STM32 motor control SDK

Introduction

This manual describes the X-CUBE-MCSDK and X-CUBE-MCSDK-FUL STM32 motor control software development kits (SDKs) designed for, and to be used with, STM32 microcontrollers. The SDKs contain a software library that implements the field oriented control (FOC) drive of 3-phase permanent magnet synchronous motors (PMSMs), both surface mounted (SM-PMSM) and interior (I-PMSM).

The STM32 family of 32-bit Flash microcontrollers is specifically developed for embedded applications. It is based on the following ARM® Cortex®-M cores: the Cortex®-M0 for the STM32F0, the Cortex®-M3 for the STM32F1 and STM32F2, and the Cortex®-M4 for the STM32F3, STM32F4 and STM32L4, and the Cortex®-M7 for the STM32F7. These microcontrollers combine high performance with first-class peripherals that make them suitable for performing three-phase motor FOC.

The PMSM FOC library 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. This library can be easily configured to make use of the STM32F30x's embedded advanced analog peripherals (fast comparators and programmable gain amplifiers (PGAs)) for current sensing and protection, thus simplifying application boards. When deployed with the STM32F103 (Flash memory from 256 Kbytes to 1Mbyte), STM32F303 or STM32F4 devices, the library allows two motors to be driven simultaneously.

The library can be customized to suit user application parameters (motor, sensors, power stage, control stage, pin-out 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 PMSM FOC library. Thanks to this, the user can run a PMSM motor in a very short time.

A set of ready-to-use examples is provided to explain the use of the motor control API and its most commonly used features. These projects usually provide a UART interface that allows convenient real-time fine-tuning of the motor control subsystem with a remote control tool, the STM32 motor control monitor.

The STM32 motor control SDK is delivered as an expansion pack for the STM32 CubeMX tool, and the PMSM FOC 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 About this document

1.1 General information

This document applies to Arm®-based devices.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

1.2 Terms and abbreviations

Table 1. Terms and abbreviations shows the list of acronyms used in this document:

Table 1. Terms and abbreviations

Acronym	Definition
A/D	Analog to digital
ADC	Analog to digital converter
API	Application programming interface
CMSIS	Cortex® microcontroller software interface standard
CORDIC	Coordinate rotation digital computer
DAC	Digital to analog converter
DC	Direct current
DMA	Direct memory access
DPP	Digit per control period
FOC	Field oriented control
GUI	Graphical user interface
HAL	Hardware abstraction layer
ICL	Inrush current limiter
ICS	Isolated current sensor
IDE	Integrated development environment
ISR	Interrupt service routine
LL	Low layers
MC	Motor control
MCU	Microcontroller unit
NTC	Negative temperature coefficient
NVIC	Nested vector interrupts controller
OCP	Over current protection
OPAMP	Operational amplifier
OS	Operating system
PGA	Programmable gain amplifier
PID	Proportional-integral-derivative (controller)
PLL	Phase-locked loop
PM	Permanent magnet
PMSM	Permanent magnet synchronous motor
PWM	Pulse width modulation
RAM	Random access memory
SDK	Software development kit
SVPWM	Space vector pulse width modulation
UI	User interface
MC WB	Motor control workbench
MC Profiler	Motor control profiler

2 STM32 motor control SDK overview

2.1 Package content and installation

The STM32 MC SDK contains the following items:

- STM32 MC firmware
- STM32 MC WB
- STM32 MC Profiler
- The present document
- The reference documentation of the STM32 MC firmware

This package is provided as an executable that installs all the items mentioned above on the user's computer.

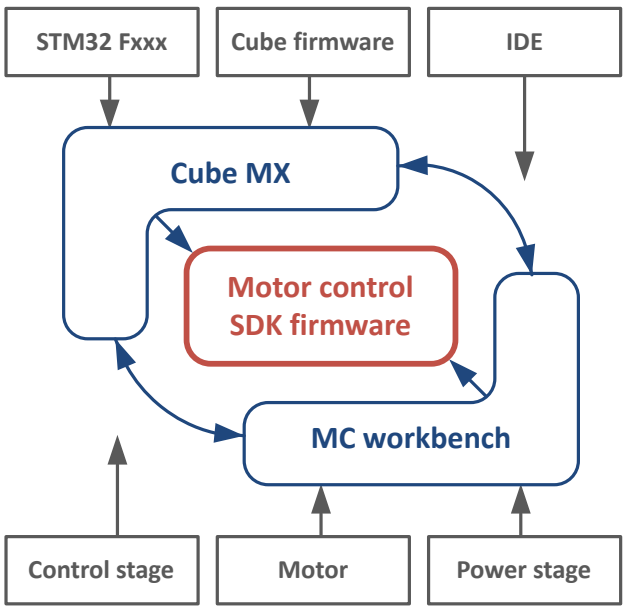
The STM32 MC SDK depends on STM32Cube and STM32CubeMx. Hence, STM32CubeMx version 4.24.0 or later must be installed before the SDK. More information about STM32CubeMx is available at www.st.com.

Note: STM32CubeMx must be run at least once before the MC SDK can be installed.

2.2 Motor control application workflow

The design of a MC software application that uses the STM32 MC SDK typically starts with the MC WB. With this tool, users configure the MC SDK according to the characteristics of their motor, their power stage, their control stage and the chosen STM32 MCU.

Figure 1. Motor control firmware in its environment



Based on these characteristics, MC WB chooses the appropriate firmware components from the PMSM FOC library, computes their configuration parameters, produces a STM32CubeMx project file (referred to as the IOC file from now on, due to its name terminated by the `.ioc` extension.) and executes STM32CubeMx with this project.

The result of this execution is the generation of a complete software project that contains the source code and libraries needed to spin the motors of the application. This software project can be directly opened in the IDE chosen in the workbench.

The code generated by STM32CubeMx configures all the peripherals required to control the application's motors, with the parameters provided by the MC WB. This code also initializes the MC firmware subsystem, sets the STM32 clocks and interrupts handlers so that the motor(s) can be controlled properly.

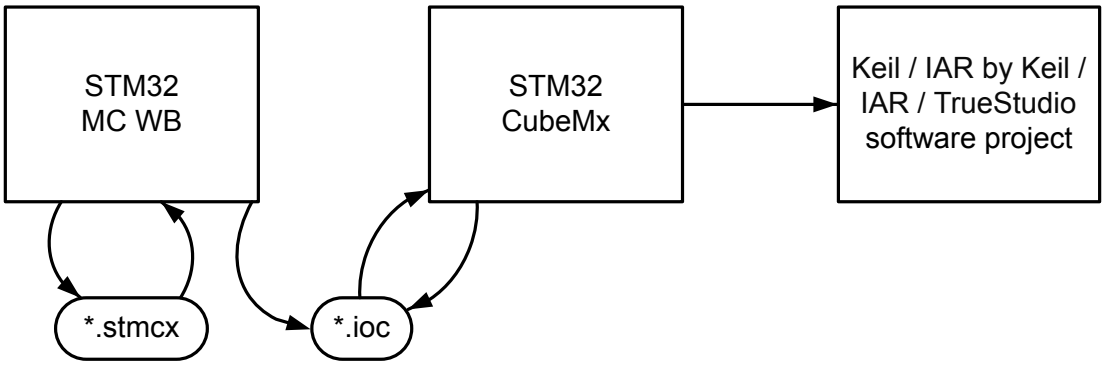
This software project can then be modified by users to add their own code. Refer to application note for detailed information on that subject.

With this workflow, the only tool that is visible to users is the STM32 MC WB. This is sufficient for many applications. If the users need to tune other aspects of their system that impacts the configuration of the STM32 MCU, they can use STM32CubeMx directly: they need to load the project generated by MC WB in CubeMx and then they can modify what they need and finally generate the project again..

Refer to STM32 MC Workbench section of AN5166 for more details on the interactions between STM32CubeMx and STM32 motor control workbench.

Figure 2 shows the MC software application design workflow.

Figure 2. STM32 motor control SDK workflow



In this workflow, STM32 motor control workbench is responsible for computing motor control parameters and instructing STM32 CubeMx on how to configure the hardware IPs needed for it while STM32 CubeMx is used to generate the project and the hardware IPs initialization code. In addition, STM32 CubeMx can be used to configure hardware IPs that are not used for motor control.

The Control stage is at the limit between the both. It is selected and partially configured in the STM32 MC WB STM32CubeMx manages the rest of the configuration.

2.3 STM32Cube firmware

STM32 MC firmware uses the low layer drivers (LL) of the Cube firmware for interfacing with the peripherals it needs to access. These LL drivers are built on the standard CMSIS library.

Using LL offers a good compromise between performances and interface stability in time.

In some places of the code of the PMSM FOC library, direct accesses to hardware registers have been used, for performance reasons.

2.4 STM32 MC firmware

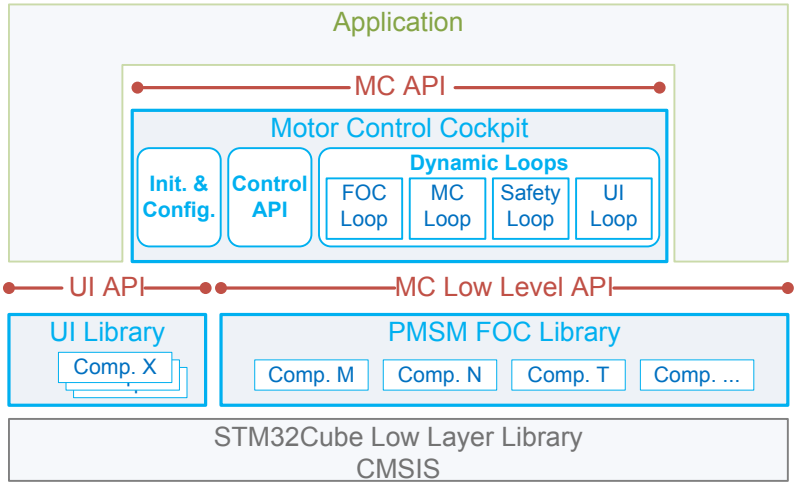
The STM32 MC firmware is the heart of the SDK. It provides all the software components needed to control PMSM using the FOC strategy, and integrates these components into a MC subsystem. It offers a versatile set of interfaces that custom applications can use to actually drive motors according to their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

Figure 3 shows the architecture of the STM32 an MC firmware.

The firmware consists of the three following functional sets:

- The **PMSM FOC 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 3. **STM32 motor control firmware architecture**



2.4.1 PMSM FOC Library

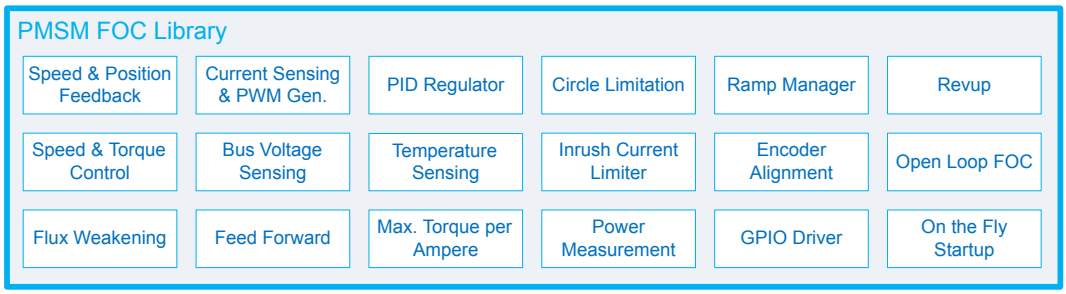
The **PMSM FOC Library** is a collection of software components. Each component implements a feature involved in MC such as, for instance, the speed and position sensing, the current sensing, or motor control algorithms.

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.

Note: For dual motor applications, each motor may use different components for a given feature.

Figure 4. PMSM FOC Library features delivered as components summarizes the features provided by the PMSM FOC library as components. The list of most of the components in the PMSM FOC library and their specificities is described in Section 3.3 Motor control firmware components.

Figure 4. **PMSM FOC Library features delivered as components**



2.4.2 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 or a DAC. This library is used to allow the STM32 MC WB to connect to the Application and control it with its Monitor feature. Refer to the STM32 motor control SDK v5.x tools (UM2380) user guide for more information on this feature.

2.4.3 Motor control cockpit integration

The **Motor control cockpit** integrates the software components into a MC firmware subsystem and implements the regulation loops. It instantiates, configures and interfaces the firmware components selected in the PMSM FOC library and in the User Interface Library for the user's application. The code of the MC Cockpit is generated by STM32Cube as outlined in Section 2.2 Motor control application workflow 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.

2.5 Examples

The STM32 Motor Control SDK is delivered with a set of ready to spin example applications.

2.6 Documentation

The documentation relevant to using the STM32 Motor Control SDK is distributed as follows:

- User manual, STM32 MC SDK, v5.x:
  - Describes the features of the SDK
  - Explains the application design workflow and the interaction with PC tools
  - Details the motor control API
- STM32 MC firmware reference documentation
  - Compressed HTML Help file
  - Provides a comprehensive documentation of all the firmware components provided by the SDK
  - Delivered with the motor control SDK
- STM32 motor control SDK v5.x tools (UM2380) documentation
  - describes the steps and parameters required to customize the library, as shown in the GUI
- STM32Fxxx HAL user manuals
  - Compressed HTML Help file
  - Provide comprehensive documentations of the STM32Cube firmware libraries
  - Available from www.st.com
- Datasheets of supported STM32 MCUs
  - Available from www.st.com
- Cortex M0, M3 and M4 technical reference manuals, available from http://infocenter.arm.com

3 The motor control firmware

3.1 Introduction to PMSM FOC drive

The PMSM FOC software library offers an implementation of the high performance, well-established Field Oriented Control (FOC) strategy for driving Permanent-Magnet Synchronous Motors (PMSM).

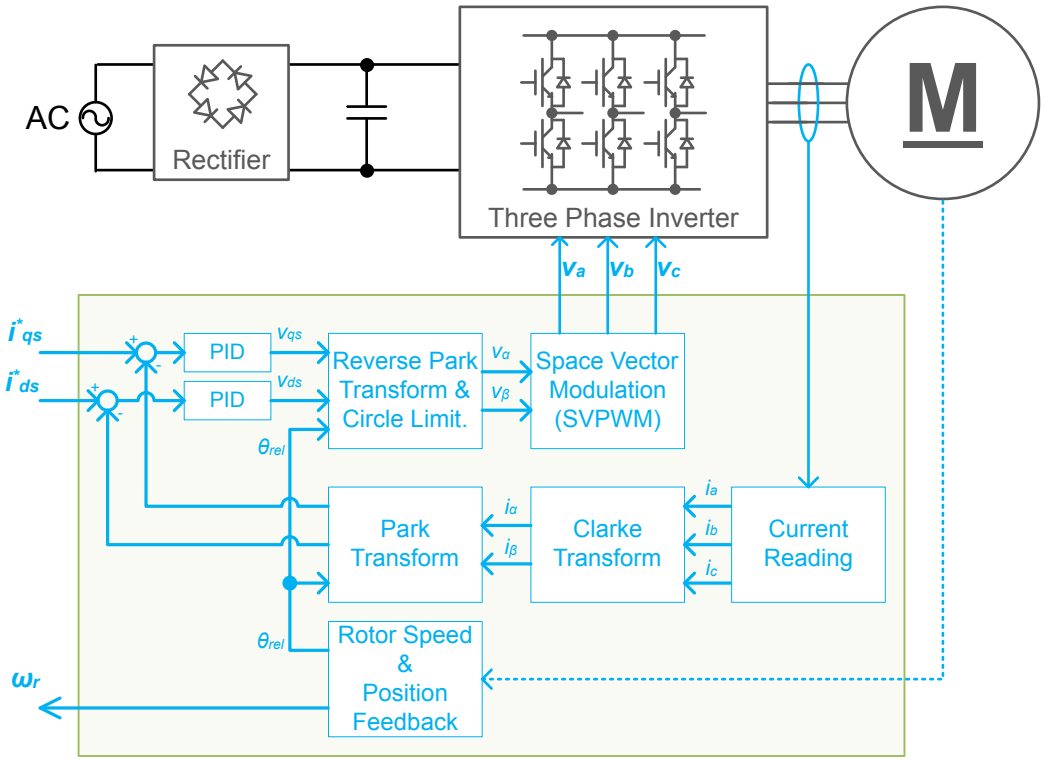
With this approach, it is possible to offer electromagnetic torque (T_e) regulation and, to some extent, flux weakening capability by controlling the two currents i_{qs} and i_{ds} , which are mathematical transformations of the stator currents.

This resembles the favourable condition of a DC motor, where those roles are held by the armature and field currents.

Therefore, one can say that FOC consists in controlling and orienting stator currents in phase and quadrature with the rotor flux. This definition makes it clear that a means of measuring stator currents and the rotor angle is needed.

The structure of the FOC algorithm is represented in Figure 5. Basic FOC algorithm structure, torque control .

Figure 5. Basic FOC algorithm structure, torque control



- The i_{qs}^* and i_{ds}^* current references can be selected to perform electromagnetic torque and flux control.
- The Space Vector PWM block (SVPWM) implements an advanced modulation method that reduces current harmonics, thus optimizing DC bus exploitation.
- The current reading block allows the system to measure stator currents correctly, using either cheap shunt resistors or market-available Hall Sensors or Isolated Current Sensors (ICS).
- The rotor speed/position feedback block allows the system to handle Hall sensor or incremental encoder signals in order to correctly acquire the rotor angular velocity or position. Moreover, this firmware library provides sensor less detection of rotor speed/position.

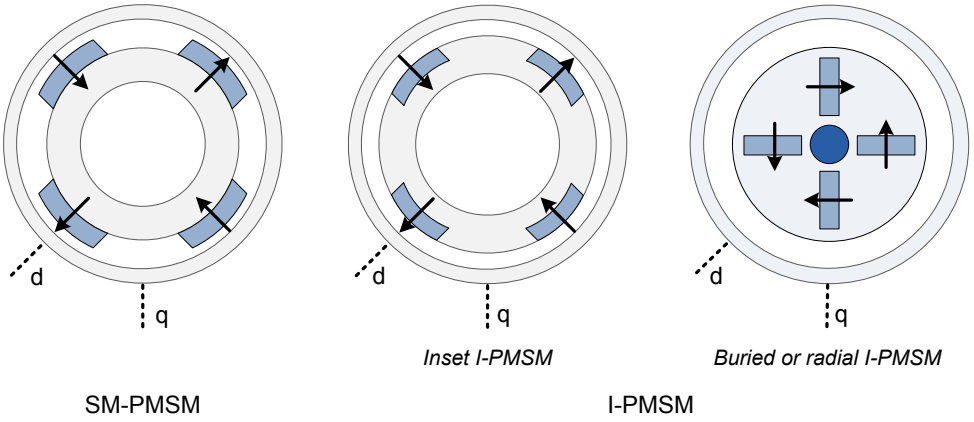
- The PID controller blocks implement proportional, integral and derivative feedback controllers (current regulation).
- The Clarke, Park, Reverse Park and Circle limitation blocks implement the mathematical transformations required by FOC.

3.1.1 Permanent Magnet Motors structure

Two different PMSM constructions are available (see Figure 6. Different Permanent Magnet Motor construction):

- The Surface Mounted PMSM – abbreviated into SM-PMSM – where the magnets are placed on the surface of the motor;
- The Interior Mounted PMSM – abbreviated into I-PMSM – where the magnets are embedded in the structure of the rotor.

Figure 6. Different Permanent Magnet Motor construction



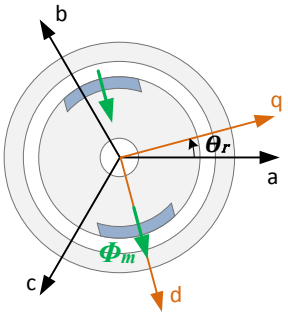
SM-PMSMs inherently have an isotropic structure, which means that the direct and quadrature inductances L_d and L_q are the same. Usually, their mechanical structure allows a wider airgap which, in turn, means lower flux weakening capability.

On the other hand, I-PMSMs show an anisotropic structure (with $L_d < L_q$, typically), slight in the case of inset PM motors, strong in the case of radial PM motors. This peculiar magnetic structure can be exploited to produce a greater amount of electromagnetic torque. Their fine mechanical structure usually shows a narrow air gap, thus giving good flux weakening capability.

This firmware library is optimized for use in conjunction with SM-PMSMs and I-PMSMs machines.

3.1.2 PMSM fundamental equations

Figure 7. PMSM Reference Frame convention



The motor voltage and flux linkage equations of a PMSM (SM-PMSM or I-PMSM) are generally expressed as:

$$V_{abc_S} = r_s i_{abc_S} + \frac{d\lambda_{abc_S}}{dt} \tag{1}$$

$$\lambda_{abc_S} = \begin{bmatrix} L_{Is} + L_{ms} & -\frac{L_{ms}}{2} & -\frac{L_{ms}}{2} \\ -\frac{L_{ms}}{2} & L_{Is} + L_{ms} & -\frac{L_{ms}}{2} \\ -\frac{L_{ms}}{2} & -\frac{L_{ms}}{2} & L_{Is} + L_{ms} \end{bmatrix} + \begin{bmatrix} \sin(\theta_r) \\ \sin\left(\theta_r - \frac{2\pi}{3}\right) \\ \sin\left(\theta_r + \frac{2\pi}{3}\right) \end{bmatrix} \Phi_m \tag{2}$$

where:

- r_s = stator phase winding resistance
- L_{Is} = stator phase winding leakage inductance
- L_{ms} = stator phase winding magnetizing inductance; in case of an I-PMSM, self and mutual inductances have a second harmonic component L_{2s} proportional to $\cos\left(2\theta_r + k \times \frac{2\pi}{3}\right)$,

with $k = 0 \pm 1$, in addition to the constant component L_{ms} (neglecting higher order harmonics)

- θ_r = rotor electrical angle
- θ_m = flux linkage due to permanent magnets

The complexity of these equations is apparent, as the three stator flux linkages are mutually coupled, and as they are dependent on the rotor position, which is time-varying and a function of the electromagnetic and load torques.

The reference frame theory simplifies the PM motor equations by changing a set of variables that refers the stator quantities abc (that can be visualized as directed along axes each 120° apart) to q and d components, directed along orthogonal axes, rotating synchronously with the rotor, and vice versa. The d "direct" axis is aligned with the rotor flux, while the q "quadrature" axis aims at 90 degrees in the positive rolling direction.

The motor voltage and flux equations are simplified to:

$$\begin{cases} v_{qs} = r_s i_{qs} + \frac{d\lambda_{qs}}{dt} + \omega_r \lambda_{ds} \\ v_{ds} = r_{ds} i_{qs} + \frac{d\lambda_{ds}}{dt} - \omega_r \lambda_{qs} \end{cases} \tag{3}$$

$$\begin{cases} \lambda_{qs} = L_{qs} i_{qs} \\ \lambda_{ds} = L_{qs} i_{qs} + \Phi_m \end{cases} \tag{4}$$

For an SM-PMSM, the inductances of the d and q axis circuits are the same (refer to Section 3.1.1 Permanent Magnet Motors structure), that is:

$$L_s = L_{qs} = L_{ds} = \frac{3}{2} L_{ms} \tag{5}$$

On the other hand, I-PMSMs show a salient magnetic structure; thus, their inductances can be written as:

$$\begin{cases} L_{qs} = L_{Is} + \frac{3}{2}(L_{ms} + L_{2s}) \\ L_{ds} = L_{is} + \frac{3}{2}(L_{ms} - L_{2s}) \end{cases} \quad (6)$$

SM-PMSM field-oriented control (FOC)

The equations below describe the electromagnetic torque of an SM-PMSM:

$$T_e = \frac{3}{2}\overline{p}\left(\lambda_{ds} i_{qs} - \lambda_{qs} i_{ds}\right) = \frac{3}{2}\overline{p}\left(L_s i_{ds} i_{qs} - L_s i_{qs} i_{ds} + \Phi_m i_{qs}\right) \quad (7)$$

$$T_e = \frac{3}{2}\overline{p}\left(\Phi_m i_{qs}\right) \quad (8)$$

The last equation makes it clear that the quadrature current component i_{qs} has linear control on the torque generation, whereas the current component i_{ds} has no effect on it (as mentioned above, these equations are valid for SM-PMSMs).

Therefore, if I_s is the motor rated current, then its maximum torque is produced for i_{qs} = I_s and $i_{ds} = 0$ (in fact, $I_s = \sqrt{i_{qs}^2 + i_{ds}^2}$). In any case, it is clear that, when using an SM-PMSM, the torque/current ratio is optimized by letting $i_{ds} = 0$. This choice corresponds to the MTPA (maximum-torque-per-ampere) control for isotropic motors.

On the other hand, the magnetic flux can be weakened by acting on the direct axis current i_{ds} ; this extends the achievable speed range, but at the cost of a decrease in maximum quadrature current i_{qs} , and hence in the electromagnetic torque supplied to the load.

In conclusion, by regulating the motor currents through their components i_{qs} and i_{ds} , FOC manages to regulate the PMSM torque and flux. Current regulation is achieved by means of what is usually called a “synchronous frame CR-PWM”.

3.1.3 PID regulator theoretical background

The regulators implemented for Torque, Flux and Speed are actually Proportional Integral Derivative (PID) regulators. PID regulator theory and tuning methods are subjects which have been extensively discussed in technical literature. This section provides a basic reminder of the theory.

PID regulators are useful to maintain a level of torque, flux or speed according to a desired target. Indeed, both the torque and the flux are a function of the rotor position. FOC needs to regulate torque and flux to maximize system efficiency. In addition, the torque is also a function of the rotor speed. Hence, performing speed regulation results into regulating the torque.

The following is the PID general equation. it is used in the general PID regulator:

$$r(t_k) = K_p \times \epsilon\left(t_k\right) + K_i \times \Sigma_{j=0}^{k} \epsilon\left(t_j\right) + K_d \times \left(\epsilon\left(t_k\right) - \epsilon\left(t_k - 1\right)\right) \quad (9)$$

where:

- $\epsilon(t_k)$ is the error of the system observed at time $t = t_k$, while $\epsilon(t_{k-1})$ is the error of the system at time $t = t_k - T_{sampling}$
- K_p is the proportional coefficient.
- K_i is the integral coefficient.
- K_d is the differential coefficient.
- $r(t_k)$ is the reference to apply as output of the PID regulator

In a motor control application, the derivative term can be disabled. This is indeed the case in the STM32 Motor Control SDK although both PID regulator implantations are provided.

3.1.4 Regulator sampling time setting

The sampling period $T_{sampling}$ needs to be modified to adjust the regulation bandwidth. As an accumulative term (the integral term) is used in the algorithm, increasing the loop time decreases its effects (accumulation is slower and the integral action on the output is delayed). Inversely, decreasing the loop time increases its effects (accumulation is faster and the integral action on the output is increased). This is why this parameter has to be adjusted prior to setting up any coefficient of the PID regulator.

In order to keep the CPU load as low as possible and as induced from Eq. (9), the sampling time is directly part of the integral coefficient, thus avoiding an extra multiplication.Eq. (10) and Eq. (11) show the link between the time domain and the discrete system.

$$\text{Time domain: } r(t) = K_p \times \epsilon(t) + K_i \times \int_0^t \epsilon(t)\,dt + K_d \times \frac{d}{dt}\epsilon(t) \quad (10)$$

$$\text{Discrete domain: } r(t_k) = K_p \times \epsilon(t_k) + K_i \times \sum_{j=0}^{k} \epsilon(t_j)dt + K_d \times (\epsilon(t_k) - \epsilon(t_k - 1)) \quad (11)$$

In theory, the higher the sampling rate, the better the regulation. In practice, one must keep in mind that:

- The related CPU load grows accordingly.
- For speed regulation, there is absolutely no need to have a sampling time lower than the refresh rate of the speed information fed back by the external sensors; this becomes especially true when Hall sensors are used while driving the motor at low speed.

3.1.5 A priori determination of flux and torque current PI gains

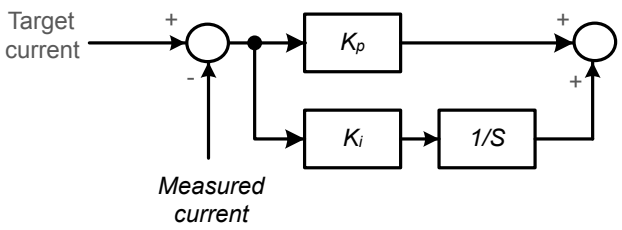
This section provides a criterion for the computation of the initial values of the torque/flux PI parameters (K_i and K_p). This criterion is also used by the STM32 MC WB in its computation.

To calculate these starting values, it is required to know the electrical characteristics of the motor (stator resistance R_s and inductance L_s) and the electrical characteristics of the hardware (shunt resistor R_{shunt} , current sense amplification network AOp and the direct current bus voltage V_{BusDC}).

The derivative action of the controller is not considered using this method.

Figure 8. Block diagram of a PI controller shows the PI controller block diagram used for torque or flux regulation.

Figure 8. Block diagram of a PI controller



For this analysis, the motor electrical characteristics are assumed to be isotropic (in other words, it is SM-PMSM motor) with respect to the q and d axes. It is assumed that the torque and flux regulators have the same starting value of K_p and the same K_i value.

Figure 9. Motor control software subsystem overview shows the closed loop system in which the motor phase is modelled using the resistor-inductance equivalent circuit in the “locked-rotor” condition.

Block “A” is the proportionality constant between the software variable storing the voltage command (expressed in digit) and the real voltage applied to the motor phase (expressed in Volt). Likewise, block “B” is the proportionality constant between the real current (expressed in Ampere) and the software variable storing the phase current (expressed in digit).

The transfer functions of the two blocks “A” and “B” are expressed as:

$$\text{Time domain: } A = \frac{V_{Bus}DC}{2^{16}} \quad (12)$$

and

$$\text{Time domain: } B = \frac{R_{Shunt}A_{op}2^{16}}{3.3} \quad (13)$$

By inserting $\frac{K_p}{K_p} = \frac{L_r}{R_s}$, it is possible to perform pole-zero cancellation.

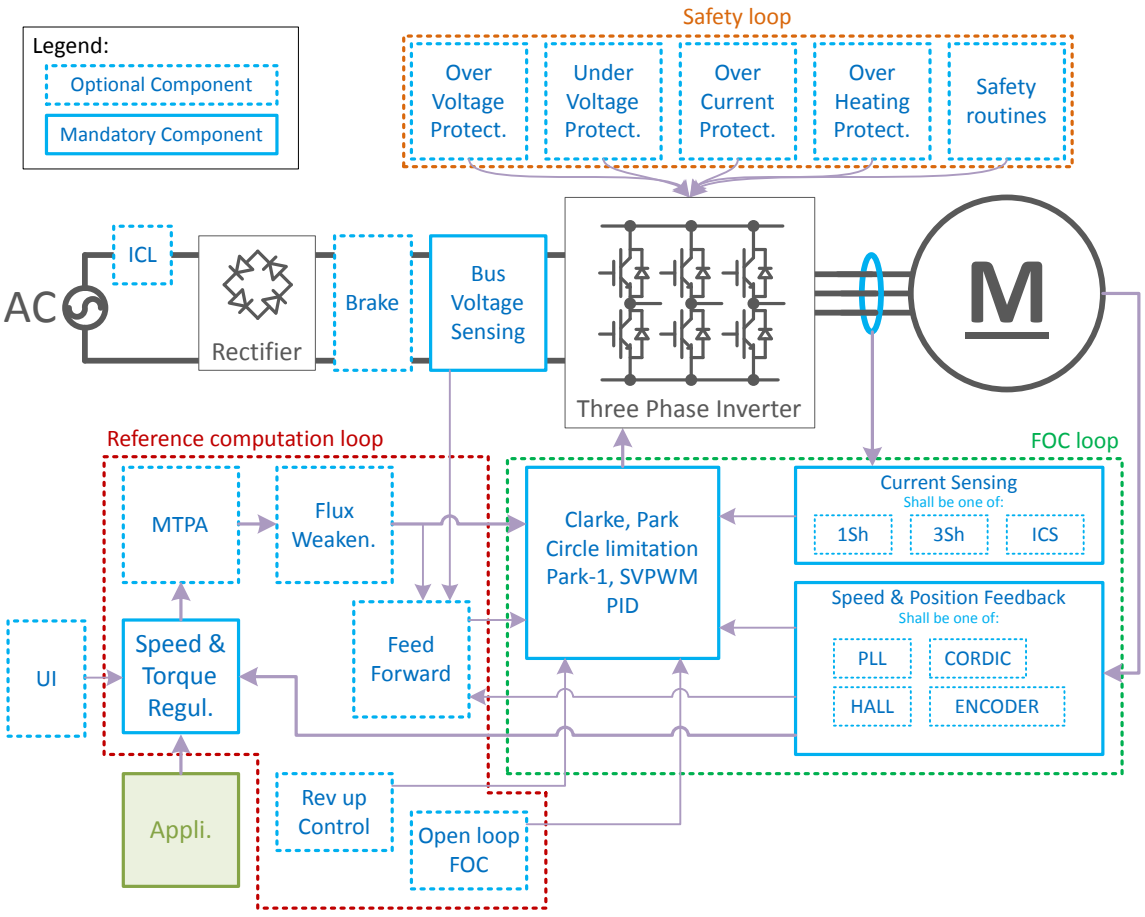
3.2 Motor control firmware subsystem

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 9. Motor control software subsystem overview 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 **FOC loop** is the core of the FOC algorithm. Its aim is to compute the phase voltages and produce the resulting SVPWM duty cycles to apply to the transistors driving the motor's phases. It executes all the mathematical transforms needed to go from the measured (i_a, i_b, i_c) phase currents down to the (i_q, i_d) currents and then back from the (v_q, v_d) voltage up to PWM duty cycles. In the middle, (i_q, i_d) values confronted with the reference (v_q, v_d) by PID regulators that output the pair in return (see Figure 5. Basic FOC algorithm structure, torque control). The FOC loop is executed at a high rate, the PWM frequency.

Figure 9. Motor control software subsystem overview



The purpose of the **Reference computation loop**, as its name suggests, is to compute the (i_q^*, i_d^*) references based on targets coming from the application. Usually, the application provides a reference expressed in a way that matches its needs: a speed or a torque reference or ramp. The reference computation loop first converts the application target into an initial (i_q^*, i_d^*) reference which is then optionally passed through one or several algorithm(s) that aim(s) at optimizing the motor drive: Maximum Torque Per Ampere, Flux Weakening, or Feed

Forward. Then, the resulting (i_q^*, i_d^*) reference is finally used by the FOC loop. 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 will take 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 in 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 Under Voltage protection, Over Heating protection and Over Current protection. In the case of Over Current 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 over current situation. The latest item, the Safety Routines is, for the time being, only a provision for inserting user defined custom code that would add functional safety to the motor control subsystem. It will be extended in future revisions of the STM32 MC SDK.

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.

About the execution rates given in this section, a more complete discussion is available in Section 3.4.2 Tasks of the motor control subsystem.

3.3 Motor control firmware components

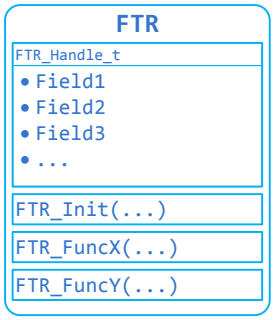
Most of the motor control firmware is organized as a set of software components to the exception of the inner FOC loop for which a decomposition in components was judged inadequate.

A **component** in a self-contained software unit that defines:

- A structure with the data needed to fulfil 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.

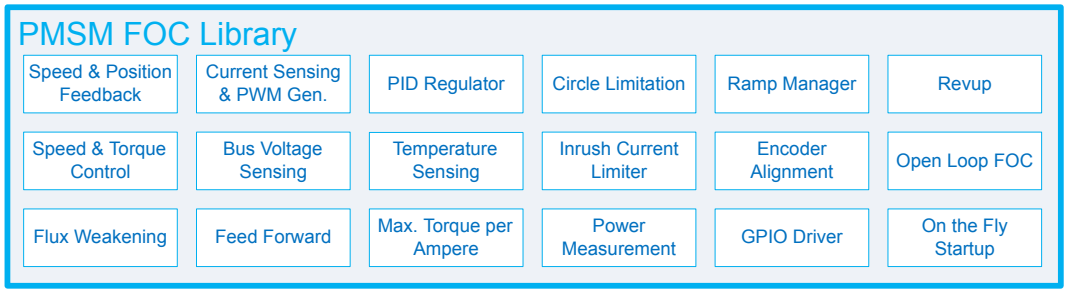
Figure 10. A component with its handle and its functions



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 first argument so that they have access to the state and the settings of the instance of the component they work for.

Figure 11. PMSM FOC Library features delivered as components

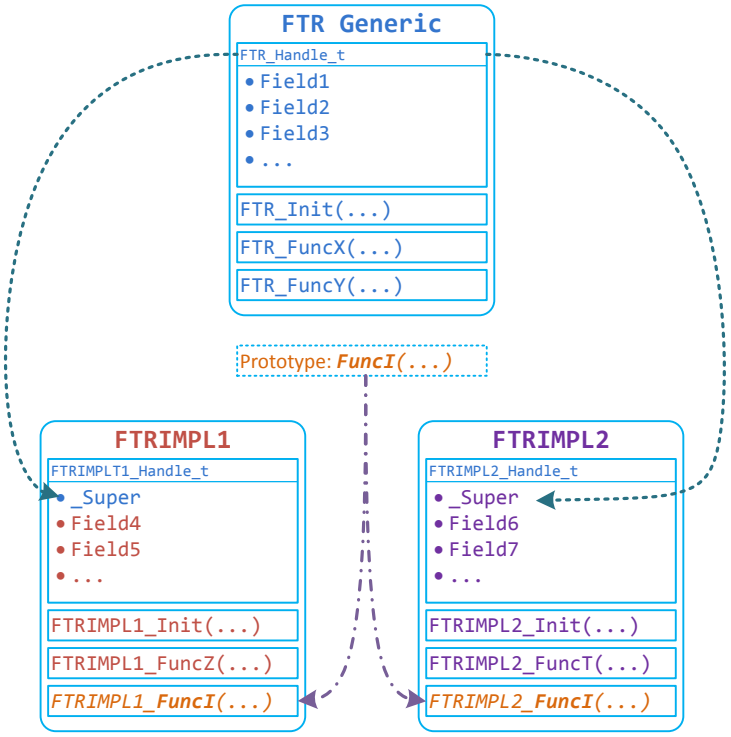


Thanks to this decomposition in Components, a given feature can be instantiated multiple times which is very helpful in a Dual Drive application (and also in Single Drive: for instance, the PID regulator component is used several times for each motor.).

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 need 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. Relationship between generic and implementing components



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. Four actual implementations are provided, one that uses a quadrature encoder to get the speed and the position of the rotor, one that uses Hall Effect sensors and two that implement the feature using state observer based algorithms. Each of these four implementations define their own handle that extends **SpeednPosFdbk_Handle_t** and each define 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.

3.3.1 Current sensing and PWM generation components

Features overview

The Current Sensing and PWM components are responsible for two key features of a FOC subsystem:

- Measuring the current that flows into three phases of the motor;
- Applying the desired voltage on these three phases.

In the STM32 MC firmware these two features are grouped in a single software component because the instants when the currents are to be measured must be synchronized with the SVPWM that applies the voltages. To that end, the STM32 MC firmware uses Timer peripherals that can trigger ADC conversions. The ADC peripherals are used to measure the currents while the Timers are used to generate the PWM signals that drive the voltages applied on the phases and to trigger the measurement of the ADC at the right time. This mechanism is described in more details below.

Concretely, the main task of these components is to compute, for each PWM period, the PWM duty cycles required to apply the reference voltage on the motor phases and the instant when to trigger the ADC for capturing currents. 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 sides transistors on, and the over current protection.

Each PWM and Current Feedback implementation handles the Timer and ADC 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 handles them. The Application shall not tamper neither with the priorities of these interrupts or with the order in which they are served in the interrupt handler. See Section 3.4.3 Interrupts and Real Time aspects.

Available Implementations and specificities

The PMSM FOC Library provides all the components needed for supporting three-shunt, single-shunt, and ICS topologies. Refer to Table 2 for a list of these components. The selection of the component that matches the topology actually in use by the application is performed through the STM32 MC WB.

All these implementations are built on a generic PWM and Current Feedback 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 and Current Feedback specific implementations.

The functions, that the generic PWM and Current Feedback component provides, form the API of the PWM and Current feedback feature. Calling those results in calling functions of the component that actually implement the feature. Hence, the MC Cockpit calls the functions of the generic PWM and Current Feedback component instead of the ones defined by the chosen implementation. This way of using components is specific to the current sensing and PWM generation components. Other sets of components like the Speed and Position Feedback ones do not work this way. Refer to the Reference User Manual for more information.

Table 2. PWM and current feedback components

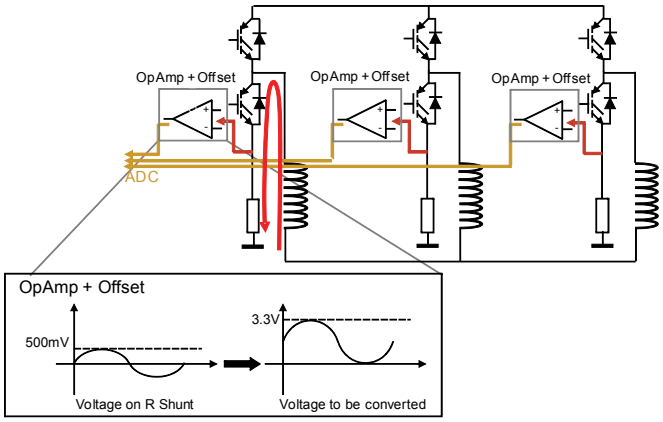
Component	Description
R1 F0xx PWM and Current Feedback	For applications using a Single Shunt current sensing topology and an STM32F0 MCU.
R3 F0xx PWM and Current Feedback	For applications using a Three-Shunt current sensing topology and an STM32F0 MCU.
R1 HD2 PWM and Current Feedback	For applications using a Single Shunt resistor current sensing topology and an STM32F103 High Density MCU (STM32F103xC, STM32F103xD and STM32F103xE).
R1 VL1 PWM and Current Feedback	For applications using a Single Shunt resistor current sensing topology and an STM32F100 Value Line MCU (STM32F100x4, STM32F100x6, STM32F100x8 and STM32F100xB).
R3 HD2 PWM and Current Feedback	For applications using a Three-Shunt resistors current sensing topology and an STM32F103 High Density MCU (STM32F103xC, STM32F103xD and STM32F103xE).
R3 LM1 PWM and Current Feedback	For applications using a Three-Shunt resistors current sensing topology and an STM32F103 Low and Medium Density MCU (STM32F103x4, STM32F103x6, STM32F103x8 an STM32F103xB).
ICS HD2 PWM and Current Feedback	For applications using an Insulated Current Sensors topology and an STM32F103 High Density MCU (STM32F103xC, STM32F103xD and STM32F103xE).
ICS LM1 PWM and Current Feedback	For applications using an Insulated Current Sensors topology and an STM32F103 Low and Medium Density MCU (STM32F103x4, STM32F103x6, STM32F103x8 and STM32F103xB).
R1 F30x PWM and Current Feedback	For applications using a Single Shunt resistor current sensing topology and an STM32F3 MCU.
R3 1 ADC F30x PWM and Current Feedback	For applications using a Three-Shunt resistor current sensing topology and an STM32F3 MCU. This component uses three channels of a single ADC for current measurement.
R3 2 ADCs F30x PWM and Current Feedback	For applications using a Three-Shunt resistor current sensing topology and an STM32F3 MCU. This component uses four channels on 2 ADCs for measuring the currents of one motor. It is designed to be used in dual drive applications where two instances of this component share the same 2 ADC peripherals. It can also use the internal PGA embedded in some STM32F3 MCUs.
R3 4 ADCs F30x PWM and Current Feedback	For applications using a Three-Shunt resistor current sensing topology and an STM32F3 MCU. This component uses four channels on 2 ADCs for measuring the currents of one motor. It is intended to be used in dual drive applications where each instances of this component use 2 ADC peripherals each, leading to a total of 4 ADCs. It can also use the internal PGA embedded in some STM32F3 MCUs.
ICS F30x PWM and Current Feedback	For applications using an Insulated Current Sensors topology and an STM32F3 MCU.
R1 F4xx PWM and Current Feedback	For applications using a Single Shunt resistor current sensing topology and an STM32F4 MCU.
R3 F4xx PWM and Current Feedback	For applications using a Three-Shunt resistor current sensing topology and an STM32F4 MCU.
ICS F4xx PWM and Current Feedback	For applications using an Insulated Current Sensors topology and an STM32F4 MCU.

Access to the Current Feedback ADCs by the application

PWM and Current Feedback 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. PWM and Current Feedback components export a dedicated function PWMC_ExecRegularConv that is used by the MC API to provide access to these channels. However, this function is not for Application use. Instead, 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.

Current sampling in three-shunt topology using two A/D converters

Figure 13. Three-shunt topology hardware architecture



The three currents I_1, I_2, and I_3 flowing through a three-phase system follow the mathematical relation:

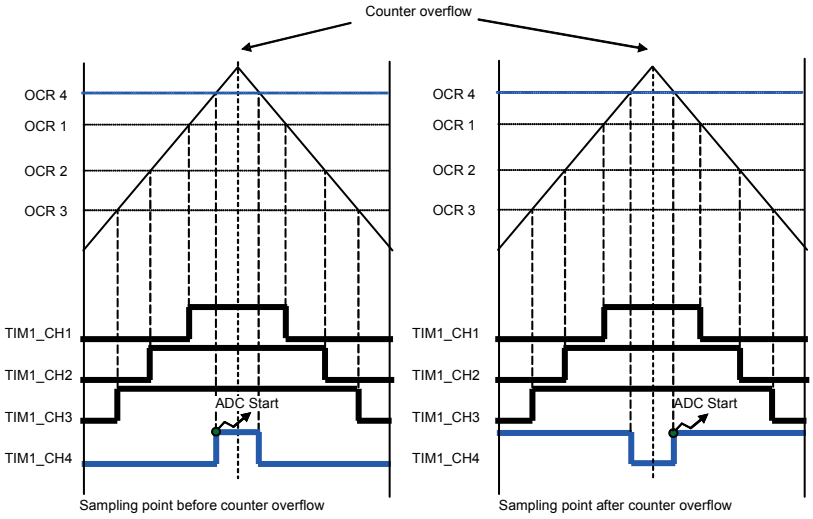
$$I_1 + I_2 + I_3 = 0 \quad (14)$$

For this reason, to reconstruct the currents flowing through a generic three-phase load, it is sufficient to sample only two out of the three currents while the third one can be computed by using the above relation.

The flexibility of the STM32 A/D converter makes it possible to synchronously sample the two A/D conversions needed for reconstructing the current flowing through the motor. The ADC can also be used to synchronize the current sampling point with the PWM output using the external triggering capability of the peripheral. Owing to this, current conversions can be performed at any given time during the PWM period. To do this, the control algorithm uses the fourth PWM channel of TIM1 to synchronize the start of the conversions.

Figure 14. PWM and ADC Synchronization shows the synchronization strategy between the TIM1 PWM output and the ADC. The A/D converter peripheral is configured so that it is triggered by the rising edge of TIM1_CH4.

Figure 14. **PWM and ADC Synchronization**



In this way, supposing that the sampling point must be set before the counter overflow, that is, when the TIM1 counter value matches the OCR4 register value during the up-counting, the A/D conversions for current sampling are started. If the sampling point must be set after the counter overflow, the PWM 4 output has to be inverted by modifying the CC4P bit in the TIM1_CCER register. Thus, when the TIM1 counter matches the OCR4 register value during the down-counting, the A/D samplings are started.

After execution of the FOC algorithm, the value to be loaded into the OCR4 register is calculated to set the sampling point for the next PWM period, and the A/D converter is configured to sample the correct channels.

Table 3. **Three-shunt current reading, used resources (single drive, F103 LD/MD)**

Adv. timer	DMA	ISR	ADC master	ADC slave	Note
TIM1	DMA1_CH1 DMA1_CH5	None	ADC1	ADC2	DMA is used to enable ADC injected conversion external trigger. Disabling is performed by software.

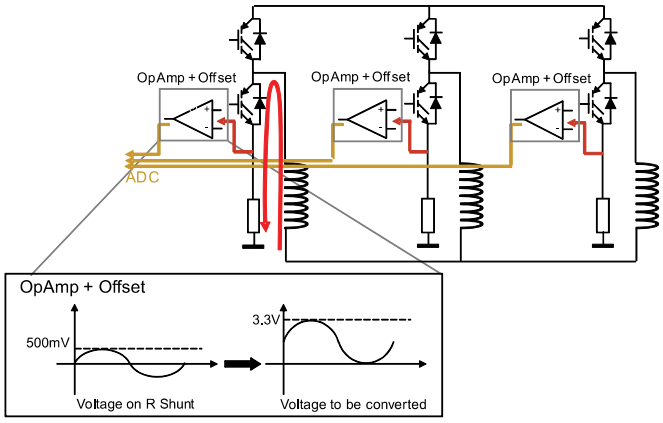
Table 4. **Three-shunt current reading, used resources (Dual drive,F103 HD, F2x, F4x)**

Adv. timer	DMA	ISR	ADC	Note
TIM1	DMA1_CH1	TIM1_UP	ADC1 ADC2	Used by first or second motor configured in three-shunt, according to user selection. ADC is used in time sharing. Trigger selection is performed in the TIM_UP ISR.
TIM8	None	TIM8_UP	ADC1 ADC2	Used by first or second motor configured in three-shunt, according to user selection. ADC is used in time sharing. Trigger selection is performed in the TIM_UP ISR.

Refer to section Current sensing and PWM generation components, for a STM32F30x microcontroller configuration.

Current sampling in three-shunt topology using one A/D converter

Figure 15. Three-shunt topology hardware architecture



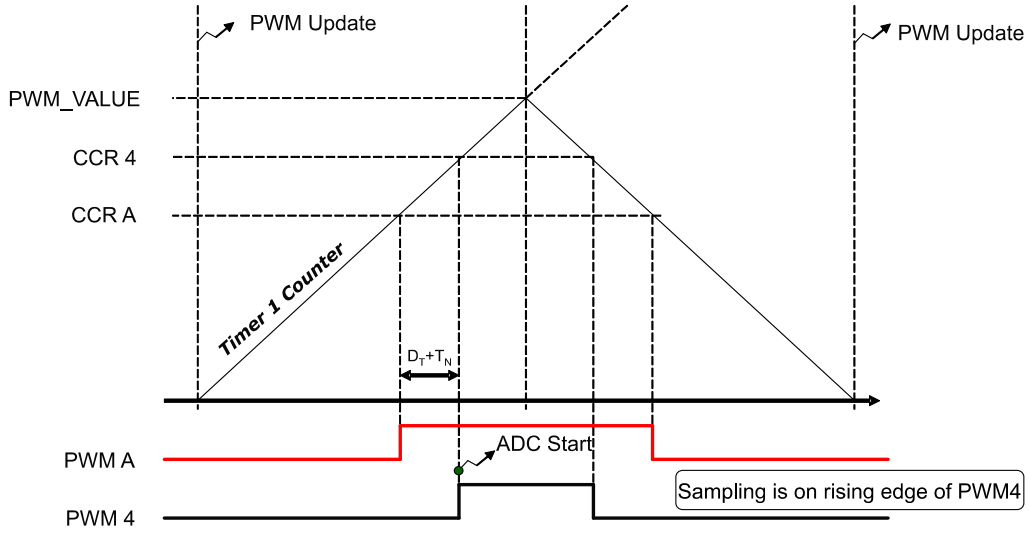
Unlike the case of current sampling with two ADCs, in the case of single ADC it is not possible to synchronously sample the two phase current A/D conversions, needed for reconstructing the current flowing through the motor, but they can be performed only in sequence mode.

The ADC can be used to synchronize the current sampling point with the PWM output using the external triggering capability of the peripheral. Owing to this, current conversion sequence can be performed at any given time during the PWM period.

To do this, the control algorithm uses the fourth PWM channel of TIM1 to synchronize the start of the conversion sequence.

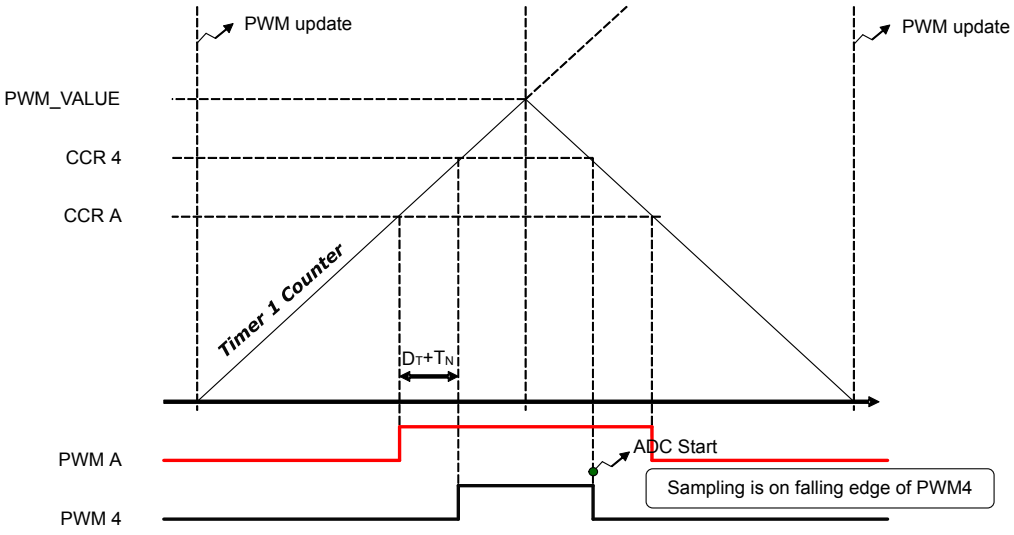
Figure 16. PWM and ADC synchronization ADC rising edge external trigger and Figure 17. PWM and ADC synchronization ADC falling edge external trigger show the synchronization strategy between the TIM1 PWM output and the ADC.

Figure 16. **PWM and ADC synchronization ADC rising edge external trigger**



If CCR A + Delay < PWM_VALUE it is possible to set the CCR 4 equal to CCR A plus the delay and set ADC External trigger as Rising edge

Figure 17. **PWM and ADC synchronization ADC falling edge external trigger**



If CCR A + delay > PWM_VALUE, it is possible to set the CCR 4 equal to CCR A plus the delay and set a falling ADC external trigger

In this way, supposing that the sampling point must be set before the counter overflow, that is, when the TIM1 counter value matches the OCR4 register value during the up counting, the A/D conversion sequence for current sampling are started. If the sampling point must be set after the counter overflow, it is necessary to set a falling edge ADC external trigger. Thus, when TIM1 counter matches the OCR4 register value during the down counting, the A/D sampling is started.

After execution of the FOC algorithm, the value to be loaded into the OCR4 register is calculated to set the sampling point for the next PWM period, and the A/D converter is configured to sample the correct channels.

Table 5. Three-shunt current reading, used resources, single drive, STM32F302x6, STM32F302x8

Adv. Timer	ISR	ADC	Note
TIM1	ADC1_IRQn TIM1_BRK_TIM15_IRQN	ADC1	The dual drive mode and the internal PGA are not available

Table 6. Three-shunt current reading, used resources, single drive, STM32F030x8

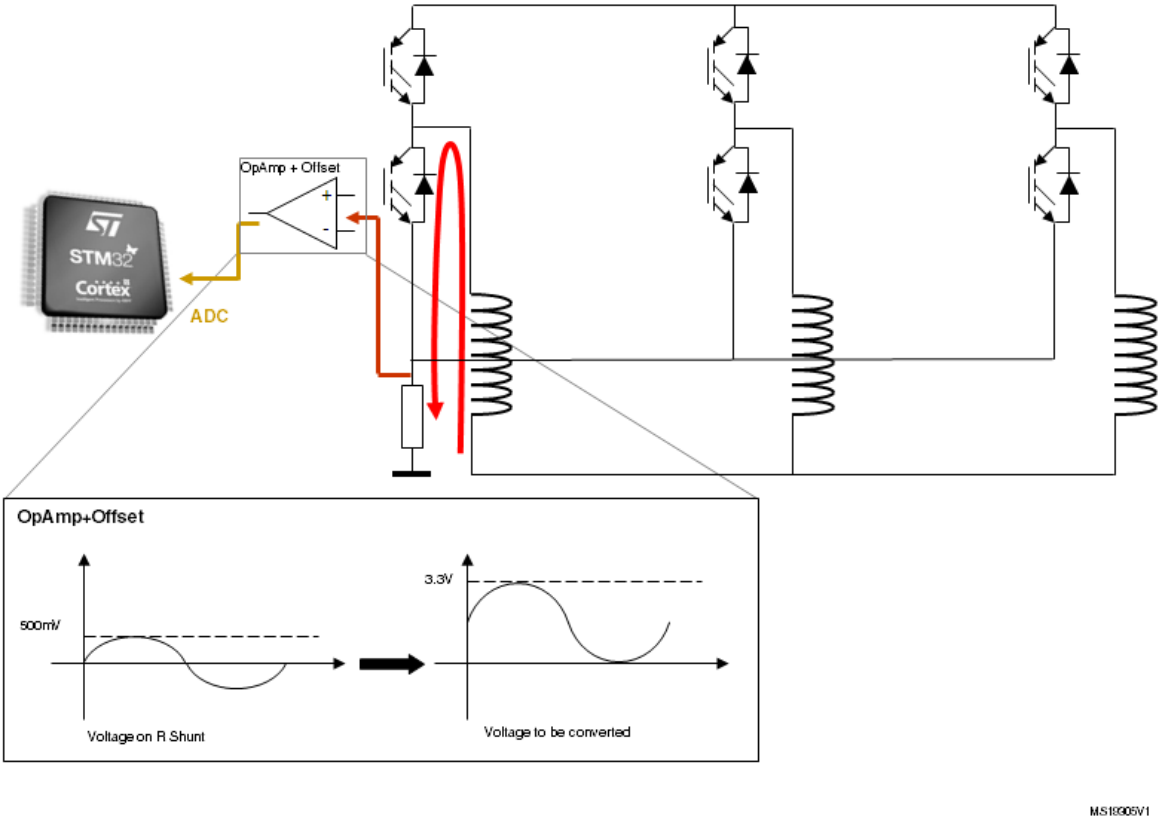
Adv. Timer	ISR	ADC	Note
TIM1	DMA1_Channel1_IRQn TIM1_BRK_UP_COM_IRQN	ADC1	The dual drive mode and the internal PGA are not available

The FOC starts after DMA1_Channel1 Transmission is complete (dual sampling). The DMA is used to manage the A/D conversion sequence since the STM32F0x ADC doesn't support the injected conversion type but only the regular conversion type.

Current Sampling in Single-Shunt topology

Figure 18. Single-shunt hardware architecture illustrates the single-shunt topology hardware architecture.

Figure 18. Single-shunt hardware architecture



It is possible to demonstrate that, for each configuration of the low-side switches, the current through the shunt resistor is given in Table 7. Current through the shunt resistor. T4, T5 and T6 assume the complementary values of T1, T2 and T3, respectively.

In Table 7. Current through the shunt resistor, value “0” means that the switch is open whereas value “1” means that the switch is closed.

Table 7. Current through the shunt resistor

T1	T2	T3	*I*Shunt
0	0	0	0
0	1	1	iA
0	0	1	*-iC*
1	0	1	*iB*
1	0	0	*-iA*
1	1	0	*iC*
0	1	0	*-iB*
1	1	1	0

Using the centered-aligned pattern, each PWM period is subdivided into 7 sub periods (see Figure 19. Single shunt current reading). During three sub periods (I, IV, VII), the current through the shunt resistor is zero. During the other sub periods, the current through the shunt resistor is symmetrical with respect to the center of the PWM.

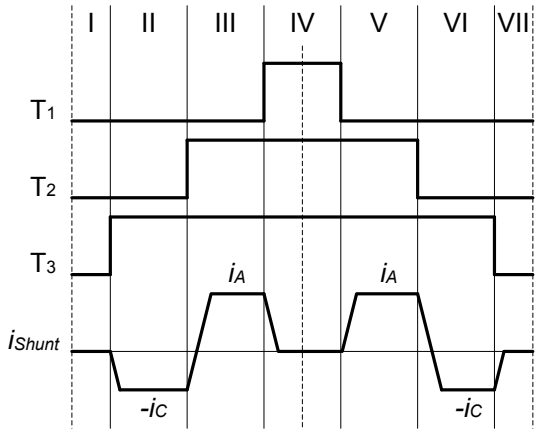
For the conditions showed in Figure 19. Single shunt current reading, there are two pairs:

- sub periods II and VI, during which i_{Shunt} is equal to $-i_C$
- sub periods III and V, during which i_{Shunt} is equal to i_A

Under these conditions, it is possible to reconstruct the three-phase current through the motor from the sampled values:

- i_A is i_{Shunt} measured during sub period III or V
- i_C is $-i_{Shunt}$ measured during sub period II or VI
- $i_B = -i_A - i_C$

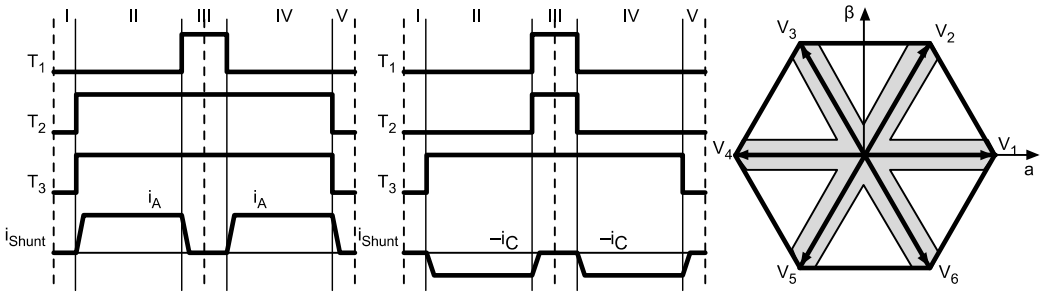
Figure 19. Single shunt current reading



If the stator-voltage vector lies in the boundary space between two space vector sectors, two out of the three duty cycles will assume approximately the same value. In this case, the seven sub periods are reduced to five sub periods.

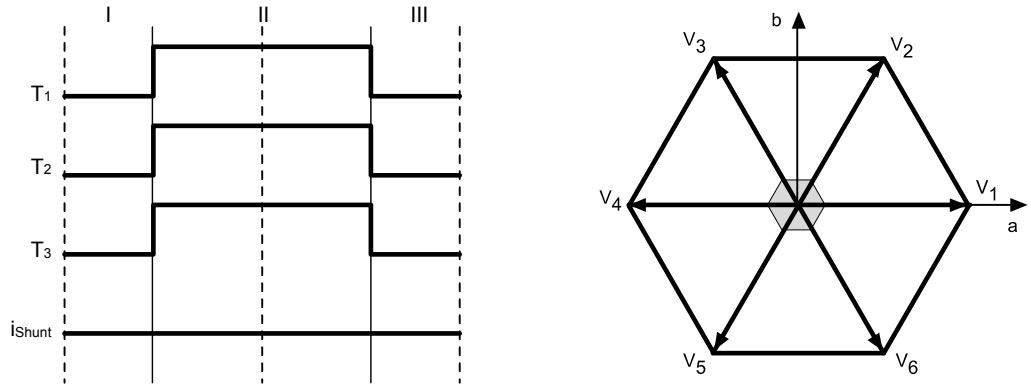
Under these conditions, only one current can be sampled, the other two cannot be reconstructed. This means that it is not possible to sense both currents during the same PWM period, when the imposed voltage demand vector falls in the gray area of the space vector diagram represented in Figure 20. Boundary between two space vector sectors.

Figure 20. Boundary between two space vector sectors



Similarly, for a low modulation index, the three duty cycles assume approximately the same value. In this case, the seven sub-periods are reduced to three sub-periods. During all three sub-periods, the current through the shunt resistor is zero. This means that it is not possible to sense any current when the imposed voltage vector falls in the gray area of the space vector diagram represented in Figure 21. Low modulation index.

Figure 21. Low modulation index



Definition of the noise parameter and boundary zone

T_{Rise} is the time required for the data to become stable in the ADC channel after the power device has been switched on or off.

The duration of the ADC sampling is called the sampling time.

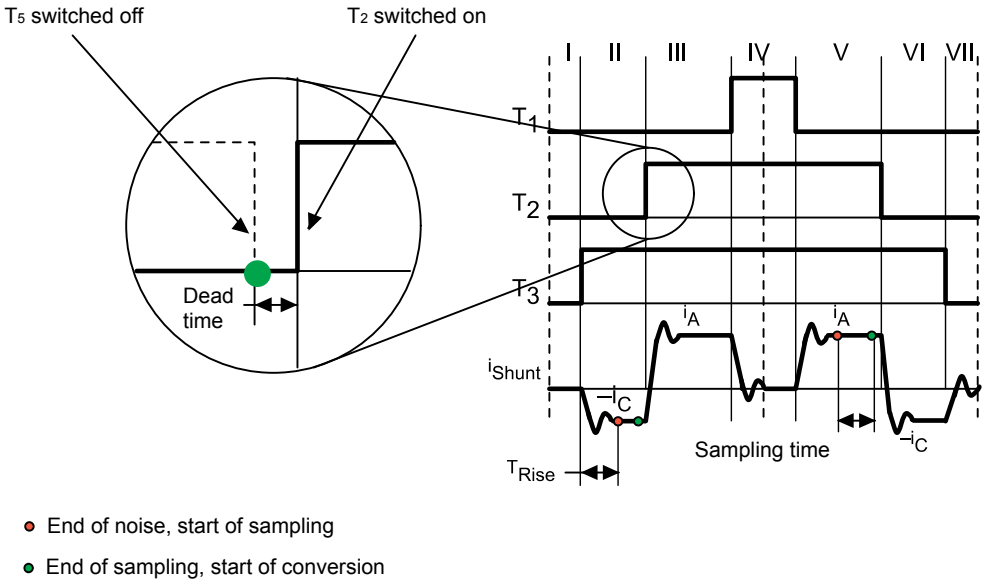
T_{MIN} is the minimum time required to perform the sampling, and:

$$T_{Min} = T_{Rise} + (sampling\ time) + (dead\ time) \quad (15)$$

D_{MIN} is the value of T_{MIN} expressed in duty cycle percent. It is related to the PWM frequency F_{PWM} as follows:

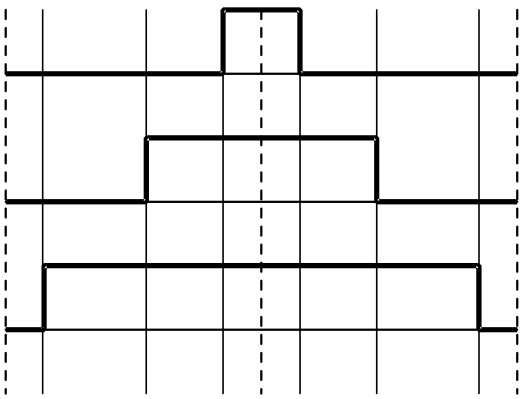
$$D_{MIN} = 100 \times T_{MIN} \times F_{PWM} \quad (16)$$

Figure 22. **Definition of noise parameters**



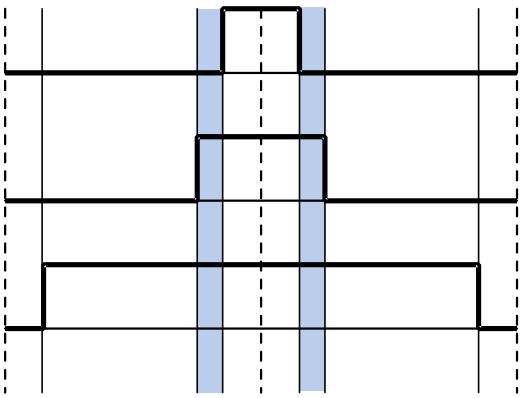
The voltage-demand vector lies in a region called the Regular region when the three duty cycles (calculated by space vector modulation) inside a PWM pattern differ from each other by more than D_{MIN} . This is represented in Figure 22. Definition of noise parameters.

Figure 23. **Regular region**



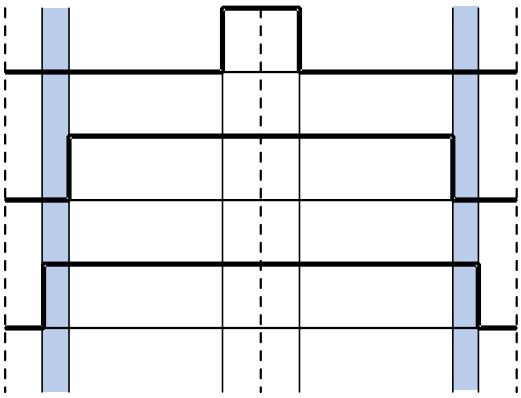
The voltage-demand vector lies in a region called Boundary 1 when two duty cycles differ from each other by less than D_{MIN} , and the third is greater than the other two and differs from them by more than D_{MIN} . This is represented in Figure 24. Boundary 1.

Figure 24. **Boundary 1**



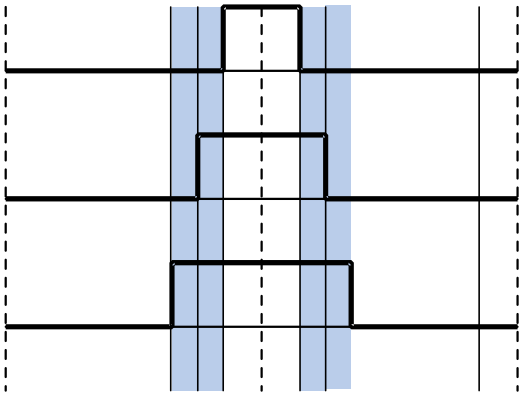
The voltage-demand vector lies in a region called Boundary 2 when two duty cycles differ from each other by less than D_{MIN} , and the third is smaller than the other two and differs from them by more than D_{MIN} . This is represented inFigure 25. Boundary 2.

Figure 25. **Boundary 2**



The voltage-demand vector lies in a region called Boundary 3 when the three PWM signals differ from each other by less than D_{MIN} . This is represented in Figure 26. Boundary 3.

Figure 26. **Boundary 3**



If the voltage-demand vector lies in Boundary 1 or Boundary 2 region, a distortion must be introduced in the related PWM signal phases to sample the motor phase current.

An ST patented technique for current sampling in the “Boundary” regions has been implemented in the firmware. Please contact your nearest ST sales office or support team for further information about this technique.

Table 8. **Single-shunt current reading, used resources (single drive, F103/F100 LD/MD, F0x)**

Adv. Timer	Aux. timer	DMA	ISR	ADC	Note
TIM1	TIM3 (CH4)	DMA1_CH1 DMA1_CH3 DMA1_CH4	TIM1_UP DMA1_CH4_TC (FOC rate > 1)	ADC1	F103/F100 LD device configuration, RC DAC cannot be used; ADC1 is used for general purpose conversions
TIM1	TIM4 (CH3)	DMA1_CH1 DMA1_CH5 DMA1_CH4	TIM1_UP DMA1_CH4_TC (FOC rate > 1)	ADC1	F103/F100 MD device configuration; ADC1 is used for general purpose conversions
TIM1	TIM15 (CH1)	DMA1_CH2 DMA1_CH5 DMA1_CH4	TIM1_UP DMA1_CH4_TC (FOC rate > 1)	ADC1	F051x device configuration
TIM1	TIM3 (CH4)	DMA1_CH2 DMA1_CH3 DMA1_CH4	TIM1_UP DMA1_CH4_TC (FOC rate > 1)	ADC1	F050x/F030x device configuration

Table 9. **Single-shunt current reading, used resources (single or dual drive, F103HD**

Adv. timer	Aux. timer	DMA	ISR	ADC	Note
TIM1	TIM5 (CH4)	DMA1_CH1 DMA2_CH1 DMA1_CH4	TIM1_UP DMA1_CH4_TC (FOC rate > 1)	ADC3	Option1: used by the first motor configured in single-shunt, or the second motor when the first is not single-shunt; ADC1 is used for general purpose conversions
TIM8	TIM4 (CH3)	DMA1_CH1 DMA1_CH5 DMA2_CH2	TIM8_UP DMA2_CH2_TC (FOC rate > 1)	ADC1	Option1: used by the second motor configured in single-shunt when the first motor is also configured in single-shunt.

Adv. timer	Aux. timer	DMA	ISR	ADC	Note
TIM8	TIM5 (CH4)	DMA1_CH1 DMA2_CH1 DMA2_CH2	TIM8_UP DMA2_CH2_TC (FOC rate > 1)	ADC3	Option2: used by the first motor configured in single-shunt or by the second motor when the first is not single-shunt; ADC1 is used for general purpose conversions
TIM1	TIM4 (CH3)	DMA1_CH1 DMA1_CH5 DMA1_CH4	TIM1_UP DMA1_CH4_TC (FOC rate > 1)	ADC1	Option2: used by the second motor configured in single-shunt when the first motor is also configured in single-shunt.

Table 10. Single-shunt current reading, used resources, single or dual drive, STM32F4xx

Adv Timer	Aux Timer	DMA	ISR	ADC	Note
TIM1	TIM5 (ch4)	DMA1, stream1, ch6; DMA2, stream4, ch6	TIM1_UP; DMA2_stream4_ TC (FOC rate>1)	ADC3	Option 1: used by first motor when it is configured in single shunt, or by second motor when the first one isn't in single shunt. ADVC1 used for general purpose conversions
TIM8	TIM4(ch2)	DMA1, stream3,ch2; DMA2, stream7, ch7	TIM8_UP; DMA2_stream7_ TC (FOC rate>1)	ADC1	Option 1: used by second motor when it is configured in single shunt and when first motor isn't in single shunt. ADVC1 used for general purpose conversions
TIM8	TIM5(ch4)	DMA1, stream1,ch6; DMA2, stream7, ch7	TIM8_UP; DMA2_stream7_ TC (FOC rate>1)	ADC3	Option 2: used by first motor when it is configured in single shunt, or by second motor when the first one isn't in single shunt. ADVC1 used for general purpose conversions
TIM1	TIM4(ch2)	DMA1, stream3,ch2; DMA2, stream4, ch6	TIM1_UP; DMA2_stream4_ TC (FOC rate>1)	ADC1	Option 2: used by second motor when it is configured in single shunt and when first motor is also in single shunt. ADVC1 used for general purpose conversions

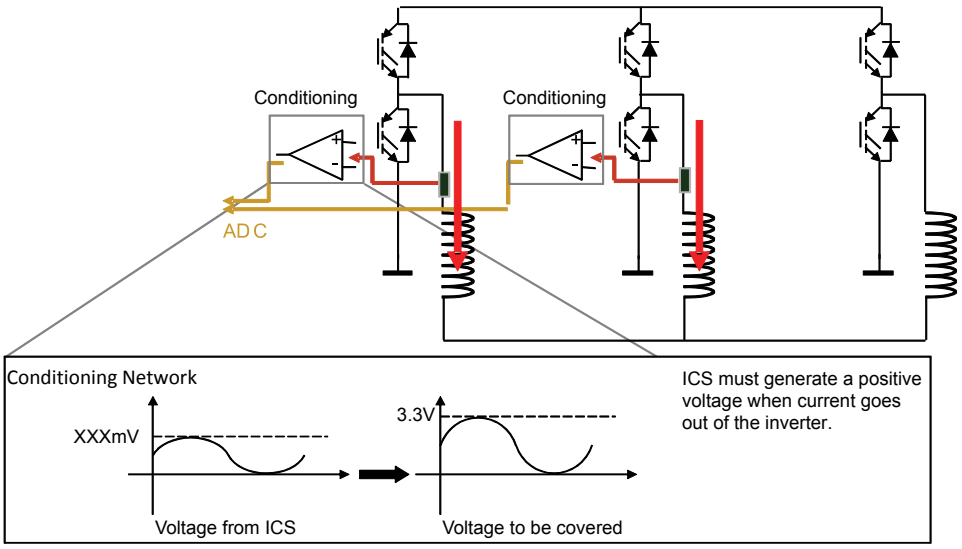
Using F103HD, or F4xx in single drive, it is possible to choose between option 1 and option 2 (See Table 9. Single-shunt current reading, used resources (single or dual drive, F103HD and Table 10. Single-shunt current reading, used resources, single or dual drive, STM32F4xx). The resources are allocated or saved accordingly.

Please refer to section Current sensing and PWM generation components for STM32F30x microcontroller configuration.

Current sampling in isolated current sensor topology

Figure 27. ICS hardware architecture illustrates the ICS topology hardware architecture.

Figure 27. **ICS hardware architecture**



To reconstruct the currents flowing through a generic three-phase load, it is therefore sufficient to sample only two out of the three currents, while the third is calculated using the above relationship.

The flexibility of the A/D converter trigger makes it possible to synchronize the two A/D conversions necessary for reconstructing the stator currents flowing through the motor with the PWM reload register updates. This is important because, as shown in Figure 28. Stator currents sampling in ICS configuration, it is precisely during the counter overflow and underflow that the average level of current is equal to the sampled current.

Refer to the microcontroller reference manual to learn more about A/D conversion triggering.

Figure 28. **Stator currents sampling in ICS configuration**

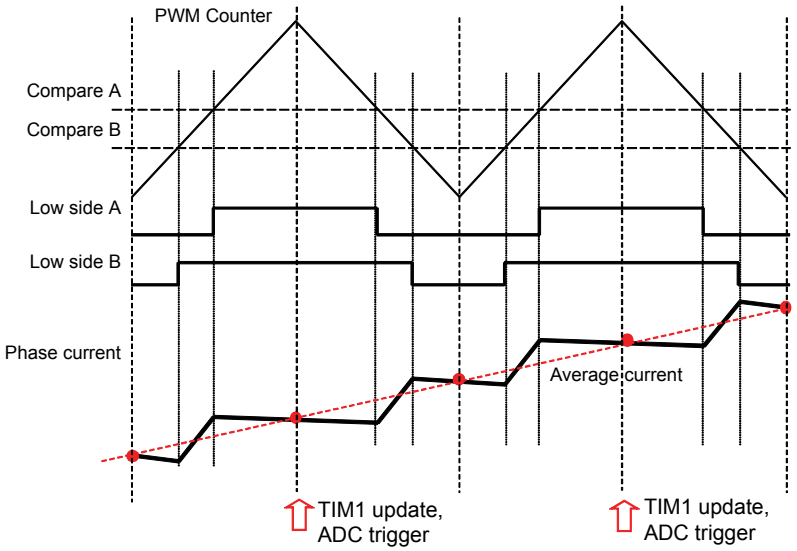


Table 11. **ICS current reading, used resources (single drive, F103 LD/MD)**

Adv. timer	DMA	ISR	ADC master	ADC slave	Note
TIM1	DMA1_CH5	None	ADC1	ADC2	DMA is used to enable ADC injected conversion external trigger. Disabling is performed by software.

Table 12. **ICS current reading, used resources (single or dual drive, F103 HD, F4xx)**

Adv. timer	DMA	ISR	ADC	Note
TIM1	None	TIM1_UP	ADC1 ADC2	Used by the first or second motors configured in three-shunt, depending on the user selection. ADC is used in time sharing. Trigger selection is performed in the TIM_UP ISR.
TIM8	None	TIM8_UP	ADC1 ADC2	Used by the first or second motor configured in three-shunt, depending on the user selection. ADC is used in time sharing. Trigger selection is performed in the TIM_UP ISR.

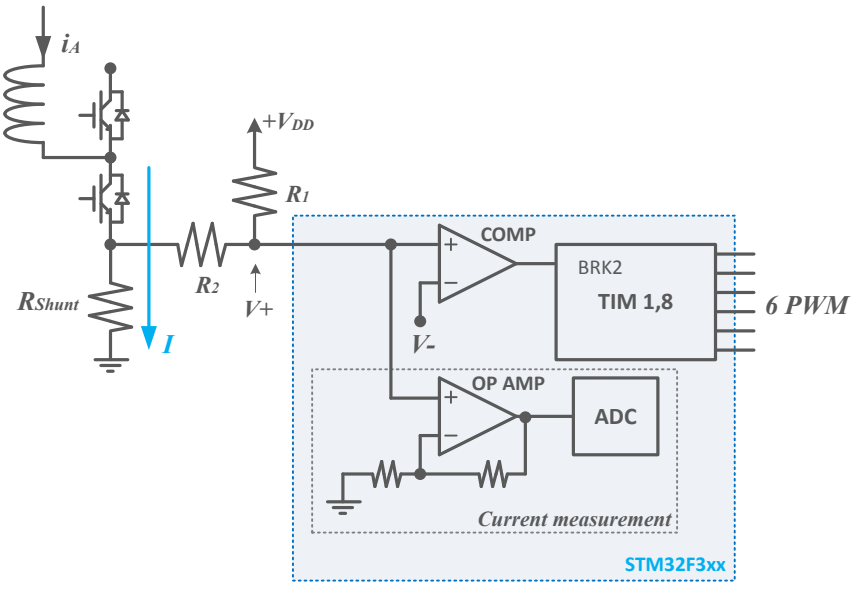
Current sensing and protection on embedded PGA

1. **Introduction**

The STM32F302xB/C or STM32F303xB/C microcontrollers feature an enhanced set of peripherals including comparators, PGAs, DACs and high-speed ADCs.

Figure 29. Current sensing network and overcurrent protection with STM32F302/303 shows a current sensing and overcurrent protection scheme that can be implemented using the internal resources of the STM32F302/303. The voltage drop on the shunt resistor, due to the motor phase current, can be either positive or negative, an offset is set by R1 and R2. The signal is linked to a microcontroller input pin that has both functionality of amplifier and comparator non-inverting.

Figure 29. **Current sensing network and overcurrent protection with STM32F302/303**



This optimized configuration using an STM32F3 reduces the number of external components and microcontroller pins assigned to the MC application.

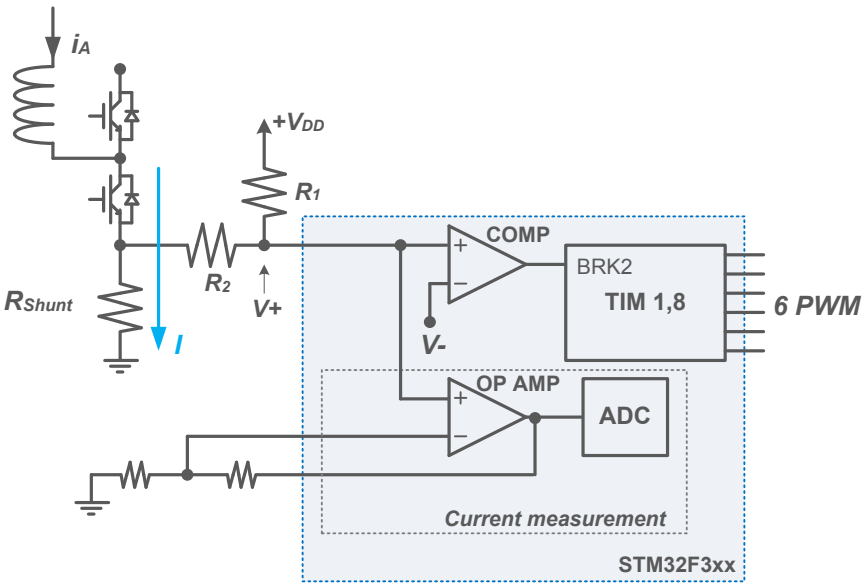
2. **Current Sensing**

In order to maximize the resolution of the measurement, the PGA can be used to adapt the level of voltage drop in the shunt resistor (R_{shunt}), caused by the motor current, up to the maximum range allowed by the analog to digital converter (ADC).

The PGA has a set of fixed internal gains (x2, x4, x8, x16) as presented in Figure 29. An alternative option in PGA mode allows you to route the central point of the resistive network on one of the I/Os connected to the non-inverting input. This feature can be used for instance to add a low- pass filter to PGA, as shown in Figure 30. Current sensing network using external gains.

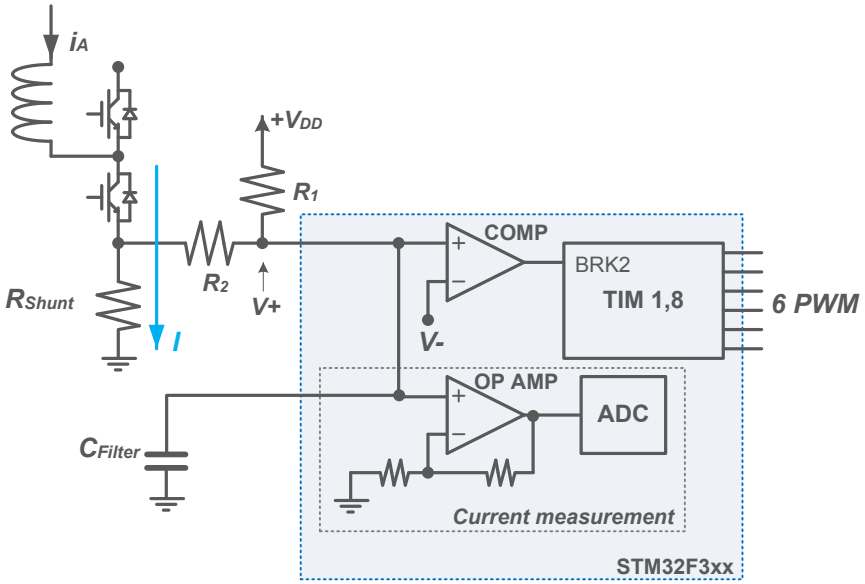
If a different value of amplification is required, it is possible to define the amplification network (for example, as shown in Figure 30.

Figure 30. Current sensing network using external gains



It is also possible to set up the motor current measurement network to use external operational amplifiers. In this case the amplified signals are directly fed to the ADC channels.

Figure 31. Current sensing network using internal gains plus filtering capacitor



The MC library can be arranged to match all the configurations shown here, thanks to the STM32 MC WB. Refer to STM32 motor control SDK v5.x tools (UM2380) for more information.

3. **Over-current Protection**

The basic principle of the hardware over-current protection mechanism can be summarized as follows:

– The phase current of the motor flows in the power transistor of the inverter bridge and passes through the shunt resistor (R_{shunt}) producing a voltage drop (V_+).

– This voltage drop is compared with a threshold (V_-) defining the maximum admissible current.

– If the threshold is exceeded, a break signal stops the PWM generation putting the system in a safe state.

All of these actions can be performed using the internal resources of the STM32F302/303 and, in particular, the embedded comparators and the advanced timer break function (BRK2). As shown in Figure 29. Current sensing network and overcurrent protection with STM32F302/303, Figure 30. Current sensing network using external gains and Figure 31. Current sensing network using internal gains plus filtering capacitor the same signal is fed to both not inverting input of embedded comparators and PGA.

The over-current threshold (V-) can be defined in three different ways:

– Using one of the available internal voltage reference (1.2V, 0.9V, 0.6V and 0.3V);

– Providing it externally using the inverting input pin of the comparator;

– Programming a DAC channel.

Here too, the STM32 MC WB allows for all these configurations when creating a project based on STM32F302xB/C or STM32F303xB/C MCUs.

On the other hand, it is possible to setup the motor over-current protection network to use external components. In this case the over-current protection signal – coming from a comparator for instance – is directly fed to the advanced-timer's BKIN2 pin.

In any case, whether using embedded comparators or external components, a digital filter, placed before the BKIN2 function, can be enabled and configured in order to reject noises.

4. **Resource allocation for Single Drive applications**

This section deals with the allocations of hardware resources for single drive applications based on the STM32F302xB/C or STM32F303xB/C MCU.

5. **Single-Shunt topology**

   Depending on the chosen configuration – see the preceding sections – 1 ADC, 1 OPAMP, 1 comparator, and or 1 DAC channel are to be assigned. Here are the conditions governing the allocation of these peripherals:

   – If the *Embedded PGA* feature is enabled, the selection of the ADC peripheral (and its input pin) is linked to this specific PGA peripheral;
   – If the *Embedded HW OCP* and *Embedded PGA* features are enabled, the ADC and the comparator (as well as their input and '+' pins) to be selected are the ones linked to the chosen PGA peripheral
   – If the *Embedded HW OCP* feature is enabled and the *Embedded PGA* feature is disabled, the selection of the comparator is free.
   – If the *Embedded HW OCP* and the *Embedded PGA* features are both disabled, the selection of the comparator and the ADC is free.
   – If both PGA and comparator for OC protection are used they will share the same input pins for the motor current measurement signal.

6. **Three-Shunt topology**

   Depending on the configuration – see the preceding sections – 2 ADCs, 2 OPAMPs, 3 comparators, 1 DAC channel must be assigned. Here are the conditions governing the allocation of these peripherals:

   – If the *Embedded PGA* feature is enabled, the selection of the ADC peripherals (and its input pins) is linked to these specific PGA peripherals;
   – If the *Embedded HW OCP* and the *Embedded PGA* features are enabled, the ADCs and comparators (and their inputs and '+" pins) to select are the ones linked to these specific PGA peripherals (and theirs '+' inputs);
   – If the *Embedded HW OCP* feature is enabled and the *Embedded PGA* feature is disabled, the selection of comparators is free;
   – If the *Embedded HW OCP* and *Embedded PGA* features are both disabled, the selection of comparators and ADCs is free;
   – The OPAMP1/OPAMP2 pair can be used in a project based on the STM32F302 or on the STM32F303; the OPAMP3/OPAMP4 pair can be used additionally in a project based on the STM32F303;
   – The ADC1/ADC2 pair can be used in a project based on the STM32F302 or on the STM32F303; the ADC3/ADC4 pair can be used additionally in a project based on the STM32F303.
   – If both PGA and comparator for OC protection are used they will share the same input pins for the motor current measurement signal.

7. **Resource allocation for Dual Drive applications**

   This section deals with dual drive applications that can be designed around an STM32F303 microcontroller. For these applications, the STM32 MC SDK supports Single-Shunt and Three-Shunt current feedback network configurations.

   Dual Single-Shunt drive, dual Three-Shunt drive and mixed Single-Shunt plus Three-Shunt drive are supported. But, the sharing of peripherals between the Single-Shunt drive and the Three-Shunt drive is not allowed, nor is the sharing of peripherals between two Single-Shunt drives.

   However, the sharing of peripherals between two Three-Shunt drive is allowed, under the conditions stated below.

8. **Single-Shunt topology**

   For each motor, depending on the chosen configuration, one ADC, OPAMP and comparator must be assigned.

   – If the *Embedded PGA* feature is enabled, the selection of ADC peripheral (and input pin) is linked to this specific PGA peripheral.
   – If the *Embedded HW OCP* and *Embedded PGA* features are enabled, the selection of ADC and comparator (and their input and '+" pins) is depends on this specific PGA peripheral (and its '+' input).
   – If the *Embedded HW OCP* is enabled and *Embedded PGA* features are disabled, the selection of comparator is free.
   – If the *Embedded HW OCP* and *Embedded PGA* features are both disabled, the selection of comparator and ADC is free.
   – If both PGA and comparator for OC protection are used they will share the same input pins for the motor current measurement signal.

9. **Three-Shunt mixed with Single-Shunt topologies**

   Depending on the configuration, 2 ADCs, 2 OPAMPs, 3 comparators, 1 DAC channel must be assigned.

   – If the *Embedded PGA* feature is enabled, the selection of ADC peripherals (and input pins) is linked to this specific PGA peripherals.
   – If the *Embedded HW OCP* and *Embedded PGA* features are enabled, the selection of ADCs and comparators (and their inputs and '+" pins) is linked to this specific PGA peripherals (and theirs '+' inputs).
   – If the *Embedded HW OCP* feature is enabled and the *Embedded PGA* feature is disabled, the selection of comparators is free.
   – If the *Embedded HW OCP* and *Embedded PGA* features are both disabled, the selection of comparators and ADCs is free.
   – The OPAMP1/OPAMP2 pair can be used in a project based on STM32F302 or STM32F303; the OPAMP3/OPAMP4 pair can be used additionally in a project based on STM32F303.
   – The ADC1/ADC2 pair can be used in a project based on STM32F302 or STM32F303; the ADC3/ADC4 pair can be used additionally in a project based on STM32F303.
   – If both PGA and comparator for OC protection are used they will share the same input pins for the motor current measurement signal.

10. **Dual Three-Shunt topology, resources not shared**

    Depending on the configuration, 4 ADCs, 4 OPAMPs, 6 comparators, 2 DAC channels must be assigned.

    – If the *Embedded PGA* feature is enabled, the selection of ADC peripherals (and input pins) is linked to this specific PGA peripherals.
    – If the *Embedded HW OCP* and *Embedded PGA* features are enabled, the selection of ADCs and comparators (and their inputs and '+" pins) is linked to this specific PGA peripherals (and theirs '+' inputs).
    – If the *Embedded HW OCP* feature is enabled and "Embedded PGA" is disabled, the selection of comparators is free.
    – If the *Embedded HW OCP* and *Embedded PGA* features are both disabled, the selection of comparators and ADCs is free.
    – The OPAMP1/OPAMP2 pair can be used in a project based on STM32F302 or STM32F303; the OPAMP3/OPAMP4 pair can be used additionally in a project based on STM32F303.
    – The ADC1/ADC2 pair can be used in a project based on STM32F302 or STM32F303; the ADC3/ADC4 pair can be used additionally in a project based on STM32F303.
    – If both PGA and comparator for OC protection are used they will share the same input pins for the motor current measurement signal.

11. **Dual Three-Shunt topology, shared resources**

    If both drives are Three-Shunts, it is possible to share the ADCs and/or the PGAs to perform the motor current measurement. Doing this implies that both drives use the same configuration for the motor current measurement signals amplification: either external operational amplifiers or embedded PGAs.

    If shared resource is selected and external operational amplifier is used, it is possible to use the pairs ADC1/ADC2 or the pairs ADC3/ADC4 for both drivers.

    If shared resource is selected and embedded PGAs are used, the following configuration is used:

    – The pair OPAMP1/OPAMP3 is used
    – OPAMP gains is only "Internal"
    – External capacitor filer is not allowed
    – Input pins are: PA5, PA7, and PB13 respectively U, V, W for motor 1 and PA1, PA3 and PB0 respectively U, V, W for motor 2.

    In this case, if the hardware over current protection is managed by internal comparators, it is mandatory to connect externally the pins PA3 with one of the pins PB14 or PD14 and connect externally the pins PA5 with one of the pins PB11 or PD11.

3.3.2 Speed and position feedback components

These components provide the speed and the angular position of the rotor of a motor (both electrical and mechanical).

Measuring the position of the rotor in real time is mandatory for the FOC strategy as the Park and inverse Park transforms use the angular electrical position of the rotor to compute the i_q and i_d or i_α and i_β current components respectively.

In addition, the rotation speed is one of the preferred control modes when driving motors. Rotor speed measurement is then crucial to close the Reference computation loop. Finally, the speed measurement is also needed for the Feed Forward algorithm.

Four implementations of the Speed and Position Feedback feature are provided by the STM32 MC firmware. Two of them use sensors embedded in some motors like Hall sensor or quadrature encoders. The other two provide estimation of the speed and the position of the rotor based on the outcome of a Luenberger observer that estimates the Back-EMF of the motor. This observer is coupled with a Phase-Locked Loop (PLL) for one and a COordinate Rotation DIgital Computer (CORDIC) for the other that reconstruct the rotor electrical angle and speed.

These four implementations are built on a generic Speed and Position Feedback component – named the Speed and Position Feedback component – which they extend and which provides the data that are common to all of them. This base component cannot be used as is since it does not constitute a complete implementation of the features. However, it provides some getter functions that are useful to other components and its factorizing of the feature's data makes it easier to navigate through the code and understand it.

Table 13. Available Speed and Position Feedback Components

Component	Description
Encoder Speed and Position Feedback	This component uses the output of a quadrature encoder to provide a measure of the speed and the position of the rotor of the motor.
Hall Speed and Position Feedback	This component uses the output of two Hall effects sensors to provide a measure of the speed and the position of the rotor of the motor.
State Observer + PLL Speed and Position Feedback	This component uses a State Observer coupled with a software PLL to provide an estimation of the speed and the position of the rotor of the motor.
State Observer + CORDIC Speed and Position Feedback	This component uses a State Observer coupled with a CORDIC (COordinate Rotation DIgital Computer) to provide an estimation of the speed and the position of the rotor of the motor.

A fifth 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, when state observer based implementations are used for closed loop mode.

3.3.3 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 to compute the power injected into the motor. It is also needed for features like the Inrush Current Limiter and the Under Voltage Protection or algorithms like the Feed Forward.

Two implementations of a bus voltage sensing component are available. One that basically uses an ADC channel and two big resistors to measure the voltage (the Resistor Divider Bus Voltage Sensor) and another one that actually does not measure anything and only report a configured value (the Virtual Bus Voltage Sensor). This latter one is useful to get an estimation of the electrical power consumed by the motor, assuming that bus voltage remains constant and equal to the configured value. Using this latter implementation with the Feed Forward algorithm cannot provide reliable results and – of course – it makes no sense with the Under Voltage Protection. However, it can be handy when prototyping or for low-cost solutions.

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 3.4.6 ADC conversions for the Application for more details.

3.3.4 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.

3.3.5 Power Measurement component

One component is available for reporting the electrical power consumed by the motor. This is the PQD Motor Power Measurement that uses measured (v_q, v_d) and (i_q, i_d) values to compute the power. This is currently the only available implementation though the STM32 MC firmware is ready for additional ones.

3.3.6 Drive Regulation components

This section presents some of the drive regulation components that are delivered with the firmware. For a 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 FOC loop and in the speed and torque controller.

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

In a single drive motor control subsystem, at least three PID components are used.

Revup

The Revup component is responsible for starting the motor. It task begins when the motor is started in open loop and ends when the current control loop can be closed. One such component is used per motor.

State Machine

The Motor Control subsystem state machine is managed by this component. See Section 3.5 Motor Control State Machine for more information on the state machine.

One such component is used per motor.

Speed and Torque Control

This component serves two purposes:

1. It produces the i_q^* current reference from the torque or speed reference submitted by the application. As such, it manages the ramps programmed by the application
2. It regulates this speed or torque reference thanks to a PID component.

One such component is used per motor.

3.4 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 FOC, reference computation and safety loops that match with 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 to 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.

3.4.1 Motor Control Cockpit main source files

This section lists the most important source files that make 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 mainly contains a function, `MX_MotorControl_Init()`, that is used by the application generated by STM32CubeMx to initialize the MC subsystem. Its companion file, **motorcontrol.h** is only useful to the CubeMX generated **main.c** file in order to get the prototype of the function it calls.

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 4.2 Motor control API for a description of this API. As such, mc_api.h is file that applications need to include 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 4.3 Motor control low level API.

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 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 4.3 Motor control low level API.

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:

- **drive_parameters.h**
- **pmsm_motor_parameters.h**
- **power_stage_parameters.h**
- **parameters_conversion.h**
- **parameters_conversion_f0xx.h**
- **parameters_conversion_f30x.h**
- **parameters_conversion_f4xx.h**
- **parameters_conversion_f10x.h**
- **parameters_conversion_f7xx.h**
- **parameters_conversion_l4xx.h**

Interrupt handling:

The motor control subsystem provides handlers for the interrupts it uses. These are defined in the following files, that depend in the chosen STM32 family:

- **stm32f0xx_mc_it.c**
- **stm32f30x_mc_it.c**
- **stm32f4xx_mc_it.c**
- **stm32f10x_mc_it.c**
- **stm32f7xx_mc_it.c**
- **stm32l4xx_mc_it.c**

More information on the handling of interruptions of the Motor Control subsystem is given in Section 3.4.3 Interrupts and Real Time aspects.

mc_tasks.c:

This file contains the implementation of the core of the MC cockpit. It contains the code of the loops described in section Section 3.2 Motor control firmware subsystem. More information on them is given below.

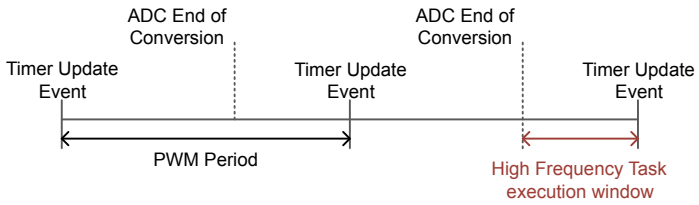
3.4.2 Tasks of the motor control subsystem

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

The **FOC 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 Section 3.3.1 Current sensing and PWM generation components). The PWM Frequency is the highest frequency in the motor control subsystem. It is executed in the handler of the interrupt that occurs when the ADC peripherals used to capture the phase current values complete their conversions. The main task of this function is to compute 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 FOC execution error.

Figure 32. High Frequency Task execution



The **Reference computation loop** is implemented in two functions, one per motor: **`TSK_MediumFrequencyTaskM1()`** and **`TSK_MediumFrequencyTaskM2()`** for motor 1 and 2 respectively (the second one is only present in a dual motor application.). These functions need 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.

Note: This calling frequency has an impact on other firmware parameters. As such, the SysTick frequency cannot be changed independently of the speed regulator parameter in the STM32 MC WB.

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

3.4.3 Interrupts and Real Time aspects

The MC firmware subsystem uses several interrupts, among which the ADC JEOS for instance. For each of these interrupts, the MC cockpit provides the handler function that is called by the NVIC thus taking a complete control over all the interrupts served by the handler even if it does not need all of them. This is done so for performance reasons.

The firmware of the STM32 MC SDK is designed to support nested interrupts (interrupts handler can be interrupted by higher priority interrupts). Interrupts priorities configured by the STM32 MC WB in the IOC file should be considered with care as the firmware relies on their ordering.

In the case where an application would require to execute in an operating system, the Reference computation and the Safety loops may be executed in an OS task context (in the same task or in separate ones) as long as the execution periodicity of these loop is respected. The FOC loop may also be executed in an OS task context but this is strongly discouraged for performance reason. Is this was absolutely necessary, this loop should be placed in the highest priority OS task and it is then mandatory that the Reference computation and Safety loops be also placed in an OS task.

In addition, it is highly recommended that no interrupt with a higher priority exist in the system. The key point here is to make sure that the FOC loop executes in the High Frequency Task execution window as defined in Figure 32. High Frequency Task execution.

3.4.4 Configuration and Parameters

Components instantiated for the MC subsystem have a handle and need to be configured. These handles are defined in the mc_config.c file. Applications that need to access these components directly simply need to include mc_config.h file to benefit from the declaration of these variables. Note that handles are variables, not constants, and are thus stored in RAM.

However, some of data placed in these handle are constant by nature. For such cases, specific structures have been created to group these constant parameters. Handles that use these contain a pointer to constant instances of such structures that are defined in the mc_parameters.c file. All elements in this file are constant and are usually placed in FLASH memory by the linker. Applications that need to access these parameter structures directly can include mc_parameters.h in order to get the declarations of these structures.

There exists a last category of parameters in the MC subsystem. It consists in C preprocessor symbols that are defined in the files listed in Motor control subsystem parameters: . These symbols are computed by the STM32 MC WB and it is hazardous to change them directly in these files.

3.4.5 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 are the following:

Table 14. Detected fault

Symbol Name	Description
MC_NO_FAULTS	No fault is currently pending on the Motor Control Subsystem
MC_FOC_DURATION	The FOC loop lasted too long (the PWM Timer update event occurred before the new PWM duty cycle values were available)
MC_OVER_VOLT	An over voltage condition was detected on the Bus
MC_UNDER_VOLT	An under voltage condition was detected on the Bus
MC_OVER_TEMP	The Temperature of the system has crossed the maximum threshold
MC_START_UP	The startup phase did end before the speed and position estimation was reliable
MC_SPEED_FDBK	The speed feedback is not reliable any more (usually happens when the rotor speed goes too low)
MC_BREAK_IN	An over current condition was detected (by the phase driver)
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 exist. 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 will remain in that state until the application acknowledges them. On the acknowledgement of the Faults, the MC state machine goes back to the **STOP** state and the subsystem is ready to start the motor again. See Section 3.5 Motor Control State Machine for a complete description of the MC state machine.

3.4.6 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 currents measurement. As described in Configuring peripherals with STM32CubeMx these ADC channels can be configured with 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 current measurements are to be made must be set with a high precision within the PWM period. In the firmware, this precision is achieved by using Injected conversions and external triggers coming from the PWM timer to start them.

Hence, the Application cannot use injected conversions on these ADC peripherals as they are reserved for motor control and they must avoid disturbing the injected conversion. The purpose of the APIs mentioned here is to allow the application to perform regular ADC conversions without disturbing the motor control subsystem. Getting a conversion done with them is a three-step process:

1. The `MC_ProgramRegularConversion()` function is called to request an ADC regular conversion on the given channel and with the given conversion time. The motor control subsystem then schedules the requested conversion that will occur right after the next Injected conversion, when there is no risk of collision;

2. The Application can then call the `MC_GetRegularConversionState()` function to determine if the requested conversion has been completed.
3. Finally, the Application calls the `MC_GetRegularConversionValue()` to retrieve the converted value.

Note: The motor control subsystem will only accept one conversion at a time. So, the application should use the `MC_GetRegularConversionState()` to determine if the conversion can be handled. In addition, all conversion requests must be performed inside routines with the that execute at the same priority level.

3.5 Motor Control State Machine

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

Figure 33. Motor state machine 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 about alignment are only useful if a quadrature encoder is used.

The state machine is never directly changed by the application. Rather, some APIs called by the Application entail changes of state. These APIs are the following ones:

- `MC_StartMotor1()` and `MC_StartMotor2()` that trigger the target motor's start procedure, switching from `IDLE` to `IDLE_START`.
- `MC_StopMotor1()` and `MC_StopMotor2()` that trigger the target motor's stop procedure, switching to `ANY_STOP`.
- `AcknowledgeFaultMotor1()` and `AcknowledgeFaultMotor2()` that acknowledge faults and makes the target motor ready to start, switching to `STOP_IDLE`.

These functions check they can perform the state switch and they fail if they cannot. They can be called from any context.

The rest of the management of the motors' state machines is handled by the Reference computation loop, that is in the `TSK_MediumFrequencyTaskM1()` and `TSK_MediumFrequencyTaskM2()` functions.

Figure 33. **Motor state machine**

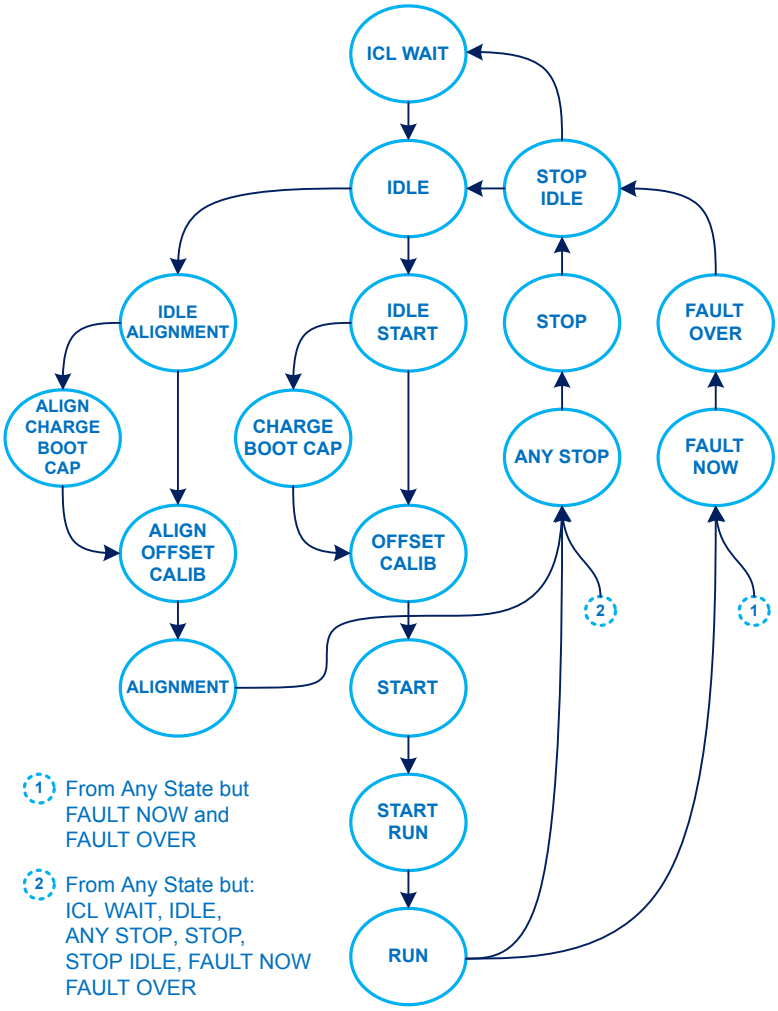


Table 15. **Motor state machine**

State	Description
ICLWAIT	The MC subsystem is waiting for Inrush Current Limiter deactivation. Is not possible to spin the motor while ICL is active.
IDLE	The Motor is not spinning, but is ready to start or to align.
IDLE_ALIGNMENT	Transition state entered on the encoder alignment command. The motor control subsystem Next states can be `ALIGN_CHARGE_BOOT_CAP` or `ALIGN_OFFSET_CALIB` according the configuration. It can also be `ANY_STOP` if a stop motor command has been given.
ALIGN_CHARGE_BOOT_CAP	Persistent state where the gate driver boot capacitors will be charged. Next states will be `ALIGN_OFFSET_CALIB`. It can also be `ANY_STOP` if a stop motor command has been given.
ALIGN_OFFSET_CALIB	Persistent state where the offset of motor currents measurements will be calibrated. Next state will be `ALIGN_CLEAR`. It can also be `ANY_STOP` if a stop motor command has been given.

State	Description
ALIGNMENT	Persistent state in which the encoder are properly aligned to set mechanical angle, following state can only be ANY_STOP.
IDLE_START	"Pass-through" state containg the code to be executed only once after start motor command. Next states can be `CHARGE_BOOT_CAP` or `OFFSET_CALIB` according the configuration. It can also be `ANY_STOP` if a stop motor command has been given.
CHARGE_BOOT_CAP	Persistent state where the gate driver boot capacitors will be charged. Next states will be `OFFSET_CALIB`. It can also be `ANY_STOP` if a stop motor command has been given.
OFFSET_CALIB	Persistent state where the offset of motor currents measurements will be calibrated. Next state will be `CLEAR`. It can also be `ANY_STOP` if a stop motor command has been given.
START	Persistent state where the motor start-up is intended to be executed. The following state is normally `START_RUN` as soon as first validated speed is detected. Another possible following state is `ANY_STOP` if a stop motor command has been executed.
START_RUN	"Pass-through" state, the code to be executed only once between `START` and `RUN` states it's intended to be here executed. Following state is normally RUN but it can also be `ANY_STOP` if a stop motor command has been given.
RUN	Persistent state with running motor. The following state is normally `ANY_STOP` when a stop motor command has been executed.
ANY_STOP	"Pass-through" state, the code to be executed only once between any state and `STOP` it's intended to be here executed. Following state is normally `STOP`.
STOP	Persistent state. Following state is normally `STOP_IDLE` as soon as conditions for moving state machine are detected.
STOP_IDLE	"Pass-through" state, the code to be executed only once between `STOP` and `IDLE` it's intended to be here executed. Following state is normally `IDLE`.
FAULT_NOW	Persistent state, the state machine can be moved from any condition directly to this state by `STM_FaultProcessing()` function. This method also manage the passage to the only allowed following state that is `FAULT_OVER`.
FAULT_OVER	Persistent state where the application is intended to stay when the fault conditions disappeared. The Following state is normally `STOP_IDLE`, state machine is moved as soon as the user has acknowledged the fault condition.

4 Application Programming Interfaces

4.1 Measurement Units

Many functions in the MC firmware take physical values as arguments. Some of these arguments are expressed with unusual measurement units in order to maximize the usage of the dynamic provided by the argument types. It is key to achieve an optimal MC precision. The following sections describe these units.

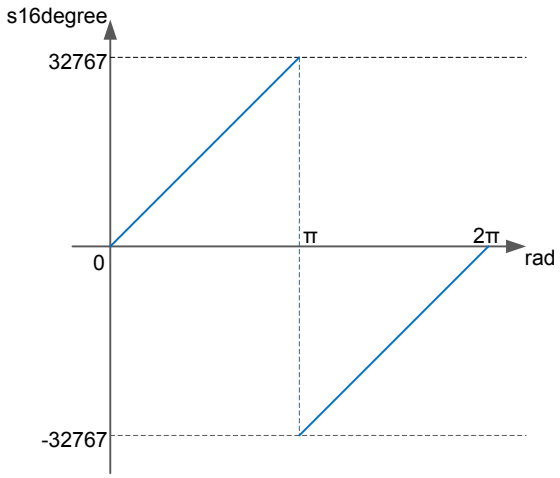
4.1.1 Rotor angle unit

The rotor angle measurement unit used in the MC API is called **s16degree**, and is defined as follows:

$$1s16\text{degree} = \frac{2\pi}{65536}rad \quad (17)$$

The following figure shows how an angle expressed in radians can be converted into the s16degree domain.

Figure 34. Radians versus s16degrees



4.1.2 Rotor speed unit

Two units are used by the MC APIs to express the rotor speed:

- Tenth of Hertz, often referred to as "**01Hz**" in the code with:

$$1dHz = 1"01Hz" = 0.1Hz \quad (18)$$

- Digit Per control Period, referred to as **dpp**: the dpp expresses the angular speed as the variation of the electrical angle (expressed in s16degree) within an FOC period.

$$1ddp = \frac{1}{T_{FOC}}s16\text{degree}/s = \frac{2\pi}{65536 \times T_{FOC}}rad/s = \frac{2\pi}{65536} \times F_{FOC}rad/s \quad (19)$$

Where T_{FOC} is the FOC period in seconds and F_{FOC} the FOC frequency in Hz.

An angular speed, expressed as the frequency in Tenth of Hertz (01Hz), can be easily converted to dpp using the formula:

$$\omega_{dpp} = \omega_{01Hz}\frac{65536}{10 \times T_{FOC}} \quad (20)$$

4.1.3 Current measurement unit

The phase current measurement unit used by the MC APIs is called **s16A** and is defined as follows:

$$1s16A = \frac{I_{MAx}}{32768} \tag{21}$$

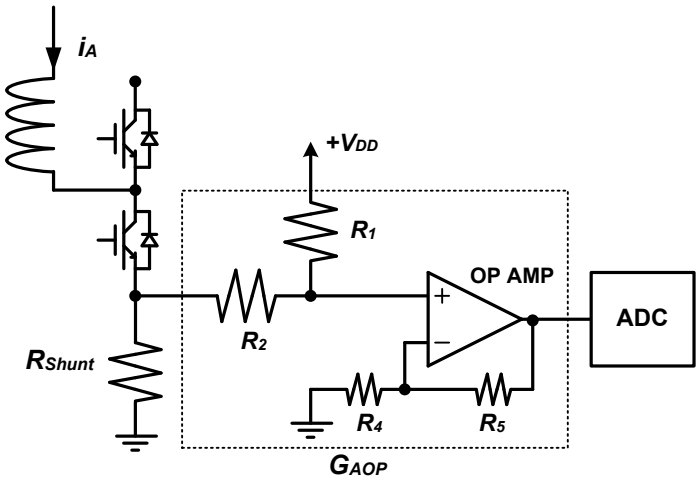
Where I_{MAX} is the Maximum measureable current. In a shunt resistor-based current sensing architecture for instance,

$$I_{MAX} = \frac{V_{DD}}{2 \times R_{Shunt} \times G_{AOP}} \tag{22}$$

Where V_{DD} is the reference voltage used for the ADC, R_{Shunt} the shunt resistor and G_{AOP} the gain of the amplification stage in front of the ADC (see Figure below). A measured current can then be converted to s16A using the formula:

$$i_{s16A} = \frac{65536 \times R_{Shunt} \times G_{AOP}}{V_{DD}} \times i_A \tag{23}$$

Figure 35. Current sensing network



4.1.4 Voltage measurement unit

The applied phase voltage unit used by the Motor Control APIs is called **s16V** and is defined as follows:

$$1s16V = \frac{V_{MAX}}{32768} \tag{24}$$

4.2 Motor control 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.

The STM32 MC SDK can drive up to 2 motors with a single STM32 MCU. For the sake of simplicity, the MC API offers one set of functions for each of the motors, restricting the number of parameters these functions expect to the bare minimum. The motor each function acts on is indicated clearly in the name of the function.

The main purpose of this API is to start, stop the motors and control their rotation. The control of the rotation of a motor is achieved by programming either a current, a torque or a speed reference that the PID regulators of the motor control subsystem will maintain. Such a reference must be set prior to starting a motor.

The current reference is programmed directly by providing direct and quadrature target current values, while the torque or speed references are programmed as ramps that move 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.

Though the current reference can be set directly as stated above, the preferred method for driving motors is to control either their speed or their torque. Which control mode is active depends on the last programmed reference. If it is a speed ramp, then the motor is controlled by the speed. Otherwise it is controlled by the torque – even if the last programmed reference is a current reference. Torque control is the default control mode.

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.

All the functions of the MC API that expect physical values as argument or that return such values express them using the units defined in Section 4.1 Measurement Units. When these physical values are two-dimension values they are embedded into dedicated structures – **Curr_Components** and **Volt_Components** – that are described below.

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.

4.2.1 Curr components

This structure is used to hold two-dimension current values such as (i_q, i_d) and (i_α, i_β) values. It is also used for the three dimension (i_a, i_b, i_c) in which case only (i_a, i_b) are stored and i_c is deduced thanks to the relation: $i_a + i_b + i_c = 0$.

Two-dimension current value structure:

```
typedef struct
{
  int16_t qI_Component1;
  int16_t qI_Component2;
} Curr_Components;
```

Usually, **Curr_Components** variables contain values expressed in the **s16A** unit. The reference documentation of each API or function in the STM32 MC firmware that uses this structure reminds which units are used.

When used for (i_q, i_d) values, **qI_Component1** holds i_q and **qI_Component2** holds i_d .

For (i_α, i_β) , **qI_Component1** holds i_α and **qI_Component2** holds i_β .

Finally, for (i_a, i_b, i_c) , **qI_Component1** holds i_a and **qI_Component2** holds i_b .

4.2.2 Volt component

This structure is used to hold two-dimension voltage values. It is used in the firmware for storing (v_q, v_d) and (v_α, v_β) values.

Usually, **Volt_Components** variables contain values expressed in the **s16V** unit. The reference documentation of each API or function in the STM32 MC firmware that uses this structure reminds which units are used.

Two-dimension voltage value structure:

```
typedef struct
{
  int16_t qV_Component1;
  int16_t qV_Component2;
} Volt_Components;
```

The assignment of the fields of the **Volt_Components** structure is similar to that of the **Curr_Components** one: **qV_Component1** is set to v_q or v_α ; **qV_Component2** is set to v_d or v_β .

4.2.3 Starting a motor

```
bool MC_StartMotor1(void);
bool MC_StartMotor2(void);
```

Starts the target Motor. Prior to calling this function, a Torque ramp, a Speed ramp or a current reference must have been set.

4.2.4 Stopping motor

```
bool MC_StopMotor1(void);
bool MC_StopMotor2(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.

4.2.5 Programming a speed ramp

```
void MC_ProgramSpeedRampMotor1(int16_t hFinalSpeed, uint16_t hDurationms );
void MC_ProgramSpeedRampMotor2( 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.

4.2.6 Programming a torque ramp

```
void MC_ProgramTorqueRampMotor1( int16_t hFinalTorque, uint16_t hDurationms );
void MC_ProgramTorqueRampMotor2( int16_t hFinalTorque, uint16_t hDurationms );
```

Programs a torque ramp on the target motor. If the target Motor is in the RUN state, the ramp is executed immediately. Otherwise, it is buffered until this state is reached.

A torque ramp takes the motor from the torque it produces at the start of the ramp to the hFinalTorque target torque of the ramp in the hDurationms duration. Note that the hFinalTorque parameter actually represents the i_q current expressed in the s16A unit.

4.2.7 Setting the current reference

```
void MC_SetCurrentReferenceMotor1( Curr_Components Iqdref );
void MC_SetCurrentReferenceMotor2( Curr_Components Iqdref );
```

Programs the current reference for the target Motor. If the target Motor is in the RUN state, the reference is immediately active. Otherwise, it is buffered until this state is reached.

4.2.8 Stopping an on-going speed ramp

```
bool MC_StopSpeedRampMotor1(void);
bool MC_StopSpeedRampMotor2(void);
```

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

4.2.9 Retrieving the status of a ramp

```
bool MC_HasRampCompletedMotor1(void);
bool MC_HasRampCompletedMotor2(void);
```

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

4.2.10 Retrieving the state of commands

```
MCI_CommandState_t MC_GetCommandStateMotor1( void);
MCI_CommandState_t MC_GetCommandStateMotor2( void);
```

Returns the state of the last submitted command for the target motor. “Command” means a speed or torque ramp or a current 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.

4.2.11 Retrieving the control mode of the motor

```
STC_Modality_t MC_GetControlModeMotor1();
STC_Modality_t MC_GetControlModeMotor2();
```

Returns the current control mode for the target motor. The returned STC_Modality_t enum value can be either STC_TORQUE_MODE for Torque of STC_SPEED_MODE for Speed.

4.2.12 Retrieving the direction of rotation of the motor

```
int16_t MC_GetImposedDirectionMotor1(void);
int16_t MC_GetImposedDirectionMotor2(void);
```

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

4.2.13 Retrieving speed sensor reliability

```
bool MC_GetSpeedSensorReliabilityMotor1(void);
bool MC_GetSpeedSensorReliabilityMotor2(void);
```

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

4.2.14 Retrieving average mechanical rotation speed of the motor

```
int16_t MC_GetMecSpeedAverageMotor1(void);
int16_t MC_GetMecSpeedAverageMotor2(void);
```

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

4.2.15 Retrieving phase current amplitude

```
int16_t MC_GetPhaseCurrentAmplitudeMotor1(void);
int16_t MC_GetPhaseCurrentAmplitudeMotor2(void);
```

Returns the amplitude of the phase current injected in the target motor, expressed in **s16A** unit.

4.2.16 Retrieving phase voltage amplitude

```
int16_t MC_GetPhaseVoltageAmplitudeMotor1(void);
int16_t MC_GetPhaseVoltageAmplitudeMotor2(void);
```

Returns the amplitude of the phase voltage applied to the target motor, expressed in **s16V** unit.

4.2.17 Retrieving electrical angle of the motor

```
int16_t MC_GetElAngledppMotor1(void);
int16_t MC_GetElAngledppMotor2(void);
```

Returns the electrical angle of the rotor of motor 1, in DDP format.

4.2.18 Motor control fault acknowledgement

```
int16_t MC_AcknowledgeFaultMotor1(void);
int16_t MC_AcknowledgeFaultMotor2(void);
```

Acknowledges MC faults pending on the target motor. This function returns **true** if faults were indeed pending and **false** otherwise. Refer to Section 3.4.5 Fault handling for more information on MC fault management.

4.2.19 Retrieving the latest motor control faults

```
int16_t MC_GetOccurredFaultsMotor1(void);
int16_t MC_GetOccurredFaultsMotor2(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 3.4.5 Fault handling for more information on MC fault management and to Section 3.5 Motor Control State Machinesection for a description of the MC state machine.

4.2.20 Retrieving all motor control faults

```
int16_t MC_GetCurrentFaultsMotor1(void);
int16_t MC_GetCurrentFaultsMotor2(void);
```

Returns a bit field showing all current faults on the target motor. Refer to section Section 3.4.5 Fault handling for more information on MC fault management.

4.2.21 Retrieving the state of the motor control state machine

```
int16_t MCI_GetSTMStateMotor1(void);
int16_t MCI_GetSTMStateMotor2(void);
```

Returns the current state of the target motor state machine. Refer to section Section 3.5 Motor Control State Machine for a description of the MC state machine and of the values of the `State_t` enumerable.

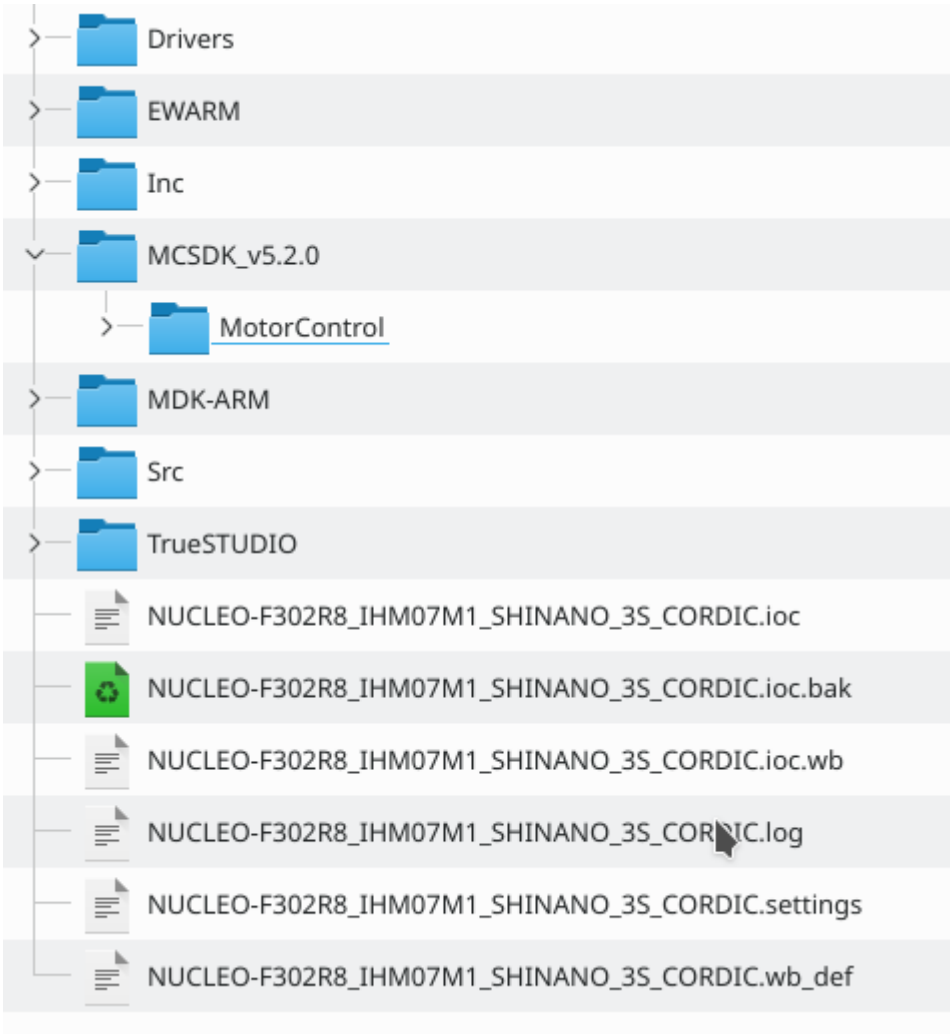
4.3 Motor control low level API

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.

5 Anatomy of a motor control project

STM32CubeMx generates a software project organized as shown in the Figure 36. Generated project disk layout.

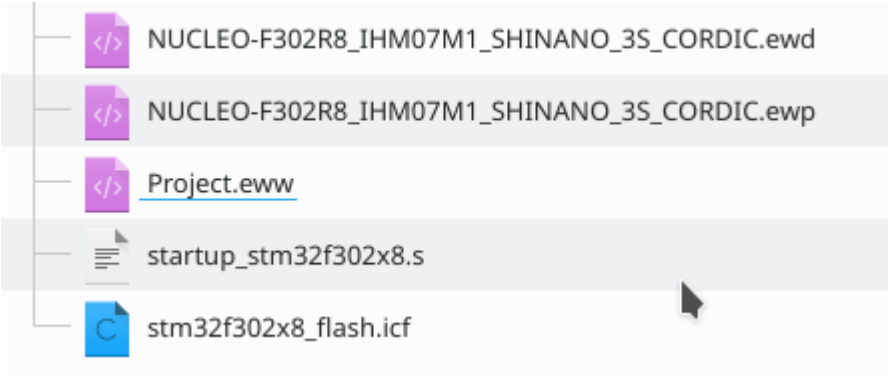
Figure 36. Generated project disk layout



The `EWARM`, `MDK-ARM` and `TrueSTUDIO` folders contain the information that describe the content of the project to the IAR EWARM, Keil µVision IDE or Atollic `TrueSTUDIO`, respectively. The STM32 MC WB / STM32CubeMx generates only one of these depending on the chosen IDE.

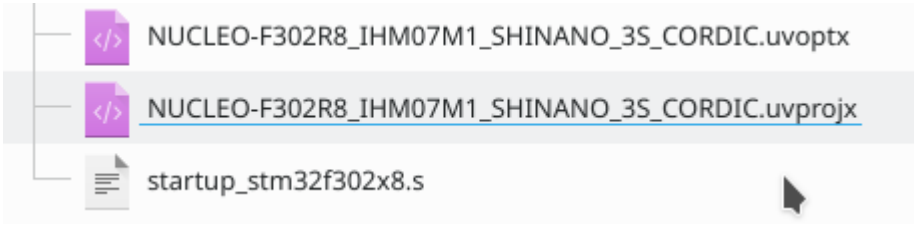
The `EWARM` folder contains a workspace file, `Project.eww` that, when open with the IAR EWARM IDE allows the project to be built, loaded in the target MCU , run and debugged (See Figure 37).

Figure 37. **EWARM project folder**



Similarly, the **MDK-ARM** folder contains an ***.uvprojx** file that allows the project to be built, loaded, run and debugged when open in Keil.

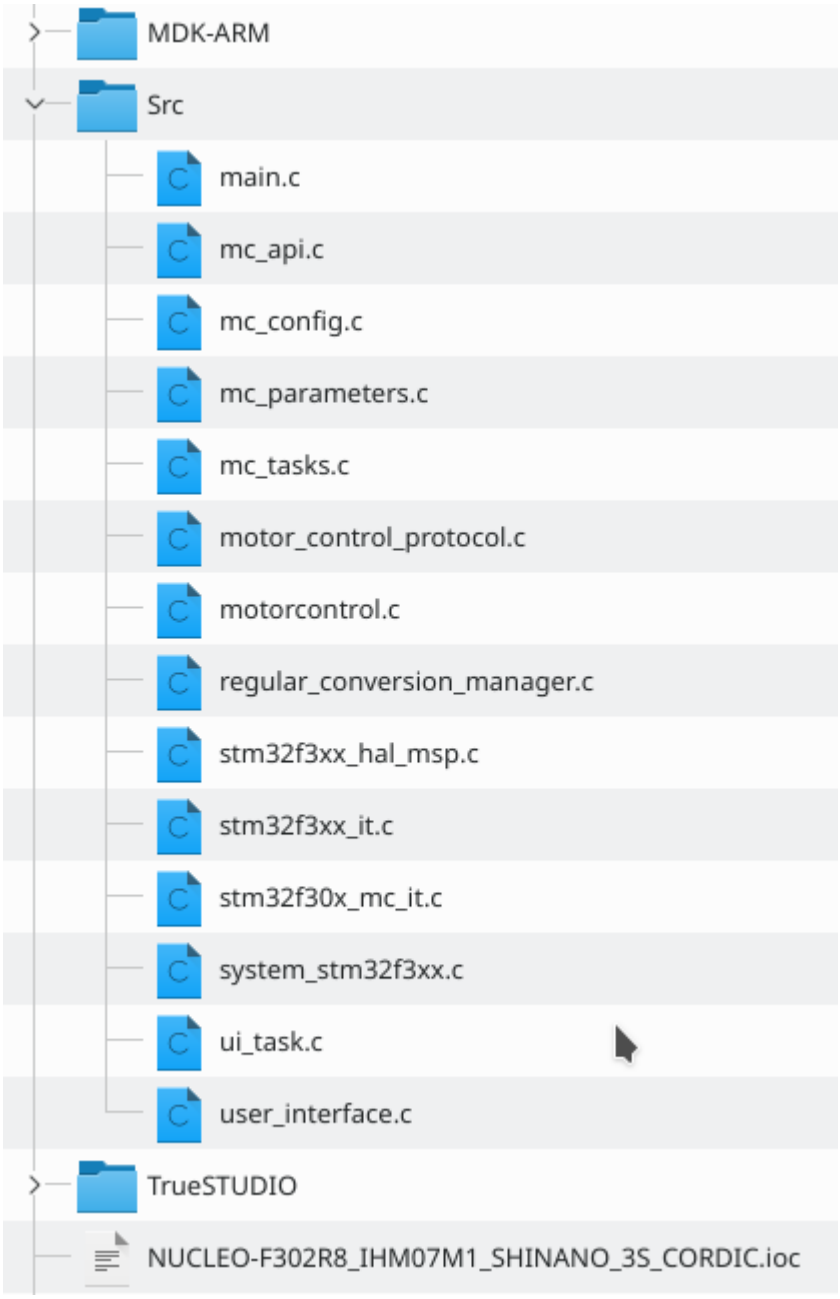
Figure 38. **Keil project folder**



Lastly, the TrueSTUDIO folder contains a sub-folder named after the project, in which a `.cproject` file can be found.

Double clicking this file opens the project in the Atollic `TrueSTUDIO` IDE which can then be built, loaded into the target MCU, run and debugged.

Figure 39. `TrueSTUDIO` project folder



The `Drivers` folder contains the STM32 HAL libraries and the CMSIS ones that are needed for the target MCU. And the `MotorControl` one hosts the code of the MC firmware components selected for the configured project.

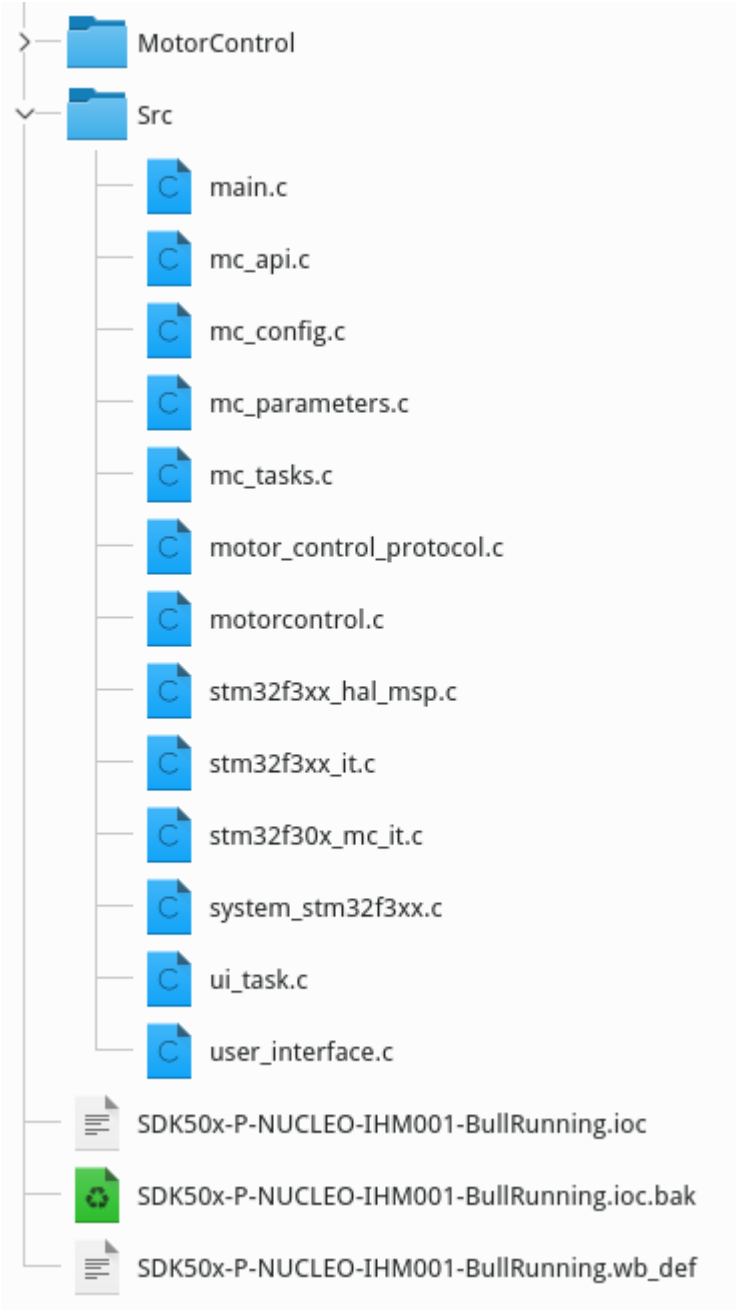
Note: This last folder resides in a folder named after the MC SDK release being used.

Usually, users do not need to modify any of the aforementioned folders directly. All the code that users may interfere with is located in the `Inc` and `Src` folders. These folders contain code that has been generated specifically for the project.

The first file of interest here is the `Src/main.c` file which main purpose is to contain the `main()` function that initializes the MCU peripherals, the MC subsystem and that runs the main application's loop. This function is the central place where the firmware aspects of the MC application would reside.

In the same folder, `Src`, are the source files generated for the MC cockpit (See Figure 40).

Figure 40. **Generated sources folder**



The example displayed here is designed for an STM32F3 MCU as can be inferred from some of the file names, but it can be easily extended to the other STM32 families.

One notable thing is the **stm32f30x_it.c** / **stm32f30x_mc_it**.c pair of files. The former is generated by STM32CubeMx and contains the definition of interrupt handling functions. This file should only contain the handlers for these interrupts that are not used by the MC firmware subsystem. Indeed, as stated in Section 3.4.3 Interrupts and Real Time aspects, the MC subsystem takes ownership of the interrupts it handles; it needs to for performance reasons.

Other noticeable files are **Inc/mc_config.h** and **Src/mc_config.c** that contain all the definitions of the handles of the components used in this MC subsystem. As explained in Section 3.4.4 Configuration and Parameters, **mc_config.h** file can be included by the application in order to access the handles of the subsystem's component – the Lower Layer API of the MC subsystem. And, as described in Section 3.4.2 Tasks of the motor control subsystem, the `Src/mc_tasks.c` file contains the implementation of the MC loops.

All these files are generated by STM32CubeMx from templates and data provided by the STM32 MC WB. Each time a change is made, either to the pure MC aspects of the project – through the Workbench – or to other aspects of the Application – through STM32CubeMx – these files get re-generated. However, it may happen that users need to modify these files.

To avoid such modifications to be lost from one generation to the other, these files contain special section – the User Sections – in which it is possible to place code that is kept across generations.

A User Section is a code fragment surrounded by special comments as shown below:

User Code Sections:

```
/* USER CODE BEGIN XXX */
User code…
/* USER CODE END XXX */
```

At the beginning of such a section a **/* USER CODE BEGIN XXX */** comment is found. **XXX** represents an identifier, unique to the file that is used to save the user code under a reference that allows it to be retrieved when the file is regenerated. The section is then terminated by a **/* USER CODE END XXX */** comment, with the same identifier.

User Code sections have been placed where they are thought to be useful. Application developers can place the code they want in these sections. STM32CubeMx guarantees that this code is kept across regenerations.

However, note that some conditions apply:

- The statements above are only valid for code inserted in User Sections originally present in the generated code. Users cannot add their own User Sections.
- Users should not move User Sections to some other places in the file. Indeed, on the next regeneration, the User Section is generated at its original place in the file.
- In the specific context of MC, it is of utmost importance to understand that some User Sections may exist only for specific configurations and that they may disappear – and so then would the code they contain – in others. A good example of this is the **/* USER CODE BEGIN M1 HALL_Update */** … **/* USER CODE END M1 HALL_Update */** section that allows users to add code in the Update event handler of the Timer used by the HALL sensor. If the configuration of the project is changed to switch to an observer based Speed and Position feedback component, this section of code is not be generated again and its content is lost.

To recover from such situations, it is strongly advised to backup or save (in a configuration management system for instance) the code prior to regeneration.

Revision history

Table 16. **Document revision history**

Date	Version	Changes
23-Oct-2018	1	Initial release.

Contents

List of tables

List of figures