreeMASTER variable watch.
2. Set the "M1 Application Switch" variable to "ON". The application state changes to “RUN”.
3. Set the required frequency value in the “M1 Scalar Freq Required” variable; for example, 15 Hz in the “Scalar & Voltage Control” FreeMASTER project tree. The motor starts running.
4. Select the “Phase Currents” recorder from the “Scalar & Voltage Control” FreeMASTER project tree.
5. The optimal ratio for the V/Hz profile can be found by changing the V/Hz factor directly using the “M1 V/Hz factor” variable. The shape of the motor currents should be close to a sinusoidal shape (Figure 20). 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!

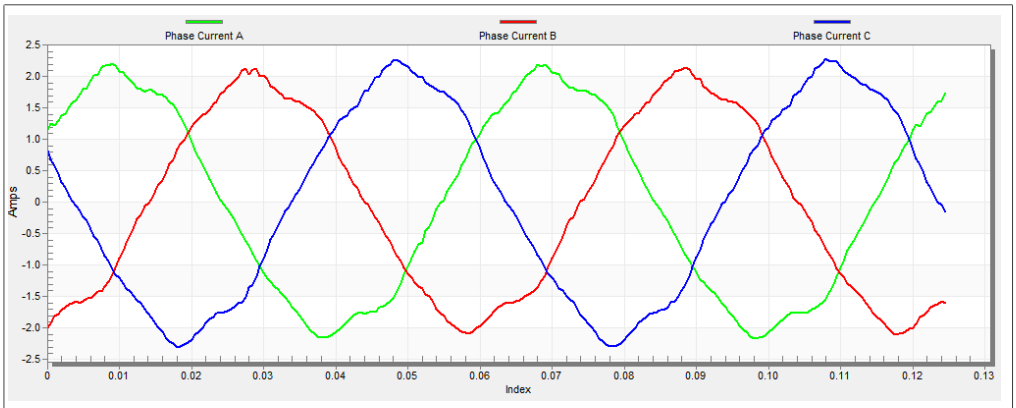

Figure 20. Phase currents

6. Select the “Position” recorder to check the observer functionality. The difference between the “Position Electrical Scalar” and the “Position Estimated” should be minimal (see Figure 21) 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.

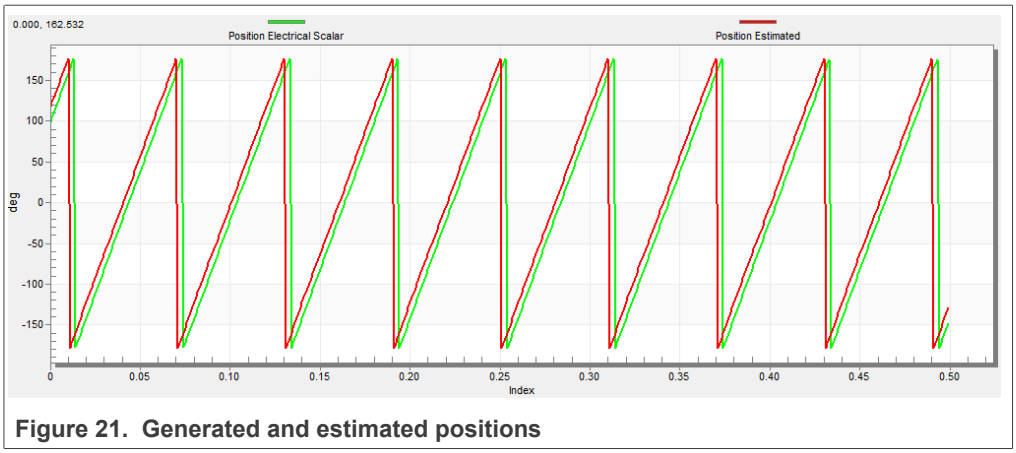

Figure 21. Generated and estimated positions

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 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.

8.12 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 22).

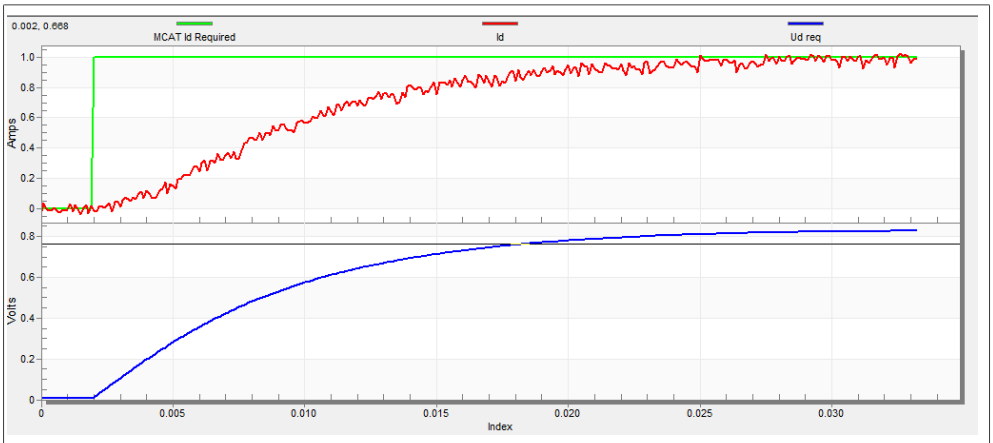

Figure 22. Slow step response of the Id current controller

   - The loop bandwidth (400 Hz) is optimal and the response time of the Id current is sufficient (see Figure 23).

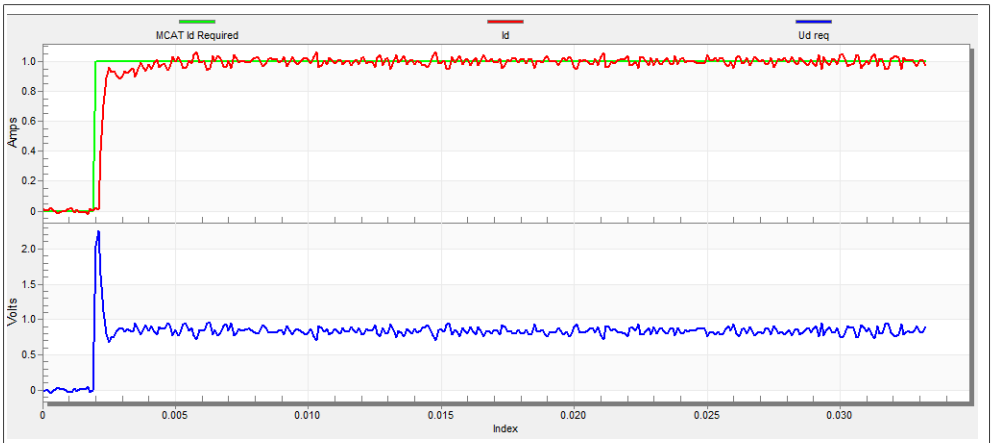

Figure 23. Optimal step response of the Id current controller

   - 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 24).

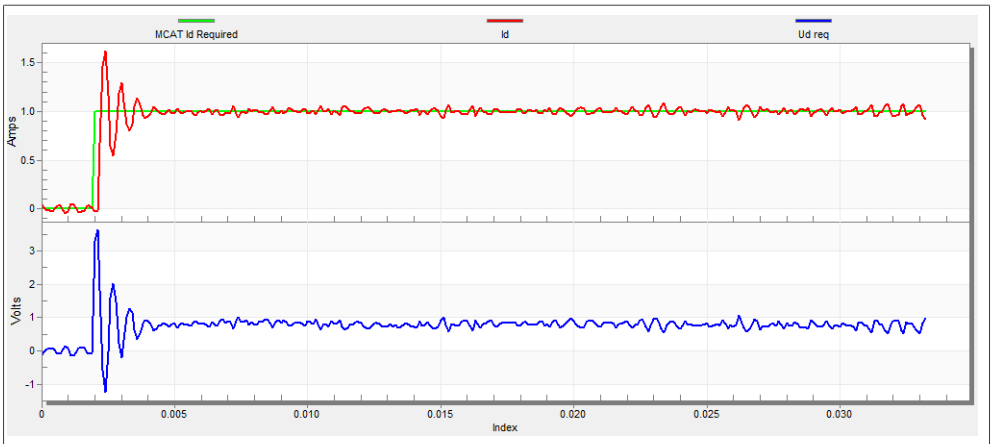

Figure 24. Fast step response of the Id current controller

8.13 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 25. 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.

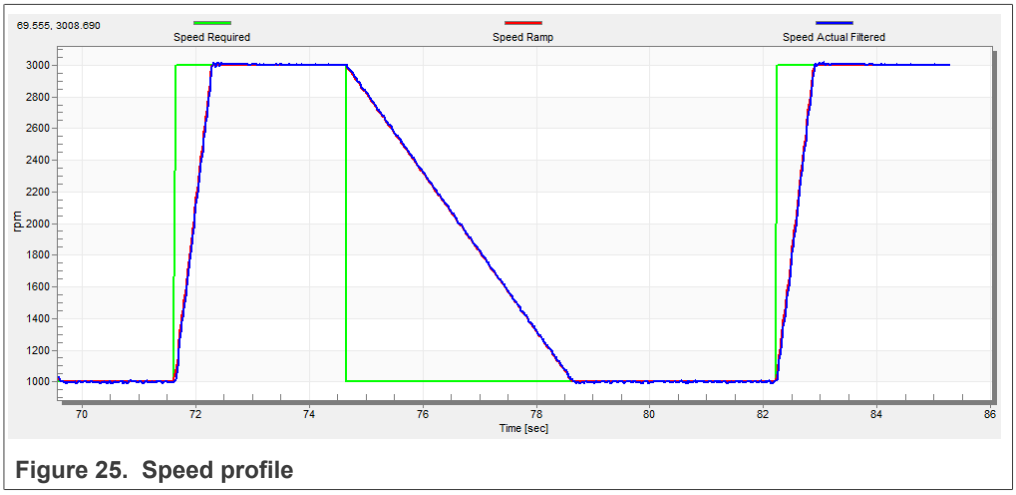

Figure 25. Speed profile

8.14 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 26.

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 %.

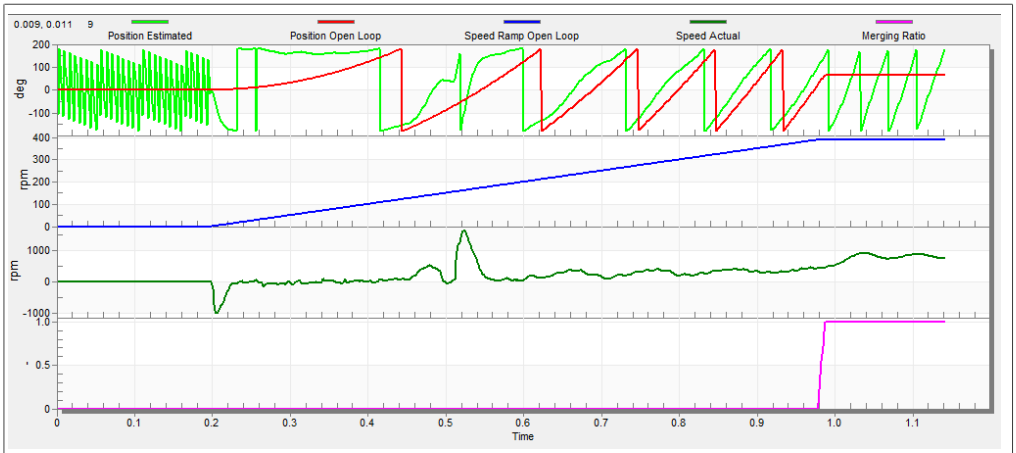

Figure 26. Motor startup

8.15 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.

8.16 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 27). 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 27).

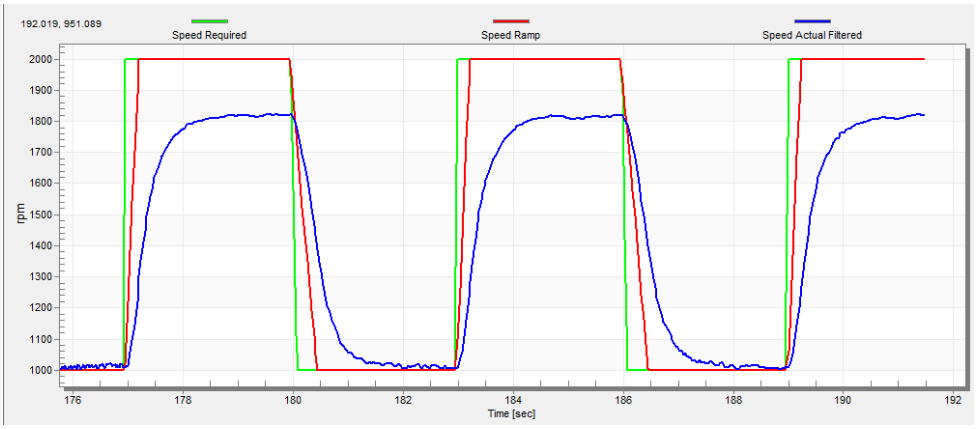

Figure 27. Speed controller response—SL_Ki value is low, Speed Ramp is not achieved

- The “SL_Kp” value is low, the “Speed Actual Filtered” greatly overshoots, and the long settling time is unwanted (see Figure 28).

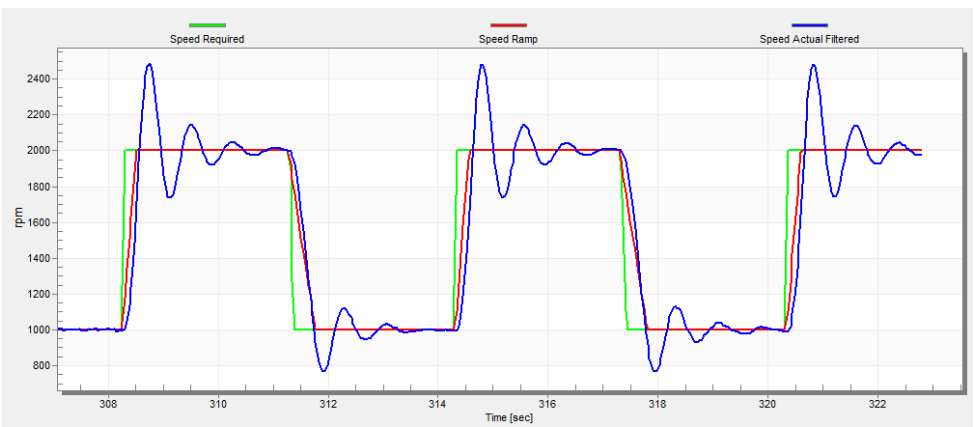

Figure 28. Speed controller response—SL_Kp value is low, Speed Actual Filtered greatly overshoots

- 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 29).

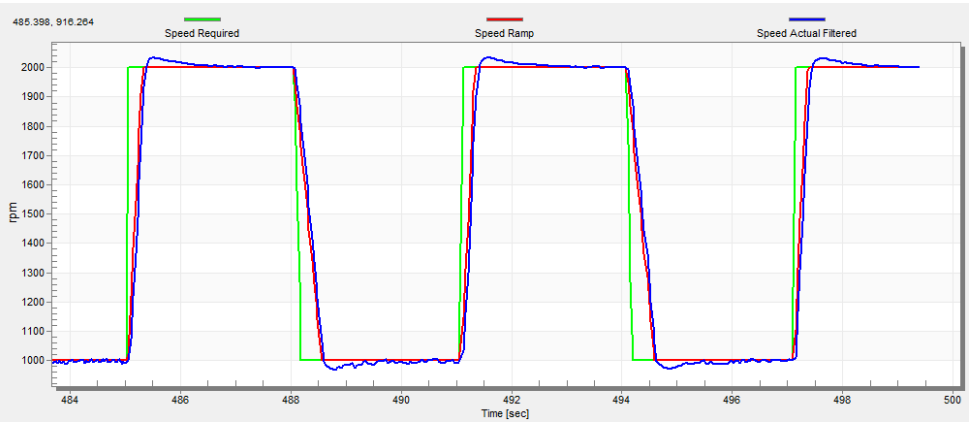

Figure 29. Speed controller response—speed loop response with a small overshoot

9 Conclusion

This application note describes the implementation of a sensorless field-oriented control of the 3-phase PMSM using 32-bit Kinetis V series devices and the Freedom development platform. The hardware-dependent part of the control software is described in Section "Hardware setup". The motor control application timing is described in Section "MCU features and peripheral settings" and the peripheral initialization in Section "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 Section "Identifying parameters of user motor".

10 Acronyms and abbreviations

Table 12. 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

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. *Sensorless PMSM Field-Oriented Control on Kinetis KV* (document AN5237).
4. *PMSM Sensorless Application Package User's Guide* (document PMSMSAPUG)

12 Useful links

1. MCUXpresso SDK for Motor Control www.nxp.com/motorcontrol
2. FRDM-MC-PMSM Freedome Development Platform
3. TWR-MC-LV3PH Tower Development Platform
4. HVP-MC3PH High-Voltage Platform
5. MCUXpresso IDE - Importing MCUXpresso SDK
6. MCUXpresso Config Tool
7. MCUXpresso SDK Builder (SDK examples in several IDEs) https://mcuxpresso.nxp.com/en/welcome

13 Revision history

Table 13 summarizes the changes done to this document since the initial release.

Table 13. Revision history

Revision number	Date	Substantive changes
0	01/2022	Initial release
1	01/2023	HVP-KV11Z75 not supported New MCAT (for fractional application) New MID (includes Pp assist and electrical parameters estimation)

14 Copyright and permission

How To Reach Us
Home Page:
nxp.com
Web Support:
nxp.com/support

Tables

Figures

Contents

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

Date of release: 12 January 2023
Document identifier: PMSMMKV11