**NXP Semiconductors**
User's Guide

Document identifier: MCUXSDK3PPMSMCWSUG
Rev. 0, 06/2020

# MCUXpresso SDK 3-Phase PMSM Control with IEC60730 Safety

# Contents

# Chapter 1
# Introduction

This user's guide describes the implementation of the three-phase high-voltage Permanent Magnet Synchronous Motor (PMSM) sensorless control reference application with IEC60730 class B safety V0.2.0 on NXP 32-bit Kinetis V series MCUs. The high voltage power stage (HVP-MC3PH) is used as a hardware platform for the PMSM control reference solution.

The first part of the document describes the hardware-dependent part of the sensorless control software, including a detailed peripheral setup. The second part describes the project file structure and safety tests. The last part describes the application control via FreeMASTER and the motor parameters identification algorithm.

# Chapter 2
# Supported development boards for Kinetis V

The High-Voltage Platform (HVP) is designed to drive high-voltage (115/230 V) applications with up to 1 kW of power. The supported development boards are shown in Table 1. For more details, see Hardware setup.

**Table 1. Supported development boards**

| Platform | Description | User's guide / product page |
|---|---|---|
| HVP-MC3PH | The high-voltage power stage | HVPMC3PHUG |
| HVP-KV11Z75M | The MKV11 development high-voltage daughter card | HVP-KV11Z75M |
| HVP-KV31F120M | The MKV31 development high-voltage daughter card | HVPKV31F120MUG |

# Chapter 3
# Motor control vs. SDK peripheral drivers

All MCU peripherals used by the pmsm_safe example are split into three groups:

- **General application peripherals** - This group covers all non-safety peripherals used by the application, namely the SysTick timer for CPU load measurement and the UART for FreeMASTER debug interface. Both these peripherals are initialized in the *app_periph_init.c* module. The MCUXpresso SDK peripheral drivers are generally used for peripherals in this group.
- **General safety-related peripherals** - This group includes generic safety-related peripherals (MCG, PORT, GPIO, CRC16, LPTMR), which are used or covered by the IEC60730 class B safety routines. These peripherals are initialized in the *safety_periph_init.c* module using proprietary drivers to ensure proper safety-related RO, RW, and CODE memory separation and checking. The general configuration of clocks, pins, and IRQs is easily done via the *hardware_cfg.h* header.
- **Motor-control specific peripherals** - The peripherals used by the safety-related motor-control application (FTM, ADC, and PDB). Motor control is a time-critical application because most control algorithms run in a 100-us loop. To optimize the CPU load and account for the fact that these peripherals are safety-related, proprietary initialization and driver routines are utilized. The initialization is done in the *mcdrv_periph_init.c* module and the drivers are split into the *mcdrv_adc_adc16.c*, *mcdrv_gpio.c*, and *mcdrv_pwm3ph_ftm.c* modules.

# Chapter 4
# Hardware setup

The PMSM sensorless application runs on the HVP platform and it is configured for the MIGE 60CST-MO1330 motor by default.

## 4.1 MIGE 60CST-MO1330 motor

The MIGE 60CST-MO1330 motor (described in Table 2) is used by the PMSM sensorless application. You can also adapt the application to other motors just by defining and changing the motor-related parameters. The motor is connected directly to the high-voltage development board via a flexible cable connected to the three-wire development board connector.

**Table 2. MIGE 60CST-MO1330 motor parameters**

| Characteristic | Symbol | Value | Units |
|---|---|---|---|
| Rated voltage | Vt | 220 | V |
| Rated speed | - | 3000 | rpm |
| Rated power | P | 400 | W |
| Number of pole-pairs | Pp | 4 | - |


**Figure 1. MIGE 60CST-MO1330 motor**

## 4.2 Running PMSM application on High-Voltage Platform (HVP)

To run the PMSM application on the NXP HVP, you need these components:

- HVP daughter card with a Kinetis V series MCU (HVP-KV11Z75M or HVP-KV31F120M).
- HVP power stage (HVP-MC3PH). The default motor is not included.

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

### 4.2.1 HVP-MC3PH

The NXP High-Voltage Platform (HVP) is an evaluation and development solution for Kinetis V and E series MCUs. This platform enables the development of three-phase PMSM, BLDC, and ACIM motor-control and power-factor-correction solutions in a safe high-voltage environment. The HVP is a 115/230-V, 1-kW power stage that is an integral part of the NXP embedded motion control series of development tools. It is supplied in the HVP-MC3PH kit in combination with an HVP daughter card and provides

a ready-made software development platform. The block diagram of a complete high-voltage motor-control development kit is in Figure 2.

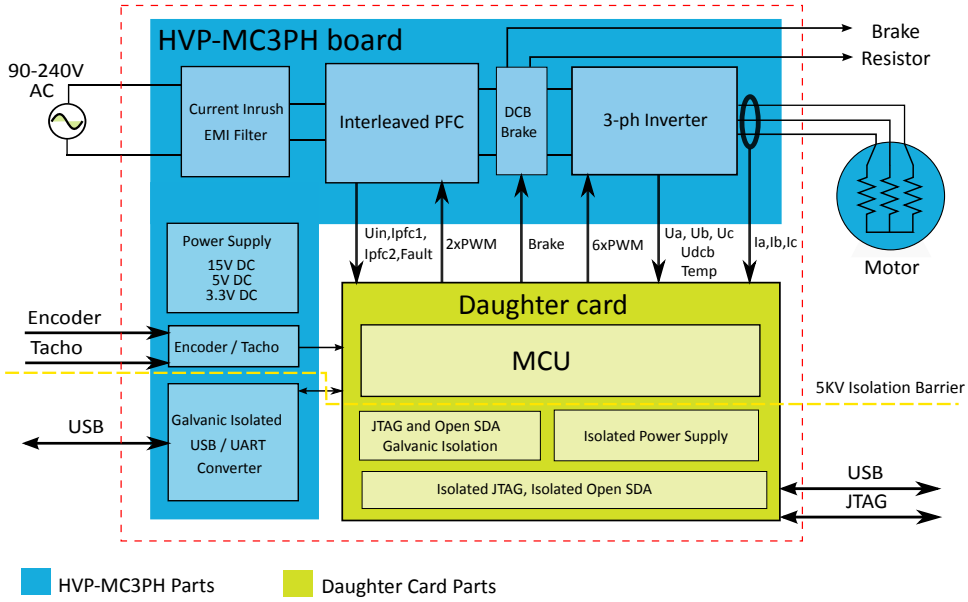


**Figure 2. High-voltage motor-control development platform block diagram**

The HVP-MC3PH power stage does not require a complicated setup and there is only one way to connect a daughter card to the HVP. The board works in the default configuration, and you don't have to set any jumpers to run the attached application. It is strongly recommended to read the complete *High-Voltage Motor Control Platform User's Guide* (document HVPMC3PHUG). Note that due to high-voltage, the HVP platform may represent safety risk when not handled correctly. For more information about the NXP high-voltage motor-control development platform, see nxp.com.

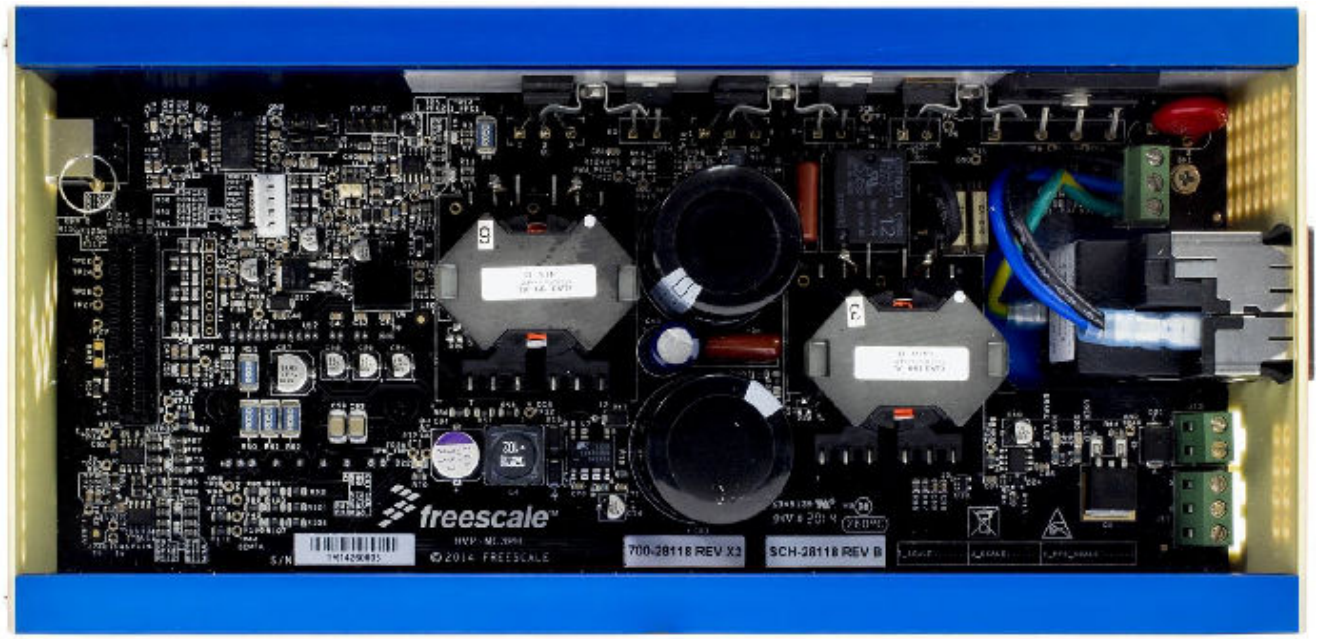


Figure 3. HVP-MC3PH high-voltage platform

### 4.2.2 HVP-KV11Z75M daughter card

The HVP-KV11Z75M MCU daughter card contains a Kinetis KV1x family MCU built around the Arm® Cortex®-M0+ core running at 75 MHz and containing up to 128 KB of flash memory. This daughter card is developed for use in motor-control applications, together with the High-Voltage Platform power stage. This daughter card features OpenSDA, the NXP open-source hardware embedded serial and debug adapter running an open-source bootloader.

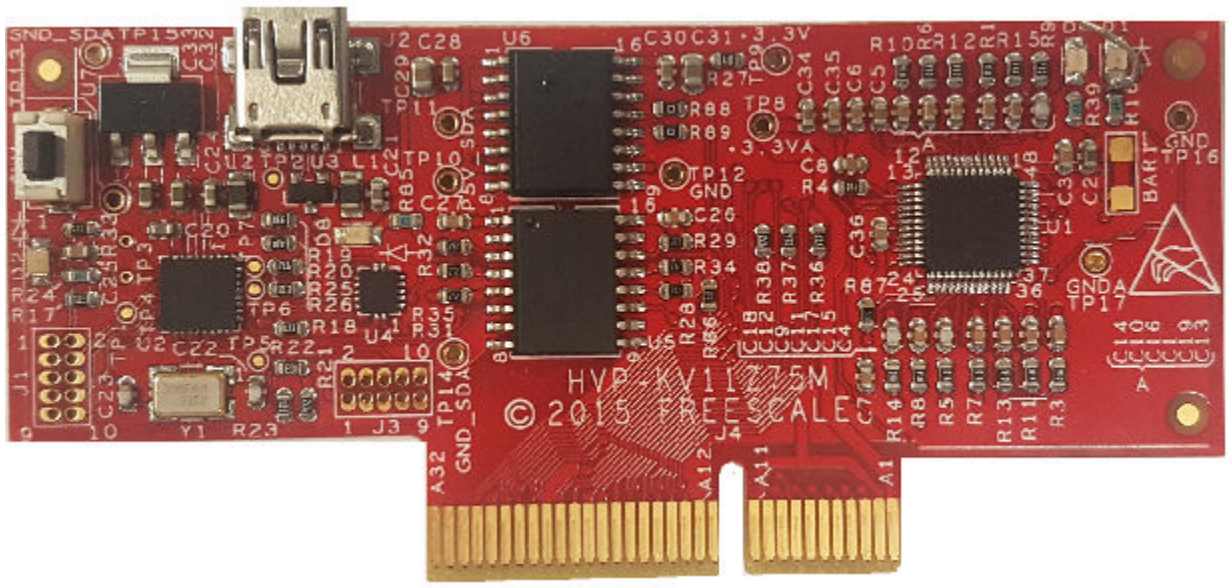


Figure 4. HVP-KV11Z75M daughter card

### 4.2.3 HVP-KV31F120M daughter card

The HVP-KV31F120M MCU daughter card contains a Kinetis KV3x family MCU built around the Arm Cortex-CM4F core with a floating-point unit, running at 120 MHz, and containing up to 512 KB of flash memory. This daughter card is developed for use in motor-control applications, together with the HVP power stage. This daughter card features OpenSDA, the NXP open-source hardware embedded serial and debug adapter running an open-source bootloader.

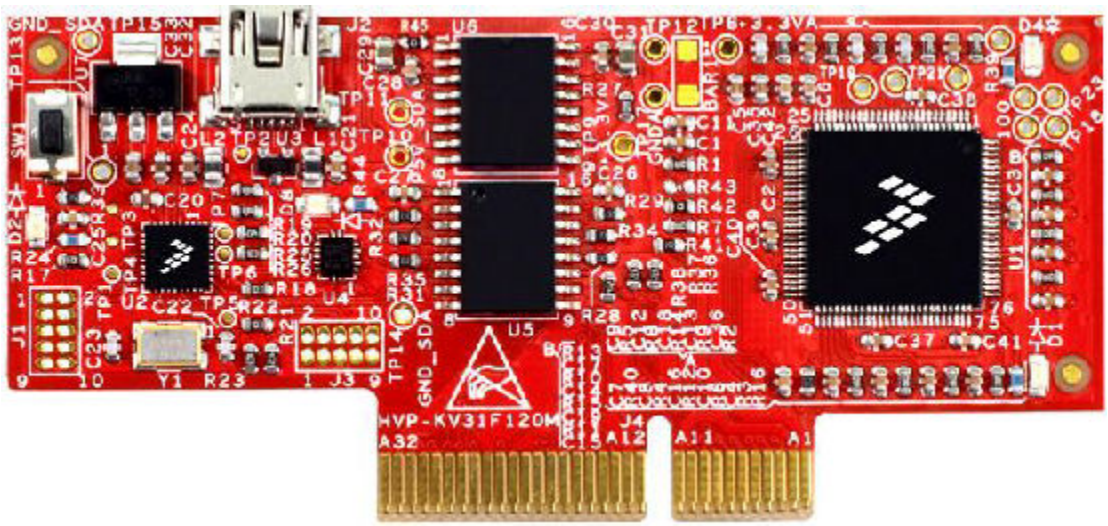


Figure 5. HVP-KV31F120M daughter card

### 4.2.4 HVP assembling

1. Check whether the HVP-MC3PH main board is unplugged from the voltage source.
2. Insert the HVP-KVxxx daughter board to the HVP-MC3PH main board (connector J11 is the only option).
3. Connect the PMSM motor three-phase wires into the screw terminals J13 on the board. The order of phases only affects the rotor spinning direction.
4. Plug the USB cable from the USB host to the OpenSDA micro USB connector on the daughter board.
5. Plug a 230-V power supply to the power connector and switch it on.

# Chapter 5
# MCU features and peripheral settings

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

## 5.1 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 a frequency of up to 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 the *KV11F Sub-Family Reference Manual* (document KV11P64M75RM).

### 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 at the hardware layer. It is also possible to set the PWM frequency as a multiple of the ADC interrupt (FOC fast control loop execution) frequency and thus lower the CPU load at the cost of longer sampling period. The **FOCfreq = PWMfreq** configuration is used by default. The low-priority synchronous tasks are executed in a slow loop interrupt generated via FTM. The timing diagram is shown in Figure 6.

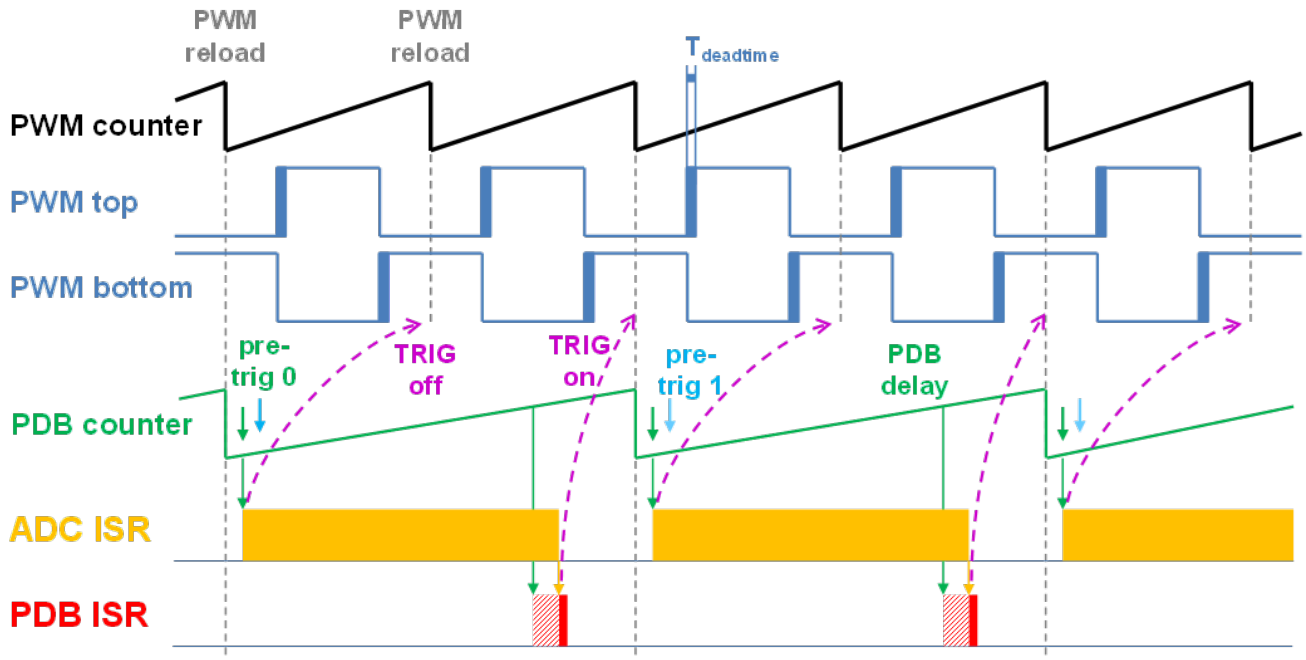


Figure 6. Hardware timing and synchronization on KV1xZ

- The top signal (**PWM counter** in Figure 6) shows the FTM counter value and its reloads (marked as **PWM reload** events). The FTM generates the **PWM top** and **PWM bottom** signals with a PWMfreq frequency. The $T_{deadtime}$ is inserted to avoid the DC-bus shoot-through. The PDB is triggered (its counter is reset) by the **FTM_TRIG** signal, which is generated together with the **PWM reload** event.
- The PDB generates the first pre-trigger to trigger acquisition of the first set of ADC samples (used for phase current measurement) with approximately $T_{deadtime}/2$ delay. This delay ensures correct current sampling at duty cycles close to 100 %.

- When the conversion of the first set of ADC samples (phase current measurement) is completed, the **ADC ISR** is envoked. 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 fast control loop 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 a lower priority than the **ADC ISR**. The **PDB delay** length determines the ratio between PWMfreq and FOCfreq.
- 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.

### Peripheral settings

This chapter describes only the peripherals used for motor control. On KV11Z, a six-channel FlexTimer (FTM) is used for six-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.

### PWM generation - FTM0

- The FTM is clocked from the 74.71-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 for each combined pair with automatic fault clearing (PWM outputs are re-enabled at the first PWM reload after the fault input returns to zero).
- The PWM period (frequency) is determined by how long it takes the FTM to count from CNTIN to MOD. By default, CNTIN = -MODULO/2 = -3735 and MOD = MODULO/2 -1 = 3734. The FTM is clocked from the System clock (74.71 MHz), so it takes 0.0001 s (10 KHz).
- Dead-time insertion is enabled for each combined pair. The dead-time counter modulo is calculated as the System clock (74.71 MHz) multiplied by $T_{deadtime}$. The 1.5-us dead time is inserted by default.
- The FTM generates a trigger to the PDB on counter reload.
- The FTM fault input zero is enabled, active low.

### Analog sensing – ADC0, ADC1

- Both ADCs operate as 12-bit, single-ended converters.
- The clock source for both ADCs is the 18.67-MHz Alternate clock (ALTCLK).
- For ADC calibration purposes, the ADC is running at 2.33 MHz in a continuous conversion mode and with 32 samples of hardware averaging. After the calibration is done, the SC register is filled with its default values and the clock is set back to 18.67 MHz.
- Both ADCs are triggered from the PDB pre-triggers.
- The fast control loop interrupt (**ADC ISR** in Figure 6) is triggered when the conversion is complete.

### 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 PDB modulo value 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 was 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 (**PDB ISR** in Figure 6) based on the ERR flag.

### Slow loop interrupt generation – FTM2

- The FTM2 is clocked from the System clock / 16, because the slow loop is by default 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) x 10.
- An interrupt is enabled and generated at the counter reload to trigger the slow control loop execution.

### CPU load and memory usage

The Table 3 shows the memory usage and the CPU load for an application built with the IAR IDE (see section Tools for the exact version). The memory usage is calculated from the **.map* linker file, including the 2-KB FreeMASTER recorder buffer (allocated in RAM). The CPU load is dependent on the fast-loop (FOC calculation) frequency (10 KHz in this case). The CPU load is calculated according to the following equation.

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

**Equation 1. CPU load**

Where:

CPU - the CPU load taken by the fast loop.

cycles - 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 (System clock frequency).

**Table 3. KV11 CPU load and memory usage**

| Configuration | Debug | Release |
|---|---|---|
| **Fast loop load [%]** | **61.5** | **51.2** |
| Flash (code + RO data) [B] | 29682 + 14312 | 23064 + 12980 |
| **Flash without FreeMASTER (code + RO data) [B]** | **18700 + 1064** | **16044 + 932** |
| RAM [B] | 3523 | 3508 |
| **RAM without FreeMASTER [B]** | **924** | **908** |

**NOTE**

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

## 5.2 KV3x family

The KV31F MCU family is a highly scalable member of the Kinetis V series and provides a high-performance, cost-competitive, motor-control solution. Built on the Arm Cortex-M4F core running at a frequency of up to 120 MHz, with up to 512 KB of flash and up to 96 KB of RAM, and combined with a floating-point unit, it delivers a platform enabling customers to build a scalable solution portfolio. The additional features include dual 16-bit ADCs sampling at up to 1.2 MS/s in a 12-bit mode, 20 channels of

flexible motor-control timers (PWMs) across four independent time bases, and a large RAM block enabling local execution of fast control loops at the full clock speed. For more information, see the *KV31F Sub-Family Reference Manual* (document KV31P100M120SF7RM).

### 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 at the hardware layer. It is also possible to set the PWM frequency as a multiple of the ADC interrupt (FOC fast control loop execution) frequency and thus lower the CPU load at the cost of a longer sampling period. The **FOCfreq = PWMfreq** configuration is used by default. The low-priority synchronous tasks are executed in a slow loop interrupt generated via FTM. The timing diagram is shown in Figure 7.

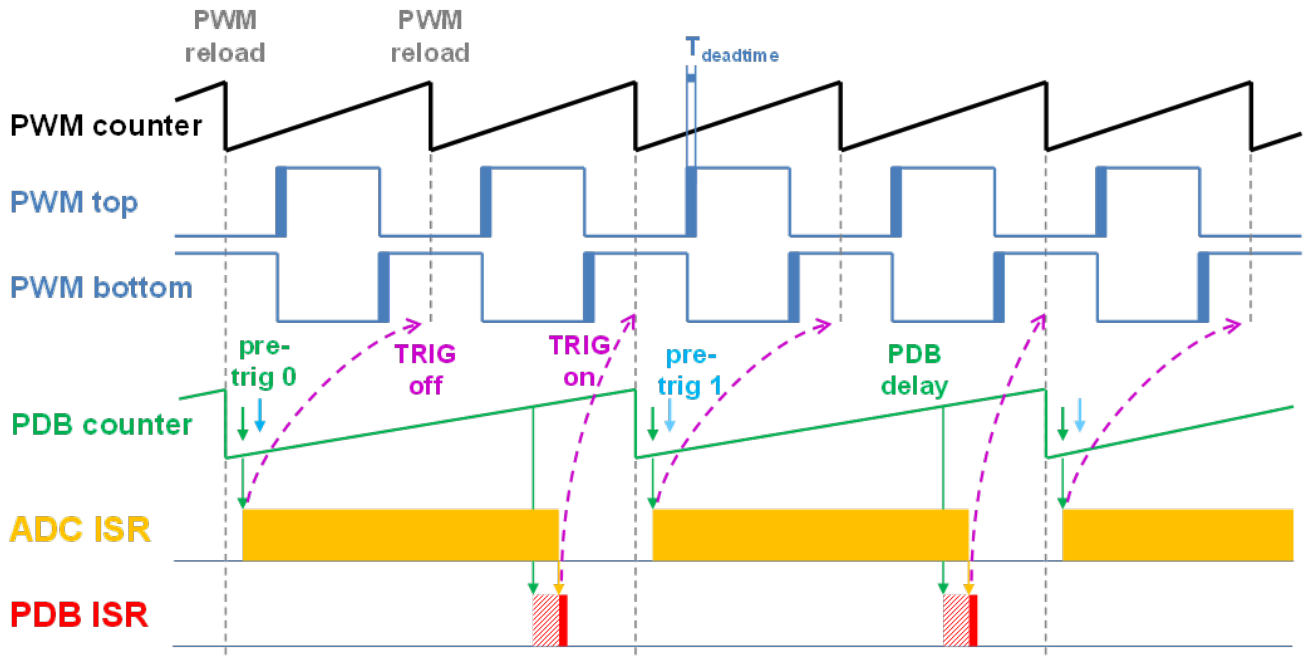


Figure 7. Hardware timing and synchronization on KV31F

- The top signal (**PWM counter** in Figure 7) shows the FTM counter value and its reloads (marked as **PWM reload** events). The FTM generates the **PWM top** and **PWM bottom** signals with the PWMfreq frequency. The $T_{deadtime}$ is inserted to avoid the DC-bus shoot-through. The PDB is triggered (its counter is reset) by the **FTM_TRIG** signal, which is generated together with the **PWM reload** event.
- The PDB generates the first pre-trigger to trigger the acquisition of the first set of ADC samples (used for the phase current measurement) with approximately **Tdeatime/2** delay. This delay ensures correct current sampling at duty cycles close to 100 %.
- When the conversion of the first set of ADC samples (phase current measurement) is completed, the **ADC ISR** is envoked. 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 fast control loop 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 a lower priority than the **ADC ISR**. The **PDB delay** length determines the ratio between the **PWMfreq** and **FOCfreq**.
- 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.

### Peripheral settings

This section describes only the peripherals used for motor control. KV31F contains a six-channel FlexTimer (FTM) used for six-channel PWM generation and two 16-bit SAR ADCs for the phase currents and DC-bus voltage measurement. The FTM and ADC are synchronized via the Programmable Delay Block (PDB). One channel from another independent FTM is also used for the slow-loop interrupt generation.

### PWM generation - FTM0

- The FTM is clocked from the 60-MHz Bus 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 the complementary mode.
- The fault mode is enabled for each combined pair with the 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 by how long it takes the FTM to count from CNTIN to MOD. By default, CNTIN = -MODULO/2 = -3000 and MOD = MODULO/2 -1 = 2999. The FTM is clocked from the System clock (60 MHz), so it takes 0.0001 s (10 KHz).
- The dead-time insertion is enabled for each combined pair. The dead-time counter modulo is calculated as the System clock (60 MHz) multiplied by $T_{deadtime}$. The 1.5-us dead-time is inserted by default.
- The FTM generates a trigger to the PDB on counter reload.
- The FTM fault input zero is enabled, active low.

### Analog sensing – ADC0, ADC1

- Both ADCs operate as 12-bit, single-ended converters.
- The clock source for both ADCs is the 60-MHz Bus clock clock divided by four, which equals to 15 MHz.
- For the ADC calibration purposes, the ADC is running at 3.75 MHz, in a continuous conversion mode and with 32 samples hardware averaging. After the calibration is done, the SC register is filled with its default values and the clock is set back to 15 MHz.
- Both ADCs are triggered from the PDB pre-triggers.
- The fast control loop interrupt (ADC ISR in Figure 7) is triggered when the conversion is complete.

### PWM and ADC synchronization – PDB0

- Like the FTM, the PDB is clocked from the 60-MHz Bus clock.
- 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 when a certain result register was 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 (**PDB ISR** in Figure 7), based on the ERR flag.

### Slow loop interrupt generation – FTM2

- The FTM2 is clocked from the System clock / 16, because the slow loop is by default ten times slower than the fast loop and the modulo value can be kept reasonably low.
- The FTM counts from CNTIN = 0 to MOD = (MODULO/16 ) x 10.
- An interrupt is enabled and generated at the reload to trigger the slow control loop execution.

### CPU load and memory usage

The Table 4 shows the memory usage and the CPU load for application built with the IAR IDE (see Tools for the exact version). The memory usage is calculated from the **.map* linker file, including the 2-KB FreeMASTER recorder buffer (allocated in RAM) and Motor Identification (MID). The CPU load is measured using the SysTick timer. The CPU load is dependent on the fast-loop (FOC calculation) frequency (10 KHz in this case). The total CPU calculation is described in CPU load and memory usage.

**Table 4. KV31 CPU load and memory usage**

| Configuration | Debug | Release |
|---|---|---|
| **Fast loop load [%]** | **33.8** | **32.5** |
| Flash (code + RO data) [B] | 39488 + 16724 | 29692 + 15360 |
| Flash without FreeMASTER (code + RO data) [B] | 28608 + 1840 | 22950 + 1710 |
| **Flash without FreeMASTER and MID (code + RO data) [B]** | **18464 + 1360** | **14936 + 1228** |
| RAM [B] | 4640 | 4625 |
| RAM without FreeMASTER [B] | 2041 | 2025 |
| **RAM without FreeMASTER and MID [B]** | **924** | **908** |

**NOTE**

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

# Chapter 6
# 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_pmsm_safe_hvpkv11* and *pack_pmsm_safe_hvpkv31* projects include all the available functions and routines, MID functions, IEC60730 class B compliant safety routines, scalar and vector control of the motor, FOC control, and FreeMASTER project. This project serves for development and testing purposes.

## 6.1 PMSM project structure

The directory tree of the PMSM project is shown in Figure 8.

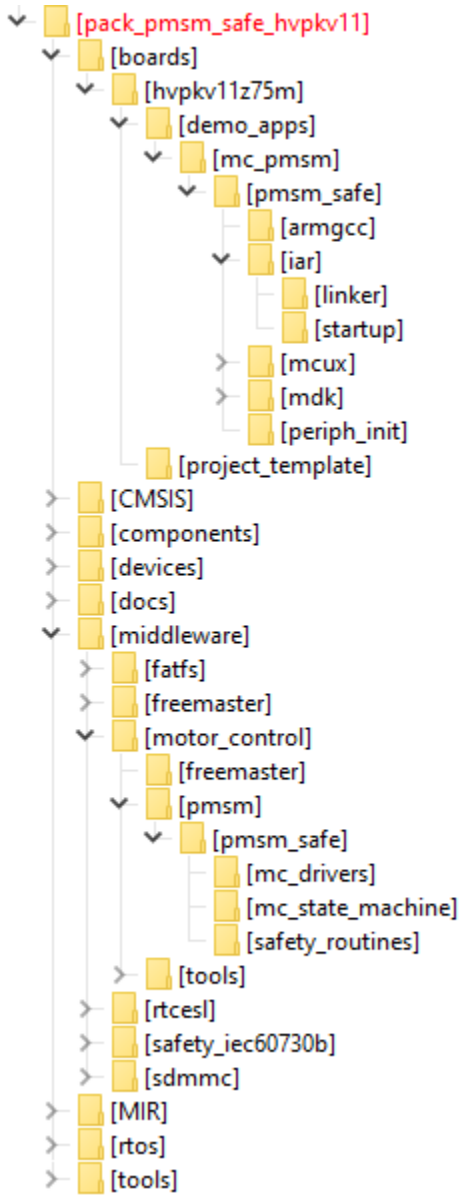


**Figure 8. Directory tree**

The main project folder *pack_pmsm_safe_hvpkvxx/boards/xkxxx/demo_apps/mc_pmsm/pmsm_safe/* contains these folders and files:

- *iar*—folder containing project files for the IAR Embedded Workbench IDE.
- *armgcc*—folder containing project files for the GNU Arm IDE.
- *mdk*—folder containing project files for the uVision Keil IDE.
- *mcux*—folder containing project files for the MCUXpresso IDE.
- *periph_init*—folder with peripheral initilization files.
- *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.
- *main.c* and *.h*—contains the basic application initialization, subroutines for accessing the MCU peripherals, and interrupt service routines. The FreeMASTER communication is processed in the background infinite loop.

- *m1_mcdrv.h*—this file provides the light abstraction layer of the *mcdrv_adc_adc16.c*, *mcdrv_gpio.c*, and *mcdrv_pwm3ph_ftm.c* drivers.
- *freemaster_cfg.h*—the FreeMASTER configuration file.
- *hardware_cfg.h*—the hardware configuration file containing hardware setup like ISRs, clocks, and pin-muxing.
- *safety_cfg.h*—the safety configuration file containing safety tests setup.

The *periph_init* folder contains these files:

- *app_periph_init.c* and *.h*—these files contain the initialization of non-safety and non-motor-control-related peripherals like UART and SysTick.
- *mcdrv_periph_init.c* and *.h*—these files contains the initialization of motor-control-related peripherals like ADC, PWMs, and so on. These peripherals are safety-related as well.
- *safety_periph_init.c* and *.h*—these files contain the initialization of safety-related peripherals, like LPTMR and others.

The main motor-control folder *pack_pmsm_safe_hvpkvxx/middleware/motor_control/* contains these folders:

- *freemaster*—folder contains also the FreeMASTER project file *pmsm_safe_mid.pmp* (*pmsm_safe.pmp* for version HVP-KV11Z75M without MID). Open this file in the FreeMASTER tool and use it to control the application (see section Remote control using FreeMASTER for more details).
- *pmsm/pmsm_safe*—folder containing main motor-control functions.
- *tools*—folder containing files needed to compute the flash CRC by an external tool.

The *pmsm/pmsm_safe* folder contains these subfolders and files common to the other motor-control projects:

- *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 (valid for HVP-KV31F120M).
- *mc_state_machine*—contains the software routines that are executed when the application is in a particular state or state transition and the algorithms used to control the FOC and speed control loop.
- *safety_routines*—contains the safety functions based on the IEC60730 class B safety library (for more information, see Safety IEC60730 class B tests).
- *freemaster_tsa_pmsm.c* and *.h*—TSA table used for reading and writing variables via FreeMASTER.

The *pack_pmsm_safe_hvpkvxx/middleware/safety_iec60730b/* folder contains the IEC 60730 Class B Safety Library V3.0 for Kinetis KV3x MCU and V4.0 for Kinetis KV1x MCU.

The *pack_pmsm_safe_hvpkvxx/middleware/rtcesl/* folder contains Real Time Control Embedded Software Motor Control and Power Conversion Libraries (RTCESL).

# Chapter 7
# Safety IEC60730 class B tests

The three-phase high-voltage PMSM sensorless control reference design application with IEC60730 class B safety contains the following safety tests:

- Clock test
- CPU register test
- Program counter test
- Stack test
- Watchdog test
- Interrupt handling and execution test
- Program flow test
- Variable memory (RAM) test
- Invariable memory (flash) test

If any safety test detects an unsafe condition, the *FS_fsErrorHandling()* function is called with the error code as a parameter (the only exception is the watchdog test, which results in the internal *FS_fsWatchdogTest_AR()* endless loop). For the complete list of error codes, see the *safety_error_handler.h* header. The *FS_fsErrorHandling()* function disables all PWM signals and interrupts and locks the MCU in an endless loop. Depending on the error code, the red LED (D1 on HVP-KV11Z75M and HVP-KV31F120M) on the daughter card blinks, where the number of short blinks signals the error code value.

More information about some of the tests can be found in *Safety Class B with PMSM Sensorless Drive* (document AN5321) and in the IEC60730B Safety Library Example user's guides.

**NOTE**

Only the IEC60730 class B safety libraries V3.0 and V4.0 are certified with respect to the IEC60730 standard. The overall reference design application is not certified and it is meant to serve as a base for the development of certified customer applications.

**NOTE**

Depending on the IDE, debugger, and activated set of safety tests, it might be necessary to completely reset the MCU after downloading the application into the MCU flash. Otherwise, some safety tests might be triggered.

## Clock test

The clock test procedure tests the oscilator frequency for the CPU core by comparing it to the independent reference LPTMR timer. The test can be disabled via the FS_CFG_ENABLE_TEST_CLOCK macro (see *safety_cfg.h*).

**NOTE**

It is recommended to disable the clock test during debugging, otherwise the clock test failure condition activation may occur.

## CPU register test

This procedure tests all CPU registers for the stuck-at condition. The only exception is the program counter register test, which is implemented as a stand-alone safety routine.

### Program counter test

This procedure tests the CPU program counter register for the stuck-at condition. The test is performed both after the reset and during run-time. The test can be disabled via the FS_CFG_ENABLE_TEST_PC macro (see *safety_cfg.h*).

**NOTE**

The program counter test cannot be interrupted.

### Stack test

This test routine is used to test the overflow and underflow conditions of the application stack. The testing of the stuck-at faults in the memory area occupied by the stack is covered by the variable memory test (see below). The overflow or underflow of the stack can occur if the stack is incorrectly controlled or by defining a "too-small" stack area for the given application. The test is performed both after the reset and during run-time.

### Watchdog test

The watchdog test tests the watchdog timer functionality. The test is executed only once after the reset. The test causes the WDOG reset and compares the preset time of the WDOG starvation with the independent timer reference. The test can be disabled via the FS_CFG_ENABLE_WATCHDOG macro (see *safety_cfg.h*).

**NOTE**

Some debuggers may have issues with the WDOG causing a reset. Therefore, it is recommended to disable the watchdog test while debugging the application.

### Interrupt handling and execution test

This simple test is based on the variables that are incremented in each periodic safety-related interrupt and, in the appropriate moment, their values are compared to the predefined values. This indicates whether the interrupts occur and whether the time ratio of their occurrence is correct or not. The test can be disabled via the FS_CFG_ENABLE_TEST_ISR macro (see *safety_cfg.h*).

### Program flow test

The purpose of the program flow check is to test whether the program goes through all important parts (nodes) of the software. The method used is called Control Flow Checking by Software Signatures (CFCSS). All signatures are available in the *safety_flow_check.h* header file. The test can be disabled via the FS_CFG_ENABLE_TEST_FLOW macro (see *safety_cfg.h*).

### Variable memory (RAM) test

The variable memory on the supported MCU is an on-chip RAM. Both stack and safety-related RW data memories are checked using the MarchC or MarchX tests after the reset and during runtime. The test copies a block of memory to the backup area defined by the linker and restores the tested memory when the test finishes.

**NOTE**

This test cannot be interupted.

### Invariable memory (flash) test

The invariable (flash) memory test provides a CRC check of a dedicated safety-related part of memory and restores the tested memory after the test finishes. The test is performed after the reset and during run-time by calculating the safety-related flash CRC and comparing it to the value calculated during the post-build operation. Both the flash test and the CRC post-build calculations are disabled by default. To enable the flash test, see the *safety_cfg.h* file and set the FS_CFG_ENABLE_TEST_FLASH macro value to 1. The activation of the post-build CRC calculation depends on the IDE:

1. In the **IAR IDE**, the CRC is calculated by the IDE directly (see "Options → Build Action"). Therefore, the flash test is fully integrated to the example project in the IAR IDE by default.
2. In the **Arm Keil IDE**, it is necessary to use a third-party tool (Srecord v1.64):

- The Srecord tool is part of the package by default. See the *middleware/motor_control/tools/srec* folder.
- In the **Arm Keil IDE**, go to the project options and select the "User" tab.
- Select the "Run #1 " checkbox and, depending on the selected configuration, copy the command within quotation marks below to "Option → User → AfterBuild".

  Debug: "*crc_hex.bat -debug\mc_pmsm_safe.hex -debug\mc_pmsm_safe_crc.hex -..\..\..\..\..\..\middleware\motor_control\tools\srec\srec_cat*"

  Release: "*crc_hex.bat -release\mc_pmsm_safe.hex -debug\mc_pmsm_safe_crc.hex -..\..\..\..\..\..\middleware\motor_control\tools\srec\srec_cat*"

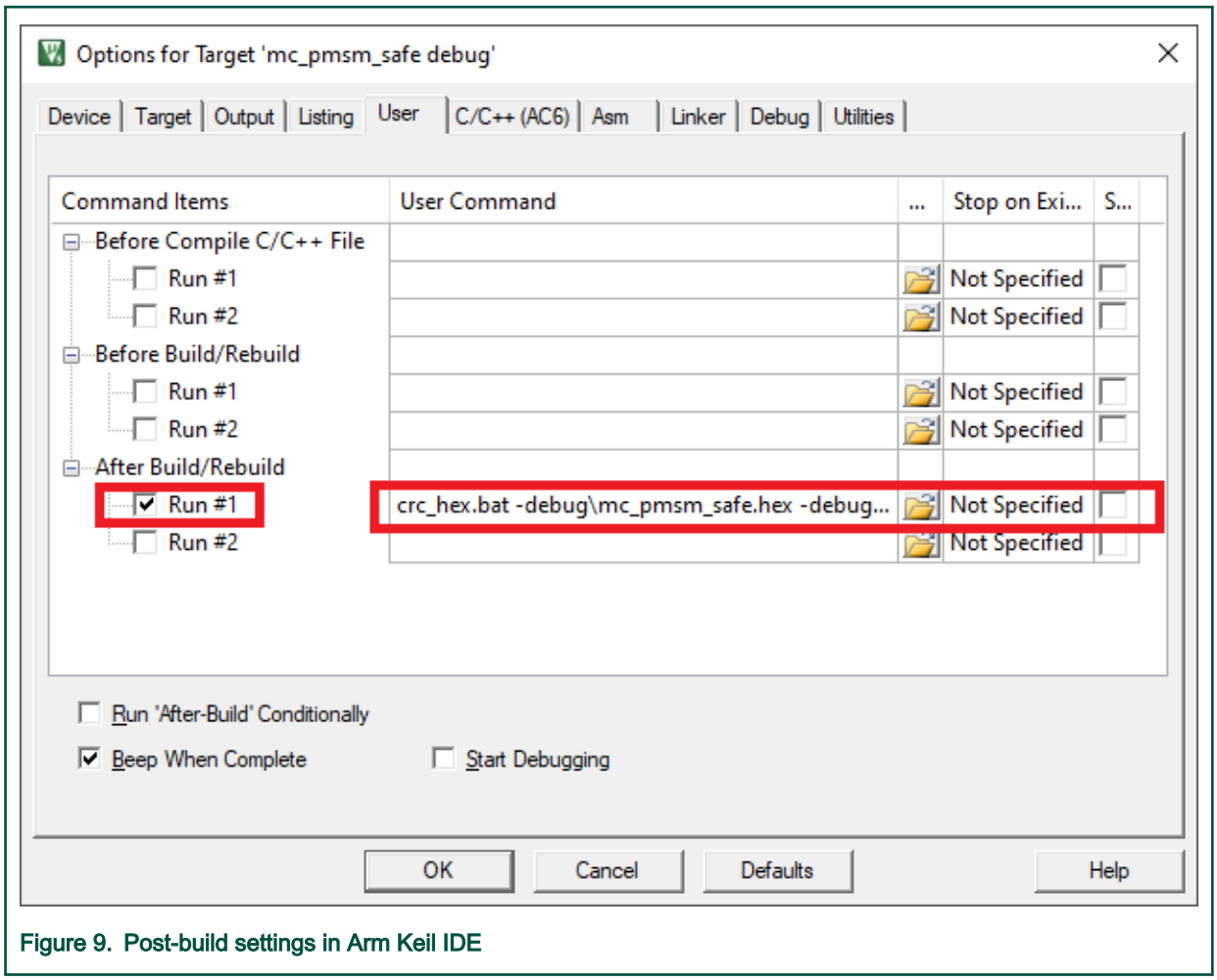


**Figure 9. Post-build settings in Arm Keil IDE**

- The final post-processed image can be downloaded to the ROM memory by clicking the "Download" button.
- For more information on using Srecord in the Arm Keil IDE, see *Calculating Post-Build CRC in Arm® Keil®* (document AN12520).

3. In the **MCUXpresso IDE**:
   - Go to "Properties → C/C++ Build → Settings → Build steps → Post-build steps" and fill the "Command" window with two lines within quotation marks below (avoid any additional linebreaks).

     The first line: "*arm-none-eabi-objcopy -v -O ihex "${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.hex"*"

     The second line: "*${ProjDirPath}/linker/crc_hex.bat -..\\\\${ConfigName}\\hvpkv31f120m_mc_pmsm_safe.hex -..\\\\${ConfigName}\\hvpkv31f120m_mc_pmsm_safe_crc.hex -..\\motor_control\\tools\\srec\\srec_cat ${BuildArtifactFileBaseName}*"

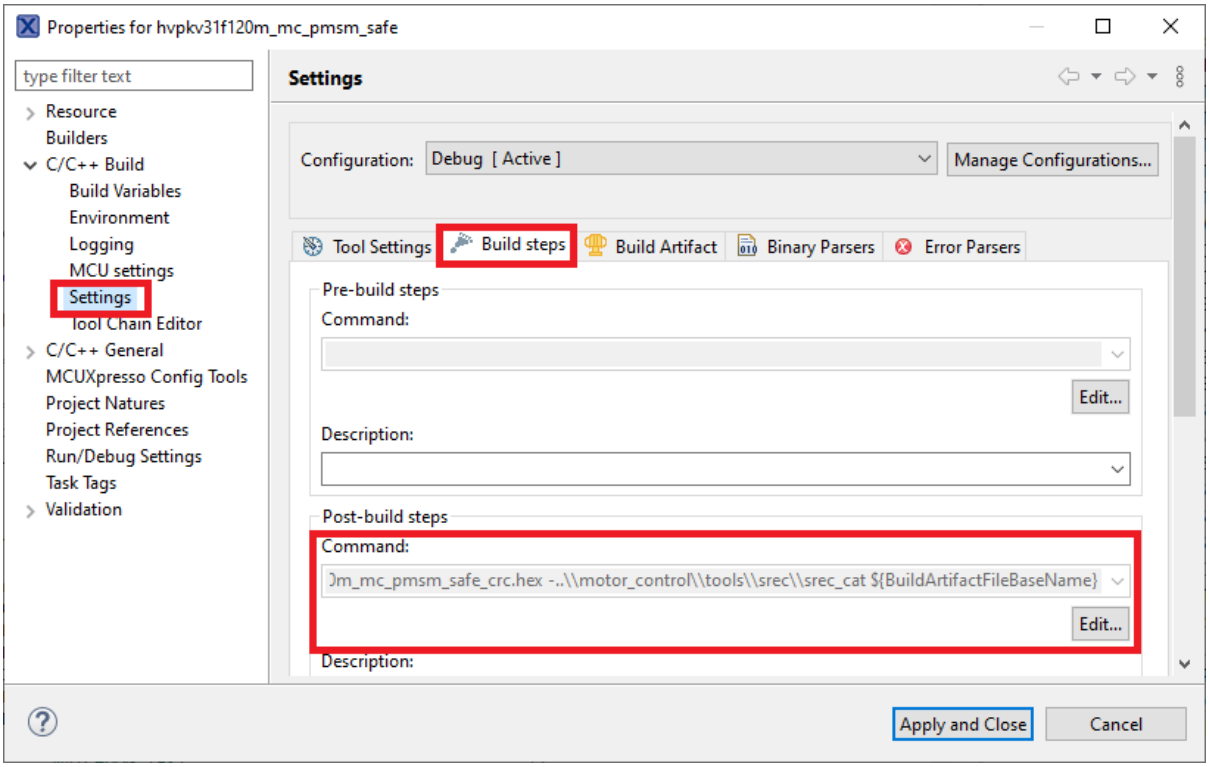


Figure 10. Post-build settings in MCUXpresso IDE

- Build the project
- The final post-processed image can be downloaded to the ROM memory using the GUI Flash Tool (choose the *<board>_mc_pmsm_safe_crc.hex* file in the *Debug*\|*Release* folders).
- The "Attach only" option must be set if you want to debug the application.

**NOTE**

When you debug your application with the flash test turned on, be careful with using breakpoints. The software breakpoint usually changes the CRC result and causes a safety error.

# Chapter 8
# Tools

Install the FreeMASTER Run-Time Debugging Tool 3.0 and one of the following IDEs on your PC to run and control the PMSM application properly:

- IAR Embedded Workbench IDE v8.50.1 or higher
- MCUXpresso v11.2.0
- ARM-MDK - Keil µVision version 5.30

**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.

## 8.1 Compiler warnings

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

The IAR Embedded Workbench IDE suppresses these warnings:

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

The Arm-MDK Keil µVision IDE suppresses these warnings:

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

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

# Chapter 9
# Remote control using FreeMASTER

This section provides information about the tools and recommended control procedures of the 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. The FreeMASTER supports non-intrusive monitoring, as well as the modification of target variables in real time, which is very useful for the algorithm tuning. Besides the target-side driver, the FreeMASTER tool requires the installation of the PC application as well. You can download FreeMASTER 3.0 at www.nxp.com/freemaster. Based on your package, you can run the FreeMASTER application by double-clicking the *pmsm_safe.pmp* file in the *pack_pmsm_safe_hvpkv11/middleware/motor_control/freemaster* folder or the *pmsm_safe_mid.pmp* file in the *pack_pmsm_safe_hvpkv31/middleware/motor_control/freemaster* folder. When the FreeMASTER application starts, the user interface environment is created automatically.

## 9.1 Establishing FreeMASTER communication

The remote operation is provided by FreeMASTER via the USB interface. Perform the following steps to control the MCU application using FreeMASTER:

1. Download the project from your chosen IDE to the MCU and run it.
2. Open the FreeMASTER file *pmsm_safe.pmp* (*pmsm_safe_mid.pmp* for the HVP-KV31F120M board). The *pmsm_safe* project uses the TSA by default, so it is not necessary to select any symbol file in FreeMASTER.
3. Click the "Start communication" button (the green “GO” button in the top left-hand corner) to establish the communication.

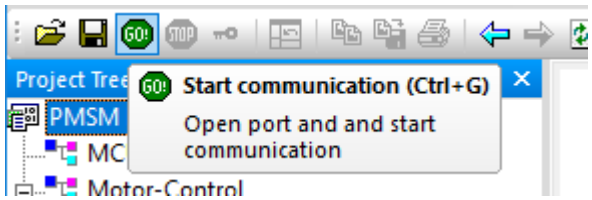


Figure 11. Green “GO” button in top left-hand corner

4. If the communication is established successfully, the FreeMASTER communication status in the bottom right-hand corner changes from “Not connected” to “RS232 UART Communication; COMxx; speed=19200”. Otherwise, the FreeMASTER warning popup window appears.


Figure 12. FreeMASTER—communication is established successfully

5. Control the MCU application directly by writing to the selected FreeMASTER variable in the variable watch window.
6. If you rebuild and download a new code to the target, reload the symbol file (Ctrl + M).

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

1. Go to the “Project → Options → Comm” tab and make sure that a correct RS232 port is selected and the communication speed is set to 19200 bps.

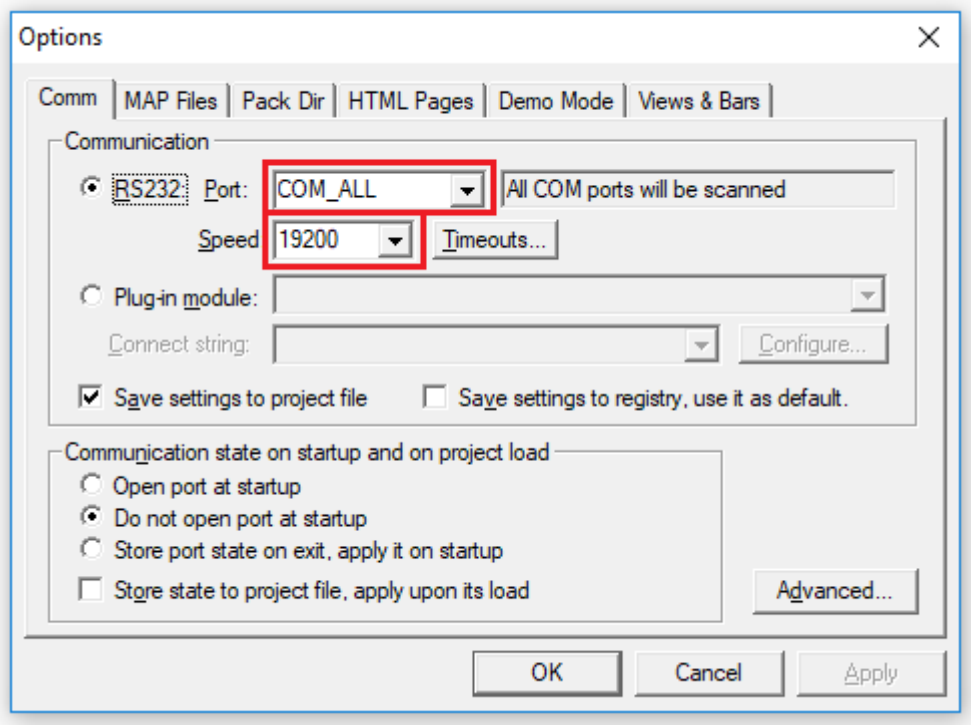


**Figure 13. FreeMASTER communication setup window**

Make sure to supply your development board with a sufficient energy source. Sometimes the PC USB port is not sufficient to supply the development board.

## 9.2 FreeMASTER interface

This section descibes the FreeMASTER interface used for application control. The attached FreeMASTER page does not offer any tools for the algorithm (controller, observer, ramp, and so on) parameter calculation. This means that all motor-control parameters accessible from FreeMASTER must be tuned manually. You can use the non-safety pmsm_snsless application which features the MCAT (Motor Control Application Tuning) tool (*pack_motor_<booard>/boards/hvpkvxxx/demo_apps/mc_pmsm/pmsm_snsless*). See the RTCESL user's guides at www.nxp.com/rtcesl for more detailed information about the motor-control algorithm parameters calculation.

### FreeMASTER layout

When FreeMASTER successfuly connects to the target, the "PMSM FOC Sensorless" page appears. The default FreeMASTER layout is shown in Figure 14 and it consists of the following windows:

- The "Welcome" page contains all essential information about the reference design. The debug "Console" window that shows logs from the program flow is at the right-hand side of the "Welcome" page. The "Console" window can be turned on and off using the nearby button.
- The "Project Tree" window is at the left-hand side, where various sub-blocks, scopes, and recorders can be selected. The individual sub-blocks are described in the following sections.
- The "Variable Watch" window at the bottom is used for editing and visualization of selected FreeMASTER and MCU variables.
- The content of the "Variable Watch" window varies according to the selected sub-block in the "Project Tree" window. Most variables are color-coded and grouped by their prefix (see the "Welcome" page for more details).

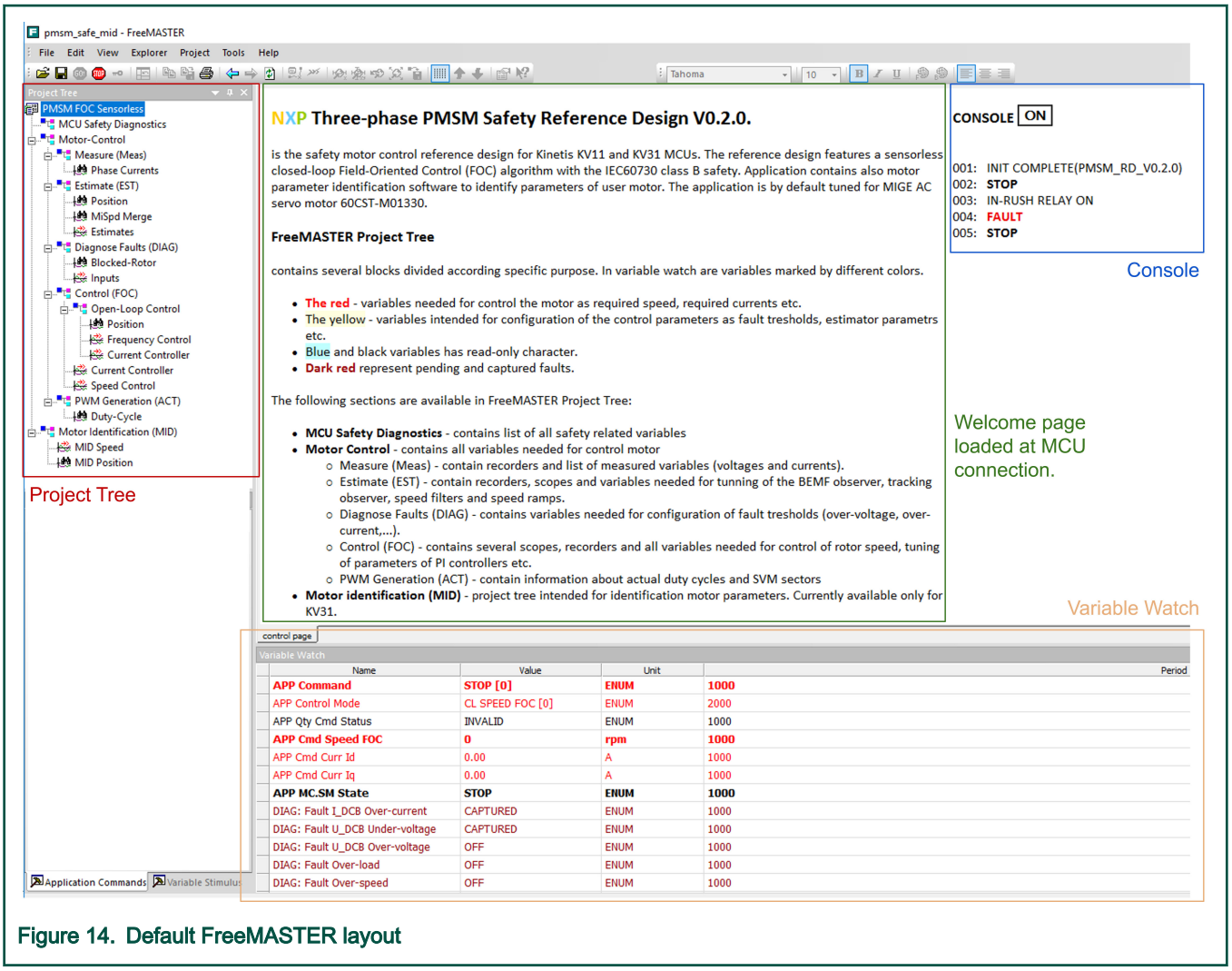


Figure 14. Default FreeMASTER layout

## PMSM FOC Sensorless sub-block

Figure 15 shows the "Variable Watch" window content of the "PMSM FOC Sensorless" root item in the "Project Tree" window. Besides the set of basic control variables, it also contains application information.

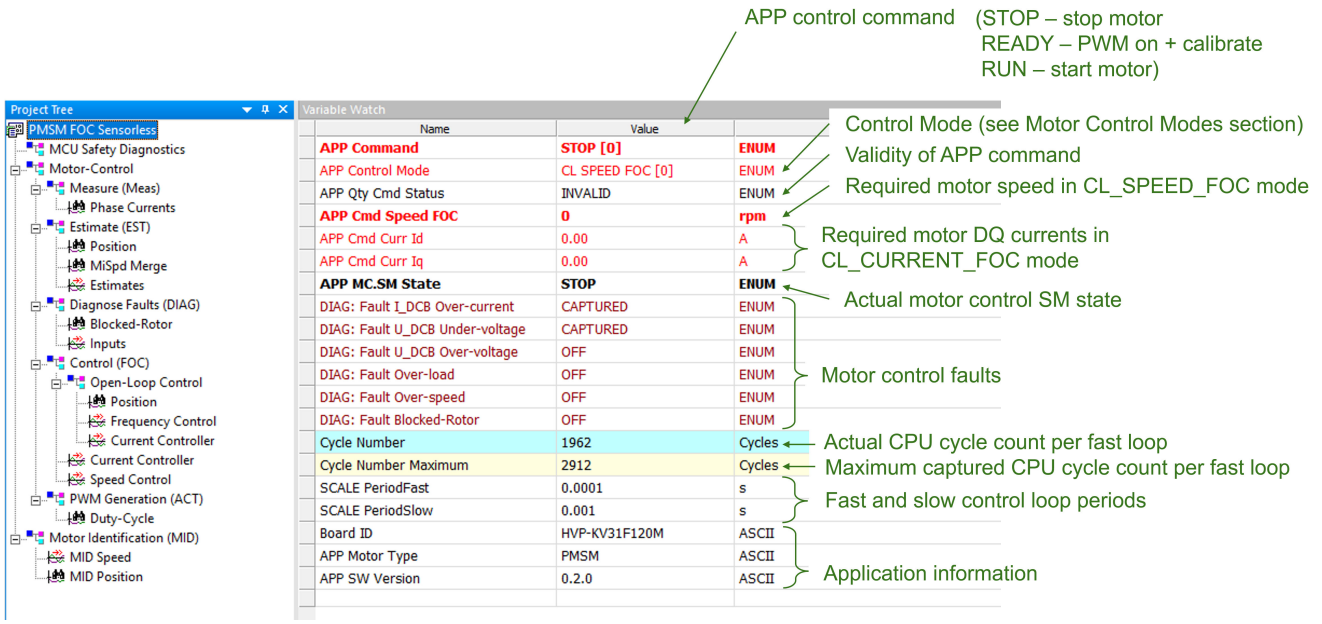


**Figure 15. FreeMASTER PMSM FOC Sensorless sub-block**

## MCU Safety Diagnostics sub-block

The actual status of safety tests is in the "Variable Watch" window of the "MCU Safety Diagnostics" sub-block (see Figure 16).

**NOTE**

This section is only informative and useful in specific cases where the safety error violation does not lead to a loss of communication between FreeMASTER and MCU.

**NOTE**

Some variables (for example *FS: Duration FLASH Test*) are available only in the debug configuration of the example application.

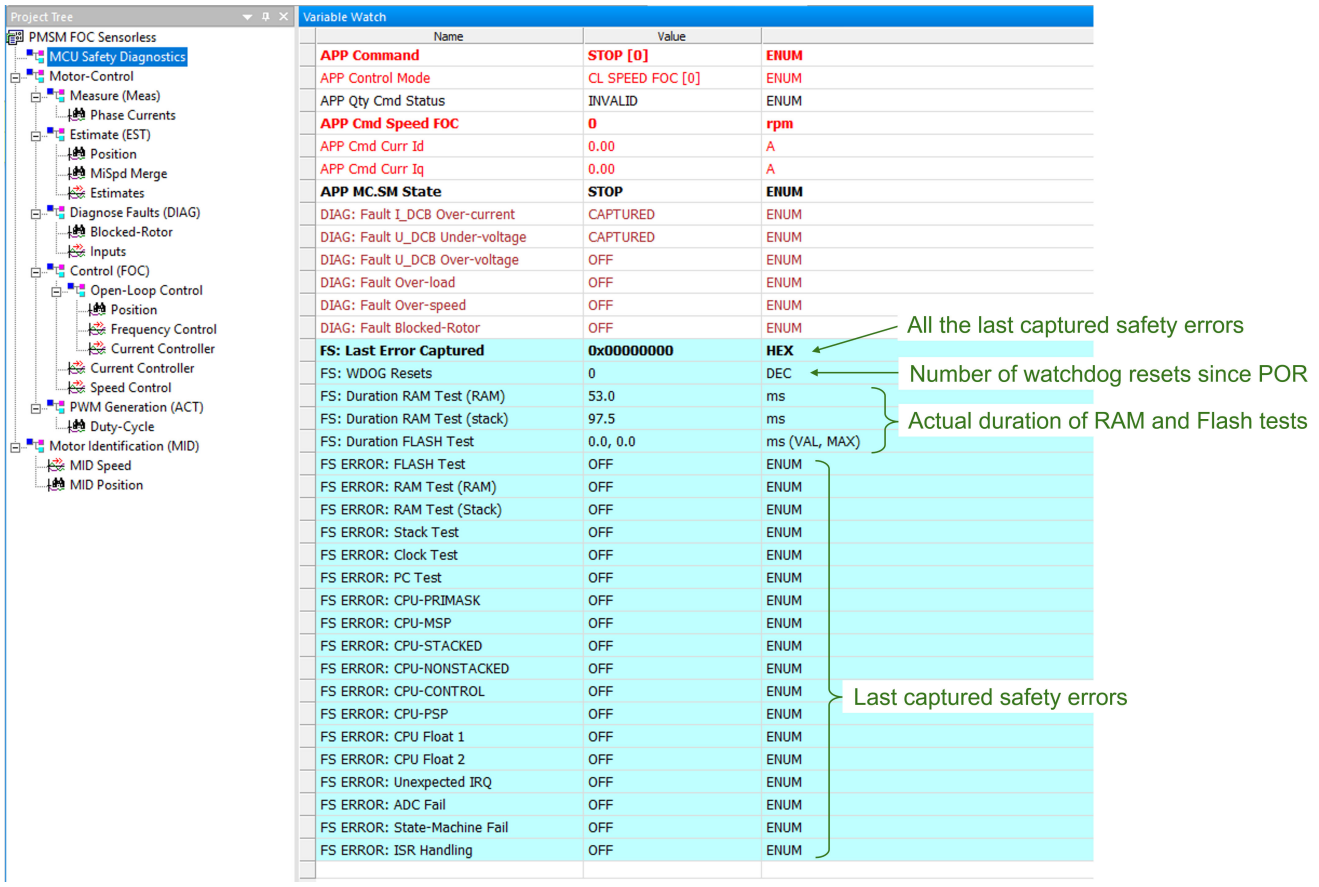


**Figure 16. FreeMASTER MCU Safety Diagnostics sub-block**

### Motor-Control sub-block

The "Motor-Control" sub-block groups motor-control-related scopes, recorders, and "Variable Watch" window sets based on the affiliation to the measurement (Meas), estimation (EST), diagnostics (DIAG), control (FOC), and actuator (ACT) parts of the motor-control software. See Figure 17 for the description of the root "Variable Watch" sub-block.

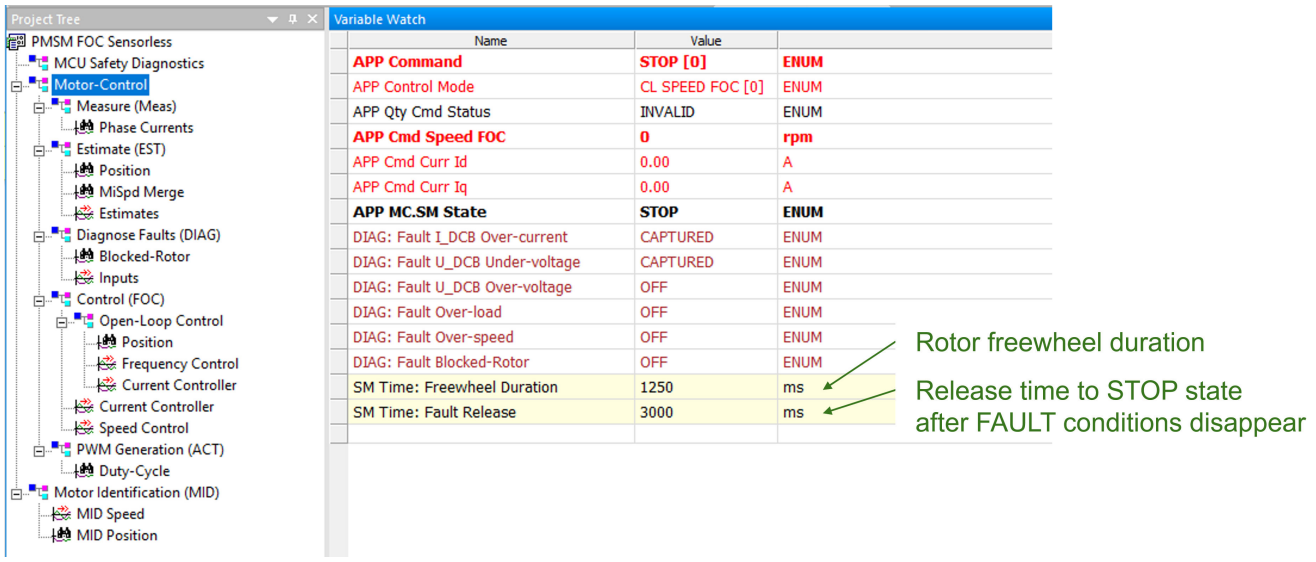


Figure 17. FreeMASTER Motor-Control sub-block

## Measure (Meas) sub-block

All the ADC measurements are in the "Variable Watch" window of the "Measure (Meas)" sub-block (see Figure 18). The measurement filters can be configured here as well.

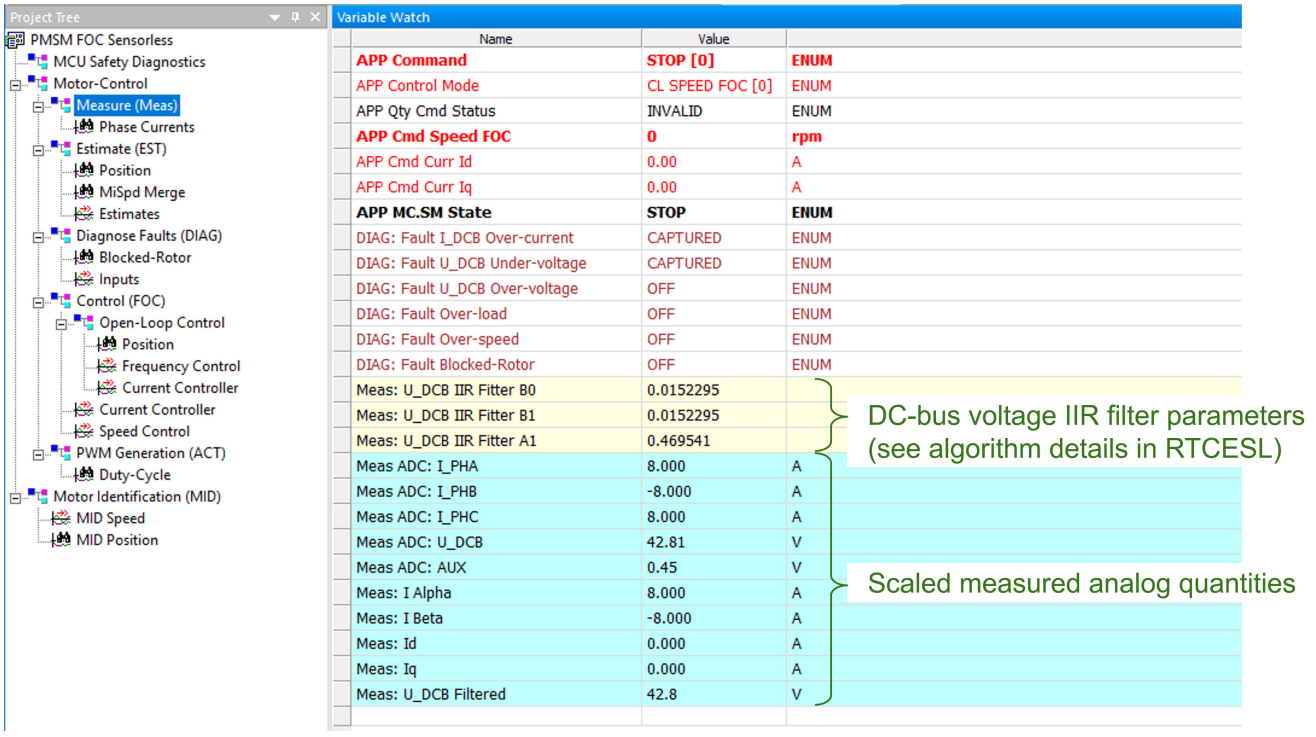


Figure 18. FreeMASTER Measure sub-block

### Estimate (EST) sub-block

All the rotor position and speed estimation algorithms can be configured in the "Variable Watch" window of the "Estimate (EST)" sub-block (see Figure 19). Multiple algorithms are implemented, each used in the given ALIGN, LOSPD, MISPD, and HISPD state-machine states (matches the zero-, low-, medium-, and high-speed regions).

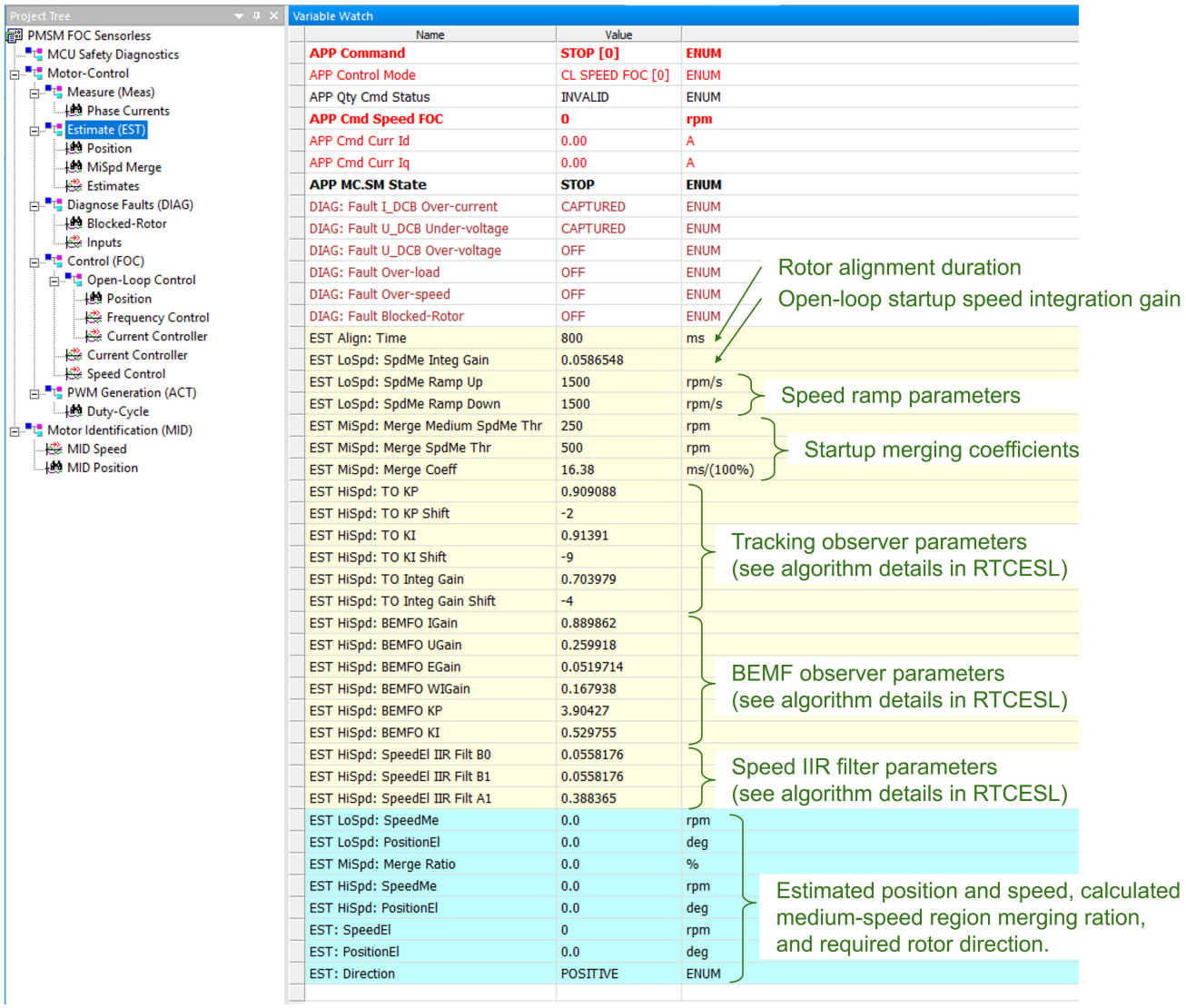


**Figure 19. FreeMASTER Estimate sub-block**

### Diagnose Faults (DIAG) sub-block

The motor-control-related fault diagnostics can be configured in the "Variable Watch" window of the "Diagnose Faults (DIAG)" sub-block (see Figure 20).

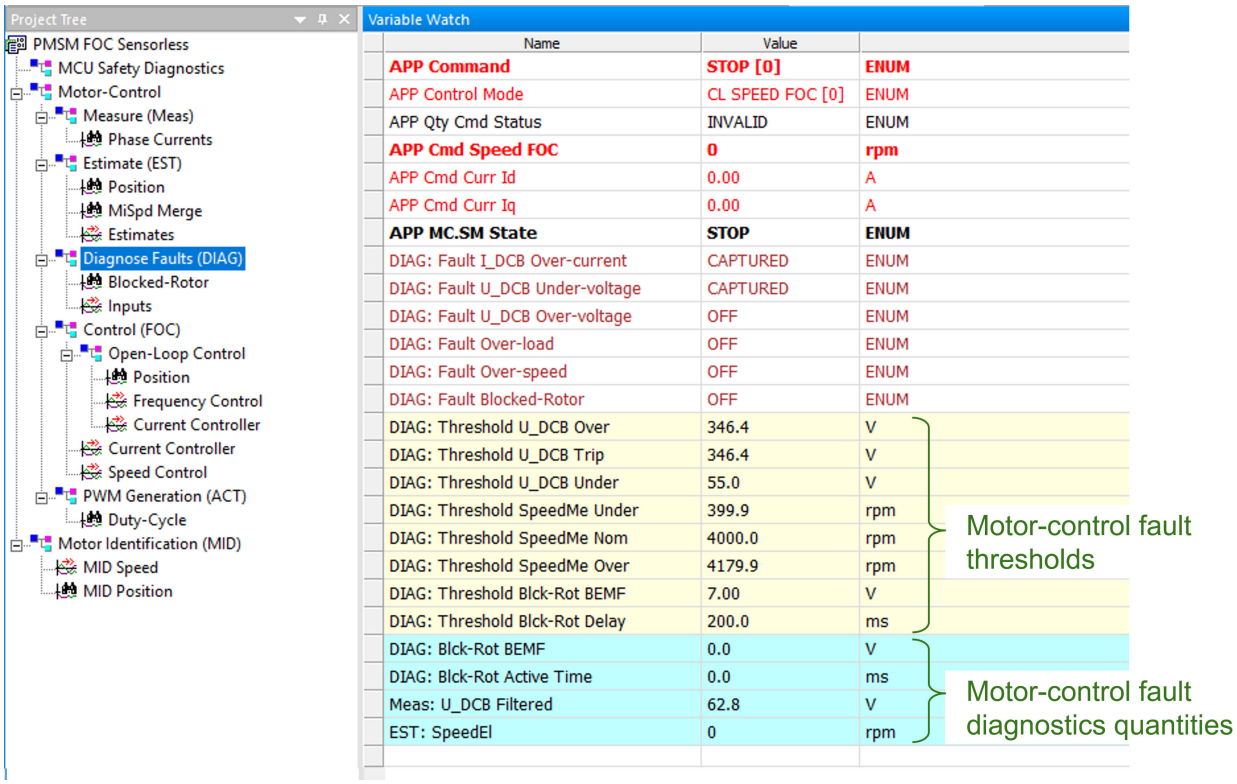


Figure 20. FreeMASTER Diagnose Faults sub-block

### Control (FOC) sub-block

The control part of the motor-control software can be visualized and configured in the "Variable Watch" window of the "Control (FOC)" sub-block (see Figure 21). Multiple algorithms are implemented and used based on the state-machine state and the control mode selected (see Motor-control modes).

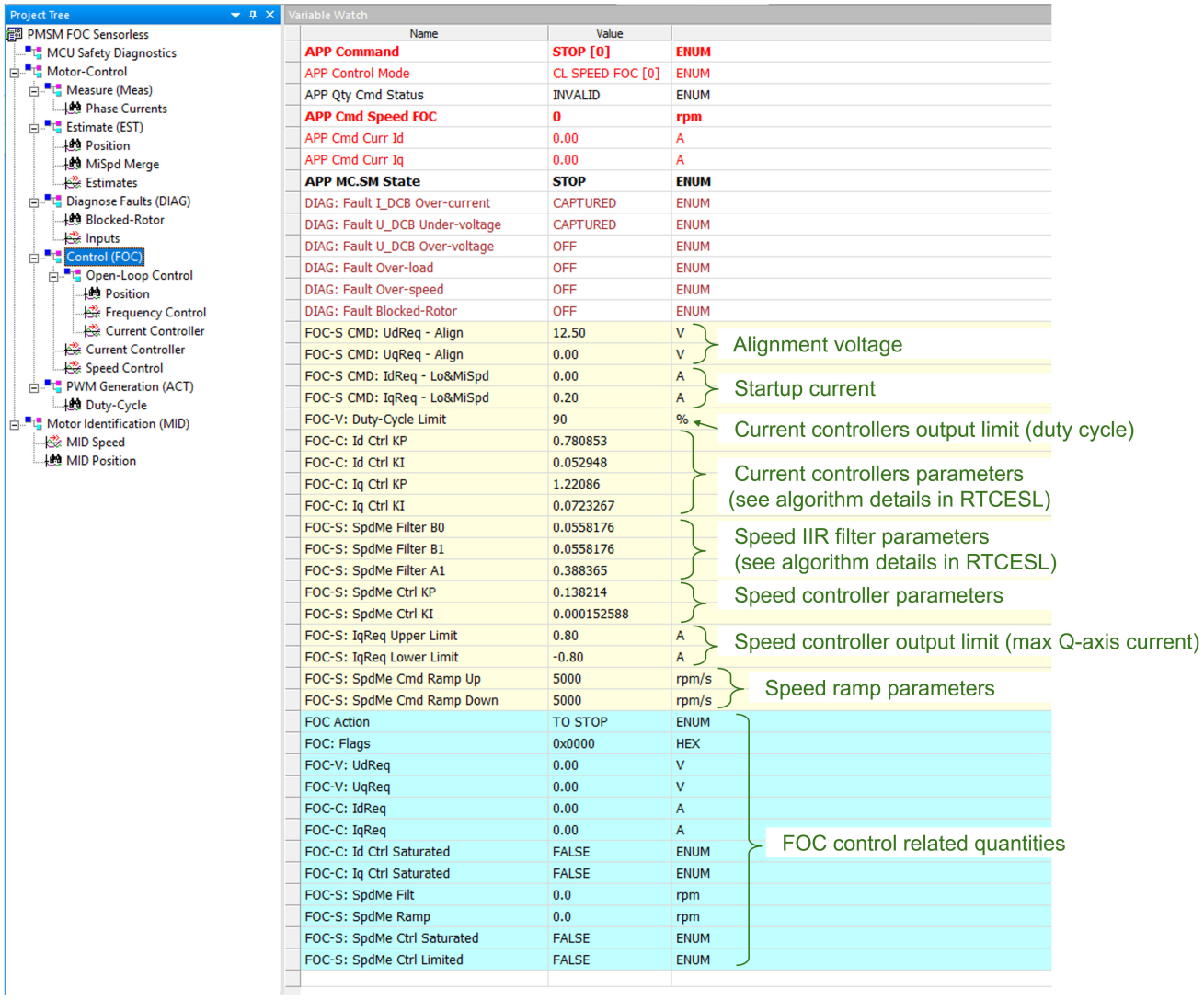


**Figure 21. FreeMASTER Control sub-block**

### Open-Loop Control sub-block

Besides the default Field-Oriented Control (FOC), which allows for full decoupled speed and torque control, other and simpler control modes are implemented as well (scalar control, open-loop current control, and so on) to allow for easier debugging and tuning. See Motor-control modes for more information. To control and configure these algorithms, see the "Variable Watch" window of the "Open-Loop Control" sub-block (shown in Figure 22).

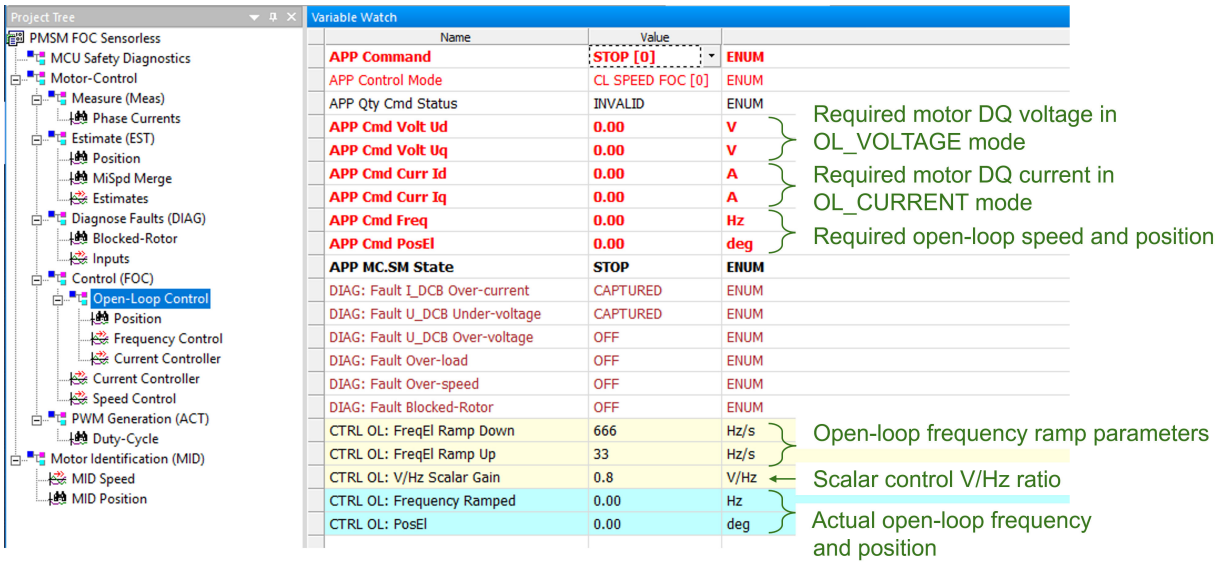


**Figure 22. FreeMASTER Open-Loop Control sub-block**

### PWM Generation (ACT) sub-block

All the PWM generation (actuator) variables are in the "Variable Watch" window of the "PWM Generation (ACT)" sub-block (see Figure 23).

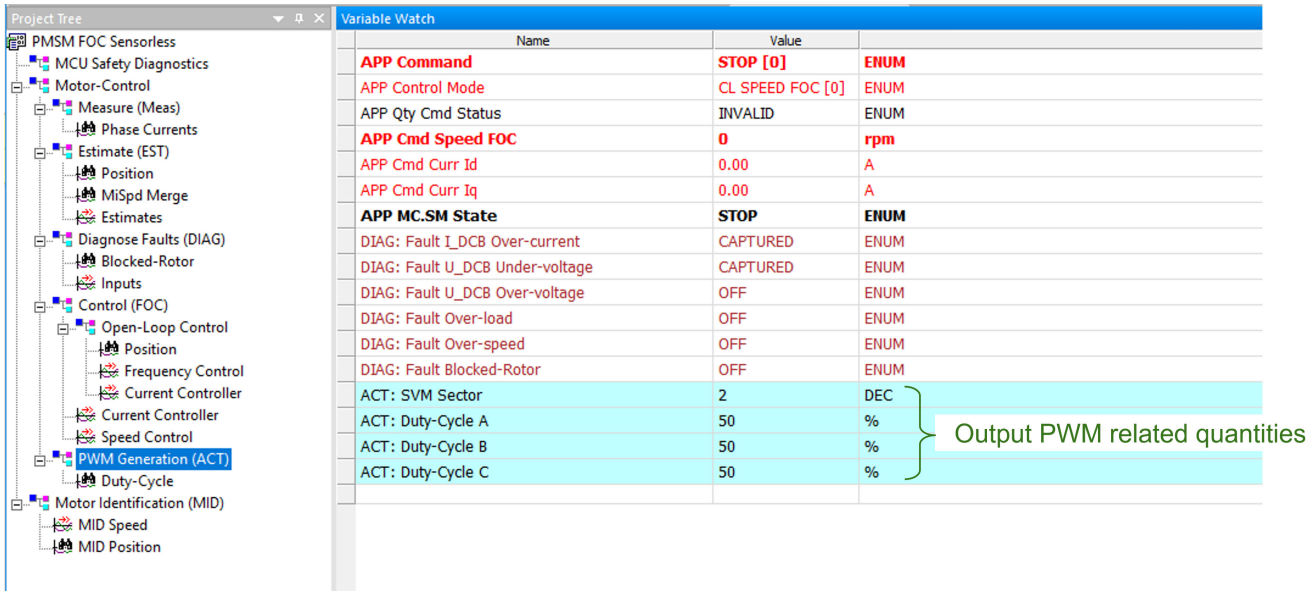


**Figure 23. FreeMASTER PWM Generation sub-block**

### Motor Identification (MID) sub-block

For detailed information about the "Motor Identification (MID)" sub-block, see Motor Identification (MID).

## 9.3 Motor-control modes

The application provides six control modes. The control mode selection is done via the *APP Control Mode* variable. The application must be in the STOP state (*APP MC.SM State* value) when the control mode is being changed.

### Speed FOC control mode

The Speed FOC control mode engages both current controllers and the speed controller. The position feedback is closed and the FOC position and speed are estimated by the BEMF and Tracking observers. The Speed FOC control scheme is depicted in Figure 25. To enable the Speed FOC control, perform the following steps (see also Figure 24):

1. Stop the MC state-machine (*APP Command* = STOP).
2. Select the Speed FOC (*APP Control Mode* = CL_SPEED_FOC).
3. Set the required speed (set the required rotor speed value in the *APP Cmd Speed FOC* variable).
4. Start the MC state-machine (*APP Command* = RUN).

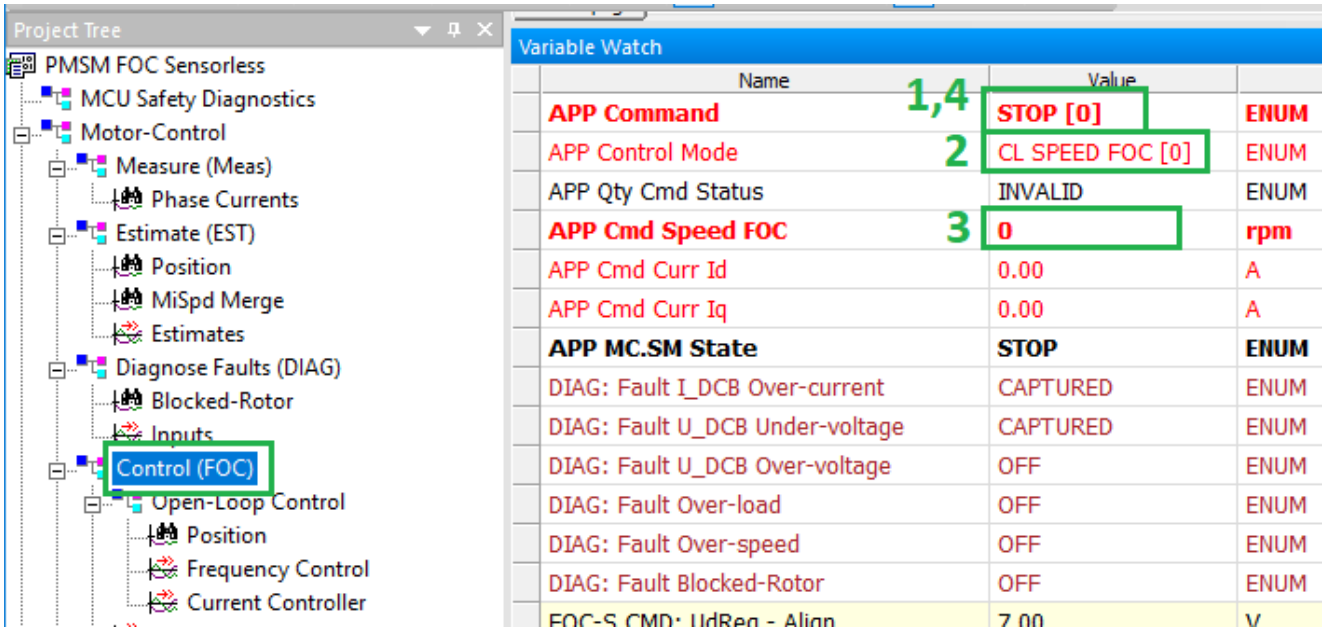


Figure 24. Speed FOC control mode variables

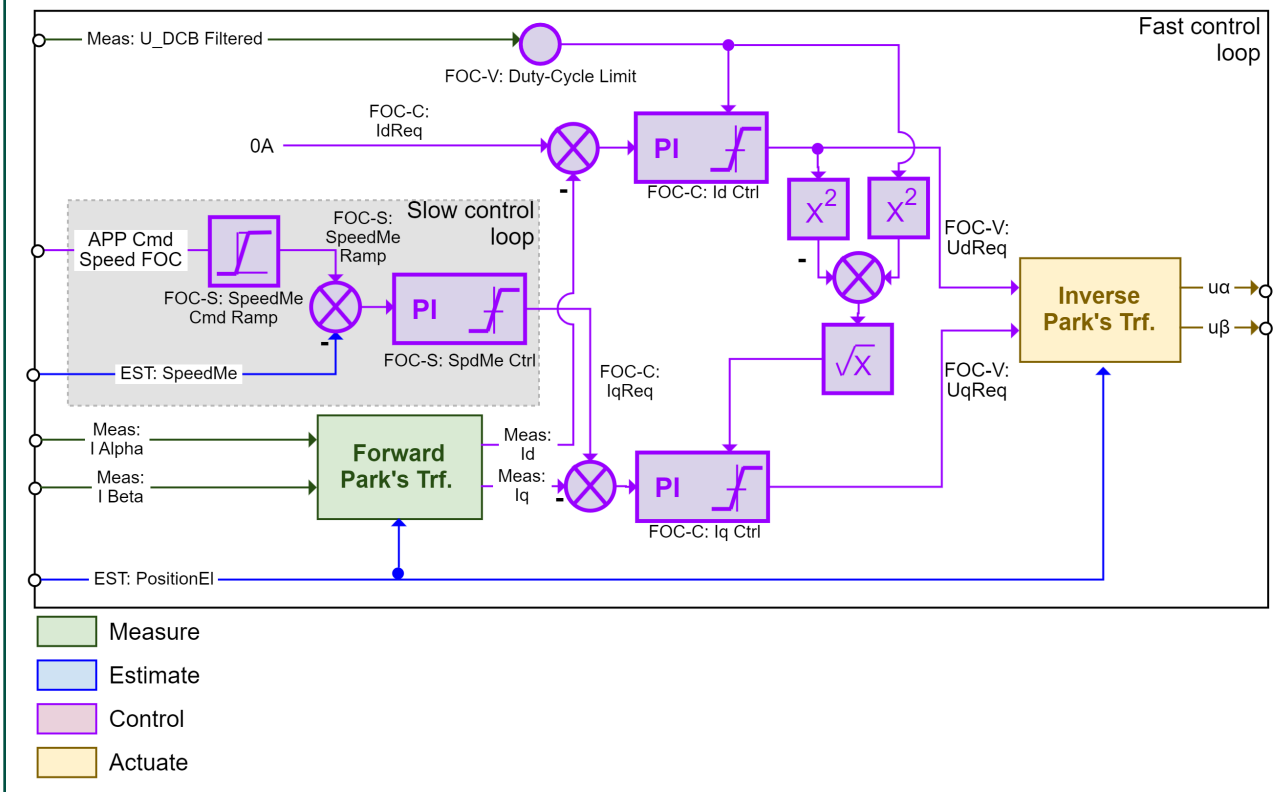


Figure 25. Speed FOC control mode block diagram

### Current FOC control mode

The Current FOC control mode engages only the current controllers. The position feedback is closed, so the FOC position and speed are estimated by the BEMF and Tracking observers. The Current FOC can be used for torque control or if the user has not tuned the speed controller yet. The Current FOC control scheme is shown in Figure 27. To enable the Current FOC control, perform the following steps (see also Figure 26):

1. Stop the MC state-machine (*APP Command* = STOP).
2. Select "Current FOC" (*APP Control Mode* = CL_CURRENT_FOC).
3. Set the required DQ currents (set the required value in the *APP Cmd Curr Id* and *APP Cmd Curr Iq* variables).
4. Start the MC state-machine (*APP Command* = RUN).

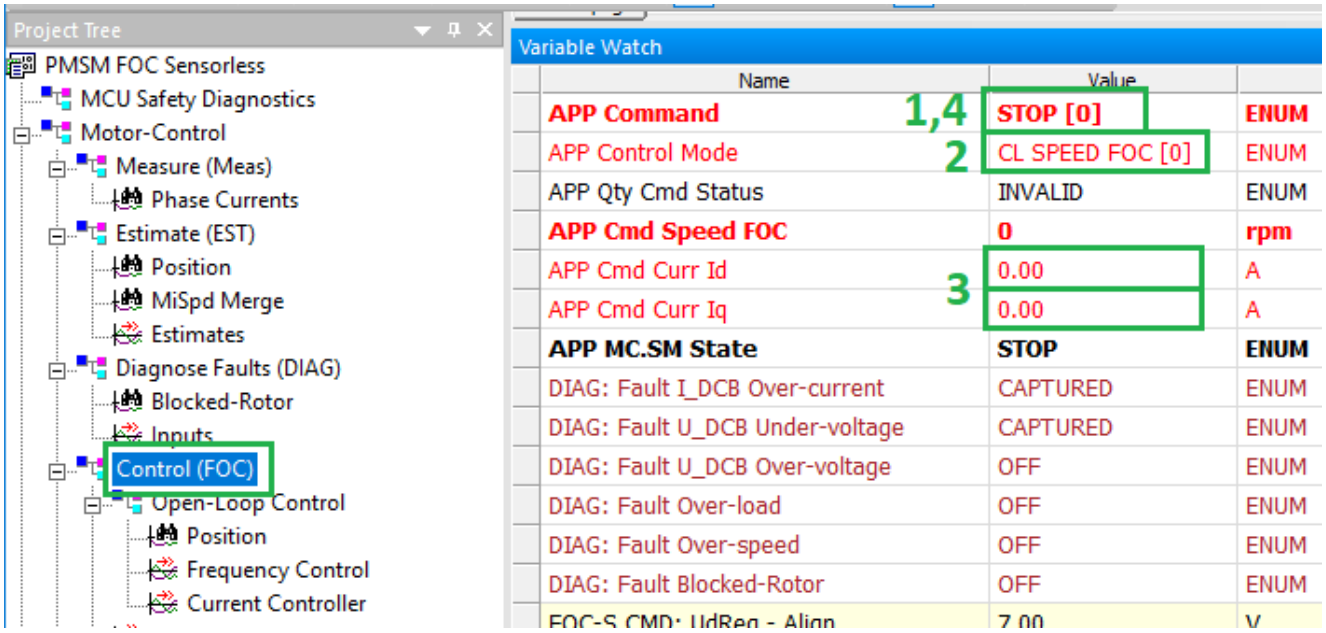


Figure 26. Current FOC control mode variables

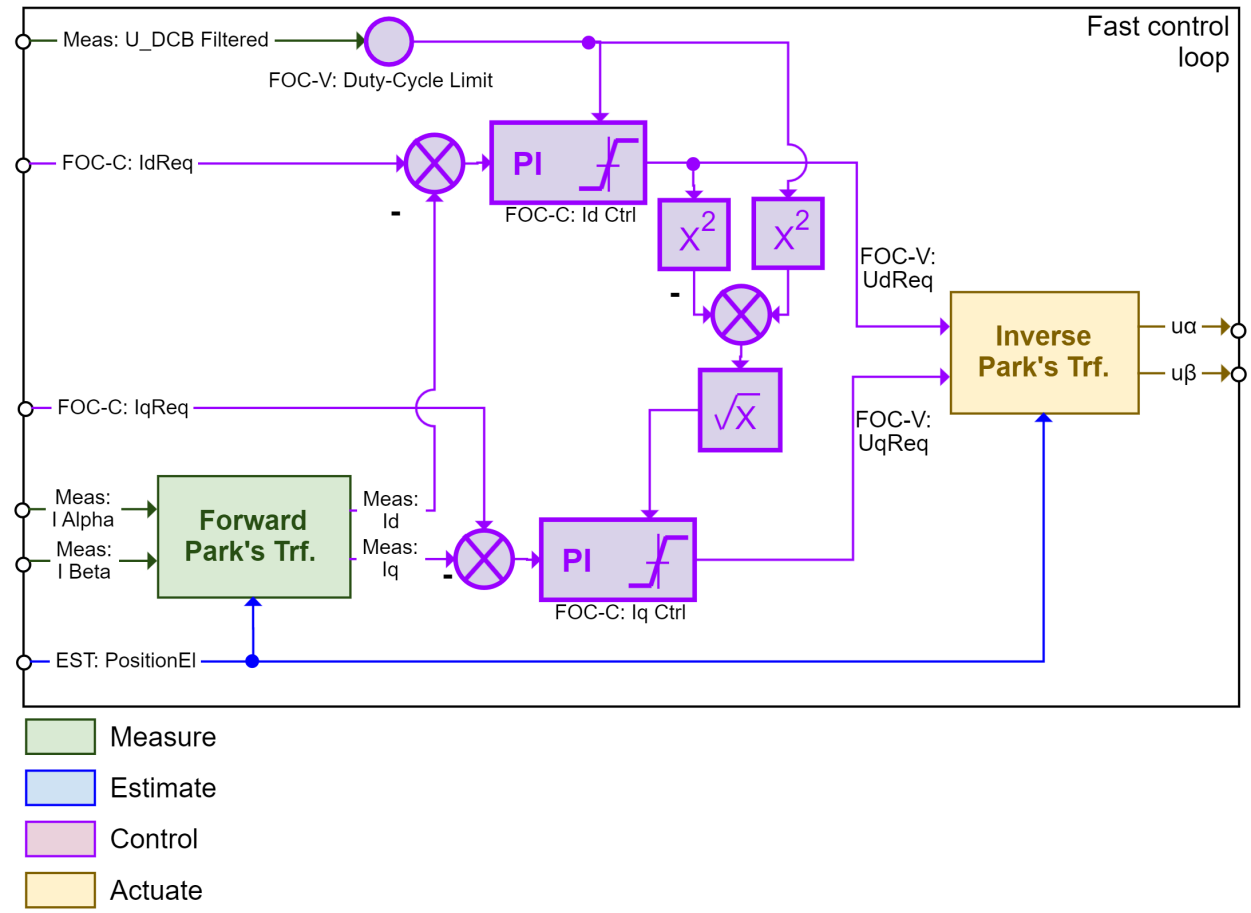


Figure 27. Current FOC control mode block diagram

### Scalar Control mode

The Scalar Control mode requires only few parameters to be configured. It is a very simple method useful for application tuning and debugging. The Scalar Control mode block diagram is shown in Figure 28. A rotating magnetic field is generated by the voltage vector, where amplitude and rotating frequency are proportional via the V/Hz coefficient (*CTRL OL: V/HZ Scalar Gain* variable). The voltage vector position is obtained via the sum of integration of the required frequency *APP Cmd Freq* and the static position bias *APP Cmd PosEl*. The BEMF and Tracking observers are still running in the background, so the Scalar Control mode can be used to debug them. To enable the Scalar Control mode, perform the following steps (see also Figure 28):

1. Stop the MC state-machine (*APP Command* = STOP).
2. Select the scalar control (*APP Control Mode* = OL_SCALAR).
3. Set the required frequency (set the required value in the *APP Cmd Freq* variable).
4. Start the MC state-machine (*APP Command* = RUN).

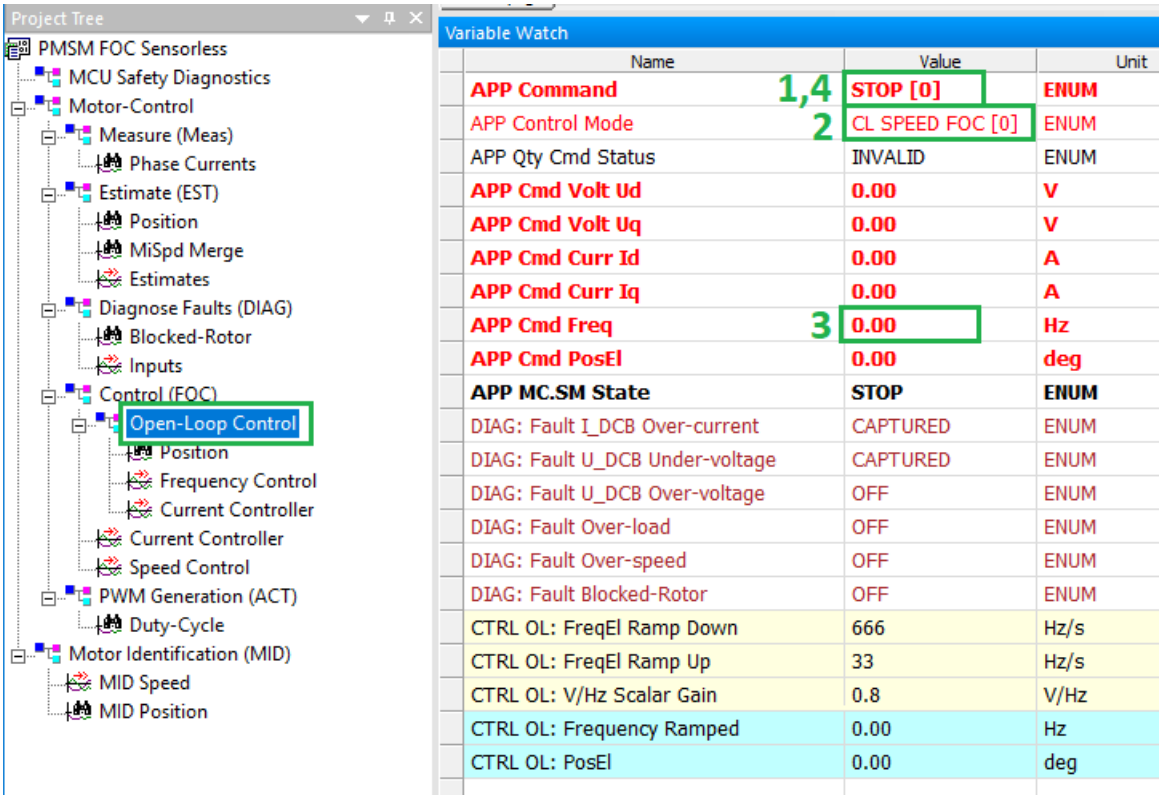


Figure 28. Scalar Control mode variables

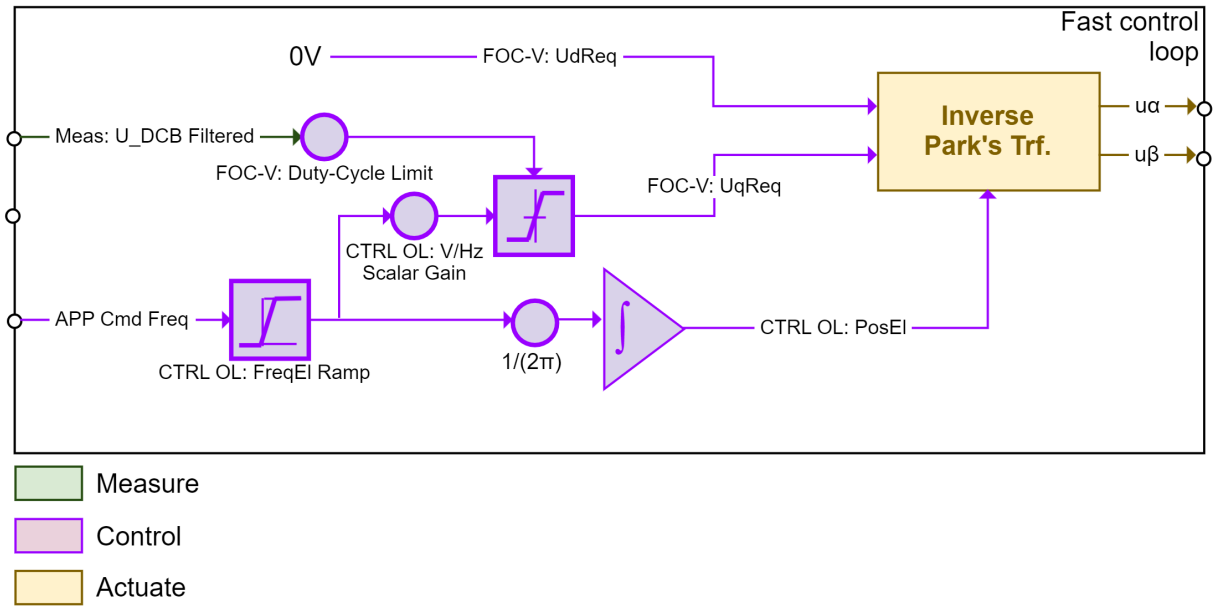


Figure 29. Scalar Control mode block diagram

### Open Loop Current control mode variables

The current controllers are engaged in this mode. Set the required amplitude of DQ currents, the current vector rotation frequency, and the position bias. The BEMF and Tracking observers are running in the background, so this control mode can be used for the parameter tuning of observers and current controllers. The Open Loop Current control scheme is shown in Figure 31. To enable the Open Loop Current control, perform the following steps (see also Figure 30):

1. Stop the MC state-machine (*APP Command* = STOP).
2. Select the scalar control (*APP Control Mode* = OL_CURRENT).
3. Set the required amplitude of the DQ currents (*APP Cmd Curr Id* and *APP Cmd Curr Iq* variables), the current vector open-loop frequency (*APP Cmd Freq* variable), and the position bias (*APP Cmd PosEl* variable).
4. Start the MC state-machine (*APP Command* = RUN).

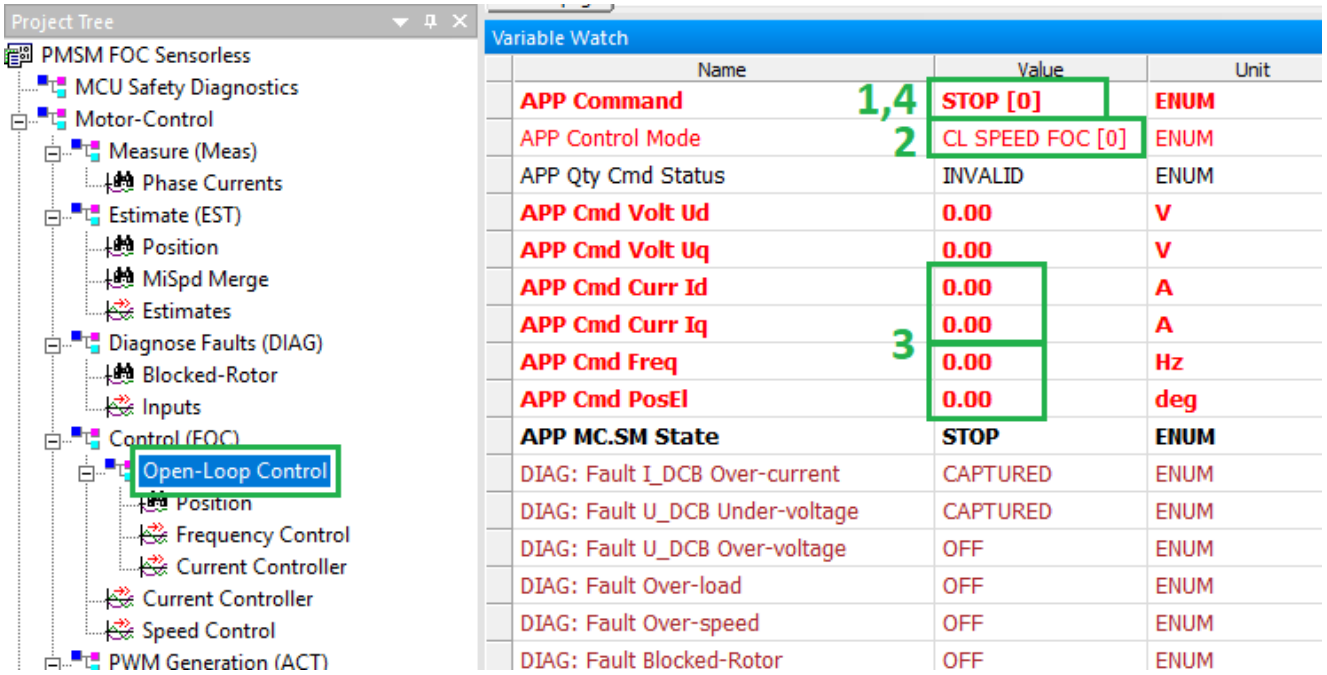


Figure 30. Open Loop Current control selection

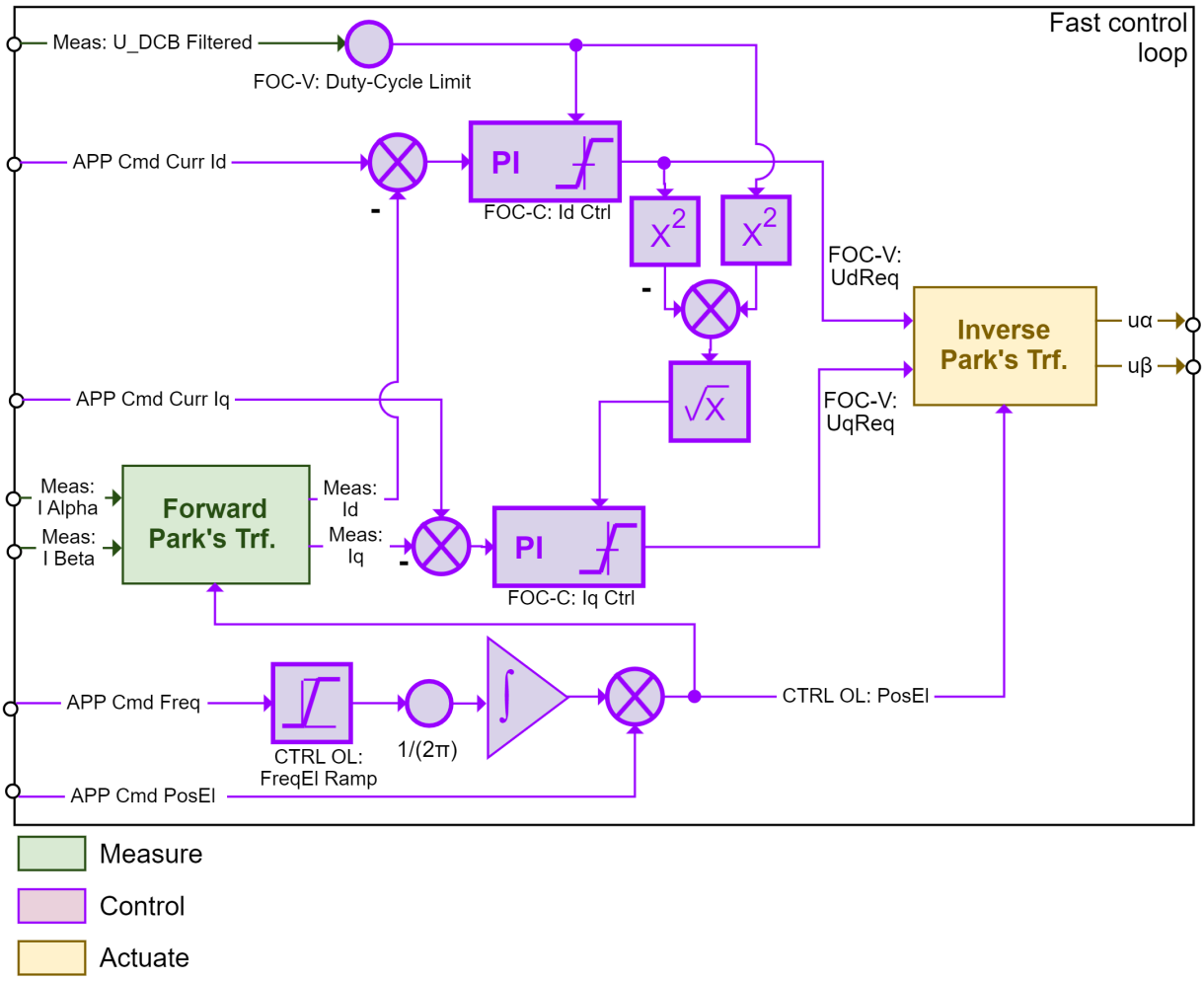


Figure 31. Open Loop Current control mode block diagram

### Open Loop Voltage control mode

The current controllers are disabled in this mode and the stator voltage is controlled directly. Set the required amplitude of DQ voltages, the voltage vector rotation frequency, and the position bias. The BEMF and Tracking observers are running in the background, so this control mode can be used for parameter tuning of observers and basic debugging for the PWM generation. The Open Loop Voltage control scheme is shown in Figure 33. To enable the Open Loop Voltage control, perform the following steps (see also Figure 32):

1. Stop the MC state machine (*APP Command* = STOP).
2. Select the scalar control (*APP Control Mode* = OL_VOLTAGE).
3. Set the required amplitude of DQ voltages (*APP Cmd Volt Ud* and *APP Cmd Volt Uq* variables), the voltage vector open-loop frequency (*APP Cmd Freq* variable), and the position bias (*APP Cmd PosEl* variable).
4. Start the MC state machine (*APP Command* = RUN).

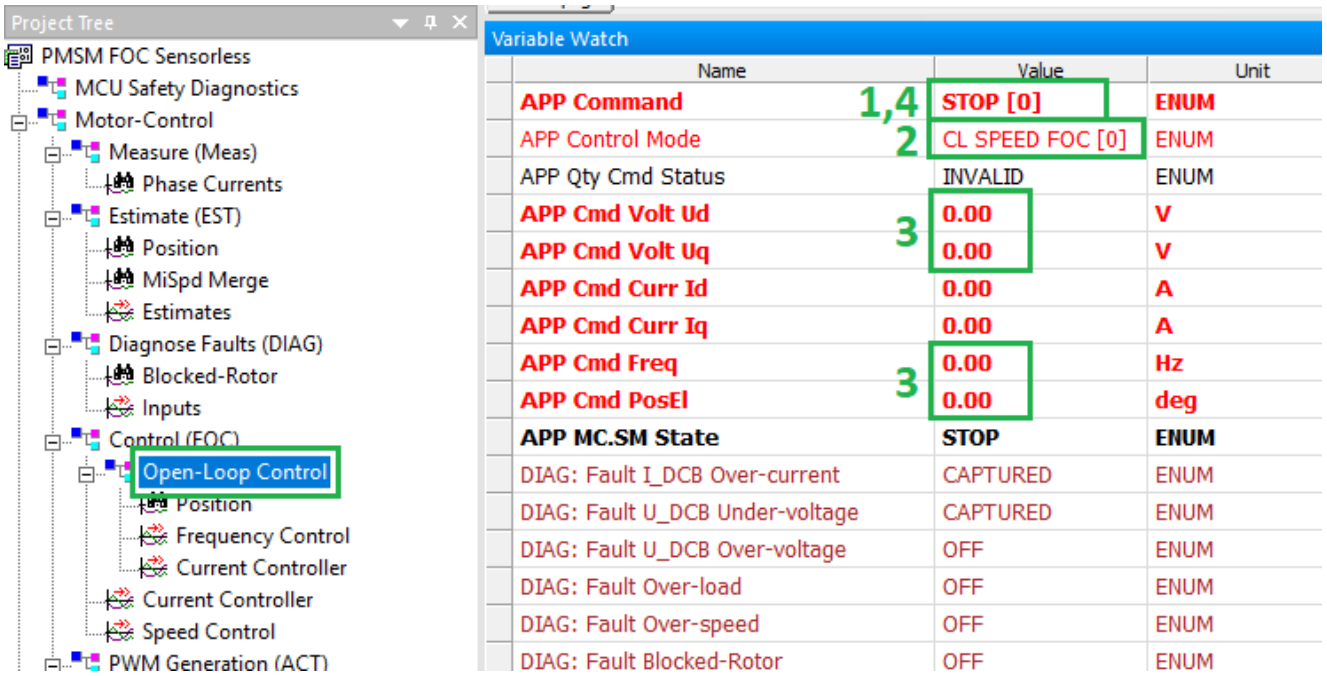


Figure 32. Open Loop Voltage control mode variables

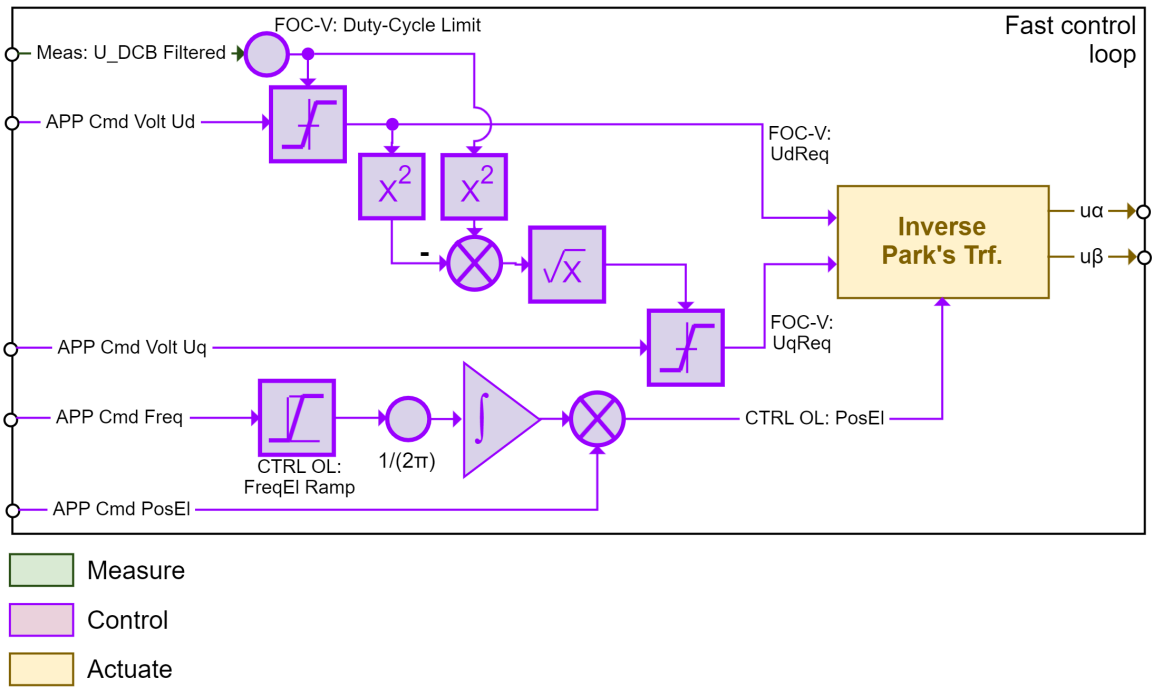


Figure 33. Open Loop Voltage control mode block diagram

### Open Loop MID control mode

This control mode is designed for MID operation and it is not intended for user tuning or debugging purposes. The superior MID state machine uses this control mode to actuate during motor identification. MID is more complex and Motor Identification (MID) guides you on how to run MID.

## 9.4 Motor Identification (MID)

Because the model-based control methods of the PMSM drives provide high performance (dynamic response, efficiency, and so on), obtaining an accurate model of a motor is an important part of the drive design and control. For the implemented FOC algorithms, it is necessary to know the value of the stator resistance $R_s$, direct inductance $L_d$, quadrature inductance $L_q$, and BEMF constant $K_e$. Unless the default PMSM motor described in Hardware setup is used, the motor parameter identification is the first step in the application tuning. This section shows how to identify user motor parameters using MID. MID is written in floating-point arithmetics, so it is available for the HVP-KV31F120M application example only. Each MID algorithm is described in detail in MID algorithms. MID is controlled via the FreeMASTER "Motor Identification (MID)" page shown in Figure 34.


Figure 34. MID FreeMASTER control

### Motor parameter identification using MID

The whole MID is controlled via the FreeMASTER "Variable Watch" "Motor Identification (MID) sub-block shown in Figure 34. The motor parameter identification is as follows:

1. Set the *APP Command* variable to STOP.
2. Select the OL_MID value in the *APP Control Mode* variable to switch to the motor-identification mode.
3. Select the measurement type you want to perform via the *MID: Measurement Type* variable:
   - HW_CALIB - Power stage hardware characterization.
   - PP_ASSIST - Pole-pair identification assistant.
   - EL_PARAMS - Electrical parameters measurement.
   - MECH_PARAMS - Mechanical parameters measurement.
4. In case of EL_PARAMS, schedule the desired electrical parameters, which should be measured using the *MID: Schedule Rs*, *MID: Schedule Ld*, *MID: Schedule Lq*, and *MID: Schedule Ke* variables.
5. Insert the known motor parameters via the *MID: Known Param* set of variables. All parameters with a non-zero known value are not measured again and they are used to infer other parameters (if necessary).
6. Set the measurement configuration paramers in the *MID: Config* set of variables.
7. Start the measurement by setting *APP Command* to RUN.
8. Observe the *MID Start Result* variable for the MID measurement plan validity (see Table 7) and the actual *MID: State*, *MID: Faults* (see Table 5), and *MID: Warnings* (see Table 6) variables.
9. When the measurement is successfully finished, the measured motor parameters are in the *MID: Measured* set of variables.

### MID faults and warnings

The MID faults and warnings are saved in the format of masks in the *MID: Faults* and *MID: Warnings* variables. Faults and warnings are cleared by automatically starting a new measurement. If a MID fault appears, the measurement process immeadiatelly stops and brings the MID state machine safely to the STOP state. If a MID warning appears, the measurement process continues. Warnings report minor issues during the measurement process. See Table 5 and Table 6 for more details on individual faults and warnings.

**Table 5. Measurement faults**

| Fault mask | Fault description | Fault reason | Troubleshooting |
|---|---|---|---|
| b#0010 | Motor is not connected. | $I_s$ > 50 mA cannot be reached with the available DC-bus voltage. | Check that the motor is connected. |
| b#0100 | $R_s$ is too high for calibration. | The calibration cannot be done with the available DC-bus voltage. | Use a motor with a lower $R_s$ for the power stage characterization. |
| b#1000 | Mechanical measurement timeout. | Some part of the mechanical measurement (acceleartion, deceleration) took too long and exceeded 10 seconds. | Raise the *MID: Config Mech Iq Accelerate* or lower the *MID: Config Mech Iq Decelerate*. |

**Table 6. Measurement warnings**

| Warning mask | Warning description | Warning reason | Troubleshooting |
|---|---|---|---|
| b#00001 | The measurement DC current $I_s$ could not be reached. | The user-defined *MID: Config Rs Id Meas* was not reached, so the measurement was taken with a lower $I_s$ DC current. | Raise the DC-bus voltage to reach the $I_s$ DC current or lower the *MID: Config Rs Id Meas* to avoid this warning. |

*Table continues on the next page...*

**Table 6. Measurement warnings (continued)**

| Warning mask | Warning description | Warning reason | Troubleshooting |
|---|---|---|---|
| b#00010 | The AC current amplitude measurement $I_s$ could not be reached. | The user-defined *MID: Config Ls Id Meas* was not reached, so the measurement was taken with a lower $I_s$ AC current. | Raise the DC-bus voltage or lower the $F_{min}$ to reach the $I_s$ AC or lower the $I_s$ AC current to avoid this warning. |
| b#00100 | $R_s$ is out of range. | The measured $R_s$ is negative. The characterization data may be wrong. | Repeat the hardware characterization process. |
| b#01000 | $L_s$ is out of range. | The measured $L_s$ is negative. | Repeat the $L_s$ identification with different *MID: Config Ls* parameters. |
| b#10000 | $K_e$ is out of range. | The measured $K_e$ is negative. | Visualy check whether the motor was spinning properly during the $K_e$ measurement. |

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

**Table 7. MID Start Result variable**

| MID Start Result mask | Description | Troubleshooting |
|---|---|---|
| b#00001 | The $R_s$ value is missing. | Schedule $R_s$ for measurement or enter its value manually. |
| b#00010 | The $L_d$ value is missing. | Schedule $L_d$ for measurement or enter its value manually. |
| b#00100 | The $L_q$ value is missing. | Schedule $L_q$ for measurement or enter its value manually. |
| b#01000 | The $K_e$ value is missing. | Schedule $K_e$ for measurement or enter its value manually. |
| b#10000 | The *Pp* value is missing. | Enter the *Pp* value manually. |

## 9.5 MID algorithms

This section describes how each MID algorithm works.

### Power stage hardware characterization

Each inverter introduces the total error voltage $U_{error}$, which is caused by the dead time, current clamping effect, and transistor voltage drop. The actual inverter output voltage is lower than the voltage required by the $U_{error}$. The total error voltage $U_{error}$ depends on the actual phase current $i_s$ and this dependency is measured during the power stage characterization process. An example of the inverter error characteristic is shown in Figure 35. To perform the characterization, a motor with a known stator resistance *MID: Config PwrStg Rs Calib* must be connected and its known value must be set. The calibration range (range of the stator current $i_s$, in which the measurement of $U_{error}$ is performed) can be set manually in *MID: Config PwrStg Id Calib*. The characterization gradually performs 65 $i_s$ current steps (from $i_s$ = *-MID: Config PwrStg Id Calib* to $i_s$ = *+MID: Config PwrStg Id Calib*) with each taking 300 ms, so the characterization process takes about 20 seconds and the motor must withstand this load. The acquired characterization data is saved to an array and used later for the phase voltage correction during the $R_s$ measurement process. The following $R_s$ measurement can be done with the maximum *MID: Config PwrStg Id Calib* current. It is recommended to use a motor with the $R_s$ ranging from 1 Ω to 30 Ω for characterization purposes.

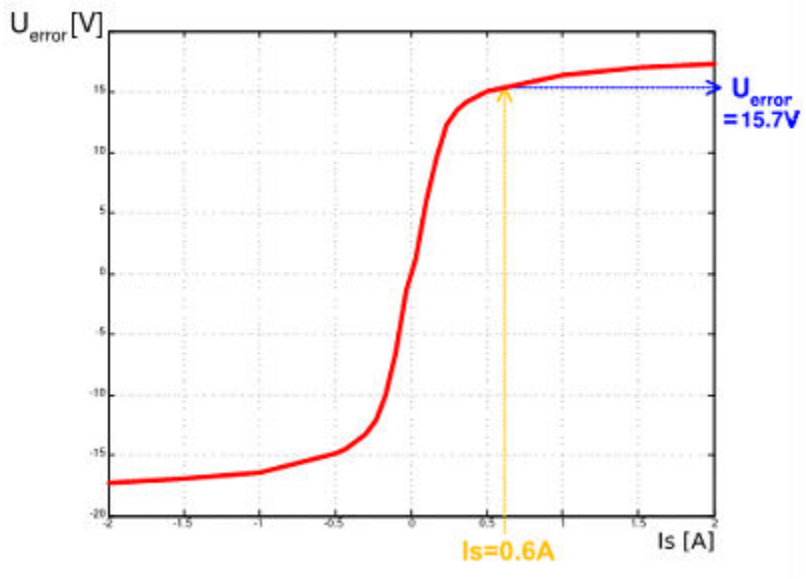


Figure 35. Example power stage characteristic

### Stator resistance measurement

The stator resistance $R_s$ is measured using the DC current *MID: Config Rs Id Meas* value, which is applied to the motor for 1200 ms. The corresponding DC voltage $U_{DC}$ is found using the current controllers. The current controller parameters are selected conservatively to ensure stability. The stator resistance $R_s$ is calculated using the Ohm's law as follows:

$$R_s = \frac{U_{DC} - U_{error}}{I_{phN}} \ [\Omega]$$

### Stator inductance

The stator inductance $L_s$ is measured with an AC signal. Sinusoidal measurement voltage is applied to the motor. The frequency and amplitude of the sinusoidal voltage are obtained before the actual measurement, during the tuning process. The tuning process begins with a 0 V amplitude and the *MID: Config Ls Freq Start* frequency, which are applied to the motor. The amplitude is gradually increased by the *MID: Config Ls Ud Increment* step up to a half of the DC-bus voltage (DCbus/2), until the *MID: Config Ls Id Meas* amplitude is reached. If *MID: Config Ls Id Meas* is not reached even with DCbus/2 and *MID: Config Ls Freq Start*, the frequency of the measuring signal is gradually decreased by *MID: Config Ls Freq Decrement* down to *MID: Config Ls Freq Min* again, until *MID: Config Ls Id Meas* is reached. If *MID: Config Ls Id Meas* is still not reached, the measurement continues with DCbus/2 and *MID: Config Ls Freq Min*. The tuning process is shown in Figure 36.

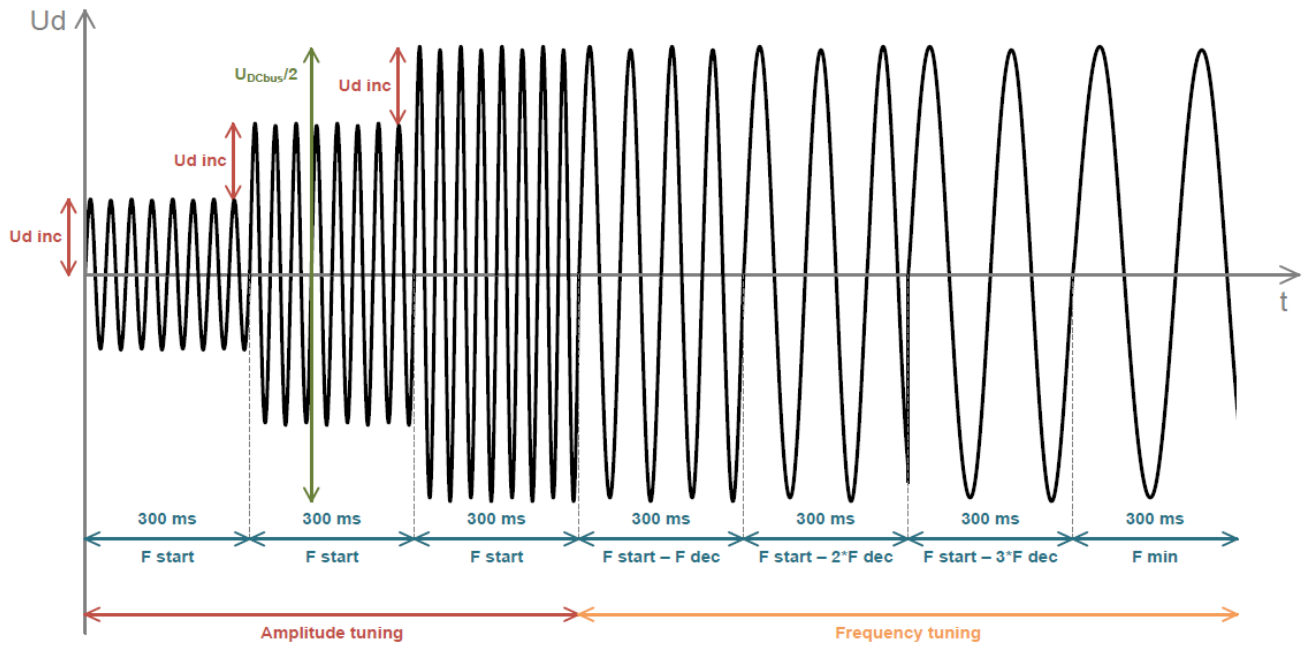


Figure 36. Tuning of Ls measuring signal

When the tuning process is complete, the sinusoidal measurement signal (with the amplitude and frequency obtained during the tuning process) is applied to the motor. The total impedance of the RL circuit is then calculated from the voltage and current amplitudes and $L_s$ is calculated from the total impedance of the RL circuit.

$$Z_{RL} = \frac{U_d}{I_{d\ ampl}}\ [\Omega]$$

$$X_{Ls} = \sqrt{Z_{RL}^2 - R_s^2}\ [\Omega]$$

$$L_s = \frac{X_{Ls}}{2\pi f}\ [\Omega]$$

The direct inductance $L_d$ and quadrature inductance $L_q$ measurements are done in the same way as $L_s$. Before the $L_d$ and $L_q$ measurement is made, DC current is applied to the D-axis and the rotor is aligned to zero electrical degrees. For the $L_d$ measurement, the sinusoidal voltage is applied in the D-axis. For the $L_q$ measurement, the sinusoidal voltage is applied in the Q-axis.

### BEMF constant measurement

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

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

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

- Increase *MID: Config Ke Id Required* to produce higher torque when spinning during the open loop.
- Decrease *MID: Config Ke Freq El. Required* to decrease the required speed for the $K_e$ measurement.

### Number of pole-pair assistant

The number of pole-pairs *Pp* cannot be measured without a position sensor. However, there is a simple assistant called PP_ASSIST, which can help you to determine the number of pole-pairs *Pp*. When PP_ASSIST is started, it performs one electrical revolution and stops for a few seconds. This repeats until the PP_ASSIST is not manually stopped. Because the *Pp* value is the ratio between the electrical and mechanical speeds, it can be determined as the number of observed rotor stops per one mechanical revolution. It is recommended not to count the rotor stops during the first mechanical revolution because the alignment occurs there and it might affect the number of observed stops. During the *Pp* measurement, the current control loop is enabled and the $I_d$ current is controlled to *MID: Config Pp Id Meas*. The electrical position is generated by integrating the open-loop frequency *MID: Config Pp Freq El. Required*. If the rotor does not move after the start of the PP_ASSIST, stop the assistant, increase *MID: Config Pp Id Meas*, and restart the process.

### Mechanical parameters measurement

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

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

The $\omega_m$ character in the equation is the mechanical speed. The mechanical parameter identification algorithm uses the torque profile (see Figure 37). The loading torque is (for simplicity reasons) said to be zero during the whole measurement. Only the friction and the motor-generated torques are considered. During the first phase of measurement, the constant torque $T_{meas}$ is applied and the motor accelerates to 50 % of its nominal speed in time $t_1$. These integrals are calculated during the period from $t_0$ (the speed estimation starts to be accurate enough) to $t_1$:

$$T_{int} = \int_{t_0}^{t_1} T dt \ [Nms]$$

$$\omega_{int} = \int_{t_0}^{t_1} \omega_m dt \ [rad/s]$$

During the second phase, the rotor decelerates freely with no generated torque, only by friction. This enables you to simply measure the mechanical time constant $\tau_m = J/B$ as the time in which the rotor decelerates from its original value by 63 %. The final mechanical parameter estimation can be calculated using integration as follows:

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

The moment of inertia is as follows:

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

The viscous friction is then derived from the relation between the mechanical time constant and the moment of inertia.

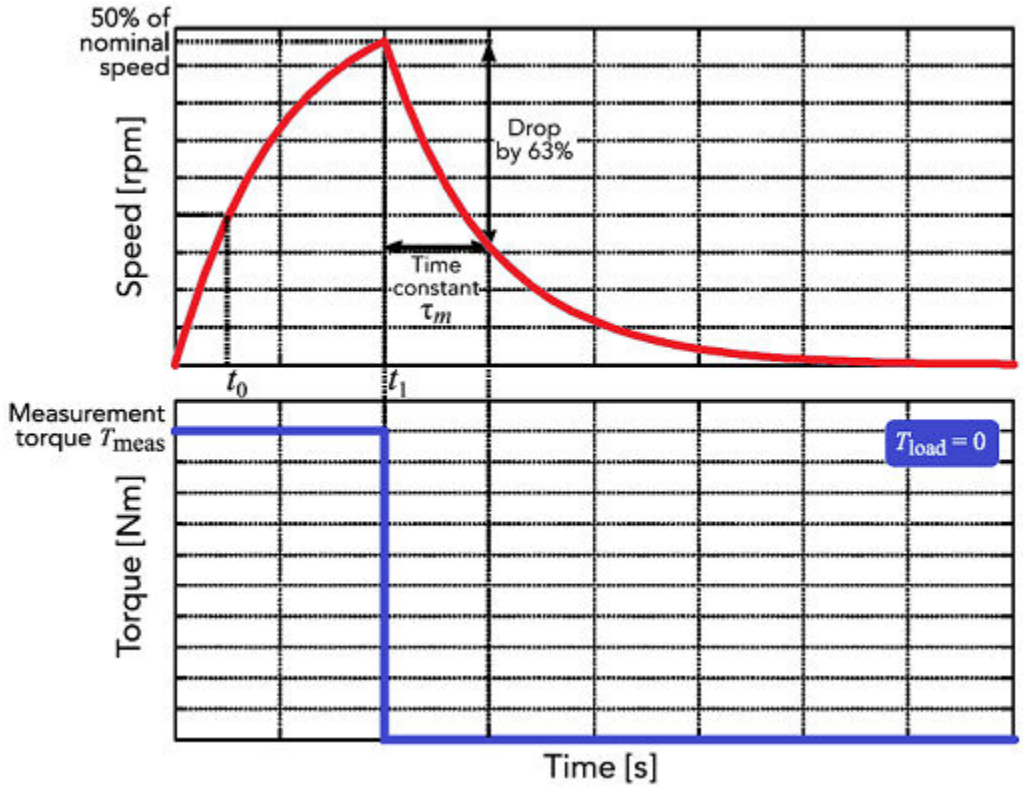


Figure 37. Mechanical parameter identification torque and speed profile

# Chapter 10
# References

The following references are available on www.nxp.com:

1. *MCUXpresso SDK 3-Phase PMSM Control (KV)* (document 3PPMSMCKVUG)
2. *IEC60730B Safety Library Example User's Guide*
3. *Safety Class B with PMSM Sensorless Drive* (document AN5321)
4. *Calculating Post-Build CRC in Arm® Keil®* (document AN12520)

# Chapter 11
# Useful links

1. 3-Phase PMSM Control home page
2. IEC 60730 Safety Standard for Household Appliances
3. RTCESL - Real-Time Control Embedded Software Motor Control and Power Conversion Libraries
4. MCUXpresso IDE - Importing MCUXpresso SDK
5. MCUXpresso SDK Builder - SDK examples in several IDEs

*How To Reach Us*

**Home Page:**

nxp.com

**Web Support:**

nxp.com/support



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

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

**Date of release: 06/2020**

**Document identifier: MCUXSDK3PPMSMCWSUG**

arm