STM32 motor control SDK - 6-step firmware library

Introduction

The UM3042 manual describes the software library that implements the 6-step algorithm, also known as a trapezoidal algorithm, contained in the X-CUBE-MCSDK-6 motor control software development kits (SDKs) designed for, and to be used with, STM32 microcontrollers. The 6-step library allows to control a 3-phase permanent magnet (PMSM) or brushless direct current motor (BLDC) and can be used to quickly evaluate ST microcontrollers, to complete ST application platforms, and to save time when developing motor control algorithms to be run on ST microcontrollers. It is written in the C language, and implements the core motor control algorithms, as well as sensor reading/decoding algorithms and sensor-less algorithms for rotor position reconstruction.

The library can be customized to suit user application parameters (motor, sensors, power stage, control stage, pinout assignment) and provides a ready-to-use application programming interface (API). A PC graphical user interface (GUI), the ST motor control workbench, allows complete and easy customization of the library. Thanks to this, the user can run a motor in a very short time.

The generated projects include a UART interface that allows convenient real-time fine-tuning of the motor control subsystem with a remote control tool, the STM32 motor control pilot.

The STM32 motor control SDK is delivered as an expansion pack for the STM32CubeMX tool, and the 6-step library is based on the STM32 Cube Firmware libraries.

The list of supported STM32 microcontrollers is provided in the release note delivered with the SDK.

1 Acronyms and abbreviations

Table 1. Acronyms and abbreviations

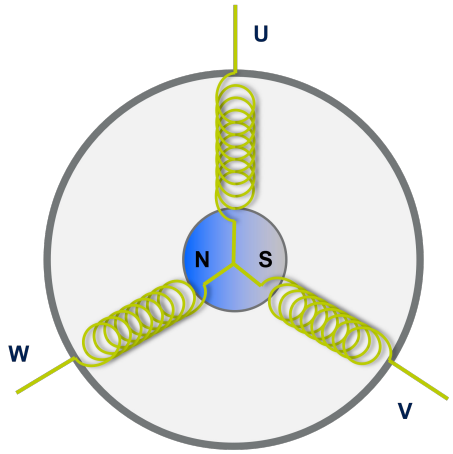
Acronym	Description
MCSDK	Motor control software development kit
HW	Hardware
IDE	Integrated development environment
MCU	Microcontroller unit
GPIO	General-purpose input/output
ADC	Analog to digital converter
VM	Voltage mode
SL	Sensor-less
HS	Hall sensors
BEMF	Back electro motive force
FW	Firmware
LF	Low frequency
MF	Medium frequency
HF	High frequency
ZC	Zero-crossing
API	Application programming interface
DMA	Direct memory access
DPP	Digit per control period
GUI	Graphical user interface
HAL	Hardware abstraction layer
ISR	Interrupt service routine
LL	Low layers
MC	Motor control
NTC	Negative temperature coefficient
NVIC	Nested vector interrupts controller
OCP	Overcurrent protection
PID	Proportional-integral-derivative (controller)
PMSM	Permanent magnet synchronous motor
SDK	Software development kit
UI	User interface
MC WB	Motor control workbench
MC profiler	Motor control profiler

2 6-step firmware algorithms

2.1 Definitions

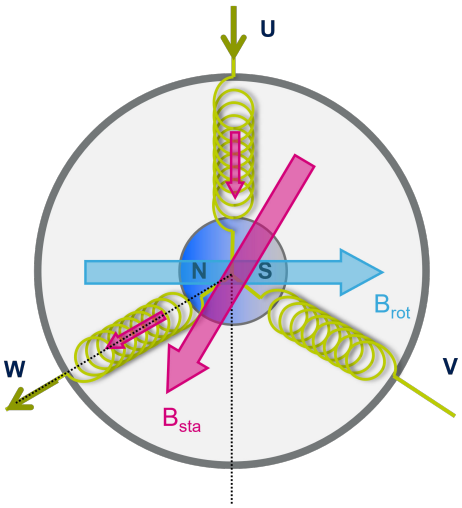
A brushless three-phase motor is composed as shown in Figure 1 by a fixed element, called stator, made of a set of three windings (that is, phases) connected together at one side and a moving element containing an internal permanent magnet, called rotor. The rotor may have several pole pairs regularly distributed around the stator.

Figure 1. Motor stator and rotor arrangement



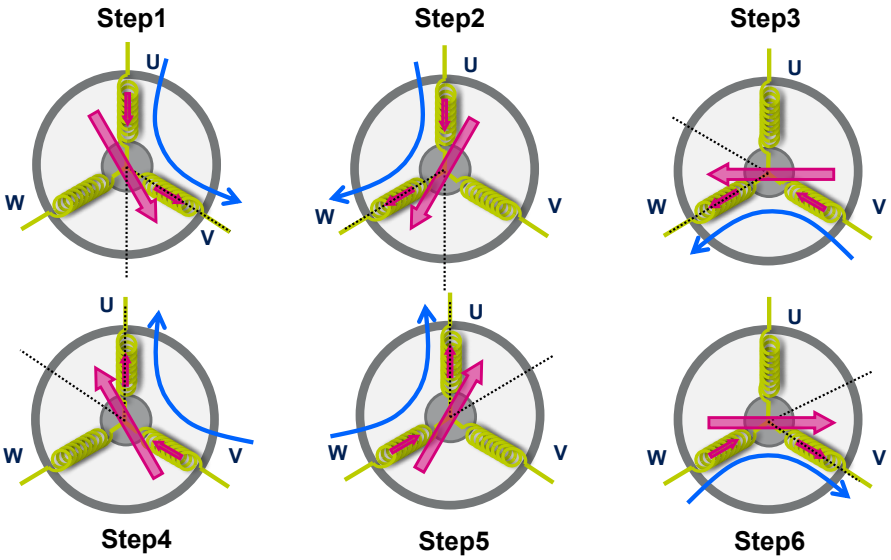
In 6-step driving, the electrical cycle is divided into six commutation steps. At each step, the bus voltage is connected to one of the three phase windings of the motor while ground is connected to a second winding, forcing a current flowing through these two windings and generating a stator magnetic field as shown in Figure 2. The third winding remains floating.

Figure 2. Motor stator and rotor magnetic fields



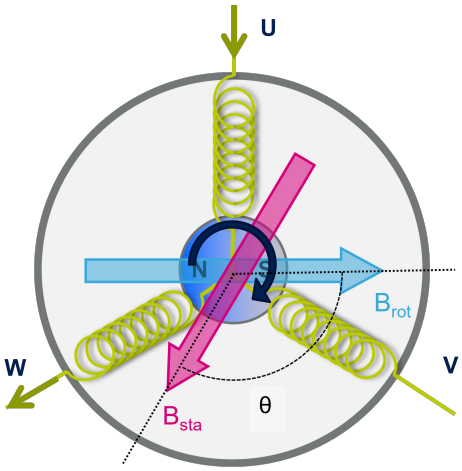
The orientation of the stator magnetic field is changed, energizing the windings in the sequence shown in Figure 3.

Figure 3. **Motor stator magnetic fields discrete positions**



Since the rotor has a permanent magnetic field, the rotating stator magnetic field creates a torque that moves the rotor. The maximum torque is obtained when the electrical angle between the rotor and the stator is 90°. The orientation of the stator magnetic field is changed thanks to the 6-step commutation keeping the motor spinning, as explained in Figure 4.

Figure 4. **Motor torque**

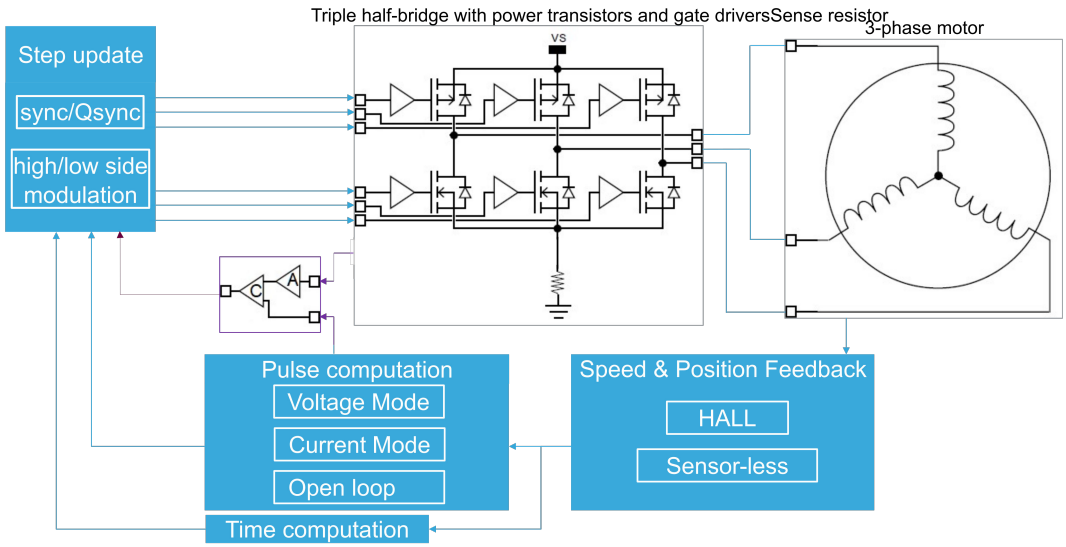


2.2 Algorithm overview

The 6-step firmware reveals the position of the motor rotor every 60 electrical degrees (in sensor-less or sensored mode). Based on this information, it computes the time for the next step commutation and calculates the duty cycles for the PWM signals that drive the output power transistors. These transistors control the motor phase voltages allowing to reach a target speed.

The 6-step firmware can be viewed as a set of components, each with a different task.

Figure 5. Basic 6-step algorithm structure



The following list describes the components shown in Figure 5:

- speed and position feedback component: BEMF sensing is used with the sensor-less driving mode while Hall sensors are exploited with the sensored driving mode
- pulse computation component (voltage mode – VM or current mode - CM) is used. A proportional-integral (PI) controller algorithm is employed for the speed loop control
- time computation component: it manages the step change and applies the configuration of the timers set by the step update component
- step update component: it updates the configuration of the timers, hence the proper energization of the phases, according to the selected driving mode (high side/low side MOSFET modulation, synchronous/ quasi-synchronous rectification).

2.2.1 Open loop mode

In open loop mode, the input duty cycle is directly applied to the motor phases. The reached speed consequently depends on the mechanical load of the motor.

2.2.2 Speed loop mode - Voltage driving

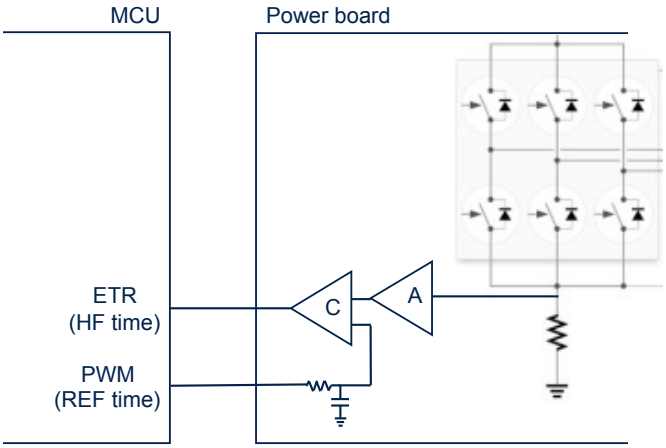
The motor speed is controlled by directly varying the duty cycle of the pulse width modulated voltages applied to the motor phases.

2.2.3 Speed loop mode - Current driving

The motor speed is controlled by limiting the peak of the current flowing through the active phases.

This driving mode exploits the presence of an amplifier A and a comparator C. The current is controlled by setting the duty cycle of a PWM generated by a timer (REF timer) used as the reference voltage of the comparator C. The output of the comparator triggers the switch-off of the PWMs connected to the motor phases when the amplified sense resistor voltage is greater than the reference voltage.

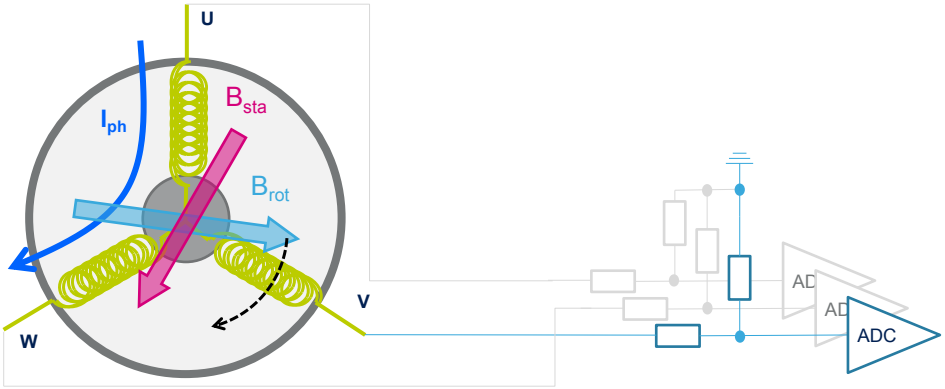
Figure 6. Current driving mode structure



2.2.4 Sensor-less algorithm

In the sensor-less mode, the position of the rotor is obtained by detecting the zero-crossing of the BEMF sensed at the floating phase. This is commonly done using an ADC as shown in Figure 7. In particular, when the magnetic field of the rotor crosses the high impedance phase, the corresponding BEMF voltage changes its sign (zero-crossing). The BEMF voltage can be scaled at the ADC input, thanks to a resistor network.

Figure 7. Motor with sensor-less circuit



2.2.5 Hall sensors algorithm

In this driving mode, the positioning of the rotor is obtained by reading the digital signals coming from the Hall sensors (connected to three GPIOs).

During the alignment time, the position of the rotor is acquired and the motor windings are energized accordingly. When a sensors commutation is detected, the new status is acquired. At the same time, the step is changed and the PWMs updated accordingly.

3 STM32 MC firmware

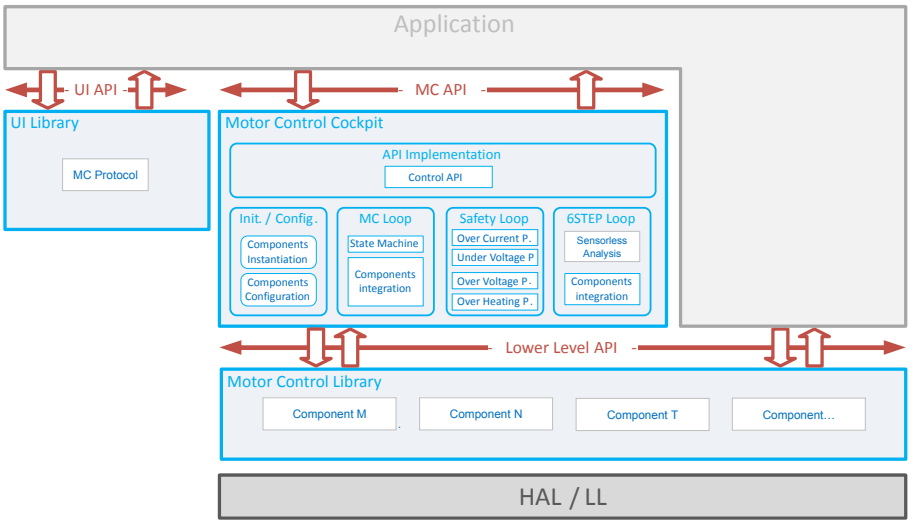
The STM32 MC firmware is the heart of the SDK. It provides all the software components needed to control 3-phase motors using the 6-step strategy and integrates these components into an MC subsystem. It offers a versatile set of interfaces that custom applications can use to actually drive motors according to their needs.

Figure 8 shows the architecture of the STM32 MC firmware.

The firmware consists of the three following functional sets:

- The 6-step library contains software components that implement the motor control features;
- The UI library contains software components that deal with the communication between the motor control firmware subsystem and either the user or an offloaded application;
- The motor control cockpit integrates all these software components into a motor control firmware subsystem and implements the regulation loops.

Figure 8. STM32 motor control firmware architecture



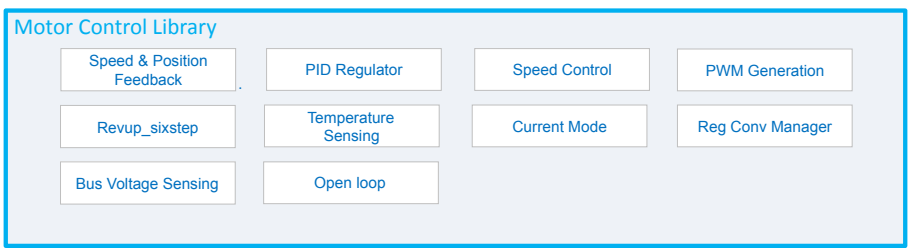
3.1 6-step MC library

The 6-step MC library is a collection of software components. Each component implements a feature involved in MC such as, for instance, the speed and position sensing, PID regulator, or motor control algorithms.

For some features, the library provides several components, each containing a different implementation. This allows for supporting various hardware configurations in an efficient way. The components to use are then chosen depending on the characteristics of the user's application and are integrated into a motor control firmware subsystem.

Figure 9 summarizes the features provided by the 6-step library as components. A list of most of the components in the PMSM FOC library and their specificities is described inMotor control firmware components.

Figure 9. 6-step MC library features delivered as components



3.2 User interface library

The user interface library or UI library contains software components that deal with the communication between the MC firmware subsystem and the outside world using a serial port. This library is used to allow the STM32 MC WB to connect to the application and control it with its monitor feature.

3.3 Motor control cockpit integration

The motor control cockpit integrates the software components into an MC firmware subsystem and implements the regulation loops. It instantiates, configures, and interfaces the firmware components selected in the 6-step library and in the user interface library for the user's application. The code of the MC cockpit is generated by the STM32Cube according to the characteristics of the application. Thanks to this generation the code of the cockpit only contains what is needed and is thus easily readable.

4 6-step motor control firmware subsystem

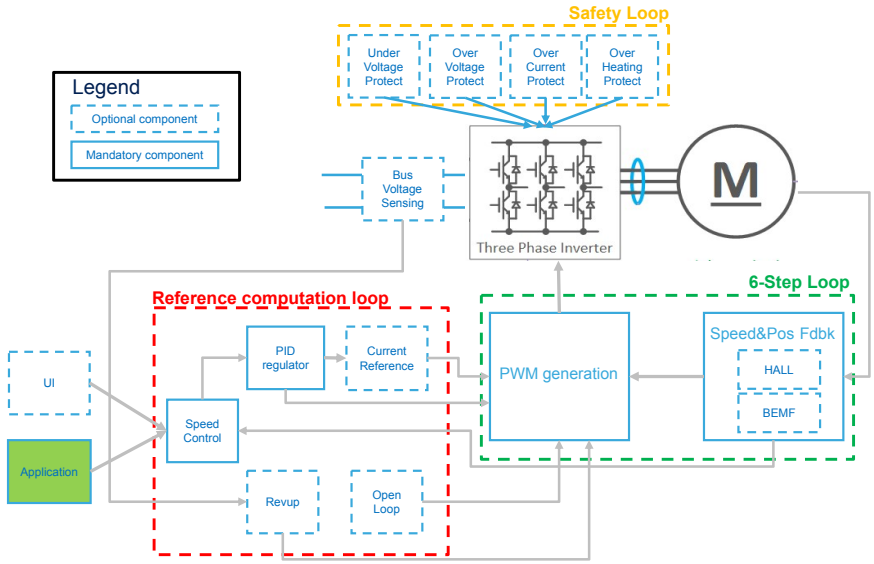
The motor control firmware subsystem is the firmware library that results from the configuration and generation of a firmware project with the STM32 MC WB / STM32CubeMX pair. Users then build their final application on top of this subsystem, adding their own code that uses one of the provided APIs (see below).

Figure 10 provides an overview of this subsystem showing optional and mandatory functional blocks as well as how they interact with one another. Note that only the most important blocks and interactions are shown for the sake of clarity. This figure highlights three sets of functional blocks.

The 6-step loop is the core of the algorithm. Its aim is to update the timer registers determining the step change and to apply the duty cycle (calculated in the reference loop) to the transistors driving the motor's phases.

The 6-Step loop is executed every 360 electrical degrees, whose period depends on the speed and the motor pole pairs number.

Figure 10. Motor control subsystem overview



The purpose of the reference computation loop, as its name suggests, is to compute the duty cycle references based on targets coming from the application. Usually, the application provides a reference expressed in a way that matches its needs: a speed reference or ramp. The reference computation loop first converts the application target into a phase voltage duty cycle or reference current duty cycle which is then used to generate motor winding PWMs and optionally a current reference PWM.

This process is in force when the motor control subsystem is executing in closed loop mode.

However, this is not the only operating mode. Indeed, depending on the chosen Speed and Position Feedback technology, a rev/up phase may be needed that takes over that process until the rotor Position estimation is judged reliable. This is the purpose of the Rev-up Control component.

In addition, some applications may require that the motor control stays in open loop. This case is handled by the Open Loop Control component that is executed in lieu of the normal regulation process

All these cases fall into the basket of the Reference computation loop that is executed at a medium rate, typically on the SysTick interrupt.

The last set of functional blocks is the safety loop. This set is called a loop because it consists in functions that gets executed on a periodic basis. They all deal with features that aim at reacting to conditions that may endanger the system from a hardware point of view: Over- and Undervoltage protection, Over Heating protection and Overcurrent protection. In the case of overcurrent protection, the STM32 MC firmware is designed to exploit hardware mechanisms implemented in the STM32 MCUs such as the Timer Break input that accelerate the system reaction to an overcurrent situation.

The safety loop is executed at the same rate as the Reference computation loop – that is at a medium rate, usually with the SysTick interrupt.

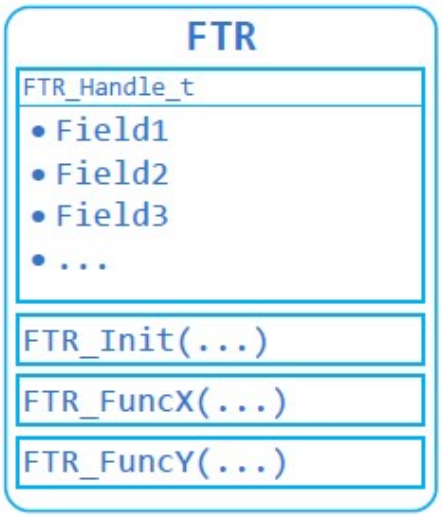
4.1 Motor control firmware components

Most of the motor control firmware is organized as a set of software components. A component in a self-contained software unit that defines:

- A structure with the data needed to fulfill the feature the component is designed to provide
- A set of functions operating on instances of the structure and that implement that feature

The data placed in the structure of a component are the parameters that characterize this component and that tune its behavior. They fully describe the state of the component. In the motor control firmware, a type is defined to hold these data together. Variables of this type are used as handles on instances of the component.

Figure 11. Component with its handle and its function



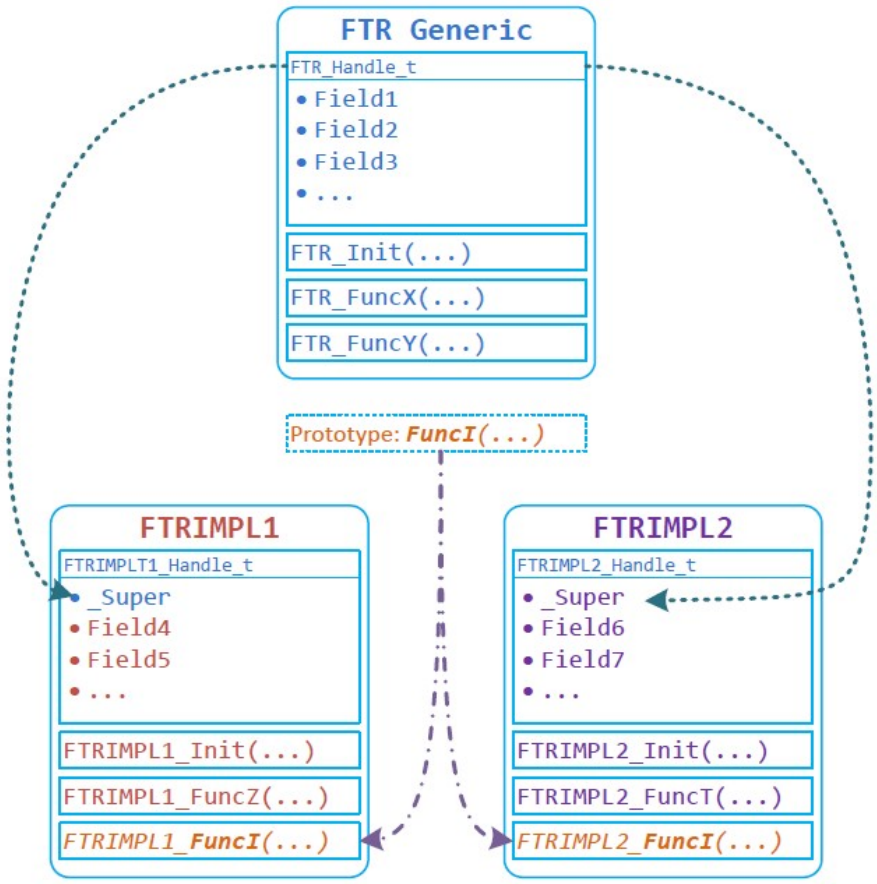
The way this principle is used is very straightforward. Where a feature is needed, the component that matches this feature is selected and a variable of the structure's type is defined. The variable is then initialized with the feature's parameters as defined for the application. This is done when the motor control firmware subsystem is initialized by the MC_boot() function.

Finally, during the operation of the motor control firmware subsystem, the functions defined for the component are called where and when needed to benefit from the feature it provides. These functions provide the component's feature. To perform their task, they expect a pointer on a handle of the component's structure as the first argument so that they have access to the state and the settings of the instance of the component they work for.

The notion of component makes it easy to offer several implementations of a given feature. For such cases, a generic component is defined for the feature. Its handle contains the data that are common to the feature whatever its actual implementation is, and its functions operate on these data. In addition, the prototypes of the functions that each component implementing the feature needs to provide are defined. These functions are the interface of the components.

Then, these implementing components reuse and extend the handle of the generic component into their own and implement the functions needed to fulfill the feature. This allows for a simplified integration and an easy replacement of an implementation by another.

Figure 12. **A component with its handle and its function**



An example of this situation is the set of speed and position feedback components. A generic component is defined, represented by the SpeednPosFdbk_Handle_t handle structure, defined in the speed_pos_fdbk.h file. The handle of this generic component only contains the data that are purely related to the speed and the position of the motor's rotor such as the current mechanical and electrical angles, the conversion factor between them and the limits within which the feature is to be used. And its functions are only about setting and getting these data. Two actual implementations are provided, one that uses Hall Effect sensors and one that implements the feature using the BEMF sensing based algorithm. Each of these two implementations define their own handle that extends SpeednPosFdbk_Handle_t and each define the interface functions based on the same prototypes.

The following sections present an overview of all the components offered by the STM32 MC SDK. For a complete description, refer to the STM32 MC firmware reference manual.

4.1.1 PWM generation component

4.1.1.1 Features overview

The PWM component is responsible for:

- applying the desired voltage on the three phases
- managing the triggering of the ADC (sensor-less mode only)

A timer peripheral is used to generate the PWM signals and to trigger the measurement of the ADC at the right time. This mechanism is described in more detail in Speed and position feedback components.

Concretely, the main task of this component is to enable and disable the outputs and apply the voltages to the proper phases in accordance with the 6-step sequence and the driving modes described here below.

This task begins when the motor is started and ends when it is stopped. In addition, these components also play a role in other matters such as the boot capacitor charging which requires switching the low-side transistors on, and the overcurrent protection. Each PWM handles the timer interrupts that are relevant to its operation. It expects these interrupts to be configured with a given priority level and it defines its own functions to handle them.

The application shall not tamper with the priorities of these interrupts or with the order in which they are served in the interrupt handler.

4.1.1.2 *Available implementations and specificities*

The 6-step library provides all the components needed to support 6-pwm and 3-pwm (plus 3 enable GPIOs) devices. The selection of the component that matches device topology actually in use by the application is performed through the STM32 MC WB.

These implementations are built on a generic PWM generation component that they extend and that provides the functions and data that are common to all of them. This base component cannot be used as is since it does not provide a complete implementation of the features. Rather, its handle structure (PWMC_Handle_t) is reused by all the PWM generation specific implementations.

The functions, that the generic PWM generation component provides, form the API of the PWM generation feature. Calling those results in calling functions of the component that actually implement the feature.

Besides the standard driving mode (one phase with PWM applied to the high side MOSFET, one phase floating and one phase to ground), three additional driving modes are implemented:

- Low side MOSFET modulation
- Mid-step alignment
- Quasi-synchronous rectification

All these three modes are explained in detail in the following sections.

4.1.1.3 *Fast-demagnetization driving mode*

When the microcontroller switches from one step to the next, the non-excited winding needs a certain demagnetization time. During this time, the current in the winding continues in the same direction but decreases to zero and the BEMF cannot be sense because it is masked by the demagnetization current.

The standard driving mode applies the PWM modulation to the high side MOSFETs and the other non-floating phase to ground. This driving mode applies the PWM modulation to the low side MOSFETs and connects the other non-floating phase to the bus voltage. The advantage of doing that is to accelerate the demagnetization of the floating phase and reduce the time during which the BEMF cannot be sensed, in particular during the steps when the demagnetizing current is flowing from the bridge to the motor phases.

Table 2. Fast demagnetization driving mode table

Step	MOSFET On (HighSide + LowSide)	Demagnetization current	PWM applied with FAST_DEMA=1
Step1	PhaseU + PhaseV	PhaseW (from bridge to motor)	PhaseV (LS)
Step2	PhaseU + PhaseW	PhaseV (from motor to bridge)	PhaseU (HS)
Step3	PhaseV + PhaseW	PhaseU (from bridge to motor)	PhaseW (LS)
Step4	PhaseV + PhaseU	PhaseW (from motor to bridge)	PhaseV (HS)
Step5	PhaseW + PhaseU	PhaseV (from bridge to motor)	PhaseU (LS)
Step6	PhaseW + PhaseV	PhaseU (from motor to bridge)	PhaseW (HS)

In the advanced configuration tab of the motor pilot, the user can decide to change the modulation state of the MOSFETs independently for every step (see Figure 13).

Figure 13. **Low-side modulation configuration in motor pilot**

This feature is available only with a hardware that supports 6-pwm inputs.

4.1.1.4 *Mid-step alignment driving mode*

In sensor-less driving mode the motor is first aligned at a predefined and known rotor position before starting the acceleration phase. In order to minimize the mechanical vibrations and make the startup more smoothly, the alignment position is chosen to be in the middle between adjacent steps. Therefore, instead of the standard 6-step phase polarization table, the following one is used, based on the initial step of the sequence and spinning direction. None of the three phases are floating.

Table 3. **Mid-alignment driving mode table**

Initial step	Direction	PWM applied	Ground
Step1	Clockwise	PhaseU + Phase W	Phase V
	Counter-clockwise	PhaseU	PhaseV + PhaseW
Step2	Clockwise	PhaseU	PhaseV + PhaseW
	Counter-clockwise	PhaseU + PhaseV	PhaseW
Step3	Clockwise	PhaseU + PhaseV	PhaseW
	Counter-clockwise	PhaseV	PhaseU + Phase W
Step4	Clockwise	PhaseV	PhaseU + Phase W
	Counter-clockwise	PhaseV + PhaseW	PhaseU
Step5	Clockwise	PhaseV + PhaseW	PhaseU
	Counter-clockwise	PhaseW	PhaseU + PhaseV
Step6	Clockwise	PhaseW	PhaseU + PhaseV
	Counter-clockwise	PhaseU + Phase W	PhaseV

4.1.1.5 *Quasi-synchronous driving mode*

By default, the power stage is driven by the algorithm in fast decay mode.

At the start of the off-time, both the power MOS of the energized phases are switched off and the current recirculates through the two opposite freewheeling diodes. The current decays with a high di/dt since the voltage across the coil is the power supply voltage. After the deadtime, the low-side MOS and the high-side MOS in parallel with the conducting diode are turned on in synchronous rectification mode (see Figure 14).

In applications where the motor current is low, the load current may decay completely to zero and rise in the opposite direction.

To avoid this, the quasi-synchrounous option may be enabled: the lower power MOS is not turned on in synchronous rectification mode preventing the current to reverse. This operation is called “Quasi-synchronous rectification mode” (see Figure 15).

The quasi-synchrounous option is available with boards with 6-pwm control devices only.

The quasi-synchrounous and the low side modulation driving modes are incompatible one another. When both enabled, only low-side modulation is considered.

Figure 14. Synchronous rectification

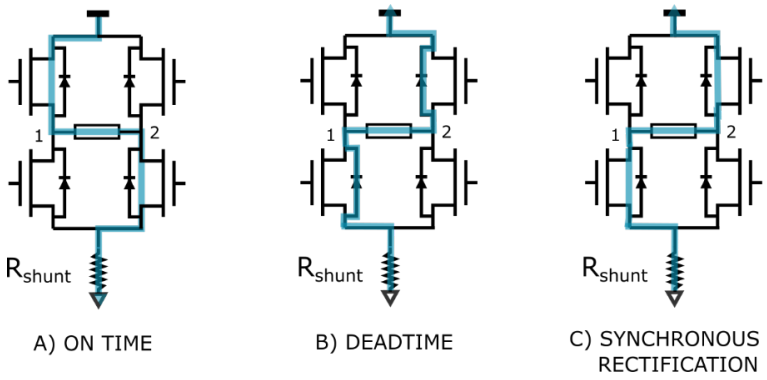
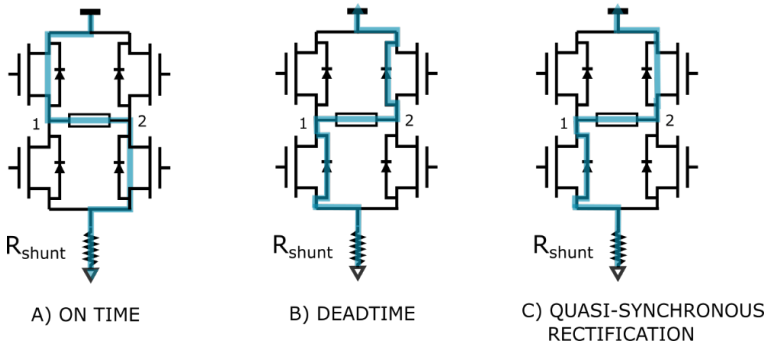


Figure 15. Quasi-synchronous rectification



4.1.2 Speed and position feedback components

These components provide the speed and the angular position of the rotor of a motor (both electrical and mechanical). While angular position is crucial to perform properly the step change at the right time, the rotor speed measurement is needed to close the speed loop.

Two implementations of the Speed and Position Feedback feature are provided by the STM32 MC firmware. One uses sensors embedded in some motors (Hall sensors) and the other one provides an estimation of the speed and the position of the rotor based on the sensing of the Back-EMF of the motor.

These two implementations are built on a generic Speed and Position Feedback component – named the Speed and Position Feedback component – which they extend and which provide the data that are common to all of them. In addition to that, also the sensing of the Back-EMF is deployed in several components, each of them dedicated to a specific microcontroller family.

Table 4. Available speed and position feedback components

Component	Description
Hall speed and position feedback	This component uses the output of two Hall effect sensors to provide a measure of the speed and the position of the rotor of the motor
BEMF speed and position feedback – G4xx micro family	This component uses the sensing of the Back-EMF at the motor phases to provide an estimation of the position of the rotor of the motor and then calculate its speed
BEMF speed and position feedback – F0xx micro family	
BEMF speed and position feedback – F3xx micro family	

Component	Description
BEMF speed and position feedback – F3xx micro family	This component uses the sensing of the Back-EMF at the motor phases to provide an estimation of the position of the rotor of the motor and then calculate its speed
BEMF speed and position feedback – F3xx micro family	
BEMF speed and position feedback – F3xx micro family	

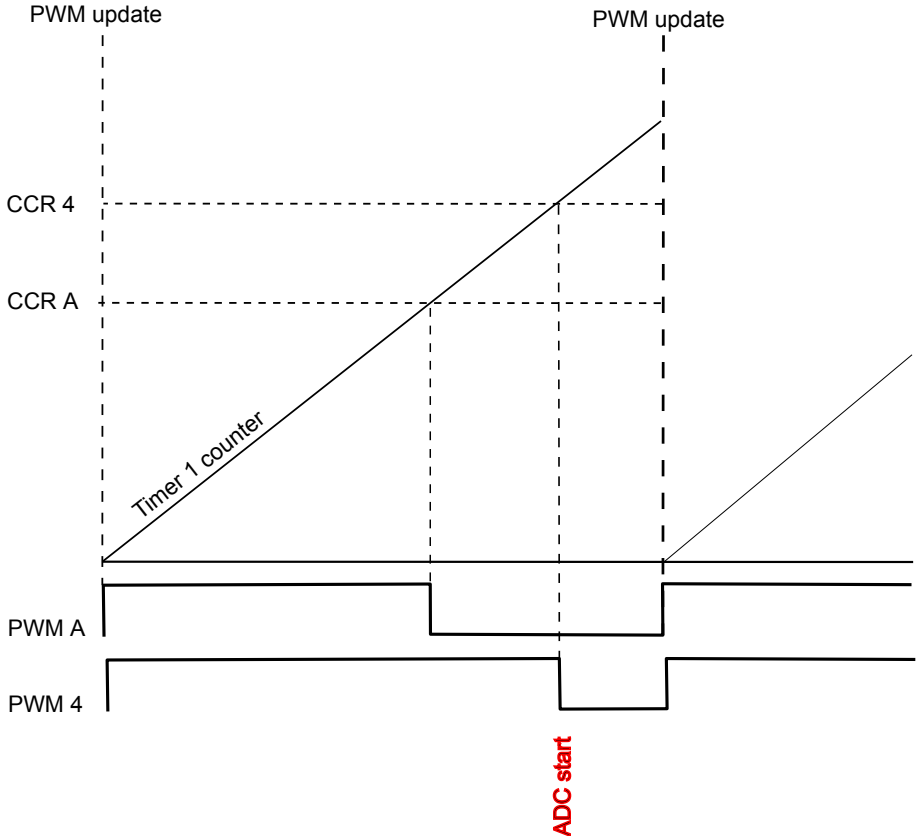
An additional implementation is also present in the firmware: the Virtual Speed and Position Feedback component. This component is only used during the rev-up phase of the motor, while Back_EMF based implementations are used for closed loop mode.

4.1.2.1 *Back-EMF sensing and zero-crossing point detection*

Back-EMF sensing components take full ownership of the ADC peripherals they use. The application can use the ADC channels left free by the motor control subsystem, but it should not interface these channels directly. The application shall use the functions of the Regular Conversion Manager (RCM) component. Refer to the reference documentation of the MC SDK for a complete description.

Figure 16 shows the synchronization strategy between the TIM1 PWM output and the ADC. Normally the A/D converter peripheral is configured so that it is triggered by the falling edge of TIM1_CH4 during the decay time (off-time of the PWM).

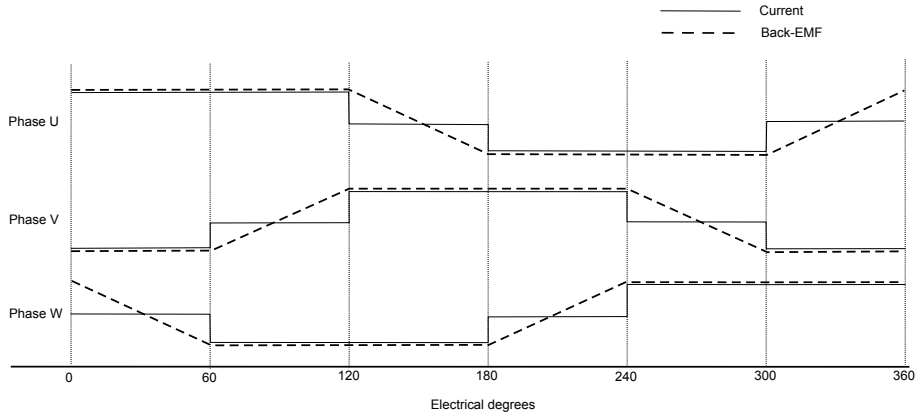
Figure 16. PWM and ADC synchronization



The Back-EMF waveform of a brushless motor changes along with the rotor position and speed and is in a trapezoidal shape.

Figure 17 shows the waveform of the current and Back-EMF for one electrification period where the solid line denotes the current (ripples are ignored for the sake of simplicity), the dashed line represents the back electromotive force, and the horizontal coordinate represents the electric perspective of motor rotation.

Figure 17. BLDC current and Back-EMF waveforms



The middle of every two phase-switching points corresponds to one point whose back electromotive force polarity is changed, the zero-crossing-point.

In order to reduce the number of interrupts and the MCU load, the Back-EMF sensing component takes advantage of the analog watchdog feature of the ADC to perform the comparison between the converted BEMF signal and the zero-crossing threshold. Once the condition for zero crossing event is satisfied, the analog watchdog interrupt is triggered. Once the zero-crossing point is identified, the phase-switching moment is set after an electrical delay of 30°.

Two different strategies are available for the identification of the zero crossing point:

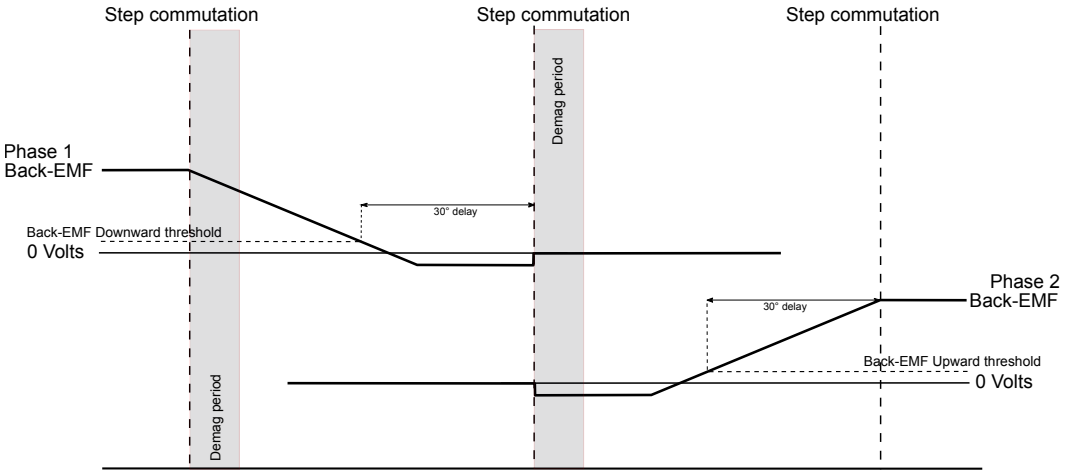
- Back-EMF sensing during the off-time
- Back-EMF sensing during the on-time.

4.1.2.2 *Back-EMF sensing during the off-time*

This kind of detection can be used with both the driving modes, current and voltage modes. The principle is the one described in Figure 16 where the ADC conversion is triggered during the off-time of the PWM. At the beginning of every 6-step commutation the ADC channel corresponding to the floating phase is selected. After a masking period corresponding to the demagnetization period of the floating phase, during which the Back-EMF reading is not reliable, the converted values are compared with a threshold to determine the Back-EMF polarity change. Once the zero-crossing point is detected a delay of 30° electrical degrees, based on the current measured speed, is programmed on a timer, whose update triggers, finally, the step commutation.

Note: In order to avoid damaging the devices, as shown in Figure 18, the minimum of the Back-EMF reading is limited by the protection diodes present on the board.

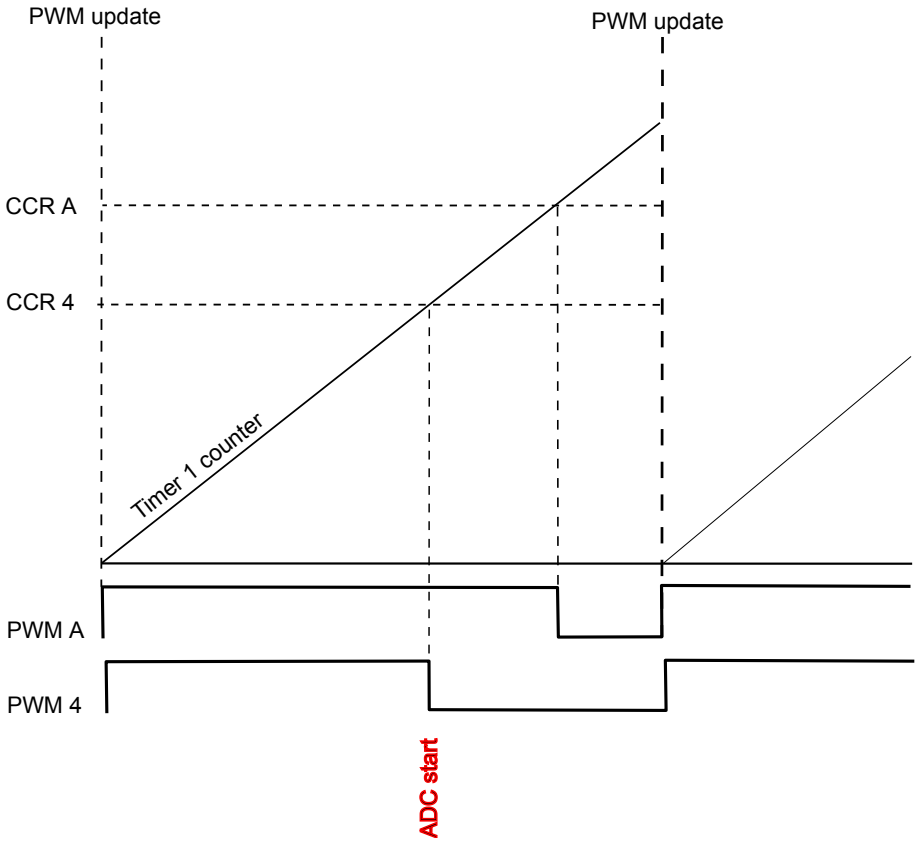
Figure 18. **Back-EMF sensing and step commutation**



4.1.2.3 *Back-EMF sensing during the on-time*

This kind of detection can be used with the voltage driving mode only where the duty cycle is fixed and doesn't change along the step. The principle is the one described in Figure 19 where the ADC conversion is triggered during the on-time of the PWM. At the beginning of every 6-step commutation the ADC channel corresponding to the floating phase is selected. After a masking period corresponding to the demagnetization period of the floating phase, during which the Back-EMF reading is not reliable, thanks to a resistor network that allows to estimate the voltage of the centre-tap point (the common point of the three motor phases), the converted values are compared with a threshold that is ideally half of the bus voltage. The threshold is set considering a resistor dividing network that rescales the phase voltage to the ADC input range. Once the crossing point based on the current measured speed, is programmed on a timer, whose update triggers, finally, the step commutation.

Figure 19. **Back-EMF during on-time**



While the Back-EMF sensing during the off-time is normally performed in sensor-less mode, the user is allowed to enable or disable the sensing during the on-time as long as the voltage drive mode is selected.

When both the sensing techniques are used, the algorithm automatically switches between one and the other based on the PWM duty cycle allowing to reach close to 100% duty cycle.

4.1.2.4 *Sensor-less configuration parameters*

In the following section all the sensor-less configuration parameters that can be found in the workbench and/or in the motor pilot GUI are explained.

Average speed FIFO depth

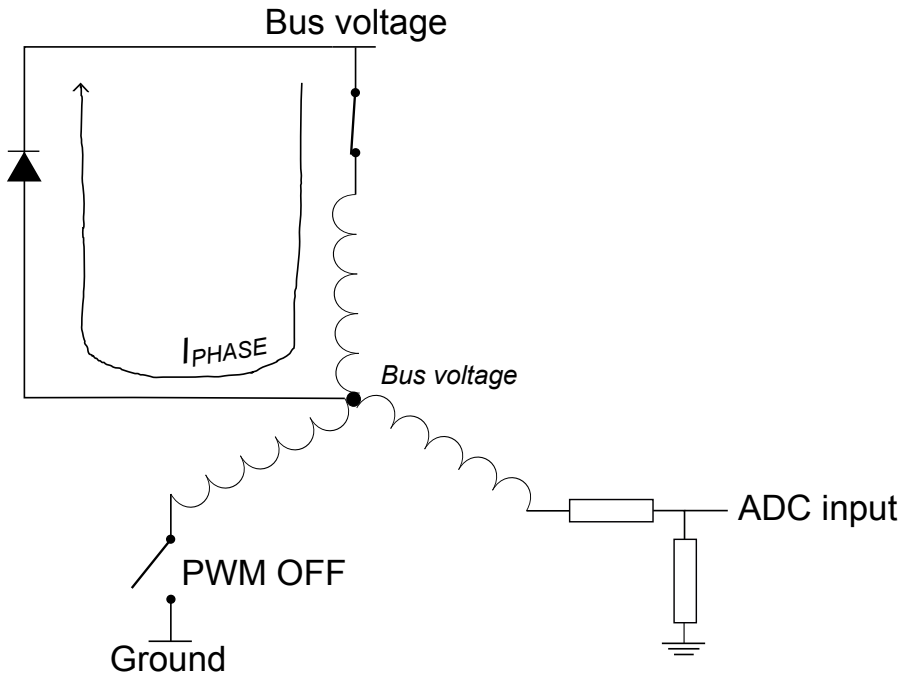
It defines the number of samples used to calculate the average speed

Threshold with high-side recirculation

This threshold is used to detect BEMF zero-crossing during PWM off-sensing and PWM applied on the low-side MOSFETs (fast demagnetization). In this condition, during PWM off-time, the load current recirculates in two high-side MOSFETs and the floating phase is pulled up to the bus voltage. Ideally, when BEMF is zero, the floating phase is a replica of the windings' center point whose voltage equals the bus voltage (see Figure 20).

This parameter can be set in the “Speed sensing config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

Figure 20. PWM off - high-side recirculation

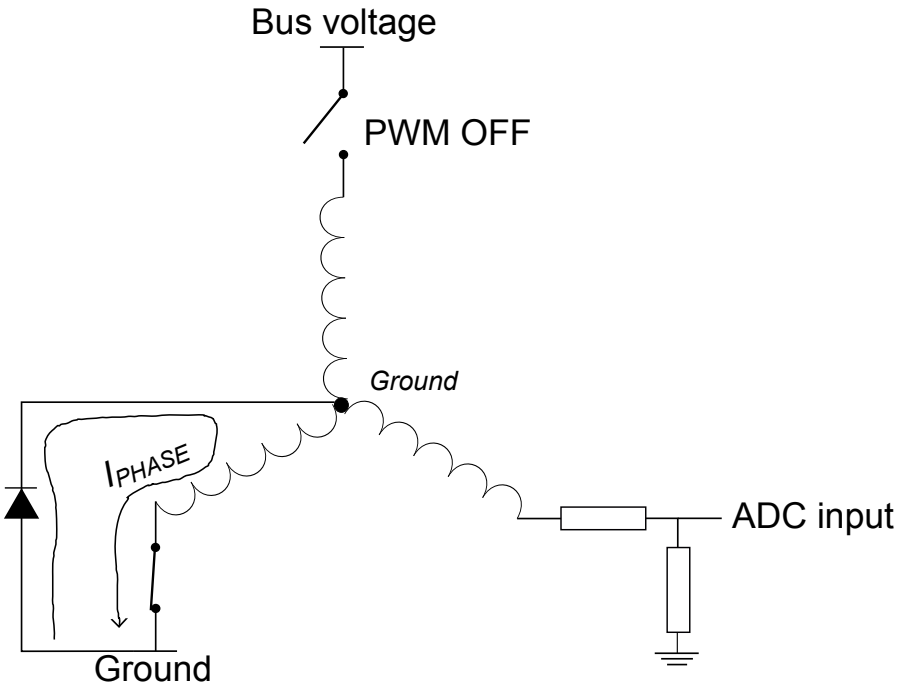


Threshold with low-side recirculation

This threshold is used to detect BEMF zero-crossing during PWM off-sensing and PWM applied on the high-side MOSFETs (standard driving). In this condition, during PWM off-time, the load current recirculates in two low-side MOSFETs and the floating phase is pulled down to the ground. Ideally, when BEMF is zero, the floating phase is a replica of the windings' center point whose voltage is zero (see Figure 21).

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

Figure 21. **PWM off - low-side recirculation**



PWM off-sampling point

It defines the sampling point of the ADC when sensing the BEMF during the PWM off-time (CCR4 point of Figure 16). It is defined as a percentage of the PWM cycle duration.

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

Threshold with PWM on

This threshold is used to detect BEMF zero-crossing during PWM on-sensing. In this condition, the load current flows through one high-side MOSFET, the two energized phases of the motor, and one low-side MOSFET. Ideally, when BEMF is zero, the floating phase is a replica of the windings' center point whose voltage is in the middle between the bus voltage and ground.

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

PWM on-sampling point

It defines the sampling point of the ADC when sensing the BEMF during the PWM on-time (CCR4 point of Figure 19). It is defined as a percentage of the PWM cycle duration.

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

On-sensing enabling

It defines the PWM duty cycle threshold where the algorithm automatically switches from BEMF off-sensing to BEMF on-sensing.

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

On-sensing hysteresis

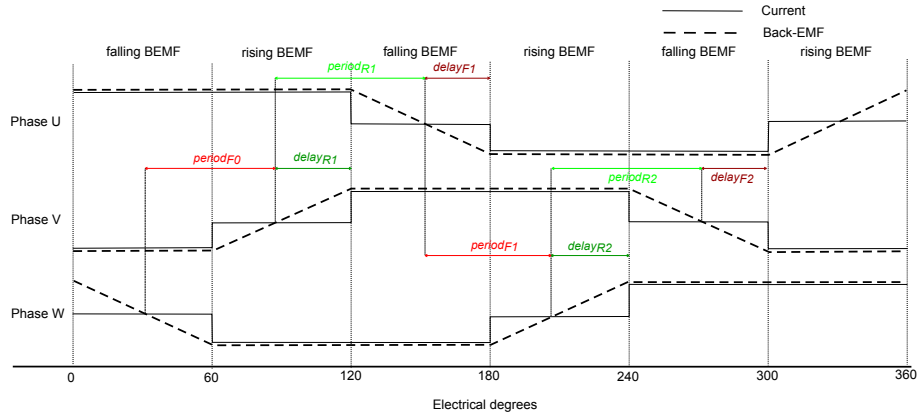
It defines the PWM duty cycle hysteresis used by the algorithm to calculate the threshold to return to BEMF off-sensing when the PWM duty cycle is decreasing.

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

Falling BEMF zero-crossing to step change delay

This parameter ($ZCD_{falling}$) is used to calculate the delay between the zero-crossing event in steps with falling BEMF and the following step change. It is nominally 30 electrical degrees (zero-crossing point in the middle of the step) but needs to be reduced to maintain rotor synchronism at high speed or when the number of PWM cycles in a step is few because, due to the sample discretization, the zero-crossing event is detected later than the actual position.

Figure 22. Delay computation between zero-crossing and step change



Referring to Figure 22 for instance:

$$delay_{F1} = \frac{ZCD_{falling}}{60} \times period_{R1}$$

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

Rising BEMF zero-crossing to step change delay

This parameter (ZCD_{rising}) is used to calculate the delay between the zero-crossing event in steps with rising BEMF and the following step change. It is nominally 30 electrical degrees (zero-crossing point in the middle of the step) but needs to be reduced to maintain rotor synchronism at high speed or when the number of PWM cycles in a step is few because, due to the sample discretization, the zero-crossing event is detected later than the actual position.

Referring to Figure 22 for instance:

$$delay_{R1} = \frac{ZCD_{rising}}{60} \times period_{F0}$$

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

Delay computation step choice

This parameter allows a 6-step computation delay after a BEMF zero-crossing detection event based on the previous step or the one before (step time – 1 or -2). In case of 2 step time, the computation delay is performed on the same kind of rising of the falling BEMF signal.

Referring to Figure 22 for instance:

$$step\ time - 1 \rightarrow delay_{F2} = \frac{ZCD_{falling}}{60} \times period_{R2}$$

$$step\ time - 2 \rightarrow delay_{F2} = \frac{ZCD_{falling}}{60} \times period_{F1}$$

This parameter can be set in the “Speed sensing Config.” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

AWD filter

Number of zero-crossing signals that trigger a zero-crossing event. It changes the analog watchdog filter configuration, and it applies to G4 microcontrollers only.

This parameter can be changed on-the-fly in the advanced configuration section of the motor pilot GUI.

Rev-up demagnetization time

Duration of the demagnetization time in terms of the percentage of the step duration that is calculated during the motor acceleration procedure.

BEMF is not sensed and the zero-crossing signal is ignored during this period.

This parameter can be set in the “Drive settings” section of the workbench.

Motor running demagnetization time

Duration of the demagnetization time in terms of the percentage of the step duration that is calculated after the switch-over phase (synchronized rotor).

BEMF is not sensed and the zero-crossing signal is ignored during this period.

This parameter can be set in the “Drive settings” section of the workbench.

Demagnetization speed threshold

Demagnetization duration after each step change is set to a minimum (defined as “Minimum demagnetization time”) when the motor speed is higher than the demagnetization speed threshold.

This parameter can be set in the “Drive settings” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

Minimum demagnetization time

When motor speed is higher than the “Demagnetization speed threshold” the minimum demagnetization time is applied. It is defined as PWM cycles after step change during which the BEMF is not sensed and the zero-crossing signal is ignored.

This parameter can be set in the “Drive settings” section of the workbench and changed on-the-fly in the advanced configuration section of the motor pilot GUI.

4.1.3 Bus voltage sensing components

The STM32 MC firmware provides components to report the value of the bus voltage. A measurement of the bus voltage is, of course, needed for features like the under/overvoltage protection.

Two implementations of a bus voltage sensing component are available. One that basically uses an ADC channel and two large resistors to measure the voltage (the Resistor Divider Bus Voltage Sensor) and another one that actually does not measure anything and only reports a configured value (the Virtual Bus Voltage Sensor).

For its measurements, the resistor divider bus voltage sensor implementation uses a channel of the ADC configured for the current feedback of motor 1, thanks to the regular conversion API. Refer to Section 4.2.4for more details.

4.1.4 Temperature measurement component

The STM32 MC firmware provides one component to report the motor control subsystem’s temperature. This component – the NTC Temperature Sensor – acts both as a real temperature sensor that uses an ADC channel to measure the temperature from a probe and as a virtual temperature sensor that basically reports a configured temperature value.

4.1.5 Drive regulation components

This section presents some of the drive regulation components that are delivered with the firmware. For more complete information on all these components, refer to the STM32 motor control reference manual.

PID

The PID component provides an implementation of a proportional–integral–derivative controller. This component is primarily used by the reference computation loop in the speed controller.

It comes in two flavors: a full PID using all three terms and a simpler one that only uses the Proportional and Integral terms. The motor control subsystem uses the latter one.

Rev-up

The rev-up component is responsible for starting the motor. Its task begins when the motor is started in open loop and ends when the current control loop can be closed.

Speed Control

This component serves two purposes:

1. It produces the duty cycle reference from speed reference submitted by the application. As such, it manages the ramps programmed by the application
2. It regulates this speed reference thanks to a PID component.

4.2 Motor control cockpit

The motor control cockpit plays a central role in a motor control subsystem; it configures and integrates the components selected for the MC application. And, in addition, it provides the implementation of the 6-step algorithm, reference computation and safety loops that match the designed application.

As such, it has to support a vast diversity of configurations that lead to a potentially huge and cumbersome source code. To avoid this issue and to provide a code that is as simple as possible, most of the cockpit's code is generated from the application's characteristics. Thanks to this generation, only these portions of the code that are needed by the MC system are present in the MC cockpit's source code.

Despite its changing nature, the code of the MC cockpit is organized in a sole and structured way.

4.2.1 Motor control cockpit main source files

This section lists the most important source files that make up the MC cockpit. Refer to the STM32 MC SDK reference documentation (delivered with the SDK) for a complete list of these files and their documentation.

motorcontrol.c, motorcontrol.h:

motorcontrol.c contains the function MX_MotorControl_Init() that initializes the MC subsystem. Its companion file, motorcontrol.h, exports the prototype of this function that is called by the generated main.c file.

mc_api.c, mc_api.h:

This pair of files contains the definition and implementation of the high level Application Programming Interface.

that the application can use to control the motors. See Section 5.1 for a description of this API. As such, mc_api.h is a file that applications need to include in order to use it.

mc_config.c, mc_config.h:

The mc_config.c file contains the structures and the data used to configure all the components used by the MC subsystem. The mc_config.h file exports the names of the structures for the application to use them as the Lower Level API as described in Section 5.2.

mc_parameters.c, mc_parameters.h:

The mc_parameters.c file contains structures and data that contain constant parameters for the MC subsystem. Its role is similar to the mc_config.c file except that its content can be fully placed in the flash memory since it is constant. The mc_parameters.h file exports the names the structures for the application to read them in the scope of the Lower Level API, as described in Section 5.2.

mc_types.h:

This file contains type definitions that are used across all the motor control subsystem. In addition, it includes all relevant STM32 Cube LL header files that are needed for the motor control subsystem.

Motor control subsystem parameters:

A series of files is generated that contain a lot of constants – defined as C preprocessor symbols – which are set to values that are meaningful to the MC subsystem and that are used in its code. Some of these files are dedicated to some STM32 family and are only present if the chosen MCU is part of this family. The list of these files can be seen here:

- drive_parameters.h
- pmsm_motor_parameters.h
- power_stage_parameters.h
- parameters_conversion.h
- parameters_conversion_g4xx.h

Interrupt handling:

The motor control subsystem provides handlers for the interrupts that it uses. These are defined in files that depend on the chosen STM32 family (stm32g4xx_mc_it.c for G4xx microcontroller family).

mc_tasks.c:

This file contains the implementation of the core of the MC cockpit. It contains the code of the loops described at the beginning of Section 4. More information on them is given below.

4.2.2 Tasks of the motor control subsystem

The code of each of the three loops that are at the heart of the MC firmware subsystem is distributed into "Task" functions.

The 6-step loop is implemented in the TSK_HighFrequencyTask() function. This function is executed at the PWM frequency rate (that is: once every PWM Period, see Section 4.1.1). The PWM Frequency is the highest frequency in the motor control subsystem. It is executed in the handler of the interrupt that occurs at each timer update.

The main task of this function is to translate the electrical angle into the 6-step sequence and update the PWM duty cycles that are to be programmed in the PWM Timer channels. Hence, the time this function has to operate is limited as it needs to complete before the next Timer update event, when new PWM duty cycles are taken into account. Failing to execute in this lapse of time results in the 6-step execution error.

The Reference computation loop is implemented in the function TSK_MediumFrequencyTaskM1(). This function needs to be invoked periodically at a frequency that is typically lower than that of the

TSK_HighFrequencyTask(). In the STM32 MC firmware subsystem, the functions are called on the SysTick interrupt.

The safety loop is implemented by the TSK_SafetyTask() function. This function basically calls one of TSK_SafetyTask_PWMOFF() or TSK_SafetyTask_LSON() depending on the chosen overvoltage protection. TSK_SafetyTask() is invoked periodically, at the same frequency as the reference computation loop and on the same interrupt.

4.2.3 Fault handling

The MC subsystem reports the faults it detects to the application. On the detection of a fault, the MC firmware first executes actions to place the motor hardware subsystem in a safe state and then it enters a fault state. These actions always result in the faulty motor being stopped.

The faults that are detected can be seen in the table below:

Table 5. Detected fault

Component	Description
MC_NO_ERROR	No fault is currently pending on the motor control subsystem
MC_DURATION	The 6-step loop lasted too long (the PWM timer update event occurred before the new PWM duty cycle values were available)
MC_OVER_VOLTAGE	An overvoltage condition was detected on the Bus
MC_UNDER_VOLTAGE	An undervoltage condition was detected on the Bus
MC_OVER_TEMP	The temperature of the system has crossed the maximum threshold
MC_START_UP	The start-up phase ended before the speed and position estimation was reliable
MC_SPEED_FDBK	The speed feedback is no longer reliable (usually happens when the rotor speed goes too low)
MC_BREAK_IN	An overcurrent condition or a general fault signal from a power device embedded protection was detected
MC_SW_ERROR	A non-motor dependent error (pure MC firmware error) was detected

The handling of faults in the MC firmware involves two states of the MC state machine. When a fault is detected, the MC state machine enters the FAULT_NOW state which indicates that a fault condition currently exists. On entering this state, the PWM output is immediately cut off. The MC state machine remains in this state as long as the fault condition remains valid, that is, as long as the condition that led to declaring the fault is true. When no fault condition is active any more, the MC state machine switches to the FAULT_OVER state and remains in that state until the application acknowledges them. On the acknowledgement of the faults, the MC state machine goes back to the IDLE state and the subsystem is ready to start the motor again. See Section 4.2.5.

4.2.4 ADC conversions for the application

There are situations where the application needs to use free channels of the ADC peripheral used by the MC subsystem for phase Back-EMF measurement. As described in configuring peripherals with the STM32CubeMX, these ADC channels can be configured with the STM32CubeMX.

However, the application must not use these channels directly. It should rather use the API functions described in programming a regular conversion on a motor control ADC, retrieving the result of a motor control ADC regular conversion and retrieving the state of a motor control ADC regular conversion. Indeed, the instants when the phase Back-EMF measurements are to be made must be set within the PWM period. In the firmware, regular conversion and preferably injected conversions (if available), are used and external triggers coming from the PWM timer starts them.

Hence, the application cannot use injected conversions on these ADC peripherals as they are reserved for motor control and it must avoid disturbing the regular (or injected) conversions dedicated to Back-EMF sensing. The purpose of the APIs mentioned here is to allow the application to perform regular ADC conversions without disturbing the motor control subsystem.

In order to get a user conversion done, at first it must be registered through the RCM_RegisterRegConv_WithCB() or RCM_RegisterRegConv() APIs included in the Regular Conversion Manager library module (RCM). Multiple conversions can be registered, but only one can be scheduled at a time.

A requested user regular conversion is executed by the medium frequency task after the MC-SDK regular safety conversions: Bus voltage and Temperature. If a callback is registered, particular care must be taken with the code executed inside it. The callback code is executed under the Medium frequency task interrupt context (Systick).

If a callback is not registered, the RCM module state machine must be polled to know if the conversion is ready to be read, scheduled, or free to be scheduled. This is performed through the RCM_GetUserConvState() API.

4.2.5 Motor control state machine

The MC firmware subsystem maintains a state machine for the motor it controls. The tasks executed on the motor and the API functions that can be called depend on the current state of its MC state machine.

Figure 23 details the full MC state machine. States are indicated in the blue circles while possible transitions between the states are marked with the arrows.

The actual state machine may be simpler depending on the configured application. Indeed, some states are only needed in specific cases. For instance, the states regarding motor acceleration and closed loop switchover are only useful if sensor-less mode is used.

The state machine is never directly changed by the application. Its management is handled by the reference computation loop that is in the TSK_MediumFrequencyTaskM1() function.

Figure 23. **Motor state machine**

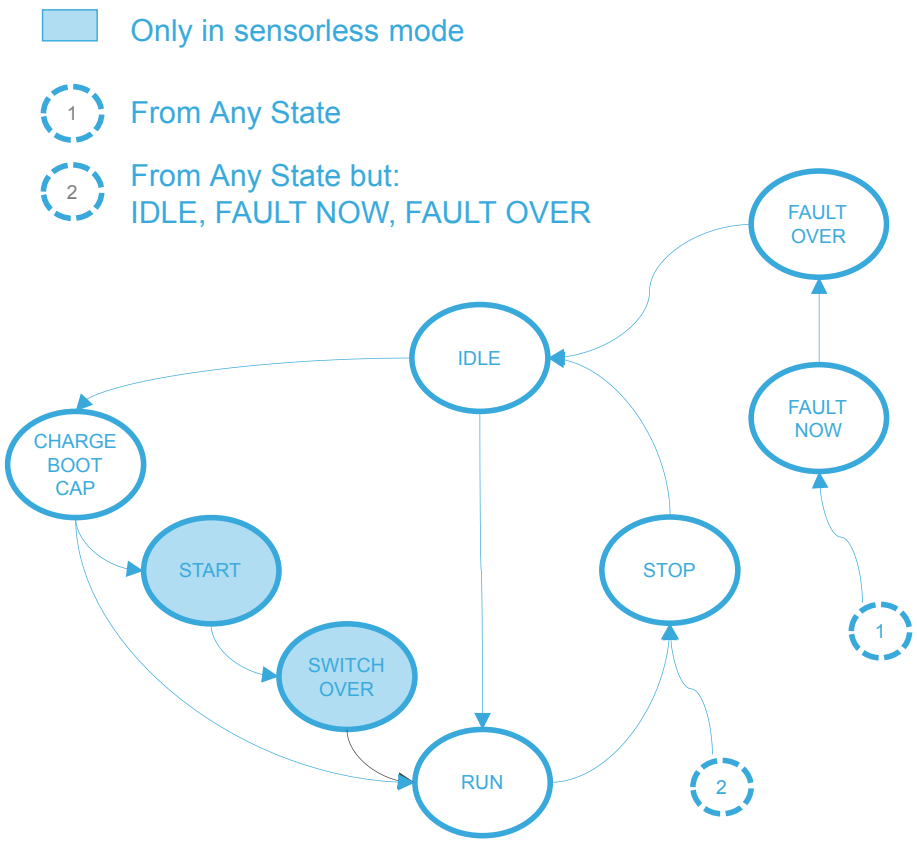


Table 6. **Motor state machine**

Component	Description
IDLE	The motor is not spinning, but is ready to start or to align
CHARGE_BOOT_CAP	State where the gate driver boot capacitors are charged
START	State where the motor alignment and rev-up steps are intended to be executed. It ends with the validation of the sensor-less estimation of the rotor position
SWITCH_OVER	State where speed loop is closed. The following state is normally RUN
RUN	State with running motor. The following state is normally STOP when a stop motor command has been executed.
STOP	The following state is normally IDLE as soon as conditions for moving state machine are detected (stop procedure is completed)
FAULT_NOW	The state machine can be moved from any condition directly to this state by the STM_FaultProcessing() function. This method also manages the passage to the only allowed following state that is FAULT_OVER.
FAULT_OVER	State where the application is intended to stay when the fault conditions disappeared. The following state is normally IDLE, state machine is moved as soon as the user has acknowledged the fault condition

4.2.6 Open loop and debug feature

To ease the debug and tuning of the BDLC motor with the MCSDK 6-step algorithm, an open loop mode, enriched with some debug features, has been introduced into the MCSDK firmware.

Open loop

When enabled, the open loop mode replaces the speed control performed by the speed PI module when the algorithm runs in a close loop. The open loop mode works either in voltage or current driving mode. In voltage driving mode, the duty cycle of the PWM timer that controls the 3 phases is no longer driven by the speed PI module, but directly by a directive coming from the user interface (that is, motor pilot). In current mode, a similar directive coming from the same interface controls the duty cycle of another PWM timer used to limit current of the phases and, consequently, the speed of the motor. The open loop mode is also designed to retrieve the information coming from a potentiometer and to convert it into duty cycle information. In this case the function OLS_Potentiometer_Run() is called instead of SPDPOT_Run(), by the successive calls of the MC_APP_PostMediumFrequencyHook_M1() and MC_RunMotorControlTasks() functions triggered by the SysTick interrupt.

Duty cycle ref. table size

The open loop mode can be enabled before performing an MC_StartMotor1(), or when the motor is running (RUN state of the motor state machine). As soon as the feature is enabled, the speed PI process is no longer executed and the duty cycle directive coming from the user interface is stored into a buffer with a maximum depth of 12 samples. A mean computation of these 12 duty cycles' stored values is performed allowing a swift and smooth transition when the user changes the directive.

Like the speed PI control, the periodicity of the open loop duty cycle computation occurrence is based on a SysTick interrupt. The call of OLS_CalcOpenLoopDutyCycleCM() in current mode or OLS_CalcOpenLoopDutyCycleVM() in voltage mode, is performed during the execution of the function TSK_MediumFrequencyTaskM1() in SWITCH_OVER and RUN states of the motor state machine.

RevUp

Like in close loop mode, the startup process with the RevUp phase can be executed before switching to the open loop mode. The alignment and acceleration phases are executed in the same way. But for debugging purposes, this RevUp phase can be bypassed allowing the motor to start from noise if its characteristics allow it. This startup phase execution control is performed during the execution of the function TSK_MediumFrequencyTaskM1() in the START and SWITCH_OVER states of the motor state machine.

On-sensing

The BEMF zero-crossing detection is generally performed when PWM is off, up to a certain speed of the motor, when the process switches to PWM on-sensing. The open loop feature offers the ability to perform this BEMF zero-crossing detection with PWM on-sensing whatever the speed. In this case, no more transition between off and on-sensing is now expected. Note that this has an impact on maintaining the motor running at the lowest speeds. The on-sensing mode enable feature is not working in the current driving mode. When it is activated (OLS_GetOnSensing() = true), the function OLS_SetSamplingPoint() is called instead of the function BADC_SetSamplingPoint(), setting all the specific parameters, that are the thresholds used for zero-crossing detection. The call of the function OLS_SetSamplingPoint() is done during the execution of the TSK_MediumFrequencyTaskM1() in CHARGE_BOOT_CAP and RUN states of the motor state machine.

Voltage factor current factor

BDLC motors are defined to work with a trapezoidal phase current shape at full speed when duty cycle reaches 100%. However, a duty cycle factor can be set to limit the maximum applied duty cycle. This is useful when, for example, in current mode the maximum speed is achieved with a lower value of the duty cycle due to the current limitation.

5 Motor control API

5.1 MC API

The motor control API, also referred to as the MC API, is the main and most straightforward interface offered to applications for controlling the motors driven by the STM32 MC subsystem.

For the sake of simplicity, the MC API offers one set of functions restricting the number of parameters these functions expect to the bare minimum.

The main purpose of this API is to start the motors, stop the motors and control their rotation. The control of the rotation of a motor is achieved by programming a speed reference that the PID regulator of the motor control subsystem maintains. Such a reference must be set prior to starting a motor.

The speed reference is programmed as a ramp that moves the actual reference from its current value to its target value in a given time.

A programmed reference or ramp is executed at once if the motor is spinning and in steady-state (its state machine is in the RUN state). Otherwise, it is buffered until the state machine of the motor reaches the RUN state. Only one reference or ramp can be programmed at a time, the last one replacing the previous.

In addition to the rotation controlling functions, the MC API also provides functions to get the values of various parameters and state variables of the MC subsystem such as the mechanical or electrical speed for instance.

A brief descriptions of the main functions the MC API consists of is given here along with the usage principles. A complete definition is available in the STM32 MC SDK reference manual.

5.1.1 Starting a motor

bool MC_StartMotor1(void);

Starts the target motor. Prior to calling this function, a speed ramp or a duty cycle reference must have been set.

5.1.2 Stopping a motor

bool MC_StopMotor1(void);

Stops the target motor. If the target motor is not spinning, this function does nothing. Otherwise, the PWM outputs are switched off, whether the MC subsystem is in closed loop or still in the rev-up phase.

5.1.3 Programming a speed ramp

void MC_ProgramSpeedRampMotor1(int16_t hFinalSpeed, uint16_t hDurationms);

Programs a speed ramp on the target motor. If the target motor is in the RUN state – that is, the Motor is spinning and is in steady-state – the ramp is executed immediately. Otherwise, it is buffered until this state is reached.

A speed ramp takes the motor from its rotation speed at the start of the ramp to the hFinalSpeed target speed of the ramp in the hDurations duration.

5.1.4 Stopping an on-going speed ramp

bool MC_StopSpeedRampMotor1(void);

Stops the execution of the current speed ramp of the target motor.

5.1.5 Retrieving the status of a ramp

bool MC_HasRampCompletedMotor1(void);

Returns true if the last submitted ramp for the target motor has completed, false otherwise.

5.1.6 Retrieving the PWM duty cycle reference

uint16_t MCI_GetDutyCycleRefMotor1 ();

Returns the current duty cycle reference applied to the output phases.

5.1.7 Retrieving the state of commands

MCI_CommandState_t MC_GetCommandStateMotor1(void);

Returns the state of the last submitted command for the target motor. “Command” means a speed ramp or a duty cycle reference setting.

The returned state is an MCI_CommandState_t enumerable value:

- MCI_BUFFER_EMPTY: no command has been submitted;
- MCI_COMMAND_NOT_ALREADY_EXECUTED: A command has been buffered but its execution has not completed yet;
- MCI_COMMAND_EXECUTED_SUCCESFULLY: Execution of the last buffered command has completed successfully;
- MCI_COMMAND_EXECUTED_UNSUCCESFULLY: Execution of the last buffered command has completed unsuccessfully.

5.1.8 Retrieving the control mode of the motor

STC_Modality_t MC_GetControlModeMotor1();

Returns the current control mode for the target motor.

5.1.9 Retrieving the drive mode of the motor

DrivingMode_t MC_GetDriveModeMotor1();

Returns the drive mode for the target motor. It can be either VM (voltage mode) or CM (current mode).

5.1.10 Retrieving the direction of rotation of the motor

int16_t MC_GetImposedDirectionMotor1(void);

Returns the direction imposed by the last command on the target motor. The returned value is either 1 or -1.

5.1.11 Retrieving speed sensor reliability

bool MC_GetSpeedSensorReliabilityMotor1(void);

Returns true if the speed sensor of the target motor provides reliable values.

5.1.12 Retrieving average mechanical rotation speed of the motor

int16_t MC_GetMecSpeedAverageMotor1(void);

Returns the last computed average mechanical rotor speed for the target Motor, expressed in dHz (tenth of Hertz).

5.1.13 Retrieving electrical angle of the motor

int16_t MC_GetElAngledppMotor1(void);

Returns the electrical angle of the rotor of the motor.

5.1.14 Motor control fault acknowledgement

int16_t MC_AcknowledgeFaultMotor1(void);

Acknowledges MC faults pending on the target motor. This function returns true if faults were indeed pending and false otherwise. Refer to Section 4.2.3 for more information on MC fault management.

5.1.15 Retrieving the latest motor control faults

int16_t MC_GetOccurredFaultsMotor1(void);

Returns a bit field showing faults that occurred since the MC state machine of the target motor was moved to the FAULT_NOW state. Refer to Section 4.2.3 for more information on MC fault management and to Section 4.2.5 for a description of the MC state machine.

5.1.16 Retrieving all motor control faults

int16_t MC_GetCurrentFaultsMotor1(void);

Returns a bit field showing all current faults on the target motor. Refer to section Section 4.2.3 for more information on MC fault management.

5.1.17 Retrieving the state of the motor control state machine

int16_t MCI_GetSTMStateMotor1(void);

Returns the current state of the target motor state machine. Refer to section Section 4.2.5 for a description of the MC state machine and of the values of the State_t enumerable.

5.2 Motor control low level API

The low level application programming interface provided by the MC firmware allows applications that need it a finer control over the internals of the MC subsystem. This API consists of all the components that are instantiated to form the subsystem. These components can be addressed by the application thanks to their handles. These handles are defined in the mc_config.c file and can be accessed by including the mc_config.h file. For more information, see the STM32 MC SDK reference manual delivered with the SDK.

Revision history

Table 7. **Document revision history**

Date	Version	Changes
9-Aug-2022	1	Initial release.
03-Jun-2024	2	Added Section 4.2.6, Section 4.1.2.4, Section 2.2.1

Contents

List of tables

List of figures