Dodo Alive! - Resurrecting the Dodo with Robotics and AI: Hardware Design - Electronics -

Version 1.0.0

Hector Betancourt, Sergio Zerpa, Thaddäus Burger

August 15, 2023

Contents

1 System Overview

1.1 Introduction

The main motivation behind this embedded project is to explore the functionalities and capabilities of the MiniCheetah motor controller. We aim to see how it could be adapted for our Dodo robot in the future. Our focus is on understanding this controller in detail. To do this, we have set up a system based on the motor control implementation available at [2]. The primary goal is to thoroughly document this implementation, making it easier for future teams to adapt it for the Dodo robot.

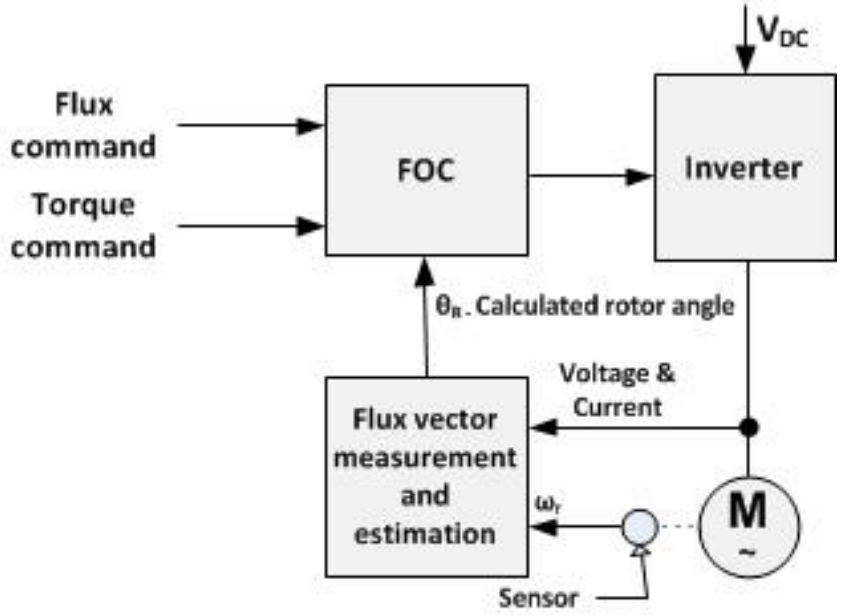

Figure 1: Simplified FOC diagram [1].

The MiniCheetah controller uses a Field Oriented Control (FOC) mechanism (See Figure 1). This allows for separate control of the motor's torque and flux by managing the torque-producing and magnetising currents independently. The process involves the following steps:

1. Measure the current flowing through the motor's phases (stator current), which includes both the magnetising and the torque-producing current (in the stator reference frame).
2. Measure rotor position and speed, and use them along with the stator current to indirectly estimate the rotor current. The rotor current also includes both the magnetising and the torque-producing current (in the rotor or dq reference frame).
3. Separate the magnetising and the torque-producing current to control them independently.
4. Use Park transformations to convert stator currents into the dq reference frame.
5. Compare desired currents with actual currents to calculate the error. Desired currents are derived from desired torque and flux references.

6. Use a PI controller to generate the control output currents.
7. Use Park transformations to convert the currents back to the stator reference frame.
8. Apply control signals through PWM.

The following sections will explain how the FOC controller was implemented on an STM32F4 microcontroller, and how this is connected to the motor, motor encoder, motor driver, and communication interfaces.

1.2 System Architecture

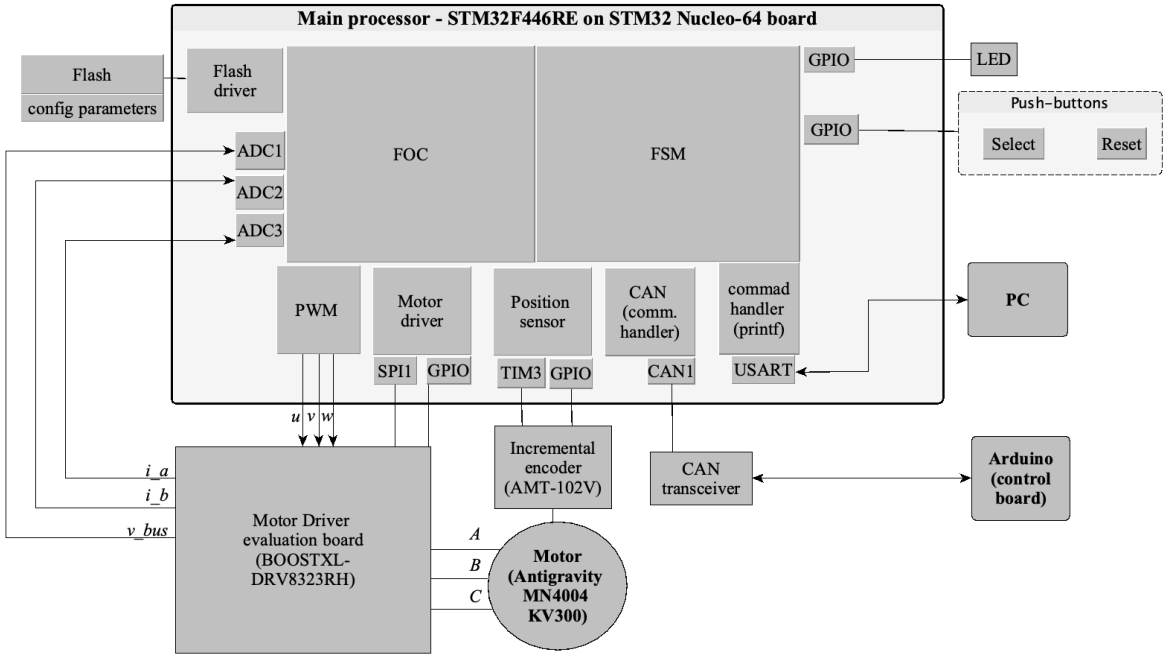

Figure 2: System block diagram.

As shown in Figure 2, the motor control system consists of the following components:

1. **Motor**: The motor to be controlled, which is connected to the motor driver and encoder (See motor specification at [10]).
2. **Motor Driver**: Responsible for driving the motor and delivering the appropriate power and signals for motor control (See specification at [7]).
3. **Encoder**: Provides feedback on the motor's position, allowing closed-loop control (See specification at [5]).
4. **Microcontroller**: An STM32F446RE microcontroller (on a Nucleo-64 board) where the different software modules are implemented (See specifications at [8] and [9])).

5. **Communication Interfaces**:
    - CAN (Controller Area Network): Enables communication with an Arduino board which can control the system.
    - Serial Communication: Serves as a user interface for interacting with the motor control system from a PC.

The microcontroller implements the following software modules:

- **Finite State Machine (FSM) Module**: This module manages the different states of the motor control system. It handles the transitions between set-up, calibration, and motor control modes. The FSM ensures a smooth and well-organized control flow throughout the system's operation.
- **FOC Module**: The FOC algorithm is the heart of the motor control system. This module is responsible for computing the appropriate control signals to achieve precise and efficient motor control based on the inputs received from the position sensor module and ADCs.
- **Position Sensor Module**: This module utilizes data from the encoder to determine the motor's position and speed accurately. The information obtained from the encoder is crucial for closed-loop control.
- **PWM Control Signal Generation**: The module generates Pulse Width Modulation (PWM) signals to control the motor driver. These PWM signals dictate the motor's speed and direction and are derived from the outputs of the FOC controller module.
- **ADCs (Analog-to-Digital Converters)**: ADCs measure the current and voltage of the motor. These measurements are fed back to the FOC controller module for closed-loop control.
- **Flash Driver**: This module handles the storage of configuration parameters. It ensures that the system's settings are preserved even after power cycling or system resets, providing a convenient and persistent solution for storing critical data.

1.3 Board Connections

The connections of the hardware modules implementing the motor control system are illustrated in Figure 3.

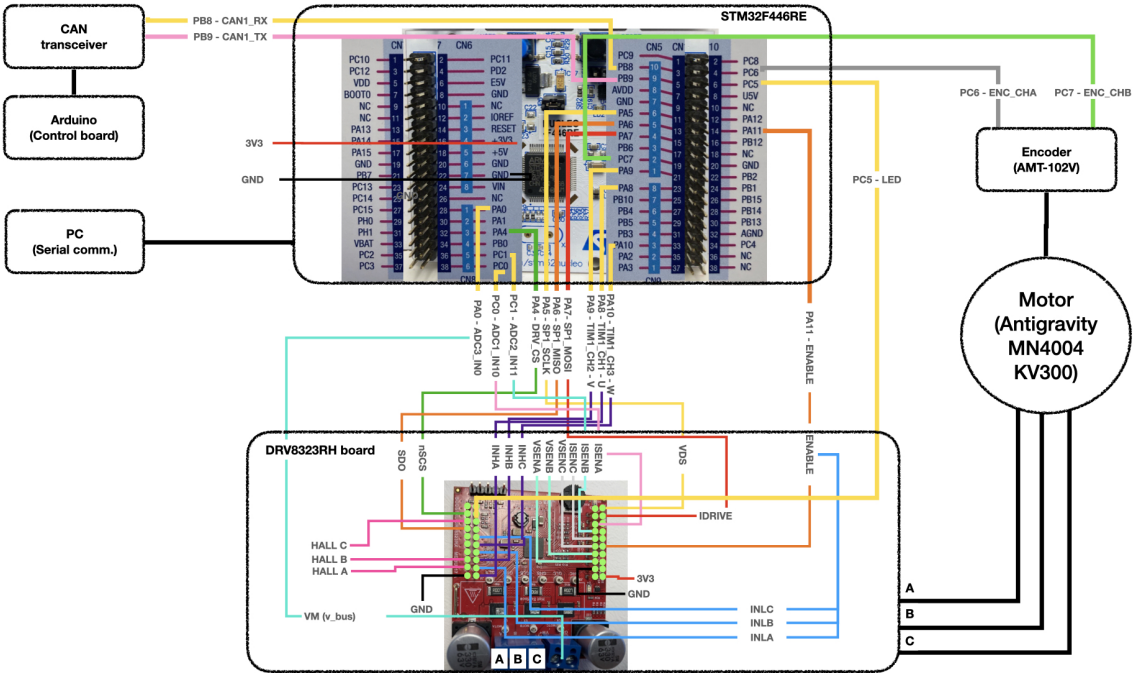

Figure 3: Hardware connections.

The connections for the motor driver were double-checked with both the schematic for the Mini Cheetah (see [4]) and the motor driver board (see [7]).

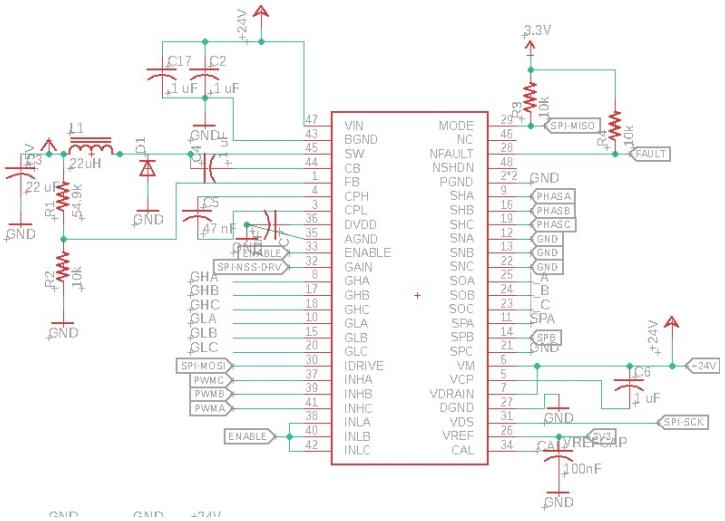

Figure 4: Mini Cheetah schematic - Motor driver [4]

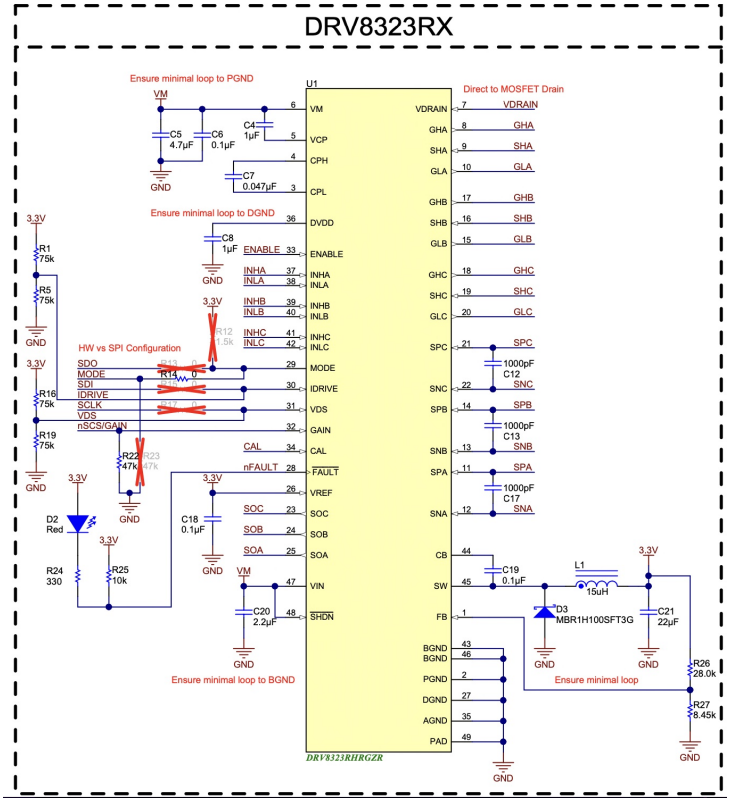

Figure 5: Motor driver board schematic [7]

1.4 System Critical Settings

For the system to work properly, the settings in the table below are very important. These settings ensure that the motor does not work above its limits operating conditions, that the encoder could correctly read the motor position, and that the CAN interface works properly. These settings can be set through the setup mode through the serial interface.

Setting	**Value**
Motor - "Antigravity MN4004 KV300 [10]" - settings:	
GR	1
I_MAX	9A
Torque Limit	0.27 N*m
KT	Torque Limit / (I_MAX * GR) = 0.03
V_SCALE	Voltage / ADC count = 24V / 4095 = 0.00586081
I_SCALE	Current / ADC count = 4.1411 A / 2056 = 0.00201416
Encoder - "AMT-102V [5]" - settings:	
CPR (counts per revolution)	8196
Communication settings:	
CAN Communication Speed	500kbps
Serial Communication	9600 baud, 8 bits, 1 stop bit, no parity bits

Table 1: Critical system settings

1.5 Documentation

Our project builds upon Ben Katz's (MIT) codebase for STM32 boards [2], incorporating significant changes and adaptations. The entire code, along with our enhancements, is available on GitLab `https://gitlab.lrz.de/dodo/motor_control`.

We have successfully implemented the code on the STM32F446RE board, which was also used by Ben Katz. Moreover the CAD data can be accessed for seamless hardware integration. Additionally, we offer the raw data for all our diagrams used in this documentation and Arduino code for the external CAN control described in chapter 4], increasing the project's versatility and potential applications. Feel free to explore the repository and contribute to this exciting project.

2 System Operation and Configuration

The system operation and configuration is described in detail at [3]. We present a summary below:

2.1 System Configuration and Interaction over Serial Interface

The serial interface allows configuration and operation as follows:

1. **Serial Terminal Configuration:** Set up a serial terminal on the computer with the following settings: 9600 baud, 8 bits, 1 stop bit, and no parity bits.
2. **Power-On and Debug Information:** Upon powering on the motor driver, it will print some debug information to the serial terminal.
3. **Menu Navigation:** The driver will display a menu of options on the serial terminal, which can be navigated by typing characters corresponding to the desired mode.

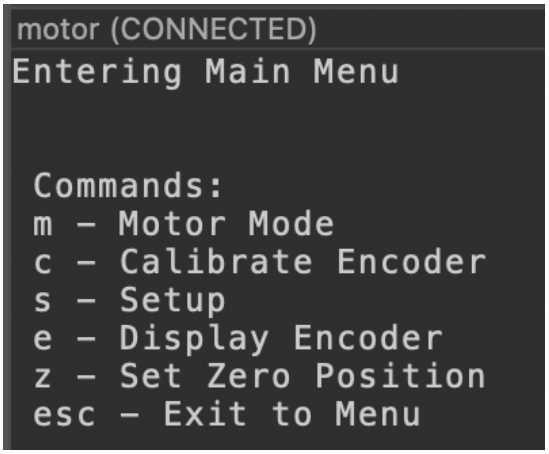

Figure 6: Main menu.

4. **Modes Available:**
   - `m` - Motor Mode: In this mode, the motor listens to position, velocity, and torque commands over CAN (Controller Area Network) and responds with actual position, velocity, and current.
   - `c` - Calibrate Encoder: This mode calibrates the position sensor on the rotor, compensating for non-linearity in the position sensor output. Ensure this calibration is run once the motor drive is attached to the motor and repeat it if you re-mount the motor drive. During the Calibration the number of pole pairs, the phase order, a lookup table for the encoder and the zero-position are determinded.
   - `s` - Setup: This menu allows you to change configuration settings, including:
     - Current control bandwidth: Higher bandwidth means faster torque response, but might cause some audible noise. Set it slightly higher than the sample rate of the device controlling the motor, typically around 1000 Hz.

```
Commands:
m - Motor Mode
c - Calibrate Encoder
s - Setup
e - Display Encoder
z - Set Zero Position
esc - Exit to Menu
Entering Setup

Configuration Options
prefix parameter                        min   max    current value

Motor:
g    Gear Ratio                         0     -      1.000
k    Torque Constant (N-m/A)            0     -      0.03000

Control:
b    Current Bandwidth (Hz)             100   2000   1000.000
l    Current Limit (A)                  0.0   5.0    4.000
p    Max Position Setpoint (rad)        -     -      12.500
v    Max Velocity Setpoint (rad)/s      -     -      65.000
x    Max Position Gain (N-m/rad)        0.0   1000.0 500.000
d    Max Velocity Gain (N-m/rad/s)      0.0   5.0    5.000
f    FW Current Limit (A)               0.0   4.0    3.000
c    Continuous Current (A)             0.0   4.0    4.000
a    Calibration Current (A)            0.0   20.0   5.000

CAN:
i    CAN ID                             0     127    1
m    CAN TX ID                          0     127    0
t    CAN Timeout (cycles)(0 = none)     0     100000 1000

To change a value, type 'prefix''value''ENTER'
e.g. 'b1000''ENTER'
VALUES NOT ACTIVE UNTIL POWER CYCLE!
```

Figure 7: Setup menu.

  - CAN ID: The ID number on the CAN bus that the motor will listen to. Each motor on the bus should have a unique CAN ID.
  - CAN Master: The ID number attached to output CAN messages. It should match the ID of your higher-level controller.
  - Current Limit: Maximum peak phase current in Amperes.
  - FW Current Limit: Maximum field weakening current (default is zero). Enables higher-speed operation than usually possible for a given drive voltage. Must be less than the motor's continuous current rating for safe continuous operation at high speeds.
  - CAN Timeout: After this many loop cycles (40 kHz loop), the torque command will become 0 for safety, e.g., if a wire comes unplugged.
- e - Display Encoder: Prints out the mechanical angle of the motor in radians and the raw encoder count.
- z - Zero: Sets the mechanical zero position to the current encoder position.

5. **Power Cycle Between Settings and Motor Control:** After changing settings, it's essential to power cycle the motor drive before running the motor. Some internal settings are calculated based on the user-configurable parameters, and this happens only on power-on.

2.2 Configuration over CAN Interface

In Motor Mode, the driver utilizes specific packet structures to send and receive data over the CAN interface.

CAN Bus Configuration: The CAN bus runs at a speed of 500kbps.

Command Packet Structure: The driver combines five commands into one packet:

- 16-bit position command (scaled between P_MIN and P_MAX).
- 12-bit velocity command (scaled between V_MIN and V_MAX).
- 12-bit Kp (Proportional Gain).
- 12-bit Kd (Derivative Gain).
- 12-bit Feed-Forward Current.

These commands are packed into an 8-byte data field within the CAN packet to be sent to the motor drive.

Response Data Packet Structure: In response, the motor drive sends back the following data in a 6-byte structure:

- 8-bit motor ID.
- 16-bit position (scaled between P_MIN and P_MAX).
- 12-bit velocity (scaled between 0 and 4095, with V_MIN and V_MAX).
- 12-bit current (scaled between -40 and 40 Amps, corresponding to peak phase current).

These response data values are packed into a 6-byte data field within the CAN packet sent back from the motor drive.

Special Commands: The following are special commands that can be sent over the CAN interface to perform specific actions:

- Enter Motor Mode: `[0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC]`
- Exit Motor Mode: `[0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFD]`
- Zero Position Sensor: Sets the mechanical position to zero. `[0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE]`

Ensure the correct commands are sent over the CAN interface to configure and control the motor drive effectively within the system. The motor drive receives 8 bytes of commands within the data field of the CAN packet and sends back 6 bytes of data in its response packets.

3 System Design and Implementation

As depicted in Figure 2, the primary components of the system include the Field-Oriented Control (FOC), Finite State Machine (FSM), motor driver, position sensor, low-level components such as the ADCs and PWMs, and communication interfaces like the serial and CAN. However, to initialize the system components and initiate periodic tasks (every 25us), two crucial modules are employed: the main module (`main.c`) and the event handling module (`stm32f4xx_it.c`).

3.1 The Main Module

The `main.c` file initializes and configures various hardware peripherals and software components before entering an infinite loop where the system operates. The initialization process involves:

- Configuring the system clock and initializing all configured peripherals, including GPIO, USART, TIMERs, CAN, SPIs, and ADCs.
- Initializing the controller parameters, which manage the motor's operation.
- Setting up the commutation encoder.
- Starting the ADCs.
- Setting up the DRV8323 motor driver.
- Starting the PWM.
- Setting up the CAN interface.
- Setting and enabling the interrupt priorities, which handle events requiring immediate attention, such as a change in the motor's position, the sampling of the motor current, and the processing of a CAN or serial message.

Following the initialization process, the system enters an infinite loop for operation. This loop primarily prints the faults from the DRV8323 motor driver every 100ms. All other operations are managed through interrupts as programmed in the `stm32f4xx_it.c` file.

3.2 The Event Handler Module

The `stm32f4xx_it.c` file is responsible for handling three critical interrupts essential for the system's operation. They manage communication over the CAN and Serial interfaces and drive the main control loop of the system.

- **The CAN interrupt:** This interrupt is triggered when a message is received on the CAN bus. The handler for this interrupt reads the incoming message, processes it, and sends a response back. It also checks for special commands within the received data and updates the Finite State Machine accordingly. If a special command is detected, it triggers a state change in the FSM, such as entering or exiting motor mode.

- **The Serial interrupt:** This interrupt is triggered when there is activity on the serial communication interface. The handler for this interrupt reads the incoming command from the serial interface and uses it to update the FSM.
- **The Timer interrupt:** This interrupt is triggered by the a Timer update event every 25us. The handler for this interrupt performs several critical tasks in each interrupt cycle. It samples the Analog-to-Digital Converters, samples the position sensor, runs the FSM, and checks for CAN messages. This interrupt essentially drives the main control loop of the system, ensuring that sensor data is regularly sampled, control decisions are made, and responses are sent over the CAN bus.

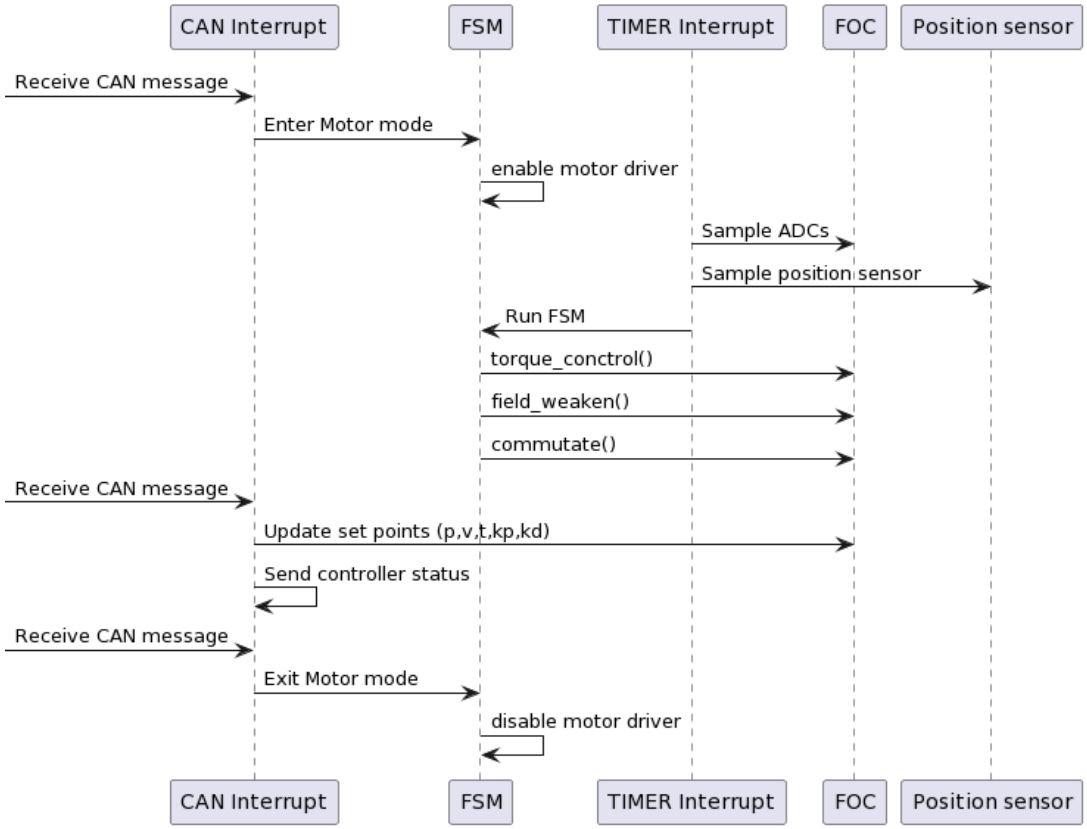

Figure 8: System flow during motor mode.

Figure 8 illustrates the system flow as three consecutive CAN messages are received. The first message puts the system into motor mode. The subsequent message sets the reference values for position, velocity, and torque for the controller. Following this, the FOC operates to achieve these set points. Eventually, a message is received to exit the motor mode. It's important to note that the TIMER interrupt ensures frequent ADC sampling and periodic motor position checks by the controller.

3.3 The Finite State Machine (FSM) Module

The FSM (fsm.c) is a critical component in the motor control system. It manages different states and transitions between them based on input signals (e.g., serial input, CAN input, and other triggers). Each state triggers specific actions upon entry and

exit, which perform necessary setup or cleanup operations associated with that state. Figure 9 shows the different states and transitions. Those states are: MENU_MODE, SETUP_MODE, MOTOR_MODE, ENCODER_MODE, and CALIBRATION_MODE. For more details about the those modes, see section 2.1 (System Configuration and Interaction over Serial Interface)

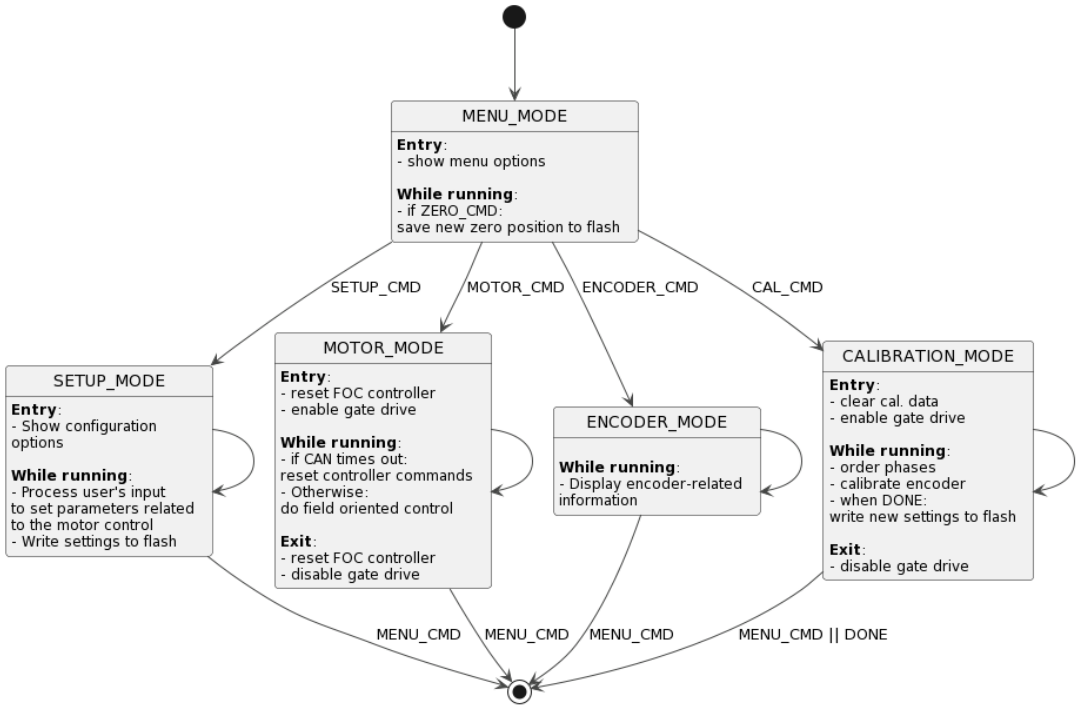

Figure 9: FSM.

The tables below summarize which parameters are stored in flash during SETUP_MODE and which parameters stored in flash during CALIBRATION_MODE.

Parameter	Description
E_ZERO	The zero position of the encoder
ENCODER_LUT	The lookup table for the encoder
PHASE_ORDER	The correct phase order
PPAIRS	Number of motor pole-pairs

Table 2: Parameters written to Flash in CALIBRATION_MODE

Parameter	Description
GR	The gear ratio of the motor
KT	The torque constant of the motor, in N-m/A
I_BW	The current bandwidth, in Hz
I_MAX	The maximum current, in A
P_MAX	The maximum position setpoint, in rad
V_MAX	The maximum velocity setpoint, in rad/s
KP_MAX	The maximum position gain, in N-m/rad
KD_MAX	The maximum velocity gain, in N-m/rad/s
I_FW_MAX	The field weakening current limit, in A
I_MAX_CONT	The maximum continuous current, in A
I_CAL	The current used during calibration, in A
CAN_ID	The CAN ID of the device
CAN_MASTER	The CAN TX ID of the device
CAN_TIMEOUT	The CAN timeout, in cycles

Table 3: Parameters written to Flash in SETUP_MODE

3.4 Field-Oriented Control (FOC) Module

The FOC module (foc.c) is the most important module in the system. It implements the FOC algorithm as shown in figure 10. During motor mode operation, the FOC works as follows:

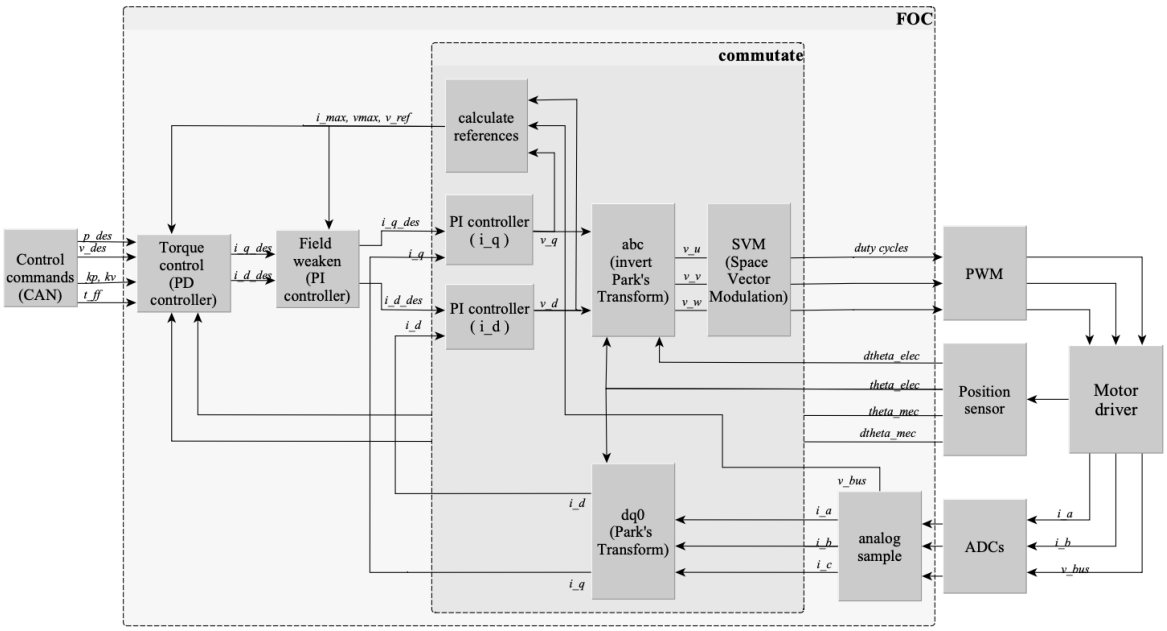

Figure 10: FOC.

1. **torque_control()**: this function is responsible for controlling the torque of the motor. It calculates the desired torque (t_{des}) based on the desired position (p_{des}), the mechanical angle ($theta_{mech}$), the feedforward torque (t_{ff}), the desired velocity (v_{des}), and the mechanical velocity ($dtheta_{mech}$). The desired torque is then used to calculate the desired q-axis current (i_{q_des}) and the desired d-axis current (i_{d_des}). The control law can be represented as follows:

$$t_{des} = K_p \cdot (p_{des} - theta_{mech}) + t_{ff} + K_d \cdot (v_{des} - dtheta_{mech}) \quad (1)$$

$$i_{d\_des} = 0 \quad (2)$$

$$i_{q\_des} = \frac{t_{des}}{KT \cdot GR} \quad \text{(KT: torque constant ; GR: gear ratio)} \quad (3)$$

2. **field_weaken()**: this function uses a PI controller to adjust the desired currents i_{q_des} and i_{d_des} based on the maximum voltages and currents, the reference voltage computed during commutation, and the field weakening integral term fw_int. The adjustments are as follows:

$$fw\_int = K_{i\_fw} \cdot (v_{max} - 1.0 - v_{ref}) \quad (4)$$

$$i_{d\_des} = \text{fw\_int} \quad (5)$$

$$i_{q\_des} = i_{q\_des} + (i_{q\_des} > 0) \cdot \text{fw\_int} + (i_{q\_des} < 0) \cdot \text{fw\_int} \quad (6)$$

3. **commutate()**: this function implements the main part of the FOC algorithm. It transforms the phase currents into the d-q frame, calculates the error between

the desired and actual d-axis and q-axis currents, and then uses two PI controllers to calculate the desired d-axis and q-axis voltages (v_q and v_d). It then transforms these voltages back into the a-b-c frame and applies them to the motor using space vector modulation. The equations (similar for d-axis and q-axis) are as follows:

$$\mathtt{i}_{\mathtt{d\_error}} = \mathtt{i}_{\mathtt{d\_des}} - \mathtt{i}_{\mathtt{d}} \tag{7}$$

$$v_d = K_d \cdot \mathtt{i}_{\mathtt{d\_error}} + d\_int \tag{8}$$

$$d\_int = d\_int + (k_d \cdot ki_d \cdot \mathtt{i}_{\mathtt{d\_error}}) \tag{9}$$

4. **analog_sample()**: This function reads the ADCs and converts the raw values into currents i_a, i_b and i_c as well as the bus voltage. The conversion is done as follows:

$$vbus = V\_SCALE \cdot adc\_vbus\_raw \tag{10}$$

$$i_a = I\_SCALE \cdot (adc\_a\_raw - adc\_a\_offset) \tag{11}$$

$$i_b = I\_SCALE \cdot (adc\_b\_raw - adc\_b\_offset) \tag{12}$$

$$i_c = -i_a - i_b \tag{13}$$

3.5 Position Sensor Module

The original position sensor was of the magnetic rotary type, as described in [6]. While it provided accurate position measurements, we needed to use a sensor that was simpler to integrate with our existing hardware and offered enhanced precision. This sensor was an incremental encoder AMT10. For a detailed overview of its features and benefits, refer to its specification [5]. To integrate the incremental encoder with our system, we

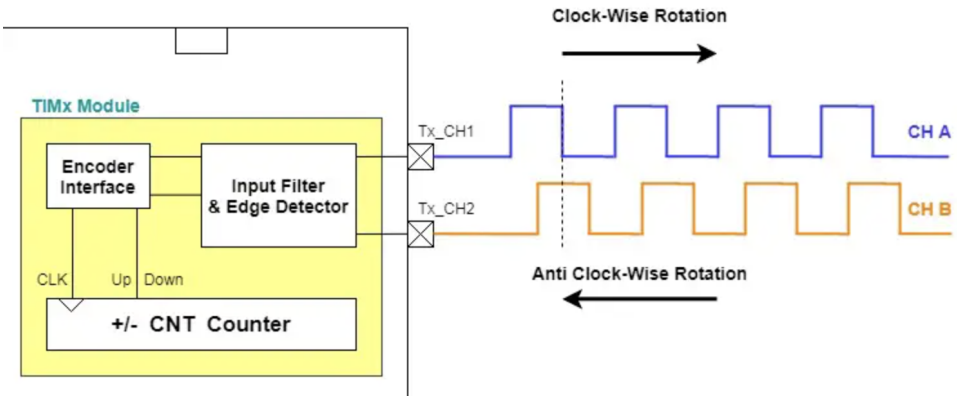

Figure 11: STM32 - timer as encoder mode.

employed the encoder-mode of an STM timer, which allows direct interfacing with the encoder's A and B signals, as detailed in [14] and depicted in Figure 11. The timer's inputs, TI1 and TI2, are clocked by transitions from these signals. The sequence of transitions sets the count direction and updates the DIR bit in the TIMx_CR1 register. In this mode, the counter continuously ranges between 0 and the auto-reload value in the TIMx_ARR register, adjusting automatically to the encoder's speed and direction. This reflects the encoder's position and aligns the count direction with the sensor's rotation, eliminating the need for extra interfacing components. The code snippet below illustrates the modifications required to integrate the new incremental encoder in position_sensor.c.

Listing 1: Code snippet for encoder initialization and reading

```
// In main.c
int main(void)
{
  ...
  MX_TIM3_Init();
  HAL_TIM_Encoder_Start(&TIM_ENCODER, TIM_CHANNEL_ALL);
  ...
}

// In position_sensor.c
void ps_get_count(EncoderStruct * encoder){
#if 0
        // SPI read/write (old encoder)
        encoder->raw = ps_spi_write(encoder, ENC_READ_WORD);

#else
        // New incremental encoder
        encoder->raw =__HAL_TIM_GET_COUNTER(&TIM_ENCODER);
```

```
#endif
}

void ps_sample(EncoderStruct * encoder, float dt){
    ...
    ps_get_count(encoder);
    ...
}
```

The core function in the position_sensor.c file is `ps_sample()`. Invoked periodically every `dt` time units, this function retrieves the current encoder count, applies linearization through a lookup table and interpolation, and computes both the mechanical and electrical positions and velocities. Additionally, `ps_sample()` oversees angle rollovers, ensuring angles are confined to the $[0, 2\pi]$ interval.

The relevant equations are:

- The mechanical angle:

$$\texttt{angle\_singleturn} = \frac{\texttt{count} - \textit{M\_ZERO}}{\texttt{ENC\_CPR}} \times 2\pi \tag{14}$$

$$\texttt{angle\_multiturn} = \texttt{angle\_singleturn} + 2\pi \times \texttt{turns} \tag{15}$$

- The electrical angle:

$$\texttt{elec\_angle} = \frac{\texttt{ppairs} \times (\texttt{count} - \textit{E\_ZERO})}{\texttt{ENC\_CPR}} \times 2\pi \tag{16}$$

- The mechanical velocity (rate of change of the multi-turn angle over time):

$$\texttt{velocity} = \frac{\texttt{angle\_multiturn}[0] - \texttt{angle\_multiturn}[\texttt{N\_POS\_SAMPLES} - 1]}{\texttt{dt} \times (\texttt{N\_POS\_SAMPLES} - 1)} \tag{17}$$

- The electrical velocity:

$$\texttt{elec\_velocity} = \texttt{ppairs} \times \texttt{velocity} \tag{18}$$

- Definitions:
  - `M_ZERO` and `E_ZERO` are zero positions, established during calibration.
  - `ENC_CPR` signifies the encoder's counts per revolution.
  - `ppairs` represents the motor's pole pairs.

3.6 Motor Driver Module

This module (drv8323.c) control the DRV8323 IC which is a three-phase gate driver for brushless DC (BLDC) motors. It provides the control signals to drive the power MOSFETs that control the motor's phases.

Some important functions are:

- Setup SPI interface for reading and writing registers
- Read fault status registers
- Configure control registers to set operating parameters (e.g. motor speed, torque, current)
- Enable or disable the gate drive that controls the motor's power stage. This allows for seamless motor startup, shutdown, and control, depending on the operational state.

3.7 CAN Communication Module

The CAN communication module (can.c), used for motor control communication, performs the following tasks:

- **Initialization:**
  - Initializes the interface with specific parameters like prescaler, synchronization jump width, and time segments.
  - Configure the receive message structure for incoming messages
- **Transmitting data about the motor controller's position, velocity, and current:**
  - Initializes the transmit message structure with parameters like data length code (DLC), identifier type, remote transmission request (RTR) type, and recipient ID.
  - Packs data into a reply packet by converting the data from floating-point format to the transmit data format.
- **Receiving data such as motor control commands and control parameters (position, velocity, proportional gain, derivative gain, and feed-forward torque):**
  - Unpacks received command packets by extracting the data.
  - Converts these values from the received format to floating-point for further processing.

For more information about the can messages see section 2.2 (Configuration over CAN Interface)

3.8 Serial Communication Module

This module (usart.c) provides functionality for the serial communication:

- **Initialization:** Initialize USART peripheral with specific settings, such as a baud rate of 9600, word length of 8 bits, stop bits of 1, no parity, no hardware flow control, and an oversampling ratio of 16. Enable the reception interrupt
- **Standard Library Function:** Implement the putchar function from the standard library. This function allows redirecting printf statements to the USART interface.

For more information about the can messages see section 2.1 (System Configuration and Interaction over Serial Interface)

3.9 Configuration Modules

The `preference_writer.c` module is responsible for writing configuration parameters into the Flash memory. It provides an interface for initializing, opening, checking readiness, writing integer and float values, flushing, loading, and closing the preference writer. The module uses a Flash writer to perform the actual writing operations.

The following table lists the parameters that can be written to Flash through calibration or setup mode. These parameters are defined in the `user_config.h` file.

Group	Parameter	Description
Motor Configuration	`R_NOMINAL`	Nominal motor resistance
	`PPAIRS`	Number of motor pole-pairs
	`KT`	Torque Constant
	`GR`	Gear ratio
	`I_CAL`	Calibration Current
	`M_ZERO`	Motor zero position
	`PHASE_ORDER`	The correct phase order
Encoder Configuration	`E_ZERO`	Encoder zero position
	`ENCODER_LUT`	Encoder offset LUT
Control Parameters	`I_BW`	Current loop bandwidth
	`I_MAX`	Current limit
	`THETA_MIN`, `THETA_MAX`	Minimum and maximum position setpoint
	`I_FW_MAX`	Maximum field weakening current
	`TEMP_MAX`	Temperature safety limit
	`I_MAX_CONT`	Continuous max current
	`P_MIN`, `P_MAX`	Position setpoint lower and upper limit
	`V_MIN`, `V_MAX`	Velocity setpoint lower and upper limit
	`KP_MAX`	Max position gain
	`KD_MAX`	Max velocity gain
CAN Parameters	`CAN_ID`	CAN bus ID
	`CAN_MASTER`	CAN bus "master" ID
	`CAN_TIMEOUT`	CAN bus timeout period

Table 4: User configuration parameters.

4 Motor control over CAN

As described in the previous sections, in motor mode the system can be controlled via CAN. To send the CAN commands, we used an Arduino Uno [11] together with a Sparkfun CAN-BUS Shield [12]. The CAN shield allows us to disable and enable the motor mode and increase or decrease the position during the latter using the integrated joystick. The code is based on a Youtube video of Skentific [13] and adapted to our hardware. He created the code by himself based on the can module of Ben Katz.

At the very beginning of the script the user can define the limits of the position, velocity, torque, and the proportional gains kp and kd. After the CAN initialization the main loop is entered. The following snippet shows this functionality:

Listing 2: Code snippet for CAN controller

```
void loop() {
  //step stize of position change
  float p_step = 0.01;

  if (digitalRead(UP)==LOW)
  {
    //move motor forward
    p_in = p_in + p_step;
  }

  if (digitalRead(DOWN)==LOW)
  {
    //move motor backward
    p_in = p_in - p_step;
  }

  p_in = constrain(p_in, P_MIN, P_MAX);
  if (digitalRead(RIGHT)==LOW)
  {
    // Enable
    EnterMotorMode();
    digitalWrite(LED2, HIGH);
  }

  if (digitalRead(LEFT)==LOW)
  {
    // Disable
    ExitMotorMode();
    digitalWrite(LED2, LOW);
  }

  // send CAN
  pack_cmd();

  // receive CAN
  if(CAN_MSGAVAIL == CAN.checkReceive())
  {
    unpack_reply();
  }
}
```

The detailed CAN structure which is used by the functions *pack_cmd()* and *pack_cmd()* can be found in chapter [2.2]. How to enter or exit the motor mode is described deeply in chapter [2.2] as well. The received CAN message including the position, velocity and currents will be printed to the console continuously.

5 Results and Conclusions

Achievements We have successfully established a hardware-software setup that can be utilized by other teams for ongoing development. It can be used as well as in the Dodo project and in all different types of electronic project's where a field oriented control of motors is suitable.

During the past months the following key milestones have been achieved:

- In total we accomplished to bring up the entire system including several hardware and software components. This includes the choice of the hardware, establishing all electrical connections and adapting the software accordingly. The code is based on Ben Katz from MIT. Moreover an entire documentation was created where all details about the software functionality and the exact hardware setup is explained.

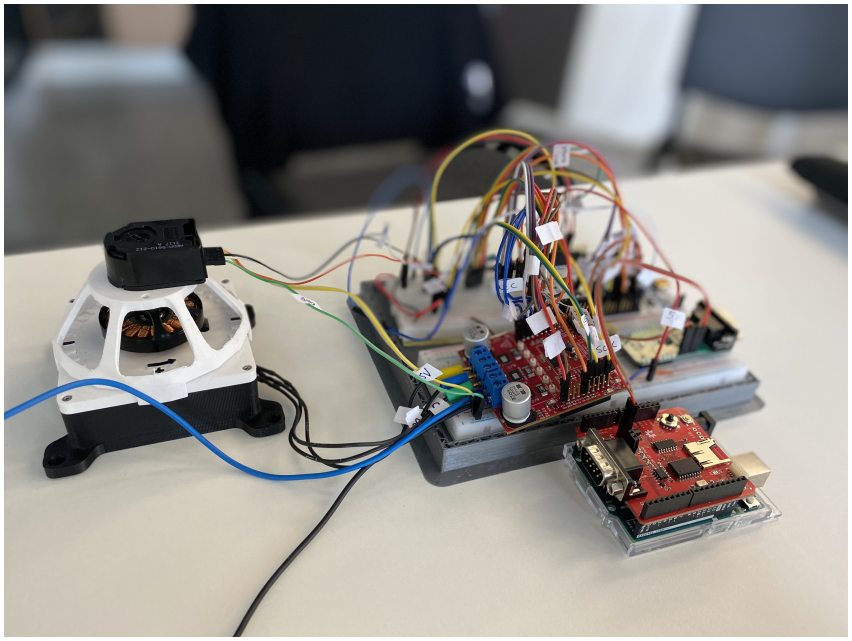

Figure 12: Entire hardware system including Encoder, Motor and CAN controller

Figure 12 shows the setup used by us. Additional images and videos of several system modes can be found on GitLab.

- We added an incremental encoder and fully integrated it to the system.
- It is possible to run the calibration mode of the FSM to it's full extend. This mode determines several parameters needed to run the motor.
- It is possible to run the encoder mode of the FSM to it's full extend. This mode displays the mechanical angel of the rotor.
- It is possible to run the motor mode of the FSM. The motor is able to turn the rotor with a constant velocity. Moreover during this mode the system is able to receive CAN messages for further control. However, be advised that a temporary workaround was implemented in `FSM.c`, as illustrated below. This workaround should be addressed and rectified in future work.

Listing 3: Code snippet for workaround in FSM.c

```
...
 void run_fsm(FSMStruct * fsmstate){
...
        case MOTOR_MODE:
        ...
#if 0
        // Original implementation (disabled for the demo)
        torque_control(&controller);
        field_weaken(&controller);
        commutate(&controller, &comm_encoder);
#else
        // Temporal demo
        float speed = 100.0f; // rad/s
        fsmstate->ref_position.elec_angle += speed*DT;
        controller.i_d_des = I_MAX;
        controller.i_q_des = 0.0f;
        commutate(&controller, &fsmstate->ref_position);
#endif
...
}
...
```

- To send the above mentioned CAN command's, an Arduino application was set up to control the system. This setup is able to send several target and control parameters and receive mechanical and electric values of the motor.
- A box for permanent and fixed storage was designed and 3D printed. As a result the entire control system can be used effortless as a ready-to-use setup.

Challenges During the past semester we were facing various challenges. The key difficulties will be mentioned in the following:

- Whereas Ben Katz provided the entire code of his control system, the enclosed sparse documentation was barely helpful to use the setup for an own purpose. That being the case we had to dig deeply into the code and study it subsequently resulting in a huge time consumption.
- Moreover the code need's many parameters to tune in and to be determined.
- In keeping with the previous point, we had sparse information and knowledge about the motor and other hardware components we were going to use. Meaningful parameters required for FOC control were not sufficiently available.

Future Work All in all the setup is ready-to-use and all modules are functional and successfully tested. In order to use the system in an electronic project, it is required to tune the control parameters in accordance to the specific motor. This ensures the correct calculation of the reference currents for the controller based on the desired position, velocity and torque.

References

[1] Wikipedia, *Vector Control (Motor)*, Available at: https://en.wikipedia.org/wiki/Vector_control_(motor)#

[2] GitHub, *Bgkatz's Motor Control Implementation*, Available at: https://github.com/bgkatz/motorcontrol

[3] Google Docs, *Bgkatz's Motor Control Documentation*, Available at: https://docs.google.com/document/d/1dzNVzblz6mqB3eZVEMyi2MtSngALHdgpTaDJIW_BpS4/edit

[4] GitHub, *MiniCheetah Schematic*, Available at: https://github.com/bgkatz/3phase_integrated/blob/master/New%20Version/Cheetah%20Driver%20Integrated%20V4/HKCV4.sch

[5] CUI Devices, *AMT10 Encoder datasheet*, Available at: https://www.cuidevices.com/product/resource/amt10.pdf

[6] AMS. *AS5147*. Available at: https://ams.com/en/as5147#tab/description.

[7] Texas Instruments, *BOOSTXL-DRV8323Rx EVM User's Guide*, Available at: https://www.ti.com/lit/ug/slvub01c/slvub01c.pdf?ts=1687473945591&ref_url=https%253A%252F%252Fwww.ti.com%252Ftool%252FBOOSTXL-DRV8323RH

[8] STMicroelectronics, *STM32F446RE Datasheet*, Available at: https://www.st.com/resource/en/datasheet/stm32f446re.pdf

[9] STMicroelectronics, *NUCLEO-F446RE Data Brief*, Available at: https://www.st.com/resource/en/data_brief/nucleo-f446re.pdf

[10] T-Motor Store, *Antigravity MN4004 KV300 - 2PCS/SET*, Available at: https://store.tmotor.com/goods-438-Antigravity+MN4004+KV300+-+2PCSSET.html

[11] Arduino, *Uno Datasheet*, Available at: https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf

[12] Sparkfun, *CAN-BUS Shield Datasheet*, Available at: https://www.sparkfun.com/datasheets/DevTools/Arduino/MCP2515.pdf

[13] Skyentific, *How to build MIT Mini Cheetah Controller*, Available at: https://www.youtube.com/watch?v=WKRLlthr9kY

[14] DeepBlueMbedded, *STM32 Timer in Encoder Mode*, Available at: https://deepbluembedded.com/stm32-timer-encoder-mode-stm32-rotary-encoder-interfacing/.