UM0483
User manual

STM32F103xx
AC induction motor IFOC software library V1.0

Introduction

This user manual describes the AC induction motor IFOC software library, an indirect field oriented control (IFOC) firmware library for 3-phase induction motors developed for the STM32F103xx microcontrollers.

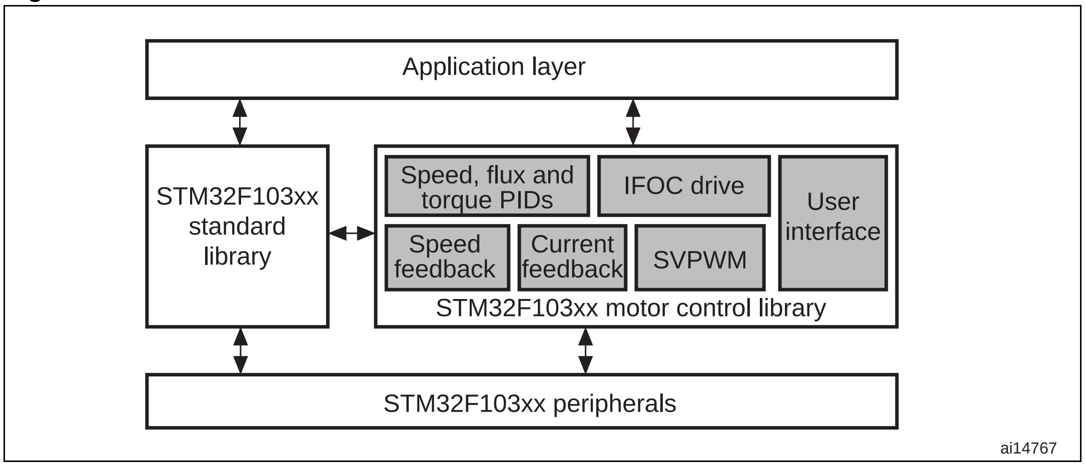
These 32-bit, ARM Cortex™-M3 cored ST microcontrollers (STM32F103xx) come with a set of peripherals which makes it suitable for performing both permanent magnet and AC induction motors FOC. In particular, this manual describes the STM32F103xx software library developed to control AC induction motors equipped with an encoder or tachogenerator, in both open and closed loop. The control of a permanent magnet (PM) motor in sinewave mode with encoder/hall sensors or sensorless is described in the UM0492 user manual.

The AC IM IFOC software library is made of several C modules, compatible with the IAR EWARM toolchain version 4.42. It is used to quickly evaluate both the MCU and the available tools. In addition, when used together with the STM32F103xx motor control starter kit (STM3210B-MCKIT) and an AC induction motor, a motor can be made to run in a very short time. It also eliminates the need for time-consuming development of IFOC and speed regulation algorithms by providing ready-to-use functions that let the user concentrate on the application layer.

A prerequisite for using this library is basic knowledge of C programming, AC motor drives and power inverter hardware. In-depth know-how of STM32F103xx functions is only required for customizing existing modules and for adding new ones for a complete application development.

Figure 1 shows the architecture of the firmware. It uses the STM32F103xx standard library extensively but it also acts directly on hardware peripherals when optimizations in terms of execution speed or code size are required.

Figure 1. Firmware architecture



Contents

List of tables

List of figures

AC IM IFOC software library V1.0 features (CPU running at 72 MHz)

- Supported speed feedback:
  - Tachogenerator
  - Quadrature incremental encoder
- Current sampling method:
  - 2 isolated current sensors (ICS)
  - 3 shunt resistors placed on the bottom of the three inverter legs
- DAC functionality for tracing the most important software variables
- Current regulation for torque and flux control:
  - PID sampling frequency adjustable up to the PWM frequency
- Speed control mode for speed regulation
- Torque control mode for torque regulation
- 16-bit space vector PWM generation frequencies:
  - PWM frequency can be easily adjusted
  - Centered PWM pattern type
  - 11 bits resolution at 17.6 kHz
- Free C source code and spreadsheet for look-up table generation
- CPU load below 30% (IFOC algorithm refresh frequency 14.4 kHz)
- Software library developed in accordance with MISRA 2004 C rules
- Code size 21 KB (three shunt resistors for current reading, tachogenerator for speed feedback) + 5 KB for LCD/joystick management

Note: *These figures are for information only; this software library may be subject to changes depending on the final application and peripheral resources. Note that it was built using robustness-oriented structures, thus preventing the speed or code size from being fully optimized.*

Related documents

Available on www.st.com:

- STM32F103xx datasheet
- '*ARM®-based 32-bit MCU STM32F101xx and STM32F103xx, firmware library*' user manual
- *'STM32F103xx Flash programming'* manual

Available on www.arm.com:

- Cortex-M3 Technical Reference Manual

1 Getting started with tools

To develop an application for an AC induction motor using the AC IM IFOC software library, you must set up a complete development environment, as described in the following sections. A PC running Windows XP is necessary.

1.1 Working environment

The AC IM IFOC software library was fully validated using the main hardware boards included in STM3210B-MCKIT starter kit (a complete inverter and control board). The STM3210B-MCKIT starter kit provides an ideal toolset for starting a project and using the library. Therefore, for rapid implementation and evaluation of the software described in this user manual, it is recommended to acquire this starter kit.

It is also recommended to install the IAR EWARM C toolchain which was used to compile the AC IM IFOC software library. With this toolchain, you do not need to configure your workspace. You can set up your workspace manually for any other toolchain. A free ‘kickstart edition’ of the IAR EWARM C toolchain with a 32Kb limitation can be downloaded from www.iar.com; it is sufficient to compile and evaluate the software library presented here.

1.2 Software tools

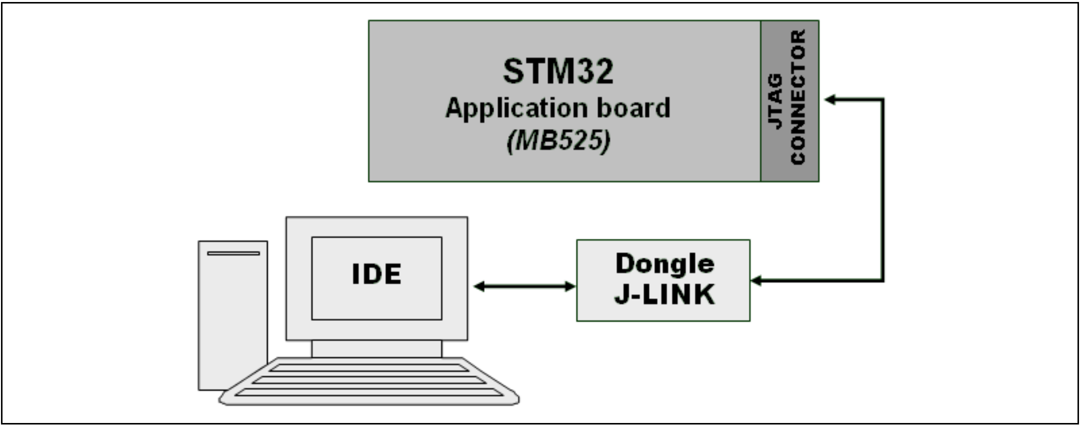
A complete software package consists of:

- A third-party integrated development environment (IDE)
- A third-party C-compiler

  This library was compiled using the third-party IAR C toolchain.
- JTAG interface for debugging and programming

  Using the JTAG interface of the MCU you can enter in-circuit debugging session with most of toolchains. Each toolchain can be provided with an interface connected between the PC and the target application.

Figure 2. JTAG interface for debugging and programming



The JTAG interface can also be used for in-circuit programming of the MCU. Other production programmers can be obtained from third-parties.

1.3 Library source code

1.3.1 Download

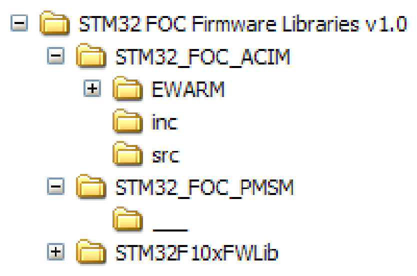
The complete source files are available for free download on the ST website (www.stmcu.com), in the Technical Literature and Support Files section, as a zip file.

Note: It is highly recommended to check for the latest releases of the library before starting any new development, and then to verify from time to time all release notes to be aware of any new features that might be of interest for your project. Registration mechanisms are available on ST web sites to automatically obtain updates.

1.3.2 File structure

The AC IM IFOC software library contains the workspace for the IAR toolchain. Once the files are unzipped, the following library structure appears, as shown in *Figure 3*.

Figure 3. File structure



```
⊟ STM32 FOC Firmware Libraries v1.0
    ⊟ STM32_FOC_ACIM
        ⊞ EWARM
          inc
          src
    ⊟ STM32_FOC_PMSM
          ___
    ⊞ STM32F10xFWLib
```

The **STM32 FOC Firmware Libraries v1.0** folder contains the firmware libraries for running 3-phase AC induction motors (with sensors) and 3-phase PMSM motors (with sensors or sensorless).

The **STM32F10xFWLib** folder contains the standard library for the STM32F103xx.

The **inc** folder contains the header and the **src** folder contains the source files of the motor control library.

Finally, **EWARM** folder contains the configuration files for the IAR toolchain.

1.3.3 Starting the IAR toolchain

When you have installed the toolchain, you can open the workspace directly from the dedicated folder, by double-clicking on the `STM32_FOC_ACIM.eww` file:

The file location is:

```
\STM32_FOC_ACIM_SR\EWARM\STM32_FOC_ACIM.eww
```

1.4 Customizing the workspace for your STM32F103xx derivative

The AC IM IFOC software library was written for the STM32F103VB. However, it works equally successfully with all the products in the STM32F103xx performance line family.

Using a different sales type may require some modifications to the library, according to the available features (some of the I/O ports are not present on low-pin count packages). Refer to the MCU datasheet for further details.

Also, depending on the memory size, the workspace may have to be configured to fit your STM32F103xx MCU derivative.

1.4.1 `Inkarm_xxx.xcl` file (internal Flash or RAM based project)

The `IAR\config` folder contains 3 files:

- `Inkarm_flash.xcl`
- `Inkarm_ram.xcl`

These files are used as an extended command linker file and contain linker options. Memory areas, start address, size, and other parameters are declared here. Refer to the Cortex-M3 Technical Reference Manual for more information.

The default extended linker file used in the standard library to configure the device for internal Flash-based resident firmware is `Inkarm_flash.xcl`. An extract of this file showing the definitions of heap and stack size is provided below. Depending on the project requirements, it may be necessary to manually edit the segment sizes.

```
...

// Code memory in FLASH
-DROMSTART=0x8000000
-DROMEND=0x801FFFF

// Data in RAM
-DRAMSTART=0x20000000
-DRAMEND=0x20004FFF

...

//*******************************************************************
********
// Stack and heap segments.
//*******************************************************************
********

-D_CSTACK_SIZE=800
-D_HEAP_SIZE=400

-Z(DATA)CSTACK+_CSTACK_SIZE=RAMSTART-RAMEND
-Z(DATA)HEAP+_HEAP_SIZE=RAMSTART-RAMEND

...
```

Memory size modifications might also be necessary according to the MCU specifications. Default settings are done for a 128 KB embedded Flash memory. If you use a different device, you must edit the `lnkarm_flash.xcl` file as explained in *Section 1.4.2*.

1.4.2 Extended linker file setting

As mentioned in the previous section, in the provided IAR workspace, the internal Flash extended linker file is set by default (`lnkarm_flash.xcl`).

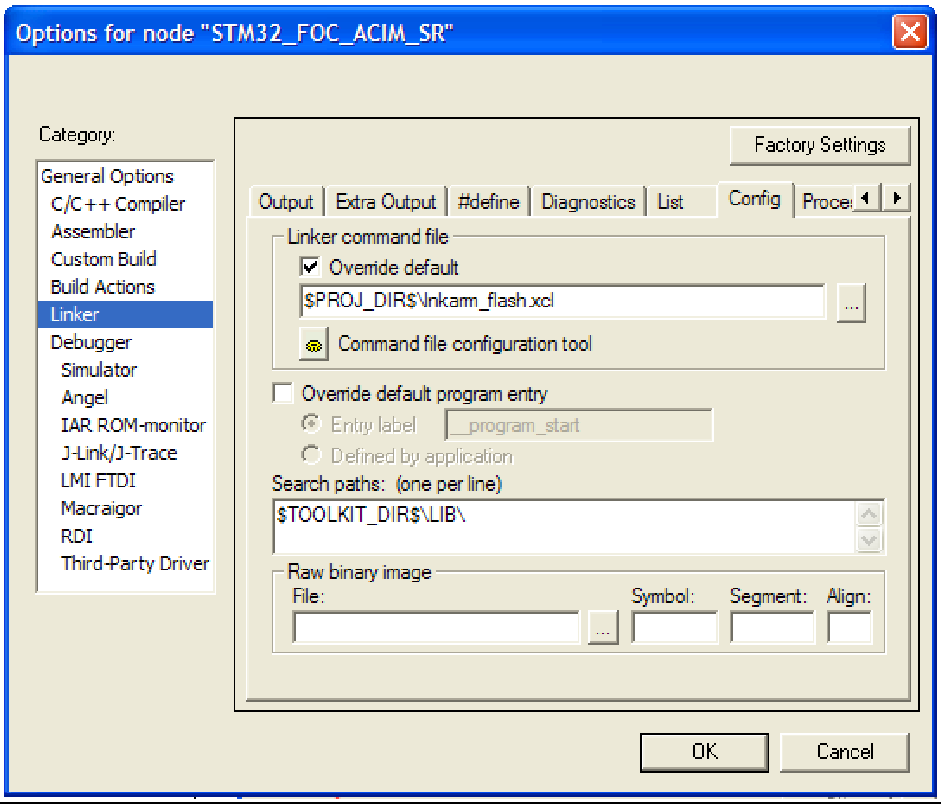
To modify the linker file to be used (for example, `lnkarm_ram.xcl`):

1. Open the IAR workspace by double-clicking on the `\STM32_FOC_ACIM_SR\EWARM\STM32_FOC_ACIM_SR.eww` file.
2. Go to the **Project** menu, select **Options...** then **Linker**, and select the **Config** sub-menu.

   The dialog box shown in *Figure 4* is displayed.
3. In the **Override default** field, type the name of the linker file you want to use, and then click `OK`.

   Selecting the `lnkarm_ram.xcl` file makes the IAR XLINK linker place the memory segments on RAM memory.

Figure 4. Extended linker file setting


Options for node "STM32_FOC_ACIM_SR"

Category:
General Options
C/C++ Compiler
Assembler
Custom Build
Build Actions
Linker
Debugger
Simulator
Angel
IAR ROM-monitor
J-Link/J-Trace
LMI FTDI
Macraigor
RDI
Third-Party Driver

Factory Settings

Output | Extra Output | #define | Diagnostics | List | Config | Proce

Linker command file
[✓] Override default
$PROJ_DIR$\lnkarm_flash.xcl
Command file configuration tool

[ ] Override default program entry
Entry label __program_start
Defined by application

Search paths: (one per line)
$TOOLKIT_DIR$\LIB\

Raw binary image
File: Symbol: Segment: Align:

OK Cancel

2 Getting started with the library

2.1 Introduction to AC induction motor FOC drive

The AC IM IFOC software library is designed to achieve the high dynamic performance in AC motor control offered by the field oriented control (FOC) strategy.

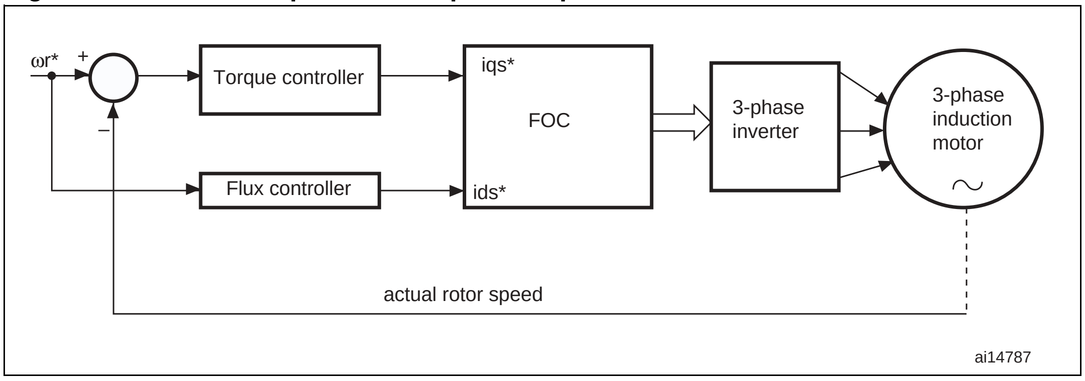
Through complex machine electrical quantity transformations, this well-established drive system optimizes the control of the motor, to the extent that it is able to offer decoupled torque (T_e) and magnetic flux (λ) regulation. That is, it offers the same optimum and favorable conditions as DC motors but, in this case, carried out with rugged and powerful AC induction motors.

With this approach, it can be stated that the two currents $i_{qs}^{\lambda r}$ and $i_{ds}^{\lambda r}$, derived from stator currents, have in AC induction motor (IM) the same role that armature and field currents have in DC motors: the first is proportional to mechanical torque the second to the rotor flux.

In more detail, in the context of FOC, rotor flux position is indirectly calculated, starting from transformed equations of the machine, by means of known motor parameters and stator current measurements: this is why the controller is an **indirect** controller and, hence the phrase IFOC drive.

In other words, it can be stated that IFOC drive is halfway between dynamic controllers (like speed and position) and machine core. So, the system may well be depicted as in *Figure 5*, if introduced in a loop for speed control.

Figure 5. FOC drive placed in a speed loop

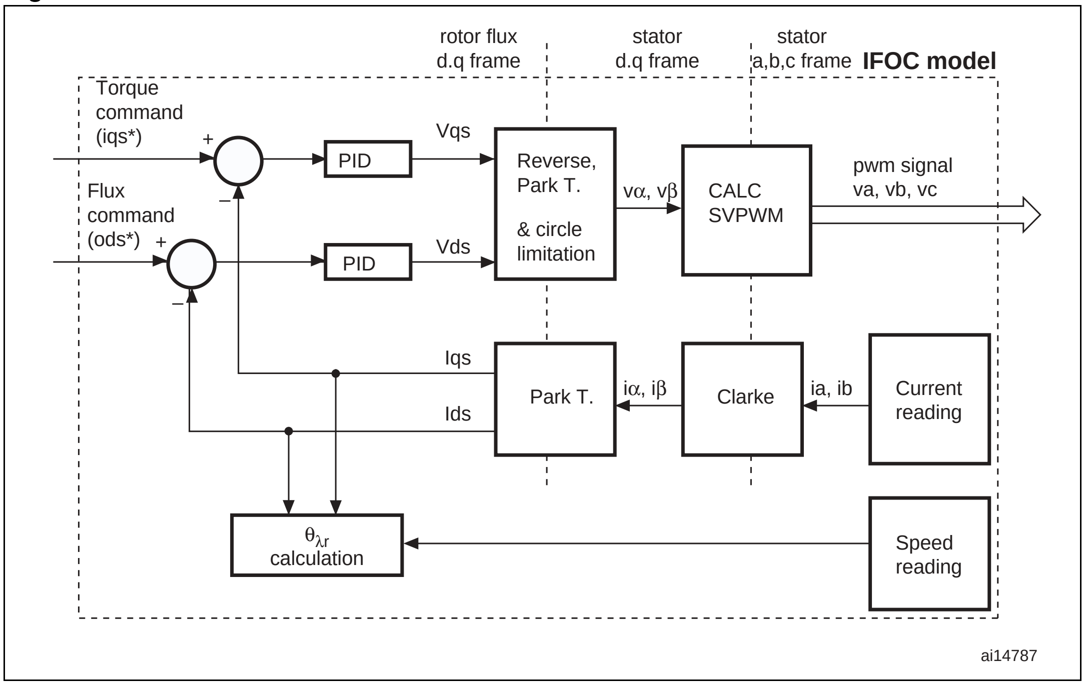


Basic information on field oriented structure and library functions is represented in *Figure 6*.

- The $\theta_{\lambda r}$ calculation block estimates rotor flux angle, which is essential to transformation blocks (park, reverse park) to perform field orientation, so that the currents supplied to the machine can be oriented in phase and in quadrature to the rotor flux vector. More detailed information about the reference frame theory and FOC structure is available in *Section 4.4.3 on page 58*.
- 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 isolated current hall sensors (ICS).
- The speed-reading block handles tachogenerator or incremental encoder signals in order to acquire properly rotor angular velocity or position.

- The PID-controller block implements a proportional, integral and derivative feedback controller, to achieve speed, torque and flux regulation.

Figure 6. FOC structure



2.2 How to customize hardware and software parameters

It is quite easy to set up an operational evaluation platform with a drive system that includes the STM3210B-MCKIT (featuring the STM32F103xx microcontroller, where this software runs) and an AC induction motor.

This section explains how to quickly configure your system and, if necessary, customize the library accordingly.

Follow these steps to accomplish this task:

1. Gather all the information that is needed regarding the hardware in use (motor parameters, power devices features, speed/position sensor parameters, current sensors transconductance);
2. Edit, using an IDE, the `stm32f10x_MCconf.h` configuration header file (as explained in more detail in *Section 2.2.1*), and the following parameter header files,
   - `MC_Control_Param.h` (see *Section 2.2.2*),
   - `MC_encoder_param.h` (see *Section 2.2.3*) or `MC_tacho_prm.h` (see *Section 2.2.4*),
   - `MC_ACmotor_prm.h` (see *Section 2.2.5*);
3. Re-build the project and download it on the STM32F103xx microcontroller.

2.2.1 Library configuration file: `stm32f10x_MCconf.h`

The purpose of this file is to declare the compiler conditional compilation keys that are used throughout the entire library compilation process to:

- Select which current measurement technique is actually in use (the choice is between three shunt or ICS sensors, according to availability).
- Select which speed/position sensor is actually performed (here the choice is between tachometer and quadrature incremental encoder, according to availability).
- Enable or disable the derivative action in the speed controller or in the current controllers in accordance with expected performance and code size.
- Enable or disable a virtual 2-channel DAC used to output in real time all the variables involved in the IFOC algorithm.

If this header file is not edited appropriately (no choice or undefined choice), you will receive an error message when building the project. Note that you will not receive an error message if the configuration described in this header file does not match the hardware that is actually in use, or in case of wrong wiring.

More specifically:

- `#define ICS_SENSORS`
  To be uncommented when current sampling is done using isolated current sensors.
- `#define THREE_SHUNT`
  To be uncommented when current sampling is performed via three shunt resistors (default).
- `#define ENCODER`
  To be uncommented when an incremental encoder is connected to the starter kit for position sensing; in parallel, fill out MC_encoder_param.h (as explained in *Section 2.2.3*);.
- `#define TACHO`
  To be uncommented when a tachogenerator is in use to detect rotor speed (default); in parallel, fill out MC_tacho_prm.h (as explained in *Section 2.2.4*);.
- `#define Id_Iq_DIFFERENTIAL_TERM_ENABLED`
  To be uncommented when differential terms for torque and flux control loop regulation (PID) are enabled;
- `#define SPEED_DIFFERENTIAL_TERM_ENABLED`
  To be uncommented when differential term for speed control loop regulation (PID) is enabled.
- `#define DAC_FUNCTIONALITY`
  If uncommented, the TIM3 timer outputs (channels 3 and 4) are configured in PWM mode to show two variables among the ones involved in the IFOC algorithm. Once this feature in enabled, an additional menu in LCD is shown for selecting the two variables to be output on the PWM channels. Refer to *Section 2.3* for more details.

Once these settings have been done, only the required blocks will be linked in the project; this means that you do not need to exclude .c files from the build.

Caution: When using shunt resistors for current measurement, ensure that the `REP_RATE` parameter (in `MC_Control_Param.h`) is set properly (see *Section 2.2.2* and *Section A.2: Selecting the update repetition rate based on PWM frequency for 3 shunt resistor configuration on page 93* for details).

2.2.2 Drive control parameters: `MC_Control_Param.h`

The `MC_Control_Param.h` header file gathers parameters related to:

- *Power device control parameters on page 16*
- *Flux and torque PID regulators sampling rate on page 16*
- *Power board protection thresholds on page 16*
- *Speed regulation loop frequency on page 17*
- *Speed controller setpoint and PID constants (initial values) on page 17*
- *Torque and flux controller setpoints and PID constants on page 18*
- *Startup torque ramp parameters on page 18*
- *Linear variation of PID constants according to mechanical speed. on page 18*

Power device control parameters

- `#define PWM_FREQ`
  Define here, in Hz, the switching frequency; in parallel, uncomment the maximum allowed modulation index definition (`MAX_MODULATION_XX_PER_CENT`) corresponding to the PWM frequency selection.
- `#define DEADTIME_NS`
  Define here, in ns, the dead time, in order to avoid shoot-through conditions.

Flux and torque PID regulators sampling rate

- `#define REP_RATE`
  Stator currents sampling frequency and consequently flux and torque PID regulators sampling rate, are defined according to the following equation:

$$\text{Flux and torque PID sampling rate} = \frac{2 \cdot \text{PWM\_FREQ}}{\text{REP\_RATE} + 1}$$

In fact, because there is no reason for either executing the IFOC algorithm without updating the stator currents values or for performing stator current conversions without running the IFOC algorithm, in the proposed implementation the stator current sampling frequency and the IFOC algorithm execution rate coincide.

Power board protection thresholds

- `#define NTC_THRESHOLD`
- `#define NTC_HYSTERIS`

These two values are used to set the maximum operating temperature rating for the power board in use (the default values refer to MB459 included in the STM3210B-MCKIT kit and correspond to a maximum temperature of about 60 °C). The second parameter is related to the hysteresis, fixed at 4 °C. When the heatsink temperature exceeds the fixed threshold, an overheat fault event is validated and the motor is stopped.

- `#define OVERVOLTAGE_THRESHOLD`
- `#define UNDERVOLTAGE_THRESHOLD`

These two values establish the minimum and maximum limits for the input bus DC voltage (the default ones refer to MB459 included in the STM3210B-MCKIT kit and correspond to 19 V for the lower voltage and 350 V for the maximum voltage). If the bus voltage exceeds the upper threshold an overvoltage fault event is validated. If the bus voltage is below the lower threshold an undervoltage fault event is triggered. If either of the two faults occurs, the motor is stopped and the display shows an error message.

Note: REP_RATE must be an odd number if currents are measured by shunt resistors (see Section A.2: Selecting the update repetition rate based on PWM frequency for 3 shunt resistor configuration on page 93 for details); its value is 8-bit long;

Speed regulation loop frequency

```
#define PID_SPEED_SAMPLING_TIME
```

The speed regulation loop frequency is selected by assigning one of the defines below:

```
#define PID_SPEED_SAMPLING_500µs  0      //min 500µs
#define PID_SPEED_SAMPLING_1ms    1
#define PID_SPEED_SAMPLING_2ms    3      //(4-1)*500µs=2ms
#define PID_SPEED_SAMPLING_5ms    9
#define PID_SPEED_SAMPLING_10ms   19
#define PID_SPEED_SAMPLING_127ms  255    //max(255-1)*500µs=127ms
```

Speed controller setpoint and PID constants (initial values)

- `#define PID_SPEED_REFERENCE`

  Define here, in 0.1Hz, the mechanical rotor speed setpoint at startup in closed loop mode;
- `#define PID_SPEED_KP_DEFAULT`

  The proportional constant of the speed loop regulation (signed 16-bit value, adjustable from 0 to 32767);
- `#define PID_SPEED_KI_DEFAULT`

  The integral constant of the speed loop regulation (signed 16-bit value, adjustable from 0 to 32767);
- `#define PID_SPEED_KD_DEFAULT`

  The derivative constant of the speed loop regulation (signed 16-bit value, adjustable from 0 to 32767);

Torque and flux controller setpoints and PID constants

- `#define PID_TORQUE_REFERENCE`

  The torque reference value, in open loop, at startup (signed 16-bit value);
- `#define PID_TORQUE_KP_DEFAULT`

  The proportional constant of the torque loop regulation (signed 16-bit value, adjustable from 0 to 32767);
- `#define PID_TORQUE_KI_DEFAULT`

  The integral constant of the torque loop regulation (signed 16-bit value, adjustable from 0 to 32767);
- `#define PID_TORQUE_KD_DEFAULT`

  The derivative constant of the torque loop regulation (signed 16-bit value, adjustable from 0 to 32767);
- `#define PID_FLUX_REFERENCE`

  Flux reference: its default value is equal to NOMINAL_FLUX defined in the `MC_ACmotor_prm.h` header file (see *Section 2.2.5*);
- `#define PID_FLUX_KP_DEFAULT`

  The proportional constant of the flux loop regulation (signed 16-bit value, adjustable from 0 to 32767);
- `#define PID_FLUX_KI_DEFAULT`

  The integral constant of the flux loop regulation (signed 16-bit value, adjustable from 0 to 32767);
- `#define PID_FLUX_KD_DEFAULT`

  The derivative constant of the flux loop regulation (signed 16-bit value, adjustable from 0 to 32767);

Startup torque ramp parameters

See *Section 3.1: Torque control mode* and *Section 3.2: Speed control mode on page 28* for details.

- `#define STARTUP_TIMEOUT`

  Define here, in ms, the overall time allowed for startup;
- `#define STARTUP_RAMP_DURATION`

  Define here, in ms, the duration of the torque ramp up;
- `#define STARTUP_FINAL_TORQUE`

  Define here, in q1.15 format, the final reference value for torque ramp up (Speed control mode only);
- `#define TACHO_SPEED_VAL`

  Define here, in 0.1Hz, the lowest speed for tachogenerator reading validation.

Linear variation of PID constants according to mechanical speed.

Refer to *Section 4.8.5: Adjusting speed regulation loop Ki, Kp and Kd vs. motor frequency on page 82*.

2.2.3 Incremental encoder parameters: `MC_encoder_param.h`

The `MC_encoder_parameter.h` header file is to be filled out if position/speed sensing is performed by means of a quadrature, square wave, relative rotary encoder (`ENCODER` defined in `stm32f10x_MCconf.h`).

- `#define ENCODER_PPR`

  Define here the number of pulses, generated by a single channel, for one shaft revolution (actual resolution will be 4x);
- `#define TIMER2_HANDLES_ENCODER`

  To be uncommented if the two sensor output signals are wired to TIMER2 input pins (default: required if using STM3210B-MCKIT);
- `#define TIMER3_HANDLES_ENCODER`

  to be uncommented if the two sensor output signals are wired to TIMER3 input pins;
- `#define TIMER4_HANDLES_ENCODER`

  To be uncommented if the two sensor output signals are wired to TIMER4 input pins.
- `#define MINIMUM_MECHANICAL_SPEED_RPM`

  Defines in rpm, the minimum speed below which the speed measurement is either not realistic or not safe in the application; an error counter is increased every time the measured speed is below the specified value.
- `#define MAXIMUM_MECHANICAL_SPEED_RPM`

  Defines in rpm, the maximum speed above which the speed measurement is either not realistic or not safe in the application; an error counter is increased every time the measured speed is above the specified value.
- `#define MAXIMUM_ERROR_NUMBER`

  Defines the number of consecutive errors on speed measurements to be detected before a fault event is generated (check rate is specified by `SPEED_MEAS_TIMEBASE in stm32f10x_Timebase.c`).
- `#define SPEED_BUFFER_SIZE`

  Defines the buffer size utilized for averaging speed measurement. Power of two is desirable for ease of computation.

2.2.4 Tachogenerator parameters: `MC_tacho_prm.h`

The `MC_tacho_prm.h` header file is to be filled out if speed sensing is performed using an AC tachogenerator. Extra details and more explanations on tacho-based speed measurement can be found in *Section 4.7 on page 71* and *Section A.4 on page 96*.

- `#define TACHO_PULSE_PER_REV`

  Define here the number of pulses per revolution given by the tachogenerator; in order to verify the correct operation of the tacho module, this parameter can be set to 1, so that the frequency measurement can be directly compared with the one of a signal generator.
- `#define TIMER2_HANDLES_TACHO`

  To be uncommented if tachogenerator-based speed measurement is performed by TIMER2. (Default: required if using STM3210B-MCKIT in conjunction with Input Capture 1 choice - see `#define TACHO_INPUT_TI1` below).

- `#define TIMER3_HANDLES_TACHO`

  To be uncommented if tachogenerator-based speed measurement is performed by TIMER3.
- `#define TIMER4_HANDLES_TACHO`

  To be uncommented if tachogenerator-based speed measurement is performed by TIMER4.
- `#define TACHO_INPUT_TI1`

  To be uncommented if sensor output signal is wired to TimerX Input Capture 1 (TIMx_CH1 pin). (Default - in conjunction with TIMER2 choice; required if using STM3210B-MCKIT).
- `#define TACHO_INPUT_TI2`

  To be uncommented if sensor output signal is wired to TimerX Input Capture 2.
- `#define MAX_SPEED_FDBK`

  This parameter defines the frequency above which speed feedback is not realistic in the application: this allows to discriminate glitches for example. The unit is 0.1Hz. By default, it is set to 6400 (640.0Hz), which corresponds to approximately 20000 RPM for a two pole pair motor.
- `#define MAX_SPEED`

  This parameter is the value returned by the function TAC_GetRotorFreqInHz if measured speed is greater than MAX_SPEED_FDBK. The default value is 640Hz, but it can be 0 or FFFF depending on how this value is managed by the upper layer software.
- `#define MAX_PSEUDO_SPEED`

  This parameter is the value returned by the function TAC_GetRotorFreq if measured speed is greater than MAX_SPEED_FDBK. The unit is rad/pwm period (2π rad = 0xFFFF). See *Section 4.7.4: Converting Hertz into pseudo frequency on page 78* for more details.
- `#define MIN_SPEED_FDBK`

  This parameter is the frequency below which speed feedback is not realistic in the application: this allows to discriminate too low frequency. This value is set to 1 Hz by default, and depends on sensor and signal conditioning stage characteristics. Typically, the tacho signal is too weak at very low speed to trigger input capture on the MCU.

Note: The MC_tacho_prm.h file includes two formulas that allow to compute the minimum sensed speed when speed is increasing (during startup) or decreasing (during motor stop).

- `#define MAX_RATIO`

  Maximum possible TIMER clock prescaler ratio:
  - This defines the lowest speed that can be measured (when counter = 0xFFFF).
  - It also prevents the clock prescaler from decreasing excessively when the motor is stopped. (This prescaler is automatically adjusted during each and every capture interrupt to optimize the timing resolution).

- `#define MAX_OVERFLOWS`

  This is the maximum number of consecutive timer overflows taken into account. It is set by default to 10: if the timer overflows more than 10 times (meaning that the tacho period has been increased by a factor of 10 at least), the number of overflows is not counted anymore. This usually indicates that information is lost (tacho time-out) or that the speed is decreasing very sharply. The corresponding duration depends on the tacho timer prescaler, which is variable; the higher the prescaler (at low speed), the longer the timeout period.
- `#define SPEED_FIFO_SIZE`

  This is the length of the software FIFO in which the latest speed measurements are stored. This stack is necessary to compute rolling averages on several consecutive data.

2.2.5 AC induction motor parameters: `MC_ACmotor_param.h`

The `MC_ACmotor_param.h` header file holds motor parameters which are essential to properly operate the IFOC vector drive.

The following parameters must be defined in all cases:

- `#define ROTOR_TIME_CONSTANT`

  Define here (in µs), the rotor open circuit time constant of the motor τ_r:

  $$\tau_r = \frac{L_r}{r_r} = \frac{L_m + L_{lr}}{r_r}$$

  where L_m is the magnetizing inductance, L_{lr} is the rotor leakage inductance, L_r is the rotor inductance, r_r is the rotor resistance.
- `#define POLE_PAIR_NUM`

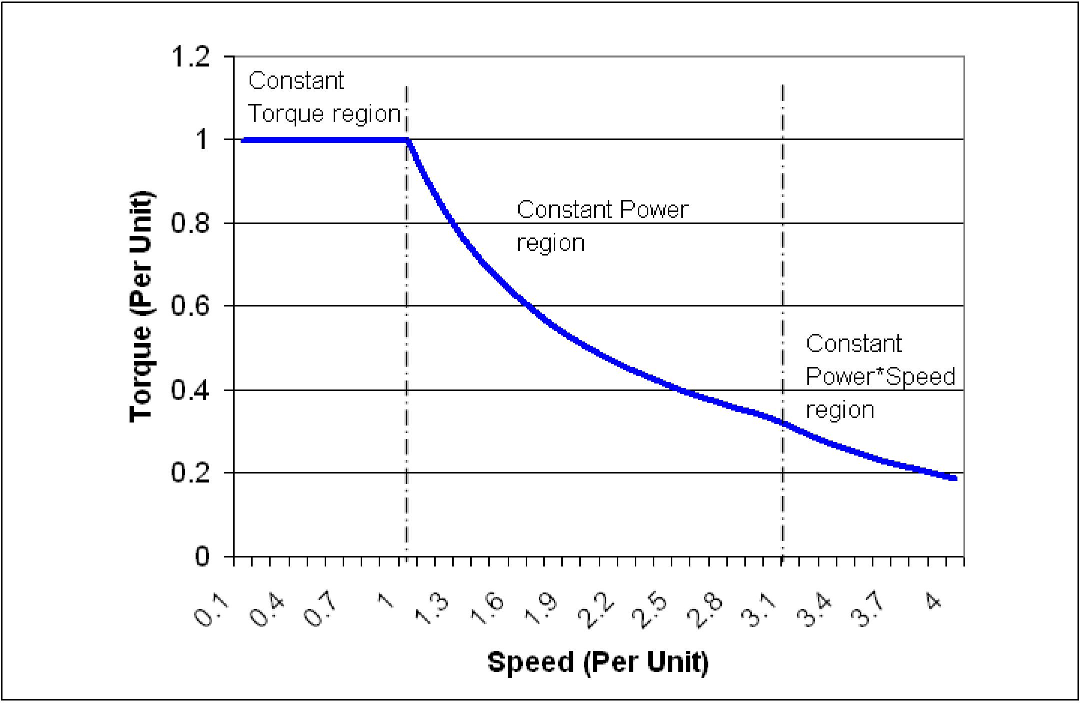
  Define here the stator winding pole pair number;
- `#define RATED_FREQ`

  Define here (in 0.1Hz) the right-hand boundary of the constant torque region (see *Figure 7*): in that region we have rated current, rated flux, rated torque, rated power.
- `#define NOMINAL_FLUX`

  Define here the required magnetizing current i_m (positive, peak value), expressed in q1.15 format (see *Section A.3 on page 95*).
- `#define NOMINAL_TORQUE`

  Defines the maximum value for the motor's rated torque expressed in q1.15 format.

Figure 7. Torque vs. speed characteristic curve



The following parameters are required only to enter the field weakening operation (constant power region begins beyond the `RATED_FREQ` boundary mentioned above):

- `#define FLUX_REFERENCE_TABLE`: this look-up table (256 signed 16-bit values) provides reference values of current i_{ds} (expressed in q1.15 format), according to increasing stator frequencies (see *Section 4.4.4 on page 60*);

Note: The first element of the table should have the same value as the `NOMINAL_FLUX` parameter.

- `TORQUE_REFERENCE_TABLE`: this look-up table (256 signed 16-bit values) provides saturation values of current i_{qs} (expressed in q1.15 format), according to increasing stator frequencies (see *Section 4.4.4 on page 60*).

2.3 How to define and add a c module

This section describes with an example how to define and include a new module in a project based on the library. The example is based on the addition of two files: *`my_file.c`* and the corresponding header file *`my_file.h`*.

1. Create a new file.

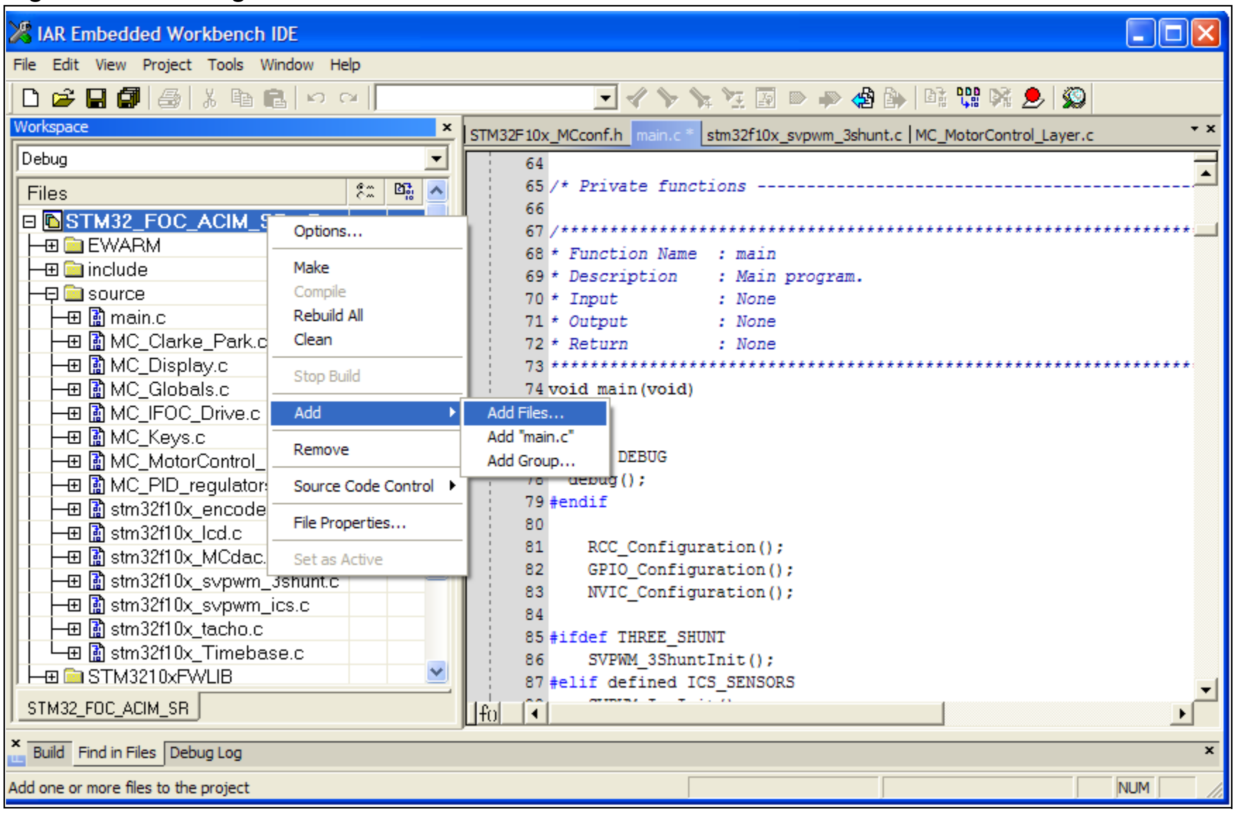
   You can either copy and paste an existing file and rename it, or in the **File** menu, choose **New**, then click the **File** icon and save it in the right format (`*.c`, `*.h` extension), as shown in *Figure 8*.

2. Declare the new file containing your code in the toolchain workspace.

   To do this, simply right-click in the workspace folder, then choose the **Add Files** sub-menu. The new file is automatically added to the workspace and taken into account for the compilation of the whole project.

The procedure of adding the module to the project is very easy with the IAR Embedded Workbench, as the makefile and linking command files are automatically generated. When rebuilding the library, the configuration files are updated accordingly.

Figure 8. Adding a new module



```
/* Private functions -------------------------------------------
/****************************************************************
* Function Name  : main
* Description    : Main program.
* Input          : None
* Output         : None
* Return         : None
*****************************************************************
void main(void)


      DEBUG
  debug();
#endif

    RCC_Configuration();
    GPIO_Configuration();
    NVIC_Configuration();

#ifdef THREE_SHUNT
    SVPWM_3ShuntInit();
#elif defined ICS_SENSORS
```

Add one or more files to the project

3 Running the demo program

This section assumes that you are using the STM3210B-MCKIT motor control kit.

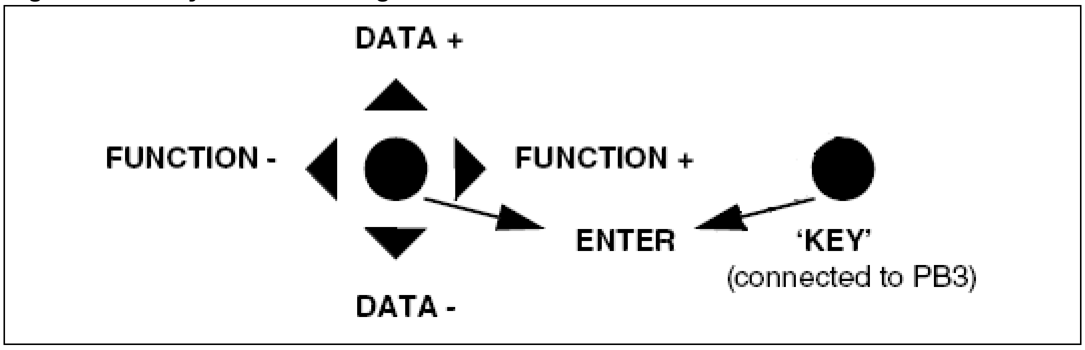
The demo program is intended to provide examples on how to use the software library functions; it includes both Torque control and Speed control operations, with the possibility of varying different parameters on the fly.

The default configuration allows the use of three shunt resistor for stator current reading and tacho generator for speed feedback. Refer to *Section 3.4 on page 32* for setting up the system when using ICS, and to *Section 3.5 on page 34* if using quadrature incremental encoder.

After the MCU initialization phase, the LCD graphical display shows the main window. Use the joystick and the button labelled **KEY** to navigate between the menus.

Key assignments are shown in *Figure 9*.

Figure 9. Key function assignments



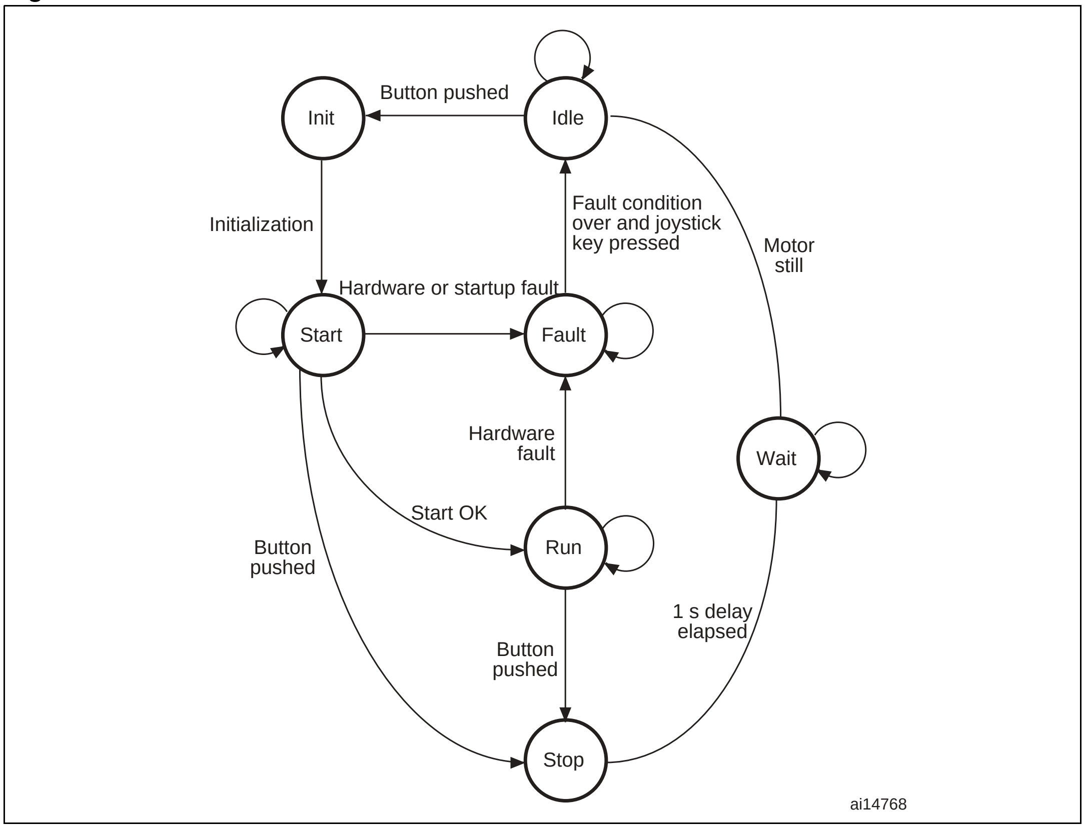
A simple state machine handles the motor control tasks in the main loop, as well as basic monitoring of the power stage. This state machine does not differentiate Torque control mode from Speed control mode. It is described in *Figure 10*.

The power stage is monitored using the ADC peripheral and the TIM1 peripheral break input (BKIN) to watch the following conditions:

- Heatsink overtemperature (ADC channel ADC_IN10 and BKIN input),
- DC bus over/undervoltage (on ADC channel ADC_IN3),
- Overcurrent protection (BKIN input).

Any of these three conditions causes the TIM1 motor control outputs (PWM signals) to be stopped and the state machine to go into Fault state until the user presses the joystick key. After the user has pressed the key, the state machine goes into Idle state (only if the cause of the fault does not persist anymore). Depending on the source of the fault, an error message is also displayed on the LCD during Fault state.

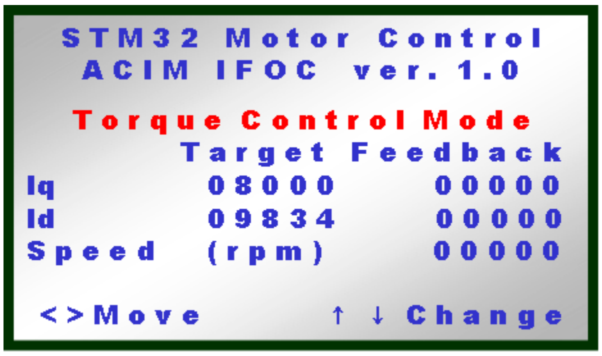
Figure 10. `Main.c` state machine



3.1 Torque control mode

Figure 11, *Figure 12* and *Figure 13* show a list of some LCD menus used to set motor control parameters in the Torque control mode. The parameter in red is the one that is being selected and whose value can be modified by acting on the joystick key.

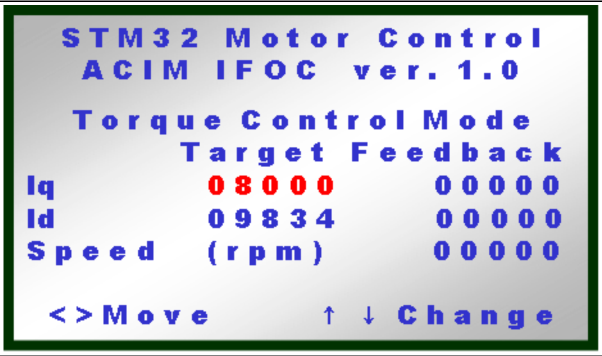
Figure 11. LCD screen for Torque control mode settings



```
STM32 Motor Control
ACIM IFOC ver. 1.0

Torque Control Mode
          Target  Feedback
Iq        08000   00000
Id        09834   00000
Speed     (rpm)   00000

<>Move           ↑ ↓ Change
```

In this condition, by moving in the joystick up/down, the active Control mode is selected (in this example, Torque control). After sending the Start motor command (by pressing the Joy or Key keys) this parameter can no longer be accessed until the motor is stopped.

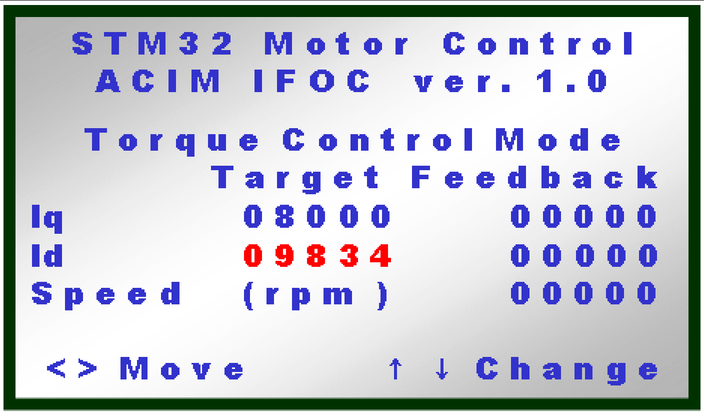
Figure 12. LCD screen for Target Iq settings



```
   STM32 Motor Control
    ACIM IFOC  ver. 1.0

   Torque Control Mode
          Target  Feedback
Iq         08000     00000
Id         09834     00000
Speed     (rpm)      00000

 <>Move         ↑ ↓ Change
```

From the previous screen (*Figure 11*), if the joystick is moved to the right, the Target Iq current component is selected. The parameter can now be changed by moving the joystick up/down. After sending the Start motor command, the Target Iq can be changed in runtime while the measured Iq current component is shown in the Feedback field.

Figure 13. LCD screen for Target Id settings

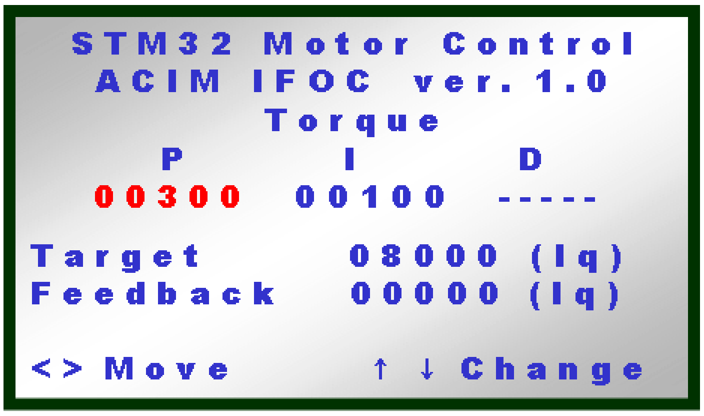


```
   STM32 Motor Control
    ACIM IFOC  ver. 1.0

   Torque Control Mode
          Target  Feedback
Iq         08000     00000
Id         09834     00000
Speed     (rpm )     00000

 <> Move        ↑ ↓ Change
```

From the previous screen (*Figure 12*), if the joystick is moved to the right, the Target Id current component becomes selected. The parameter can now be changed by moving the joystick up/down. After the Start motor command, the target can be changed in runtime while the measured Id current component is shown in the Feedback field.

Together, the Target Iq and Id currents constitute the reference inputs for two PID controllers: one to control the current component of the motor torque and the other, to control the magnetizing current component Id. Two additional menus allow the user to change up to three parameters for each PID in runtime. *Figure 14* shows an example screenshot used to enter the proportional term P of the torque controller.

Figure 14. LCD screen to set the P term of the Torque PID



```
STM32 Motor Control
ACIM IFOC ver. 1.0
Torque
P        I        D
00300    00100    -----

Target      08000 (Iq)
Feedback    00000 (Iq)

<> Move        ↑ ↓ Change
```

Finally press either the Key button or the joystick to stop the motor (main state machine moves from Run to Stop state).

The ramp up strategy is illustrated in *Figure 15*. Basically, the actual torque reference reaches the final Target Iq value (set with the joystick) in the time interval specified in the `STARTUP_RAMP_DURATION` parameter (defined in `MC_Control_Param.h`) by following a linear ramp.

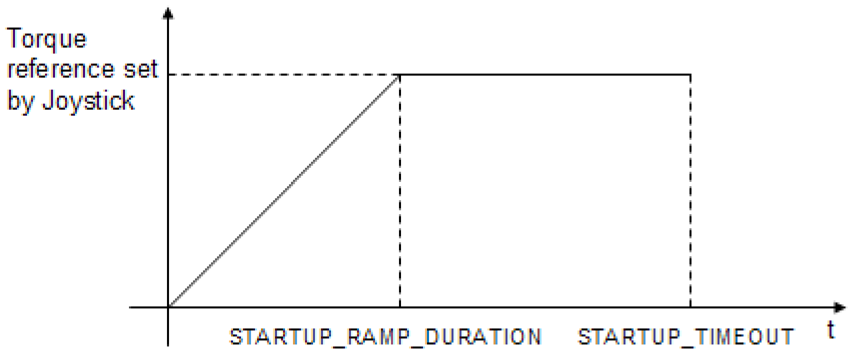
After `STARTUP_RAMP_DURATION`, if valid information from the speed sensor (tachometer or encoder) is detected, the torque reference becomes adjustable on the fly from the joystick.

On the contrary, if no valid information from the speed sensor is detected, for example because a problem occurred with speed sensor connections or because the load torque is higher then the value that you set, then the final torque reference is kept constant until `STARTUP_TIMEOUT`.

Finally, when no valid speed information comes from the motor and `STARTUP_TIMEOUT` is elapsed, the main state machine goes into FAULT state for two seconds and the error message ‘Startup failed’ is displayed on the LCD. In this case, it is strongly advised to check speed sensor feedback connections first and then, if necessary, to increase the final ramp torque reference in case the load torque is too high.

Caution: In Torque control mode operation, a constant torque reference is produced. Depending on the load torque applied, this could lead to constant acceleration of the motor, making the speed rise up to the motor’s physical limits.

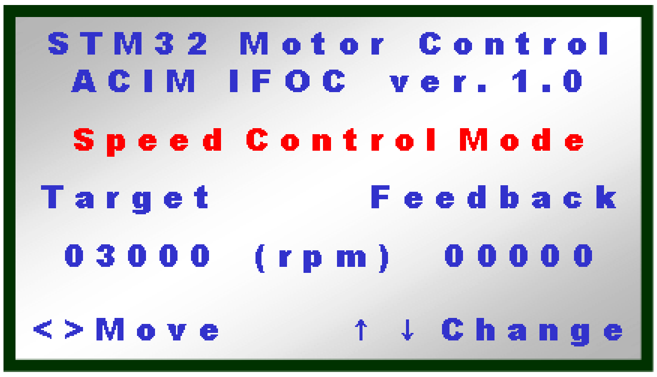
Figure 15. Torque control startup strategy



3.2 Speed control mode

Figure 16, *Figure 17*, *Figure 18* show a list of some LCD menus used to set motor control parameters in the Speed control mode. The parameter in red is the one that is being selected and whose value can be modified by acting on the joystick key.

Figure 16. LCD screen for Speed control settings

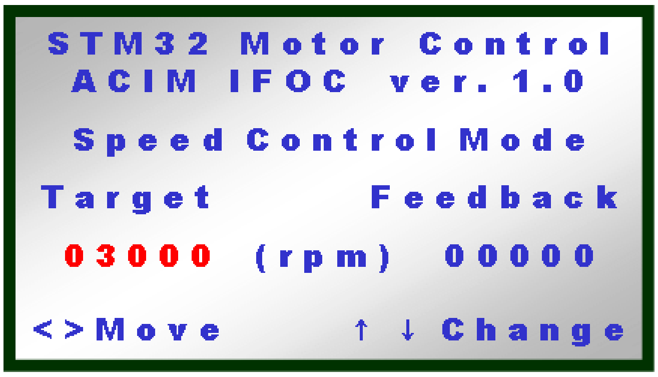


```
STM32 Motor Control
 ACIM IFOC ver. 1.0

 Speed Control Mode

Target        Feedback

03000  (rpm)  00000

<>Move       ↑ ↓ Change
```

From the menu screen shown in *Figure 11*, it is possible to switch from Torque control to Speed control mode operations (and vice versa) by moving the joystick up (or down) when the motor is stopped.

From the menu screen shown in *Figure 16*, moving the joystick to the right selects the Target speed. The parameter can then be incremented/decremented by moving the joystick up/down, respectively. The motor is then started by pressing the joystick. With the motor on, it is still possible to modify the target speed while checking its actual value in the Feedback field.

Figure 17. LCD screen to set the Target speed

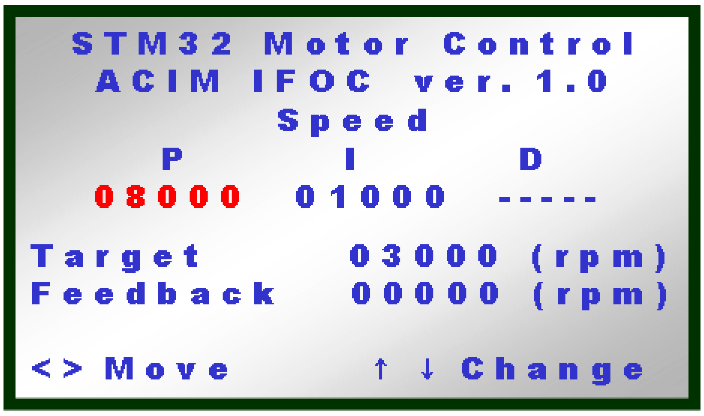


```
STM32 Motor Control
 ACIM IFOC ver. 1.0

 Speed Control Mode

Target        Feedback

03000  (rpm)  00000

<>Move       ↑ ↓ Change
```

Figure 18, *Figure 19* and *Figure 20* show the three PID screens from which the coefficient parameters can be entered. The screens also display the targets and feedbacks for each value of speed, torque and flux.

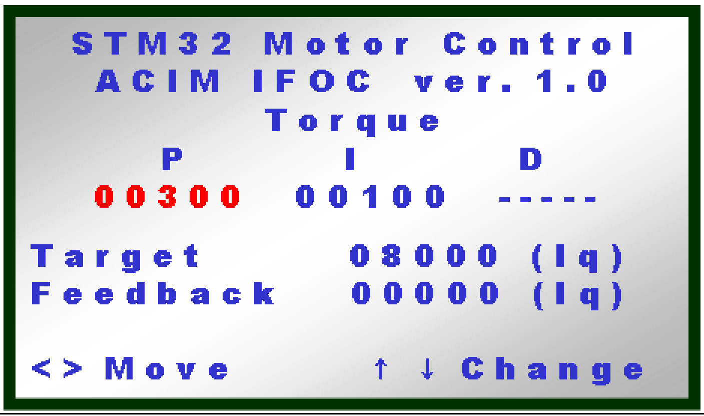
From the screens shown in *Figure 18*, *Figure 19* and *Figure 20*, move the joystick left or right to alternately select the P, I or D coefficients (when present). The selected coefficient can then be changed (incremented or decremented) by moving the joystick up/down.

Figure 18. LCD screen for setting P term of Speed PID



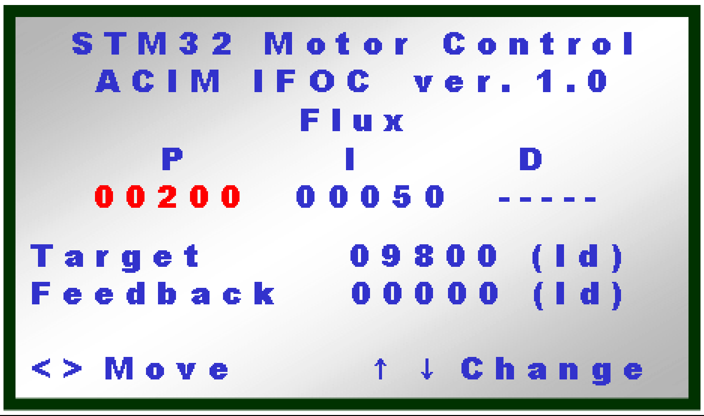
```
   STM32 Motor Control
    ACIM IFOC ver. 1.0
          Speed
     P        I        D
   08000    01000    -----

Target        03000 (rpm)
Feedback      00000 (rpm)

<> Move          ↑ ↓ Change
```

Figure 19. LCD screen to set the P term of the Torque PID



```
   STM32 Motor Control
    ACIM IFOC ver. 1.0
          Torque
     P        I        D
   00300    00100    -----

Target        08000 (Iq)
Feedback      00000 (Iq)

<> Move          ↑ ↓ Change
```

Figure 20. LCD screen to set the P term of the Flux PID



```
   STM32 Motor Control
    ACIM IFOC ver. 1.0
           Flux
     P        I        D
   00200    00050    -----

Target        09800 (Id)
Feedback      00000 (Id)

<> Move          ↑ ↓ Change
```

Finally, although you cannot directly act on torque and flux references, both the target and measured flux and torque stator current components can be observed. In fact, in closed loop, both flux and torque references are the outputs of the speed PID regulator and field weakening blocks.

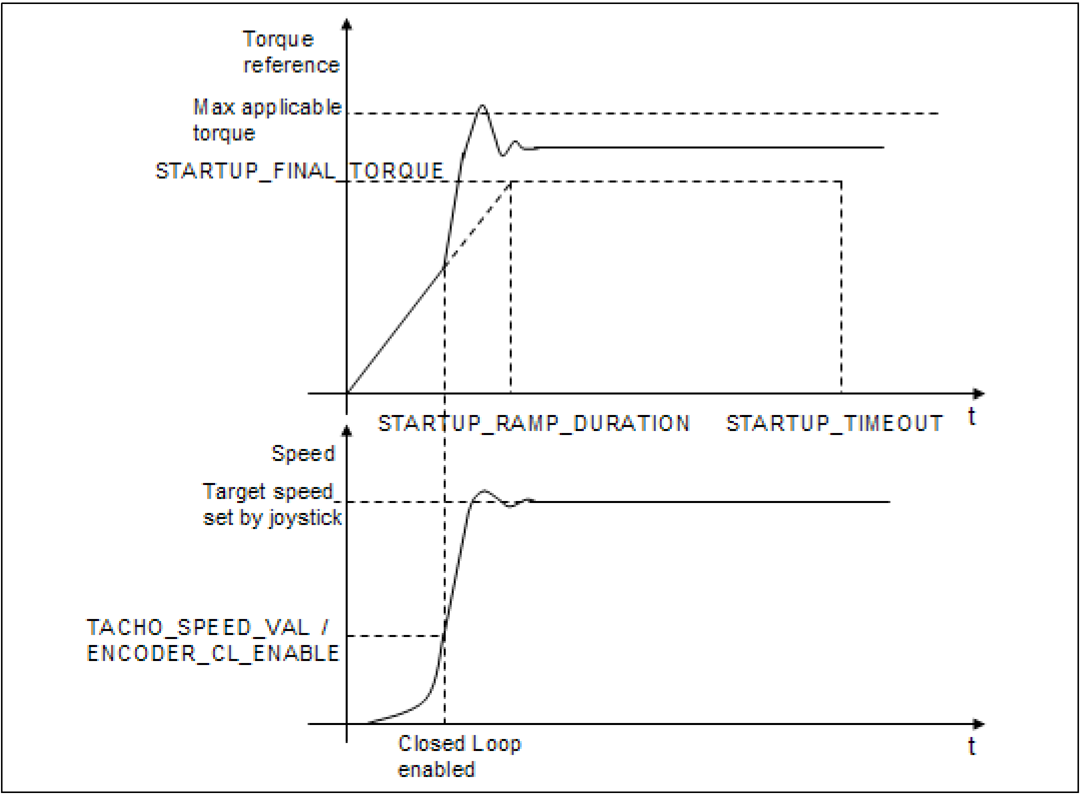
As in Torque control mode, pressing the joystick or the **Key** button starts the motor.

The Speed control ramp-up strategy is shown in *Figure 21*. Basically, a linear torque ramp is applied to the motor until it reaches speed `TACHO_SPEED_VAL` (if a tacho speed sensor is used) or `ENCODER_CL_ENABLE` (if an encoder is used). Then, the speed PID regulator is enabled and takes control of the torque reference.

However, if the motor does not reach the above mentioned speeds before `STARTUP_RAMP_DURATION`, the final torque reference value (`STARTUP_FINAL_TORQUE`) is further applied until `STARTUP_TIMEOUT`. Finally, in the case where the speeds that enable the Speed control are not reached before `STARTUP_TIMEOUT`, the state machine goes into Fault state for a few seconds and the error message **Startup failed** is displayed on the LCD. In this case, it is strongly advised to check speed sensor feedback connections first and then, if necessary, to increase `STARTUP_FINAL_TORQUE` if the load torque is too high.

With reference to *Figure 21*, note that parameters `TACHO_SPEED_VAL`, `ENCODER_CL_ENABLE`, `STARTUP_FINAL_TORQUE`, `STARTUP_RAMP_DURATION`, and `STARTUP_TIMEOUT` are fully configurable so that you can customize the startup depending on the motor and load conditions. Parameters definitions are done in the `MC_Control_Param.h` header file.

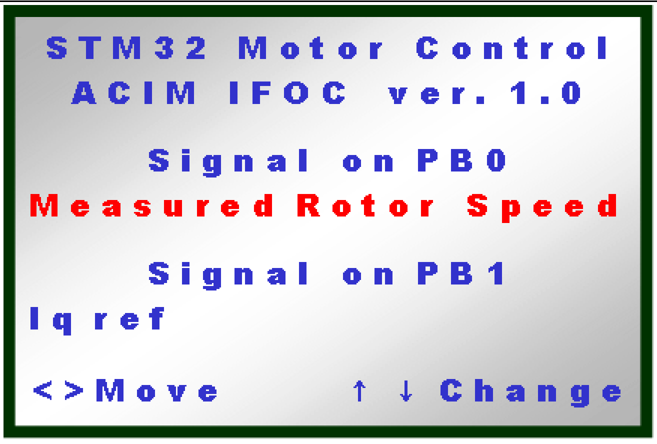
Figure 21. Speed control startup strategy



3.3 Output IFOC variables using a 2-channel DAC

In the `stm32f10x_MCconf.h` configuration file, if the DAC_FUNCTIONALITY option is uncommented, two variables can be output by using the PWM outputs of the TIM3 timer (channels 3 and 4 on the PB0 and PB1 microcontroller pins). The PWM duty cycles are then mapped to the actual value of the variable the user would like to see. By using an external RC low-pass filter, the averaged value of the PWM signal will represent the digital value of the software variable, reproduced by a scope in real time. This advanced feature makes it easier to debug the IFOC algorithm efficiency, as the actual values processed by the firmware can be graphically traced in the time domain at the current control loop frequency. Moreover, when the DAC functionality is enabled, it is possible to choose the two variables to check by using a menu screen on the LCD display. *Figure 22* shows the corresponding menu screen.

Figure 22. LCD screen to select and view IFOC variables


STM32 Motor Control
ACIM IFOC ver. 1.0

Signal on PB0
Measured Rotor Speed

Signal on PB1
Iq ref

<>Move ↑ ↓ Change

As usual, by pushing on the joystick key (to the right/left), it is possible to navigate between the two DAC channels PB0 and PB1, and then, to change the variables to be output by moving the key up/down.

The list below shows all the IFOC values that can be selected for each channel:

- Measured rotor speed (measured rotor speed)
- Measured el. angle (measured electrical angle position of rotor flux)
- V_β (Reverse Parke voltage component)
- V_α (Reverse Parke voltage component)
- V_d (flux PID controller output voltage component)
- V_q (torque PID controller output voltage component)
- I_d ref (flux command)
- I_q ref (torque command)
- I_d (measured flux component)
- I_q (measured torque component)
- I_β (Clarke current component)
- I_α (Clarke current component)
- I_b (stator current component b)
- I_a (stator current component a)

For the meaning of each value, please, refer to *Section 2.1* where the theory of the IFOC algorithm is treated.

3.4 Setting up the system when using ICS sensors

The default configuration provides for the use of three shunt resistors and tacho-generator. *Section 3.4.1* describes how to change the firmware configuration from three shunt resistors to two ICS stator current reading. This section gives you information about how to provide the STM32F103xx with ICS feedback signals and to properly customize the firmware.

Caution: When using two ICS for stator current reading, you must ensure that the conditioned sensors output signal range is compatible with the STM32F103xx supply voltage.

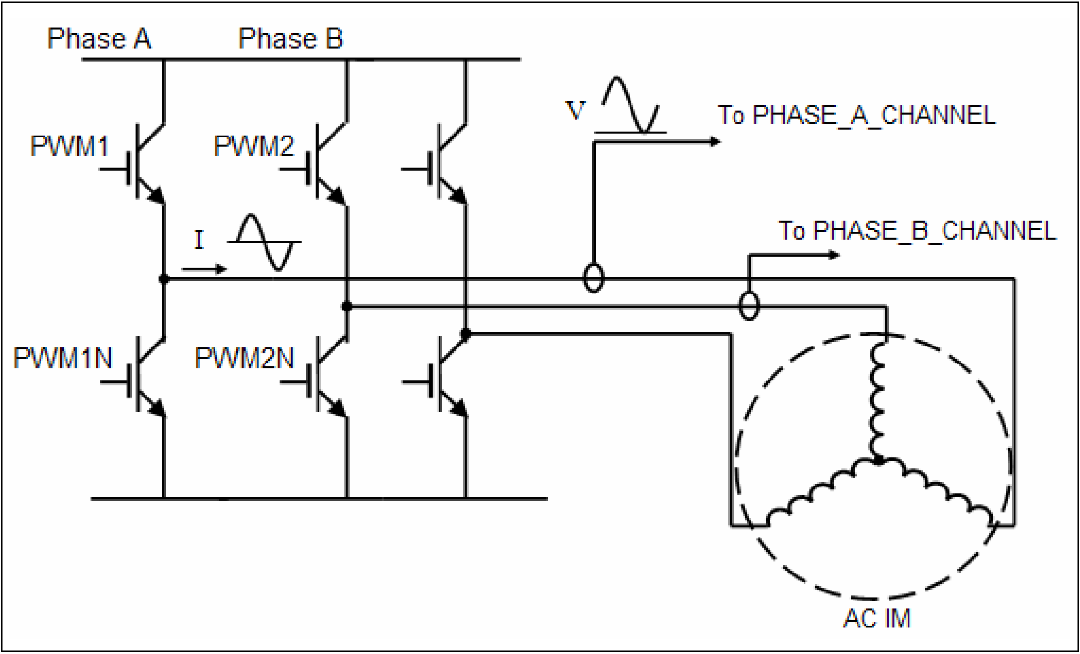
3.4.1 Connecting the two ICS sensors to the motor and to STM32F103xx

In order for the implemented IFOC algorithm to work properly, it is necessary to ensure that the software implementation of the `stm32f10x_svpwm_ics` module and the hardware connections of the two ICS are consistent.

As illustrated in *Figure 23*, the two ICS must act as transducers on motor phase currents coming out of the inverter legs driven by STM32F103xx TIM1 signals PWM1 (Phase A) and PWM2 (Phase B). In particular, the current coming out of inverter Phase A must be read by an ICS whose output has to be sent to the analog channel specified by the PHASE_A_ADC_CHANNEL parameter in `MC_pwm_ics_prm.h`. Likewise, the current coming out of inverter Phase B must be read by the other ICS and its output has to be sent to the analog channel specified by the PHASE_B_ADC_CHANNEL parameter in `MC_pwm_ics_prm.h`.

About the positive current direction convention, a positive half-wave on PHASE_X_ADC_CHANNEL is expected, corresponding to a positive half-wave on the current coming out of the related inverter leg (see direction of I in *Figure 23*).

Figure 23. ICS hardware connections



3.4.2 Selecting PHASE_A_ADC_CHANNEL and PHASE_B_ADC_CHANNEL

Default settings for PHASE_A_ADC_ CHANNEL and PHASE_B_ADC_CHANNEL are respectively `ADC_CHANNEL11` and `ADC_CHANNEL12`. You can change the default settings if the hardware requires it by editing the `MC_pwm_ics_prm.h` file.

```
///////////////////////////// Current reading parameters
//////////////////////

#define PHASE_A_ADC_CHANNEL     ADC_Channel_11
#define PHASE_A_GPIO_PORT       GPIOC
#define PHASE_A_GPIO_PIN        GPIO_Pin_1

#define PHASE_B_ADC_CHANNEL     ADC_Channel_12
#define PHASE_B_GPIO_PORT       GPIOC
#define PHASE_B_GPIO_PIN        GPIO_Pin_2
```

Caution: The proper GPIOs must be initialized as analog inputs.

An example for ADC channel 8 is given below:

```
/* ADC Channel 8 pin configuration */
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

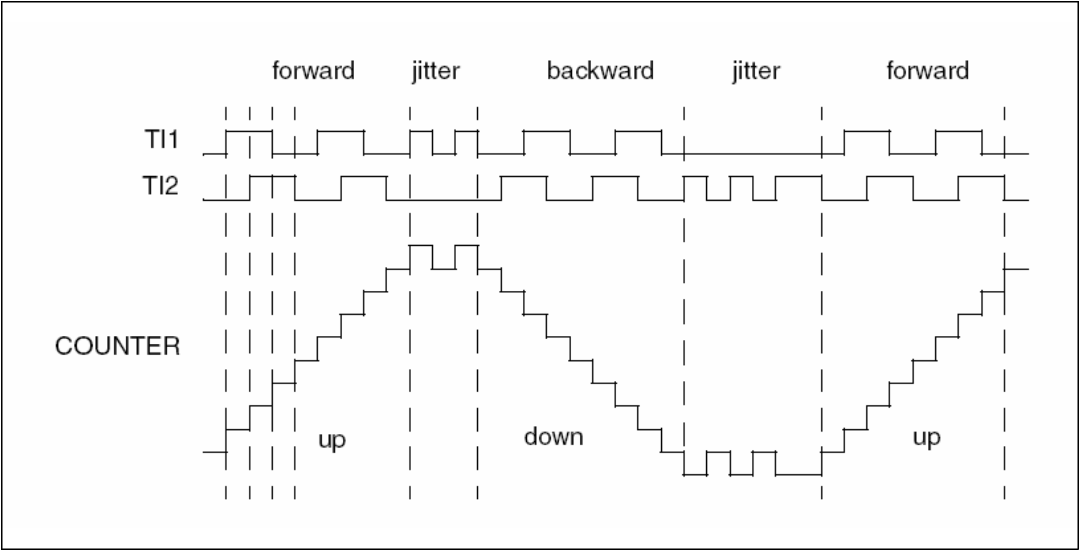
3.5 How to build the system when using an incremental encoder

Quadrature incremental encoders are widely used to read the rotor position of electric machines.

As the name implies, incremental encoders actually read angular displacements with respect to an initial position: if that position is known, then rotor absolute angle is known too.

Quadrature encoders have two output signals (represented in *Figure 24* as TI1 and TI2). With these, and with the STM32F103xx standard timer in encoder interface mode, it is possible to get information about rolling direction.

Figure 24. Encoder output signals: counter operation



In addition, rotor angular velocity can be easily calculated as a time derivative of angular position.

To set up the AC IM IFOC software library for use with an incremental encoder, simply modify the `stm32f10x_MCconf.h` and `MC_encoder_param.h` header files according to the indications given in *Section 2.2.1 on page 15* and *Section 2.2.3 on page 19* respectively.

However, some extra care should be taken, concerning what is considered to be the positive rolling direction: this software library assumes that the positive rolling direction is the rolling direction of a machine that is fed with a three-phase system of positive sequence.

Because of this, and because of how the encoder output signals are wired to the microcontroller input pins, it is possible to have a sign discrepancy between the real rolling direction and the direction that is read. To avoid this kind of reading error, you can apply the following procedure:

1. Set the DC source at low voltage (50V).
2. Run the system in closed loop operation, and on the LCD, observe the target and measured speeds.

   The error occurs if the sign of the measured speed is opposite to the sign of the target speed. (For help with the LCD menus see *Section 3.2 on page 28*):.
3. If the error occurs, you can correct it by simply swapping and rewiring the encoder output signals.

   If this isn't practical, you can modify a software setting instead: in the `stm32f10x_encoder.c` file, replace the code line:

   ```
   TIM_EncoderInterfaceConfig(ENCODER_TIMER, TIM_EncoderMode_TI12,
   TIM_ICPolarity_Rising, TIM_ICPolarity_Rising);
   ```

by

```
TIM_EncoderInterfaceConfig(ENCODER_TIMER, TIM_EncoderMode_TI12,
TIM_ICPolarity_Rising, TIM_ICPolarity_Falling);
```

3.6 Fault messages

This section provides a list of possible fault message that can be displayed on the LCD when using the software library together with the STM3210B-MCKIT:

- **Overcurrent**

  A Break Input was detected on the TIM1 peripheral dedicated pin. If using STM3210B-MCKIT it could mean that either the hardware overtemperature protection or the hardware overcurrent protection were triggered. Refer to the STM3210B-MCKIT user manual for details
- **Overheating**

  An overtemperature was detected on the dedicated analog channel; the digital threshold `NTC_THRESHOLD` and the relative hysteresis (`NTC_HYSTERESIS`) are specified in the `MC_Control_Param.h` header file. Refer to the STM3210B-MCKIT user manual for details
- **Error on speed feedback**

  The speed feedback timed out. Verify speed sensor connections
- **Startup failed**

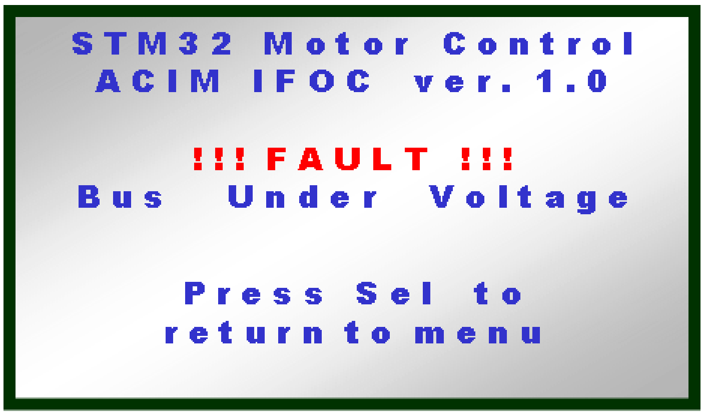
  The motor ramp-up failed. Refer to *Section 3.1* and *Section 3.2* for in-depth information,
- **Bus overvoltage**

  An overvoltage was detected on the dedicated analog channel. The digital threshold (`OVERVOLTAGE_THRESHOLD`) is specified in the `MC_Control_Param.h` header file. Refer to the STM3210B-MCKIT user manual for details.
- **Bus undervoltage**

  The bus voltage is below 20 V DC. This threshold is specified in the `UNDERVOLTAGE_THRESHOLD` parameter in the `MC_Control_Param.h` header file.

Figure 25 shows the example of the error message screen that appears when the applied DC bus voltage level is too low.

Figure 25. Error message shown in case of an undervoltage fault



STM32 Motor Control
ACIM IFOC ver. 1.0

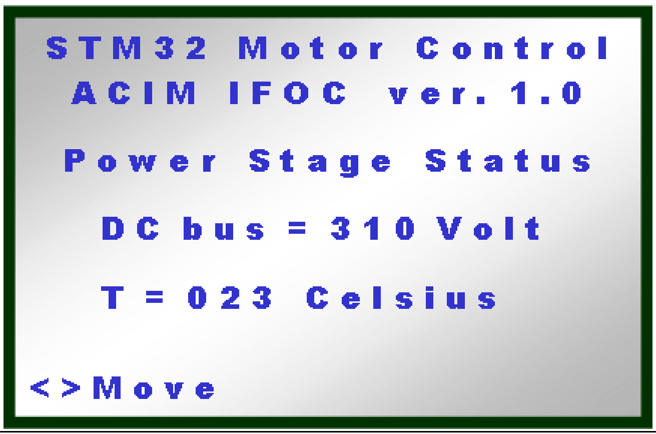
!!! FAULT !!!
Bus Under Voltage

Press Sel to
return to menu

3.7 Actual values of DC bus voltage and IGBT heatsink temperature

In case of overtemperature and bus overvoltage/undervoltage, the system is autoprotected, it warns the user about the fault event while blocking motor operations. The user may however be interested in seeing how these values change during the motor control run phase. For this purpose, an LCD screen was added that continuously shows the DC bus voltage level (in volts) and the power switch heat sink temperature, in Celsius degrees.

Figure 26. LCD screen to monitor power stage status



STM32 Motor Control
ACIM IFOC ver. 1.0

Power Stage Status

DC bus = 310 Volt

T = 023 Celsius

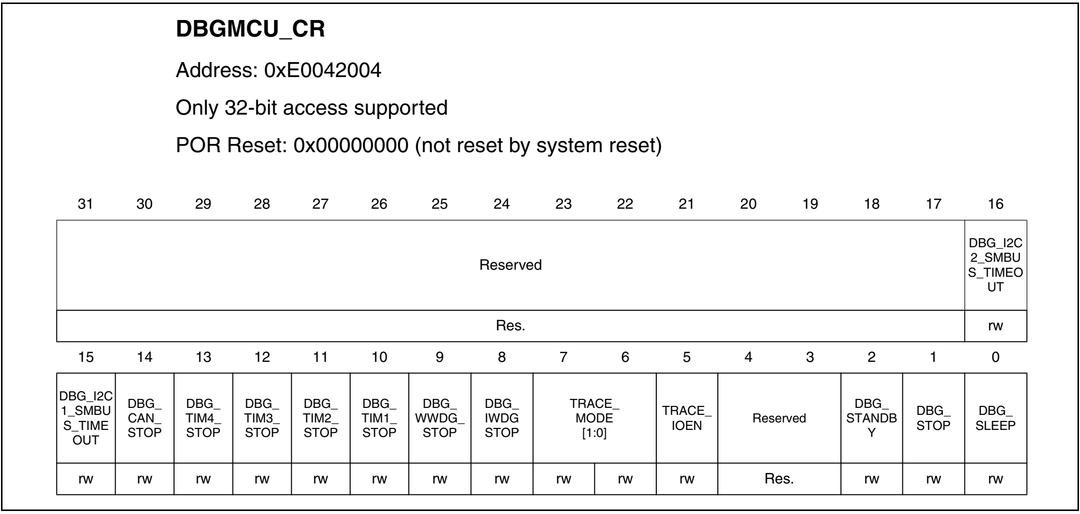
<>Move

3.8 Note on debugging tools

The third party JTAG interface should always be isolated from the application using the MB535 JTAG opto-isolation board; it provides protection for both the JTAG interface and the PC connected to it.

Caution: During a breakpoint, when using the JTAG interface for the firmware development, the motor control cell clock circuitry should always be enabled; if disabled, a permanent DC current may flow in the motor because the PWM outputs are enabled, which could cause permanent damage to the power stage and/or motor. A dedicated bit in the DBGMCU_SR register, the DBG_TIM1_STOP bit, must be set to 1 (see Figure 25). In the `main.c` module the DBG->CR |= DBG_TIM1_STOP; instruction performs the above described task.

Figure 27. DBG_TIM1_STOP bit in TIM1 control register (extract from STM32 reference manual)



DBGMCU_CR

Address: 0xE0042004

Only 32-bit access supported

POR Reset: 0x00000000 (not reset by system reset)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															DBG_I2C2_SMBUS_TIMEOUT
Res.															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DBG_I2C1_SMBUS_TIMEOUT	DBG_CAN_STOP	DBG_TIM4_STOP	DBG_TIM3_STOP	DBG_TIM2_STOP	DBG_TIM1_STOP	DBG_WWDG_STOP	DBG_IWDG STOP	TRACE_MODE [1:0]		TRACE_IOEN	Reserved		DBG_STANDBY	DBG_STOP	DBG_SLEEP
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	Res.		rw	rw	rw

4 Library functions

4.1 Function description conventions

Functions are described in the format given below:

Synopsis	Lists the prototype declarations.
Description	Describes the functions specifically with a brief explanation of how they are executed.
Input	Gives the format and units.
Returns	Gives the value returned by the function, including when an input value is out of range or an error code is returned.
Note	Indicates the limits of the function or specific requirements that must be taken into account before implementation.
Caution	Indicates important points that must be taken into account to prevent hardware failures.
Functions called	Lists called functions. Useful to prevent conflicts due to the simultaneous use of resources.
Code example	Indicates the proper way to use the function, and if there are certain prerequisites (interrupt enabled, etc.).

Some of these sections may not be included if not applicable (for example, no parameters or obvious use).

4.2 Current reading in three shunt resistor topology and space vector PWM generation: `stm32f10x_svpwm_3shunt` module

4.2.1 Overview

Two important tasks are performed in the `stm32f10x_svpwm_3shunt` module:

- Space vector pulse width modulation (SVPWM)
- Current reading in three shunt resistor topology

In order to reconstruct the currents flowing through a three-phase load with the required accuracy using three shunt resistors, it is necessary to properly synchronize A/D conversions with the generated PWM signals. This is why the two tasks are included in a single software module.

4.2.2 List of available functions

The following is a list of available functions as listed in the `stm32f10x_svpwm_3shunt.h` header file:

- *`SVPWM_3ShuntInit` on page 39*
- *`SVPWM_3ShuntCurrentReadingCalibration` on page 40*
- *SVPWM_3ShuntGetPhaseCurrentValues on page 40*
- *SVPWM_3ShuntCalcDutyCycles on page 41*
- *`SVPWM_3ShuntAdvCurrentReading` on page 41*

`SVPWM_3ShuntInit`

Synopsis	void SVPWM_3ShuntInit(void);
Description	The purpose of this function is to set-up microcontroller peripherals for performing 3 shunt resistor topology current reading and center aligned PWM generation. The function initializes NVIC, ADC, GPIO, TIM1 peripherals. In particular, the ADC and TIM1 peripherals are configured to perform two synchronized A/D conversions per PWM switching period. Refer to *Section 4.2.3* for further information.
Input	None.
Returns	None.
Note	It must be called at main level.
Functions called	**Standard library:** RCC_ADCCLKConfig, RCC_AHBPeriphClockCmd, RCC_APB2PeriphClockCmd, GPIO_StructInit, GPIO_Init, GPIO_PinRemapConfig, TIM1_DeInit, TIM1_TimeBaseStructInit, TIM1_TimeBaseInit, TIM1_OCStructInit, TIM1_OC1Init, TIM1_OC2Init, TIM1_OC3Init, TIM1_OC1PreloadConfig, TIM1_OC2PreloadConfig, TIM1_OC3PreloadConfig, TIM1_BDTRConfig, TIM1_SelectOutputTrigger, TIM1_ClearITPendingBit, TIM1_ITConfig, TIM1_Cmd, ADC_DeInit, ADC_Cmd, ADC_StructInit, ADC_Init, ADC_StartCalibration, ADC_GetCalibrationStatus, ADC_RegularChannelConfig, ADC_InjectedSequencerLengthConfig, ADC_InjectedChannelConfig, ADC_ExternalTrigInjectedConvCmd, NVIC_PriorityGroupConfig, NVIC_StructInit, NVIC_Init. **Motor control library:** SVPWM_3ShuntCurrentReadingCalibration

`SVPWM_3ShuntCurrentReadingCalibration`

Synopsis	void SVPWM_3ShuntCurrentReadingCalibration(void);
Description	The purpose of this function is to store the three analog voltages corresponding to zero current values for compensating the offset introduced by the amplification network.
Input	None.
Returns	None.
Note	This function is called by MCL_Init which is executed at every motor startup. The function reads the analog voltage on A/D channels utilized for current reading before TIM1 outputs are enabled so that the current flowing through the inverter is zero. Those values are then stored into Phase_x_Offset variables.
Functions called	**Standard library:** ADC_ITConfig, ADC_ExternalTrigInjectedConvConfig, ADC_ExternalTrigInjectedConvCmd, ADC_InjectedSequencerLengthConfig, ADC_InjectedChannelConfig, ADC_SoftwareStartInjectedConvCmd, ADC_GetFlagStatus, ADC_GetInjectedConversionValue, ADC_SoftwareStartInjectedConvCmd **Motor control library:** SVPWM_InjectedConvConfig

`SVPWM_3ShuntGetPhaseCurrentValues`

Synopsis	Curr_Components SVPWM_3ShuntGetPhaseCurrentValues(void);
Description	This function computes current values of Phase A and Phase B in q1.15 format starting from values acquired from the A/D Converter peripheral.
Input	None.
Returns	Curr_Components type variable.
Note	In order to have a q1.15 format for the current values, the digital value corresponding to the offset must be subtracted when reading phase current A/D converted values. Therefore, the function must be called after SVPWM_3ShuntCurrentReadingCalibration.
Functions called	None.

`SVPWM_3ShuntCalcDutyCycles`

Synopsis	void SVPWM_3ShuntCalcDutyCycles (Volt_Components Stat_Volt_Input);
Description	After execution of the IFOC algorithm, new stator voltage components V_α and V_β are computed. The purpose of this function is to calculate exactly the three duty cycles to be applied to motor phases from the values of these voltage components. Moreover, once the three duty cycles to be applied in the next PWM period are known, this function sets the TIM1 Channel 4 capture/compare register (TIM1_CCR4) to set the sampling point for the next current reading. In particular, depending on the duty cycle values, the sampling point is computed (see *Section 4.2.5 on page 45*). Refer to *Section 4.2.3* for information on the theoretical approach of SVPWM.
Input	V_α and V_β
Returns	None.
Note	None.
Functions called	None.

`SVPWM_3ShuntAdvCurrentReading`

Synopsis	void SVPWM_3ShuntAdvCurrentReading(FunctionalState cmd);
Description	It is used to enable or disable advanced current reading. If advanced current reading is disabled, current reading is performed in conjunction with the update event.
Input	cmd (ENABLE or DISABLE)
Returns	None.
Note	None.
Functions called	TIM1_ClearFlag, TIM1_ITConfig

4.2.3 Space vector PWM implementation

Figure 28 shows the Stator Voltage components V_α and V_β while *Figure 29* illustrates the corresponding PWM for each of the six space vector sectors:

Figure 28. V_α and V_β stator voltage components

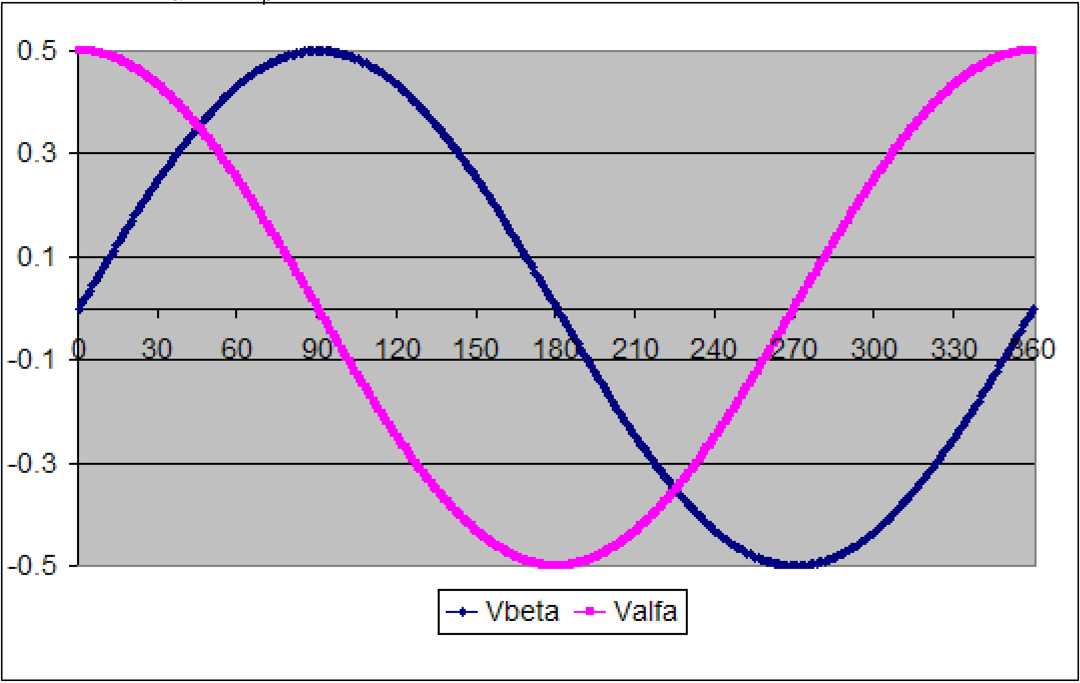
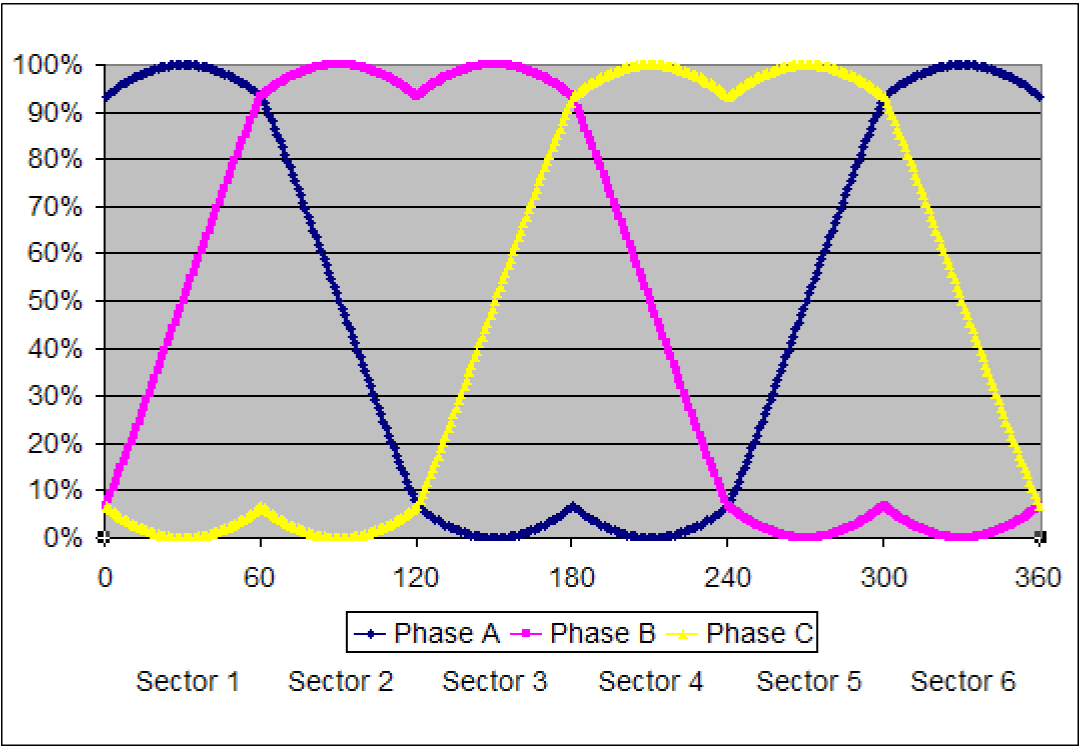


Figure 29. SVPWM phase voltages waveforms



With the following definitions for: $U_\alpha = \sqrt{3} \times T \times V_{alfa}$, $U_\theta = -T \times V_{beta}$ and $X = U_\beta$,

$Y = \frac{U_\alpha + U_\theta}{2}$ and $Z = \frac{U_\theta - U_\alpha}{2}$.

literature demonstrates that the space vector sector is identified by the conditions shown in *Table 1*.

Table 1. Sector identification

	Y < 0			**Y ≥ 0**		
	Z < 0	**Z ≥ 0**		**Z < 0**		**Z ≥ 0**
		X ≤ 0	**X > 0**	**X ≤ 0**	**X > 0**	
Sector	V	IV	III	VI	I	II

The duration of the positive pulse widths for the PWM applied on Phase A, B and C are respectively computed by the following relationships:

Sector I, IV: $t_A = \frac{T + X - Z}{2}$, $t_B = t_A + Z$, $t_C = t_B - X$

Sector II, V: $t_A = \frac{T + Y - Z}{2}$, $t_B = t_A + Z$, $t_C = t_A - Y$

Sector III, VI: $t_A = \frac{T - X + Y}{2}$, $t_B = t_C + X$, $t_C = t_A - Y$, where T is the PWM period.

Now, considering that the PWM pattern is center aligned and that the phase voltages must be centered at 50% of duty cycle, it follows that the values to be loaded into the PWM output compare registers are given respectively by:

Sector I, IV: $TimePhA = \frac{T}{4} + \frac{T/2 + X - Z}{2}$, $TimePhB = TimePhA + Z$, $TimePhC = TimePhB - X$

Sector II, V: $TimePhA = \frac{T}{4} + \frac{T/2 + Y - Z}{2}$, $TimePhB = TimePhA + Z$, $TimePhC = TimePhA - Y$

Sector III,VI: $TimePhA = \frac{T}{4} + \frac{T/2 + Y - X}{2}$, $TimePhB = TimePhC + X$, $TimePhC = TimePhA - Y$

4.2.4 Current sampling in three shunt topology and general purpose A/D conversions

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

$$I_1 + I_2 + I_3 = 0$$

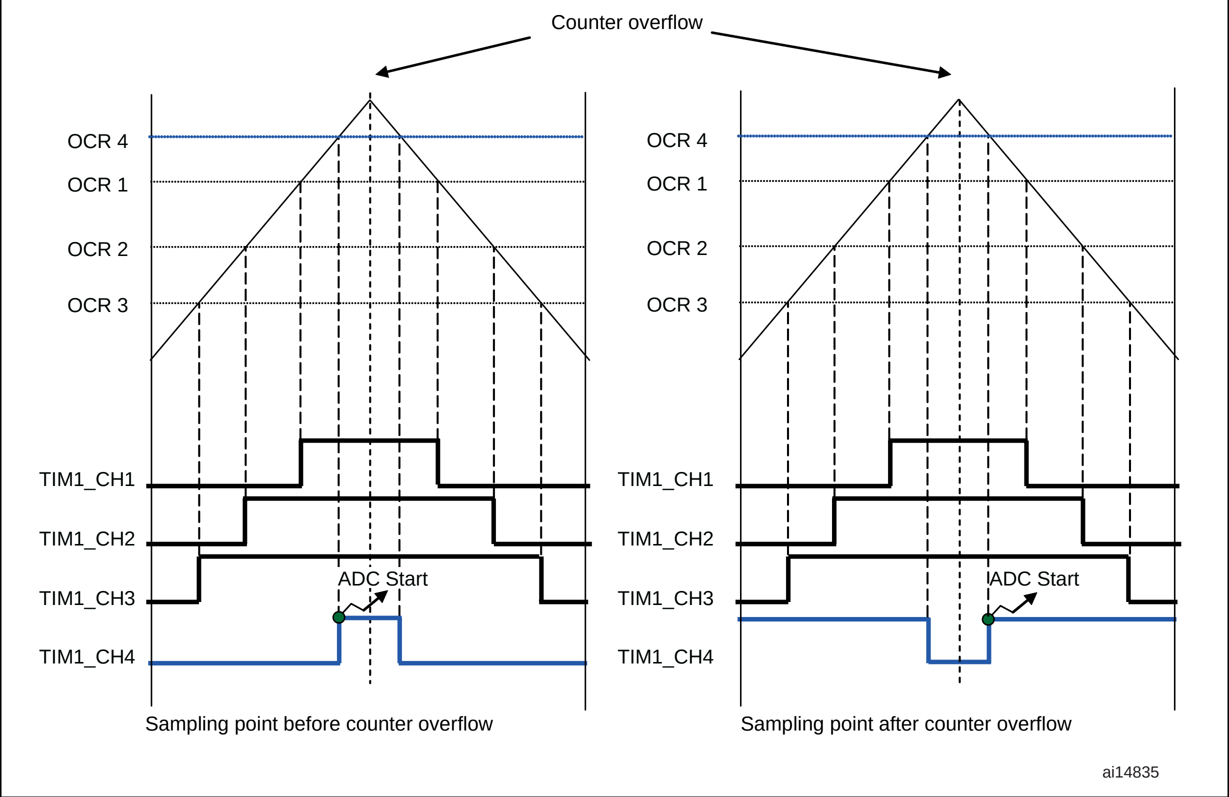
For this reason, to reconstruct the currents flowing through a generic three-phase load, it is sufficient to sample only two out of the three currents while the third one can be computed by using the above relation.

The flexibility of the STM32F10xxx 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.

Injected conversions, as described above, are used for current-reading purposes whereas regular conversions are reserved for the user. As soon as A/D conversions for current-reading purposes have completed, bus voltage and temperature sensing conversions are simultaneously performed by the dual A/D.

Figure 30 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 30. PWM and ADC synchronization



In this way, supposing that the sampling point must be set before the counter overflow, that is, before the TIM1 counter value matches the OCR4 register value during the upcounting, 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. In so doing, when the TIM1 counter matches the OCR4 register value during the downcounting, the A/D samplings are started.

After the first two simultaneous conversions other two simultaneous conversions are started, one for the bus voltage and the other for the temperature sensor. At the end of the second conversion, the three-phase load current has been updated and the FOC algorithm can then be executed in the A/D end of injected conversion interrupt service routine (JEOC ISR).

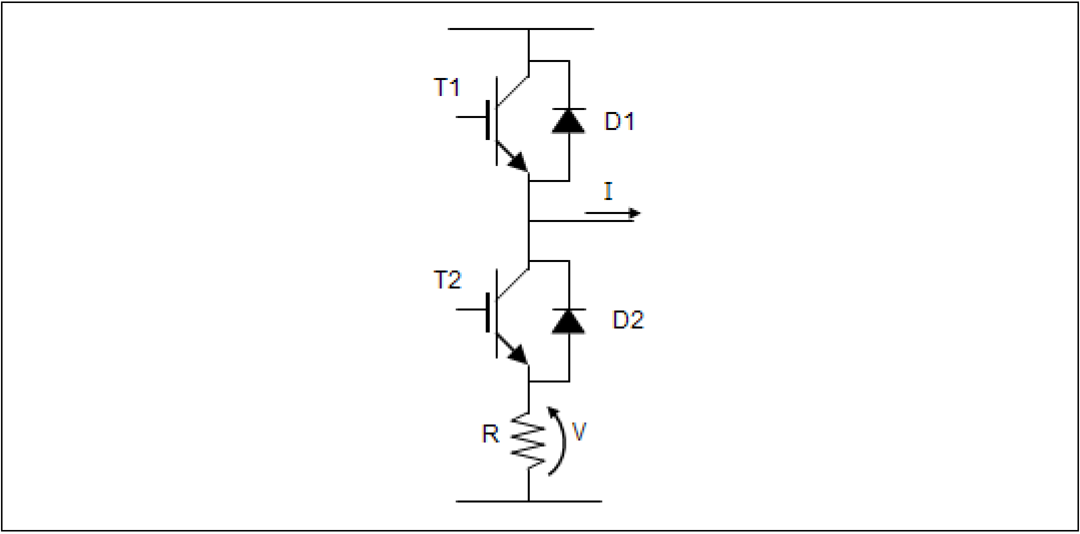
After execution of the FOC algorithm, the value to be loaded into the OCR4 register is calculated to set the sampling point for the next PWM period, and the A/D converter is configured to sample the correct channels.

Regular conversions are reserved for user purposes and must be configured manually (See also firmware standard library user manual UM0427).

4.2.5 Tuning delay parameters and sampling stator currents in three shunt resistor topology

Figure 31 shows one of the three inverter legs with the related shunt resistor:

Figure 31. Inverter leg and shunt resistor position



To indirectly measure the phase current I, it is possible to read the voltage V providing that the current flows through the shunt resistor R.

It is possible to demonstrate that, whatever the direction of current I, it always flows through the resistor R if transistor T2 is switched on and T1 is switched off. This implies that in order to properly reconstruct the current flowing through one of the inverter legs, it is necessary to properly synchronize the conversion start with the generated PWM signals. This also means that current reading cannot be performed on a phase where the duty cycle applied to the low side transistor is either null or very short.

Fortunately, as discussed in *Section 4.2.4*,to reconstruct the currents flowing through a generic three-phase load, it is sufficient to simultaneously sample only two out of three currents, the third one being computed from the relation given in *Section 4.2.4*. Thus, depending on the space vector sector, the A/D conversion of voltage V will be performed only on the two phases where the duty cycles applied to the low side switches are the highest. In particular, by looking at *Figure 29*, you can deduct that in sectors 1 and 6, the voltage on the Phase A shunt resistor can be discarded; likewise, in sectors 2 and 3 for Phase B, and finally in sectors 4 and 5 for Phase C.

Moreover, in order to properly synchronize the two stator current reading A/D conversions, it is necessary to distinguish between the different situations that can occur depending on PWM frequency and applied duty cycles.

Note: *The explanations below refer to space vector sector 4. They can be applied in the same manner to the other sectors.*

Case 1: Duty cycle applied to Phase A low side switch is larger than DT+T_N

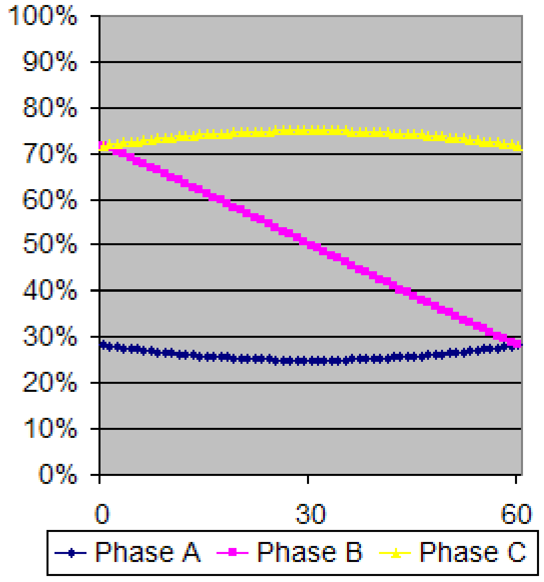
Where:

- DT is dead time.
- T_N is the duration of the noise induced on the shunt resistor voltage of a phase by the commutation of a switch belonging to another phase.
- T_S is the sampling time of the STM32F10xxx A/D converter (the following consideration is made under the hypothesis that $T_S < DT + T_N$). Refer to the STM32F10xxx reference manual for more detailed information.

This case typically occurs when SVPWM with low (<60%) modulation index is generated (see *Figure 32*). The modulation index is the applied phase voltage magnitude expressed as a percentage of the maximum applicable phase voltage (the duty cycle ranges from 0% to 100%).

Figure 33 offers a reconstruction of the PWM signals applied to low side switches of Phase A and B in these conditions plus a view of the analog voltages measured on the STM32F10xxx A/D converter pins for both Phase B and C (the time base is lower than the PWM period).

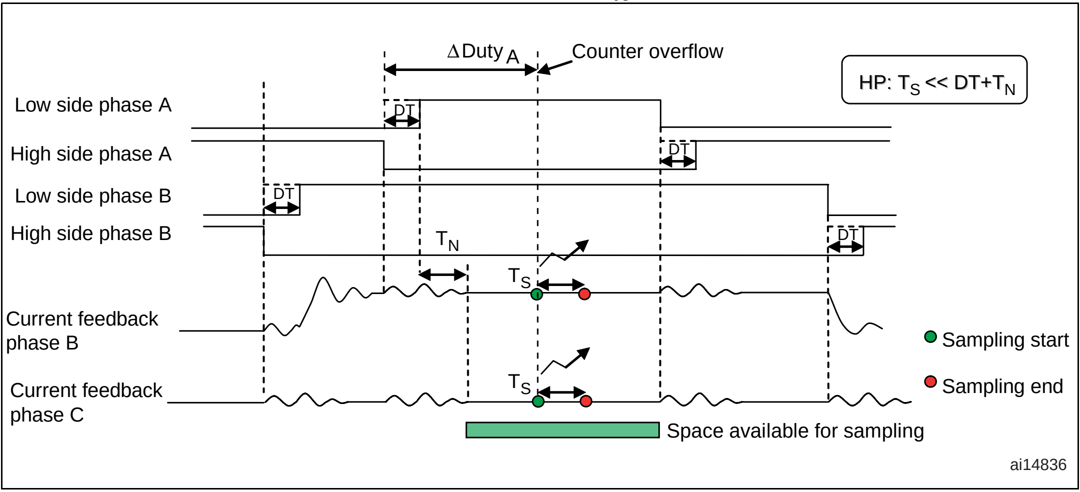
Figure 32. Low side switches gate signals (low modulation indexes)



Note that these current feedbacks are constant in the view in *Figure 33* because it is assumed that commutations on Phase B and C have occurred out of the visualized time window.

Moreover, it can be observed that in this case the two stator current sampling conversions can be performed synchronized with the counter overflow, as shown in *Figure 33*.

Figure 33. Low side Phase A duty cycle > $DT+T_N$



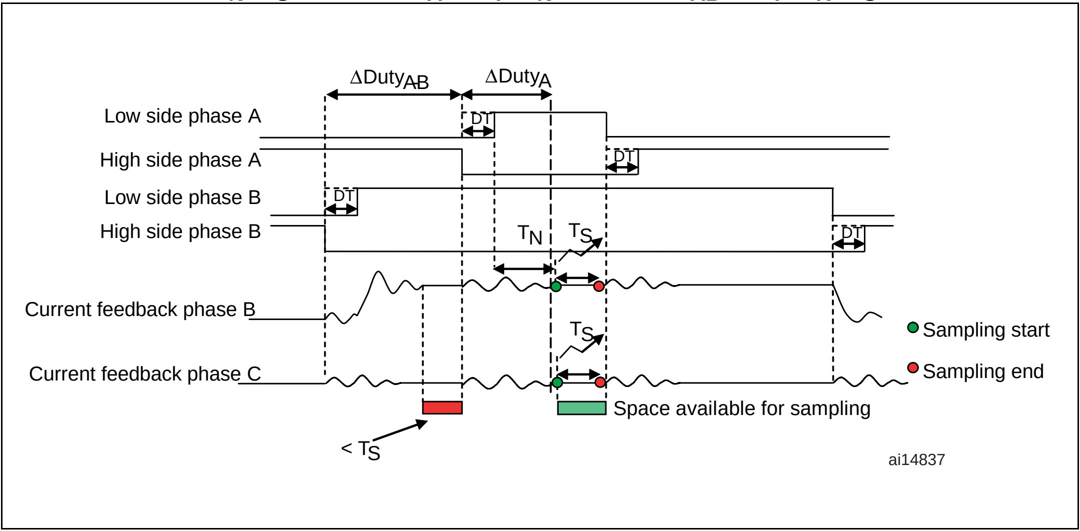
Case 2: $DT+(T_N+T_S)/2 < \Delta Duty_A < D_T+T_N$ and $\Delta Duty_{AB} < D_T+T_R+T_S$

With the increase in modulation index, $\Delta Duty_A$ can assume values smaller than D_T+T_N. Sampling synchronized with the counter overflow could be impossible.

In this case, the two currents can be sampled between the two Phase A low side commutations, after the counter overflow.

Consider that in order to avoid the acquisition of the noise induced on the phase B current feedback by phase A switch commutations, it is required to wait for the noise to be over (T_N). See *Figure 34*.

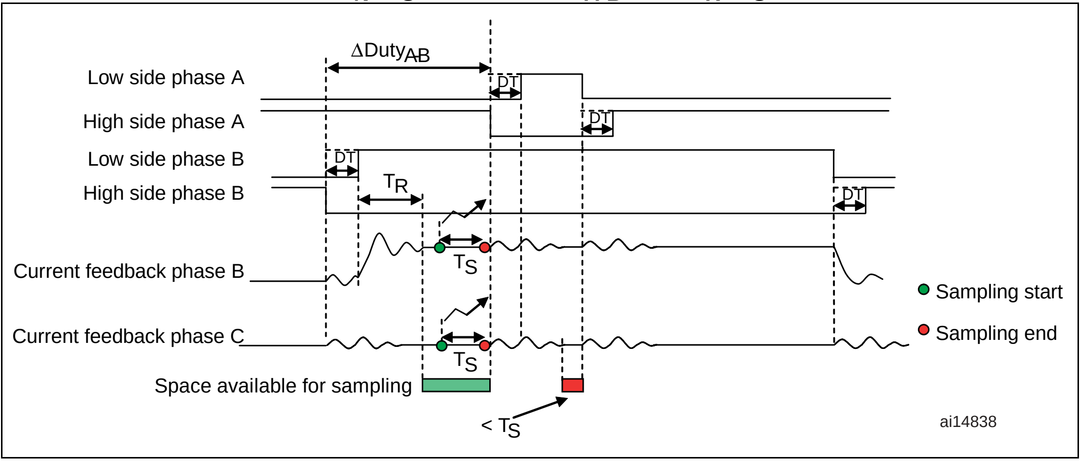
Figure 34. $DT+(T_N+T_S)/2 < \Delta Duty_A < D_T+T_N$ and $\Delta Duty_{AB} < D_T+T_R+T_S$



Case 3: $\Delta DutyA < DT+(T_N+T_S)/2$ and $\Delta Duty_{A-B} > DT+T_R+T_S$

In this case, the two currents can be sampled between Phase B low-side switch-on and Phase A high-side switch-off. The choice was therefore made to sample the currents T_S µs before of phase A high-side switch-off (see *Figure 35*).

Figure 35. ΔDutyA < DT+(T_N+T_S)/2 and $\Delta Duty_{A\text{-}B}$>DT+T_R+T_S



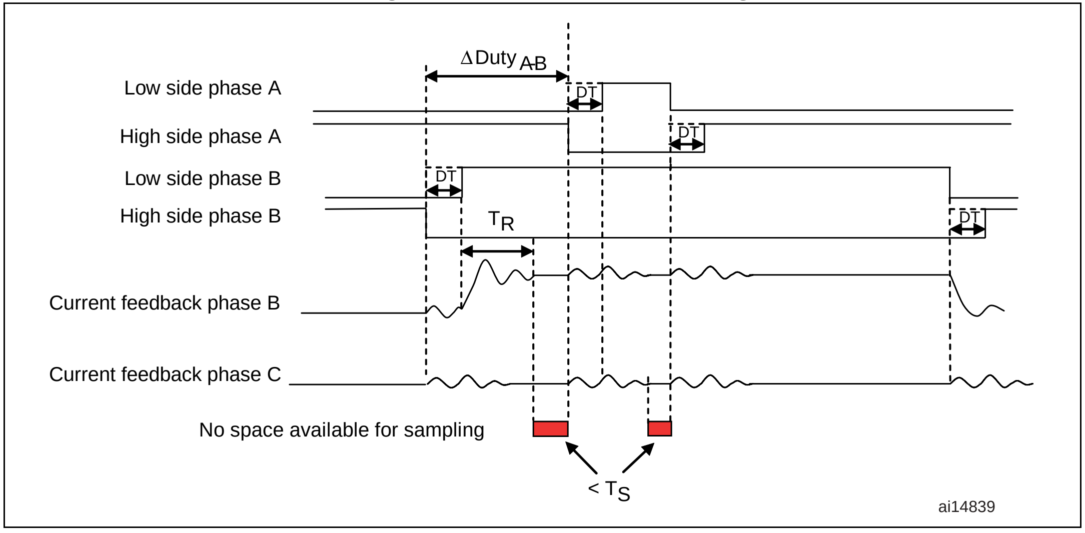
Case 4: $\Delta Duty_A$<DT+(T_N+T_S)/2 and $\Delta Duty_{A\text{-}B}$<DT+T_R+T_S

In this case, the duty cycle applied to Phase A is so short that no current sampling can be performed between the two low-side commutations.

Furthermore if the difference in duty cycles between Phases B and A is not long enough to allow the A/D conversions to be performed between Phase B low-side switch-on and Phase A high-side switch-off, it is impossible to sample the currents (See *Figure 36*).

To avoid this condition, it is necessary to reduce the maximum modulation index or decrease the PWM frequency.

Figure 36. $\Delta Duty_A$<DT+(T_N+T_S)/2 and $\Delta Duty_{A\text{-}B}$<DT+T_R+T_S



The following parameters have been set as default in the firmware. They are related to the MB459 board:

- DT = 0.8 µs
- T_N = 2.55 µs
- T_S = 0.7 µs
- T_R = 2.55 µs

The maximum applicable duty cycles are listed in *Table 2* as a function of the PWM frequency.

Table 2. PWM frequency vs. maximum duty cycle relationship

PWM frequency	Max duty cycle	Max modulation Index
Up to 11.4 kHz	100%	100%
12.2 kHz	99%	98%
12.9 kHz	98.5%	97%
13.7 kHz	98%	96%
14.4 kHz	98%	96%
15.2 kHz	97%	94%
16 kHz	96.5%	93%
16.7 kHz	96.5%	93%
17.5 kHz	95.5%	91%

Note: The values above were measured using the MB459 board. This evaluation platform is designed to support several motor driving topologies (PMSM and AC induction) and current reading strategies (single- and three-shunt resistor). Therefore, the figures provided in Table 2 should be understood as a starting point and not as a best case.

It is possible to adjust the noise parameters based on customized hardware by editing the following definitions in the `MC_pwm_3shunt_prm.h` header file:

```
#define SAMPLING_TIME_NS 700 //0.7usec
#define TNOISE_NS 2550 //2.55usec
#define TRISE_NS 2550 //2.55usec
```

Changing the noise parameters, sampling time and dead time affects the values provided in *Table 2*.

4.3 Isolated current sensor reading and space vector PWM generation: `stm32f10x_svpwm_ics module`

4.3.1 Overview

Two important tasks are performed in the `stm32f10x_svpwm_ics` module.

- Space vector pulse width modulation (SVPWM),
- Three-phase current reading when two isolated current sensors (ICS) are used.

In order to reconstruct the currents flowing through a three phase load with the required accuracy using two ICS', it is necessary to properly synchronize A/D conversions with the generated PWM signals. This is why the two tasks are included in a single software module.

4.3.2 List of available functions and interrupt service routines

The following is a list of available functions as listed in the `stm32f10x_svpwm_ics.h` header file:

- *SVPWM_IcsInit on page 50*
- *SVPWM_IcsCurrentReadingCalibration on page 51*
- *SVPWM_IcsGetPhaseCurrentValues on page 52*
- *SVPWM_IcsCalcDutyCycles on page 52*

`SVPWM_IcsInit`

Synopsis	void SVPWM_IcsInit(void);
Description	The purpose of this function is to set-up microcontroller peripherals for performing ICS reading and center aligned PWM generation. The function initializes NVIC, ADC, GPIO, and TIM1 peripherals. In particular ADC and TIM1 peripherals are configured to perform two pairs of simultaneous injected A/D conversions every time PWM registers are updated (event called U event). The first pair of conversions reads the current values while the second one acquires the bus voltage and the voltage on the temperature sensor. Refer to *Section 4.3.3* for further information on A/D conversion triggering in ICS configuration.
Input	None.
Returns	None.
Note	It must be called at main level.

Functions called	**Standard library:** RCC_ADCCLKConfig, RCC_AHBPeriphClockCmd, RCC_APB2PeriphClockCmd, GPIO_StructInit, GPIO_Init, GPIO_PinRemapConfig, TIM1_DeInit, TIM1_TimeBaseStructInit, TIM1_TimeBaseInit, TIM1_OCStructInit, TIM1_OC1Init, TIM1_OC2Init, TIM1_OC3Init, TIM1_BDTRConfig, TIM1_SelectOutputTrigger, TIM1_ClearITPendingBit, TIM1_ITConfig, TIM1_Cmd, ADC_DeInit, ADC_Cmd, ADC_StructInit, ADC_Init, ADC_StartCalibration, ADC_GetCalibrationStatus, ADC_InjectedSequencerLengthConfig, ADC_InjectedChannelConfig, ADC_ExternalTrigInjectedConvCmd, NVIC_PriorityGroupConfig, NVIC_StructInit, NVIC_Init. **Motor control library:** SVPWM_IcsCurrentReadingCalibration

`SVPWM_IcsCurrentReadingCalibration`

Synopsis	void SVPWM_IcsCurrentReadingCalibration(void);
Description	The purpose of this function is to store the two analog voltages corresponding to zero current values for compensating the offset introduced by both ICS and amplification network.
Input	None.
Returns	None.
Note	This function is called by MCL_Init which is executed at every motor startup. It reads the analog voltage on the A/D channels used for current reading before the PWM outputs are enabled so that the current flowing through the inverter is zero.
Functions called	Standard Library: ADC_ITConfig, ADC_ExternalTrigInjectedConvConfig, ADC_ExternalTrigInjectedConvCmd, ADC_InjectedSequencerLengthConfig, ADC_InjectedChannelConfig, ADC_SoftwareStartInjectedConvCmd, ADC_GetFlagStatus, ADC_GetInjectedConversionValue **Motor Control library:** SVPWM_IcsInjectedConvConfig

`SVPWM_IcsGetPhaseCurrentValues`

Synopsis	Curr_Components SVPWM_IcsGetPhaseCurrentValues(void);
Description	This function computes current values of Phase A and Phase B in q1.15 format from the values acquired from the A/D converter.
Input	None.
Returns	Curr_Components type variable
Note	In order to have a q1.15 format for the current values, the digital value corresponding to the offset must be subtracted when reading phase current A/D converted values. Thus, the function must be called after SVPWM_IcsCurrentReadingCalibration.
Functions called	None.

`SVPWM_IcsCalcDutyCycles`

Synopsis	void SVPWM_IcsCalcDutyCycles (Volt_Components Stat_Volt_Input);
Description	After execution of the IFOC algorithm, new stator voltages component V_α and V_β are computed. The purpose of this function is to calculate exactly the three duty cycles to be applied to motor phases from the values of these voltage components. Refer to *Section 4.2.3* for details about the theoretical approach of SVPWM and its implementation.
Input	V_α and V_β
Returns	None.
Note	None.
Functions called	None.

4.3.3 Current sampling in isolated current sensor topology and integrating general-purpose A/D conversions

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

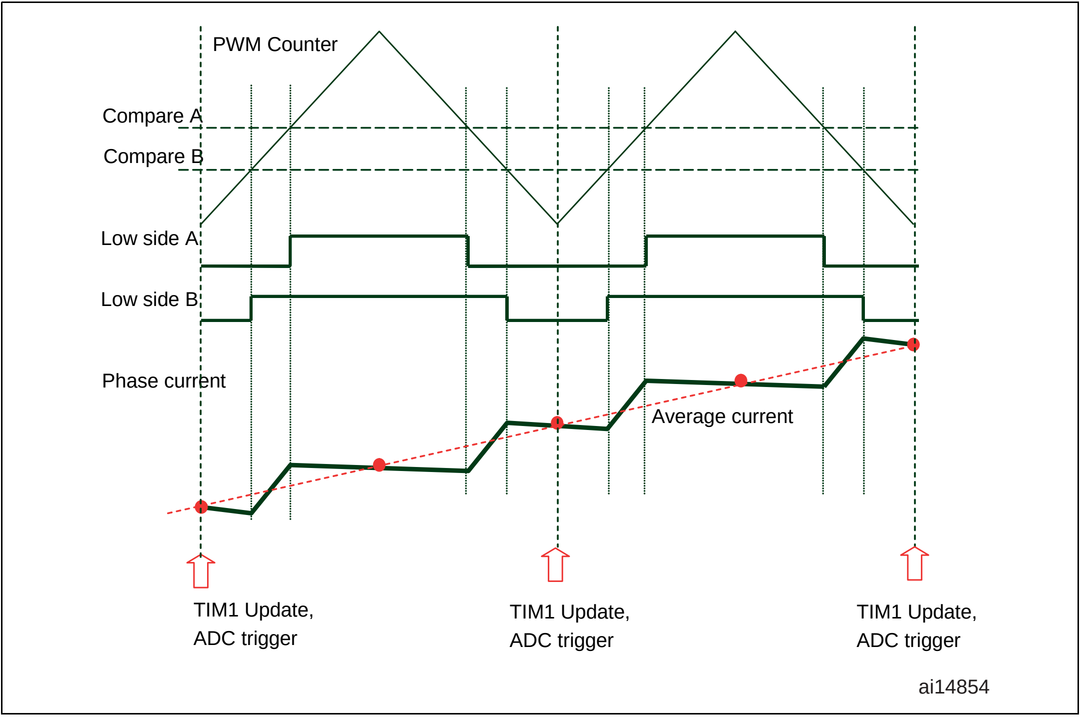
$$I_1+I_2+I_3=0$$

Therefore, 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 relationship.

The flexibility of the STM32F103xx A/D converter trigger makes it possible to synchronize the two A/D conversions necessary for reconstructing the stator currents flowing through the three-phase AC induction motor with the PWM reload register updates. The update rate can be adjusted using the repetition counter. This is important because, as shown in *Figure 37*, it is precisely during counter overflow and underflow that the average level of current is equal to the sampled current. Refer to the STM32F103xx Reference Manual to learn more about A/D conversion triggering and the repetition counter.

Regular conversions are reserved for the user and must be configured manually (See also firmware standard library user manual UM0427).

Figure 37. Stator currents sampling in ICS configuration (`REP_RATE=1`)



4.4 Induction motor IFOC vector control: `MC_IFOC_Drive` module

4.4.1 Overview

The `MC_IFOC_Drive` module, designed for AC induction machines, provides, at the core, decoupled torque and flux regulation, relying on indirect field oriented control algorithm.

In addition, it makes available other important features:

- speed regulation by PID feedback control,
- flux weakening for extended speed range.

It works, requiring no adjustment, with all of the selectable current or speed sensing configurations (in accordance with the settings in the `stm32f10x_MCconf.h` file):

- isolated current sensing (ICS),
- three shunt resistors current sensing,
- encoder position and speed sensing,
- tachometer speed sensing.

It handles several functions of other modules, and has no direct access on the microcontroller peripheral registers.

4.4.2 List of available C functions

- *IFOC_Init on page 54*
- *IFOC_Model on page 55*
- *IFOC_CalcFluxTorqueRef on page 57*
- *CalcIm on page 58*
- *CalcRotFlxSlipFreq on page 58*

IFOC_Init

Synopsis	void IFOC_Init(void)
Description	This function is normally called at every motor startup. It performs the initialization of some of the variables used for IFOC implementation by the MC_IFOC_Drive.c module.
Input	None.
Returns	None.
Note	None.
Functions called	if working with ICs: SVPWM_IcsCalcDutyCycles; if working with three shunts: SVPWM_3ShuntCalcDutyCycles.

IFOC_Model

Synopsis void IFOC_Model (void)

Description The purpose of this function is to perform AC-IM torque and flux regulation, implementing the IFOC vector algorithm.

Current commands $i_{qs}^{\lambda r}$ * and $i_{ds}^{\lambda r}$ * (which, under field oriented conditions, can control machine torque and flux respectively) are defined outside this function (in Speed control mode they are provided, by means of speed and flux regulators, by the IFOC_CalcFluxTorqueRef function, while in Torque control mode they are settled by the user).

Therefore, as a current source is required, the function has to run the power converter as a CR-PWM. For this purpose, it implements a high performance synchronous (d, q) frame current regulator, whose operating frequency is defined, as explained in *Section 2.2.2*, by the parameter REP_RATE (in conjunction with PWM_FREQ).

Triggered by ADC JEOC ISR, the function loads stator currents (read by ICS or shunt resistors) and carries out Clark and Park transformations, converting them to $i_{qs}^{\lambda r}$ and $i_{ds}^{\lambda r}$ (see *Figure 6*).

Then, these currents are fed to PID regulators together with reference values $i_{qs}^{\lambda r}$ * and $i_{ds}^{\lambda r}$ *. The regulator output voltages $v_{qs}^{\lambda r}$ * and $v_{ds}^{\lambda r}$ * then must be transformed back to a stator frame (through Reverse Park conversion), and finally drive the power stage.

In order to correctly perform Park and Reverse Park transformation, it is essential to accurately estimate the rotor flux position ($\theta^{\lambda r}$) (because currents have to be oriented in phase and in quadrature with rotor flux). To manage this task:

– function CalcIm is called to provide Im, that is the estimated value of the rotor flux as a response to the variation of input current $i_{ds}^{\lambda r}$ (see CalcIm function description);

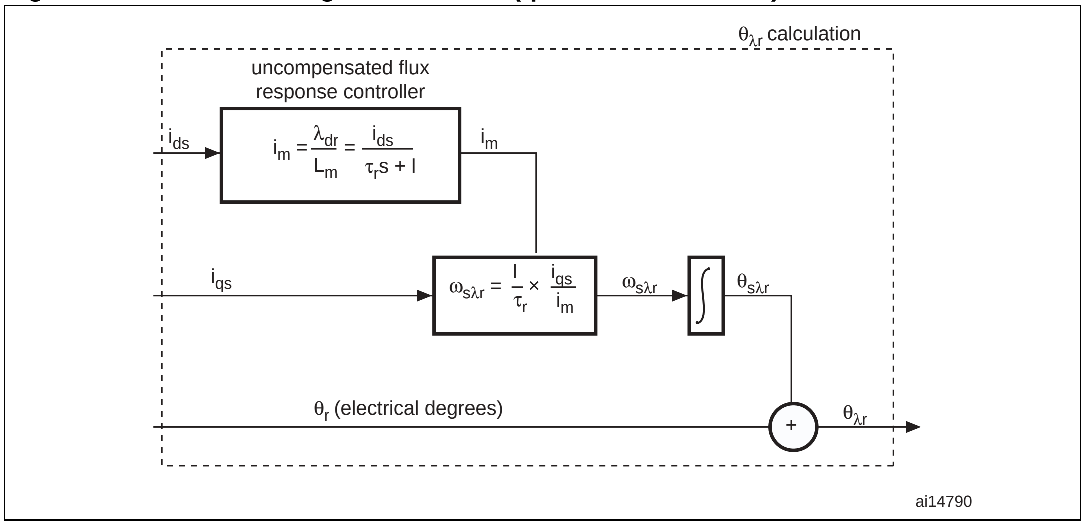
– function CalcRotFlxSlipFreq (see CalcRotFlxSlipFreq function description) evaluates rotor flux slip frequency $\omega_{s\lambda r}$ (relying on known rotor time constant); if using a tachogenerator, the rotor flux position $\theta^{\lambda r}$ is calculated by integrating the sum of $\omega_{s\lambda r}$ and rotor electrical speed ω_r (*Figure 39*) while, with an incremental encoder, $\theta^{\lambda r}$ is determined by summing the rotor electrical angle and the integral of $\omega_{s\lambda r}$ (*Figure 38*).

Input None.

Returns None.

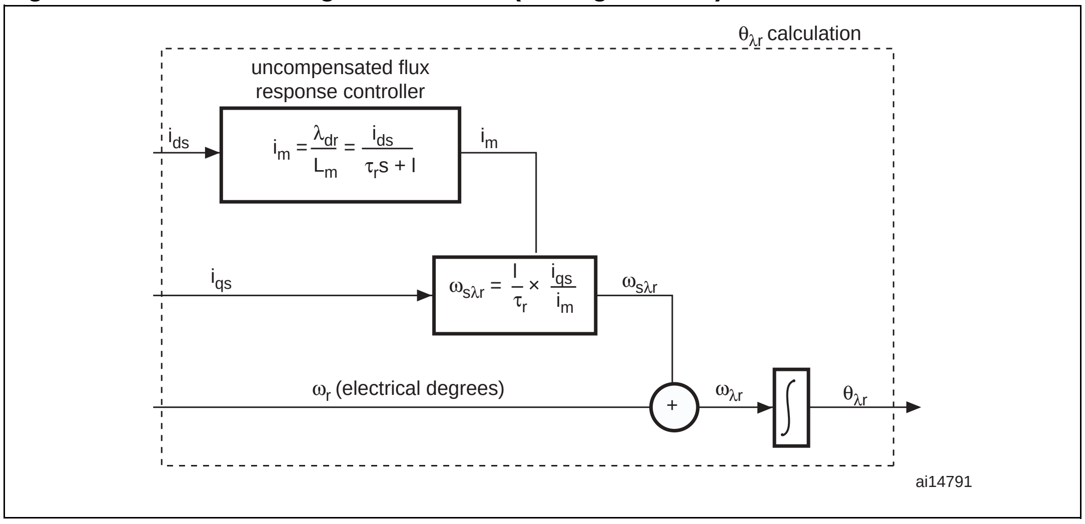
Functions called	CalcIm, CalcRotFlxSlipFreq;
	Clarke, Park, RevPark_Circle_Limitation;
	PID_Torque_Regulator, PID_Flux_Regulator, Rev_Park;
	If working with encoder:
	ENC_Get_Electrical_Angle;
	if Working with tachogenerator:
	TAC_GetRotorFreq;
	if working with 'ICS':
	SVPWM_IcsGetPhaseCurrentValues, SVPWM_IcsCalcDutyCycles;
	if working with 'three shunt':
	SVPWM_3ShuntGetPhaseCurrentValues,
	SVPWM_3ShuntCalcDutyCycles.

Figure 38. Rotor flux angle calculation (quadrature encoder)



$\theta_{\lambda r}$ calculation
uncompensated flux response controller
i_{ds}
$i_m = \frac{\lambda_{dr}}{L_m} = \frac{i_{ds}}{\tau_r s + 1}$
i_m
i_{qs}
$\omega_{s\lambda r} = \frac{1}{\tau_r} \times \frac{i_{qs}}{i_m}$
$\omega_{s\lambda r}$
$\theta_{s\lambda r}$
θ_r (electrical degrees)
+
$\theta_{\lambda r}$
ai14790

Figure 39. Rotor flux angle calculation (tachogenerator)

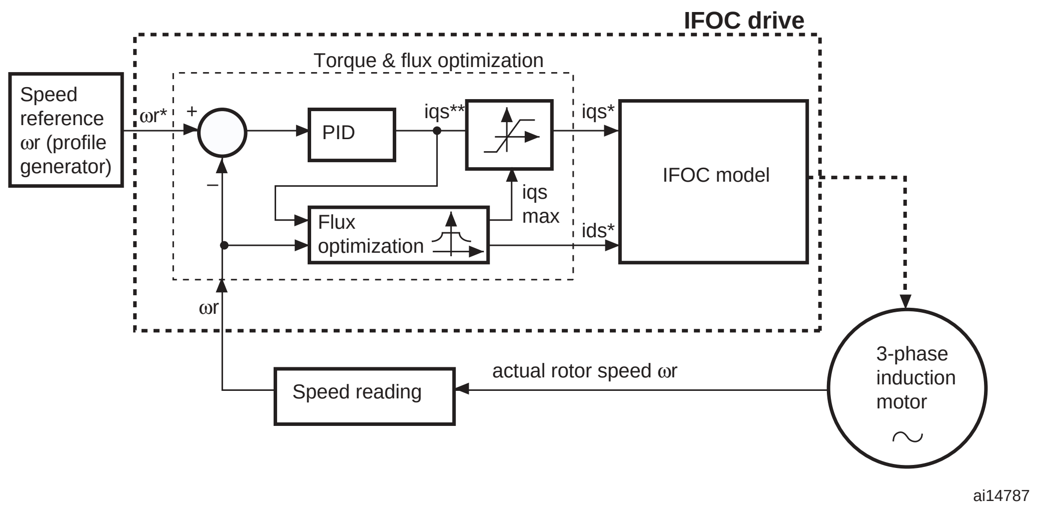


$\theta_{\lambda r}$ calculation
uncompensated flux response controller
i_{ds}
$i_m = \frac{\lambda_{dr}}{L_m} = \frac{i_{ds}}{\tau_r s + 1}$
i_m
i_{qs}
$\omega_{s\lambda r} = \frac{1}{\tau_r} \times \frac{i_{qs}}{i_m}$
$\omega_{s\lambda r}$
ω_r (electrical degrees)
+
$\omega_{\lambda r}$
$\theta_{\lambda r}$
ai14791

`IFOC_CalcFluxTorqueRef`

Synopsis	void IFOC_CalcFluxTorqueRef (void)
Description	This function provides current components iqs* and ids* to be used as reference values (by the IFOC_Model function) in Speed control mode (see “Torque & Flux optimization” block in *Figure 40*). Speed setpoint and actual rotor speed ω_r are compared in a PID control loop, whose output is iqs**. This component, together with the previous flux reference and the rotor speed ω_r, is used to work out the stator frequency that has to be generated. With this information, two lookup-tables (described in MC_ACmotor_prm, *Section 2.2.5*, defined by taking into account the field weakening strategy explained in *Section 4.4.4*) are run through, in order to get the optimal flux reference (ids*) and the saturation value of the torque current component (iqs max) that allow to reach the desired speed (under the obvious limitations of rated torque and rated power).
Input	None.
Returns	None.
Functions called	PID_Speed_Regulator; mul_q15_q15_q31, div_q31_q15_q15.

Figure 40. Torque and flux optimization block



`CalcIm`

Synopsis	s16 CalcIm (s16 hId_input);
Description	The purpose of this routine is to supply (to the calling function) the estimated value of the rotor flux, as a response to variations of the input current value $i_{ds}^{\lambda r}$ (see “uncompensated flux response controller” block in *Figure 38* and *Figure 39*). See *Section 4.4.3* for in-depth information about the computations implemented.
Input	Stator current $i_{ds}^{\lambda r}$ in q1.15 format.
Returns	Magnetizing current i_m (defined as rotor flux λ_r divided by magnetizing inductance L_m) in q1.15 format.
Functions called	mul_q15_q15_q31 (MC_qmath.h)

`CalcRotFlxSlipFreq`

Synopsis	s32 CalcRotFlxSlipFreq (s16 hIq_input, s16 hIm_input)
Description	This function estimates the rotor flux slip frequency $\omega_{s\lambda r}$ (central block in *Figure 38* and *Figure 39*), as result of currents $i_{qs}^{\lambda r}$ and i_m ($\lambda_{dr}^{\lambda r}/L_m$). See *Section 4.4.3* for an in-depth comprehension of the implemented computations.
Input	Stator current $i_{qs}^{\lambda r}$ and magnetizing current i_m, both in q1.15 format.
Returns	Rotor flux slip frequency, expressed in pulses per PWM period * 65536 (65536 pulses = 2π radiants).
Functions called	mul_q15_q15_q31 div_q31_q15_q15 (MC_qmath.h)

4.4.3 Detailed explanation about indirect field oriented control (IFOC)

Consider the voltage equations of an induction machine, being transformed on a reference frame (q, d) that is synchronous with the rotor flux λ_r (about reference frame theory see [1] in Appendix *A.7: References on page 102*):

$$v_{qs}^{\lambda_r} = r_s i_{qs}^{\lambda_r} + \frac{d\lambda_{qs}^{\lambda_r}}{dt} + \omega_{\lambda_r}\lambda_{ds}^{\lambda_r}$$

$$v_{ds}^{\lambda_r} = r_s i_{ds}^{\lambda_r} + \frac{d\lambda_{ds}^{\lambda_r}}{dt} - \omega_{\lambda_r}\lambda_{qs}^{\lambda_r}$$

$$0 = r_r i_{qr}^{\lambda_r} + \frac{d\lambda_{qr}^{\lambda_r}}{dt} + (\omega_{\lambda_r} - \omega_r)\lambda_{dr}^{\lambda_r}$$

$$0 = r_r i_{dr}^{\lambda_r} + \frac{d\lambda_{dr}^{\lambda_r}}{dt} - (\omega_{\lambda_r} - \omega_r)\lambda_{qr}^{\lambda r}$$

where:

$$\lambda_{qs}^{\lambda_r} = L_{ls} i_{qs}^{\lambda_r} + L_m \left(i_{qs}^{\lambda_r} + i_{qr}^{\lambda_r} \right)$$
$$\lambda_{ds}^{\lambda_r} = L_{ls} i_{ds}^{\lambda_r} + L_m \left(i_{ds}^{\lambda_r} + i_{dr}^{\lambda_r} \right)$$
$$\lambda_{qr}^{\lambda_r} = L_{lr} i_{qr}^{\lambda_r} + L_m \left(i_{qs}^{\lambda_r} + i_{qr}^{\lambda_r} \right)$$
$$\lambda_{dr}^{\lambda_r} = L_{lr} i_{dr}^{\lambda_r} + L_m \left(i_{ds}^{\lambda_r} + i_{dr}^{\lambda_r} \right)$$

By choosing the phase of the reference system in such a way to arrange the rotor flux exactly on the d-axis, we will have $\lambda_{qr}^{\lambda r} = 0$, $\lambda_{dr}^{\lambda r} = \lambda_{r}$..

With this choice, the electromagnetic torque can be written as:

$$T_e = \frac{3}{2} \times \frac{p}{2} \times \frac{L_m}{L_r} \times (\lambda_{dr}^{\lambda r} \cdot i_{qs}^{\lambda r})$$

i.e. as a product of a flux and a current component (P= number of stator poles).

Let us investigate further on the rotor flux $\lambda_{dr}^{\lambda r}$.

Considering the d-axis rotor flux equation: $\lambda_{dr}^{\lambda r} = L_{lr} i_{dr}^{\lambda r} + L_m \cdot (i_{ds}^{\lambda r} + i_{dr}^{\lambda r})$

then, the equation for $i_{dr}^{\lambda r}$ is: $i_{dr}^{\lambda r} = \dfrac{\lambda_{dr}^{\lambda r} - L_m i_{ds}^{\lambda r}}{L_r}$

Combining the latter with the d-axis rotor voltage equation, leads to:

$$\frac{d\lambda_{dr}^{\lambda r}}{dt} + \frac{r_r}{L_r} \times (\lambda_{dr}^{\lambda r} - L_m i_{ds}^{\lambda r})$$

$$\frac{d}{dt} \times \frac{\lambda_{dr}^{\lambda r}}{L_m} + \frac{1}{\tau_r} \times \frac{\lambda_{dr}^{\lambda r}}{L_m} = \frac{1}{\tau_r} \times i_{ds}^{\lambda r}$$

where τ_r is the rotor time constant, $\tau_r = L_r / r_r$.

Therefore, a lag in flux response is caused to this first order transfer function between $i_{ds}^{\lambda r}$ and $\lambda_{dr}^{\lambda r}$.

The CalcIm routine performs a numerical integration using Euler's method which, for a first order ODE written as $y' = f(t,y)$, may be summarized in this way: $y_{n+1} = y_n + \Delta t \cdot f(t_n, y_n)$, where t is the sampling time.

Putting the equation above in the explicit form, we have: $\left(\dfrac{\lambda_{dr}^{\lambda r}}{L_m}\right)' = \dfrac{1}{\tau_r} \times \left(i_{ds}^{\lambda r} - \dfrac{\lambda_{dr}^{\lambda r}}{L_m}\right)$

$$\left(\frac{\lambda_{dr}^{\lambda r}}{L_m}\right)_{n+1} = \left(\frac{\lambda_{dr}^{\lambda r}}{L_m}\right)_n + \frac{\Delta t}{\tau_r} \times \left((i_{ds}^{\lambda})_n - \left(\frac{\lambda_{dr}^{\lambda r}}{L_m}\right)_n\right)$$

On the other hand, under the same conditions, the q-axis rotor flux equation becomes:

$$\lambda_{qr}^{\lambda r} = L_{lr} i_{qr}^{\lambda r} + L_m \times (i_{qs}^{\lambda r} + i_{qr}^{\lambda r}) = 0$$

So, the equation for $i_{qr}^{\lambda r}$ is: $i_{qr}^{\lambda r} = -\frac{L_m}{L_r} i_{qs}^{\lambda r}$

Combining the last with the q-axis rotor voltage equation, leads to:

$$\omega_{s\lambda} = \omega_{\lambda r} - \omega_r = -\frac{r_r \cdot i_{qr}^{\lambda r}}{\lambda_{dr}^{\lambda r}} = \frac{r_r}{L_r} \times \frac{L_m}{\lambda_{dr}^{\lambda r}} \times i_{qs}^{\lambda r}$$

This equation (implemented in the CalcRotFlxSlipFreq function, see *CalcRotFlxSlipFreq on page 58*) is at the foundation of indirect field oriented control: it tells us that the rotor flux slip frequency $\omega_{s\lambda r}$ may be simply calculated from stator current components (relying on knowledge of the rotor time constant of the machine).

If rotor angle or rotor speed is known (see *Figure 38* and *Figure 39* respectively), then we have managed to determine the rotor flux position $\theta_{\lambda r}$. This information is essential to achieve optimum control.

4.4.4 Detailed explanation about field weakening operation

Many applications need to operate induction machines above their rated speed: this is achieved by means of field weakening.

The conventional method for the field weakening operation is to vary the rotor flux reference in proportion to the inverse of the rotor speed ω_r.

In this approach, if maximum inverter modulation index is required when attaining rated speed and rated power, then the voltage margin, enough to regulate current beyond that point, is not available: this is caused by increased voltage drop across the stator leakage inductance.

That's why, when $1/\omega_r$ method is implemented, the inverter voltage is generally limited at 95% of its means.

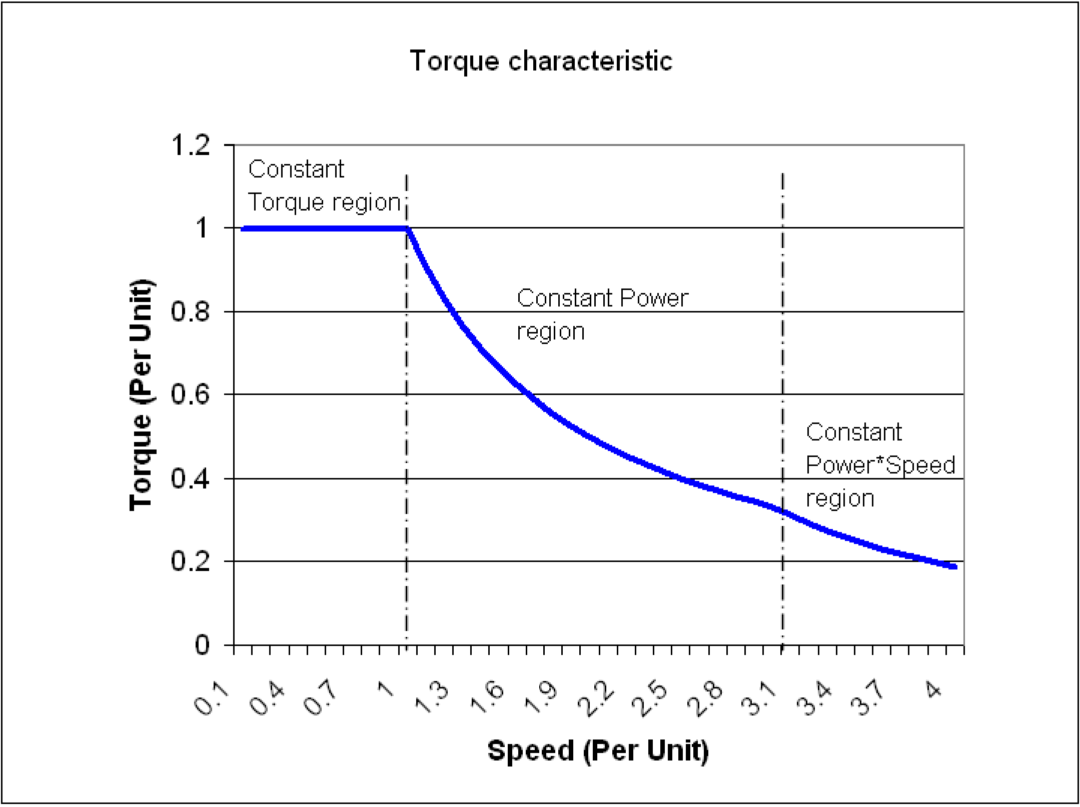
The AC IM IFOC software library, however, makes use of a maximum torque capability scheme (see [2] in Appendix *A.7: References on page 102*), which aims to exploit the system resources completely.

In both cases, DC bus voltage limitation (V_{DCmax}), inverter current rating and motor thermal rating (usually, in order to provide better dynamic response, the inverter current rating is higher than that of the machine) must be considered, and a precise knowledge of motor parameters, such as magnetizing inductance L_m, rotor leakage inductance L_{lr}, rotor resistance r_r, is required.

There are two different field weakening operation regions (see *Figure 41*):

- the constant power region, where rotor flux is decreased inversely with the speed (considering the influence of the voltage drop across L_{ls}) while slip frequency increases until breakdown value;
- the constant power speed region, where rotor flux is decreased, but keeping the slip frequency fixed at breakdown value.

Figure 41. Torque vs. speed characteristic curve



In order to help you select the most suitable values of flux reference and torque saturation (as needed by the CalcRotFlxSlipFreq function), a spreadsheet is available, to be filled out with the following system parameters:

- Mains AC voltage, rms, Volt (cell B1, Volt);
- motor rated current, peak amplitude, (cell B2, Ampere); as said before, this data is to be matched with inverter current rating;
- motor rated magnetizing current, peak amplitude, (cell B3, Ampere);
- magnetizing inductance L_m, (cell B4, Henry);
- leakage inductance L_{ls} (L_{lr}), (cell B5, Henry);
- stator resistance r_s, (cell B6, Ohm)
- rotor resistance r_r, (cell B7, Ohm);
- maximum measurable current I_{max}, peak amplitude, (cell B8, Ampere).

As a result of data processing, the following information can be obtained:

- highest frequency of constant torque region, i.e. the maximum allowable frequency before entering field weakening; content of cell B13 should be inserted (as parameter `RATED_FREQ`) in `MC_ACMotor_Prm.h` (see *Section 2.2.5*);.
- reference values of i_{ds}, in q1.15 format, according to increasing stator frequency; column P should be copied (as FLUX_REFERENCE_TABLE) in `MC_ACMotor_Prm.h`.
- saturation values of current component i_{qs}, in q1.15 format, according to increasing stator frequency; column Q should be copied (as TORQUE_REFERENCE_TABLE) in `MC_ACMotor_Prm.h`.

4.5 Reference frame transformations: `MC_Clarke_Park` module

4.5.1 Overview

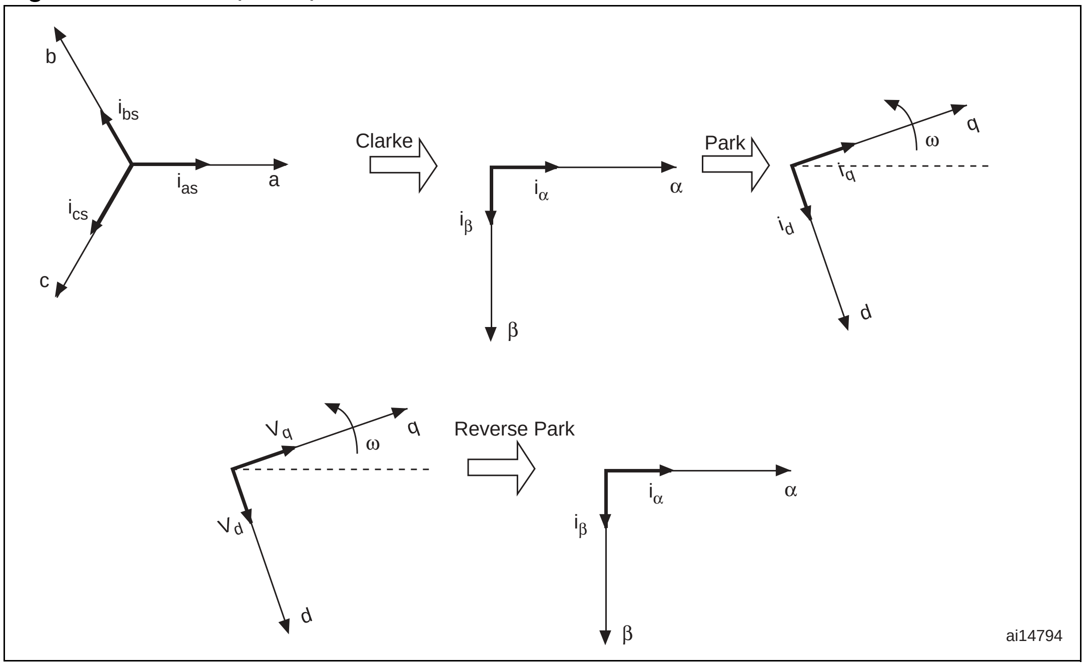
This module, intended for AC machines (induction, synchronous and PMSM), is designed to perform transformations of electric quantities between frames of reference that rotate at different speeds.

Based on the arbitrary reference frame theory, the module provides three functions, named after two pioneers of electric machine analysis, E. Clarke and R.H. Park.

These functions implement three variable changes that are required to carry out field-oriented control (FOC):

- Clarke transforms stator currents to a stationary orthogonal reference frame (named (*q,d)*, see *Figure 42*);
- then, from that arrangement, Park transforms currents to a frame that rotates at an arbitrary speed (which, in IFOC drive, is synchronous with the rotor flux);
- Reverse Park transformation brings back stator voltages from a rotating frame (*q,d)* to a stationary one.
- The module also includes a function to correct the voltage vector command (the so-called “circle limitation”)

Figure 42. Clarke, Park, and reverse Park transformations



4.5.2 List of available C functions

- *Clarke on page 63*
- *Park on page 63*
- *Rev_Park on page 64*
- *Rev_Park_Circle_Limitation on page 64*

`Clarke`

Synopsis	Curr_Components Clarke (Curr_Components Curr_Input)
Description	This function transforms stator currents i_{as} and i_{bs} (which are directed along axes each displaced by 120 degrees) into currents i_{α} and i_{β} in a stationary (α, β) reference frame; α, β axes are directed along paths orthogonal to each other. See *Section 4.5.3* for the details.
Input	Stator currents i_{as} and i_{bs} (in q1.15 format) as members of the variable Curr_Input, which is a structure of type Curr_Components.
Returns	Stator currents i_{α} and i_{β} (in q1.15 format) as members of a structure of type Curr_Components.
Functions called	mul_q15_q15_q31

`Park`

Synopsis	Curr_Components Park (Curr_Components Curr_Input, s16 Theta)
Description	The purpose of this function is to transform stator currents i_{α} and i_{β}, which belong to a stationary (α, β) reference frame, to a rotor flux synchronous reference frame *(q,d)* (properly oriented), so as to obtain i_{qs} and i_{ds}. See *Section 4.5.3* for details.
Input	Stator currents i_{α} and i_{β} (in q1.15 format) as members of the variable Curr_Input, which is a structure of type Curr_Components; rotor flux angle $\theta_{\lambda r}$ (65536 pulses per revolution).
Returns	Stator currents i_{qs} and i_{ds} (in q1.15 format) as members of a structure of type Curr_Components.
Functions called	mul_q15_q15_q31

`Rev_Park`

Synopsis	Volt_Components Rev_Park (Volt_Components Volt_Input)
Description	This function transforms stator voltage v_q and v_d, belonging to a rotor flux synchronous rotating frame, to a stationary reference frame, so as to obtain v_α and v_β. See *Section 4.5.3* for details.
Input	Stator voltages v_{qs} and v_{ds} (in q1.15 format) as members of the variable Volt_Input, which is a structure of type Volt_Components.
Returns	Stator voltages v_α and v_β (in q1.15 format) as members of a structure of type Volt_Components.
Functions called	mul_q15_q15_q31

`Rev_Park_Circle_Limitation`

Synopsis	void RevPark_Circle_Limitation(void)
Description	After the two new values (V_d and V_q) of the stator voltage producing flux and torque components of the stator current, have been independently computed by flux and torque PIDs, it is necessary to saturate the magnitude of the resulting vector, equal to $\sqrt{V_d^2 + V_q^2}$ before passing them to the Rev_Park function. The purpose of this routine is to perform the saturation. Refer to *Section 4.5.4: Circle limitation on page 66* for more detailed information
Input	None.
Returns	None.
Note	The limitation of the stator voltage vector must be done in accordance with the PWM frequency as shown in *Table 2: PWM frequency vs. maximum duty cycle relationship on page 49*.
Functions called	None.

4.5.3 Detailed explanation about reference frame transformations

Induction machines show very complex voltage equations, because of the time-varying mutual inductances between stator and rotor circuits.

By making a change of variables, that refers stator and rotor quantities to a frame of reference rotating at any angular velocity, it is possible to reduce the complexity of these equations.

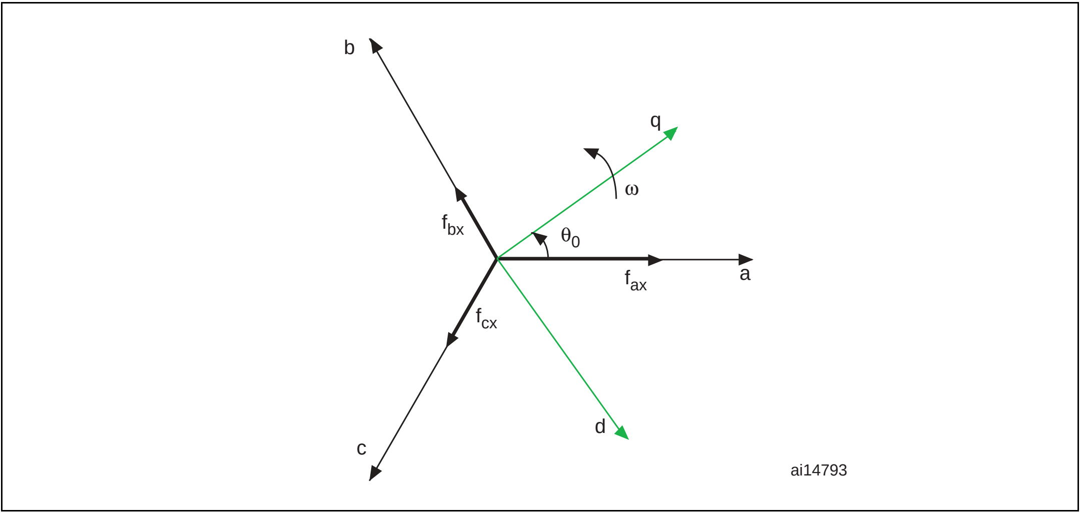
This strategy is often referred to as the Reference-Frame theory (see [1] in Appendix *A.7: References on page 102*).

Supposing f_{ax}, f_{bx}, f_{cx} are three-phase instantaneous quantities directed along axis each displaced by 120 degrees, where x can be replaced with s or r to treat stator or rotor quantities (see *Figure 43*); supposing f_{qx}, f_{dx}, f_{0x} are their transformations, directed along paths orthogonal to each other; the equations of transformation to a reference frame (rotating at an arbitrary angular velocity ω) can be expressed as:

$$f_{qdox} = \begin{bmatrix} f_{qx} \\ f_{dx} \\ f_{0x} \end{bmatrix} = \frac{2}{3} \times \begin{bmatrix} \cos\theta & \cos\left(\theta - \frac{2\pi}{3}\right) & \cos\left(\theta - \frac{2\pi}{3}\right) \\ \sin\theta & \sin\left(\theta - \frac{2\pi}{3}\right) & \sin\left(\theta - \frac{2\pi}{3}\right) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{ax} \\ f_{bx} \\ f_{cx} \end{bmatrix}$$

where θ is the angular displacement of the (q, d) reference frame at the time of observation, and θ_0 that displacement at t=0 (see *Figure 43*).

Figure 43. Transformation from an *abc* stationary frame to a rotating frame (q, d)



With Clark's transformation, stator currents i_{as} and i_{bs} (which are directed along axes each displaced by 120 degrees) are resolved into currents i_α and i_β on a stationary $\alpha\beta$ reference frame.

Appropriate substitution into the general equations (given above) yields:

$$i_\alpha = i_{as}$$

$$i_\beta = -\frac{i_{as} + 2i_{bs}}{\sqrt{3}}$$

In Park's change of variables, stator currents i_α and i_β, which belong to a stationary (α, β) reference frame, are resolved to a rotor flux synchronous reference frame (properly oriented), so as to obtain i_{qs} and i_{ds}.

Consequently, with this choice of reference, $\omega = \omega_{\lambda r}$; thus:

$$i_{qs} = i_\alpha \cos\theta_r - i_\beta \sin\theta_r$$

$$i_{ds} = i_\alpha \sin\theta_r + i_\beta \cos\theta_r$$

On the other hand, reverse Park transformation takes back stator voltage v_q and v_d, belonging to a rotor flux synchronous rotating frame, to a stationary reference frame, so as to obtain v_α and v_β:

$$v_\alpha = v_{qs} \cos\theta_r + v_{ds} \sin\theta_r$$

$$v_\beta = -v_{qs} \sin\theta_r + v_{ds} \cos\theta_r$$

4.5.4 Circle limitation

As discussed above, FOC allows to separately control the torque and the flux of a 3-phase permanent magnet motor. After the two new values(V_d^* and V_q^*) of the stator voltage producing flux and torque components of the stator current, have been independently computed by flux and torque PIDs, it is necessary to saturate the magnitude of the resulting vector ($|\vec{V}^*|$) before passing them to the Reverse Park transformation and, finally, to the SVPWM block.

The saturation boundary is normally given by the value (S16_MAX=32767) which produces the maximum output voltage magnitude (corresponding to a duty cycle going from 0% to 100%).

Nevertheless, when using three shunt resistor configuration and depending on PWM frequency, it might be necessary to limit the maximum PWM duty cycle to guarantee the proper functioning of the stator currents reading block.

For this reason, the saturation boundary could be a value slightly lower than S16_MAX depending on PWM switching frequency when using three shunt resistor configuration.

Table 2 on page 49, repeated below for convenience, shows the maximum applicable modulation index as a function of the PWM switching frequency when using the STM3210B-MCKIT.

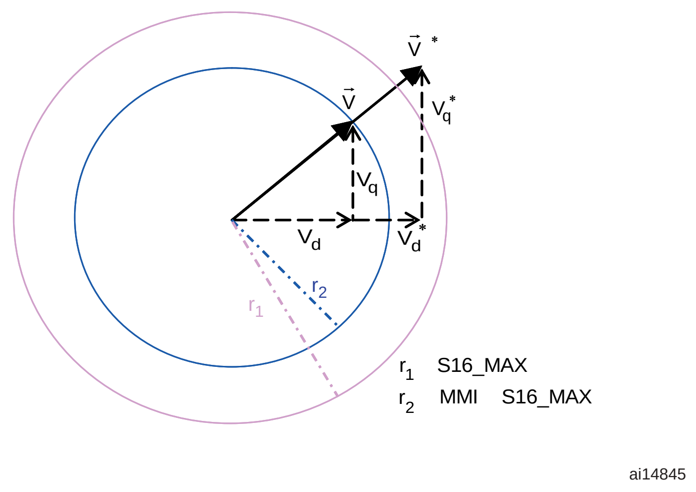
Table 3. PWM frequency vs. maximum duty cycle relationship

PWM frequency	Max duty cycle	Max modulation Index
Up to 11.4 kHz	100%	100%
12.2 kHz	99%	98%
12.9 kHz	98.5%	97%
13.7 kHz	98%	96%
14.4 kHz	98%	96%
15.2 kHz	97%	94%
16 kHz	96.5%	93%
16.7 kHz	96.5%	93%
17.5 kHz	95.5%	91%

Note: *The figures above were measured using the MB459 board. This evaluation platform is designed to support several motor driving topologies (PMSM and AC induction) and current reading strategies (single and three shunt resistors). Therefore, the figures provided in should be understood as a starting point and not as a best case.*

The `RevPark_Circle_Limitation` function performs the discussed stator voltage components saturation, as illustrated in *Figure 44*.

Figure 44. Circle limitation working principle



V_d and V_q represent the saturated stator voltage component to be passed to the Reverse Park transformation function, while V_d* and V_q* are the outputs of the PID current controllers. From geometrical considerations, it is possible to draw the following relationship:

$$V_d = \frac{V_d^* \cdot MMI \cdot S16\_MAX}{\left|\vec{V}^*\right|}$$

$$V_q = \frac{V_q^* \cdot MMI \cdot S16\_MAX}{\left|\vec{V}^*\right|}$$

In order to speed up the computation of the above equations while keeping an adequate resolution, the value

$$\frac{MMI \cdot S16\_MAX^2}{\left|\vec{V}^*\right|}$$

is computed and stored in a look-up table for different values of $\left|\vec{V}^*\right|$. Furthermore, considering that MMI depends on the selected PWM frequency, a number of look-up tables are stored in ‘MC_Clarke_Park.c’ (with MMI ranging from 92 to 100).

Once you have selected the required PWM switching frequency, you should uncomment the Max Modulation Index definition corresponding to the selected PWM frequency in the `MC_Control_Param.h` definitions list shown below.

```
//#define MAX_MODULATION_100_PER_CENT     // up to 11.4 kHz PWM frequency
//#define MAX_MODULATION_99_PER_CENT      // up to 11.8 kHz
//#define MAX_MODULATION_98_PER_CENT      // up to 12.2 kHz
//#define MAX_MODULATION_97_PER_CENT      // up to 12.9 kHz
#define MAX_MODULATION_96_PER_CENT      // up to 14.4 kHz
//#define MAX_MODULATION_95_PER_CENT      // up to 14.8 kHz
//#define MAX_MODULATION_94_PER_CENT      // up to 15.2 kHz
//#define MAX_MODULATION_93_PER_CENT      // up to 16.7 kHz
//#define MAX_MODULATION_92_PER_CENT      // up to 17.1 kHz
//#define MAX_MODULATION_91_PER_CENT      // up to 17.5 kHz
```

For information on selecting the PWM switching frequency, you will find advice in *Section A.2 on page 93*. To determine the max modulation index corresponding to the PWM switching frequency, refer to *Table 2 on page 49*.

4.6 Encoder feedback processing: `stm32f10x_encoder` module

4.6.1 List of available functions and interrupt service routines

The following is a list of available functions as listed in the `stm32f10x_encoder.h` header file:

- *`ENC_Init` on page 69*
- *`ENC_Get_Electrical_Angle` on page 69*
- *`ENC_Get_Mechanical_Angle` on page 69*
- *ENC_Clear_Speed_Buffer on page 70*
- *`ENC_Get_Mechanical_Speed` on page 70*
- *`ENC_Calc_Average_Speed` on page 70*
- *ENC_ErrorOnFeedback on page 70*
- *TIMx_IRQHandler - interrupt routine on page 71*

`ENC_Init`

Synopsis	void ENC_Init(void)
Description	The purpose of this function is to initialize the encoder timer. The peripheral clock, input pins and update interrupt are enabled. The peripheral is configured in 4X mode, which means that the counter is incremented/decremented on the rising/falling edges of both timer input 1 and 2 (TIMx_CH1 and TIMx_CH2 pins).
Functions called	RCC_APB1PeriphClockCmd, RCC_APB2PeriphClockCmd, GPIO_StructInit, GPIO_Init, NVIC_Init, TIM_DeInit, TIM_TimeBaseStructInit, TIM_TimeBaseInit, TIM_EncoderInterfaceConfig, TIM_ICInit, TIM_ClearFlag, TIM_ITConfig, TIM_Cmd
See also	STM32F103xx datasheet: synchronizable standard timer.

`ENC_Get_Electrical_Angle`

Synopsis	s16 ENC_Get_Electrical_Angle(void)
Description	This function returns the electrical angle in signed 16-bit format. This routine returns: 0 for 0 degrees, -32768 (S16_MIN) for -180 degrees, +32767 (S16_MAX) for +180 degrees.
Input	None
Output	Signed 16 bits
Functions called	None

`ENC_Get_Mechanical_Angle`

Synopsis	s16 ENC_Get_Mechanical_Angle(void)
Description	This function returns the mechanical angle in signed 16-bit format. This routine returns: 0 for 0 degrees, -32768 (S16_MIN) for -180 degrees, +32767 (S16_MAX) for +180 degrees.
Input	None
Output	Signed 16 bits
Functions called	None
Note	Link between Electrical/Mechanical frequency/RPM: Electrical frequency = number of pair poles x mechanical frequency RPM speed = 60 x Mechanical frequency (RPM: revolutions per minute) **Example**: electrical frequency = 100 Hz, motor with 8 pair poles: *100Hz electrical <-> 100/8 =12.5Hz mechanical <-> 12.5 x 60=750 RPM*

`ENC_Clear_Speed_Buffer`

Synopsis	void ENC_Clear_Speed_Buffer(void)
Description	This function resets the buffer used for speed averaging.
Functions called	None

`ENC_Get_Mechanical_Speed`

Synopsis	s16 ENC_Get_Mechanical_Speed(void)
Description	This function returns the rotor speed in Hz. The value returned is given with 0.1Hz resolution, which means that 1234 is equal to 123.4 Hz.
Input	None
Output	Signed 16 bits
Functions called	None
Note	This routine returns the mechanical frequency of the rotor. To find the electrical speed, use the following conversion: *electrical frequency = number of pole pairs * mechanical frequency*

`ENC_Calc_Average_Speed`

Synopsis	void ENC_Calc_Average_Speed(void)
Description	This function must be called every `SPEED_MEAS_TIMEBASE` ms; it computes the latest speed measurement, if it is out of the range specified in `MC_encoder_param.h`, then the error counter is incremented and the speed is saturated. Furthermore, if the error counter is higher than `MAXIMUM_ERROR_NUMBER`, the boolean variable storing the error status is set. Finally, the new average value is computed based on the latest `SPEED_BUFFER_SIZE` speed measurement.
Functions called	ENC_Calc_Rot_Speed
Input	None
Returns	None

`ENC_ErrorOnFeedback`

Synopsis	bool ENC_ErrorOnFeedback(void)
Description	This function simply returns the status of the boolean variable containing the speed measurement error status which is updated every `SPEED_MEAS_TIMEBASE` ms by the `ENC_Calc_Average_Speed` function. In the proposed firmware library this function is called in Run state by the main to check for possible faults of the speed feedback (such as disconnected encoder wires).
Functions called	None
Input	None
Returns	boolean, TRUE if an error occurred, FALSE otherwise.

`TIMx_IRQHandler` - interrupt routine

Synopsis	void TIMx_IRQHandler(void)
Description	This is the encoder timer (TIMER 2, 3 or 4) update routine. An interruption is generated whenever an overflow/underflow of the counter value occurs (TIMx_CNT). The ‘Encoder_Timer_Overflow’ variable is then incremented.
Functions called	None
Note	This is an interrupt routine.
See also	STM32F103xx reference manual: TIMx in encoder interface mode.

4.7 Tachogenerator feedback processing: `stm32f10x_tacho` module

4.7.1 List of available functions and interrupt service routines

The following is a list of available functions as listed in the `stm32f10x_tacho.h` header file:

- *`TAC_TachoTimerInit` on page 72*
- *`TAC_InitTachoMeasure` on page 72*
- *`TAC_GetRotorFreqInHz` on page 72*
- *`TAC_GetRotorFreq` on page 73*
- *`GetLastTachoPeriod` on page 73* (this is a private function defined inside the `stm32f10x_tacho.c` file)
- *`GetAvrgTachoPeriod` on page 73* (this is a private function defined inside the `stm32f10x_tacho.c` file)
- *`TAC_IsTimedOut` on page 74*
- *`TAC_ClrTimeOut` on page 74*
- *`TAC_GetCaptCounter` on page 74*
- *`TAC_ClrCaptCounter` on page 74*
- *`TAC_StartTachoFiltering` on page 75*
- *`TAC_ValidSpeedInfo` on page 75*
- *`TIMx_IRQHandler` on page 76*

`TAC_TachoTimerInit`

Synopsis	void TAC_TachoTimerInit(void)
Description	The purpose of this function is to initialize the timer that will perform the tacho signal period measurement (the timer can be chosen in the MC_tacho_prm.h file). The peripheral clock and the capture interrupt are enabled, and the timer is initialized in Slave reset mode.
Functions called	TIM_ICStructInit, TIM_TimeBaseInit, TIM_ICInit, TIM_PrescalerConfig, TIM_InternalClockConfig, TIM_SelectInputTrigger, TIM_SelectSlaveMode, TIM_UpdateRequestConfig, NVIC_Init, TIM_ClearFlag, TIM_ITConfig
Note	The timer starts counting at the end of the routine.
See also	STM32F103xx datasheet: general-purpose timer (TIMx).

`TAC_InitTachoMeasure`

Synopsis	void TAC_InitTachoMeasure(void)
Description	This function clears the software FIFO where the latest speed data are stored. This function must be called every time the motor is started to initialize the speed measurement process.
Input	None.
Output	None.
Functions called	TIM_ITConfig, TIM_Cmd
Note	The first measurements following this function call are done without filtering (the rolling average mechanism is disabled).
See also	STM32F103xx datasheet: general-purpose timer (TIMx).

`TAC_GetRotorFreqInHz`

Synopsis	u16 TAC_GetRotorFreqInHz (void)
Description	This routine returns the rotor frequency with [0.1Hz] definition. The result is given by the following formula: F_{rotor} = K x (F_{OSC} / (Capture + number of overflow x FFFF)) where K depends on the number of motor and tacho pole pairs.
Input	None.
Output	Rotor mechanical frequency, with 0.1 Hz resolution, unsigned 16 bits (direction cannot be determined using a tacho).
Functions called	GetAvrgTachoPeriod, GetLastTachoPeriod (both private functions)
Note	Result is zero if speed is too low (glitches at start for instance). Excessive speed (or glitches) will result in a pre-defined value returned (see *Section 2.2.4 on page 19*). Maximum expectable accuracy depends on CKTIM: 72 MHz will give the best results.
Caution	This routine returns the mechanical frequency of the rotor. To find the electrical speed, use the following conversion: *electrical frequency = mechanical frequency * number of pole pairs*

`TAC_GetRotorFreq`

Synopsis	u16 TAC_GetRotorFreq (void)
Description	This routine returns rotor frequency with a unit that can be directly integrated (accumulated) to get the rotor angular position in the main control loop.
Input	None.
Output	Rotor mechanical frequency with rad/PWM period unit (2π rad = 0xFFFF), assuming the control loop is executed in each and every PWM interrupt service routine.
Functions called	GetAvrgTachoPeriod, GetLastTachoPeriod (both private functions)
Note	Result is zero if speed is too low (glitches at start for instance). Excessive speed (or glitches) will result in a pre-defined value returned (see *Section 2.2.4 on page 19*). Maximum expectable accuracy depends on CKTIM: 72 MHz will give the best results.

`GetLastTachoPeriod`

Synopsis	u32 GetLastTachoPeriod(void)
Description	This routine returns the rotor period based on the last tacho capture.
Input	None.
Output	Tacho signal period, unit is 1 CKTIM period, unsigned 32-bit format.
Functions called	None.
Note	This function is private to the stm32f10x_tacho.c module.

`GetAvrgTachoPeriod`

Synopsis	u32 GetAvrgTachoPeriod(void)
Description	This routine returns the rotor period based on the average of the four last tacho captures.
Input	None.
Output	Tacho signal period, unit is 1 CKTIM period, unsigned 32-bit format.
Functions called	None.
Note	This function is private to the stm32f10x_tacho.c module.

`TAC_IsTimedOut`

Synopsis	bool TAC_IsTimedOut(void)
Description	This routine indicates to the upper layer software that tacho information has disappeared (or that the period of the signal has drastically increased).
Input	None.
Output	Boolean, TRUE in case of time-out
Functions called	None.
Note	The time-out duration depends on tacho timer pre-scaler, which is variable: the time-out is higher at low speed. The boolean will remain set to TRUE until the TAC_ClrTimeOut is called.

`TAC_ClrTimeOut`

Synopsis	void TAC_ClrTimeOut (void)
Description	This routine clears the flag indicating that information is lost, or that speed is decreasing sharply.
Input	None.
Output	None.
Note	This function must be called to re-arm the time-out detection mechanism and re-start rotor frequency measurements: the returned frequency is 0 as long as the timeout flag is set.

`TAC_GetCaptCounter`

Synopsis	u16 TAC_GetCaptCounter(void)
Description	This routine gives the number of tacho capture interrupts since the last call to the TAC_ClrCaptCounter function.
Input	None.
Output	Unsigned 16-bit integer. This variable cannot roll-over (this is prevented in the tacho capture routine itself): it will be limited to max u16 value.
Note	This function is typically used to monitor the interrupts activity (while the motor is running, tacho-related interrupts must not be stopped or too frequent).
See also	TAC_ClrCaptCounter

`TAC_ClrCaptCounter`

Synopsis	void TAC_ClrCaptCounter(void)
Description	This routine clears the number of capture events variable.
Input	None.
Output	None.

`TAC_StartTachoFiltering`

Synopsis	void TAC_StartTachoFiltering(void)
Description	This routine initiates the tacho value smoothing mechanism. The result of the next capture will be copied in all storage array locations to have the first average equal to the last value.
Input	None.
Output	None.
Note	The initialization of the FIFO used to do the averaging will be done when the next tacho capture interrupt occurs. Consequently, the TAC_GetRotorFreq will continue to return a raw period value until the next interrupt event.

`TAC_ValidSpeedInfo`

Synopsis	bool TAC_ValidSpeedInfo(u16 hMinRotorFreq)
Description	This routine indicates if the information provided by the tachogenerator is reliable: this is particularly important at startup, when the signal of the tacho is very weak and cannot be properly conditioned by the external circuitry (glitches). It is also used in startup functions to find out if the rotor shaft is turning at the right speed.
Input	Rotor frequency (0.1Hz resolution) above which speed information is not considered reliable (rolling averages cannot be computed).
Output	Boolean, TRUE if the tacho provides clean signals.
Caution	Because there is no way to differentiate rotation direction with a tachogenerator, you must be aware that this routine may return TRUE in certain conditions (re-start with very short or no stop time and high inertia load). You should, therefore, manage a minimal amount of time before re-starting. This function is not effective if the startup duration (time for the voltage to settle) is much shorter than the time needed to obtain at least two consecutive speed data.

`TIMx_IRQHandler`

Synopsis	void TIMx_IRQHandler(void)
Description	This function handles ● the capture event interrupt in charge of tacho signal period measurement. It updates an array where the 4 latest period measurements are stored, resets the overflow counter and updates the clock prescaler to optimize the accuracy of the measurement. If the average is enabled, the last captured measurement is copied into the whole array. Period captures are managed as follows: – If too low (capture value below 0x5500), the clock prescaler is decreased for the next measurement – If too high (for example, the timer overflowed), the result is re-computed as if there was no overflow and the prescaler is increased to avoid overflows during the next capture. ● the overflow of the timer in charge of the tacho signal period measurement. It updates a Counter of overflows, which is reset when next capture occurs
Input	None.
Output	None.
Note	This is an interrupt routine.

4.7.2 Integration tips

In the `MC_tacho_prm.h` file of your project, select the Timer you have chosen and the input channel on which the tacho signal arrives, using the right `#define` (see *Section 2.2.4 on page 19*).

In the `main.c` module (or the c module just above `stm32f10x_tacho`), include the `stm32f10x_tacho.h` file, call `TAC_TachoTimerInit()` after MCU reset and `TAC_InitTachoMeasure()` before motor startup. `TAC_GetRotorFreqInHz` returns a frequency directly with 0.1Hz, while `TAC_GetRotorFreq` returns a value that can be directly accumulated in the FOC algorithm to get the rotor angular position (the unit is 2π rad (that is 0xFFFF) per sampling period).

4.7.3 Operating principle

Although the principle of measuring a period with a timer is quite simple, you must pay attention to keeping the best resolution, in particular for signals such as the one provided by a tachogenerator, which can vary with a ratio of up to 1:100.

In order to have always the best resolution, the timer clock prescaler is constantly adjusted in the current implementation.

The basic principle is to speed-up the timer if captured values are too low (for an example of low periods, see *Figure 45*), and slow it down when the timer overflows between two consecutive captures (see example of large periods in *Figure 46*).

The prescaler modification is done in the capture interrupt, taking advantage of the buffered registers: the new prescaler value is taken into account only on the next capture event, by the hardware, without disturbing the measurement.

Further details are provided in the flowcharts in *Section A.4 on page 96*.

Figure 45. Automatic tacho timer prescaler decrease

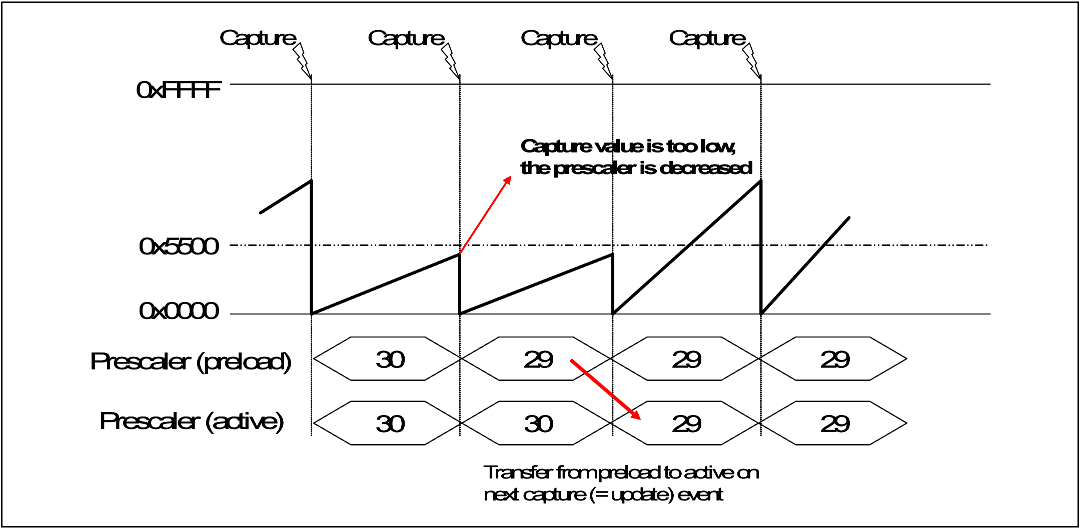

Capture
Capture
Capture
Capture
0xFFFF
Capture value is too low, the prescaler is decreased
0x5500
0x0000
Prescaler (preload)
30
29
29
29
Prescaler (active)
30
30
29
29
Transfer from preload to active on next capture (= update) event

Figure 46. Automatic tacho timer prescaler increase

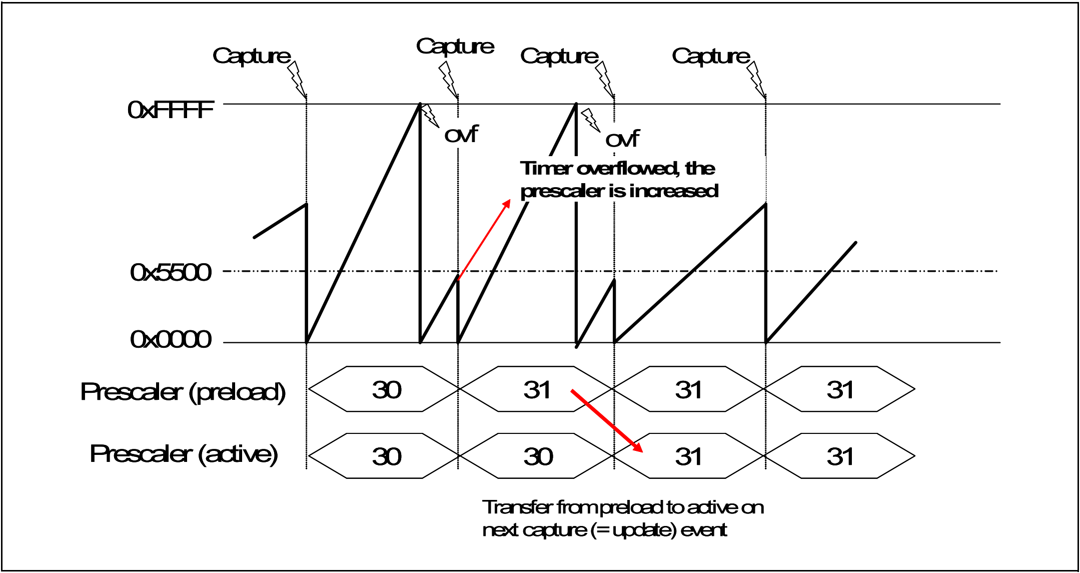

Capture
Capture
Capture
Capture
0xFFFF
ovf
ovf
Timer overflowed, the prescaler is increased
0x5500
0x0000
Prescaler (preload)
30
31
31
31
Prescaler (active)
30
30
31
31
Transfer from preload to active on next capture (= update) event

Figure 46 shows that the prescaler is not decreased although the captured value is below 0x5500, due to an overflow interrupt.

4.7.4 Converting Hertz into pseudo frequency

From the definition of frequency (1Hz is equal to 2π rad.s^{-1}), it is easy to define a pseudo frequency format, so that the rotor angular position can be easily determined by accumulating the rotor speed information every time the control loop is executed (for example, during PWM update interrupt service routine). Providing that 2π = 0xFFFF (so that angle roll-overs do not need to be managed), the frequency with 0.1Hz unit can easily be converted into pseudo frequency using the following formula:

$$F_{pseudo} = F_{[0.1Hz]} \times \frac{0xFFFF}{10 \times F_{pwm(Hz)}}$$

4.8 Flux, torque and speed regulators: `MC_PID_regulators` module

4.8.1 Overview

The `MC_PID_regulators` module contains all the functions required for implementing the necessary PID regulators for controlling flux, torque and, in case of Speed control mode, motor speed.

4.8.2 List of available functions and interrupt service routines

The following is a list of available functions in the `MC_PID_regulators` module:

- *`PID_Init` on page 78*
- *`PID_Flux_Regulator` on page 79*
- *`PID_Torque_Regulator` on page 79*
- *`PID_Speed_Regulator` on page 79*
- *`PID_Reset_Integral_terms` on page 80*
- *`PID_Speed_Coefficients_update` on page 80*
- *`PID_Integral_Speed_update` on page 80*

`PID_Init`

Synopsis	void PID_Init(void)
Description	The purpose of this function is to initialize the PIDs for torque, flux and speed regulation. For each one, a set of default values are loaded: target (speed, torque or flux), proportional, integral and derivative gains, lower and upper limiting values for the output.
Functions called	None
Note	Default values for PID regulators are declared and can be modified in the MC_Control_Param.h file (see *Section 2.2.2 on page 16*).

`PID_Flux_Regulator`

Synopsis	s16 PID_Flux_regulator(PID_FluxTYPEDEF *PID_Flux, s16 qId_input)
Description	The purpose of this function is to compute the proportional, integral and derivative terms (if enabled, see ıd_Iq_DIFFERENTIAL_TERM_ENABLED in *Section 2.2.1 on page 15*) for the flux regulation.
Input	PID_FluxTYPDEF (see MC_type.h for structure declaration) signed 16 bits
Output	Signed 16 bits
Functions called	None
Note	Default values for the PID flux regulation are declared and can be modified in the MC_Control_Param.h file (see *Section 2.2.2 on page 16*).
See also	*Figure 55 on page 98* shows the PID block diagram.

`PID_Torque_Regulator`

Synopsis	s16 PID_Torque_regulator(PID_TorqueTYPEDEF *PID_Torque, s16 qIq_input)
Description	The purpose of this function is to compute the proportional, integral and derivative terms (if enabled, see Id_Iq_DIFFERENTIAL_TERM_ENABLED in *Section 2.2.1 on page 15*) for the torque regulation.
Input	PID_TorqueTYPDEF (see MC_type.h for structure declaration) signed 16 bits
Output	signed 16 bits
Functions called	None
Note	Default values for the PID torque regulation are declared and can be modified in the MC_Control_Param.h file (see *Section 2.2.2 on page 16*).
See also	*Figure 55 on page 98* shows the PID block diagram.

`PID_Speed_Regulator`

Synopsis	s16 PID_Speed_regulator(PID_SpeedTYPEDEF *PID_Speed, s16 speed)
Description	The purpose of this function is to compute the proportional, integral and derivative terms (if enabled, see SPEED_DIFFERENTIAL_TERM_ENABLED in *Section 2.2.1 on page 15*) for the speed regulation.
Input	PID_SpeedTYPDEF (see MC_type.h for structure declaration) signed 16 bits

Output	signed 16 bits
Functions called	None
Caution	Default values for the PID speed regulation are declared and can be modified in the MC_Control_Param.h file (see *Section 2.2.2 on page 16*).
See also	*Figure 56 on page 99* shows the PID block diagram.

`PID_Reset_Integral_terms`

Synopsis	void PID_Reset_Integral_terms(void)
Description	The purpose of this function is to reset all the integral terms of the torque, flux and speed PID regulators.

`PID_Speed_Coefficients_update`

Synopsis	void PID_Speed_coefficients_update(s16 motor_speed)
Description	This function automatically computes the proportional, integral and derivative gain for the speed PID regulator according to the actual motor speed. The computation is done following a linear curve based on 4 set points. See *Section 4.8.5 on page 82* for more information.
Functions called	None
Caution	Default values for the four set points are declared and can be modified in the MC_Control_Param.h file (see *Section 2.2.2 on page 16*).

`PID_Integral_Speed_update`

Synopsis	void PID_Integral_Speed_update(s32 value)
Description	The purpose of this function is to load the speed integral term with a default value.

4.8.3 PID regulator theoretical background

The regulators implemented for Torque, Flux and Speed are actually Proportional Integral Derivative (PID) regulators (see note below regarding the derivative term). 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.

Figure 47. PID general equation

$$\left.\begin{array}{r}\text{torque} = \text{f(rotor position)}\\ \text{flux} = \text{f(rotor position)}\end{array}\right\}\ \text{torque and flux regulation for maximum system efficiency}$$

$$\text{torque} = \text{f(rotor speed)}\ \}\ \text{torque regulation for speed regulation of the system}$$

Where: $Error_{sys_T}$ Error of the system observed at time t = T

$Error_{sys_{T-1}}$ Error of the system observed at time t = T - Tsampling

$$f(X_T) = K_p \times Error_{sys_T} + K_i \times \sum_{0}^{T} Error_{sys_t} + \underbrace{K_d \times (Error_{sys_T} - Error_{sys_{T-1}})}_{\textit{Derivative term can be disabled}} \quad (1)$$

Equation 1 corresponds to a classical PID implementation, where:

- K_p is the proportional coefficient,
- K_i is the integral coefficient.
- K_d is the differential coefficient.

Note: *As mentioned in Figure 47, the derivative term of the PID can be disabled independently (through a compiler option, see `stm32f10x_MCconf.h` file) for the torque/flux or the speed regulation; a PI can then be quickly implemented whenever the system doesn't require a PID control algorithm.*

4.8.4 Regulator sampling time setting

The sampling time 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 shown in equation (1) in *Figure 47*, the sampling time is directly part of the integral coefficient, thus avoiding an extra multiplication. *Figure 48* describes the link between the time domain and the discrete system.

Figure 48. Time domain to discrete PID equations

Time domain

$$f(t) = K_p \times Error_{sys}(t) + K_i \times \int_0^t Error_{sys}(t)dt + K_d \times \frac{d}{dt}(Error_{sys}(t))$$

$$\updownarrow$$

Discrete domain

$$f(X_T) = K_p \times Error_{sys_T} + \boxed{k_i \times T_s}\sum_0^T Error_{sys_t} + K_d \times (Error_{sys_T} - Error_{sys_{T-1}})$$

(sampling done at Fs = 1/Ts frequency)

$$\hookrightarrow k_i \times T_s = K_i$$

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

- The related CPU load will grow 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 a tacho-generator sensor is used while driving the motor at low to medium speed.

As discussed in *Section 2.2.2 on page 16*, the speed regulation loop sampling time can be customized by editing the `PID_SPEED_SAMPLING_TIME` parameter in the `MC_Control_Param.h` header file. The flux and torque PID regulator sampling rates are given by the relationship

$$\text{Flux and torque PID sampling rate} = \frac{2 \cdot \text{PWM\_FREQ}}{\text{REP\_RATE} + 1}$$

Note: *`REP_RATE` must be an odd number if currents are measured by shunt resistors (see also Section A.2 on page 93); its value is 8-bit long.*

4.8.5 Adjusting speed regulation loop Ki, Kp and Kd vs. motor frequency

Depending on the motor frequency, it might be necessary to use different values of Kp, Ki and Kd.

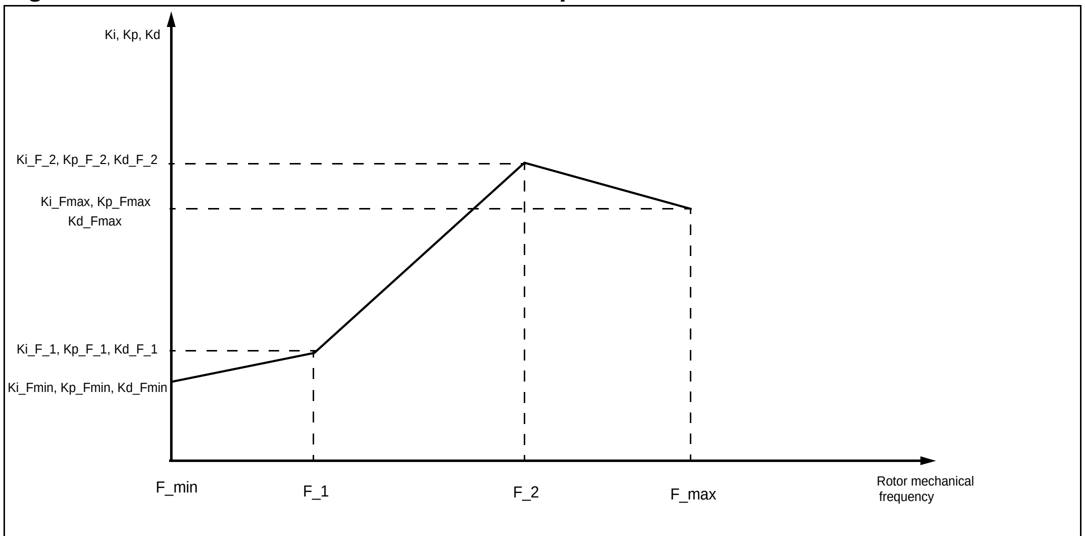
These values have to be input in the code to feed the regulation loop algorithm. A function performing linear interpolation between four set-points (`PID_Speed_Coefficient_update`) is provided as an example in the software library (see `MC_PID_regulators.c`) and can be used in most cases, as long as the coefficient values can be linearized. If that is not possible, a function with a larger number of set-points or a look-up table may be necessary.

To enter the four set-points, once the data are collected, edit the `MC_Control_param.h` file and fill in the field dedicated to the Ki, Kp and Kd coefficient calculation as shown below.

```
//Settings for min frequency
#define Freq_Min   10      // 1 Hz mechanical
#define Ki_Fmin    1000    // Frequency min coefficient settings
#define Kp_Fmin    2000
#define Kd_Fmin    3000

//Settings for intermediate frequency 1
#define F_1                 50 // 5 Hz mechanical
#define Ki_F_1     2000    // Intermediate frequency 1 coefficient settings
#define Kp_F_1     1000
#define Kd_F_1     2500

//Settings for intermediate frequency 2
#define F_2                 200 // 20 Hz mechanical
#define Ki_F_2     1000     // Intermediate frequency 2 coefficient settings
#define Kp_F_2     750
#define Kd_F_2     1200

//Settings for max frequency
#define Freq_Max   500     // 50 Hz mechanical
#define Ki_Fmax    500     // Frequency max coefficient settings
#define Kp_Fmax    500
#define Kd_Fmax    500
```

Once the motor is running, integer, proportional and derivative coefficients are computed following a linear curve between F_min and F_1, F_1 and F_2, F_2 and F_max (see *Figure 49*). Note that F_min, F_1, F_2, F_max are mechanical frequencies, with 0.1 Hz resolution (for example F_1 = 1234 means F_1 = 123.4Hz).

Figure 49. Linear curve for coefficient computation



Disabling the linear curve computation routine, `stm32f10x_Timebase` module

If you want to disable the linear curve computation, you must comment out the `PID_Speed_Coefficients_update(..)` routine. In this case, the default values for Ki, Kp, Kd for torque, flux and speed regulation are used. See `PID_TORQUE_Kx_DEFAULT`, `PID_FLUX_Kx_DEFAULT`, `PID_SPEED_Kx_DEFAULT`, in the `MC_control_Param.h` file.

To disable the linear curve computation routine in `stm32f10x_Timebase.c`:

```
void SysTickHandler(void)
{
[…]
 if(State == RUN)
 {
  if ((wGlobal_Flags & CLOSED_LOOP) == CLOSED_LOOP)
  {
    […]
   //PID_Speed_Coefficients_update(hRot_Freq_Hz); //to be commented out
     […]
}
```

4.9 Main interrupt service routines: `stm32f10x_it` module

4.9.1 Overview

The stm32f10x_it module can be used to describe all the exception subroutines that might occur within your application. When an interrupt happens, the software will automatically branch to the corresponding routine accordingly with the interrupt vector table.

With the exception of the ADC and TIM1 Break Input interrupt requests, all the routines are empty, so that you can write your own code for exception handlers and peripheral interrupt requests.

4.9.2 List of non-empty interrupt service routines

As mentioned above only two interrupts are managed by motor control tasks:

- *TIM1_BRK_IRQHandler on page 85*
- *TIM1_UP_IRQHandler on page 85*
- *ADC_IRQHandler on page 85*

`TIM1_BRK_IRQHandler`

Synopsis	void TIM1_BRK_IRQHandler(void)
Description	The purpose of this function is to manage a break input signal on the dedicated BRK IN pin. In particular, TIM1 outputs are disabled, the main state machine is put into FAULT state.
Input	None.
Returns	None.
Functions called	MCL_SetFault, TIM1_ClearITPendingBit
See also	Advanced control timer (TIM1) in STM32F103xx reference manual

`TIM1_UP_IRQHandler`

Synopsis	void TIM1_UP_IRQHandler(void)
Description	This interrupt handler is executed after an update event when an underflow of the TIM1 counter occurs. It is used to externally trigger the ADC when a rising edge of the TIM1_OC4Ref signal occurs.
Input	None.
Returns	None.
Functions called	ADC_ClearFlag, TIM1_ClearFlag

`ADC_IRQHandler`

Synopsis	void ADC_IRQHandler(void)
Description	The purpose of this function is to handle the ADC interrupt request. The end of the stator current conversion interrupt routine (JEOC) is utilized to trigger execution of the IFOC algorithm. External ADC triggering is disabled until the next TIM1 update event is generated.
Input	None.
Returns	None.
Functions called	IFOC_Model
See also	*Section 4.2.4* and *Section 4.3.3 on page 52* for more details.

4.10 General purpose time base: `stm32f10x_Timebase` module

4.10.1 Overview

The purpose of the `stm32f10x_Timebase` module is to generate a time base that can be used by the other modules of the applications.

4.10.2 List of available functions and interrupt service routines

The following is a list of available functions as listed in the `stm32f10x_Timebase.h` header file:

- *`TB_Init` on page 86*
- *TB_Wait on page 86*
- *`TB_StartUp_Timeout_IsElapsed, TB_Delay_IsElapsed, TB_DisplayDelay_IsElapsed, TB_DebounceDelay_IsElapsed` on page 87*
- *`TB_Set_Delay_500us, TB_Set_DisplayDelay_500us, TB_Set_StartUp_Timeout, TB_Set_DebounceDelay_500µs` on page 87*
- *SysTickHandler on page 88*

`TB_Init`

Synopsis	void TB_Init(void)
Description	The purpose of this function is to initialize the STM32 system tick timer to generate an interrupt every 500 µs, thus providing a general purpose timebase.
Input	None
Returns	None
Functions called	SysTick_CLKSourceConfig, SysTick_SetReload, SysTick_CounterCmd, NVIC_SystemHandlerPriorityConfig, SysTick_ITConfig

`TB_Wait`

Synopsis	void TB_Wait(u16 time)
Description	This function produces a programmable delay equal to variable ‘time’ multiplied by 500µs.
Input	Unsigned 16 bit
Returns	None
Functions called	None
Caution	This routine exits only after the programmed delay has elapsed. Meanwhile, the code execution remains frozen in a waiting loop. Care should be taken when this routine is called at main/interrupt level: a call from an interrupt routine with a higher priority than the timebase interrupt will freeze code execution.

`TB_Set_Delay_500us, TB_Set_DisplayDelay_500us, TB_Set_StartUp_Timeout, TB_Set_DebounceDelay_500µs`

Synopsis	void TB_Set_Delay_500us(u16)
	void TB_Set_DisplayDelay_500us(u16)
	void TB_Set_StartUp_Timeout(u16)
	void TB_Set_DebounceDelay_500us
Description	These functions are used to respectively update the values of the hTimebase_500us, hTimebase_display_500us, hStart_Up_TimeBase_500us and hKey_debounce_500us variables. They are used to maintain the main state machine in Fault state, to set the refresh rate of the LCD, the Startup timeout and, to filter the user key bouncing.
Input	Unsigned 16 bits
Returns	None
Functions called	None

`TB_StartUp_Timeout_IsElapsed, TB_Delay_IsElapsed, TB_DisplayDelay_IsElapsed, TB_DebounceDelay_IsElapsed`

Synopsis	bool TB_StartUp_Timeout_IsElapsed(void)
	bool TB_Delay_IsElapsed(void)
	bool TB_DisplayDelay_IsElapsed(void)
	bool TB_DebounceDelay_IsElapsed(void)
Description	These functions return TRUE if the related delay is elapsed, FALSE otherwise.
Input	None
Returns	Boolean
Functions called	None

`SysTickHandler`

Synopsis	void SysTickHandler(void)
Description	This is the System Tick timer interrupt routine. It is executed every 500µs, as determined by TB_Init and is used to refresh various variables used mainly as counters (for example, PID sampling time). Moreover, this routine implements the startup torque ramp described in *Section 3: Running the demo program on page 24*.
Input	None
Returns	None
Functions called	IFOC_CalcFluxTorqueRef, TB_ClearFlag,
	If Encoder is used:
	ENC_Get_Average_Speed
	If Tacho is used:
	TAC_GetRotorFreqInHz
Note	This is an interrupt routine

4.11 Routines for monitoring and handling the system critical parameters: `MC_MotorControl_Layer` module

4.11.1 Overview

This module brings together additional functions that take into account various tasks related to preliminary motor operations like software initializations, ADC calibrations, parameter monitoring, and critical fault checks and managements.

4.11.2 List of available functions

- *`MCL_Init` on page 89*
- *`MCL_Init_Arrays` on page 89*
- *`MCL_Chk_OverTemp` on page 89*
- *`MCL_Chk_BusVolt` on page 90*
- *`MCL_SetFault` on page 90*
- *`MCL_ChkPowerStage` on page 90*
- *`MCL_ClearFault` on page 90*
- *`MCL_Compute_BusVolt` on page 91*
- *`MCL_Compute_Temp` on page 91*

MCL_Init

Synopsis	void MCL_Init(void)
Description	This function implements the motor control initializations to be performed at each motor startup; it affects the initialization of PID regulators, ADC calibration, speed sensors and high side driver boot capacitors.
Functions called	PID_Reset_integral_terms, ENC_Clear_Speed_Buffer (or, alternatively, TAC_InitTachoMeasure), IFOC_Init, TB_Set_StartUP_Timeout, TIM1_CtrlPWMOutputs, TB_StartUp_Timeout_IsElapsed, SVPWM_3ShuntCurrentReadingCalibration (or, alternatively, SVPWM_IcsCurrentReadingCalibration)

Note: When in Speed control mode, the Torque profile in this function is initialized for motor startup settings.

See also *Section 3.2: Speed control mode on page 28*.

MCL_Init_Arrays

Synopsis	void MCL_Init_Arrays(void)
Description	This function initializes the arrays used for temperature and bus voltage measurements to avoid erroneous fault detection after an MCU reset.

MCL_Chk_OverTemp

Synopsis	bool MCL_Chk_OverTemp(void)
Description	This function checks for overtemperature fault occurrences on the heatsink connected to the power stage switches. The thresholds for temperature and hysteresis values are defined by the user.
Returns	Returns TRUE if the voltage on the thermal resistor connected to ADC channel ADCIN_10 has reached the threshold level (or if it has not yet returned to the threshold level minus the hysteresis value after an overheat detection).
Functions called	ADC_GetInjectedConversionValue.
See also	*Section 3.6: Fault messages on page 35 on how to set temperature thresholds.*

`MCL_Chk_BusVolt`

Synopsis	BusV_t MCL_Chk_BusVolt(void)
Description	This function checks for over and/or undervoltage faults on inverter DC bus. The thresholds for under and overvoltages are defined by the user.
Returns	It returns a BusV_t type variable reporting the fault value.
Functions called	ADC_GetInjectedConversionValue
See also	*Section 3.6: Fault messages on page 35* on how to set DC bus voltage thresholds.

`MCL_SetFault`

Synopsis	void MCL_SetFault (u16)
Description	On an event fault occurrence, this function puts the state machine in Fault state and disables the motor control outputs of Advanced Control Timer TIM1 (PWM timer).
Input	Sets the motor control status flag variable according to the cause of the fault.
Functions called	TB_Set_Delay_500us, TIM1_CtrlPWMOutputs

`MCL_ChkPowerStage`

Synopsis	void MCL_ChkPowerStage(void)
Description	This function checks the power stage operation under critical conditions of temperature and bus voltage.
Functions called	MCL_SetFault, MCL_Chk_OverTemp, MCL_Chk_BusVolt

`MCL_ClearFault`

Synopsis	bool MCL_ClearFault(void)
Description	This function checks if the fault source is over. If it is the case, it clears the related flag and returns TRUE. Otherwise it returns FALSE.
Returns	TRUE or FALSE according to fault source status.
Functions called	MCL_Chk_BusVolt, MCL_Chk_OverTemp, GPIO_ReadInputDataBit, TAC_ClrTimeout (if Tacho speed sensor is used).

Note: This function checks for all the sources of fault handled by the Motor Control software, that is is overcurrent (break event), Startup failure, speed feedback timeout, power stage overtemperature and over/undervoltage on the DC bus.

`MCL_Compute_BusVolt`

Synopsis	u16 MCL_Compute_BusVolt(void)
Description	This function computes the DC bus voltage in volt units.
Returns	A value representing the bus voltage in volt units

`MCL_Compute_Temp`

Synopsis	u8 MCL_Compute_Temp(void)
Description	This function computes the power stage heatsink temperature in Celsius degrees.
Returns	An integer representing a temperature value expressed in Celsius degrees.

4.12 Application layer

Besides the implementation of the motor control state machine, the application layer (`main.c` module) includes the configuration and initialization of basic peripherals. Moreover, in this layer the LCD display and user input keys are managed by means function calls belonging to two other modules. These modules are briefly described below.

- `MC_Keys.c` module

  Centralizes all information regarding the keyboard reading. Any action on the keyboard is processed in the `Keys_process` routine.
- `MC_Display` module

  Centralizes all information regarding the LCD display management.
- *stm32f10x_lcd* module

  Contains some dedicated routines for the control of the LCD embedded with the starter kit.

Appendix A Additional information

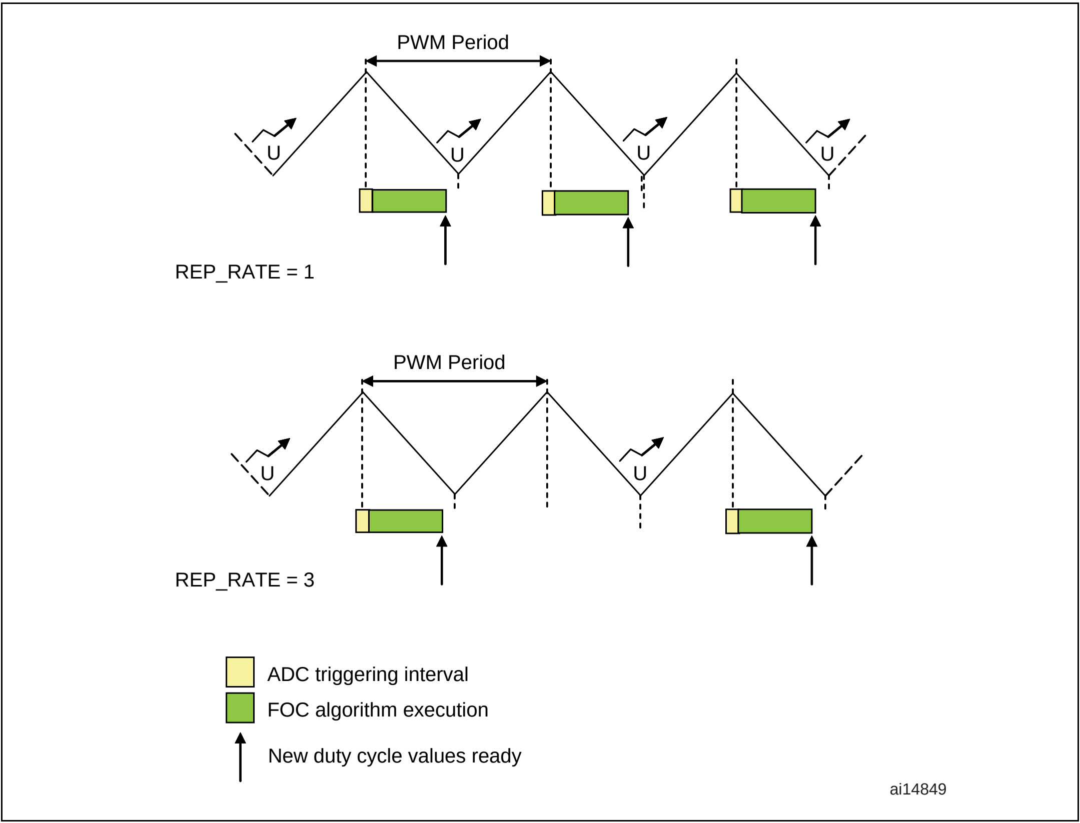
A.1 Adjusting CPU load related to IFOC algorithm execution

The advanced control timer (TIM1) peripheral has the built-in capability of updating PWM registers only after a given number of PWM semi-periods. This feature is handled by a programmable repetition counter. It is particularly useful to adjust the CPU load related to IFOC algorithm execution for a given PWM frequency (refer to STM32F103xx reference manual for more information on programmable repetition counter).

When using ICS, the injected chain of conversions for current reading is directly triggered by a PWM register update event. Moreover, since the IFOC algorithm is executed at the end of the injected chain of conversions in the related ISR, changing repetition counter has a direct impact on IFOC refresh rate and thus on CPU load.

However, in the case of three shunt topology current reading, to ensure that the IFOC algorithm is executed once for each PWM register update, it is necessary to keep the synchronization between current conversion triggering and PWM register update. In the proposed software library, this is automatically performed, so that you can reduce the frequency of execution of the IFOC algorithm by simply changing the default value of the repetition counter (the `REP_RATE` parameter in the `MC_Control_Param.h` header file). *Figure 50* shows current sampling triggering, and IFOC algorithm execution with respect to PWM period when `REP_RATE` is set to 3.

Figure 50. AD conversions for three shunt topology stator currents reading and IFOC algorithm execution when `REP_RATE=3`



Note: Because three shunt resistor topology requires low side switches to be on when performing current reading A/D conversions, the REP_RATE parameter must be an ***odd*** *number in this case.*

Considering that the raw IFOC algorithm execution time is about 21 µs when in three shunt resistor stator current reading configuration, the related contribution to CPU load can be computed as follows:

$$\text{CPU Load}_{\%} = \frac{F_{PWM}}{\text{Refresh\_Rate}} \cdot 21 \cdot 10^{-6} \cdot 100 = \frac{F_{PWM}}{(\text{REP\_RATE} + 1)/2} \cdot 21 \cdot 10^{-6} \cdot 100$$

A.2 Selecting the update repetition rate based on PWM frequency for 3 shunt resistor configuration

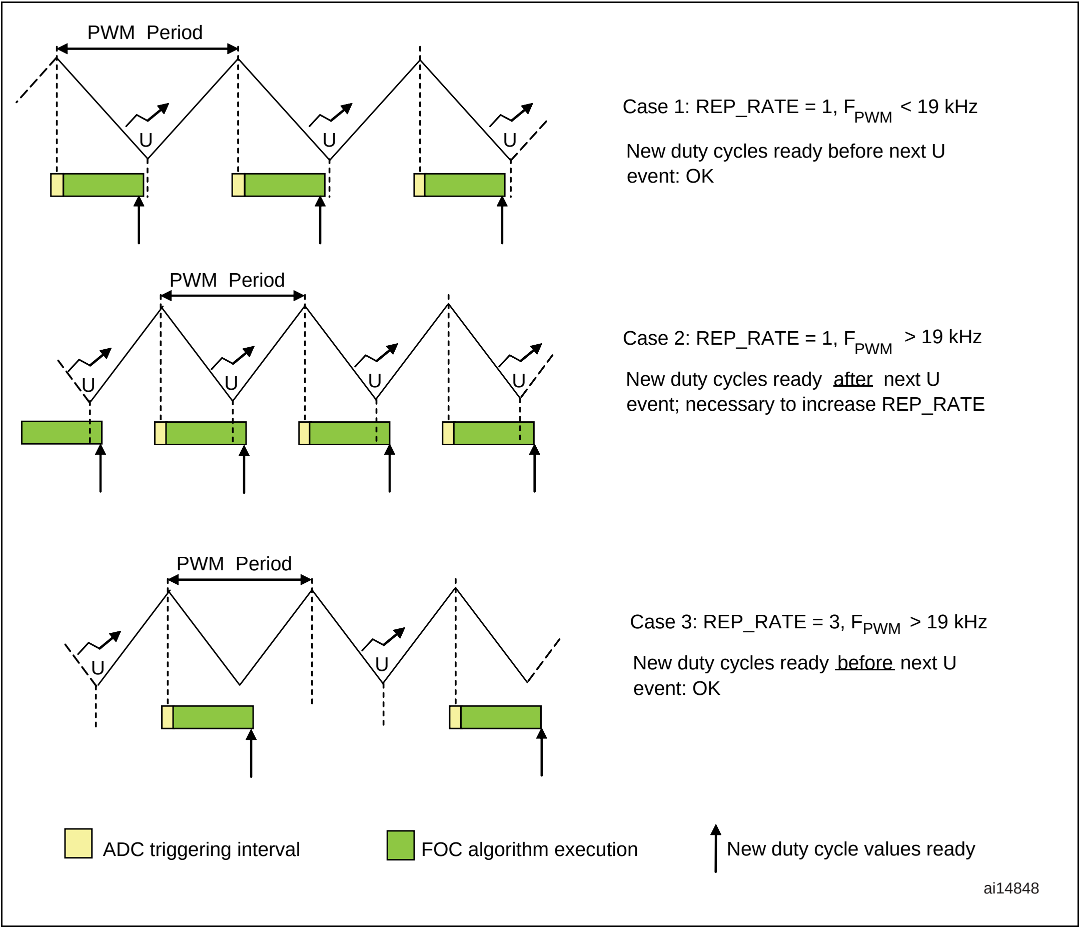
Beyond the well known trade-off between acoustical noise and power dissipation, consideration should be given to selecting the PWM switching frequency using the AC IM IFOC software library.

As discussed in *Section 4.2.5 on page 45*, depending on the PWM switching frequency, a limitation on the maximum applicable duty cycle could occur if using three shunt resistor configuration for current reading. *Table 2: PWM frequency vs. maximum duty cycle relationship on page 49*, summarizes the performance of the system when the software library is used in conjunction with STM3210B-MCKIT hardware.

Note: The MB459 board is an evaluation platform; it is designed to support different motor driving topologies (PMSM and AC induction) and current reading strategies (single and three shunt resistors). Therefore, the figures given in Table 2 on page 49 should be understood as a starting point and not as a best case.

Moreover, in order to guarantee the proper working of the algorithm and be sure that the new computed duty cycles will be applied in the next PWM period, it is always necessary to finish executing the IFOC algorithm before the next PWM period begins as shown in *Figure 51*.

Figure 51. AD conversion intervals for three shunt topology stator currents reading and IFOC algorithm execution


PWM Period
U
Case 1: REP_RATE = 1, F_{PWM} < 19 kHz
New duty cycles ready before next U event: OK
Case 2: REP_RATE = 1, F_{PWM} > 19 kHz
New duty cycles ready after next U event; necessary to increase REP_RATE
Case 3: REP_RATE = 3, F_{PWM} > 19 kHz
New duty cycles ready before next U event: OK
ADC triggering interval
FOC algorithm execution
New duty cycle values ready
ai14848

Considered that as seen in *Section 4.2.5*, ADC conversions are triggered latest $(T_N-T_S)/2$ after the TIM1 counter overflow and, considered the time required for the A/D converter to perform injected conversions, it can been stated that the IFOC algorithm is started about 5 µs after the TIM1 counter overflow (worst case). Furthermore, given that the raw execution time of the IFOC algorithm is around 21 µs, to compute the new duty cycle values before the next update event, it is necessary to guarantee a minimum duty cycle period of about [(5 + 21) × 2] µs. That means a maximum achievable IFOC execution rate of about 19 kHz. This allows to set the repetition counter (REP_RATE) equal to 1.

Moreover, the above stated execution rate is beyond the maximum PWM frequency supported by the firmware. For a PWM frequency above 19 kHz, a repetition counter of 3 must be used (REP_RATE = 3).

A.3 Fixed-point numerical representation

The AC IM IFOC software library uses fixed-point representation of fractional signed values. Thus, a number *n* is expressed as $n = m \cdot f$, where *m* is the integer part (magnitude) and *f* the fractional part, and both *m* and *f* have fixed numbers of digits.

In terms of two's complement binary representation, if a variable *n* requires QI bits to express - as powers of two - its magnitude (of which 1 bit is needed for the sign), QF bits – as inverse powers of two - for its fractional part, then we have to allocate QI + QF bits for that variable.

Therefore, given a choice of QI and QF, the variable representation has the following features:

- Range: $-2^{(QI-1)} < n < 2^{(QI-1)} - 2^{(-QF)}$;
- Resolution: $= 1 / 2^{QF}$.

The equation below converts a fractional quantity *q* to fixed-point representation *n*:
$n = \text{floor}(q \cdot 2^{QF})$

A common way to express the choice that has been made is the "q QI.QF" notation.

So, if a variable is stored in q3.5 format, it means that 3 bits are reserved for the magnitude, 5 bits for the resolution; the expressible range is from -4 to 3.96875, the resolution is 0.03125, the bit weighting is:

bit n.	7	6	5	4	3	2	1	0
value	-4	2	1	1/2	1/4	1/8	1/16	1/32

This software library uses the PU ("Per Unit") system to express current values. They are always referred to a base quantity that is the maximum measurable current I_{max} (which, for the proposed hardware, can be estimated approximately at $I_{max} = 0.6 / R_{shunt}$); so, the "per unit" current value is obtained by dividing the physical value by that base:

$$i_{PU} = \frac{i_{S.I.}}{I_{\max}}$$

In this way, i_{pu} is always in the range from -1 to +1. Therefore, the q1.15 format, which ranges from -1 to 0.999969482421875, with a resolution of 0.000030517578125, is perfectly suitable (taking care of the overflow value (-1)·(-1)=1) and thus extensively used.

Thus, the complete transformation equation from SI units is:

$$i_{q1.15} = \text{floor}\left(\frac{i_{S.I.}}{I_{max}} \cdot 2^{QF}\right)$$

A.4 Tacho-based speed measurement flow charts

This section summarizes the main tasks achieved in the tacho capture interrupt in the form of flow charts. The purpose of these flow charts is to help understand how the automatic prescaler adjustment is done.

Figure 52. Tacho capture overview

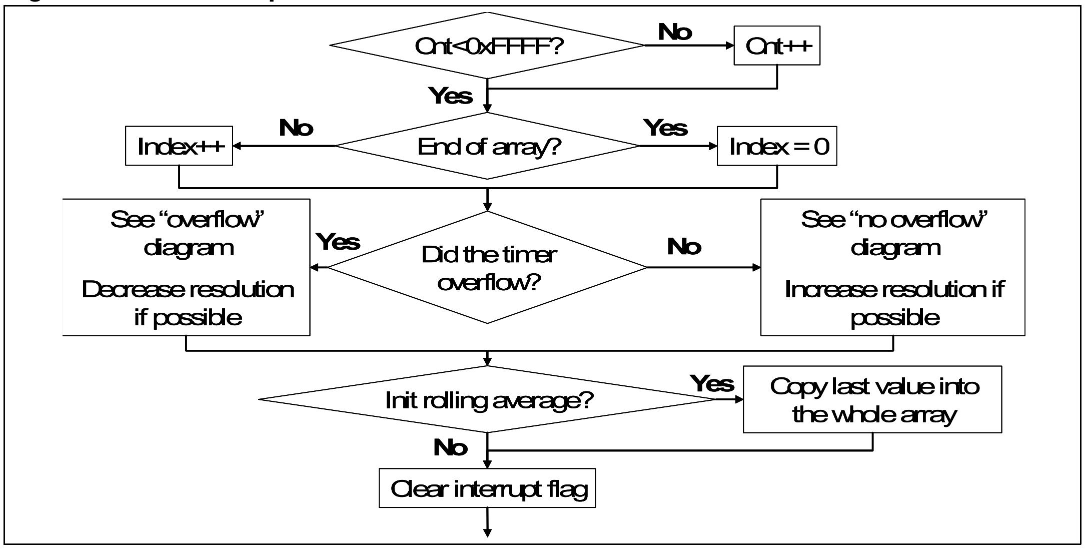


Figure 53. Processing captured value when timer did not overflow

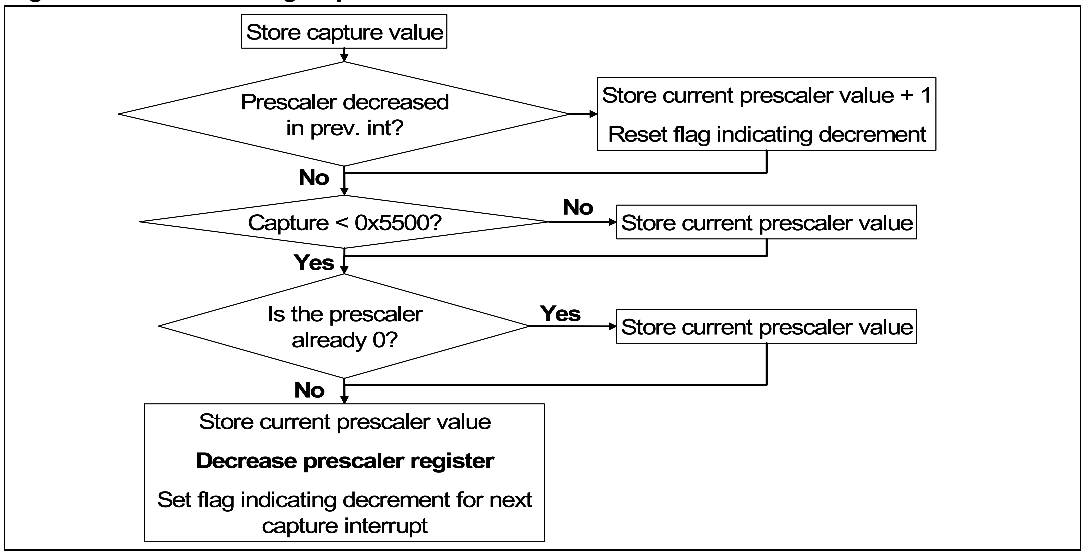
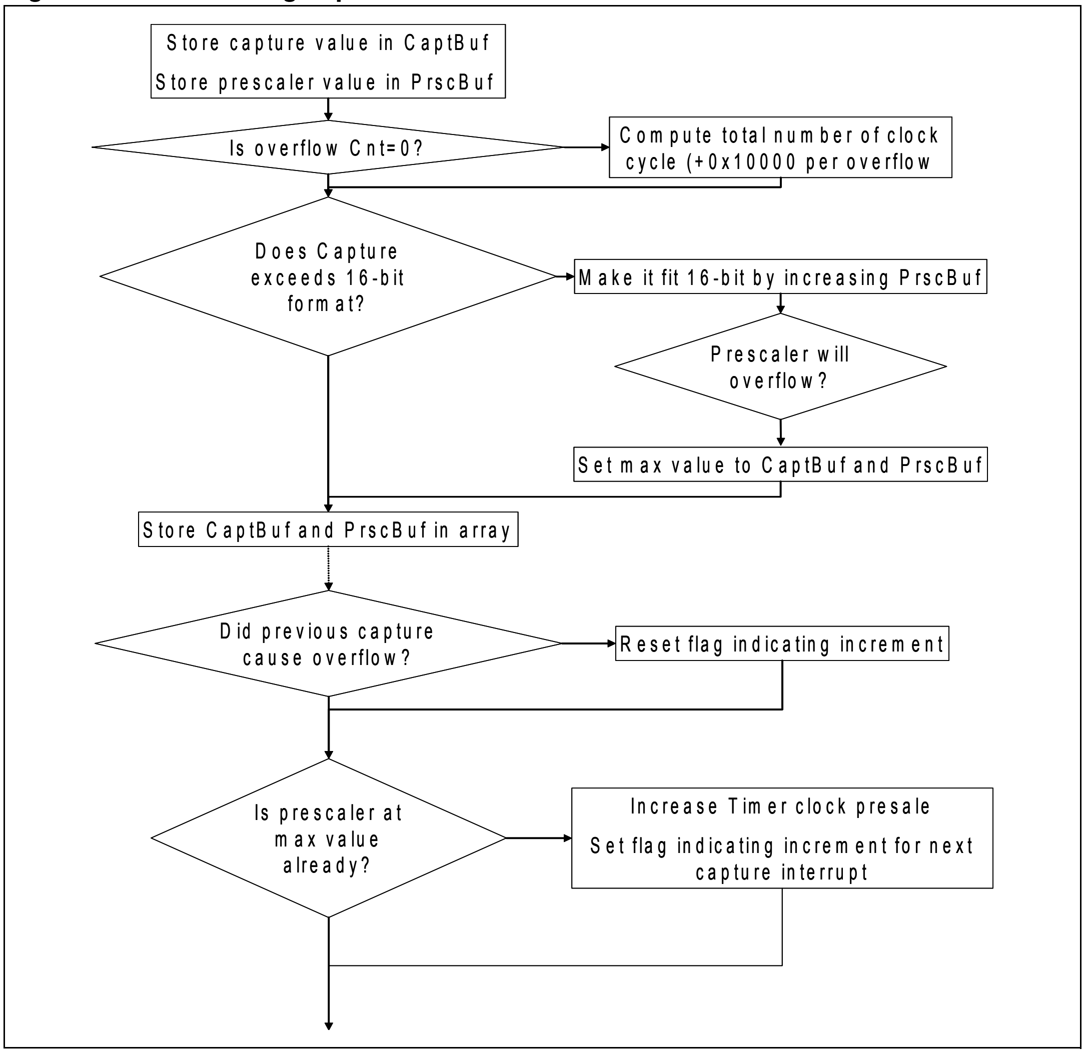

Store capture value
Prescaler decreased in prev. int?
Store current prescaler value + 1
Reset flag indicating decrement
No
Capture < 0x5500?
No
Store current prescaler value
Yes
Is the prescaler already 0?
Yes
Store current prescaler value
No
Store current prescaler value
Decrease prescaler register
Set flag indicating decrement for next capture interrupt

Figure 54. Processing captured value when timer did overflow


Store capture value in CaptBuf
Store prescaler value in PrscBuf
Is overflow Cnt=0?
Compute total number of clock cycle (+0x10000 per overflow
Does Capture exceeds 16-bit format?
Make it fit 16-bit by increasing PrscBuf
Prescaler will overflow?
Set max value to CaptBuf and PrscBuf
Store CaptBuf and PrscBuf in array
Did previous capture cause overflow?
Reset flag indicating increment
Is prescaler at max value already?
Increase Timer clock presale
Set flag indicating increment for next capture interrupt

A.5 PID block diagrams

The following flow diagrams (*Figure 55* and *Figure 56*) show the decision tree for the computation of the torque/flux and speed regulation routines.

Figure 55. Torque/flux control loop block diagram

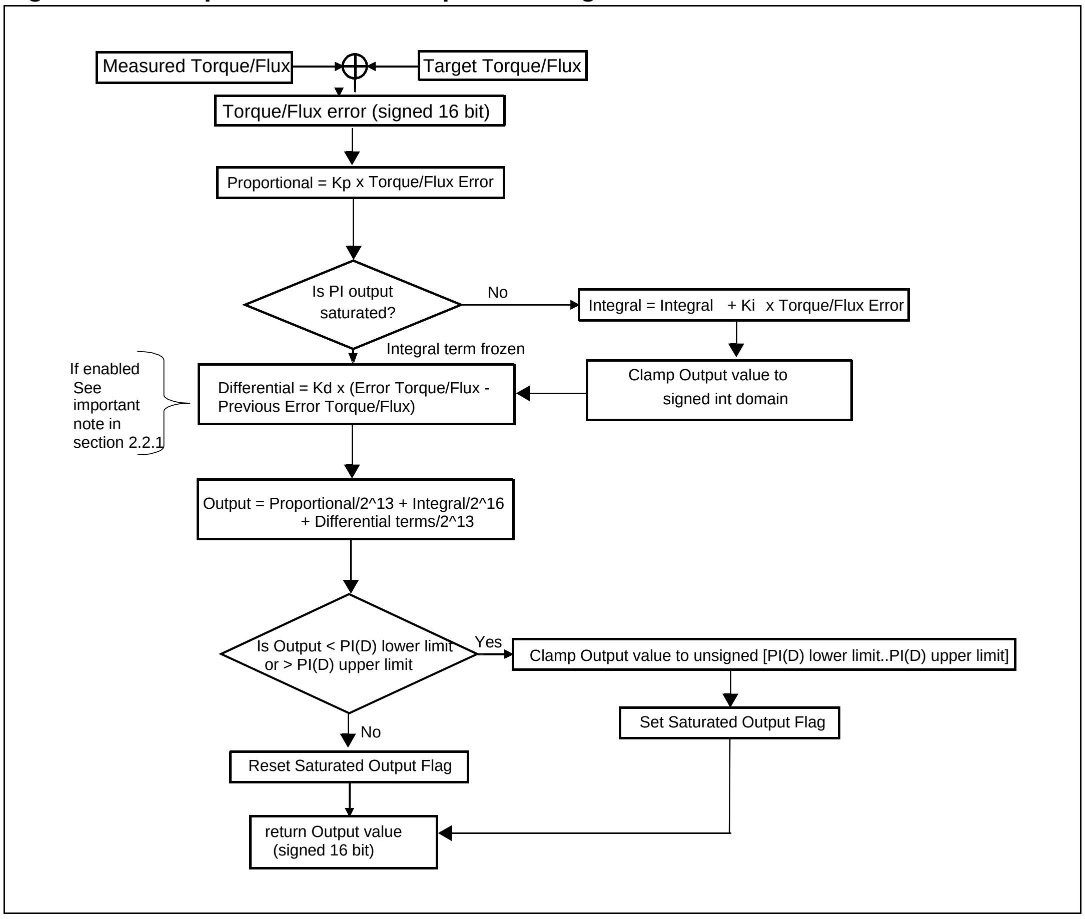
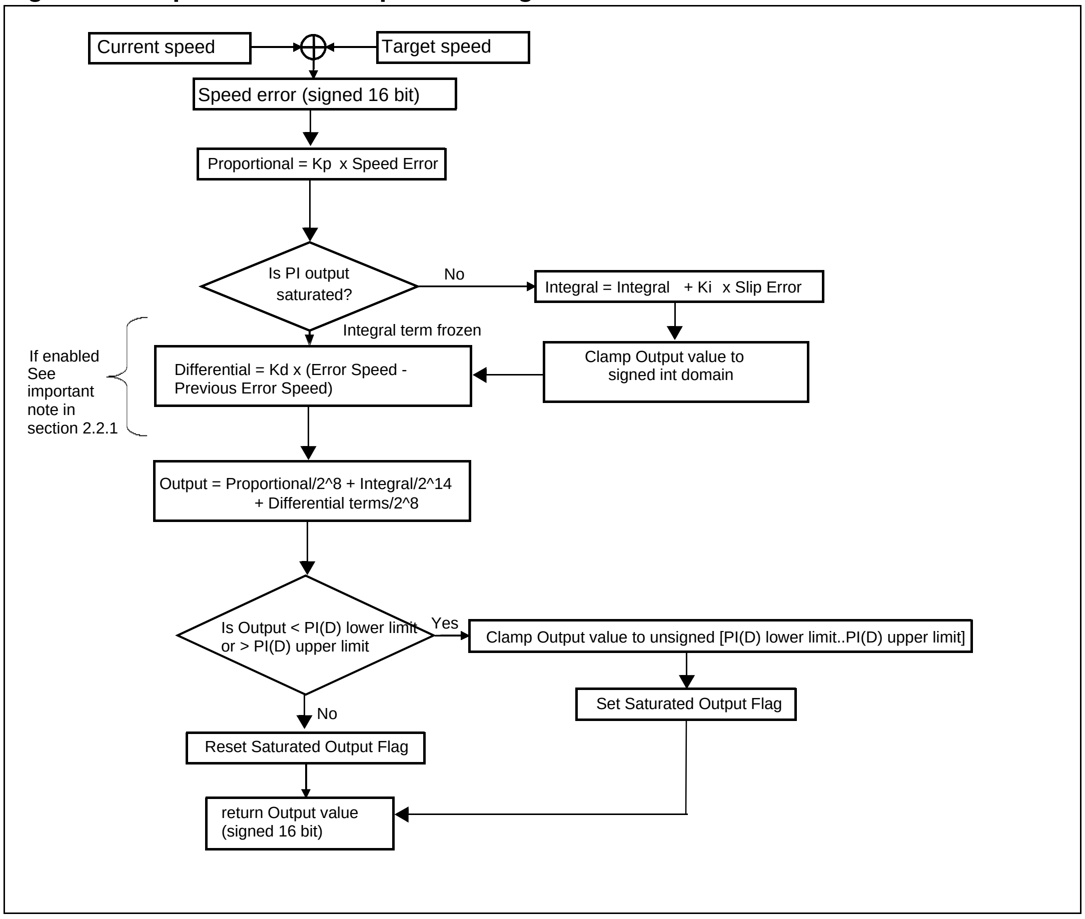

Measured Torque/Flux
Target Torque/Flux
Torque/Flux error (signed 16 bit)
Proportional = Kp x Torque/Flux Error
Is PI output saturated?
No
Integral = Integral + Ki x Torque/Flux Error
Integral term frozen
Clamp Output value to signed int domain
If enabled See important note in section 2.2.1
Differential = Kd x (Error Torque/Flux - Previous Error Torque/Flux)
Output = Proportional/2^13 + Integral/2^16 + Differential terms/2^13
Is Output < PI(D) lower limit or > PI(D) upper limit
Yes
Clamp Output value to unsigned [PI(D) lower limit..PI(D) upper limit]
Set Saturated Output Flag
No
Reset Saturated Output Flag
return Output value (signed 16 bit)

Figure 56. Speed control loop block diagram


Current speed
Target speed
Speed error (signed 16 bit)
Proportional = Kp x Speed Error
Is PI output saturated?
No
Integral = Integral + Ki x Slip Error
Integral term frozen
Clamp Output value to signed int domain
If enabled See important note in section 2.2.1
Differential = Kd x (Error Speed - Previous Error Speed)
Output = Proportional/2^8 + Integral/2^14 + Differential terms/2^8
Is Output < PI(D) lower limit or > PI(D) upper limit
Yes
Clamp Output value to unsigned [PI(D) lower limit..PI(D) upper limit]
Set Saturated Output Flag
No
Reset Saturated Output Flag
return Output value (signed 16 bit)

A.6 A priori determination of flux and torque currents PI gains

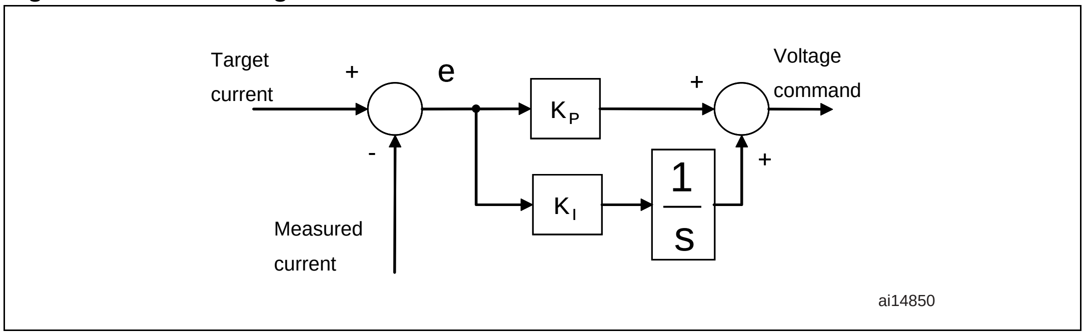
The purpose of this appendix is to provide a criterion for the computation of the initial values of torque/flux PI parameters (K_I and K_P). Successive fine tuning is then performed in the practical system.

To calculate these starting values it is required to know the electrical characteristics of the motor in terms of stator resistance R_S and inductance L_S plus the electrical characteristics of the hardware board, that is, shunt resistor R_{Shunt}, current sense amplification gain A_{op} and direct current bus voltage $V_{Bus}DC$.

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

Figure 57 shows the PI controller block diagram used for torque or flux regulation.

Figure 57. Block diagram of PI Controller

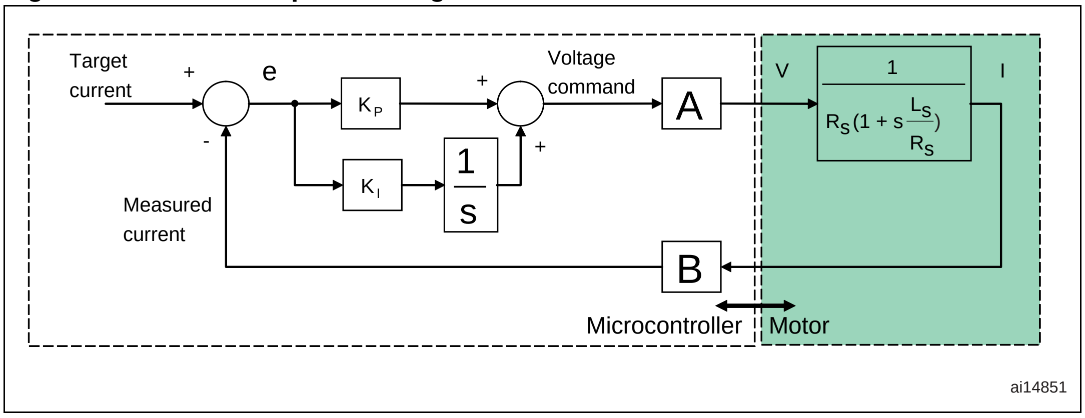


For this analysis, it is assumed that the driving strategy is isotropic with respect to the q and d axis, so it can be assumed that the starting values of K_P are the same for the torque regulator and the flux regulator. The same assumption is done for K_I coefficients.

Figure 58 shows the closed loop system in which the motor phase is modeled using the “resistance-inductance” equivalent circuit in locked-rotor condition.

Block “A” converts the “Voltage command” software digital value into the applied stator voltage V and block “B” converts the real motor current into the “Measured current” software digital value.

Figure 58. Closed loop block diagram

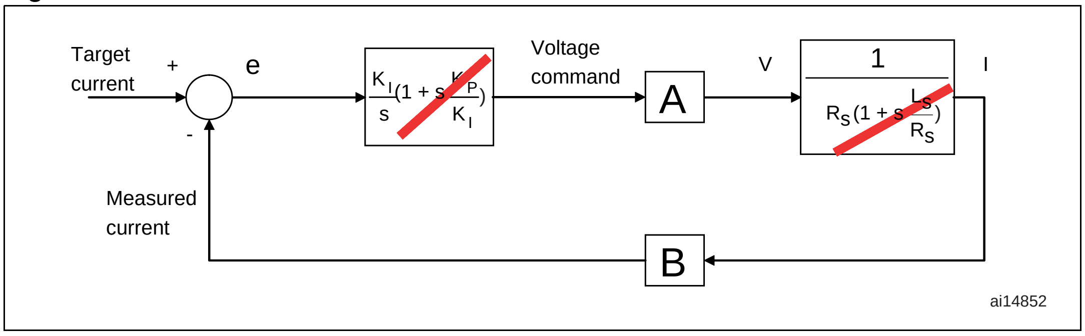


The transfer functions of the two blocks “A” and “B” are expressed by the following formulas:

$A = \frac{V_{Bus}DC}{2^{16}}$ and $B = \frac{R_{shunt}A_{op}2^{16}}{3.3}$, respectively.

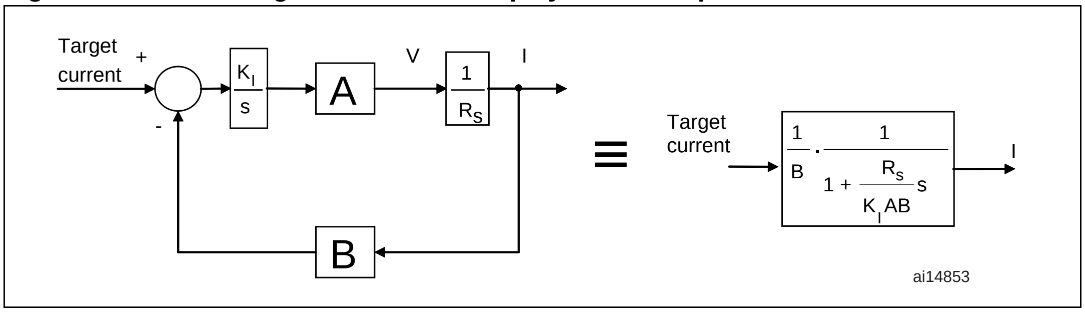
By putting $K_P/K_I = L_S/R_S$, it is possible to perform pole-zero cancellation as described in *Figure 59*. Figure A6.3.

Figure 59. Pole-zero cancellation


Target current
+
-
e
$\frac{K_I(1+s\frac{K_P}{K_I})}{s}$
Voltage command
A
V
$\frac{1}{R_S(1+s\frac{L_S}{R_S})}$
I
Measured current
B
ai14852

In this condition, the closed loop system is brought back to a first-order system and the dynamics of the system can be assigned using a proper value of K_I. See *Figure 60*.

Figure 60. Block diagram of closed loop system after pole-zero cancellation



It is important to note that the K_I and K_P parameters used inside the PI algorithms will be scaled by a constant factor equal to 256. So the computed values of K_P and K_I must be multiplied by this factor when used in the MC_Control_Param.h file.

Moreover, in the computation of the integral part, the PI algorithm does not include the PI sampling time (T). See the following formula:

$$k_i\int_0^t e(\tau)d\tau = k_iT\sum_{k=1}^{n} e(kT) = K_i\sum_{k=1}^{n} e(kT)$$

Since the integral part of the controller is computed as the sum of successive errors, it is required to include T in computation of the K_I.

So the final formula can be expressed as: $K_P = L_S\frac{\omega_C}{AB}256$

$$K_I = \frac{R_S\cdot\ \omega_C\cdot\ 256}{AB}\cdot\ T_1$$

$$AB = \frac{V_{Bus}DC\cdot\ R_{shunt}\cdot\ A_{op}}{3.3}$$

For example it is possible to set ω_C (the bandwidth of the closed loop system) equal to 1500 rad/s, so that the time constant of the current control system is 0.66 ms.

The A_{op} measured for the MB459 is 2.57. It is then possible to compute the values of the parameters knowing the motor parameters (R_S, L_S), V_{BUS}DC and R_{Shunt}.

A.7 References

- [1] P. C. Krause, O. Wasynczuk, S. D. Sudhoff, Analysis of Electric Machinery and Drive Systems, Wiley-IEEE Press, 2002.
- [2] T. A. Lipo and D. W. Novotny, Vector Control and Dynamics of AC Drives, Oxford University Press, 1996.

Revision history

Table 4. Document revision history

Date	Revision	Changes
08-Jan-2008	1	Initial release.