AN5464

Application note

# Position control of a three-phase permanent magnet motor using X-CUBE-MCSDK or X-CUBE-MCSDK-FUL

## Introduction

This document describes how to set up and generate a motor control firmware project enabling the position control feature to drive a three-phase PMSM motor using the X-CUBE-MCSDK STM32Cube Expansion Package. It also explains how to tune the system and develop a demonstration program using the available API (application programming interface).

The descriptions for X-CUBE-MCSDK in this document apply also entirely to the X-CUBE-MCSDK-FUL STM32Cube Expansion Package.

# 1 Overview

The position control mode is available since the version v5.4.3 of the X-CUBE-MCSDK Expansion Package. It allows the electric drive to move the motor to a specified target mechanical position (angle) in a settled time (duration) following a programmed trajectory composed of three phases: acceleration, constant speed, and deceleration. Following such defined trajectory is mandatory to reach the target position with zero angular speed and therefore avoid oscillations around the settled position.

## 1.1 Main features

- Position control mode available for all the control boards supported by X-CUBE-MCSDK
- Compatible with:
  - Quadrature encoder (with index signal)
  - Magnetic sensor (with encoder interface)
- Trajectory calculator
  - Constant jerk
  - Acceleration integration
  - Speed integration
  - Position integration
- Current control for field orientation (FOC)

## 1.2 Hardware prerequisites

- A control board supported by X-CUBE-MCSDK, featuring an STM32 microcontroller based on the Arm® Cortex®-M core
- A power board or expansion board supported by X-CUBE-MCSDK (three-shunt current sensing is advised)
- Optionally, a complete inverter supported by X-CUBE-MCSDK
- The main power supply (according to the host board used)
- A USB Type-A to Mini-B cable for programming and debugging (see the exact requirements of the board used)
- A PMSM motor with quadrature encoder (with index signal)

*Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*
*All other trademarks are the property of their respective owners.*

arm

## 1.3 Software prerequisites

- STSW-LINK009: ST-LINK USB driver
- STSW-LINK004: STM32 ST-LINK Utility
  - STSW-LINK004 is needed only to update ST-LINK firmware to the latest version
- A Windows® PC with the supported development toolchain:
  - IAR™ – EWARM
  - Keil® - MDK-ARM
  - STMicroelectronics - STM32CubeIDE
- X-CUBE-MCSDK version v5.4.3 or later

# 2 Algorithm description

The method implemented makes use of three controllers as shown in Figure 1:

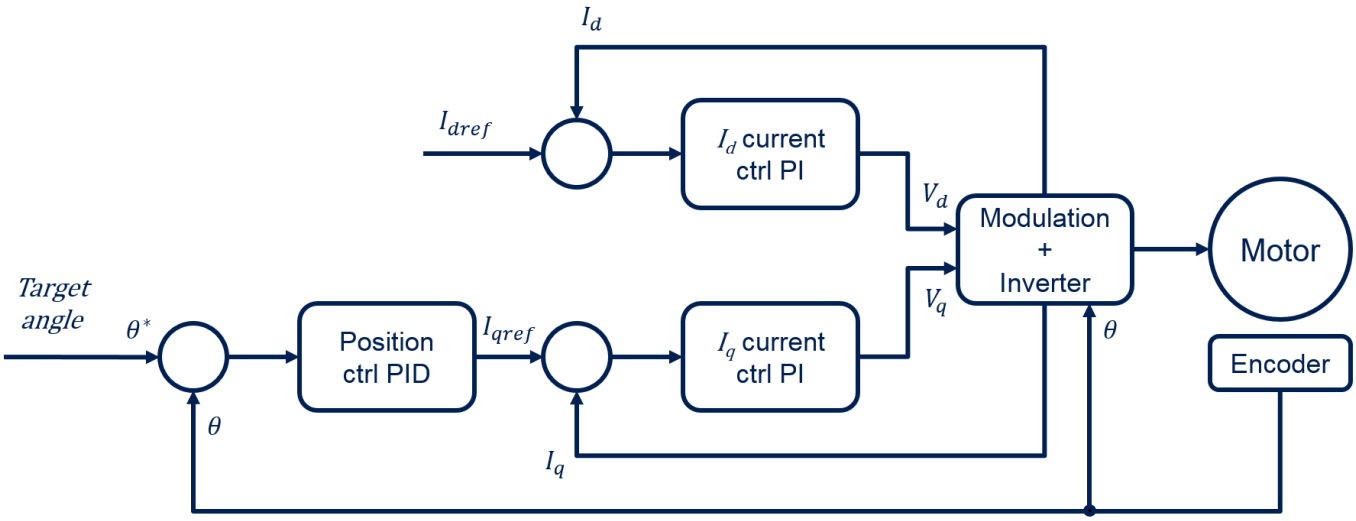
- A position controller implemented using a PID (with a proportional, integral and derivative action) executed at 1 kHz
- Two current controllers implemented using a PI (with only proportional and integral action) executed at 30 kHz for $I_d$ and $I_q$ current regulations

Both frequencies can be customized by means of the MC Workbench.

As shown in Figure 1, the position controller PID influences only the $I_q$ reference ($I_{qref}$). Instead, the $I_d$ reference ($I_{dref}$) is managed, as usual, according to the MTPA trajectory.

This scheme is the best approach using an encoder, for the motor position measurement, since it does not require a precise measurement of the motor speed.

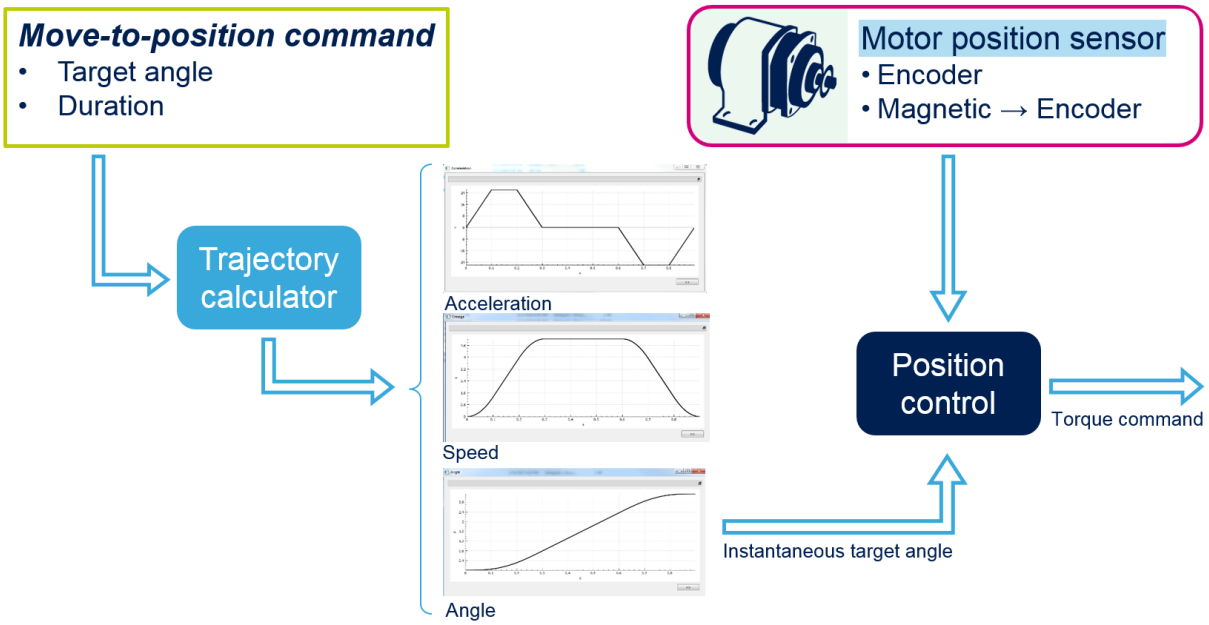
**Figure 1. Block diagram**




To achieve a smooth movement, a trajectory (considered as a sequence of target positions) is calculated starting from the movement command.

The trajectory is computed applying a constant jerk approach as illustrated in Figure 2.

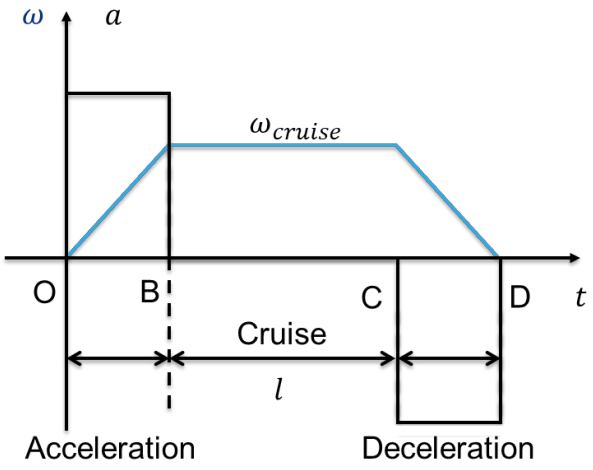
**Figure 2. Algorithm overview**



# 3 Trajectory calculator

In the trajectory control mode, a trapezoidal target angular-velocity trajectory is implemented with three time intervals: a constant angular acceleration (from O to B), a constant angular velocity (from B to C), and a constant angular deceleration (from C to D) as shown in blue in Figure 3, where $l$ is the duration of the cruise angular-velocity interval.

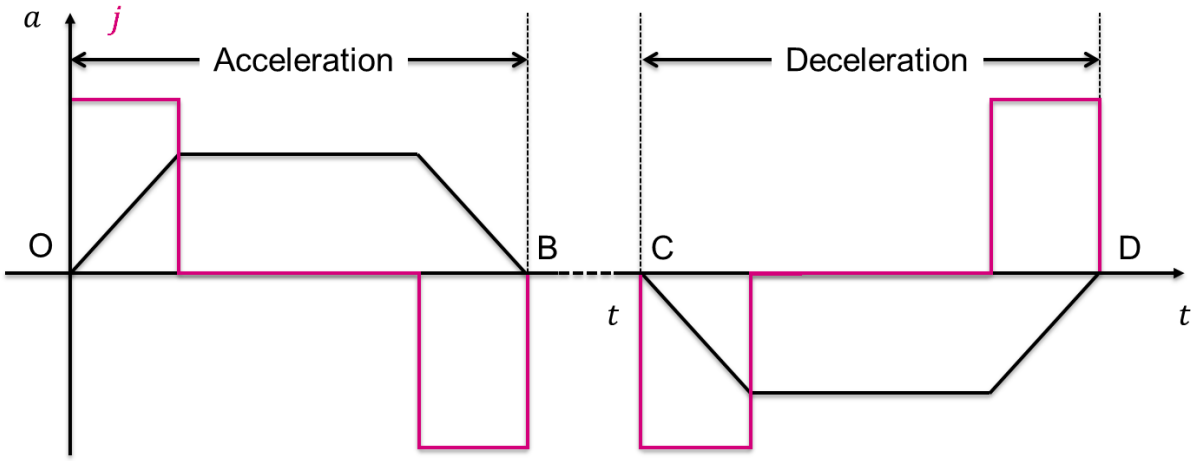
**Figure 3. Trapezoidal target speed trajectory**




To avoid step changes during the angular acceleration interval, which is physically impossible, and to avoid mechanical stress, a constant angular jerk is implemented during the acceleration and deceleration intervals as shown in pink in Figure 4. The angular jerk, also known as jolt, is the rate of change of the angular acceleration with respect to time.

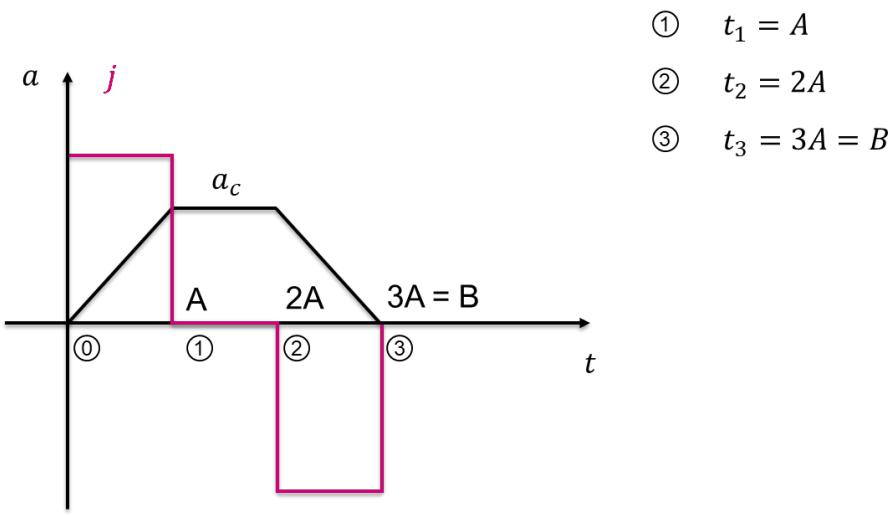
**Figure 4. Trajectory with constant jerk**

$$j(t) = \frac{da(t)}{dt}$$



The acceleration and deceleration intervals are further split into three subintervals with the same durations as shown in Figure 5.

**Figure 5. Acceleration subintervals**




For the first subinterval from ⓪ to ① in Figure 5 with angular jerk $j(t) = constant = J$ , the motion is defined by the equations shown in Table 1.

**Table 1. Equations in the first subinterval of the acceleration phase**

| Initial conditions | Law of motion | Values at subinterval end |
|---|---|---|
| $\begin{cases} a_0 = 0 \\ \omega_0 = 0 \\ \theta_0 = 0 \end{cases}$ | $\begin{cases} a = Jt + a_0 = Jt \\ \omega = \frac{Jt^2}{2} + a_0 t + \omega_0 = \frac{Jt^2}{2} \\ \theta = \frac{Jt^3}{6} + \frac{a_0 t^2}{2} + \omega_0 t + \theta_0 = \frac{Jt^3}{6} \end{cases}$ | $\begin{cases} a_1 = JA = a_c \\ \omega_1 = \frac{JA^2}{2} \\ \theta_1 = \frac{JA^3}{6} \end{cases}$ |

For the second subinterval from ① to ② in Figure 5 with angular jerk $j(t) = 0$ and acceleration $a = constant$, the motion is defined by the equations shown in Table 2.

**Table 2. Equations in the second subinterval of the acceleration phase**

| Initial conditions | Law of motion | Values at subinterval end |
|---|---|---|
| $\begin{cases} a_1 = JA = a_c \\ \omega_1 = \frac{JA^2}{2} \\ \theta_1 = \frac{JA^3}{6} \end{cases}$ | $\begin{cases} a = a_c \\ \omega = a_c(t - A) + \omega_1 \\ \theta = \frac{a_c}{2}(t - A)^2 + \omega_1\left(t - A\right) + \theta_1 \end{cases}$ | $\begin{cases} a_2 = a_c = JA \\ \omega_2 = \frac{3JA^2}{2} \\ \theta_2 = \frac{7JA^3}{6} \end{cases}$ |

For the third subinterval from ② to ③ in Figure 5 with angular jerk $j(t) = constant = -J$, the motion is defined by the equations shown in Table 3.

**Table 3. Equations in the third subinterval of the acceleration phase**

| Initial conditions | Law of motion | Values at subinterval end |
|---|---|---|
| $\begin{cases} a_2 = a_c = JA \\ \omega_2 = \frac{3JA^2}{2} \\ \theta_2 = \frac{7JA^3}{6} \end{cases}$ | $\begin{cases} a = -J(t-2A) + a_c \\ \omega = -\frac{J}{2}(t-2A)^2 + a_c(t-2A) + \omega_2 \\ \theta = -\frac{J}{6}(t-2A)^3 + \frac{a_c}{2}(t-2A)^2 + \omega_2\left(t-2A\right) + \theta_2 \end{cases}$ | $\begin{cases} a_3 = 0 \\ \omega_3 = 2JA^2 \\ \theta_3 = 3JA^3 \end{cases}$ |

Considering that the deceleration step is symmetrical to the acceleration step, we can assume that the $\Delta\theta$ in acceleration and deceleration steps is equal to $\theta_3$ given in Table 3.

The angular variation $\Delta\theta$ in the interval where the angular velocity is constant (cruise interval in Figure 3) is equal to $l \cdot \omega_{cruise}$ where $l \cdot \omega_{cruise} = \omega_3$ given in Table 3. The total angular variation from the beginning to the end of the movement is expressed in (1).

$$\Delta\theta_{tot} = 2\theta_3 + l\omega_3 = 6JA^3 + 2JA^2 l \tag{1}$$

The total duration of the movement is the sum of each interval. It is expressed in (2).

$$\Delta t = 3A + 3A + l \tag{2}$$

To simplify the calculations, it is possible to make the hypothesis that each interval in Figure 3 (acceleration, cruise and deceleration) has the same duration, resulting in $l = 3A$. Under this condition, (1) and (2) become (3).

$$\begin{cases} \Delta\theta_{tot} = 12JA^3 \\ \Delta t = 9A = l \end{cases} \tag{3}$$

The variables shown in (4) are computed according the input variables, $\Delta\theta_{tot}$ (total angle movement) and $\Delta t$ (movement duration). The jerk $J$ is used by the trajectory calculator to integrate (step-by-step) the target position.

$$\begin{Bmatrix} A = \frac{\Delta t}{9} & l = \frac{\Delta t}{3} \\ J = \frac{\Delta\theta_{tot}}{12A^3} & \omega_{cruise} = 2JA^2 \end{Bmatrix} \tag{4}$$

# 4 Position control API functions

The API features the following functions, which are specific to the position control mode:

- `void MC_ProgramPositionCommandMotor1 (float fTargetPosition, float fDuration)`

  Sets the target rotation angle for the motor to `fTargetPosition` expressed in radians. The angles are calculated with respect to the mechanical zero of the motor that corresponds to the position of the Z (index) pulse from the encoder. The movement of the motor can be executed in two different ways:

  – Trajectory control mode

    When `fDuration` is different from 0, the trajectory of the movement, and therefore its acceleration and speed, are computed by firmware. `fDuration` is the duration of the movement expressed in seconds.

  – Follow mode

    When `fDuration` is set to zero, the user can send at fixed rate different target positions according to a required trajectory and the position control algorithm computes the intermediate points to reach (follow) the target with a smooth movement. This mode is for instance useful when the trajectory is computed by an external controller, or by an algorithm defined by the user and executed by the same microcontroller.

- `AlignStatus_t MC_GetAlignmentStatusMotor1(void)`

  Used to check if the encoder is absolutely aligned. This occurs when the z signal pulse has been received for the first time and firmware aligns the angular references with respect to this position.

*Note: Before starting to send target positions, it is mandatory to perform an absolute alignment of the encoder at least once after each microcontroller reset. An encoder absolute alignment procedure is explained in Application example.*

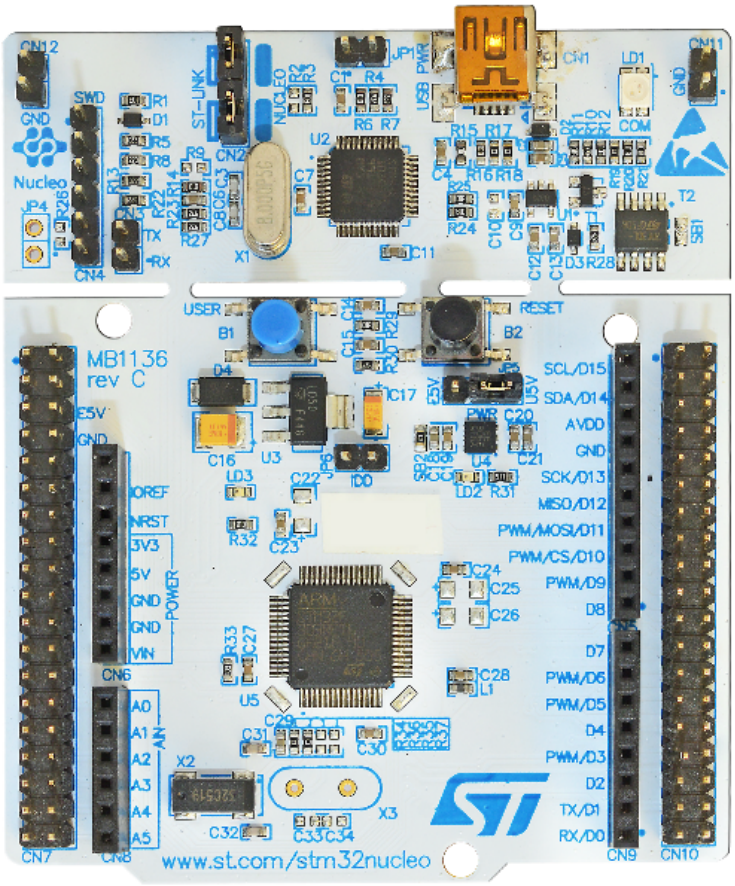
For more details, refer to the motor control firmware reference documentation accessible from the [**Documentation**] menu of the MC Workbench.

# 5 Application example

The application example is developed and tested with the control board and power board presented below:

1. NUCLEO-F302R8 control board (refer to Figure 6)

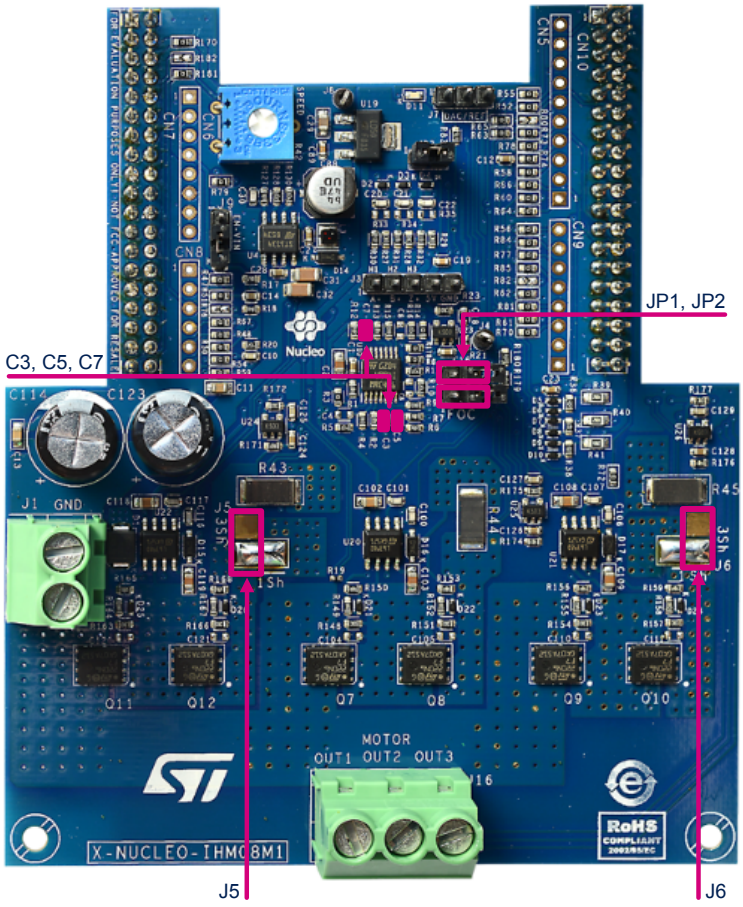
**Figure 6. NUCLEO-F302R8**



2. X-NUCLEO-IHM08M1 power board (refer to Figure 7), stacked on top of the NUCLEO-F302R8 by means of the ST morpho connector.
   a. Three-shunt configuration: J5 and J6 soldered on the 3-Sh side (indicated in the serigraphy).
   b. FOC configuration: JP1 and JP2 ON (closed).
   c. Remove C3, C5 and C7. For more information, refer to user manual *Getting started with X-NUCLEO-IHM08M1 low-voltage BLDC motor driver expansion board based on STL220N6F7 for STM32 Nucleo* (UM1996).
   d. Connect a DC power supply (12 V - 2 A) on J1 respecting the polarity indicated in the silkscreen. When using a different motor rated higher than 12 V, keep jumper J9 on the power board OFF (open) before applying power-on voltage at J1 to avoid damaging the Nucleo board.

      In this application demonstration, only 12 V is used as main power supply, even if the motor has 36 V nominal voltage.
   e. Connect the PMSM motor phases on connector J16.

**Figure 7. X-NUCLEO-IHM08M1**




## 5.1 Test motor

- Supplier: Maxon Motor
- Part number: 496661
- Description: EC-i 40 Ø40 mm, brushless, 100 W
- Nominal voltage: 36 V
- No-load speed: 4550 RPM
- Nominal current: 2.72 A
- Nominal torque: 207 mNm

## 5.2 Position sensor

- Type: quadrature encoder
- Part Number: 499361
- Description: encoder 16 EASY, 3 channels, with line driver RS-422
- Counts per turn: 1024

Connect the motor to the power board as described in Table 4 and Table 5.

**Table 4. Motor phases connections**

| Description | Wire color | X-NUCLEO-IHM08M1 |
|---|---|---|
| Winding 1 | Red | J16 pin 3 (out 1) |
| Winding 2 | Black | J16 pin 2 (out 2) |
| Winding 3 | White | J16 pin 1 (out 3) |

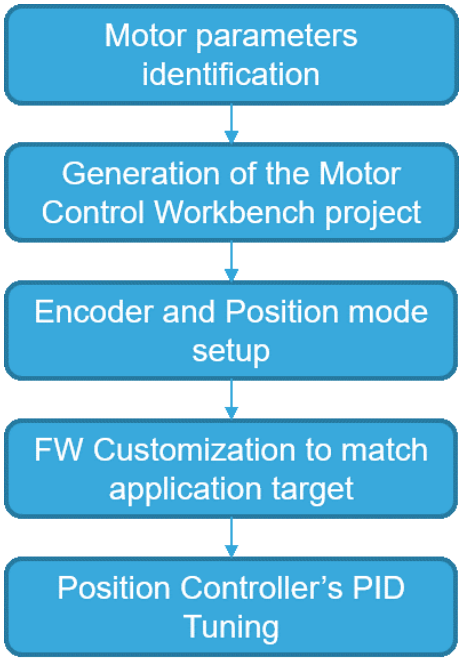
**Table 5. Sensor connections**

| Description | Motor connector | X-NUCLEO-IHM08M1 |
|---|---|---|
| Vcc (+5 V) | Pin 2 | J3 pin 4 |
| GND | Pin 3 | J3 pin 5 |
| Channel A | Pin 6 | J3 pin 1 |
| Channel B | Pin 8 | J3 pin 2 |
| Channel I | Pin 10 | J3 pin 3 |

## 5.3 Project setup

To setup the project, follow the steps described in Figure 8.

**Figure 8. Set up the project**



## 5.4 Motor parameters identification

The first step to generate a firmware project using the Motor Control Workbench is to identify the electrical and mechanical parameters of the motor in use. These parameters are usually specified in the motor datasheet. If they are not available, try to estimate them using the Motor Profiler tool included in the X-CUBE-MCSDK Expansion Package.

The procedure is easy to follow. In a few minutes, the motor parameters are identified and stored in the Motor Control Workbench motors database.

## 5.5 Creation of the Motor Control Workbench project

In MC Workbench, create a new project selecting the hardware combination in use (control and power boards) and the motor characterized in the previous step.

Keep all the settings to their default values (if there are no special needs such as changing the PWM frequency for instance).

## 5.6 Encoder and position mode setup

The position control application requires some additional customization steps, which are further described in this section.

### Configuration of the encoder

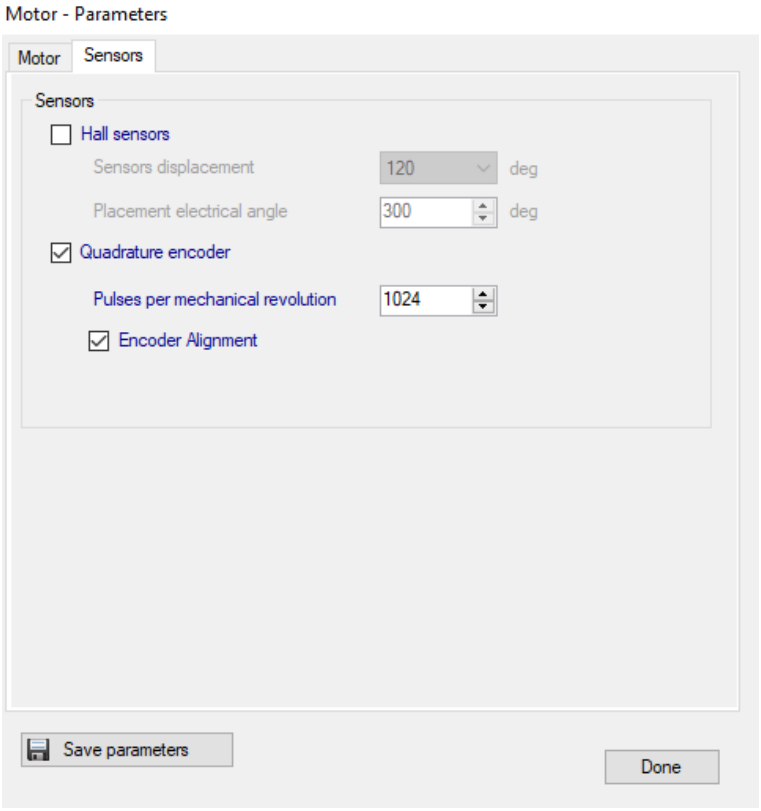
Select in the motor properties window the type of sensor used (position control requires a quadrature encoder) and its resolution in terms of pulse per mechanical revolution as described in Figure 9. Encoder configuration.

Two different quadrature encoders are supported:

- Quadrature encoder with Z (index) signal
- Quadrature encoder without Z (index) signal

If the target encoder has the Z (index), it is necessary to check the *Encoder Alignment* checkbox as shown in Figure 9. Otherwise, the checkbox must be left unchecked.

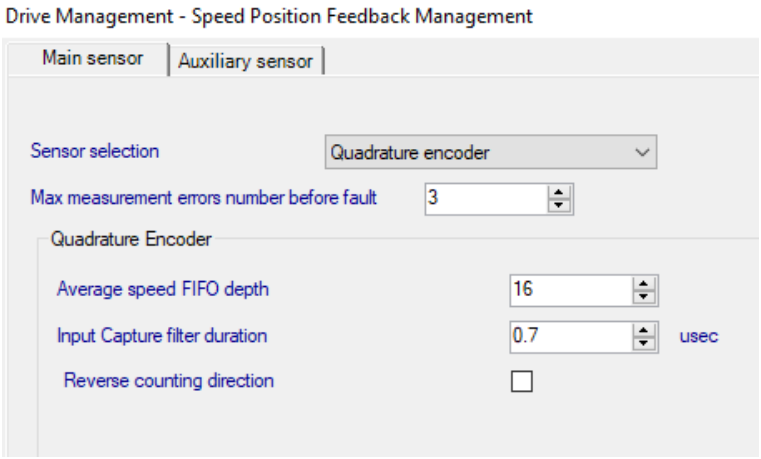
**Figure 9. Encoder configuration**




### Quadrature encoder as primary sensor

Set the *Quadrature encoder* as primary sensor in [**Drive Management**]>[**Main sensor**] section as shown in Figure 10. Main sensor settings.

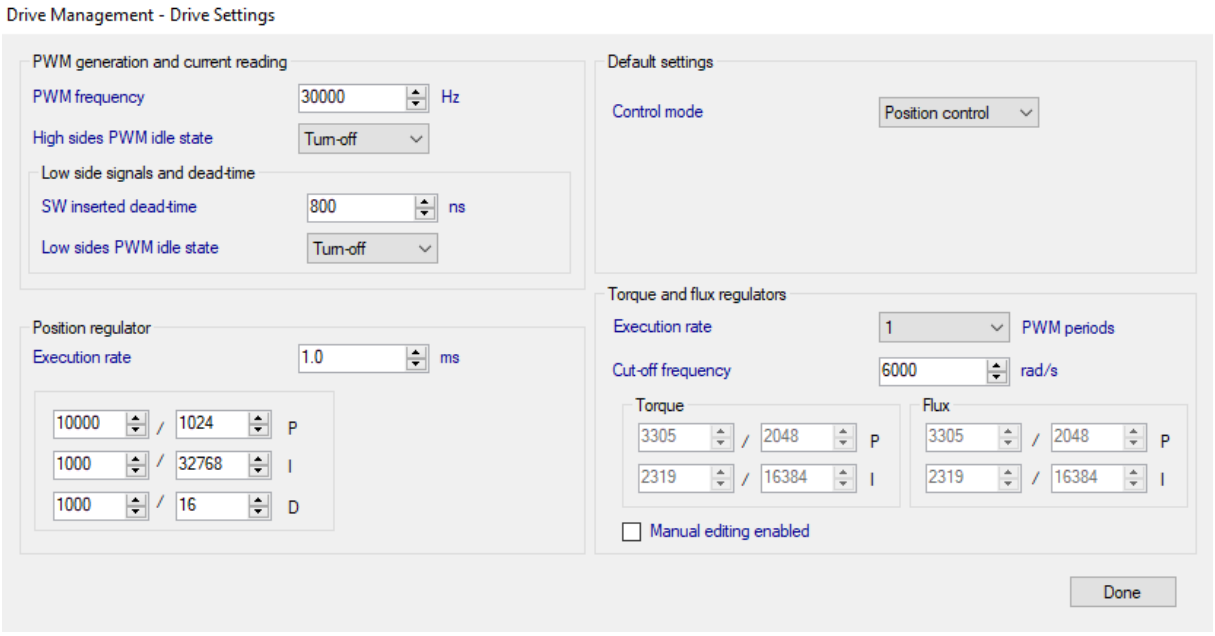
**Figure 10. Main sensor settings**



### Position Control as Control mode

Select the *Position control* as *Control mode* in [**Drive Management-Drive Settings**] window as shown in Figure 11. Selection of position control mode.

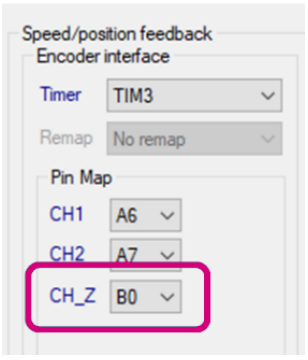
**Figure 11. Selection of position control mode**




### Pin for the Z pulse

Using an encoder with Z (index) input and setting the MC Workbench as described in Configuration of the encoder, Quadrature encoder as primary sensor and Position Control as Control mode, it is possible to define in [**Control Stage – Digital I/O**] which pin of the microcontroller is used for receiving the Z pulse. The pin used can be set in the CH_Z drop-down menu as described in Figure 12. Setting of the pin used for Z (index).

**Figure 12. Setting of the pin used for Z (index)**




The pin that can be set for this purpose can be also different from the one proposed by the MC Workbench. In such case, the following modifications are required to setup firmware manually.

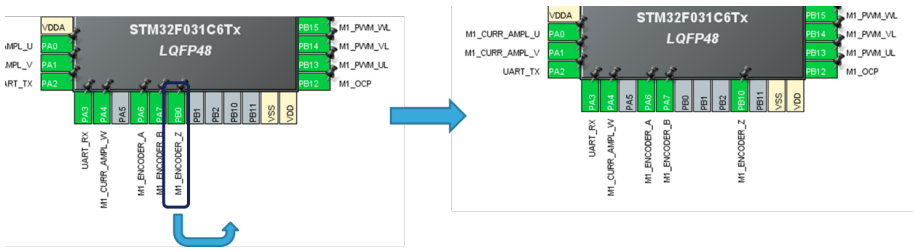
1. Generate the project using the ST Motor Control Workbench with a dummy setting for the encoder CH_Z (for instance PB0 like shown in Figure 10. Main sensor settings).

2. Open the generated `.ioc` file with a text editor and change manually the definition of the ENC_Z_GPIO_PORT and ENC_Z_GPIO_PIN with the required pin. For instance, to use pin PB10 instead of pin PB0, modify the two definitions as:

```
MotorControl.ENC_Z_GPIO_PIN = LL_GPIO_PIN_10
MotorControl.ENC_Z_GPIO_PORT = GPIOB
```

3. Open the modified `.ioc` file with STM32CubeMX. The selected pin for encoder CH_Z is the one generated by the MC WB and not the one modified manually.
4. Remap the M1_ENCODER_Z pin from the generated one (PB0) to the required one (PB10 in this example).
   Left click on the pin (PB0) and select *Reset State* from the drop-down list. Left click on the pin to be used (PB10) and select GPIO_EXTIxx from the drop-down list.
   Figure 13 shows the modification related to the example taken in consideration (from PB0 to PB10). Take care to maintain the coherency between the manual modification of the `.ioc` file and the modification done graphically in STM32CubeMX.

**Figure 13. Remap the generated M1_Encoder_Z pin**



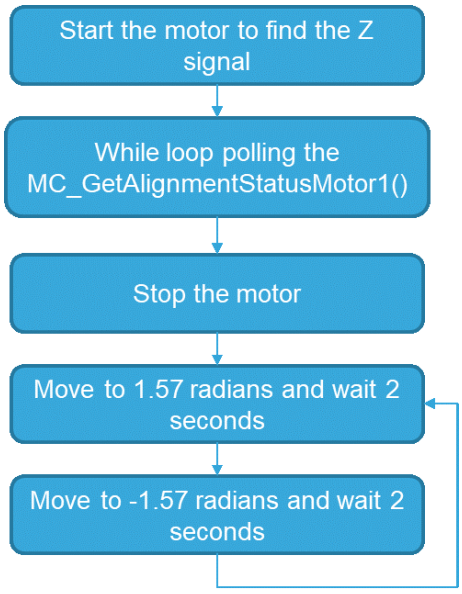

5. Generate the code using STM32CubeMX. This produces the firmware project with the correct settings.
6. The pin that can be used as encoder Z (index) can be any free pin with interrupt capability (verify it in the related microcontroller datasheet).

## 5.7 Firmware customization

At this stage, the generated firmware project is a blank application, able to perform the position control but with no application logic.

The next sections explain how to add the application logic shown in Figure 14 as an example application.

**Figure 14. Application logic**



## 5.8 API usage

The test application makes use of the following functions:

- `MC_StartMotor1()`

  This command performs an auto-start of the driving with both the encoder alignment procedure and the encoder zero reset.

  – The encoder alignment procedure is needed to synchronize the stator flux reference with the quadrature encoder electrical reference. This procedure must be done only once after each reset of the microcontroller and before running the motor. It must be done whether the quadrature encoder has the Z (index) signal or not. It forces a constant magnetic field into the stator at a programmed angle and sets the encoder angle to this value.

  – The encoder zero reset procedure is needed to synchronize the Z (index) pulse, if present, with the zero mechanical reference for the position control (any angle reference is relative to this zero reference). This procedure must be done only once after each reset of the microcontroller and before any movement command. If the encoder has the Z (index), the motor is rotated by 360° until the Z pulse is reached and then set as zero. If the encoder does not have the Z (index), the starting position is set as zero reference. Set the initial mechanical position manually before starting the motor.

```
/* USER CODE BEGIN 2 */
MC_StartMotor1();
/* USER CODE END 2 */
/* Infinite loop */
```

- `MC_GetAlignmentStatusMotor1()`

  While the motor is spinning searching the Z pulse, a polling on this function is done to check when the alignment is completed (the Z pulse reception is detected). The code is:

```
while(MC_GetAlignmentStatusMotor1()!=TC_ALIGNMENT_COMPLETED){}
```

- `MC_ProgramPositionCommandMotor1`

  In this simple example, two different movement commands are given. The first one to +90° (π/2) in 100 ms and the second one to -90° (-π/2) in 100 ms. A delay of two seconds is set between the two commands, which are given in a loop.

The complete code of the main loop is:

```
/* USER CODE BEGIN 2 */
MC_StartMotor1();
/* USER CODE END 2 */
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while(MC_GetAlignmentStatusMotor1()!=TC_ALIGNMENT_COMPLETED){}
while (1)
{
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
  MC_ProgramPositionCommandMotor1(3.14/2,0.1);
  HAL_Delay(2000);
  MC_ProgramPositionCommandMotor1(-3.14/2,0.1);
  HAL_Delay(2000);
}
/* USER CODE END 3 */
```

## 5.9 PID tuning

At this stage, it is time to fine tune the position controller by adjusting the coefficients.

This is an essential step in setting up the application since wrong parameters can lead to dangerous oscillations of the motor when it tries to reach the target position. It can even happen that the motor does not spin if the parameters are not correcty adjusted.

The position PID can be tuned manually on-the-fly using the live watch feature of the debugger. The structure to be modified is `PIPosParamsM1`:

```
PID_Handle_t PID_PosParamsM1=
{
…
int16_t hKpGain;
int16_t hKiGain;
int16_t hKdGain;
…
};
```

To tune a PID configuration, use the following steps while the motor is executing the demonstration code:

1. Set all gains to zero
2. Increase the P gain until the motor starts to oscillate
3. Increase the D gain until the oscillations disappear (meaning that critical damping is achieved) and the demonstration movements are performed in a stable way
4. Repeat steps 2 and 3 until increasing the D gain cannot stop the oscillations
5. Set P and D to the last stable values
6. Increase the I gain until the set point with the number of oscillations desired (normally zero, but a quicker response can be reached provided a couple oscillations of overshoot are accepted)

The above-mentioned movements are given to tune dynamically the PID with the proposed method. Once fine-tuned, the parameters can be stored in the MC Workbench project to be used in final firmware. From that point, it is possible to customize the program by adding all the movements and logic required by the application.

# Revision history

Table 6. **Document revision history**

| Date | Version | Changes |
|---|---|---|
| 2-Apr-2020 | 1 | Initial release. |

# Contents

# List of tables

# List of figures