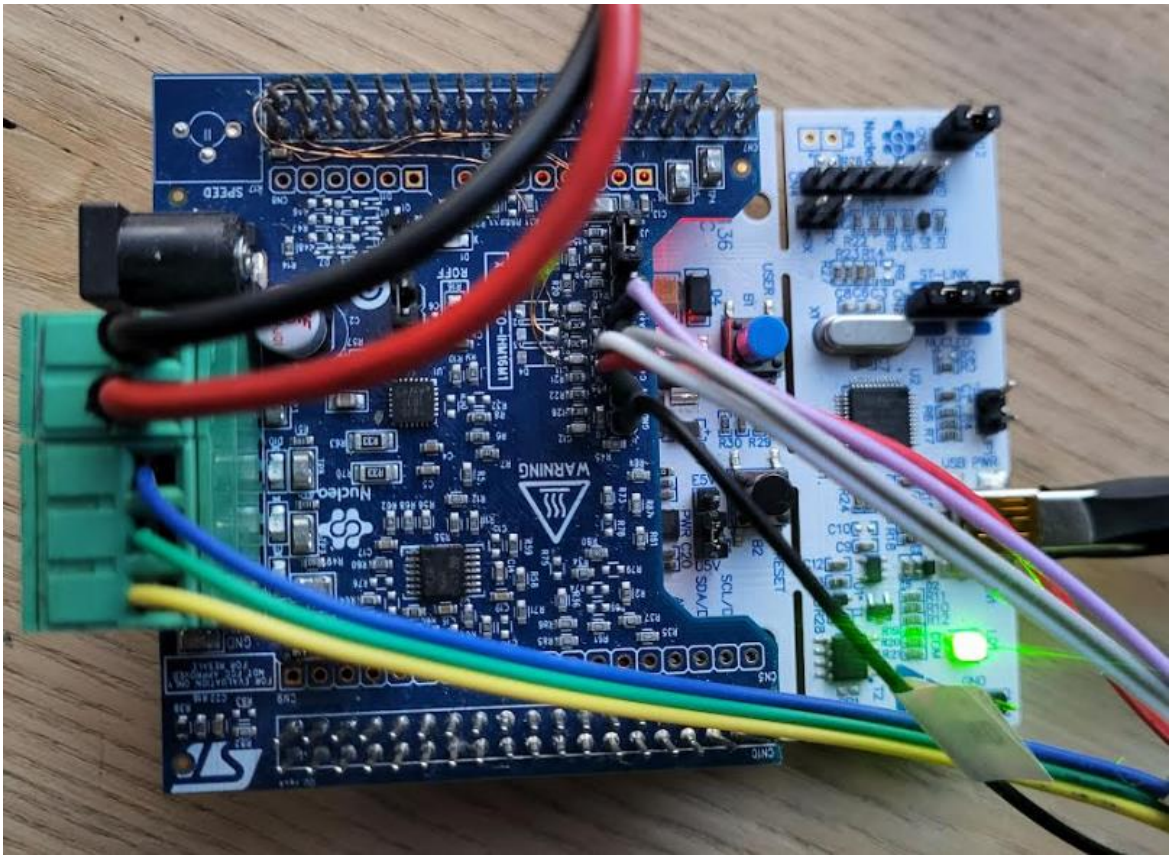
Bystronic Laser AG
Industriestrasse 21
CH–3362 Niederönz
Switzerland

T +41 62 956 33 33
F +41 62 956 33 80
info.laser@bystronic.com
www.bystronic.com

ca

Best choice.

Technical Report



Infos

Writer	Younès Ghanem
Version	0.0
Comment	Initial Version
Contact	younes.ghanem@nanotronic.ch
Interesting Links to get started	AN_Analog_Hall_Sensor AN5464_PositionControl

Introduction

This report is intended for the embedded systems team at Bystronic. It aims to provide an overview of the progress made so far in the development of software for precise motion control applications. The goal is to document key concepts, developments, and solutions to ensure the project can be easily picked up and continued by others in the future.

Scope of the Work

The focus of this report is on the concepts and development of the software related to precise motion control. I will not cover the installation or configuration of the various software tools in detail, but relevant links and resources are provided for reference. The core of this report emphasizes the software development process, results, and solutions discovered thus far.

Requirement

A simple proof of concept consists of achieving a mechanical angle resolution of 0.5 degree (mechanical).

Feasibility

For a BLDC motor controlled by an STSPIN830 and Nucleo STM32L476, using 12-bits ADCs and and analog Hall Sensors, with a PID loop running at 1000 Hz and FOC current regulation at 16 KHz.

Prerequisites & Know Information

PWM Frequency (FOC loop):

`PWM_FREQUENCY = 16000 Hz`
The FOC execution rate is synchronized with the PWM cycles, running at 16 KHz.

FOC Execution Rate:

`ISR_FREQUENCY_HZ = 16000 Hz` (as `REGULATION_EXECUTION_RATE = 1`).

Position Control Loop

The PID controller for mechanical angle control is updated every millisecond.

Analog Hall Sensors
We are using analog Hall sensors, continuously capturing the rotor position data. The signal is read by a 12-bit ADC.

ADC Resolution

```
12-bit ADC = 4096 steps
```
The analog signal from the Hall sensors provides 4096 distinct values for the rotor position.

Mechanical Angle Resolution Target
We aim for a resolution of **0.5 degrees** for the rotor's mechanical angle.

Rotor Position Update
Rotor position is updated every millisecond via DMA, with the data from the Hall sensors processed by the ADC.

Motor Pole Pairs
The motor has 2 pole pairs, meaning for every electrical revolution, the rotor makes half a mechanical revolution.

Estimation of Feasibility

Total Mechanical Range

Since the motor has 2 pole pairs, one complete electrical revolution corresponds to:

$$\frac{360\ deg.mechanical}{2} = 180\ deg.mechanical\ per\ revolution$$

To achieve a 0.5° resolution, we need:

$$\frac{360\ deg.mechanical}{0.5\ deg} = 720\ mechanical\ steps\ per\ full\ mechanical\ revolution$$

However, with 2 pole pairs, we are only considering 180° mechanical per electrical cycle.

Hall Sensor Output (Analog with 12-bit ADC)

With a 12-bit ADC, we have 4096 steps for each signal. The sensor must be able to discern 0.5° per step, and for 2 pole pairs (180° mechanical per electrical revolution), this translates to:

$$\frac{4096\ ADC\ steps}{180\ deg.mechanical} = 22.75\ ADC\ steps\ per\ mechanical\ degree$$

For 0.5-degree mechanical resolution, we get approximately:

$$22.75 \text{ x } 0.5 = 11.37 \text{ ADC steps per } 0.5°$$

This is more than enough to distinguish the rotor position changes with the desired resolution

Update Rates

Position Control loop: 1000 Hz (1ms) PID loop updates ensure that position errors are corrected frequently.

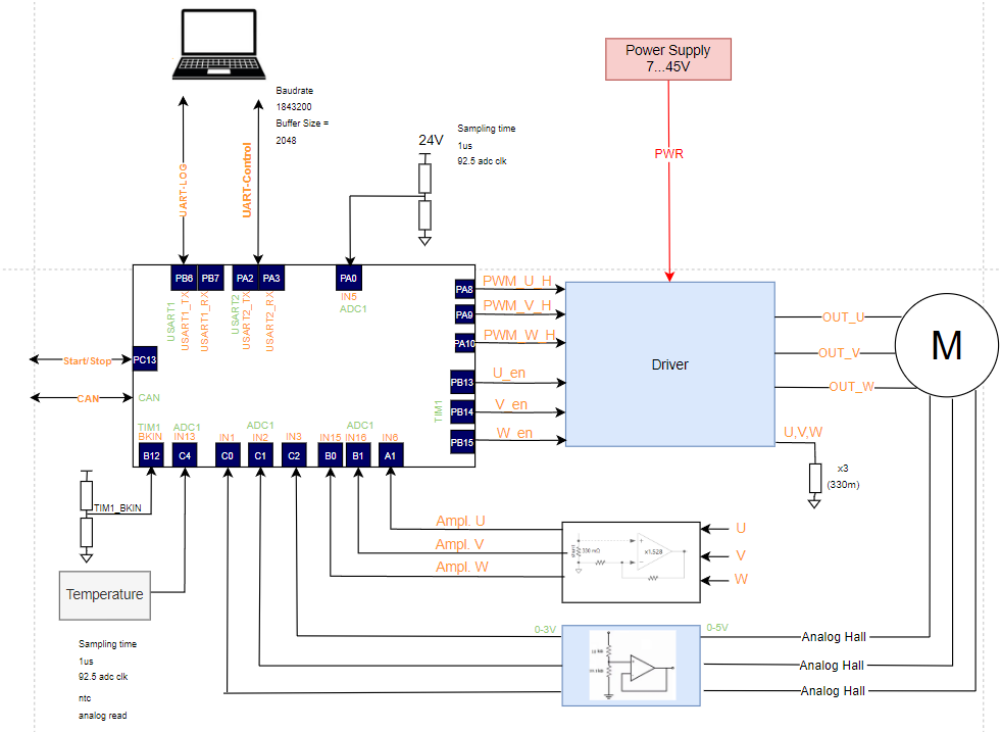
FOC Execution Rate: 16000 Hz allows for accurate and real-time phase current regulation.

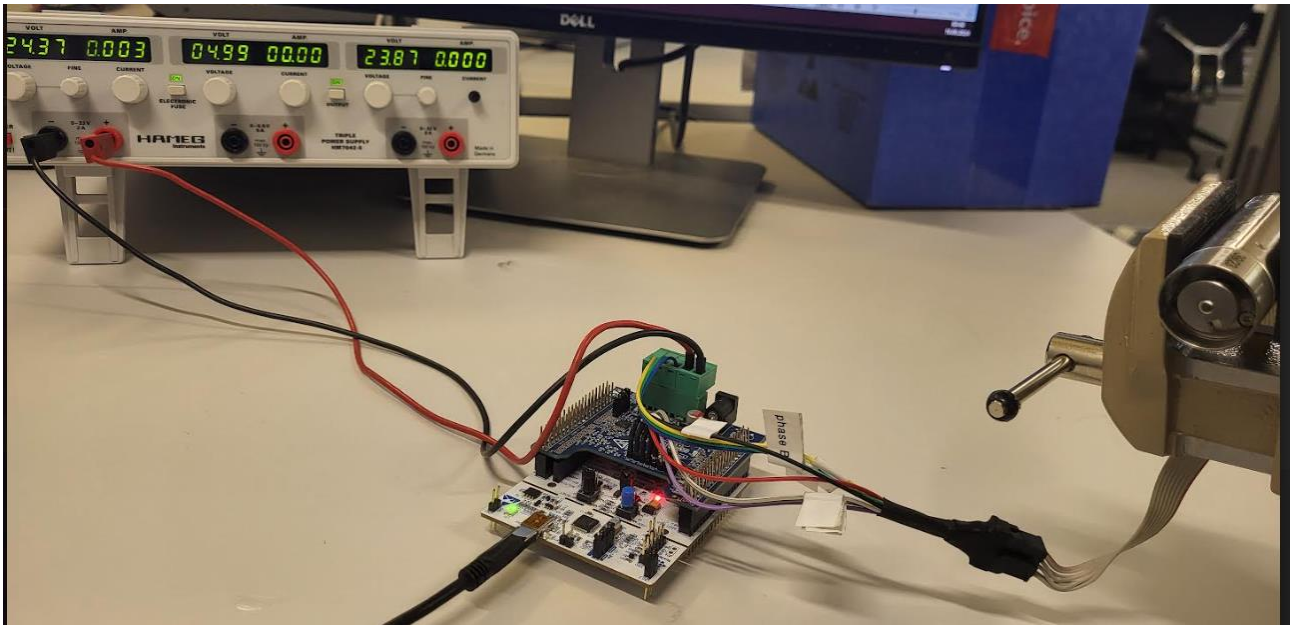
Getting Started

Hardware setup

You'll need following hardware to get started.

- NUCLEO-L476
- A modified X-NUCLEO-IHM16
- BLDC BX4
- Power supply +24V DC
- USB mini cable
- PC


Power Supply
7...45V
PWR
Baudrate
1843200
Buffer Size =
2048
UART-LOG
UART-Control
24V
Sampling time
1us
92.5 adc clk
PB8
PB7
PA2
PA3
PA0
IN5
ADC1
USART1_TX
USART1_RX
USART2_TX
USART2_RX
PA8
PA9
PA10
PWM_U_H
PWM_V_H
PWM_W_H
Start/Stop
PC13
CAN
PB13
PB14
PB15
U_en
V_en
W_en
TIM1
Driver
OUT_U
OUT_V
OUT_W
M
U,V,W
x3
(330m)
TIM1_BKIN
B12
C4
C0
C1
C2
B0
B1
A1
Ampl_U
Ampl_V
Ampl_W
U
V
W
Temperature
0-3V
0-5V
Analog Hall
Analog Hall
Analog Hall
Sampling time
1us
92.5 adc clk
ntc
analog read


HAMEG
DELL

Software tool setup

You'll need to install the following software to get started and communicate with the system:

- MotorControl Workbench Link
  MotorControl Workbench is a user-friendly interface that allows you to configure, tune, and monitor the motor control parameters for your BLDC motor. It helps you set up the hardware and software environment quickly by providing various configuration wizards. It also provides real-time data monitoring and control capabilities

- ST Motor Pilot Link
  ST Motor Pilot is a software tool designed for easy and intuitive control of motors. It allows you to interact with and command the motor system, providing a real-time interface to control motor operations, monitor key parameters, and troubleshoot motor performance. It's used in conjunction with MotorControl Workbench to fine-tune motor behavior and ensure optimal performance in various applications.

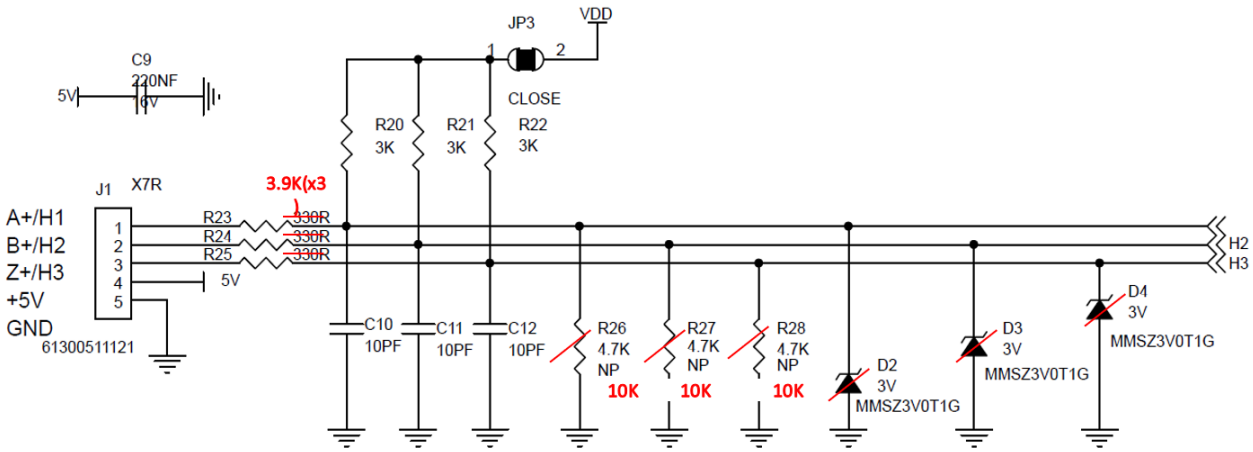
Toolchain

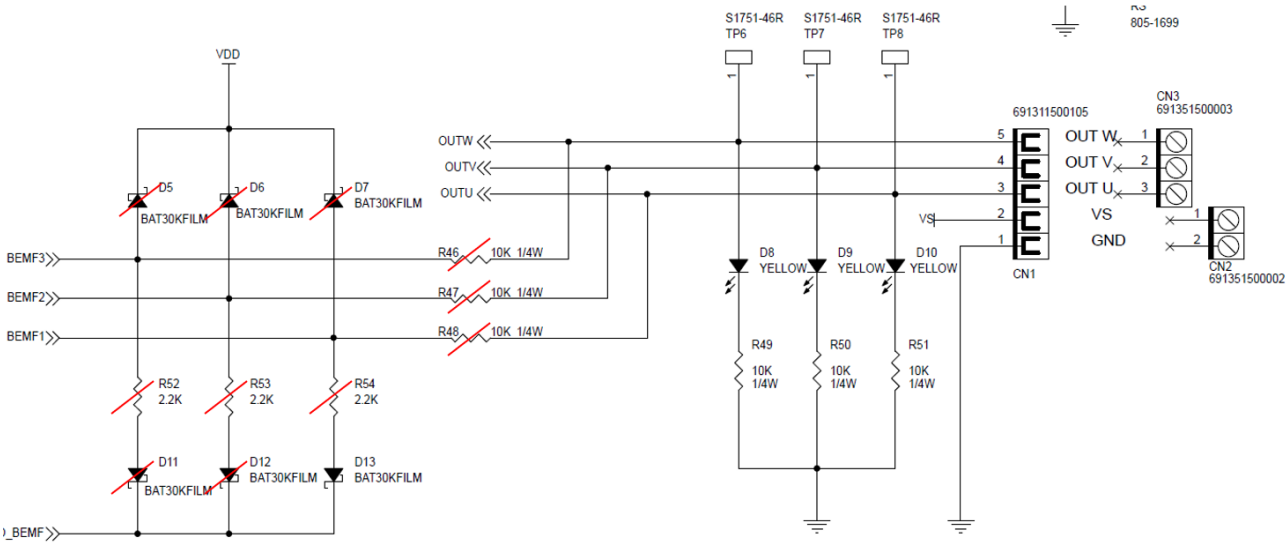
For the development environnement, I used VSCode, configured with STLink and GDB for debugging. This setup was done to support efficient code writing, compiling and debugging. VSCode, along with the STLink, allows you to flash the firmware onto the NUCLEO board and perform real-time debugging using GDB. This combination provides a flexible, lightweight toolchain ideal for embedded development and motor control applications.

HW-Changes

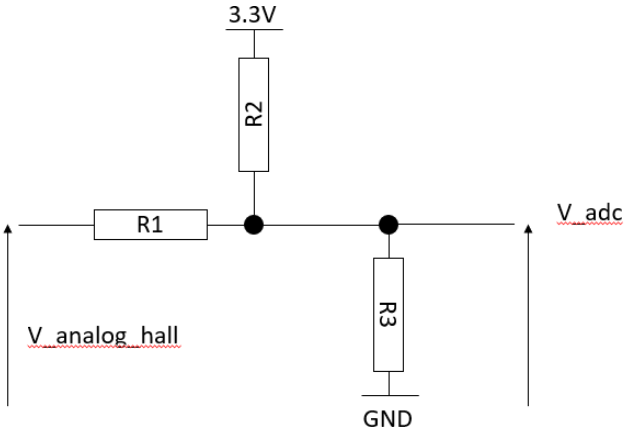
Note: We are using a Faulhaber BLDC motor with analog Hall sensors, which requires some modifications to the hardware setup. The Nucleo evaluation board is designed for use with encoders or digital Hall sensors, so adjustments need to be made in order to properly interface the analog Hall sensors with the motor control system.

Hall sensors





Equivalent Circuit

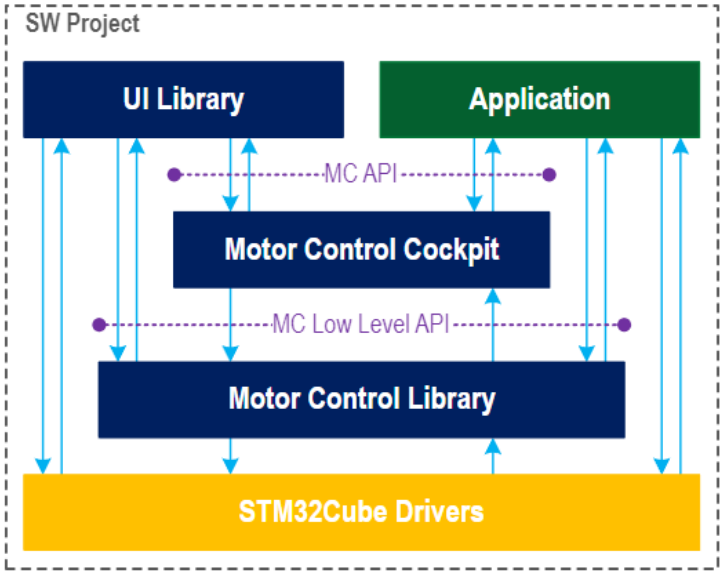


$$V_{adc} = V_{analog\_hall} \cdot \frac{R3}{R1 + R3} + 3.3V \cdot \frac{R2}{R1 + R3}$$

Here is the graph showing the transfer function V_{adc} as a function of V_{hall} for the given resistor values. In practices, we never exceed the VREF Hall sensor voltage.

Software Architecture

Here’s the architecture in ST’s motor control framework.
It abstracts away much of the low level complexity.

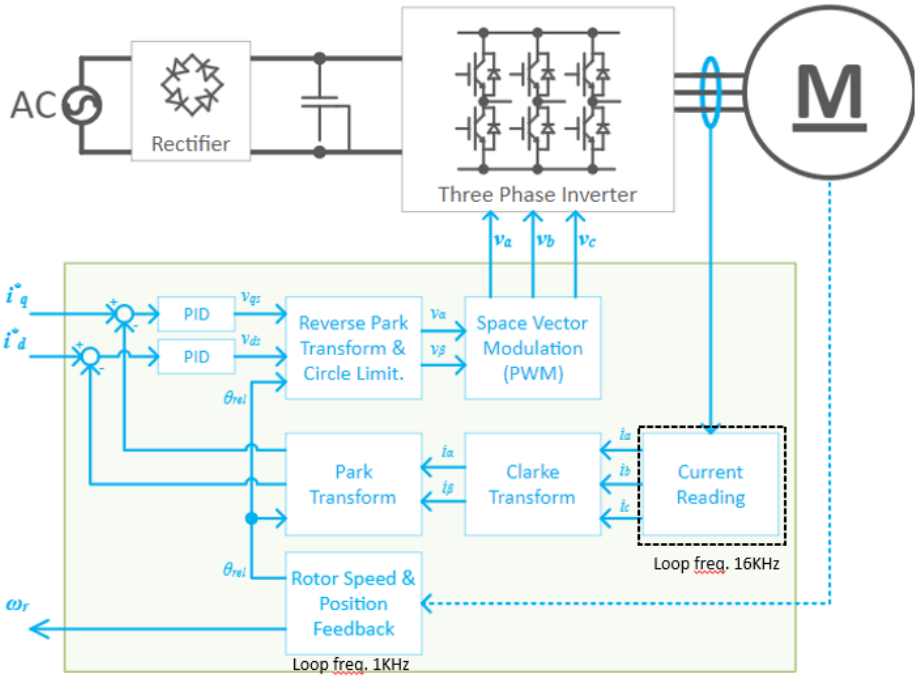


The architecture breaks down the control system into layers:

- STM32Cube Drivers (containng the HAL)
- Motor Control Library (Core Algorithm Handling)
- Motor Control Cockpit (GUI)
- UI Library (HMI, MotorControl Workbench)
- The Application (where we put our code)

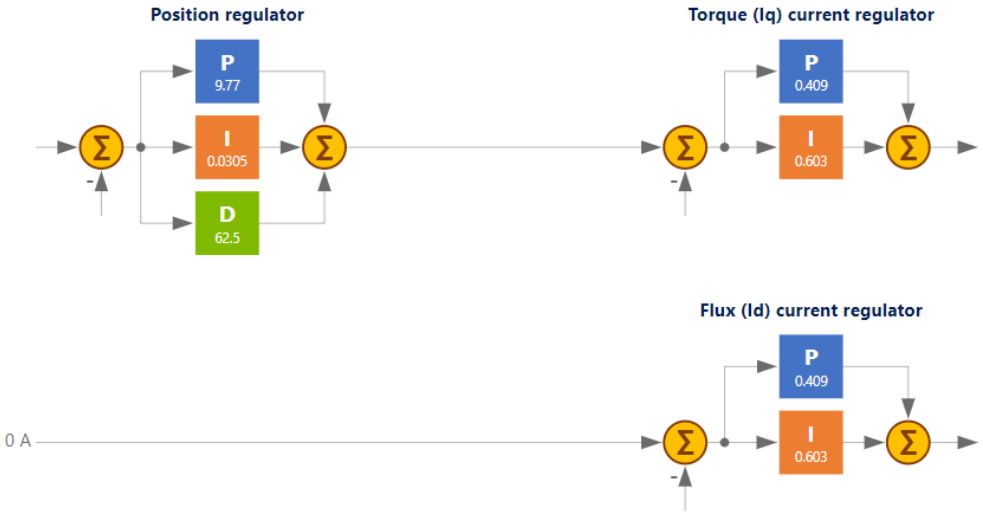
Software

Global overview



Position Control loop

Here’s a global overview of the position control loop.



Structure

We use a cascaced control structure, involving nested control loop (eg., position, speed, and current control). This structure allows a hierarchical control over specific variables.

In our case, we're working on position control, the position loop provides a speed reference, which is translated in torque (current) at it cascades down to the lower loops. This allows precise position control with smooth transitions in speed and torque.

Position control loop

- Starting from a position reference (the desired rotor position)
- The position control loop compares the desired position to the actual position
- The output of this loop is a speed command

Speed control loop

- The speed control loop takes the speed command from the position lop and compares it to the actual speed
- The output of this loop is a torque command which becomes in practice a current reference. This is because the torque is proportional to current.
-

Current loop

- The current control loop takes the current reference (derived from the torque command) and ensures the motor's phase current follow this reference by controlling the applied voltage.

In our case, we're working on position control, the position loop provides a speed reference, which is translated in torque (current) at it cascades down to the lower loops. This allows precise position control with smooth transitions in speed and torque.

Timing

Current Control Loop (FOC)	16 KHz
Speed and Position Control loop	1KHz

Initialization

In our code, a large number of structures are initialized separately and then assigned to the high-level `MCI_Handle_t` structure, which consolidates all these components. This modular approach helps manage different parts of the motor control system (like speed, torque, position, etc.) in an organized and maintainable way.

Individual components

Various components like PID controllers, Hall), current regulators, speed and torque controllers, etc., are initialized independently. Each component has its own handle structure.

For example, the following commands initialize specific modules:

```
PID_HandleInit(&PIDSpeedHandle_M1);
HALL_Init(&HALL_M1);
HAC_Init(&HallAlignCtrlM1, pSTC[M1], &VirtualSpeedSensorM1, &HALL_M1);
PID_HandleInit(&PID_PosParamsM1);
TC_Init(&PosCtrlM1, &PID_PosParamsM1, &SpeednTorqCtrlM1, &HALL_M1);
```

These commands configure separate components such as the speed PID controller, Hall sensor, position controller, etc.

Assignment to MCI_Handle_t:

After initializing these structures, they are linked to the high-level MCI_Handle_t structure. This structure contains pointers to each component, which allows grouping all the motor control elements together.

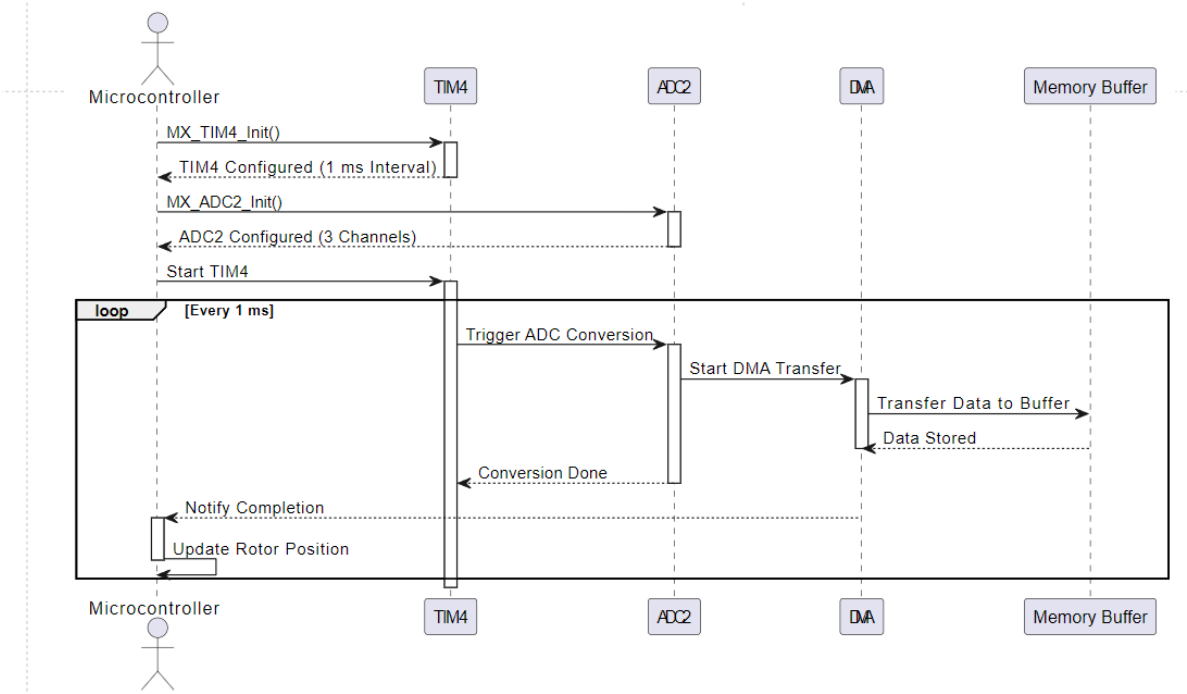
For example:

```
MCI_Init(&Mci[M1], pSTC[M1], &FOCVars[M1], pPosCtrl[M1], pwmcHandle[M1]);
Mci[M1].pScale = &scaleParams_M1;
```

Here, `MCI_Init` ties together the speed and torque controller (`pSTC[M1]`), the FOC variables (`&FOCVars[M1]`), the position controller (`pPosCtrl[M1]`), and the PWM handler (`pwmcHandle[M1]`) to the `MCI_Handle_t` structure (`Mci[M1]`).

```
MCI_Handle_t
  ├── pSTC (Speed and Torque Control)
  ├── pPWM (PWM Control)
  ├── pFOCVars (FOC Variables)
  ├── pPosCtrl (Position Control)
  └── ... (other pointers and parameters)
```

Position Regulation



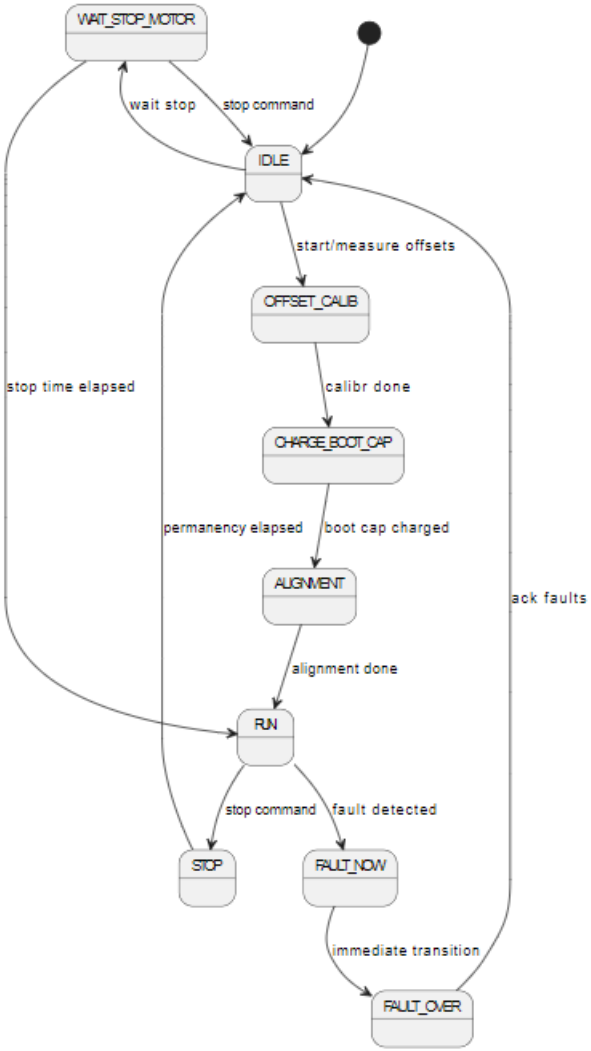
TIMER4:
TIM4 is initialized to create a 1ms interval trigger using the TRGO output.
This trigger is crucial for ensuring that the ADC2 conversions occur at the precise intervals needed to maintain accurate rotor position.

ADC2 Initialization:
The ADC2 is configured with three channels. Each channel corresponds to a specific input related to the rotor's position sensors. The ADC operates in 12-bit resolution mode and is triggered externally by TIM4.

Trigger and Conversion:
When TIM4 triggers, ADC2 starts converting the input channels. The DMA then takes over the task of moving the converted data to the memory buffer.

Completion and Processing:
Once all three channels are converted and the data is transferred, the DMA signals completion. The microcontroller can then process the data, typically to update the rotor position for the BLDC motor.

Motor State machine



GUI

The code is configuring the Motor Control Protocol (MCP), specifically for communication between the Motor Control SDK and the Motor Pilot external interface, using UART. The configuration is designed to handle both synchronous and asynchronous communication with the GUI, using buffers and DMA channels.

Key Elements of the Code for UART Communication with the GUI

Buffers for Synchronous and Asynchronous Communication:

Synchronous Buffers

```
static uint8_t MCPSyncTxBuff[MCP_TX_SYNCBUFFER_SIZE]
__attribute__((aligned(4)));
static uint8_t MCPSyncRXBuff[MCP_RX_SYNCBUFFER_SIZE]
__attribute__((aligned(4)));
```

These buffers are used for synchronous communication, which requires the data to be transmitted and received in a coordinated manner, ensuring that both the sender and receiver are ready at the same time.

Asynchronous Buffers for UART_A

```
static uint8_t MCPAsyncBuffUARTA_A[MCP_TX_ASYNCBUFFER_SIZE_A]
__attribute__((aligned(4)));
static uint8_t MCPAsyncBuffUARTA_B[MCP_TX_ASYNCBUFFER_SIZE_A]
__attribute__((aligned(4)));
```

These buffers are used for asynchronous communication, which allows transmission and reception to happen independently without the need for both sides to be ready at the same time.

Transport Layer Setup (ASPEP)

ASPEP (Asynchronous Serial Protocol for Embedded Platforms) is initialized for UART_A communication:

```
static UASPEP_Handle_t UASPEP_A =
{
  .USARTx = USARTA,
  .rxDMA = DMA_RX_A,
  .txDMA = DMA_TX_A,
  .rxChannel = DMACH_RX_A,
  .txChannel = DMACH_TX_A,
};
```

The `ASPEP_Handle_t` structure defines the communication interface for the UART, including:
The USART peripheral to use (`USARTA`).
DMA channels for RX and TX, which handle the data transfers in the background, improving efficiency and offloading the CPU.

High-Level Protocol Handler (MCP and MCPA)

The `MCP_Handle_t` structure connects the MCP protocol to the transport layer (in this case, UART_A):

```
MCP_Handle_t MCP_Over_UartA =
{
 .pTransportLayer = (MCTL_Handle_t *)&aspepOverUartA, //cstat !MISRAC2012-Rule-11.3
};
```

This structure enables the MCP protocol to use the underlying UART communication setup (`aspepOverUartA`), which manages how data is sent and received.

```
MCPA_Handle_t MCPA_UART_A =
{
```

```
  .pTransportLayer = (MCTL_Handle_t *) &aspepOverUartA, //cstat !MISRAC2012-
Rule-11.3
  .dataPtrTable = dataPtrTableA,
  .dataPtrTableBuff = dataPtrTableBuffA,
  .dataSizeTable = dataSizeTableA,
  .dataSizeTableBuff = dataSizeTableBuffA,
  .nbrOfDataLog = MCPA_OVER_UARTA_STREAM,
};
```

This structure holds pointers to data that will be streamed over UART_A to the GUI, allowing real-time data logging and monitoring.

How This Communicates with the GUI:
The MCP protocol is responsible for communicating configuration, status, and control information between the embedded system and the ST Motor Control Workbench GUI via UART.
The data exchange can happen synchronously (using the MCPSyncTxBuff and MCPSyncRXBuff) or asynchronously (using MCPAsyncBuffUARTA_A and MCPAsyncBuffUARTA_B).
ASPEP handles the low-level UART communication, while MCP (Motor Control Protocol) manages the higher-level communication protocol for motor control, such as exchanging parameters or commands.
The GUI can request data from the MCU, send control commands, and log real-time data such as speed, torque, and voltage values using this protocol.

Results

We are encountering an issue where the motor starts abruptly when we try to move it to a specific position using MC Pilot, and then either stops or keeps spinning, eventually triggering a driver error. The fact that we're seeing large **Iq** and **Id** values (where **Id** should ideally stay close to zero) suggests there may be a deeper problem with our control loop or motor configuration.

Possible Causes to investigate

Incorrect PID Tuning

If the PID controller is not tuned correctly, it can overshoot or oscillate, leading to large current (Iq, and Id) and instability.
When the PID is too aggressive, we might see sudden jumps in torque (reflected by Iq).

FOC misconfiguration

If the FOC isn’t properly configured, the decoupling between Id and Iq may not work as expected. This can lead to large Id values even though they should remain zero during normal operation.

Ensure that the FOC is properly calibrated, particularly the phase alignment between the rotor and stator. Double-check your Hall sensor feedback or encoder data for rotor position sensing accuracy.

Position feedback issue

Position feedback is noisy, delayed, or incorrect, the motor could be receiving erroneous information about its current position. This can cause the control system to apply incorrect torque, leading to high Iq and Id values.

Verify the integrity and accuracy of your position feedback. Ensure there is no noise or jitter in the sensor readings. Also, ensure the rotor position updates (e.g., via DMA) are occurring correctly.

Electrical/Hardware Issues

High current values could also be due to a hardware issue such as poor wiring connections, faulty drivers, or even incorrect power supply levels, which could cause erratic motor behavior. Solution: Double-check the power connections, drivers, and any hardware that could influence the current readings or motor response.