NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTK4900 - Master project

Retrofitting a legacy robotic arm using open-source solutions

Author:
Kristian Blom

07.06.24

Table of Contents

List of Figures

List of Tables

1 Preface

2 Introduction

2.1 Purpose

This report describes the master thesis project conducted by Kristian Blom during the spring semester of 2024. The project builds directly on the specialisation project concluded the previous semester of autumn 2023. This report contains relevant excerpts from the specialisation project report.

2.2 Background and motivation

From the initial project description: *This project is defined and commissioned by a student in co-operation with Omega Verksted (OV). It builds on the specialisation project named "From hardware to control algorithms: Retrofitting a legacy robot using open source solutions", in which new control hardware was developed for an old robotic arm. The arm is a 6DOF industrial arm originally developed for chemical analysis and consists of a 5DOF arm with a pincer/manipulator installed on a rail. The developed hardware controls 6 brushed DC motors and processes information from 6 incremental encoders, 6 current sensors, 3 accelerometer/gyroscope units, 2 optocouplers and 1 mechanical switch.*

This document is not just a thesis, but also intended for people with less experience, and the language generally tries to reflect that

2.3 Project scope

The primary goal of this project has been to develop a robotic software system compatible with hardware developed during the specialisation project, such that the robotic arm may be operated by a non-expert third party. The primary use case is the mixing of beverages during informal meetings at Omega Verksted. The secondary goal of the project is that the software system should be readily expandable to support more sophisticated use cases and sensors, such as 3D printing and robotic vision, by an expert third party.

Project scope as itemised list:

1. Implement and verify required hardware improvements from the specialisation project.
2. Write hardware drivers for the relevant microcontroller peripherals.
3. Develop and implement a software architecture around the primary and secondary goals of the project.
4. Develop and implement tests to evaluate the system's performance.

Project has three phases: Hardware improvement Software development Test and verif

3 Abbreviations and glossary

4 System specification

This section provides a high level description of the robotic system(4.1), a description of the hardware developed during the specialisation project(4.2), and a requirement specification for the software developed in this master project(4.3). The latter elaborates on scope items 2 and 3 from section 2.3.

4.1 Overview

The system consists of a robotic arm known from the original manufacturer, Beckman Coulter, as Optimized Robot for Chemical Analysis (ORCA); hardware developed as part of the specialisation project; and a general purpose desktop computer running the Linux/Ubuntu operating system. The arm is actuated by 6 DC motors, each connected to one joint via belts and gears within the arm. The arm is illustrated in figure 1, annotations marking each joint's positive direction. The gripper is mounted on a linear actuator, and is designed to bend around an object placed near the middle. It is capable of lifting and manipulating objects of approximately one kilogram.

The arm has several "hardpoints" on which PCBs or other equipment may be mounted, named after their physical location on the arm. The three control units developed in the specialisation project are mounted on the torso, shoulder and hand hardpoints, respectively. They are each responsible for the control of two joints, named after their function and/or physical location. For instance, the shoulder control unit is mounted on the shoulder hardpoint, and is responsible for controlling the elbow and wrist joints. A high level overview of the robotic system is presented in figure 2.

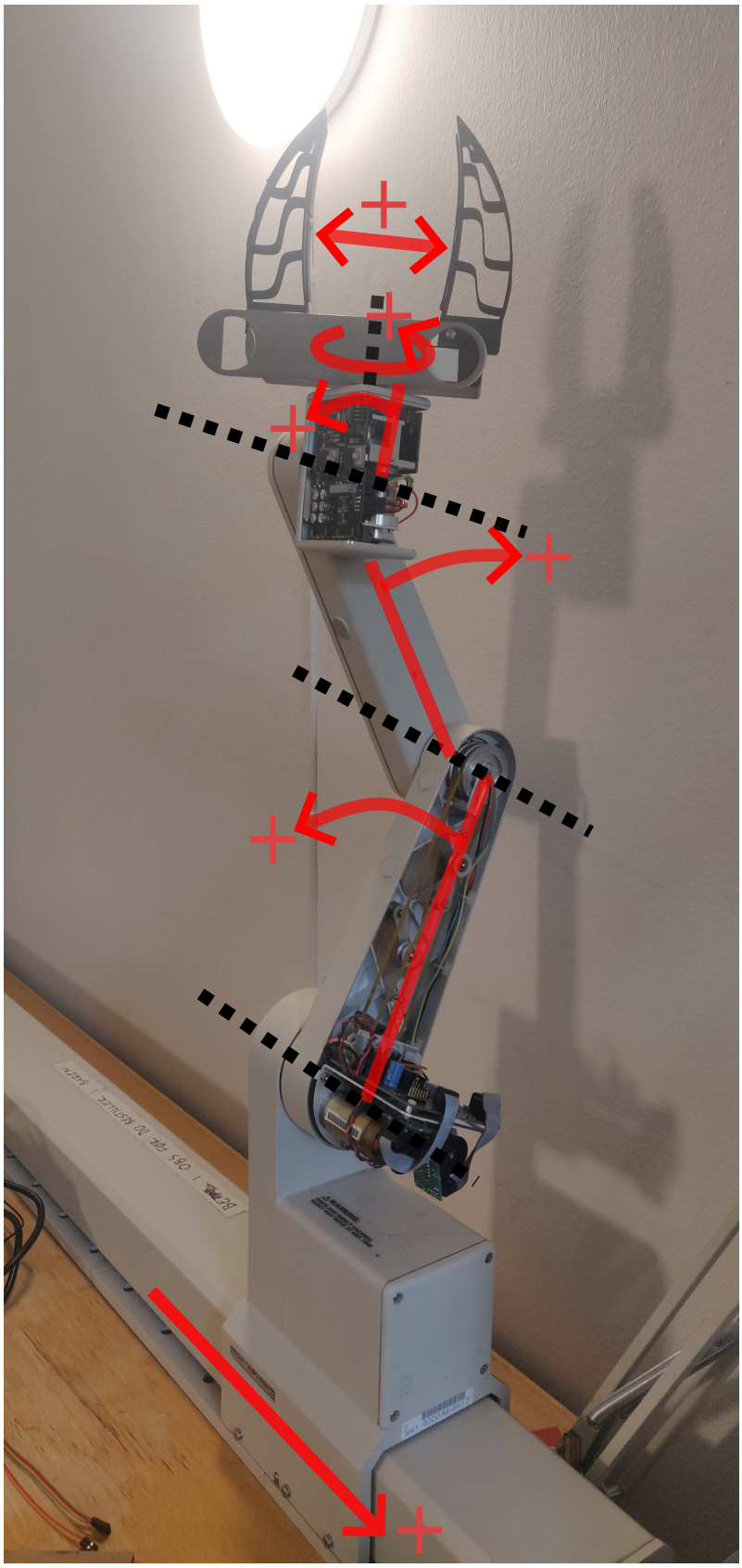


Figure 1: The ORCA arm, illustrated with joints and their directions. From the bottom up, joints are named rail, shuolder, elbow, wrist, twist and pinch.

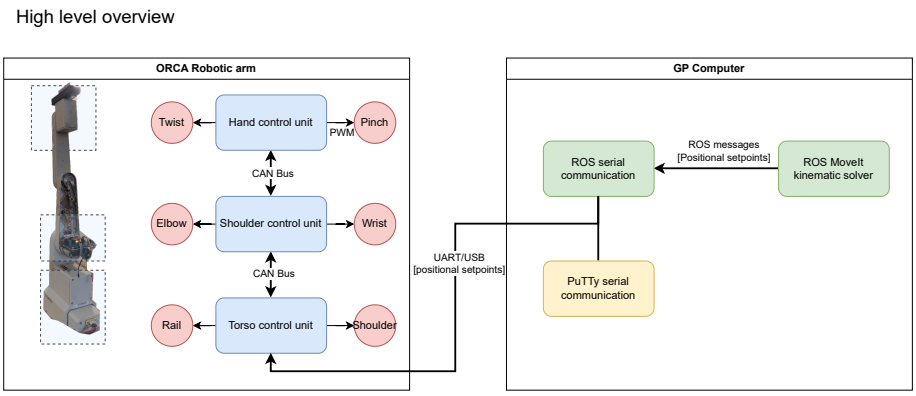


Figure 2: An overview of the robotic system

4.2 Hardware

As mentioned in section 2.2, all mechanical and some electromechanical parts were kept during the specialisation project in order to limit its scope. The parts are presented in table 1, originally presented in the specialisation project report. These parts were all found to be in working order, and replacing them would have required modification of the arm chassis.

For an in-depth discussion and presentation of the hardware, see sections 6 and 7 of the specialisation report. A key point from those sections is that there are space limitations in the arm preventing IMUs from being placed such that they may interface with their respective joint controllers directly. As a consequence, a critical function of the CAN bus is to enable transmission of IMU data between control units.

The following subsections explain each PCB developed in the specialisation project, and are summarised in figure 3. Figure 4 provides a detailed description of the arm, and was originally presented in the specialisation project report[2].

Table 1: Original parts kept in the project

Part name	**Description**	**Part no.**
Rail motor	Pittman 40mm 30.3V BDC motor	9233C59-R1[9]
Shoulder motor	Pittman 54mm 30.3VDC BDC motor	14205C389-R1[10]
Elbow motor	Pittman 40mm 24V BDC motor	9234C59-R1[9]
Wrist motor	Pittman 30mm 30.3V BDC motor	8224C143-R2[8]
Twist motor	Escap 23mm 23LT2R, 12V BDC motor	23LT2R[6]
Pinch motor	Escap 23mm 23LT2R, 12V BDC motor	23LT2R[6]
Wrist rotational sensor	TT Electronics Photologic Slotted Optical Switch	OPB971N51[5]
End switch	TT Electronics Photologic Slotted Optical Switch	OPB971N51
Motor encoder	Optical quadrature encoder 500CPR	HEDS-9100[18]

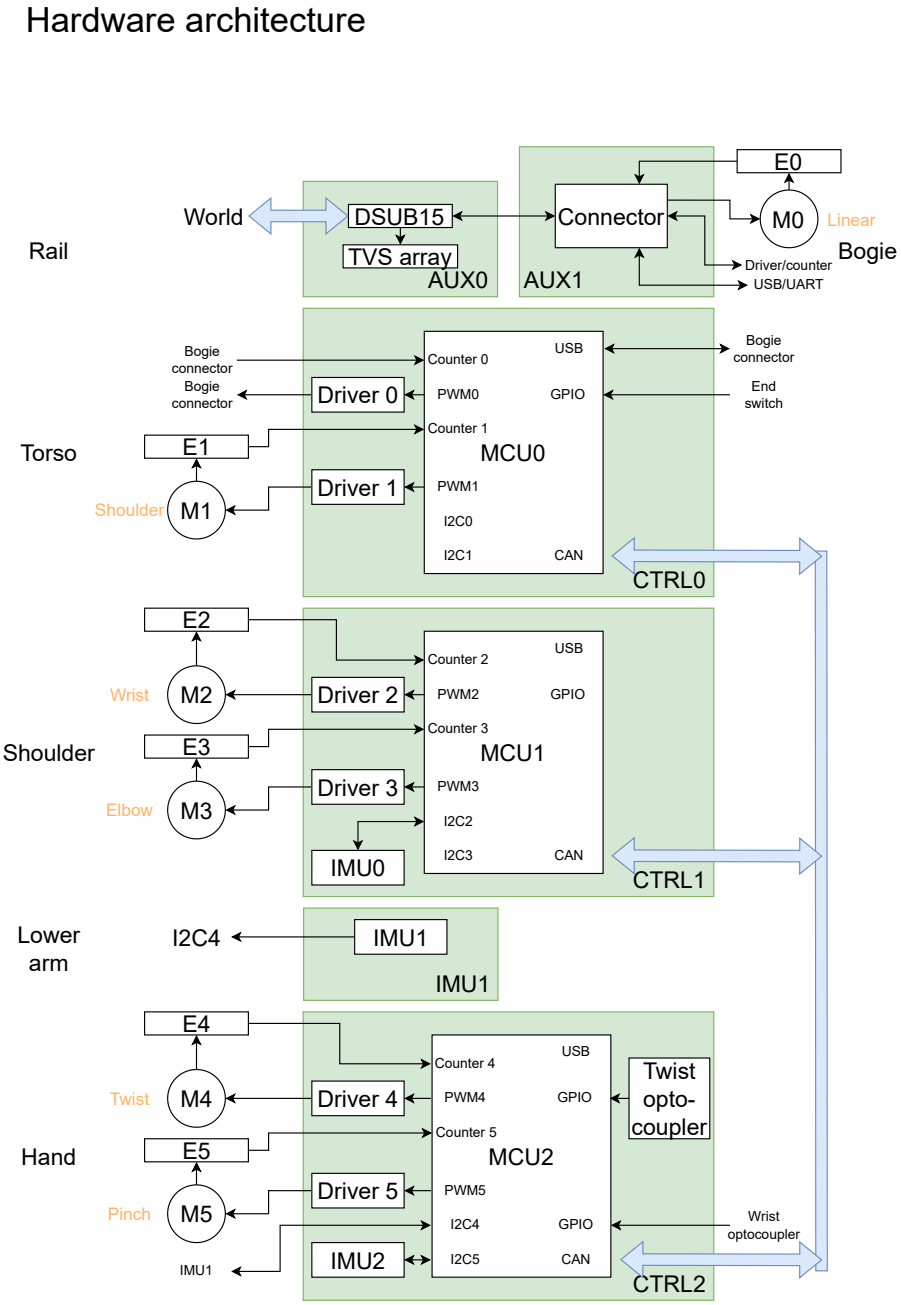
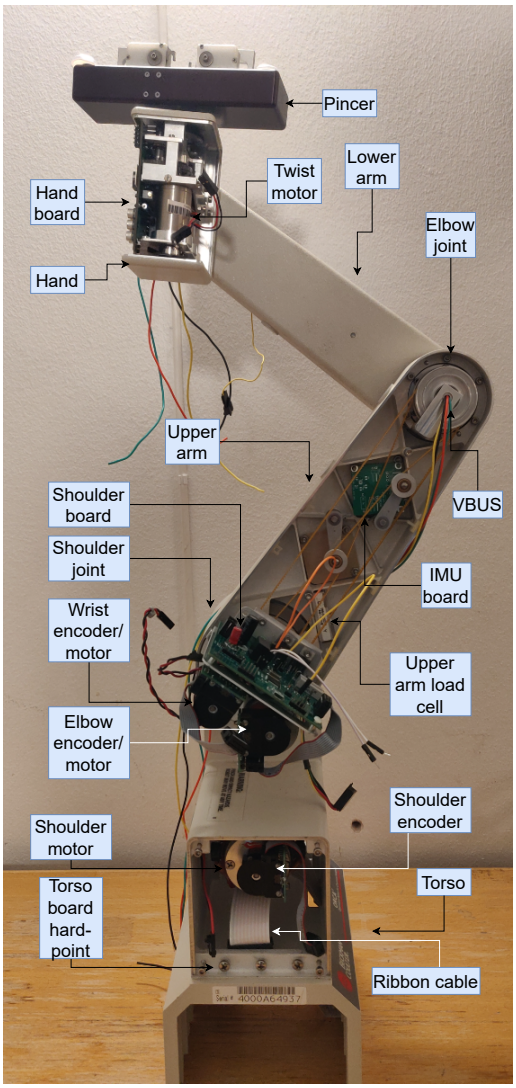
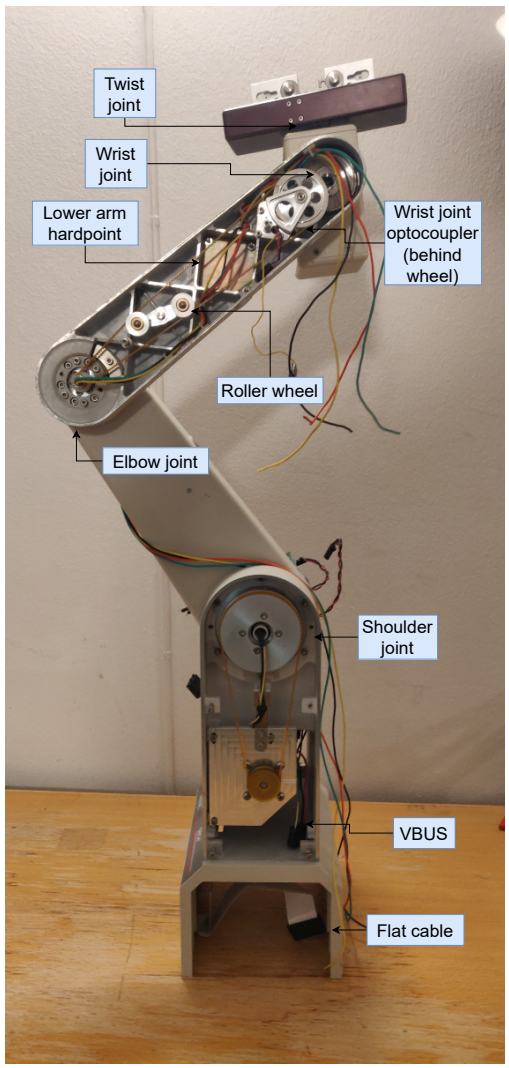


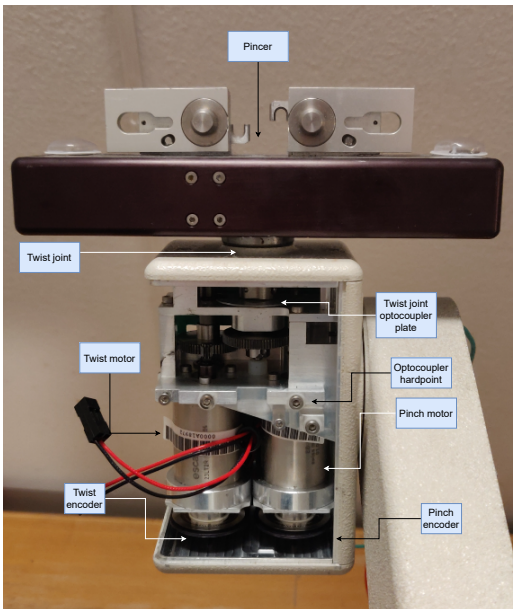
Figure 3: Hardware architecture. Green squares represent a PCB designed in the specialisation project, with key components as white squares. Left hand keywords indicate PCB mount location inside the arm. Originally presented in the specialisation report.



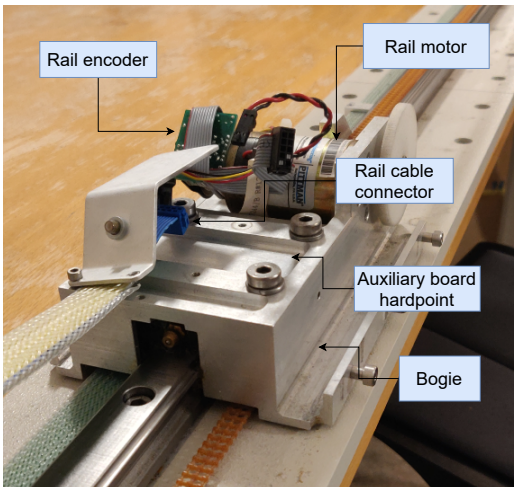
Arm viewed from the front.



Arm from the back.



A more detailed image of the hand and pincer.



The bogie on which the torso is mounted.

Figure 4: The arm with annotations. Orginally presented in the specialisation report

4.2.1 Auxiliary 0: Rail

The rail PCB is responsible for the arm's interface, a DSUB15 connector with signals for USB, UART and power. It is mounted on the rail hardpoint, and provides a socket for the 16 wire ribbon cable through which it interfaces with the bogie PCB(4.2.2). Additionally, it protects the USB and UART data lines from voltage spikes via its TVS array CDSC706[4]. The TVS array is clamped at 5V sourced from the torso control unit(4.2.3).

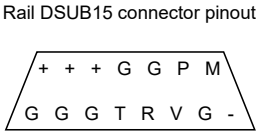


Figure 5: DSUB15 pinout

Table 2: Legend for figure 5, DSUB15 connector

Legend	**Meaning**
+	Input voltage
G	Ground
P	USB_DP
M	USB_DM
T	UART Tx
R	UART Rx
V	USB_VBUS
-	Not connected

4.2.2 Auxiliary 1: Bogie

The bogie PCB provides sockets for the 16 and 20 wire ribbon cables, as well as the rail motor and encoder connectors. Its purpose is to bundle the 16 wire ribbon cable with the rail motor and encoder wires into the 20 wire ribbon cable, through which it interfaces with the torso control unit(4.2.3).

4.2.3 Control unit 0: Torso

The torso control unit is responsible for the control of the rail and shoulder joints via motor 0 and motor 1 respectively, as well as external communication via the USB/UART lines. It interfaces with the upper arm IMU, IMU0 (4.2.4), via the shoulder control unit and the CAN bus. IMU0 is used in conjunction with E1 for the estimation of shoulder joint position. See table 3 for a presentation of the symbols in figure 3.

4.2.4 Control unit 1: Shoulder

The shoulder control unit is responsible for the control of the elbow and wrist joints via motors 2 and 3 respectively. It interfaces with the lower arm and hand IMUs, IMU1 and IMU2, via the hand control unit (4.2.6) and the CAN bus, for control of the elbow and wrist joints in conjunction with encoders 2 and 3, respectively. Additionally, it is responsible for direct communication with IMU0 via I2C. For remaining items, see table 3.

Table 3: Control unit 0

Symbol	Description	Unit name
Encoder 0 (E0)	Registers movement in the rail motor, used in estimation of the rail joint position. Relative, quadrature	HEDS-9100
Encoder 1 (E1)	Shoulder motor, see E0	HEDS-9100
End switch	Optical sensor, detects linear rail end position	See table 1
Driver 0	Motor driver, supplies power to the rail motor and provides current draw output. PWM control, analog current mirror	DRV8251A
Driver 1	Shoulder motor, see driver 0	DRV8251A
IMU0	Upper arm IMU	LSM6DSM
Microcontroller 0 (MCU 0)	Processing unit for control unit 0	STM32F303
Motor 0 (M0)	Linear rail motor, brushed DC	See table 1
Motor 1 (M1)	Shoulder joint motor	See table 1

4.2.5 IMU board 1: Lower arm

The IMU board serves as a mount point for the lower arm IMU, IMU1, which is used in the estimation of elbow joint position, and interfaces with the hand control unit via I2C. The IMU board also interfaces with the wrist optical sensor, relaying 5V from the hand control unit and sensor output to the hand control unit. IMU board 1 is mounted on the lower arm hardpoint.

4.2.6 Control unit 2: Hand

The hand control unit is responsible for the control of the twist and pinch joints via motors 4 and 5 respectively. It interfaces with the twist optical sensor and the wrist optical sensor via GPIO interrupts, and the CAN bus. Additionally, it is responsible for direct communication with IMUs 1 and 2 via I2C.

The twist optical sensor activates when the twist joint is in one specific position. As the twist joint has no hardpoints on which an IMU may be mounted, this sensor is essential to the estimation of twist joint position.

The wrist optical sensor activates for a set of joint positions, and may be used in the estimation of wrist joint position.

For remaining items, see table 3.

4.3 Software

The hardware presented in section 4.2 provides constraints for the software architecture. As mentioned, the software covers scope points 2 and 3 from section 2.3. This section elaborates on the functional requirements of the system, and presents a requirement specification which the finished software should fulfill.

Some keywords must be defined in the context of the requirement specification:

Must, shall: Denote a requirement which the failure to heed would significantly compromise the system's ability to achieve the project's two goals.

Should: Denotes a requirement which the failure to heed would only reduce the system's ability to achieve the project's two goals

4.3.1 Functional analysis

The primary goal for the system is to be able to "mix beverages" in informal settings, and be useable by non-expert personnel. This involves manipulating glasses, bottles and other objects that would typically fit in a human hand[1]. In order to achieve this, the system must implement the following functions:

- F1: Control the position and orientation of the pincer with an accuracy such that it may manipulate objects commonly involved with the mixing of beverages
- F1.1: Control the position of 6 joints concurrently
- F1.2: Take positional setpoints from a source external to the arm

"Informal settings" imply the presence of personnel and equipment which may not be accustomed to or intended for working with a robotic system during operation. Non-expert personnel refers to persons who may be familiar with robotic or autonomous systems in general, but do not have intimate knowledge of this system. In order to ensure safe operations in such a setting, the system should implement the following functions:

- F2: Operate in a predictable manner
- F2.1: Avoid fast or jerking motions
- F2.2: Follow predictable and/or intuitive movement patterns
- F3: Detect and respond when safe operational parameters are exceeded
- F4: Provide a user interface which requires limited preparation and/or education to make use of.

4.3.2 Architectural requirements

The secondary goal for the system is that it should be readily expandable to support use cases requiring a higher degree of movement accuracy, and/or more advanced sensors than currently exist within the hardware, by expert personnel. Expert pesonnel refers to persons who are experienced with robotic systems and embedded programming. This puts constraints on the development of the system's architecture:

- AR1: The system should consist of clearly defined modules
- AR1.1: A module's interface should be clearly defined
- AR1.2: A module should be limited in scope and purpose
- AR1.3: It should not be possible to pass information to a module outside of its interface.
- AR1.4: Naming conventions should apply across modules
- AR2: The system should adhere to common standards and best practices for embedded programming
- AR2.1: SOLID principles should be adhered to, to the extent that they apply
- AR3: Design patterns should apply across modules

[1] This is a postulate

4.3.3 Documentation requirements

The secondary goal, as well as this being a master thesis project, sets expectations for code documentation:

- DR1: All modules should be documented
- DR1.1: All functions, classes, structs et cetera should have unique documentation
- DR2: Documentation should be readily available
- DR3: Documentation conventions should apply across modules
- DR3.1: Language should be similar across modules

5 Theory

SOLID Coupling and cohesion, other relevant concepts CAN UART TVS arrays, voltage clamping Switching vs linear voltage regs Interrupts

5.1 Inter-integrated circuits (I2C)

5.2 Software development

5.3 Communication protocols

Message round trips vs bitrates

5.4 Circuit design

5.5 ADC voltage conversion

5.6 CAN bus

Also cover CAN filter setup, as promised in peripheral config

5.7 UART

5.8 Timer frequency

interrupt frequency, also counter period vs capture/compare (duty cycle) in pwm

5.9 SOLID principles

General considerations about the applicability of SOLID in a procedural language in an embedded system.

5.9.1 Single responsibility principle

A module should be responsible for one and only one thing; only have "one reason to change". Leads to a crapload of modules, but lowers risk of dependency problems. This is very inconvenient when using a language that doesn't have classes, in a system that is strongly oriented around hardware.

5.9.2 Open-closed principle

Modules should be open for extension, but closed for modification.

Avoid cascading changes when making a small change to a module by never changing modules; they are open to be expanded upon, i.e. getting child classes etc, but approved code should never be changed. Keywords: abstraction and polymorphism

5.9.3 Liskov substitution principle

If program P uses object type T successfully, then if object type S is a subtype of T, P should also be able to use S sucessfully.

Replacing an object with a child object should not break the program, and a base object should not need to know about child objects.

5.9.4 Interface segregation principle

No module should be forced to depend on methods it does not use.

5.9.5 Dependency inversion principle

Depend upon abstractions, not concretions. A consequence of OCP and LSP.

Modules that depend on concrete functionality become rigid; difficult to maintain and near impossible to reuse. A module should always depend on an abstraction, which may be made concrete in a specific use case.

6 Tools and workflow

This section describes the tools used in the project, for the purpose of reproduction and/or future development.

6.1 Software

6.1.1 Tools

KiCad 7, CERN distro[SOURCE] was used in the development of PCBs in the project's initial phase, without non-standard plugins. KiCad is an open source, freely available CAD software for the design of PCBs. When necessary, component footprints were downloaded from the UltraLibrarian[SOURCE] website when available, or otherwise drawn in KiCad based on component datasheets. These footprints may be found in FOLDER.

Git was used during all phases of the project, and the project's repository may be found at Github[SOURCE].

STMCubeMX[SOURCE] was used for the initialisation of microprocessor peripherals, i.e. the generation of peripheral drivers, and the generation of the project's Makefile. STMCubeMX is a configuration tool published and maintained by ST, the manufacturer of the STM microprocessor family, providing a GUI through which the user may create a pinout, enable interrupts et cetera for their microprocessor. It is freely available, but requires the registration of a user account with ST[SOURCE]. The tool generates .h and .c files for the relevant peripherals based on choices made in the GUI, as well as either a Makefile for use with development tools outside the ST software ecosystem, or the equivalent for use with ST's own STMCubeIDE IDE. This project utilised the Makefile option as it seeks to minimise the use of non-open-source solutions.

VSCode[SOURCE] with ST extension[SOURCE] was the primary IDE for this project. The extension *stm32-for-vscode* was used to build and flash binaries for the MCUs.

PuTTy was used for serial communication with and debugging of the MCUs.

ROS2 Iron

MoveIt

6.1.2 Workflow

MCU binaries:

- Set peripheral configuration parameters in STMCubeMX
- Ensuring that the "overwrite user code" option is unchecked, generate code.
- Edit generated files as necessary, ensuring code is added between USER CODE labels to avoid being overwritten the next time STMCubeMX is used to generate/update peripheral settings.
- Add and edit source files in the TTK4900_drivers folder under the root folder created by STMCubeMX.
- Ensure source files are registered in the Makefile generated by STMCubeMX.
- Build binary file using the ST extension in VSCode.
- Flash binary to MCU using the ST extension and ST-LINK unit embedded in the development kit6.2.

ROS nodes

- Do stuff

6.2 Electronic/Hardware

6.3 Mechanical

- KiCad
- Git
- ST CubeMX
- VSCode (extensions: ST, C/C++)
- Development kit
- Angular measurement tool
- Power source
- Oscilloscope
- MoveIt
- Putty
- Categories: Electr(on)ic, Software, Mechanical
- Fusion 360
- Prusa Slicer

7 Hardware design

This section discusses the hardware presented in section 4.2 in further detail. Hardware produced for this project is an iteration of hardware produced for the specialisation project[2], and is dubbed Mk1.1. The material presented overlaps with sections 7 and 8 of the specialisation project report[2], but summarises key improvements over Mk1. Relevant copper layers are presented as a visual reference for pin headers and connectors.

Additionally, this section discusses results from the verification of Mk1.1 hardware. As they lay the foundation for the master project they are not considered "results" as such, and will only be adressed to the extent to which they affected the project's software implementation in section 15.

7.1 Improvement matrix

Table 4 summarises the "Further work" section of the specialisation report and was originally presented there. Each entry in the left column represents a point of improvement, and a cross in a subsequent column indicates that the issue affects the hardware unit in the top row of that column. The matrix is presented here as it was a key tool in organising the design of Mk1.1 and summarises a significant part of the master project, but not all improvement points will be adressed here.

Table 4: A summary of further work

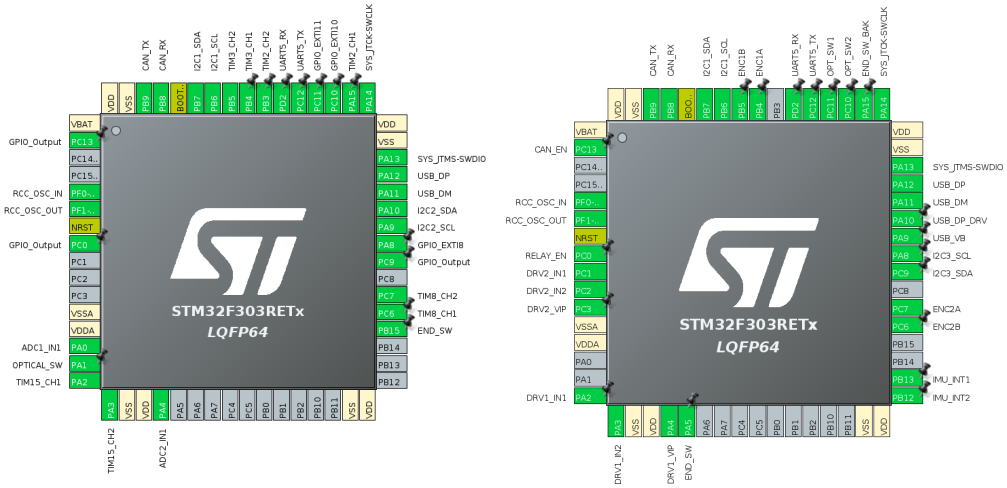
Item/affects unit	**Hand**	**Shoulder**	**Torso**	**Bogie**	**Rail**	**IMU (board)**	**MCU**
USB VBUS voltage divider		X	X				
USB VBUS pin		X	X				X
USB pullup on DM		X	X				
USB pullup transistor		X	X				
IMU 5V and 3.3V lines	X	X				X	
IMU/OPT header design	X	X				X	X
IMU header placement		X					
CAN verification	X	X	X				
CAN/48V connector placement			X				
CAN transciever placement	X						
Nylon bolts	X						
Mount point placement		X	X		X		
Mount point size		X	X				
20 pin connector			X	X			
Motor driver to PWM output	X	X	X				X
Motor driver ADC/PWM							X
Motor driver header silk	X						
Optocoupler 5V	X					X	X
Wrist optocoupler function	X					X	X
Bulk cap/pin header swap	X						
1.27mm jumpers	X						
Switching regulators	X	X	X				
Horisontal voltage regulators			X				
Voltage regulator LED resistor	X	X	X				
Encoder circuits							
Encoder header rotation				X			
TVS array					X		
Motor driver relay control		X	X				
Motor characterisation							
PCB production	X			X	X		

7.2 MCU pinout

The MCU pinout lays the foundation for PCB layout, and was changed first. The pinout is presented in figure 6.

- USB VBUS detection was set to pin PA9 in accordance with AN4879 chapter 2.6[12].
- PA10 was set to output as USB DP driver in accordance with AN4879 figure 5.
- PB12, PB13, PC10 and PC11 were opened as interrupt inputs to accommodate optical switches and IMU programmable interrupts, as well as ensuring optical sensor inputs are placed on 5V tolerant pins according to table 13 ofthe MCU datasheet[16].
- PWM output for motor driver 2 was moved to PC1 and PC2 from PC6 and PC7 in order to simplify PCB layout by gathering all motor driver outputs along one edge of the MCU.
- Current sense ADC input for motor driver 2 was moved from PA0 to PC3 in order to avoid an interference mode between peripherals, see elaboration below.

A key finding from the specialisation project was the discovery of an interference mode between the TIM2 (timer 2) and ADC1 peripherals[2], see chapter 13.3[2]. When TIM2 was configured for PWM generation, and ADC1 was activated on pin PA0, PA0 acted as an output pin with an analog voltage output proportional to the PWM duty cycle on TIM2. The cause of the interference mode could not be established beyond the fact that PA0 may also be configured for output from TIM2. The solution was to move ADC1 to pin PC3, which is not compatible with TIM2. Additionally, TIM2 was not used for PWM generation. For more information about timer configuration, see section 8.8.



MCU pinout configuration prior to hardware improvements

MCU pinout configuration after hardware improvements

Figure 6: Comparison of the old and new MCU pinout configuration

7.3 IMU board

As mentioned in section 4.2, the IMU boards act primarily as mounting points for the IMUs, and were to be mounted on hardpoints in the upper and lower arm sections, see figure 4. Secondarily, the lower arm IMU board would act as an interface with the wrist optical sensor. The LSM6DSM IMU has two configurable interrupt output pins[13] in addition to its I2C interface, and it was

[2]The report mentions pin PA4. This is erroneous; the interference was present on pin PA0

decided that at least one of these should be available for use in the system in the event that they prove relevant to the software implementation. The circuit is presented in figure 21.

The OPB971 optical sensor requires an input voltage of minimum 4.5V[5], while the LSM6DSM IMU has a maximum input voltage of 3.6V[13]. In order to save space in the wrist joint hole, it was decided that only the 5V line should be pulled from the hand control unit rather than both 3.3V and 5V lines. The ZMR330 step-down regulator, which outputs 3.3V for an input voltage above 4.8V[7], was intruduced as a replacement for a simple voltage divider in Mk1.

Figure 7 shows the improved IMU layout. Several headers/jumpers have been introduced to accommodate the various signal output options, and the PCB's interface header `I2C_PWR1` has been reduced from 8 to 6 pins compared to Mk1 (see chapter 9.8 of the report[2]). Jumper `INT_SEL1` lets the user select IMU interrupt 1 or 2 for output to the hand control unit. Jumper `INT_EN1` lets the user select whether or not to send the selected interrupt to output. The two jumpers implement the following boolean function:

$$I_{I2C\_PWR1} = (1_{INT\_SEL1} \oplus 2_{INT\_SEL1}) \cdot (I_{INT\_EN1} \oplus O_{INT\_EN1}) \tag{1}$$

If `INT_EN1` is set to O, the interrupt selected by `INT_SEL1` will be grounded. The other interrupt will be left floating.

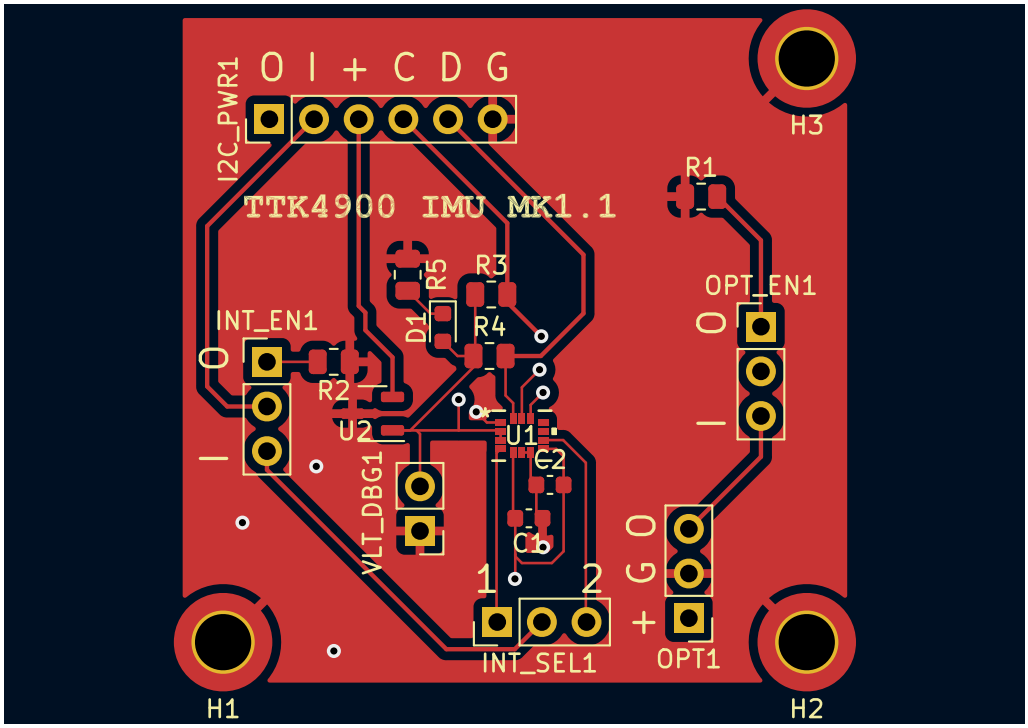


Figure 7: Front copper and silk layers of the improved IMU board

7.4 Rail

Rail board mount points were reevaluated for Mk1.1 with minor adjustments. The specialisation report suggests replacing the CDSC706 TVS array with one capable of protecting all 14 lines entering the arm. This was elected against due to uncerainty regarding correct clamping and the unprotected lines to some extent being protected by diodes on the control boards (see section 8.2.1 of the specialisation report). The circuit is presented in figure 15.

Table 6 explains the silk symbols on the 16 pin connector socket of the rail board as seen in figure 8, which shows the back copper and silk of the rail board. The silk should be interpreted as an

Table 5: IMU board header pin legend

I2C_PWR1	**Meaning**	**OPT_EN1**	**Meaning**
O	Optical output	I	Optical sensor output enable
I	IMU interrupt output	O	Optical sensor output disable
+	5V in	**OPT1**	**Meaning**
C	I2C Clock	+	Optical sensor 5V input
D	I2C Data	G	Optical sensor ground
G	Ground	O	Optical sensor output
INT_EN1	**Meaning**	**INT_SEL1**	**Meaning**
O	Disable IMU interrupt output	1	IMU interrupt 1 select
I	Enable IMU interrupt output	2	IMU interrupt 2 select
VLT_DBG1	**Meaning**	-	-
3.3V	output debug	-	-

overlay of the pins, such that the label "5" corresponds with connector pin 8. The 14 pin header on the upper right of figure 8 is connected to the DSUB15 connector presented in section 4.2, silk symbols explained in table 2. The circuit is presented in figure

Note: The input voltage pins were initially left partially unconnected due to a design error. This was corrected, and the rail version number was elevated to 1.2.

Table 6: Rail board 16 pin connector

16 pin connector	**Meaning**
G	Ground
P	USB_DP
M	USB_DM
V	USB_VBUS
R	UART Rx
T	UART Tx
5	TVS 5V clamping
+	Input voltage

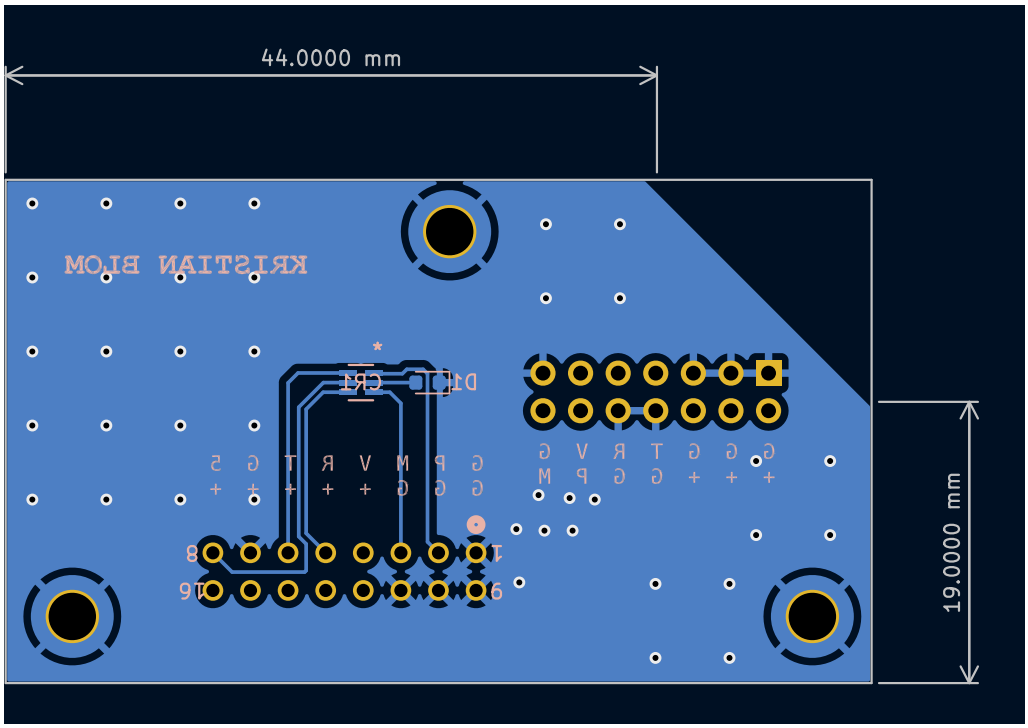


Figure 8: Back copper and silk layers of the improved rail board

7.5 Bogie

A key error in the design of the bogie board was the pin assignment of the 20 pin connector socket through which it interfaces with the torso control unit. It was discovered that one row of pins on the connector had been mirrored compared to the pin assigment on the torso board due to differing pin numbering schema in the circuit schematic.

The improved bogie board is presented in figure 9, and the silk symbols for the 20 pin connector socket is presented in table 7. The silk symbols for the 16 pin connector socket is presented in table 6. The circuit is presented in figure 16.

Note: Finding the cause of the connector issue took two attempts, and the bogie version number is therefore 1.2.

Table 7: Bogie board 20 pin connector socket, end switch connector header

MAIN1	**Meaning**	**END_SWITCH1**	**Meaning**
ES	End switch output	G	Ground
VB	USB_VBUS	E	End switch output
DM	USB_DM	+	5V input
DP	USB_DP		
5	5V output		
B	Encoder ch B		
A	Encoder ch A		
T	UART Tx		
R	UART Rx		
G	Ground		
+	Input voltage		
M1	Motor input 1		
M2	Motor input 2		

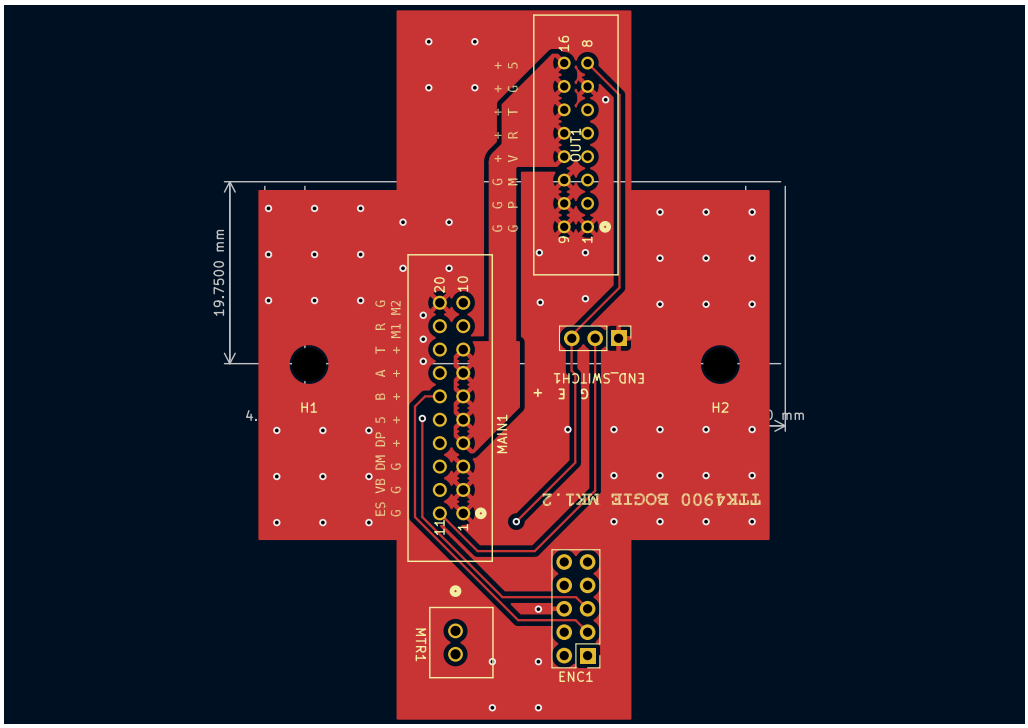


Figure 9: Front copper and silk layers of the improved bogie board

7.6 Torso

In addition to changes made uniquely to the torso, this section summarises improvements made to the various assemblies which make up the control units. Assemblies were originally introduced in section 8.2 of the specialisation report. Figure 11 presents the improved torso control unit. Legends for various pin headers are presented in 8. The torso circuit is presented in figure 17.

7.6.1 USB assembly

The USB assembly is present in the torso and shoulder control units. In the torso it is used for communication with the external control computer, as suggested in figure 3, and the data lines are pulled to a micro USB port on the board as well as the torso unit's 20 pin connector. In the shoulder it acts as a test/debug unit, and the data lines are pulled to a micro USB port which is covered when the arm is in normal operation. The USB assembly circuit is presented in figure 24.

The circuit was redesigned in accordance with chapter 2.6 and figure 5 of AN4879[12], see figure 10. R1 was set to $33k\Omega$, R2 to $82k\Omega$. The specialisation report suggests to add a transistor between PA10 (USB DP driver) and the DP line such that the MCU would not drive the line directly. However, this was deemed unnecessary on the basis that figure 5 and the following quote imply that the DP line may be driven directly via a resistor: "A DP pull-up must be connected only when VBUS is plugged. A GPIO from the MCU is used to drive it after the VBUS detection.[...]" - Chapter 3.1.1 of AN4879.

Figure 5. **USB FS upstream port without embedded pull-up resistor in self-powered applications**

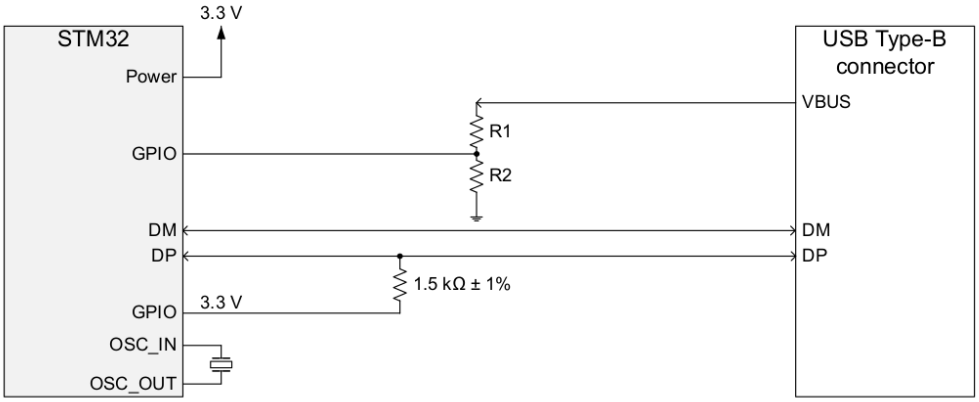


Figure 10: Figure 5 from ST AN4879, illustrating a valid USB circuit

Based on this, a $1.5k\Omega$ resistor was placed between PA10 and PA12. In order to fulfill the requirement of only connecting the DP pullup when VBUS is plugged, it was decided that an interrupt would be used to activate PA10 in software upon VBUS-detection, and that the pin would otherwise be held in its reset state.

Additionally, AN4879 chapter 2.3 recommends protecting data lines with a USBLC6 IC, which was also implemented.

7.6.2 CAN bus assembly

The CAN bus assembly is present in all three control units, and consists of the TJA1057BT[11] CAN transciever and its power supply components. The CAN_L/H lines are connected to the other CAN transcievers, while the CAN_Rx/Tx lines are connnected to the transciever's MCU. The circuit is presented in figure 23.

The CAN bus assembly was not satisfactorily verified during the specialisation project: *The CAN circuit was tested between the development kit and the MCU on the breadboard. While a correctly configured CAN message could be observed by oscilloscope on any point of the data lines, no indication was observed that the recipient MCU (breadboard) had registered the message.*(section 13.1.1)

The cause was likely that the wrong CAN transciever had been used in one of the test units as well as ordered for Mk1.1: TJA1057**G**T instead of TJA1057**B**T. The BT has a pin dedicated to measuring logic voltage levels other than the CAN standard of 5V, while the GT does not (TJA1057[11] chapter 6, pin 5).

7.6.3 Motor driver assembly

The motor driver assembly is present in all three control units, and consists of two DRV8251A motor drivers[19], motor bulk capacitors and, in the case of the torso and shoulder control units, a JV-3S-KT relay operated via a 2N551 NPN BJT transistor from the MCU's PC0 pin. The transistor and relay act as a dead man's switch for the motor drivers, ensuring that motors will lose power should the MCU go offline. The motor drivers are controlled from the MCU by PWM signals. A key function of the motor drivers is their proportional current output: pin 1 will output a current proportional to the current drawn by the motors (I_{PROPI}), which is measured as a voltage (V_{IPROPI}) across the current resistor (R_{IPROPI}). The assembly is presented in figure 25.

In addition to changing the choice of transistor, motor driver PWM signal lines were changed in

accordance with the update to MCU pinout described in section 7.2 for Mk1.1.

7.6.4 Voltage regulator assembly

The voltage regulator assembly is present in all three control units, and consists of two adjustable LM317HV linear voltage regulators[20] and their adjustment circuits. The voltage adjustment potmeters RVn are set such that the regulators step the system input voltage down to 3.3V and 5V, respectively, in order to supply relevant ICs. The LM317HV were found to rise to a substantial temperature at an input voltage of 20V during the specialisation project, and the target input voltage for the system is 48V[3]. The report therefore suggests replacing the LM317HV with an equivalent switching regulator, as switching regulators tend to develop less heat than linear regulators. However, due to the limited time available for implementation of Mk1.1, this was elected against. The assembly circuit is presented in figure 26.

PCB footprints of the regulators were changed to horisontal from vertical, in part to improve heat dissipation by using the PCB itself as a heat sink, and in part to make placement of the regulators independent of the mount points available on the external heat sink which the regulators were attached to in Mk1.

Additionally, the output voltage indicator LED resistor was increased from 200Ω to 1.5$k\Omega$ in order to dim the LEDs and reduce eye strain when working with the control units.

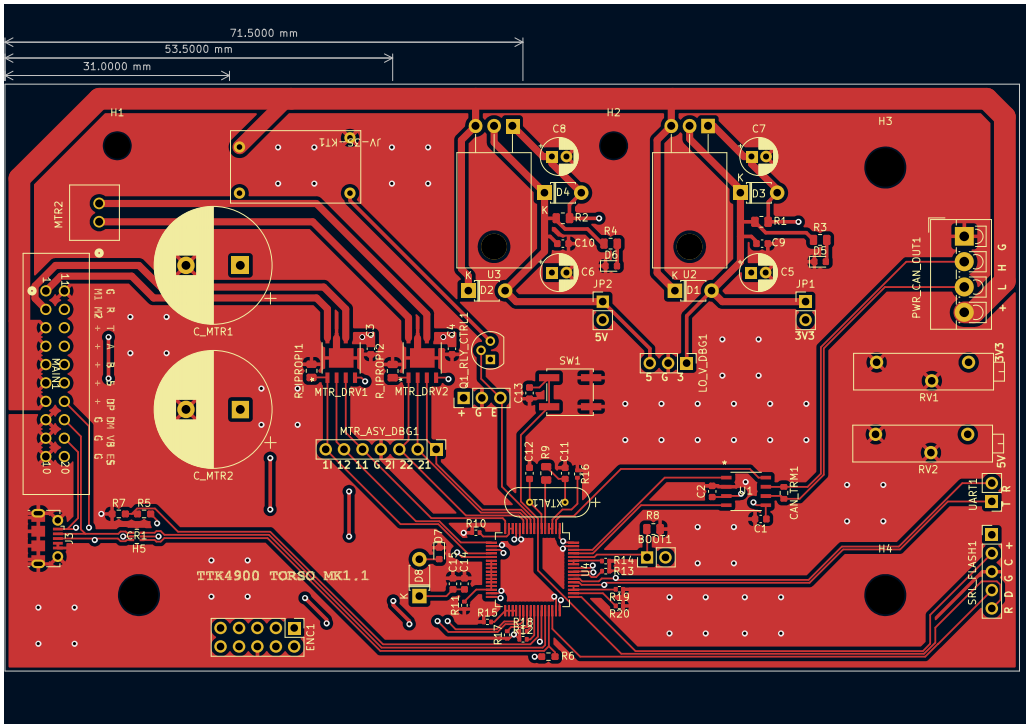


Figure 11: Front copper and silk layers of the improved torso board

7.7 Shoulder

As shown in table 4, changes made in the torso control unit largely apply to the shoulder control unit as well. The two units are almost identical with respects to which assemblies are present, with the IMU assembly being the only difference. The shoulder front copper layer is presented in figre 12, and the shoulder circuit is presented in figure 18.

[3]Investigation during the specialisation project suggested this was the system's original operating voltage.

MTR_ASY_DBG1	**Meaning**	**Q1_RLY_CTRL1**	**Meaning**
1I	DRV1 VIPROPI	+	3.3V
12	DRV1 PWM2	G	Ground
11	DRV1 PWM1	E	Relay enable
G	Ground	**LO_V_DBG1**	**Meaning**
2I	DRV2 VIPROPI	5	5V output
22	DRV2 PWM2	G	Ground
21	DRV2 PWM1	3	3.3V output
PWR_CAN_OUT1	**Meaning**	**SRL_FLASH1**[a]	**Meaning**
+	Sys voltage in	+	VDD Target
L	CANL line	C	Programmer clock
H	CANH line	G	Ground
G	Sys ground out	D	Programmer data
		R	Reset target

Table 8: Pin header legends for the torso control unit. [a]The flash header was designed for use with an ST-LINK programmer, and is described in chapter 6.2.4 of [17].

7.7.1 Mount points

Mount points were reevaluated, and fastening bolts were changed from metal to nylon to prevent damage to the board.

7.7.2 IMU assembly

The IMU assembly is present in the shoulder and hand control units, and consists of the LSM6DSM IMU[13] and its power supply circuit. A modified version is also present in the IMU board, and the circuit is presented in figure 22. The SDA and SCL lines are pulled to I2C port 3 of the hand and shoulder MCUs.

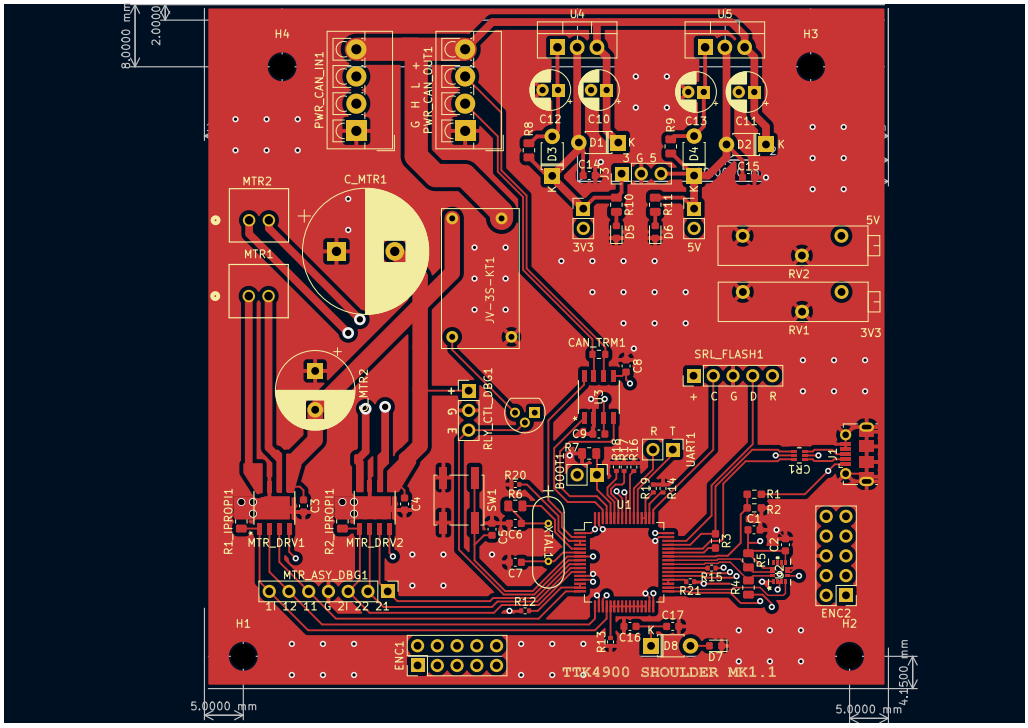


Figure 12: Front copper and silk layers of the improved shoulder board

7.8 Hand

The hand control unit could not be verified during the specialisation project due to a short circuit between the power input line and ground layer. The cause was found to be a misplaced stitching via, and care was taken to avoid this error in this and every other board produced afterwards. As the various subassemblies had largely been verified in the other control units, it was assumed that this was the only error preventing the hand unit from functioning correctly. Pin header legend for the hand control unit is presented in table 9, and boards are presented in figures 13 and 14.

The hand circuit is almsot identical to the shoulder with regards to present assemblies, except that it does not have a USB assembly. The circuit is presented in figure 19.

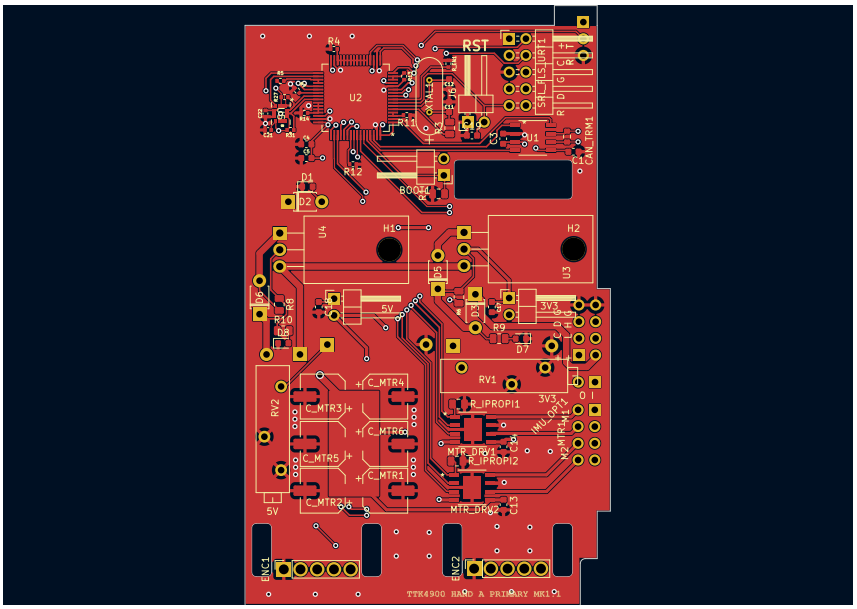
7.8.1 Hand B: Optical sensor

The twist optical sensor is mounted on a separate PCB due to placement constraints in the hand chassis, and connected to the hand control unit via a three pin header. The PCB is referred to as hand B, and the circuit is presented in figure 20.

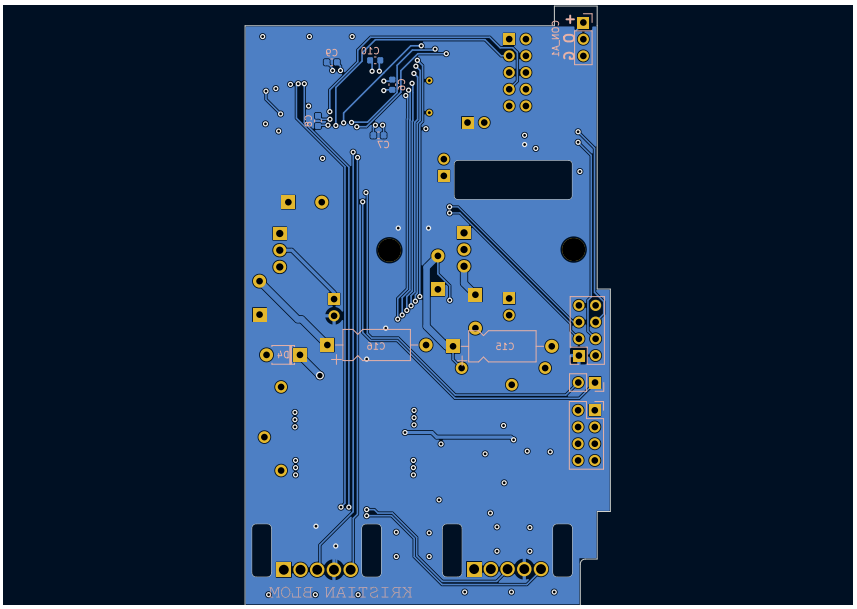
The OPB971 optical sensor activates when its aperture is obstructed, such that a short circuit occurs between the output and ground pins. A voltage sensor placed between the output and voltage input pins will then sense the differential. A breadboard test was conducted around this concept with an LED placed between the input and output pins: when the aperture was obstructed, the LED lit. The optical sensor output pin was routed to a GPIO pin on the hand MCU on the assumption that it would act as a substitute for the LED, sensing the input/output differential upon activation of the optical sensor.

SRL_FLS_URT1	**Meaning**	**PWR_CAN_IMU1**	**Meaning**
+	VDD Target	+	5V output
C	Programmer clock	C	I2C clock
G	Ground	D	I2C data
D	Programmer data	G	Ground
R	Reset target	+	Input voltage
T	UART Tx	L	CANL
R	UART Rx	H	CANH
CON_A1	**Meaning**	G	Ground
+	5V output	**IMU_OPT1**	**Meaning**
O	Twist optical sensor	O	Wrist optical sensor
G	Ground	I	IMU interrupt

Table 9: Pin header legends for the hand control unit. **CAUTION:** The 5V output on the I2C header, adjacent to power input on the CAN/power header, MUST NOT have a voltage applied to it. As before, the symbols are to be interpreted as overlays of the pins.



Front copper of the improved hand control unit



Back copper of the improved hand control unit

Figure 13: Front and back copper layers of the improved hand unit

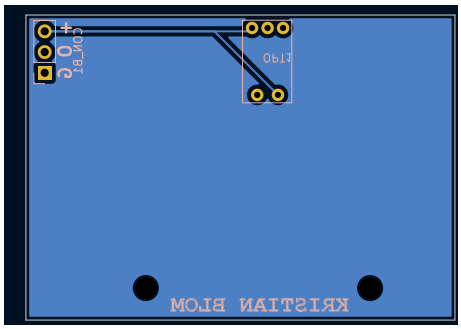


Figure 14: Back copper and silk layers of the hand B board

7.9 Verification

All PCBs were tested before installation into the arm before installation, similar to the specialisation project. They are presented in order of dependency on previously tested systems, sorted by assembly.

7.9.1 Voltage regulators

For each of the control units, voltage adjustment potentiometers RV1 and RV2 were turned until 3.3V and 5V were measured between ground and the relevant pin of `LO_VLT_DBG1`, see fig 26. Jumpers JP1 and JP2 were then connected with jumpers, and voltages were measured again to ensure no voltage drops, which would be indicative of a short circuit, were present. No voltage drops were measured, and the voltage regulators were judged to be operational.

7.9.2 MCU programming

The previously generated MCU pinout was compiled using VSCode and the ST extension (see section 6.1), and attempts were made to flash each of the control units also using the ST extension with the resultant binary file. This yielded no errors, effectively verifying the MCU voltage supply, serial/flash header designs, and the connection between the external computer and ST-LINK programming unit embedded in the Nucleo development kit. The MCUs were deemed operational.

7.9.3 Motor control relay, GPIO

The motor control relay makes an audible click when activated. The program mentioned in the previous test was expanded to include a simple loop activating and deactivating the relay every second by writing to the `RELAY_EN_PIN` on PC0. This tests the aliasing of GPIO pins to human readable names (see section 8.5), the transistor circuit, and the relay. Audible clicks were heard, and relay control was deemed operational. This test applies to the torso and shoulder control units, as the hand does not use a relay.

7.9.4 Motor drivers, PWM generation

PWM was tested using a test program from the specialisation project, updated to the Mk1.1 MCU pinout. The program generates a triangle wave at approximately 0.5Hz, which is used to scale the PWM duty cycle output from pins PA[2,3] and PC[1,2] (see section 8.8) for motor driver 1 and 2, respectively. The duty cycles must be inverse in order for the motor drivers to activate correctly, i.e. if PA2 has 40%DT, PA3 must have 60%DT, see datasheet[19] chapter 8.4.1. This test applies to all control units.

PWM output was verified by probing the `MTR_ASY_DBG1` header, see fig 25, with an oscilloscope. PWM signals with waxing/waning duty cycles according to the generated triangle wave were observed for both motor drivers on all control units. The test was repeated for the motor driver output pins, and the PWM signals were observed here, too, at a voltage identical to the system input voltage of 20V[4].

Addressing the key improvement point from the specialisation project, no voltage was measured on the `DRVn_IPROPI` lines during this test. This implies that the discussed interference mode is not present in the current peripheral configuration, and the motor driver assemblies were deemed operational.

[4]The highest setting of the avaiable power supply

7.9.5 UART data transmission

UART transmission was tested by sending the character string `HELLO, WORLD!` using configuration parameters discussed in section 8.9 to the external computer via the Nucleo development kit's UART-to-USB adapter (see chapter 6.8 of UM1724[17]). The signal was listened for using PuTTy with configuration settings mirroring those of the MCU, and was received successfully from all control units.

UART reception was tested by activating the motor control relay upon reception of the character `R`, detected via the UART reception interrupt handler. This was successful for the torso and shoulder control units. The hand control unit had no simple way to test reception, and this was assumed to be functional based on the results from the torso and shoulder units. UART data transmission was deemed operational.

7.9.6 Twist optical sensor

The twist optical sensor was tested by enabling an interrupt on PC11 upon a rising edge, and using sending a message over UART in the interrupt handler. The sensor was obstructed using a paper sheet, but no interrupt was triggered.

The error was found to be in the design of the sensor circuit. As mentioned in section 7.8, the optical sensor shorts its output pin to ground. Thus, the MCU will always measure 0V between output and ground. To amend this, two resistors of $10k\Omega$ were added between V_{cc} and OUT, OUT and Ground, respectively, of the Hand B connector pin header[5]. Before obstruction, the MCU will measure 2.5V, or half of the optical sensor V_{cc}, which is above the five volt tolerant pin activation threshold of 1.85V for a MCU supply voltage of 3.3V (MCU datasheet, table 66[16]). On obstruction of the sensor, the measured voltage will be 0V.

The interrupt was reconfigured to activate on a falling edge, and the test was repeated. The interrupt was triggered, and the optical sensor was deemed operational. This test applies to the hand control unit.

7.9.7 End switch and wrist optical sensors

The end switch and wrist sensors were tested by applying a 5V input voltage, and measuring the voltage between the OUT and ground pins when the switch was pressed/wrist joint manipulated to obstruct the sensor. Output voltage was found to be 2.7V. As discussed in section 7.9.6, this is above the five volt pin activation threshold, and the sensors were deemed to be operational.

7.9.8 I2C data transfer

I2C was tested by attempting to read the `WHO_AM_I` register of the three LSM6DSM IMU units[13] using a program developed for the specialisation project, and displaying the value via UART. The program utilises the STM32 HAL library discussed in section 8.1, which returns a status message of `HAL_ERROR` if a function fails in hardware. If the test is succesful, IMU assemblies, IMU boards and I2C peripheral configuration have been designed/assembled correctly.

The tests were generally not successful, and further testing was necessary to isolate the error(s). Several units were used in the process:

- Hand control unit: has one onboard IMU (IMU2) on I2C port 3, and one external IMU (IMU1) on I2C port 1.
- Shoulder control unit: has one onboard IMU (IMU0) on I2C port 3.

[5] Changing the PCB design itself was not deemed worth the time and money

- Breadboard IMU: an IMU on a breakout board, previously used for circuit design.
- IMU PCB 1: One copy of the IMU board design seen in figure 7, intended to function as IMU1.
- IMU PCB 2: Similar to IMU PCB 1, made for these tests.
- Adafruit unit: Adafruit MMA8451 accelerometer breakout[1], a COTS IMU bought for these tests, and to act as IMU board replacements should a solution not be found.
- Arduino UNO: Hobby/development kit for embedded programming, explicitly compatible with the Adafruit unit.

Attempts were made at reading the `WHO_AM_I` registers of all availabe IMUs, results presented in table 10.

IMU\MCU	Hand I2C1	Hand I2C3	Shoulder I2C3	Arduino UNO
Hand onboard	-	HAL_ERROR	-	-
Breadboard	WHO_AM_I	-	-	-
IMU PCB 1	HAL_ERROR	-	-	-
IMU PCB 2	HAL_ERROR	-	-	-
Shoulder onboard	-	-	WHO_AM_I	-
Adafruit	HAL_ERROR	-	-	WHO_AM_I

Table 10: Summary of attempts to read IMU registers

No debug headers were included in the I2C/IMU assemblies. Oscilloscope readings were therefore limited to Hand I2C1, where a breadboard was used to insert probes along the data and clock lines. Common for negative (`HAL_ERROR`) results was that the waveform looked correct up until the point where a slave ACK (see 5.1) should have appeared. This indicates that the circuit is correctly designed, and that I2C1 is correctly configured (see 8.6).

Furthermore, the breadboard IMU worked with hand I2C1, and the shoulder onboard IMU worked with I2C3. This proves definitively that I2C peripherals are viably configured, and it strengthens the indication that circuit design is viable: Shoulder and hand onboard IMUs both use the IMU assembly schematic (fig 22), so any difference between the two would be limited to circuit layout. No noteworthy differences were found in the layout of the hand and shoulder onboard IMU circuits. The IMU assembly itself is based on the breadboarded design, which did also work.

One difference between the IMU PCB design (fig 21) and the IMU assembly is that both interrupt pins are never grounded at the same time, see equation 1. No indication was found that this would lead to undefined behaviour beyond a note that the output is "forced to ground" by default. See table 19 of the datasheet[13]. However, when the breadboard IMU circuit was modified to let one or both interrupt(s) float, communication with this unit failed, too. This indicates that the IMU board circuit design may need to be revisited.

All circuits except the Adafruit unit were hand soldered, and some difference in quality may be expected. Soldering was done by reflow oven and solder paste, and had generally been successful thus far in the project. However, due to its 14 pads at a pitch of 0.5mm in an LGA package, the LSM6DSM was significantly more difficult to solder than other ICs. Relevant solder points of MCUs and IMUs were inspected using a stereoscope, but no difference between functional and non-functional units could be established[6]. As a test for visual inspection, IMU board was resoldered with a deliberately uneven amounts of solder paste on the IMU pads. The IMU was clearly tilted after soldering, but there was no indication of short circuiting between pads with too much paste. However, pins with very little paste appeared to have been disconnected as the IMU tilted. None of these clues to improper connections could be seen in the other units[7]. The MCUs were also inspected, and I2C pins appeared to be in order. At this point, six IMUs had been soldered with

[6]No figures could be acquired from the stereoscope

[7]Getting a clear view of the connection was difficult in all cases, which in itself may be a source of error

only two positive results. Considering that all were soldered with little variation in process, it seems improbable that the soldering process should be the only cause of the negative results.

An attempt was made at establishing whether the IMUs themselves may be defunct. This was initially assigned a very low probability considering the effect it would have on ST's business model should it routinely ship defunct batches of ICs, but one result lends the hypothesis credibility: An attempt at reading the shoulder onboard IMU's Z axis acceleration, it failed. Reading other axes, both acceleration and rotation rate, was successful. No further tests could be made to establish whether the remainder of the batch was defunct.

The Adafruit units which were supposed to act as a replacement for the IMU boards also failed when paired with hand I2C1. It is assumed to be because the MMA8451 requires a repeated start condition when initialising communication with the master (Overview chapter[1]). While the STM32F303 is configurable for repeated start condition (chapter 25.2.1 of UM1786[15], this was not implemented due to time constraints.

Summarised:

- The I2C peripheral configuration is viable.
- The IMU assembly circuit is not incorrect.
- The IMU board design may need revisiting – grounding both interrupt pins.
- The soldering process may need to be revisited.
- Two axes are available on the shoulder joint: X and Y.

I2C data transfer was deemed minimally operational.

7.9.9 USB data transfer

Didn't work, lost motivation due to time and UART working.

7.9.10 CAN bus data transfer

In order to verify CAN bus, the hand control unit was programmed to send a CAN message to the torso control unit upon activation of the twist optical sensor via the interrupt handler. The torso control unit was programmed to send a message over UART upon reception of a CAN message from the hand unit. The twist joint was then moved to the position where the sensor should activate. A message was read in PuTTy, and CAN bus was deemed operational.

7.9.11 verification summary

- Voltage regulators correctly output 3.3V and 5V.
- MCUs may be programmed via the Nucleo devkit ST-LINK programmer.
- Motor control relays correctly function as a dead man's switch.
- Motor drivers are controllable via PWM.
- Information may be exchanged with the arm via UART.
- Twist and end stop sensors are operational and in use.
- Wrist sensor is operational, but not in use.
- One of three IMUs are partially operational.

- USB was dropped due to complexity and time, lack of need.
- Information may be exchanged between control units via CAN bus.

7.10 Installation

Following verification, all boards were installed into the arm except the IMU board: No plan existed to make use of the wrist optical sensor, and the IMU, as discussed, was not operational.

The verification process was repeated after installation, and revealed no major design flaws. Certain THT components in the hand control unit touched the hand chassis, and had to be covered in electrical tape, however.

The installation process is described in detail in section 18.1.

7.11 Circuit diagrams

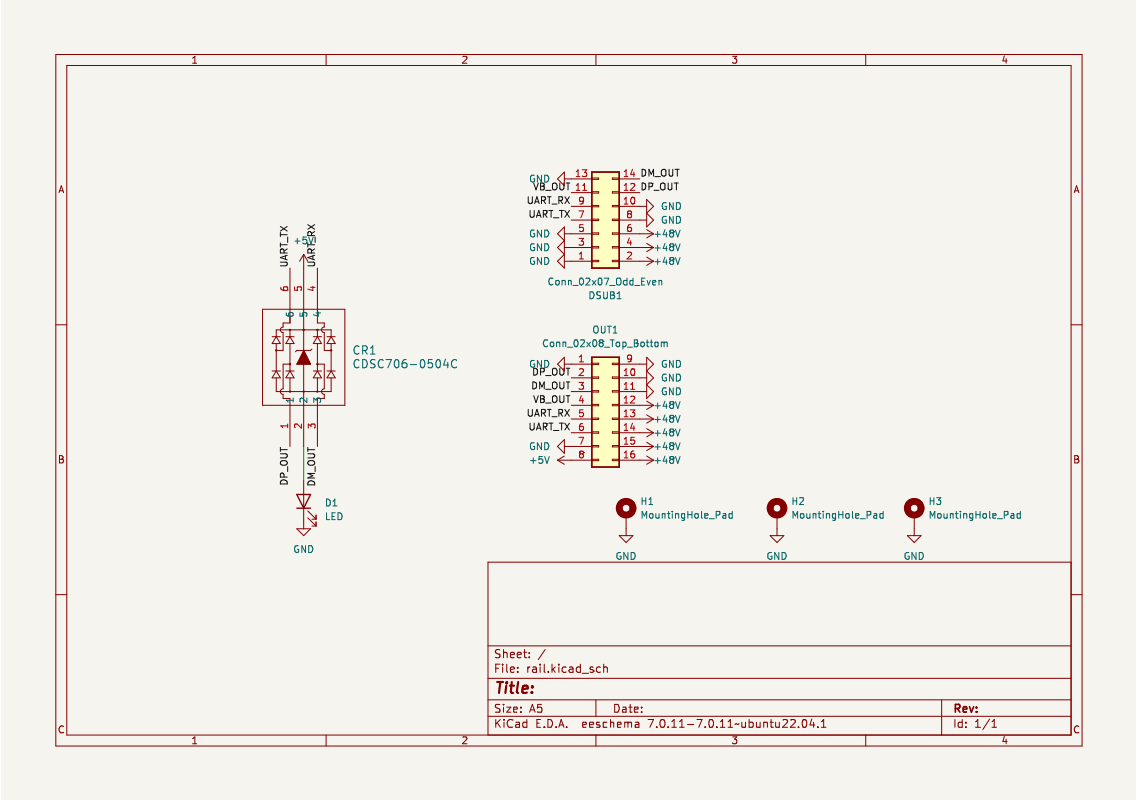


Figure 15: Circuit diagram of the Mk1.1 rail board

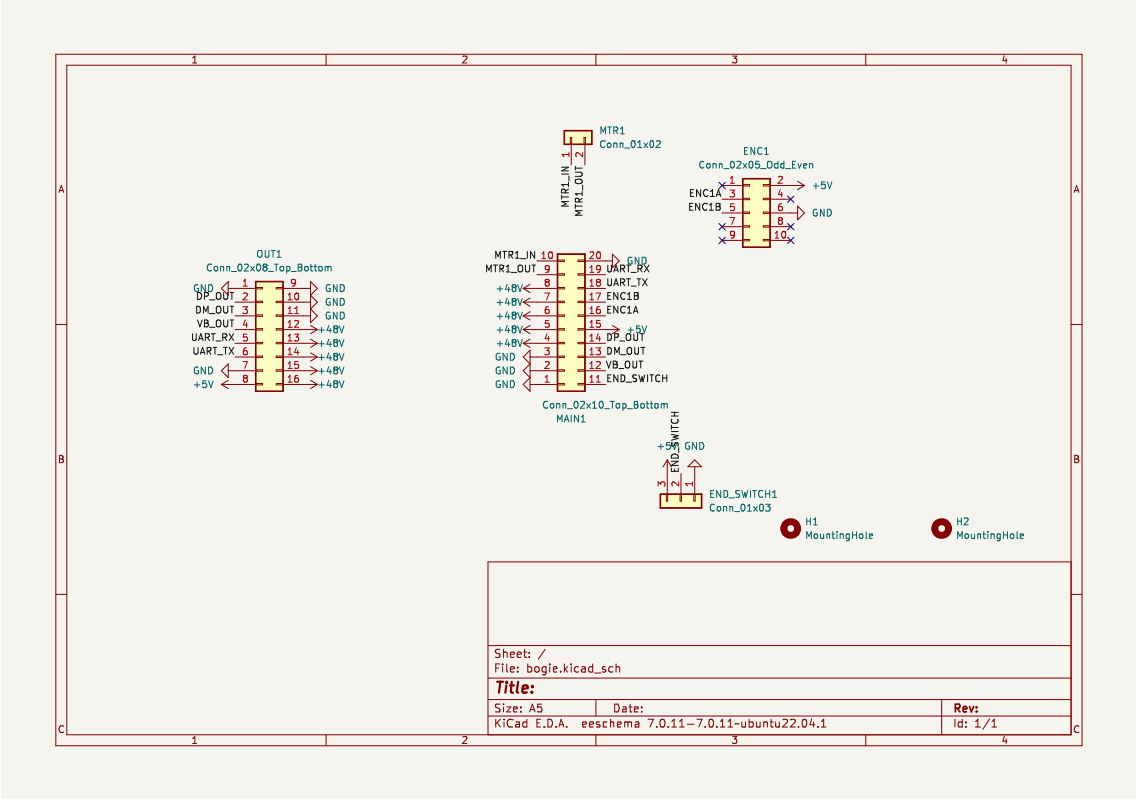


Figure 16: Circuit diagram of the Mk1.1 bogie board

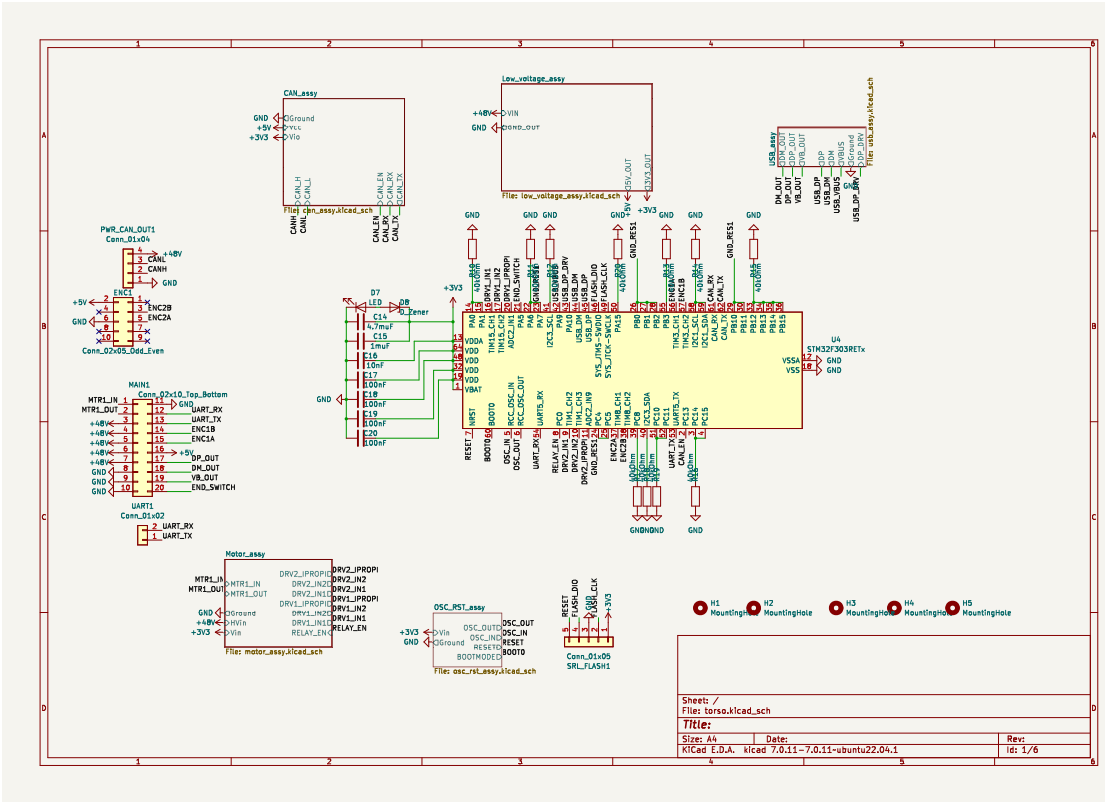


Figure 17: Circuit diagram of the Mk1.1 torso control unit

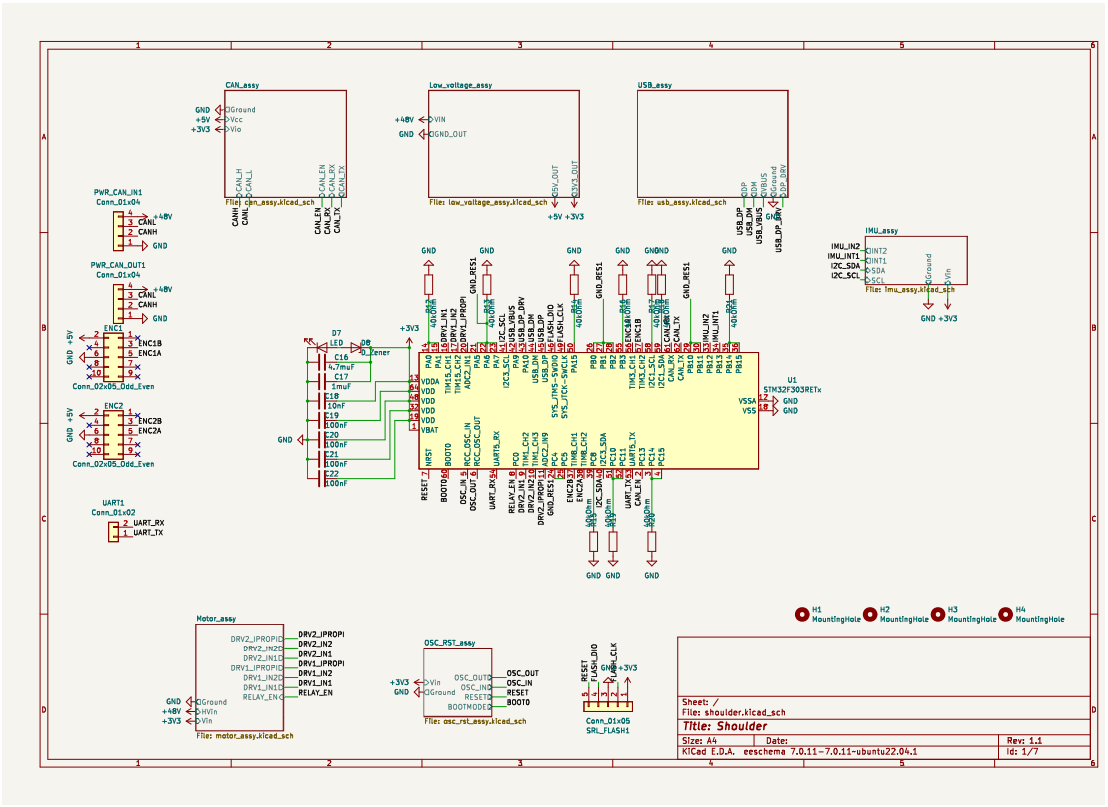


Figure 18: Circuit diagram of the Mk1.1 shoulder control unit

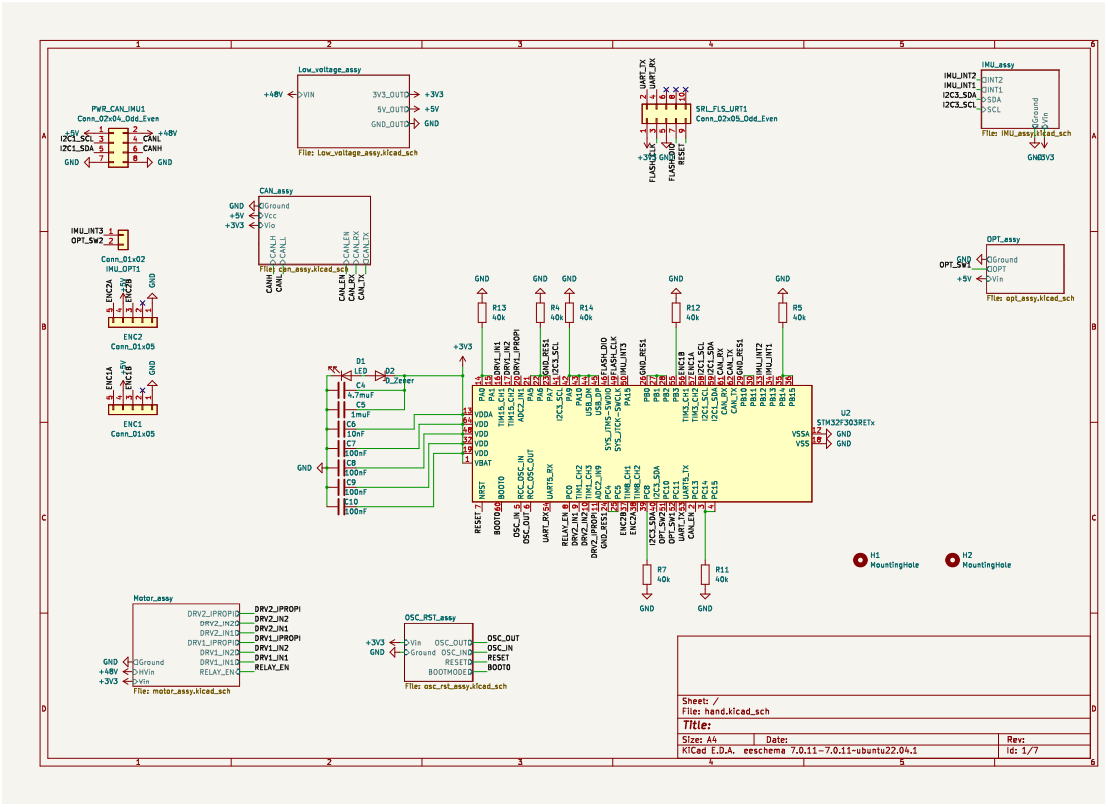


Figure 19: Circuit diagram of the Mk1.1 hand control unit

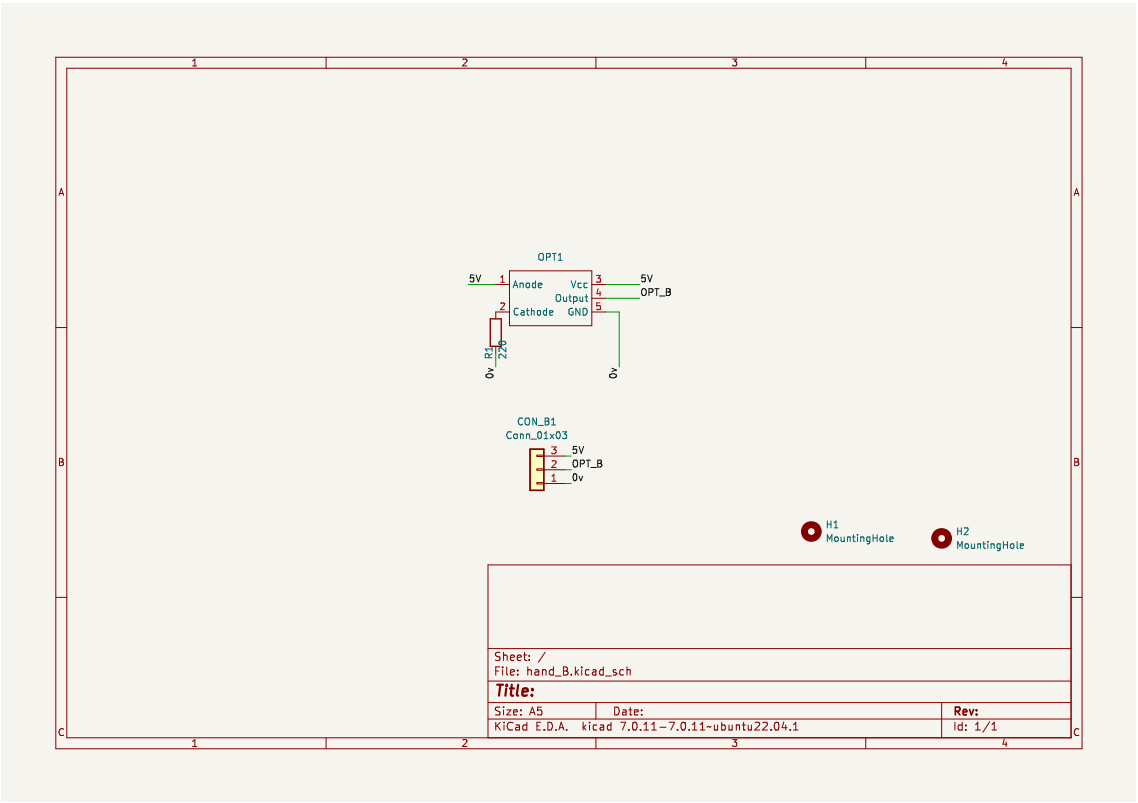


Figure 20: Circuit diagram of the twist optical sensor

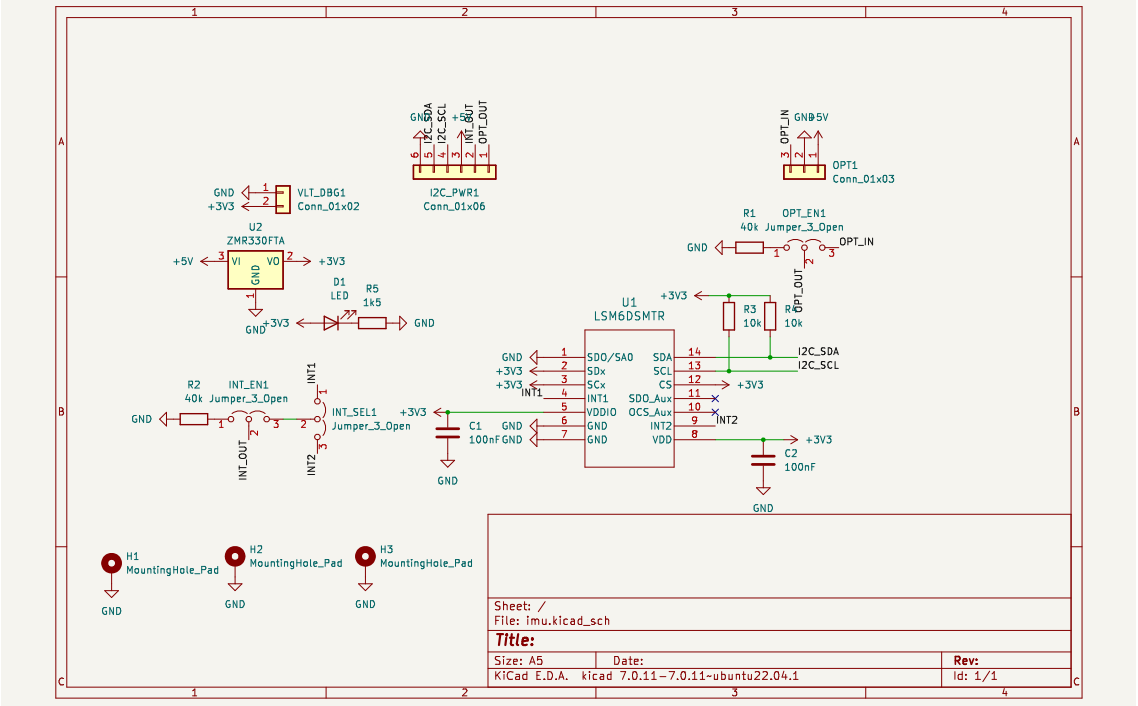


Figure 21: Circuit diagram of the Mk1.1 IMU board

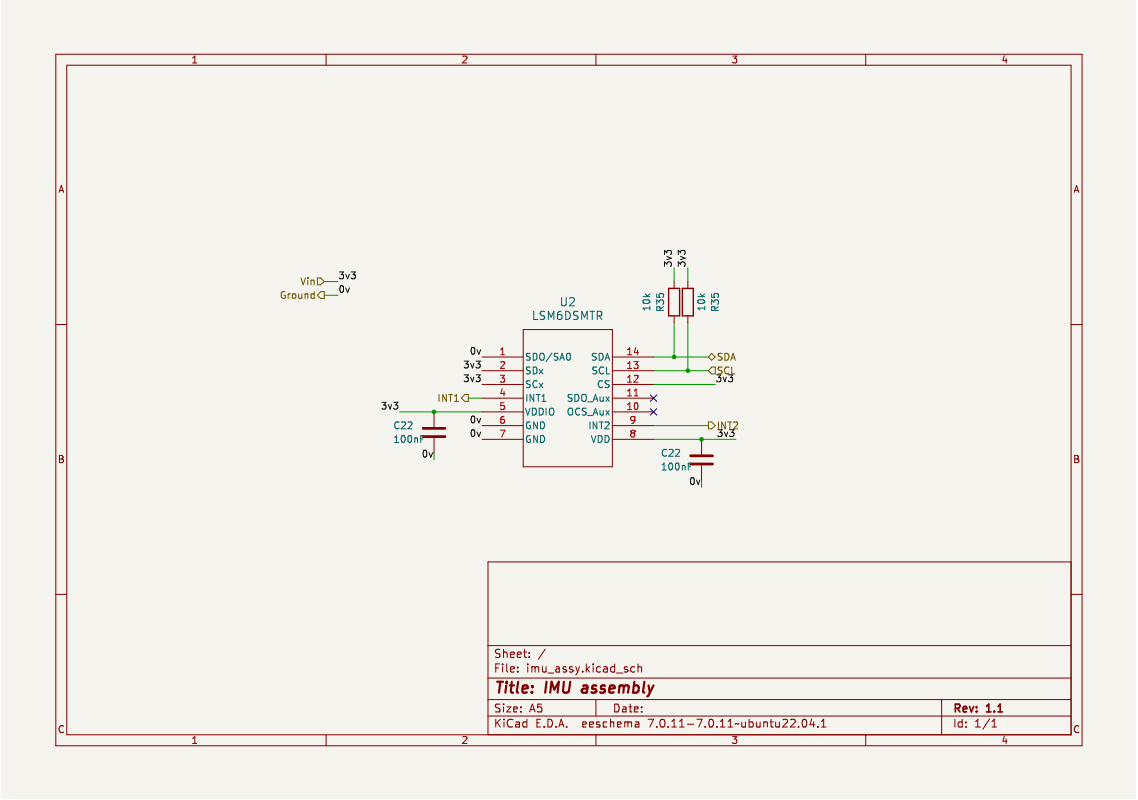


Figure 22: Circuit diagram of the Mk1.1 IMU assembly, present in the IMU board, shoulder and hand control units

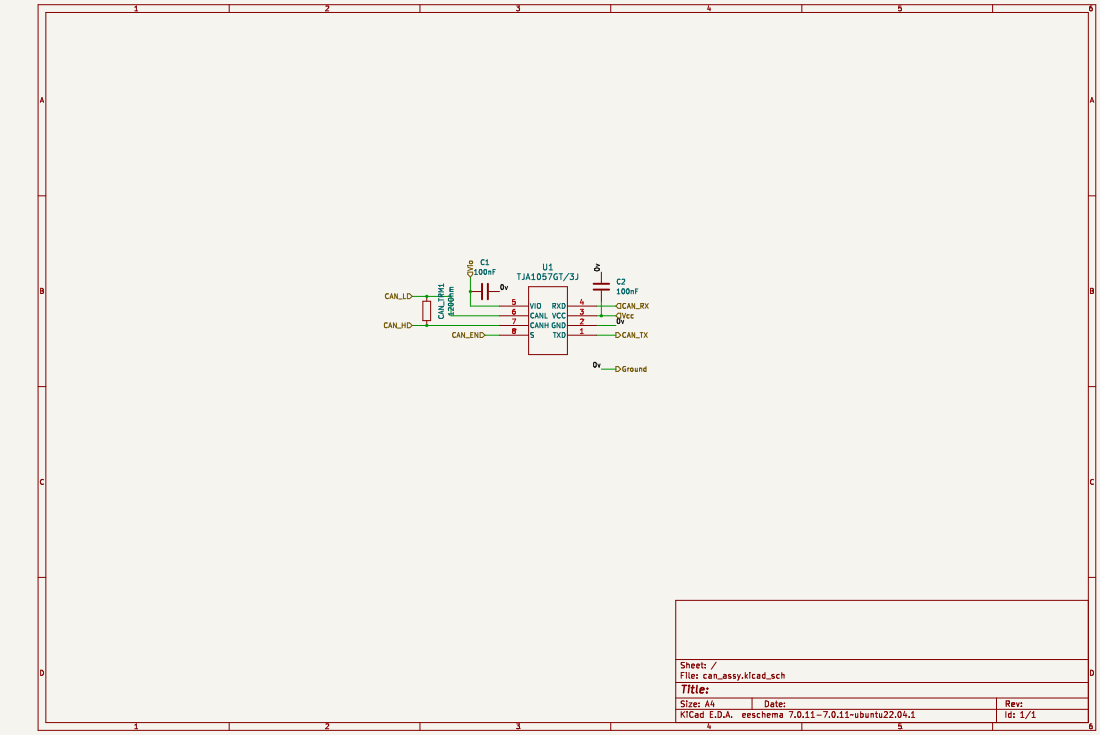


Figure 23: Circuit diagram of the Mk1.1 CAN assembly, present in all control units. Note that the schematic shows the erroneous GT transceiver instead of the correct BT transceiver

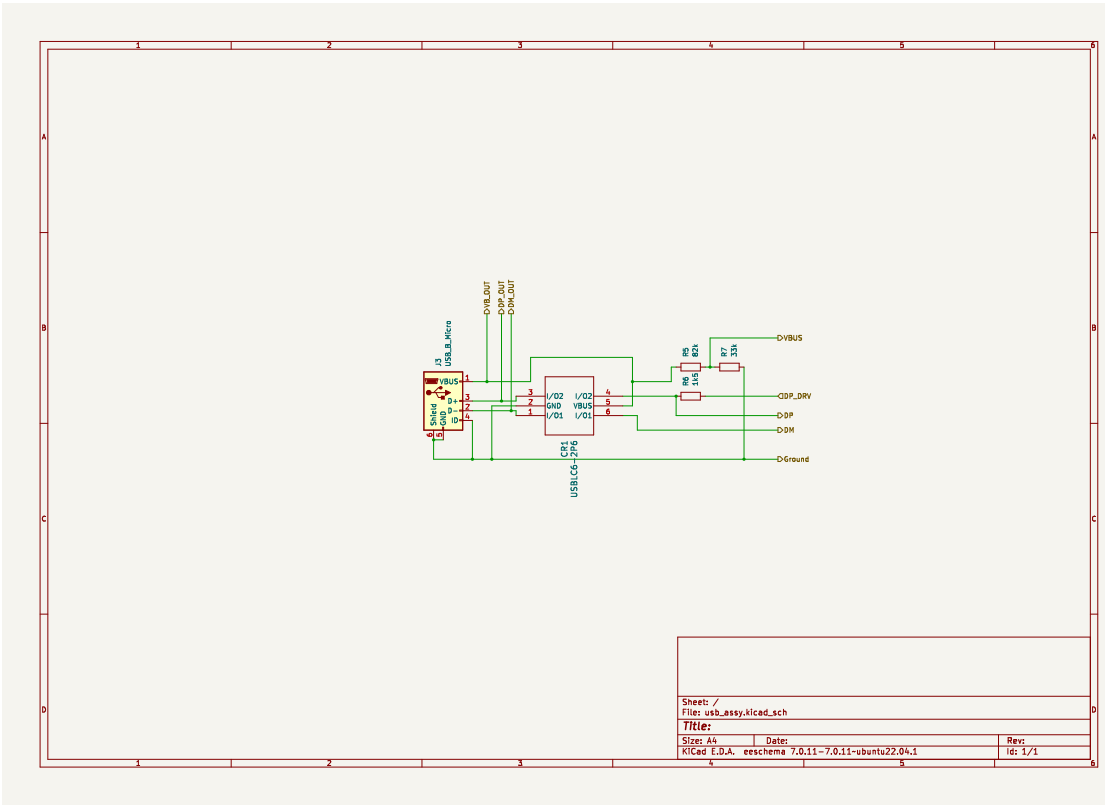


Figure 24: Circuit diagram of the Mk1.1 USB assembly, present in the torso and shoulder control units

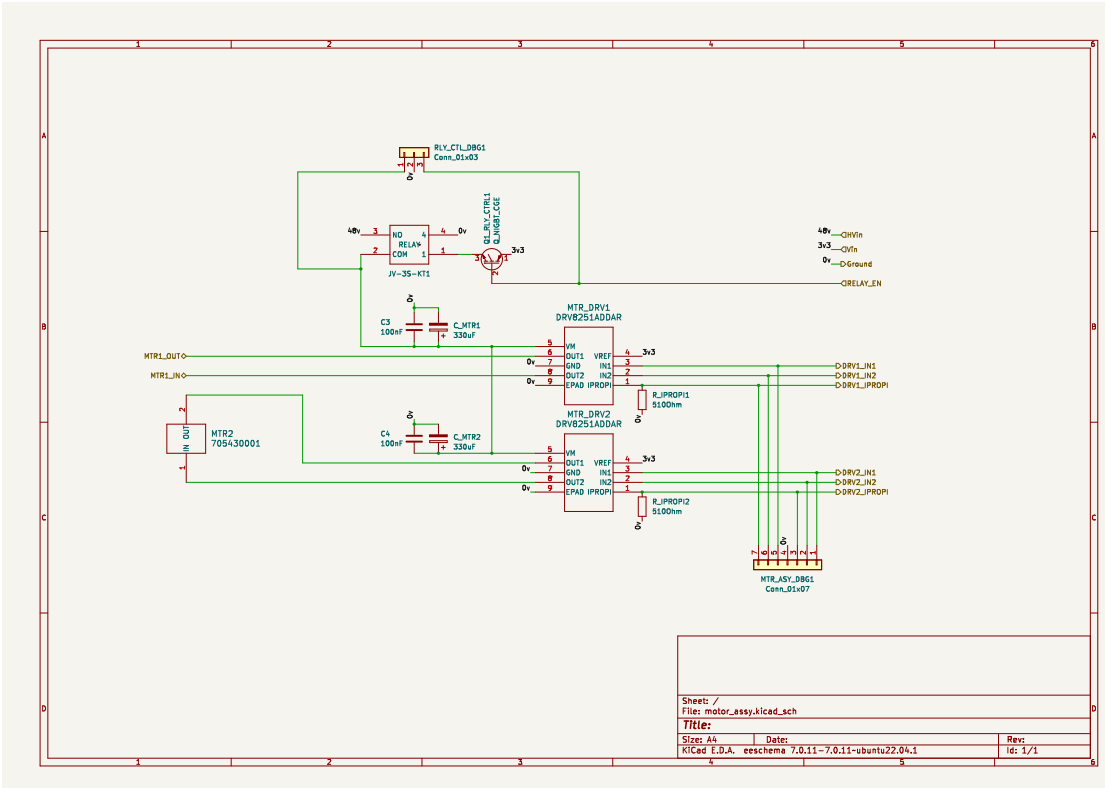


Figure 25: Circuit diagram of the Mk1.1 motor driver assembly, present in the control units

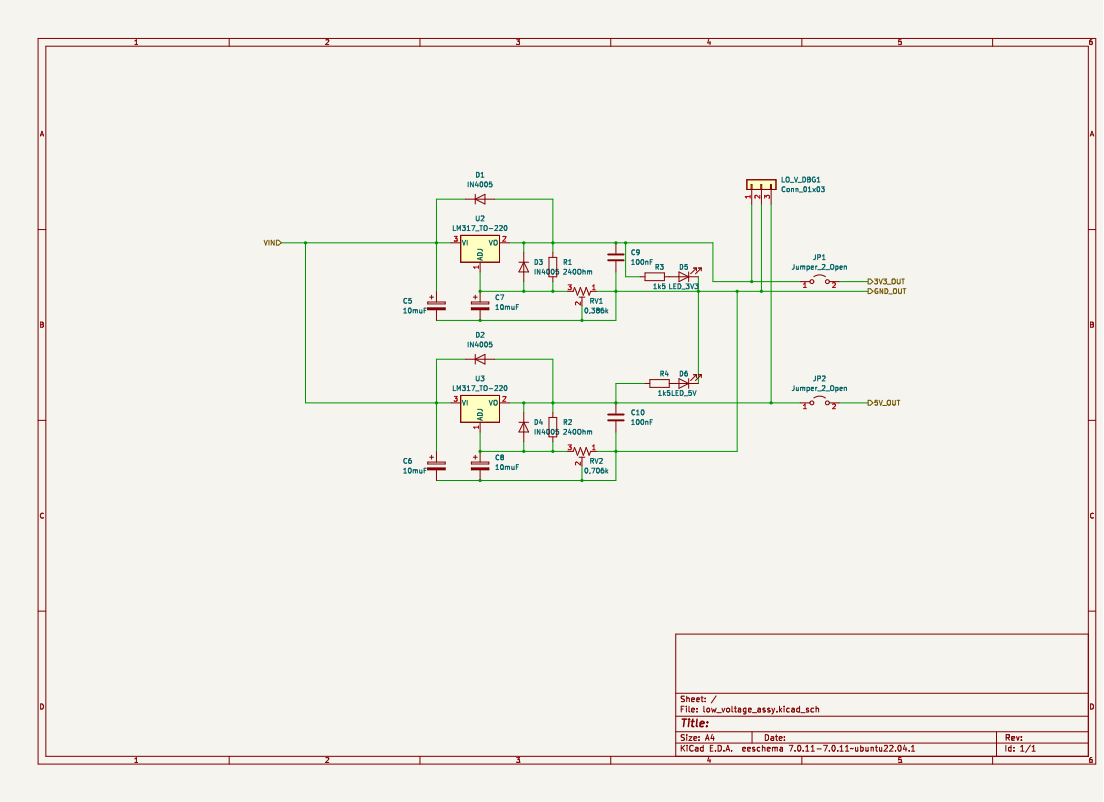


Figure 26: Circuit diagram of the Mk1.1 voltage regulator assembly, present in the control units

8 Peripheral configuration

This section presents the configuration of peripherals and certain core functions of the STM32F303RE microprocessor. Peripheral usage lays the foundation for the software architecture presented in section 9, and configuration was done in STM32CubeMX, see section 8.2.

8.1 STM32F303 overview

The STM32F303RE microprocessor unit ("the MCU") couples an Arm Cortex M4 core with several peripheral units such as ADCs/DACs, multipurpose timers and communication interfaces, as well as advanced memory functions such as direct memory access (DMA) and memory area protection. It has a nested vectored interrupt controller (NVIC) with 73 channels, enabling most peripherals to be associated with at least one interrupt channel. With 512KB of flash memory, it could also fit a basic operating system such as FreeRTOS without memory expansions. It operates at a maximum frequency of 72MHz.

This project does not utilise the F303 to its full extent, and only functionality relevant to the project is presented here. Primary sources for the configuration and usage of the STM32F303 have been the MCU datasheet[16], reference manual RM0316 for the STMF3 family[14] and user manual UM1786 for the Harware Abstraction Layer (HAL) software library[15].

8.2 STM32CubeMX

As mentioned in section 6.1, ST provides a GUI tool for the configuration of their microprocessors called STM32CubeMX ("CubeMX"). The tool generates initialisation functions in C for most functionality in the processors, with some exceptions – for instance, CAN bus message filters must

be configured in code. Upon code generation, CubeMX translates the chosen settings to HAL function arguments, preprocessor definitions et cetera and outputs .c/.h files in a folder structure with a user specified root folder. All code generated by CubeMX could have been written manually, but it was generally found to save much time which would otherwise have been spent scouring RM0316 and UM1786 for how to flip which bits in which registers. CubeMX is not open source, but is free to download upon registration with ST.

The folder structure generated by CubeMX lays the foundation for project management, and is illustrated below. The root folder is `stm_config_official`, a user chosen name. The `Core` folder contains .h and .c files for peripheral initialisation, including `main.c`. `Drivers` and `Middlewares` contain various library functions necessary for more advanced functionality such as USB, and may be configured to contain code examples[8]. The `USB_DEVICE` folder contains library functions unique to the USB peripheral. Other peripherals do not have unique folders dedicated to them.

```
EXAMPLE GENERATED FOLDER STRUCTURE
|- stm_config_official
    |- .mxproject
    |- Core
        |- Inc
        |- Src
    |- Makefile
    |- Drivers
    |- Middlewares
    |- USB_DEVICE
```

```
EXAMPLE GENERATED CODE STRUCTURE
  /* USER CODE BEGIN TIM15_Init 0 */
  /* USER CODE END TIM15_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM15_Init 1 */
  /* USER CODE END TIM15_Init 1 */
  htim15.Instance = TIM15;
  htim15.Init.Prescaler = 0;
  htim15.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim15.Init.Period = 2880;
```

Files generated by CubeMX may be edited. However, CubeMX must be configured to "keep user code when re-generating", and code written outside the `USER CODE BEGIN/END` tags may be deleted regardless. In the above example, snipped from a generated `Core` file called `tim.c`, CubeMX sets certain initialisation variables for a timer. Some of these are mirrored in the lower middle square of27, while others are opaque. All generated files have several sections dedicated to user code.

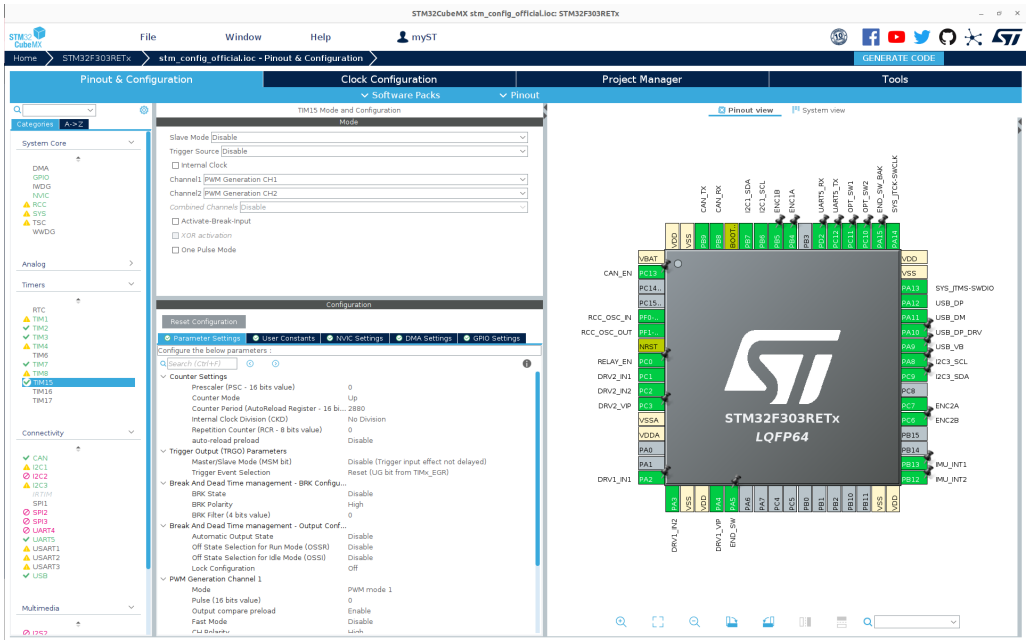


Figure 27: An example of peripheral configuration in STMCubeMX. Peripherals are selected in the left column, and configured in the middle. Mapping to pins may be selected from the MCU representation on the right, where relevant.

[8] About 2GB of code examples. Be sure to opt out!

8.3 ADC

ADCs are used to measure motor current. Channel 9 on ADC1 is pulled to pin PC1 and coupled with motor driver 2, while channel 1 on ADC2 is pulled to pin PA4 and coupled with motor driver 1, see figs 17 and 25. The ADC peripheral is described in chapter 15 of RM0316[14].

8.3.1 Parameters

Parameters not mentioned are left to their default values.

- Channels: Single-ended (measurement relative to ground)
- Resolution: ADCs are set to their maximum resolution of 12 bits
- Continuous conversion mode: Enabled
- Analog watchdog 1: Enabled
- Analog watchdog channel: Channel 9 for ADC1, channel 1 for ADC2
- Watchdog high threshold: 4000
- Watchdog interrupt mode: Enabled

8.3.2 Configuration description

The choice of channels was given by compatibility with the chosen MCU pinout, and resolution set to maximum because no advantage could be found in choosing a lower setting.

Continuous conversion mode makes the ADC start a conversion as soon as the previous has finished – "fire and forget". As soon as the ADC has been started in the program main function, it will continue working until stopped. This saves complexity in software, as the associated driver (see section 10.6) only needs to access the latest conversion result rather than triggering and waiting for a conversion.

The watchdog is part of the implementation of the safety function mentioned in section 4.3. It will trigger an interrupt if the ADC conversion result is above 4000, corresponding to a motor current draw of approximately 4A (see section 5.5), which then may be acted upon.

8.4 CAN

CAN bus is the communication backbone of this system, and CubeMX only covers transmission setup. CAN message filtering and filter configuration is covered in sections 10.5 and 9.6. The CAN bus peripheral is described in chapter 31 of RM0316.

8.4.1 Parameters

Parameters not mentioned are left to their default values

- Prescaler: 9
- Time quanta in bit segment 1: 2 times
- NVIC settings: `CAN_RX0` interrupts: Enabled

8.4.2 Configuration description

Prescaler and time quanta values were chosen to obtain a baud of 1Mb/s, which is the maximum attainable transmission rate for the MCU's CAN peripheral. The CAN peripheral has two configurable message reception FIFO queues, and an interrupt has been enabled for the first of the two, `CAN_RX_FIFO0`. As not using an interrupt for the handling of incoming messages would be impractical, this implies that FIFO1 will not be in use.

8.5 GPIO

GPIO is discussed in section 7.2, and concerns the configuration of all MCU pins.

8.5.1 Parameters

Parameters not mentioned are either discussed in other sections, or left to their default values.

- NVIC EXTI line[9:5] interrupts: Enabled
- NVIC EXTI line[15:10] interrupts: Enabled

8.5.2 Configuration description

Enabling external interrupts and events controller (EXTI) lines ensures that interrupts will be enabled for PA5, PA15, PC10 and PC11, which are reserved for use with the system's optical switches. EXTI is described in chapter 14.2 of RM0316.

8.6 I2C

I2C is used for communication with the system's IMUs. I2C1 and I2C3 are enabled, chosen because they were available. The I2C peripheral is described in chapter 28 of RM0316.

8.6.1 Parameters

Parameters not mentioned are left to their default values.

- Speed mode: Fast mode
- Frequency: 400 kHz
- Primary address length selection: 7 bit

8.6.2 Configuration description

The bus frequency of 400kHz was chosen in order to maximise bus capacity. The LSM6DSM is compatible with fast mode I2C (datasheet chapter 6.4[13]), and 400kHz was successfully tested during the specialisation project. The number of address bits is also given by the slave unit, here 7.

Other configurations, including several transmission rates, were tested during the verification process described in section 7.9.8. As the one functional unit (shoulder) remained operational at the highest possible frequency (400kHz), this setting was kept.

8.7 Clock

Clock configuration is done in a separate tab of the CubeMX GUI, and lets the user set clock frequencies for most peripherals of the MCU. Clocks are described in chapter 9 of RM0316.

8.7.1 Parameters

Parameters not mentioned are left to their default values.

- HSE input frequency: 16MHz
- PLL source mux: HSE
- System clock mux: PLLCLK
- HCLK 72MHz

8.7.2 Configuration description

The high speed external clock (HSE) frequency was set to 16MHz to match that of the system's oscillator crystal. Phased-locked loop (PLL) source was set to HSE, enabling the PLL to use the external oscillator rather than the MCU's internal 8MHz oscillator (HSI). System clock mux was set to PLLCLK, enabling the system clock to reach 72MHz rather than 16MHz. HCLK was set to the maximum frequency of 72MHz, and PLL values were calculated automatically based on this choice. Other configurations, such as using the HSI, would have resulted in invalid frequencies for the USB or other peripheral clocks. Clock configuration is illustrated in figure 28.

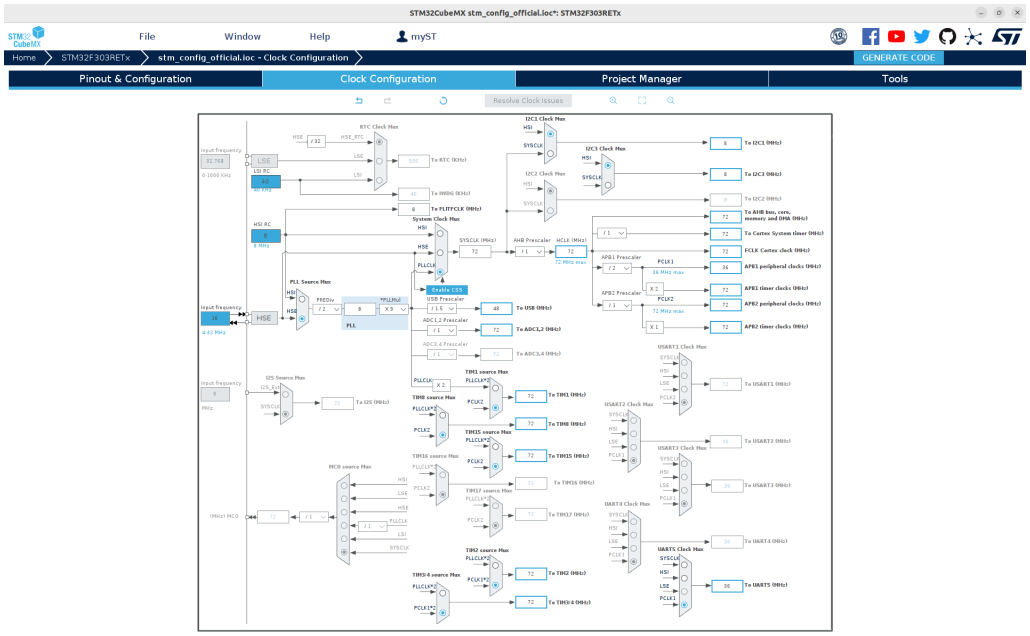


Figure 28: Clock configuration in CubeMX. The GUI presents a number of configration options to the user for tuning clock frequencies of various functions. Prescaler values are generally calculated automatically, and will yield error messages if the user attempts to set an invalid configuration.

8.8 Timers

Timers have multiple roles in the system. They act as encoder input channels, PWM signal generators and interrupt generators for various tasks discussed in section 9.8. The timer peripherals

are described in chapters 20, 21 and 22 of RM0316.

8.8.1 Encoder timers: TIM3, TIM8

TIM3 and TIM8 were configured for encoder input and were pulled to pins PB4 and PB5, and PC6 and PC7, respectively.

8.8.1.1 Parameters Parameters not mentioned are left to their default values.

- Combined channels: encoder mode
- Counter period: 65535

8.8.1.2 Configuration description Selecting encoder mode enables the timer to automatically update its counter register based on the state of its two input pins, connected directly to the output pins of the encoder. The counter register is 16 bits wide, and setting the counter period to 65535 makes use of the whole register. Depending on the resolution of the encoders versus the mechanical movement range of the relevant joint, it may be necessary to handle overflow/underflow of the register.

8.8.2 PWM generators: TIM1, TIM15

TIM1 and TIM15 were configured for PWM generation and pulled to pins PC1 and PC2, and PA2 and PA3, respectively. They were chosen because they are compatible with PWM generation and a pin placement practical for PCB design.

8.8.2.1 Parameters Parameters not mentioned are left to their default values.

- PWM generation: CH2 and CH3 on TIM1, CH1 and CH2 on TIM15
- Counter period: 2880

8.8.2.2 Configuration description Selecting PWM mode enables the timer to output a PWM signal on the selected channels. The PWM frequency is determined by a relation between the system clock frequency and the counter period register, while the PWM duty cycle is set in the timer's capture/compare register. The counter period of 2880 corresponds to a PWM frequency of approximately 25kHz, which is outside the human audible range of 20kHz. The counter period was initially set to 7200, corresponding to a PWM frequency of 10kHz, but this was found to be disturbing to humans nearby the arm as the motor drivers tend to emit a constant noise of the same frequency[9]. Timer frequency calculations are described in section 5.8.

8.8.3 Interrupt timers: TIM2, TIM4, TIM7

TIM2, TIM4 and TIM7 were activated to serve as periodic interrupt generators, primarily to prevent overloading of data buses.

[9]Presumably, as no appropriate testing equipment was available

8.8.3.1 Parameters

Parameters not mentioned are left to their default values.

- TIM2 prescaler: 7199
- TIM2 counter period: 29
- TIM4 prescaler: 719
- TIM4 counter period: 10
- TIM7 prescaler: 7199
- TIM7 counter period: 200
- NVIC settings: global interrupts enabled for all three timers

8.8.3.2 Configuration description

Prescaler and counter period values were calculated such that the capture/compare registers of each timer would reach the counter period value at frequencies of 345Hz, 10kHz and 50Hz respectively, with interrupts triggering on this event. Frequency calculations are discussed in section 5.8, and the reasoning behind the target frequencies is presented in 9.8.

8.9 UART

The UART peripheral is responsible for communication with the external computer, and UART5 was routed to pins PC12 and PD2. UART5 was chosen over other UART peripherals due to PC12 and PD2 being easy to trace to the UART header pin on the torso control unit (see 11). USART/UART is described in chapter 29 of RM0316.

8.9.1 Parameters

Parameters not mentioned are left to their default values.

- Mode: Asynchronous
- Baud: 115200 b/s
- Word length: 8 bits, including parity
- Parity: None
- Stop bits: 1

8.9.2 Configuration description

Bitrate was set to 115200 as this was the highest commonly used bitrate which was successfully tested. Remaining parameters are default, but are included in the interest of documentation.

8.10 USB

The USB peripheral is responsible for communication with the external computer, as an alternative to UART, and is routed to pins PA[9..12]. The USB peripheral of the STM32F303RE may act as a USB full-speed device according to the USB2.0 standard, utilising an internal USB physical interface. USB is described in chapter 31 of RM0316 as well as AN4879[12].

8.10.1 Parameters

Parameters not mentioned are left to their default values.

- USB device (FS): enabled
- Middleware/usb_device class for FS IP: Communication device class (Virtual COM port)

8.10.2 Configuration description

In addition to enabling the peripheral device, a set of hardware drivers provided by ST was included via the "Middlewares" section in CubeMX. These drivers provide library functions similar to the HAL.

9 Software architecture

This section presents the structure of the software written during this project. This includes naming conventions, development patterns and other general considerations which informed the implementation of modules described in section 10. Details of implementation are mentioned here if they were relevant to the system's structure as a whole.

9.1 Introduction

Impact of hw on sw; joints vs motors, state machine

9.2 Implementation of Architectural Requirements

See section 4.3 for the complete list of, and motivation behind, architectural requirements.

9.2.1 Naming conventions

9.2.1.1 Definition: Module This convention implements **AR1: The system should consist of clearly defined modules**

A module is defined as a set of .c and .h files sharing the same name. A module has a specific purpose or is associated with one specific category of hardware. As the C language has no clear definition of classes/objects, modules function as a conceptual replacement. *Example:* The CAN bus module consists of the `can_driver.c` and `can_driver.h` files.

9.2.1.2 Module names This convention is part of the implementation of **AR1.2: A module should be limited in scope and purpose** and **AR1.4: Naming conventions should apply across modules**

Modules are named by their role in the system, and adhere to one of the following patterns:

- `<noun>_driver`: Module is responsible for interaction with the hardware type given by `<noun>`. These modules represent the abstraction level above the CubeMX generated modules, and typically make use of the HAL functions associated with that module.
- `<noun>_controller`: Module is responsible for control of the hardware type given by `<noun>`. These modules represent the abstration level above `_driver`, and may make use of several of those modules.
- `<noun>_parser`: Module is responsible for parsing input from the UART peripheral.

9.2.1.3 Function names This convention implements **AR1.1: A module's interface should be clearly defined**, and is part of the implementation of **AR1.4: Naming conventions should apply across modules**.

Functions are typically named by what module they belong to and whether they are part of the module's interface, analogous with a public/private modifier in other languages.

- `<module>_interface_<verb>_<noun>`: `interface` indicates that the function may be used outside of the module, and uses a short form of the module name.
- `<module_name>_<verb>_<noun>`: Function does something with the data type indicated by `<noun>`, and uses the full name of the module. These functions may not be used outside the module.

Verbs indicate what the function does to the data type indicated by `<noun>`, and generally fall into one of the following categories:

- `get/set/clear`: Manipulate data in a struct associated with the module.
- `calculate`: Calculate a value indicated by `<noun>` based on a locally relevant heuristic/algorithm.
- `update`: Renew data in a struct associated with the module. Typically incorporate calls to the associated `calculate` and `set` functions.
- `handle`: Exclusive to the CAN module, indicate that the function handles an incoming CAN message of a type indicated by `<noun>`.

9.2.2 Development patterns

Some abstract design patterns were adhered to in order to comply with the architectural requirements. Together, they implement **AR3: Design patterns should apply across modules**. Additionally, standardising development patterns reduces mental overhead when developing a new module, and should make it easier to develop an intuition for how to use the code base for future developers.

9.2.2.1 Sharing information between modules: This pattern implements **AR1.3: It should not be possible to pass information to a module outside of its interface**.

As mentioned in section 9.2.1, functions are primarily divided between driver functions and interface functions; "private" and "public" functions of a module, respectively. The C language enables "private" variables and functions via the `static` keyword, which limits the visibility of that symbol to the compilation unit in which it is defined. In this project, information containers (usually structs) which are module specific are defined as static and accessed by pointer.

`Driver` functions frequently access static variables by passing their pointers as arguments, and would therefore fail or otherwise exhibit undefined behaviour if called outside their modules. This is why an `interface` function must be used when interacting with a module. These functions typically call a `driver` function with a scalar (or otherwise non-pointer) substitute for the protected variable as an argument.

Having a set of functions which is explicitly public provides safety in two directions: the user knows that the function is safe to call from outside the module, and the programmer knows which functions whose safety needs to be guaranteed.

9.2.2.2 Structs as hardware abstractions: The system consists of several unique hardware units with similar properties. Analogous to classes in C++, this project uses structs to store static and variable information about each unit. For instance, all motors have a dedicated struct defined in the motor driver storing information such as its associated timer peripherals, torque constant et cetera.

9.2.2.3 Lists of structs: In order to avoid writing selection logic to account for unique variable names given to hardware abstracting structs in the `driver` type functions, structs are typically grouped in lists of pointers to the relevant structs. Again using the motor driver as an example in the following pseudocode and including `interface` usage:

```
//We want to avoid the following:
motor_driver_set_power(motor_struct* motor, int power):
    if(motor == motor1):
        &motor1.power_setting = power;
    else if(motor == motor2):
        &motor2.power_setting = power;

//Listing structs allows us to do this instead:
motor_struct* motors[2] =
    &motor1,
    &motor2;

motor_driver_set_power(motor_struct* motor, int power):
    motor.power_setting = power;

motor_interface_set_power(int desired_motor, int power):
    motor_driver_set_power(motors[desired_motor], power);

//And then make the following calls as needed:
motor_driver_set_power(motors[desired_motor], power);
motor_interface_set_power(desired_motor, power);
```

This pattern moves the responsibility of selecting the correct motor from the function to the caller, makes the function easier to read, and lets the developer focus on the purpose of the function rather than hardware selection logic. This is particularly beneficial in cases where the number of available units is subject to change. For instance, the shoulder control unit has one IMU, while the hand has two.

Relating to **AR2.1: SOLID principles should be adhered to, to the extent that they apply** and theory discussed in section 5.9, this pattern demonstrates application of the open-closed, Liskov substitution and dependency inversion principles:

- The motor driver function does not need to be changed if the circumstances of motor selection changes (OCP, LSP); and
- the motor driver function is agnostic to the existence of specific motor structs (LSP, DIP).

9.2.2.4 Preprocessor statements This project uses conditional preprocessor statements to choose which modules, functions et cetera are compiled at a given time. The primary purpose of this is to differentiate between the three MCUs: The torso unit does not need the accelerometer driver, and the other two do not need the UART driver during arm operation. However, all units may need the UART driver for debugging – or the USB driver, depending on which method of communication is desired.

Preprocessor statement makes it relatively simple to enable and disable modules (or parts of modules) as they are needed, and was chosen over other methods such as having one development branch or an entire project dedicated to each MCU. Conditionals are set in a unit configuration file, examplified below:

```
//Definitions set and used in the configuration file:
#define TORSO 0
#define SHOULDER 1
```

```
#define HAND 2
#define ACTIVE_UNIT HAND

//Conditional statement wrapping code in a module:
#if ACTIVE_UNIT == HAND
//Do things
#endif
```

9.3 Code organisation

All code is managed by a single git repository, available at GITHUB, and placed in one of two directories: code pertaining to ROS nodes (`ros`), and code pertaining to the MCUs (`stm_config_official`).

Code pertaining to the MCUs is structured according to the description given in section 8.2. Code written as part of this project can be found in a directory called `TTK4900_drivers`, which is underneath the CubeMX generated directory `stm_config_offical`. This structure makes the Makefile easier to read and modify compared to placing the `TTK4900_drivers` directory outside of the `stm_config_official` directory. Some exceptions to this rule exist, where CubeMX generated files have been edited to configure peripherals beyond what CubeMX supports. They are discussed in the relevant subsections of 10.

Code pertaining to ROS nodes is structured according to best practice for ROS packages. Each node is given a dedicated directory with the typical `src/inc` structure, and placed in the `ros/src` directory.

9.4 Arm onboard

The following sections discuss specifics of the structure of software designed to run on the MCUs, giving context to choices made when writing the individual drivers discussed in section 10.

9.5 Peer to peer vs master/slave

While the torso MCU is uniquely responsible for administering communication with the external computer and relaying information to and from the other MCUs, all MCUs are considered to be peers. This simplifies architecture design somewhat compared to a master/slave structure, as there is no need to implement functionality unique to one MCU beyond hardware capabilities. As CAN bus is inherently peer to peer (see 5.6), a master/slave structure would also need to be implemented on top of the CAN bus protocol, increasing overall system complexity.

9.6 CAN bus message ID structure

As discussed in section 4.2, the MCUs communicate via CAN bus, a key design element in which is the message ID which the MCU's CAN peripheral uses to filter messages (see 5.6). The 11 bit standard ID is used in this system, and this section discusses how the ID is used in the CAN driver. The implementation is discussed in section 10.5.

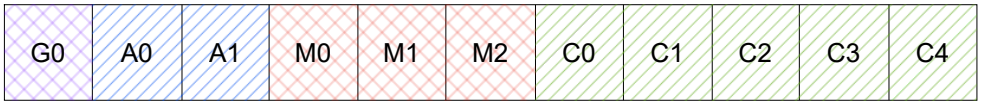


Figure 29: CAN bus message ID structure

The 11 bits of the CAN message ID field has been split in two sections which may be understood as noun/verb indicators: The first six bits indicate which recipient/target (noun) and the last five bits what command (verb) the message represents. Furthermore, noun bits are split in three categories: Global/general (G), accelerometer (A) and motor (M). This is illustrated in figure 29, with command bits designated C.

This structure may support 32 commands per category, for a total of 96 message types. While wasteful compared to the 2048 message types 11 bits could theoretically support, it makes message filtering efficient compared to a system where the recipient may be indicated by interdependent bit values: The CAN peripheral may filter out messages not intended for that unit, rather than the MCU core having to spend cycles on processing the ID to reach the same conclusion.

9.6.1 Motor messages

Motor messages are for joint control and motor telemetry requests[10]. Three bits were dedicated for this category as the arm has six joints.

- M0 and M1: Determine which control unit the addressed joint belongs to.
- M2: Determines which of the two joints of a control unit the command should be applied to.

9.6.2 Accelerometer messages

Accelerometer messages are used to request and send IMU data. Two bits were dedicated for this category as there are the arm has three IMUs.

- A0: Determines which control unit the addressed IMU belongs to.
- A1: Determines which IMU the command should be applied to.

9.6.3 Global messages

Global messages contain state information or other information relevant to every control unit.

9.7 Input: UART vs USB

As the arm may communicate with its external control computer via either USB or UART, the `string_cmd_parser` module, which is responsible for parsing (human written) string commands, was written such that the relevant serial interface could be enabled via a preprocessor statement.

9.7.1 ROS vs Terminal

As the robotic arm may be controlled either automatically through MoveIt (see 9.9) or through a terminal with human-readable commands, two modules were written for the parsing of serial input data: `ros_uart_parser`[11] and `string_cmd_parser`. A preprocessor statement is used to enable one or the other.

[10] The category was established before the joint/motor distinction was made

[11] This was written long after USB was given up on, and may not be compatible with USB – hence the name

9.8 Interrupts

As mentioned in section 8.8, three interrupts have been set to trigger at various frequencies. As the MCU does not run an operating system which could have locked the relevant tasks in threads with local timers, this scheduling system ensures that the motor control loop, CAN message transmitter and UART transmitter operate at appropriate frequencies. In the case of the control loop, the primary concern is consistent calculation of time derivatives/integrals, while in the case of the buses the concern is bus contention.

The timers trigger significantly more slowly than the MCU clock at 72MHz, so it assumed that the MCU will be able to complete the main loop at least once between the triggering of an interrupt. The interrupt handlers set flags which are polled by the main loop, and used to determine whether or not to enter a given procedure.

9.8.1 Control loop timer: 10kHz

The control loop timer was arbitrarily set to operate on 10kHz, meaning that the PID controller is updated at a frequency no higher than 10kHz. The primary concern with this loop is consistent calculation of the arm's movement, as the PID controller uses time data when calculating its output.

9.8.2 UART timer: 50Hz

The UART timer was set to 50Hz, meaning that the torso control unit will not try to send telemetry updates more often than this, but may respond to specific messages as they are received. The frequency was chosen on the basis of a 115200b/s UART link, a full state report of 192 bits and a significant margin. See section 5.7 for the full calculation.

9.8.3 CAN timer: 344Hz

The CAN timer was set to 344Hz, meaning that the control units will not try to generate CAN messages more often than this. As sending CAN messages is handled by the CAN peripheral, the CAN timer's responsibility is to ensure that the system does not generate messages faster than the peripheral is likely to be able to send them. The frequency is based on a round trip estimate of an accelerometer data request, see section 5.6 for the full calculation.

9.9 ROS nodes and the external computer

As illustrated in figure 2, the external control computer runs a number of programs in order to control and communicate with the arm.

With one exception, these programs are ROS nodes. As mentioned in section 6.1, ROS nodes do specialised work and exchange communication over a virtual network with subscriber/publisher topology. The motivation behind introducing ROS in this system is that it has several mature systems for robotic control, including kinematic solvers and graphical user interfaces. These are useful with respect to the project's primary goal, and necessary with respect to the secondary goal. The purpose of the ROS nodes, then, is to bridge a kinematic solver, a GUI and serial communication with the arm's control system. This section gives an overview of their functions, and the nodes as well as ROS in general are discussed in detail in section 12.

9.9.1 Serial reader/writer: PuTTy

Any serial communication program should be applicable, but PuTTy was used for text based communication with the arm throughout most of the project's development. PuTTy enables the user to set positional setpoints for every joint, and request basic telemetry. PuTTy is always used for reading output from the arm, but a ROS node takes write control when ROS is selected as the arm's input mode. This asymmetry exists for the simple reason that no ROS serial reader node could be written in time.

Note: This implies that the ROS system is not a control loop, as no state information or telemetry from the arm is fed back into the ROS nodes.

9.9.2 Kinematic solver and GUI: ROS MoveIt

MoveIt is a software package for robotic control with kinematic solvers, real-time 3D simulator, control GUI and a configuration tool. It can be configured to let the user control the manipulator of any robot with a sufficiently complete URDF description (see 12) via its GUI, and perform motion planning to calculate a time series of positional setpoints for the robot's joints. As MoveIt is part of the ROS ecosystem, the setpoints are published to the ROS virtual network by default.

9.9.3 ROS control listener, HMI and serial communications

In addition to MoveIt, the ROS network consists of three nodes. The "control listener" subscribes to a subset of messages published by MoveIt, namely positional setpoints, and converts them to byte data on a format determined by the arm's UART parser (see 10.8). The "HMI" node offers a text based user interface which replaces PuTTy as keyboard input source when the arm is configured for ROS, and converts input to byte data similar to the control listener node. Finally, the "serial communications" node receives input from the two former nodes, and writes it to the computer's serial interface.

9.10 Architectural overview

Figure 30 summarises the most important interactions between the arm's system modules, but are not a one-to-one representation. For instance "UART parser" could be either the ROS parser module or the terminal input parser module. Furthermore, "UART" should be interchangeable with equivalent "USB" modules. As the USB hardware could not be verified, however, this capability was not tested. Neither UART nor USB is relevant to the shoulder and hand units when the arm is operational.

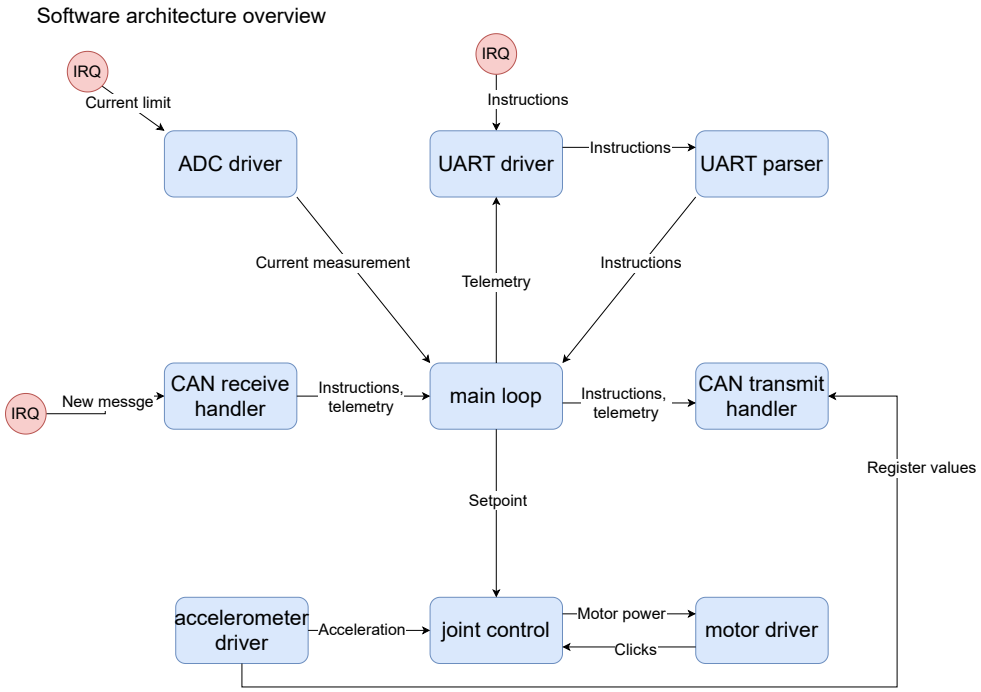


Figure 30: An overview of the central components of the arm's onboard software

Figure 31 illustrates relations between entities on the external computer as described in previous sections as well as the MoveIt motion planning GUI. Note that the XOR gate is normative; no exclusive access to the computer's serial interface is enforced.

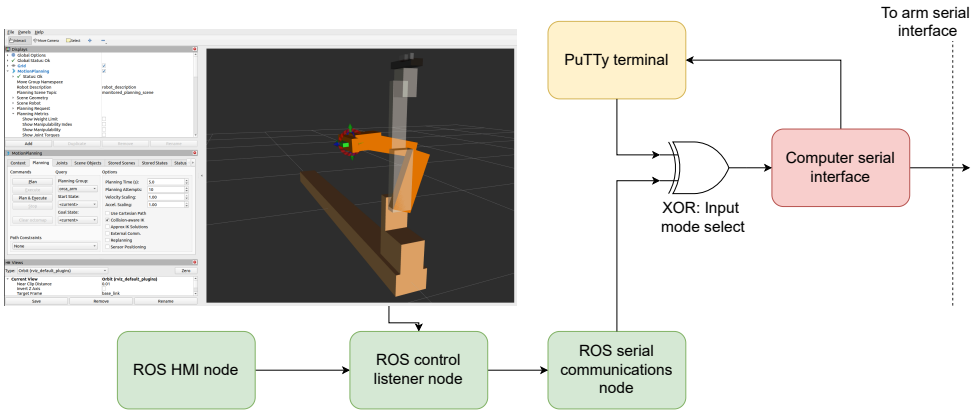


Figure 31: An overview of the central components of software running on the external control computer

10 Hardware drivers

This section presents the hardware drivers written for the project, which include most of the modules defined in the `TTK4900_drivers` directory. The term "driver" is arguably a misnomer, as these modules interact with hardware via the HAL library generated in CubeMX thus are one abstraction level above the hardware. "Firmware" might be a more accurate term, but "driver" was established early in the project development. Higher level control modules, such as the joint controller and main loop are discussed in section 11. The github repository for the project may be found at https://github.com/kristblo/TTK4900_Masteroppgave[3].

Modules are presented roughly in the order in which they were developed. Low coupling between modules was strived for, but some design choices were inevitably a consequence of previous choices. Figures which summarise key elements are provided for most modules. The syntax is as follows:

- Arrow: Implies the direction of information flow
- Yellow bubble: Information container such as a struct
- Red bubble: Iterable, usually a list of structs
- Blue bubble: Module function
- Green bubble: Various, see individual figure
- `[NAME]` in bubble: Item belongs to module indicated by NAME
- "Main loop" bubble: Item is used in the controller's main function loop

10.1 UART driver

The UART driver is responsible for handling incoming data from the UART peripheral, and sending data to it. Depending on the choice of input source, i.e. ROS or human input, it will call one of two handler functions upon triggering of the `UART_RxCplt` ("UART reception complete") interrupt. This module was developed first in the interest of establishing debugging capabilities as soon as possible after development had started[12]. The module's structure is illustrated in figure 32 along with the string parser and ROS input parser modules.

Note: this module was written before the driver/interface pattern was established.

10.1.1 HAL interactions

- `HAL_UART_Transmit`: Transmit a specified number of bytes in blocking mode
- `HAL_UART_Transmit_IT`: Transmit a specified number of bytes in interrupt (non-blocking) mode
- `HAL_UART_Receive_IT`: Receive a specified number of bytes in interrupt mode, store in a specified data structure (buffer). Triggers the `UART_RxCplt` interrupt when the specified number of bytes have been received
- `HAL_UART_RxCpltCallback`: Callback function, triggered by `UART_RxCplt`

[12] Attempts at using the debugging function of the VSCode ST extension were not successful

10.1.2 Key functionality: Human input mode

The module provides a text based interface with the arm via the string parser module (10.2), and is designed around usage with PuTTy. Key features are the ability to write characters input by the user back to PuTTy, and buffering characters until the 'enter' key is registered. The latter hands the buffered characters over to the string command processing module.

The following pseudocode illustrates how incoming characters are handled by the UART module after a character has been placed in `uartRxChar` by the interrupt callback function:

```
uint8 bufferPosition = 0;
uint8 uartRxChar;
uint8 uartRxHoldingBuffer[64];

uart_parse_hmi_input(uartRxChar, uartRxHoldingBuffer, bufferPosition):
    if(uartRxChar == '\r'): //'enter' key registered
        string_cmd_processor(uartRxHoldingBuffer); //Hand buffer over to string processor
        clear_buffer(); //Erase content and reset position counter

    //Actual implementation has additional clause for backspace key.
    else: //Any other key registered
        uartRxHoldingBuffer[bufferPosition] = uartRxChar;
        *bufferPosition = (++(*bufferpos))%64; //Simple overflow protection

uart_send_string(uartRxHoldingBuffer); //Send the updated string back to the user
```

In the implementation of the above pseudocode, all processing happens inside the recption interrupt handler. This is a weakness as the processor spends much time in an interrupt handler which may potentially be preempted by an interrupt of higher priority, or a keystroke may be lost if entered shortly after another. In practice, human input is generally too slow for this to be an issue.

10.1.3 Key functionality: ROS input mode

In ROS input mode, lack of expectation for readback makes handling input significantly less complex than in human input mode. The input interrupt is triggered for every 32 bytes received in the interest of creating a standard message size. The data is handed over to the ROS parser module via its interface, and parsing happens outside of the interrupt handler.

10.2 String command parser

The string command parser module does not interact with the HAL library, but was included in this section as it is closely related to the UART driver and ROS parser. This module defines a set of keywords with arguments which the user may input via the chosen user interface, and a corresponding set of functions to handle each keyword. A string consisting of a keyword and arguments is called a command.

The module receives a string of characters, and divides it into "tokens" separated by a space character. If the first token matches one of the command keywords, it will try to parse the remaining tokens using that keyword's handler function. Handler functions are paired with their keywords in a struct of a string and function pointer, which in turn is stored in a list of string-command-pairs. The string parser is illustrated in figure 32 along with the UART module.

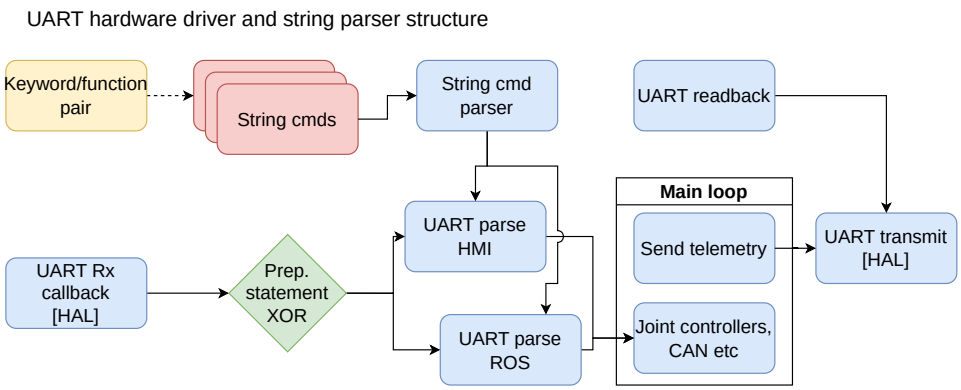


Figure 32: Structural diagram of the UART driver and string parser modules

10.2.1 Commands

- `<joint> stp <e/r> <num>`: Lets the user set a positional setpoint for the joint defined by `<joint>`. The `<e/r>` argument determines whether the numerical setpoint given by `<num>` should be interpreted as encoder clicks or radians. The setpoint is absolute in the sense that "0" is either the joint's position at startup or its calibrated zero position.
- `can`: Not implemented. Intended to let the user queue a CAN message for transmission.
- `s`: Soft emergency stop, sets global state to `GS_IDLE`.
- `acc1 <hexAddr>`: Reads the shoulder accelerometer register specified by `<hexAddr>`. Register must be specified as a hexadecimal value with the '0x' prefix.
- `relay <0/1>`: Activates or deactivates the motor control relay of the control unit.
- `home`: Activates the arm's calibration sequence.
- `state <globalState>`: Lets the user set the arm in one of the three global states (11.2).

Commands were registered as they were needed or thought to be relevant during the development process, leading to a somewhat inconsistent implementation. The setpoint command, for instance, can be used to control any joint while the user interface is connected to any control unit. That is, `elbow stp r -1.5` will always trigger the elbow joint to move to a radian position of 1.5 in the negative direction regardless of which control unit is connected to the external computer[13]. On the other hand, `relay 0` will only turn off the relay of the currently connected control unit. String commands are intended to primarily be a debugging tool, and provide limited value compared to the time it takes to implement them without advanced parsing techniques.

10.2.2 Processing logic

The main string processor function receives a string (i.e. `uint8_t` array) from its caller and inserts whitespace-separated substrings (i.e. words) into a `uint8_t` matrix where, if the command is valid, the first row will equate to a command keyword. Using a list of keyword/function pointer pair, calling the appropriate command handler is straightforward:

```
//Pairs are defined as such:
struct string_cmd_pair:
    uint8_t* cmdString;
    void (*cmdFuncPointer)();
```

[13] By checking if the motor is local or remote, and sending a CAN message in the latter case

```
//...and added to a list as needed:
#define NUM_CMD 12
string_cmd_pair stringCmdList[NUM_CMD]:
    {"rail", string_cmd_rail},
    {"shoulder", string_cmd_shoulder},
    //...
    {"acc1", string_cmd_acc1}

//...then used in the string processor:
void string_cmd_processor(uint8* inputString):
    uint8 tokens[64][64]
    //Tokenisation omitted for brevity, straight to command handler selection:
    for(i >= NUM_CMD):
        if(tokens[0] == stringCmdList[i].cmdString):
            stringCmdList[i].cmdFuncPointer(tokens);
```

The final line in this example calls the associated function directly using the full token matrix as its argument. The processor function is oblivious to the number of existing commands, and how each command is treated. Adding new commands requires that they be registered in the command list, but no switch/case or other selection logic need be changed. This pattern examplifies the Single Responsibility Principle (5.9.1):

- The string command list keeps track of valid command keywords, but is oblivious to their useage.
- The handler function only receives a set of tokens to process.
- The command processor bridges the two, but does not have information about the implementation of either.

10.2.3 Pattern test: Default arguments

This module tested an implementation of default function arguments, a concept not natively supported by the C language. The motivation was that default arguments may enable code reuse and/or improve encapsulation. The pattern was not adopted systemwide as it was found to have limited use, but is worth a brief discussion.

The implementation has four parts:

- A "base" function with the actual implementation:
  `retType function_base(argList)`
- A struct containing the elements of the base's arguments:
  `struct sct_argList`
- A "wrapper" function which takes the struct as an argument:
  `retType function_wrp(sct_argList)`
- A preprocessor statement defining the function name as the wrapper:
  `#define function(...) function_wrp((sct_argList*)(__VA_ARGS__)`

Note the use of the variable argument keyword and ellipsis in the definition. When `function` is called, its input arguments are interpreted as an instantiation of `sct_argList`, which is then passed to `function_wrp`. In turn, the wrapper checks if each argument is non-zero. If not, it passes a default value to the base function.

```
struct funcArgs:
    type1 arg1;
    type2 arg2;

void function_base(type1 arg1, type2 arg2);
void function_wrp(funcArgs* args_in):
    arg1_out = args_in->arg1 ? args_in->arg1 : default1;
    arg2_out = args_in->arg2 ? args_in->arg2 : default2;
    function_base(arg1_out, arg2_out);
#define function(...) function_wrp((funcArgs*)(__VA_ARGS__))
```

This pattern has multiple weaknesses: If `function` is called with 0 as an argument, the base function will be called with the default argument instead. Additionally, if an argument is undefined, all following arguments must also be undefined. In the example above the call `function(arg2)` would lead to `arg2` being interpreted as the first argument. A correct call would then be `function(0, arg2)`, which defeats the purpose somewhat. Finally, the implementation has significant overhead and would likely only be applied to certain functions which were expected to be used often and in multiple contexts. Such functions should not exist at all as they are likely to violate the Single Responsibility Principle. This pattern initially seemed useful in the context of parsing string commands considering that they tend to have multiple formats (length, number of substrings, decision trees et cetera), but was ultimately not used even here.

10.3 Motor driver

The motor driver module is primarily responsible for interacting with the physical motor drivers and the motor encoders. The module name `motor_driver` refers to the DC motors rather than the DRV8251 motor driver ICs, and each motor is represented in a struct containing key information from its datasheet as well as its associated timer peripherals. The motor driver module is illustrated in figure 33.

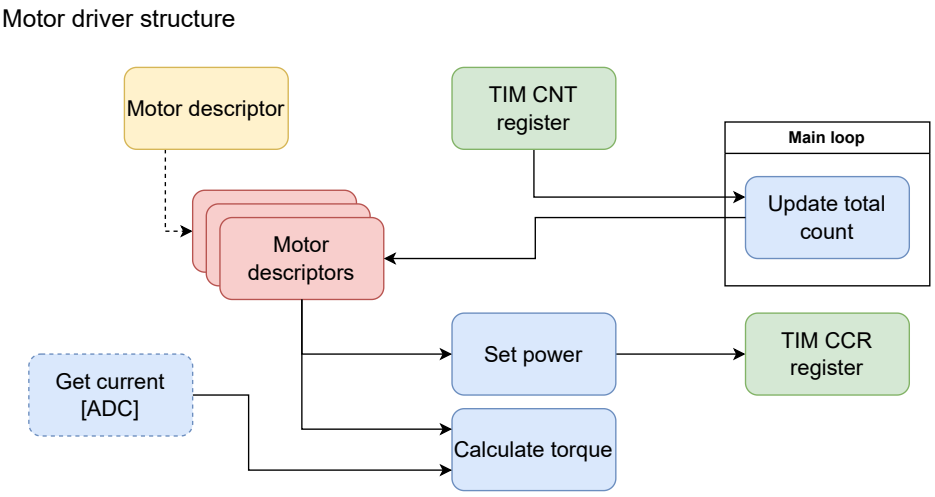


Figure 33: Structural diagram of the motor driver

10.3.1 HAL interactions

The motor driver does not use any HAL library functions, but writes or reads the following timer registers:

- `TIM15.CCRn`: Write to set PWM duty cycle of motor 1

- `TIM1.CCRn`: Write to set PWM duty cycle of motor 2
- `TIM3.CNT`: Read to get raw encoder count of motor 1
- `TIM8.CNT`: Read to get raw encoder count of motor 2

10.3.2 Key functionality: Motor selection and interaction

All six motors are described in a unique struct, a `motor_descriptor`. When developing the hardware, the motor interfaces (i.e. motor drivers, encoder connectors and power connectors) were assigned `Motor1` and `Motor2` for each of the control units, see silk on figure 11 for examples. In order to maintain traceability with hardware, `motor_descriptors` were named `motor1` and `motor2`, and a preprocessor statement determines which pair of structs is active when flashing a given MCU. The two structs are collected in a list which is used by the driver/interface functions, with the consequence that a call such as `motor_interface_function(1)` will act on `motor2` for the given control unit. The below pseudocode illustrates the process:

```
//All motors described in unique structs matching
//their designations in hardware
#if ACTIVE_UNIT == TORSO
static motor_descriptor motor1:
    //Rail motor definition
static motor_descriptor motor2:
    //Shoulder motor description
#endif
#if ACTIVE_UNIT == SHOULDER
static motor_descriptor motor1:
    //Wrist motor definition
static motor_descriptor motor2:
    //Elbow motor description
#endif
//...repeat third time for ACTIVE_UNIT == HAND

//Structs collected in a list
motor_descriptor*[2] motors:
    &motor1,
    &motor2;

//Interface function with call to driver:
motor_interface_set_power(1, 30):
    motor_driver_set_power(motors[1], 30);
```

10.3.3 Key functionality: Motor power setting

As mentioned in section 4.2, the DRV8251 motor driver ICs are controlled by dual channel PWM signals from two timer peripherals. This functionality is abstracted to "power" and measured in per cent duty cycle. Setting a motor's power to 100% means applying the full bus voltage to it, and -100% means applying the full reverse voltage. Duty cycle is setting the timer peripheral's capture compare register (CCR) to a percentage of the timer's counter period. Each motor driver has two timer channels dedicated to control. In accordance with its datasheet[19], the two channels are set reciprocally. That is, "forward 70%" could translate to $CCR1 = 0.7CTR\_PRD$ and $CCR2 = 0.3CTR\_PRD$ for `TIM15` depending on the motor's installed polarity.

CAUTION: Motors have varying maximum voltage ratings. In order to ensure safe operations, a "voltage percentage cap" has been calculated for each motor at 25V main bus voltage and stored in the relevant struct. The driver's `motor_interface_set_power` function will never set a higher

duty cycle than the corresponding motor's voltage percentage cap, and a **bus voltage above 25V should never be applied to the system** without updating the caps.

10.3.4 Key functionality: Compensating for encoder overflow

Encoder timer perpherals store their counter value in a 16 bit register, `TIMn.CNT`, which is insufficient to accurately track joint movement during operation (see section 13.3). Instead, the motor descriptor struct has a 32 bit variable named `encoderTotalCount` to track this, which is updated in the main loop as indicated in figure 33. This function compares the current encoder counter value with the previous encoder counter value, and adds the difference to `encoderTotalCount`.

The counter register will overflow/underflow after reaching an end point (i.e. 0 or 65535). In order to compensate for encoder overflow/underflow, the function will add or subtract 65535 to the difference since the previously measured encoder count if the difference is greater than 1000 clicks. This assumes that the main loop will always repeat faster than the motors will do two full revolutions.

10.4 Accelerometer driver

The accelerometer driver is responsible for interacting with the IMUs via the I2C peripheral. An effort was made to make the driver compatible with both the LSM6DSM[13] and MMA8451[1], but compatibility with the latter was never tested. The driver defines a struct `imu_descriptor` with relevant register addresses of the IMU, and a set of functions to read them.

As mentioned in section 4.2, the only operative IMU is the LSM6DSM mounted on the shoulder. It is initialised to a conversion rate of 833Hz with a range of $\pm 2G$ for all acceleration axes, and $\pm 250DPS$ for all rotational axes (chapters 10.13 and 10.14 of the LSM6DSM datasheet[13]). The resolution is 16bits, and converted measurements are stored in two adjacent single-byte addresses.

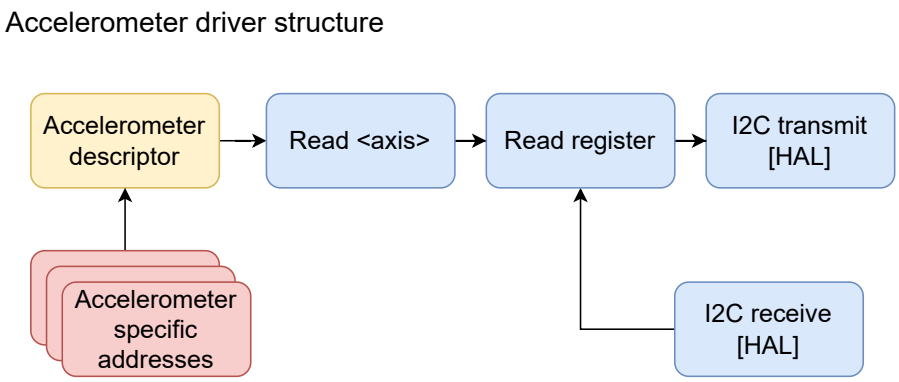


Figure 34: Structural diagram of the accelerometer driver

10.4.1 HAL interactions

- `HAL_I2C_Master_Transmit`: Transmit a number of bytes in blocking mode
- `HAL_I2C_Master_Receive`: Receive a number of bytes in blocking mode

10.4.2 Key functionality: Read and write registers

For either model of IMU, converted values are stored in two single-byte, individually accessible addresses. These address pairs are referred to as "registers" in the driver module. When the

module interface is used to read an axis from an IMU, the function finds the start address of the register in the relevant descriptor struct and makes two consecutive byte read operations on the I2C bus. The bytes are then concatenated and returned as a signed 16 bit integer to be interpreted by the user.

The I2C peripheral is operated in blocking mode, suspending other operations in the IMU while the peripheral anticipates a reply from its slave. This, combined with making single byte, rather than multiple byte, requests likely makes accessing IMU data unnecessarily slow and disruptive to the arm's control system. However, as discussed in section 7.9.8, much time had been spent debugging the I2C/IMU hardware with little success. Consequently, optimising I2C bus access was not prioritised.

10.5 CAN driver

The CAN driver is responsible for handling transmission and reception of CAN messages transmitted between the MCUs. It defines a set of transmit and receive mailboxes, message types, message type callback functions and message type handlers. Message types are defined by the five least significant bits of the CAN message ID, corresponding to its "command" as discussed in section 9.6, and each message type has one transmit and one receive mailbox associated with it. These mailboxes are software defined, and not to be confused with the hardware mailboxes handled by the CAN peripheral. The driver is illustrated in figure 35.

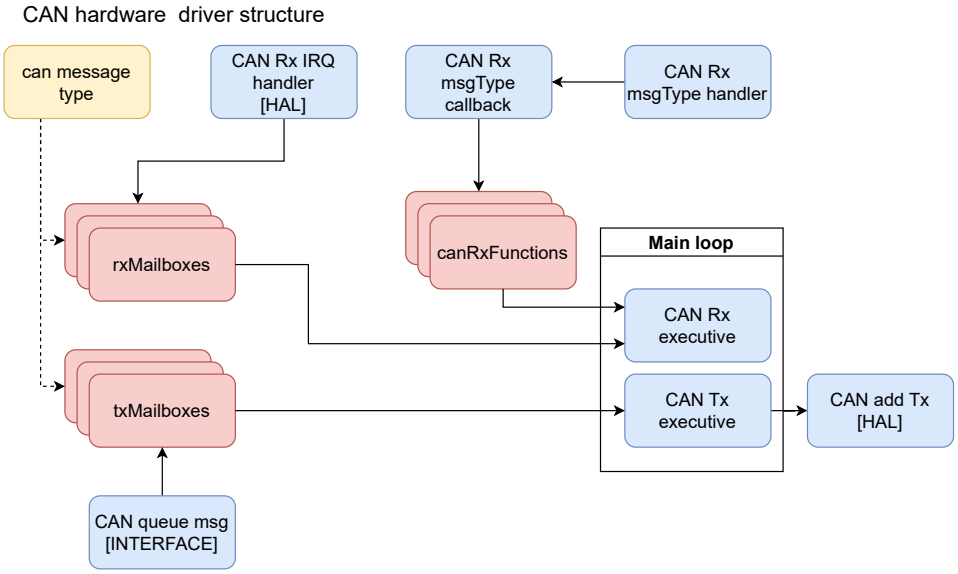


Figure 35: Structural diagram of the CAN driver

10.5.1 HAL interactions

- `HAL_CAN_AddTxMessage`: Submit a CAN message to the peripheral's transmission mailbox queue.
- `HAL_CAN_GetRxMessage`: Returns a message stored in the peripheral's receive queue.
- `HAL_CAN_RxFifo0MsgPendingCallback`: Message received callback function, triggered when the peripheral receives a message.

10.5.2 Component: Types, mailboxes and receive callbacks

Valid CAN message types are defined in an `enum`erated list, forming the backbone of the driver. Message types correspond to the five LSBs of a CAN message ID discussed in section 9.6. A `can_mailbox` contains a boolean `newMsg` flag, an 11 bit message ID and the eight bytes of payload data a CAN message may contain. The driver defines two lists of mailboxes, one for incoming and one for outgoing messages. The number of mailboxes in each list is defined by the number of enumerated CAN message types.

When a message is queued for transmission, it is placed in the mailbox corresponding to its message type. For instance, a `JOINT_POS_SP` (joint position setpoint) message has index number 5, as it was the sixth message type to be defined, and will be placed in transmission mailbox 5. The same happens when the message is received: The CAN peripheral receive handler sees that the 5 LSBs of the message ID corresponds to the number 5, and stores it in receive mailbox 5. This system preserves the priority system of the CAN protocol where low IDs are handled before high IDs, as the list of mailboxes is iterated through in the main loop. When a new message type is registered in the enumerated list, it may be inserted near the top or bottom according to its perceived importance relative to other message types.

Handling message transmission is straightforward: Add the message ID and payload to the relevant mailbox, and raise the `newMsg` flag. Handling reception is less straightforward, as each message type is associated with a unique action. In the interest of standardisation, the driver defines a list of "callback" function pointers similar to the keyword/function pointer list in the string command module (10.2). The length of the list is defined by the number of enumerated CAN messages. Callback function `can_driver_cmd_rx5` will be called when a message is discovered in mailbox number 5, in turn calling the handler function required by message type number 5, `can_cmd_handle_jointSp`, as well as checking that the incoming message number (5LSB of ID) corresponds with `JOINT_POS_SP`. The handler function finally updates the joint's setpoint based on the message payload.

By separating the callback and handler function, *action* associated with a CAN message type is abstracted away from its *number* in the enumerated list. A handler function could be defined to do whatever, using whichever arguments, and need only be registered with the correct callback function.

The following pseudocode presents the CAN driver components discussed so far:

```
typedef enum can_message_type:
    ACC_X_TX,
    //...
    JOINT_POS_SP,
    //...
    num_types;

static can_mailbox rxMailboxes[num_types];
static can_mailbox txMailboxes[num_types];

void (*canRxFunctions[num_types])():
    can_driver_cmd_rx0,
    //...
    can_driver_cmd_rx5,
    //...
    can_driver_cmd_rxN; //where N is num_types

void can_driver_cmd_rx5(ID, DATA):
    //Register joint setpoint action with rx5
    if ID == JOINT_POS_SP:
        can_cmd_handle_jointSp(ID, DATA);
    else:
```

```
        //throw error

void can_cmd_handle_jointSp(ID, DATA):
    //Use ID to determine target joint
    //Use data to determine setpoint
```

10.5.3 Component: Executors and interface

The CAN driver interface, i.e. "public" functions available to the user, presents only two options: queuing a message for transmission by placing it in a mailbox, or immidiately sending it to the CAN peripheral. The latter is intended for cases where the mailbox priority system needs to be circumvented, but has thus far not been used. When the queue function is used, the `newMsg` flag is raised in the associated mailbox, and message ID/data are inserted.

CAN traffic is handled by two functions running in the main loop of each control unit: `can_rx_executive` and `can_tx_executive`. These functions loop through their respective mailboxes, queueing messages for transmission in the peripheral or calling the receive handler functions depending. Both rely on the `newMsg` flag to determine whether a mailbox needs processing. The executors will process all messages before returning to the main loop, suspending other (not interrupt driven) actions in the MCU for the duration. While potentially disruptive, it does ensure that even low priority messages are processeed. Thus far, no indication of disruption from long CAN message processing times have been registered.

Building on the previous pseudocode, and note that the implementation uses getters and setters for all mailbox fields:

```
void can_rx_executive():
    for(i < num_types):
        if(rxMailboxes[i].newMsg):
            inData = rxMailboxes[i].data;
            inID = rxMailboxes[i].ID;

            //Callback and handler are called directly
            canRxFunctions[i](inData, inID);
            rxMailboxes.newMsg = 0;

void can_tx_executive():
    for(i < num_types):
        if(txMailboxes[i].newMsg):
            can_driver_send_msg(txMailboxes[i].data,
                                txMailboxes[i].ID);
            txMailboxes[i].newMsg = 0;
```

10.5.4 Key functionality: Using ID bits to select hardware

As mentioned in section 9.6, the 6 MSBs of the CAN message ID are used to determine which hardware unit is the target of a message type. This mechanism is connected to the use of lists to represent hardware, such as in the motor and ADC drivers.

When an incoming CAN message is processed by its handler, the handler will filter the relevant hardware bits. For instance, the `cand_handle_jointSp` handler will filter the M2 bit and use its value in a call to `joint_controller_update_setpoint` (see 11.1), which takes the index of a joint in that module's "hardware" list as an argument. That is, if the M2 bit is 1, then the action (update setpoint) will be applied to joint 1 of the relevant controller. M0 and M1 bits are used by the CAN peripheral to determine whether the target joint is controlled by the relevant MCU.

10.5.5 CAN message ID filter configuration

A discussion of CAN filter configuration was relegated here from section 8.4 as it is inherently connected to the CAN driver module. The CAN peripheral's filter banks are configured manually in `can.c`, a CubeMX generated file.

The CAN peripheral has 12 configurable filter banks, the purpose of which are to filter out messages irrelevant to the MCU. Of the 12 banks, three were configured: one for each of CAN message categories motor, accelerometer and global. There are two primary parameters for each filter bank: ID and mask.

The filter mask determines which bits of an incoming CAN message ID are relevant, and the filter ID determines what the value of each relevant bit in the incoming message ID must be. When an incoming message ID is evaluated by the peripheral, 0 bits in the filter mask are DC, while 1 bits must match the filter ID. That is, if ID bit 3 of the filter mask is 1 and ID bit 3 of the filter ID is 0, then bit 3 of the incoming message ID must be 0 in order for the peripheral to accept the message.

Filter banks were configured such that for each message category, i.e. bank, 6MSBs not associated with the category must be 0, while other bits are DC. 5LSBs are always DC, as any CAN message type may apply to any hardware category. Filter configuration is displayed in table 11.

	Bank 0: Motor	Bank 1: Accelerometer	Bank 2: Global
Torso ID	0x000	0x000	0x400
Torso mask	0x300	0x0E0	0x7E0
Shoulder ID	0x040	0x000	0x400
Shoulder mask	0x3C0	0x2E0	0x7E0
Hand ID	0x080	0x200	0x400
Hand mask	0x3C0	0x2E0	0x7E0

Table 11: CAN peripheral filter bank configuration

10.5.6 Comment: The number of supported message types

As mentioned in section 9.6, the 6 MSBs of the CAN message ID are reserved for categories corresponding to hardware, and it was mentioned that the system could support 96 message types. However, the current implementation would need some modification to support this number, as message types are defined by their action. For instance, message type 0, `ACC_X_TX` is reserved for transmission of accelerometer data from the X axis. The callback and handler functions use ID bits A0 and A1 to determine which accelerometer the data comes from (figure 29), disregaring remaining four upper ID bits. Attempting to use message type 0, or any other number associated with accelerometer data types, in a joint message handler would throw an error, effectively reducing the number of message types available for joint messages and vice versa.

Separate lists of enumerated message types (5LSB) could have been written for each of the three message categories of global, accelerometer and motor/joint (6MSB) in order to support 96 messages. The current implementation of a single list effectively only supports 32, of which 16 have been implemented. This may prove to be a hindrance to future development, but opens for a tongue-in-cheek paraphrasing of a certain industry figure: *this system will never need more than 32 messages.* Furthermore, a singular list also means that CAN message types are not locked to a system of three categories of messages, should the driver ever need an overhaul.

10.6 ADC driver

The ADC driver is responsible for reading motor current consumption via the ADC peripherals. The driver defines a `current_measurement_descriptor` struct which stores static information about the hardware as well as the most recently read ADC value.

10.6.1 HAL interactions

- `HAL_ADC_GetValue`: Returns the latest converted value as a 16 bit integer

10.6.2 Key functionality: Convert ADC values

As mentioned in section 8.3, the ADC peripherals have been set to continuous conversion, enabling the driver to read a recently updated conversion upon request. This as opposed to requesting and waiting for a conversion to complete. Upon a call to `adc_driver_calculate_current`, the driver will use a the latest ADC reading and a precalculated conversion constant to return the motor current in Ampere (see section 5.5).

Similar to the motor driver, the driver uses a list of current measurement descriptors to access the correct ADC when requesting an updated current measurement.

10.7 GPIO driver

The GPIO driver is responsible for interacting with the the end and twist joint optical switches. The driver defines two handler functions which are called by interrupt routines associated with the relevant MCU pins. The handlers raise flags in the state machine, informing it that the sensors have been triggered.

10.8 ROS UART parser

The ROS UART parser is responsible for parsing UART messages sent from the ROS nodes running on the control computer. Messages enter the parser via the `uart_driver` module, and are placed in a 32 byte message buffer. This module is tailored for the torso control unit, unlike the `string_cmd_parser`. Figure 36 illustrates the module.

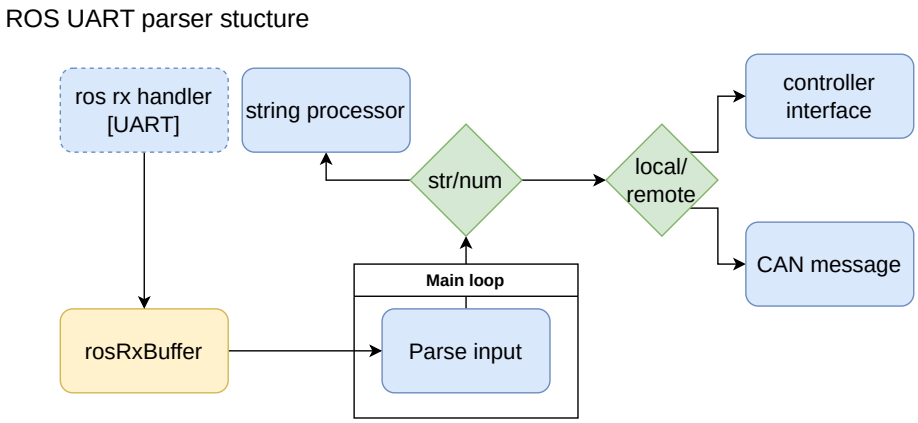


Figure 36: Structural diagram of the ROS UART driver

10.8.1 Component: Messages

The ROS modules output 32 byte messages read by the parser. These messages adhere to one of two structures: `numerical` or `string`. Numerical messages contain joint setpoints generated by MoveIt, while string messages are sent from a text interface and enables usage of the string commands described in section 10.2. The first three bytes are dedicated to a header which contains either `"num"` or `"str"` depending on message type.

Messages generated by MoveIt are referred to by the generic name "numerical" rather than something more descriptive like "setpoint" because future development of the system may expand the ROS modules' functionality beyond passing setpoints. The name was given to distinguish them from human generated string messages.

10.8.2 Key functionality: Parsing messages

10.8.2.1 Numerical messages Numerical messages contain setpoints for all joints except pinch, on the format of 16 bit floating point values. Rail and shoulder joint setpoints are sent to the joint controller module (see section 11.1), while elbow and wrist joint setpoints are sent as `WRIST_ELBOW_SP` CAN messages. A separate CAN message type was created for two reasons: 1) CAN messages are always eight bytes, and sending setpoints separately would waste bandwidth; and 2) sending all joint setpoints as `JOINT_POS_SP` messages would require an additional queueing system as the mailbox would otherwise be overwritten by the last joint setpoint to be parsed.

Setpoints for the twist and pinch joints are sent similarly as a `PINCH_TWIST_SP` message, but require a detour via the string parser module.

10.8.2.2 String messages The string message format was created in order to preserve compatibility with the previously developed string command parser. In part, this was done because rewriting the module sepecifically for ROS communication would have required a significant time investment for little to no new functionality. Most importantly, however, it was done because a misconfiguration of MoveIt prevents it from controlling the pinch joint. Consequently, the pinch joint must be controlled manually via a terminal using the command `pinch stp r <setpoint>`.

When the ROS parser receives a message with the header `"str"`, strips the header and hands the message over to the string command parser module. If the message is a pinch command, the string parser sends the setpoint back the ROS parser, which will include the updated setpoint in the next `PINCH_TWIST_SP` CAN message it sends.

10.8.3 Key functionality: Receive data

When in ROS mode, the UART driver will act for every 32 bytes of data it receives, i.e. handle an interrupt generated by the UART peripheral by raising a `newRosMsg` flag and storing the 32 bytes in the ROS UART module's message buffer. If the message is numerical, the ROS UART module will update setpoints for all joints, *including pinch*, via the CAN bus as described. This implies that the pinch joint cannot be operated when the system is in ROS mode unless MoveIt is also running.

10.9 Unit configuration

The unit configuration "module" is an .h file containing a set of preprocessor definitions. Most importantly, this file sets set the "ACTIVE UNIT" parameter which decides which control unit a binary should be compiled for. The motivation was to collect a set of frequently adjusted project parameters in single file, and collect relevant initialisation functions in a single interface, for easy access as part of the project management. It was not utilised to a great extent, but some important parameters are listed below.

- `PWM_CTR_PRD`: Sets the PWM timer counter period and, implicitly, PWM frequency.
- `ACTIVE_UNIT`: Takes one of three values `TORSO`, `SHOULDER`, `HAND`, must be set before the project is compiled.
- `SW_INTERFACE`: Takes one of two values `CMD_MODE_TERMINAL` and `CMD_MODE ROS`, and determines which input mode is in use. `CMD_MODE_ROS` should only be used with the torso unit.

- `CAN_FILTER_<X>`: A set of definitions where X is either M, A or G[14]. Sets the CAN ID filter for the relevant filter bank/control unit.
- `CAN_FILTERBANK_<X>`: A set of definitions where X is either M, A or G. Sets the CAN filtermask for the relevant filter bank/control unit.

[14]For Motor, Accelerometer and Global, respectively

11 Control system

This section presents the high level modules of the software system: joint controller, state machine and main file.

11.1 Joint controller

The `joint_controller` module controls the joints for which a control unit is responsible. At its core, it runs a PID controller with a joint's position as the process variable and motor power setting as controlled variable. It uses the motor driver interface to adjust the joint's motor power setting. The distinction between a joint and a motor is key: a joint represents the physical connection between two robotic links, and it is actuated by a motor. The joint controller may use a combination of IMU and encoder sensor data to determine the position of a joint, and will use this information to control the relevant motor.

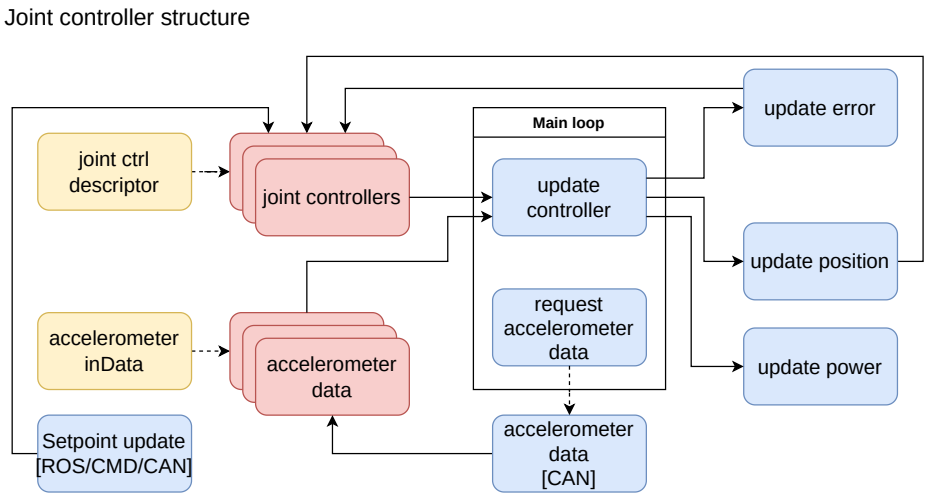


Figure 37: Structural diagram of the joint controller

11.1.1 HAL interactions

The joint controller itself does not interact with the HAL, as it uses the hardware modules to interact with relevant peripherals. However, the callback function for periodically triggered interrupts was defined here these interrupts primarily concern the joint controller module.

- `HAL_TIM_PeriodElapsedCallback`: Sets flags checked by the controller to determine whether a control action may be taken.

11.1.2 Component: Controller descriptor and accelerometer inData

Similar to the motor and CAN drivers, the joint descriptor defines structs `joint_controller_descriptor` and `accelerometer_inData`. Controller descriptors contain information such as its PID parameters, position and whether or not it is associated with an accelerometer. The accelerometer data structs are inspired by CAN mailboxes. They define data fields for each axis of acceleration and rotation as well as associated `newData` flags for each field. Accelerometer data was included in the joint controller module rather than the accelerometer driver in part because no control unit uses data from its own accelerometers, and in part because it is associated with joint control. This decision is further discussed in section 16.

As with the motor driver, joint descriptors are collected in a list of two units, and preprocessor statements determine which of the six descriptors are compiled for a given control unit. Accelerometer data structs are listed depending on how many accelerometers the control unit needs to support.

11.1.3 Key functionality: Setpoint input

The joint controller operates with positional setpoints as its process variable. Setpoints are defined relative to the joint's "start position", which, depending on the method by which the operating state was entered (see section 11.2), will be either whatever position the joint was in at powerup or its calibrated home position (see section 13.1). A joint's position and setpoint are initialised to 0 on powerup. Setpoints may be input to the joint controller automatically via ROS or manually via string command (and distributed via CAN bus, where applicable). When a setpoint has been registered, it will be acted upon the next time the controller is updated (see following subsections).

When updating setpoints manually, it is recommended to use the `<joint> stp r <radians>` string command. A setpoint is absolute; it is always relative to the joint's start position. For instance, passing `shoulder stp r 0.7` after calibration will always cause the shoulder joint to seek a position of 0.7 radians off the vertical axis in the positive direction indicated by figure 1.

WARNING: The joint controller has no velocity control or joint movement limits, and PIDs are generally tuned aggressively. Passing setpoints with deltas greater than 0.5 radians or 300 millimeters may cause severe overshoot and/or instability, potentially harming equipment and/or personnel within movement range.

11.1.4 Key functionality: Flags and joint state update

As described in section 8.8, the timer peripheral has been configured to trigger interrupts at 10kHz and 344Hz. These interrupts raise flags in the joint controller module indicating that it is safe to update the PID loop (10kHz) or use the CAN bus (344Hz). The main loop runs the `controller_interface_update_controller` and `controller_interface_request_acc_axis` functions, which check whether the flags have been raised before performing their respective actions.

`controller_interface_request_acc_axis` queues an accelerometer axis request CAN message for transmission. When the data is received, the associated CAN handler function inserts it into the relevant `inData` struct.

`controller_interface_update_controller` collects three functions: update position, update error and update power.

11.1.4.1 joint controller update position: A joint's position is primarily calculated by converting encoder clicks to radians or meters. In the case of the shoulder joint, the function will use the latest accelerometer data when new data is available. Encoder clicks are counted from a calibrated zero position (see section 13), and the function uses a conversion constant stored in the associated motor descriptor struct, `resolution` defined as clicks per radian. When using accelerometer data, the function will convert the Y axis acceleration measurement to an angle using the `asinf` function of the `math.h` C library. This effectively assumes that the joint is not in motion, and will yield an incorrect result if the shoulder joint is past 90 degrees vertical in either direction.

11.1.4.2 joint controller update error: A joint's error is defined as the difference between its setpoint and measured position. This function is called after the position update, ensuring that the most recent data is used. The function updates the fields `posError` (positional error) and `prevError` (previous error) of the joint descriptor. The latter is needed for calculation of the PID controller D term.

11.1.5 Key functionality: PID controller

The PID controller is implemented in `joint_controller_update_power`. It is called by the controller update function after position and error have been calculated. When the power setting of a motor has been calculated, it writes this to the motor driver interface. The PID controller implements the following equation:

$$P = K_p E + \int_{\tau_0}^{\tau} K_{pti} E dt - K_d \frac{dE}{dt}, \tag{2}$$

where P is the power setting, K_p is the proportional gain, K_{pti} is an almagamation of the traditional $\frac{K_p}{T_i}$ integral gain, K_d is the derivative gain and E is the setpoint error. Note that the integral gain has been placed inside the integration, and see the I term paragraph below.

11.1.5.1 P term: The most recent error is multiplied by the joint's proportional gain.

11.1.5.2 I term: The integral term is calculated by summing previous errors, as usual in a PID controller, but with modifications to prevent overshoot and windup. Windup is prevented by limiting the term to ± 100.

Several attempts were made at tuning the PID regulators to be somewhat aggressive without overshooting for relatively changes in setpoints. This was unsuccessful, and motivated the introduction of a countermeasure to limit integral action until the joint is close to its setpoint. The solution was to apply a sigmoid gain function to the joint's integral gain value:

$$G_{kpti} = 1 - \frac{e^{20|E|-5}}{1 + e^{20|E|-5}}, \tag{3}$$

where G_{kpti} is the sigmoid gain and e is Euler's number. Numerical values 20 and 5 were set experimentally (see section 13) and ensure that the gain slope is appropriately shaped. The final expression for one iteration of the integral error is as follows:

$$I = K_{pti} G_{kpti} E, \tag{4}$$

which is added to the joint descriptor's `intError` field provided it does not exceed 100.

11.1.5.3 D term: $\frac{dE}{dt}$ is defined as the difference between the current and previous error measurements.

In code the final expression for the power setting is as follows:

```
float power = (joint->Kp)*error + (joint->intError) - (joint->Kd)*dedt;
```

In order for the integral and derivative terms to be consistent and meaningful, they must be calculated regularly. This motivated the introduction of the 10kHz interrupt. As the interrupt itself does not trigger an update of the power setting, this frequency is effectively an upper limit and relies on the main loop not taking longer than $500\mu s$ to complete.

11.2 State machine

A state machine was introduced late in the project's development to differentiate between calibration and normal operations. The module defines two sets of states: global and calibration states, where the calibration states should be understood as substates of the global state `GS_CALIBRATING`.

States affect all control units, hence the name "global", but they may exhibit different behaviour for a given state.

When the arm is powered, the state machine enters the `GS_IDLE` state. In this state, the arm will not respond to movement orders, but will register encoder clicks if the arm is manipulated manually. Transition to the `GS_CALIBRATING` state is triggered when the user inputs the string command `home` in a terminal. Transistion to any global state may be triggered at any time with the command `state <state>`. The state machine is illustrated in figure 38. Blue bubbles are global states, yellow bubbles are calibration states.

WARNING: If the arm has been manipulated between powerup and a triggering of a state transition from `GS_IDLE` to `GS_OPERATING`, **the arm may jerk violently** as it will seek to move all joints to their default "setpoints" of 0 relative to their positions at powerup. This may cause harm to the arm and personnel within movement range.

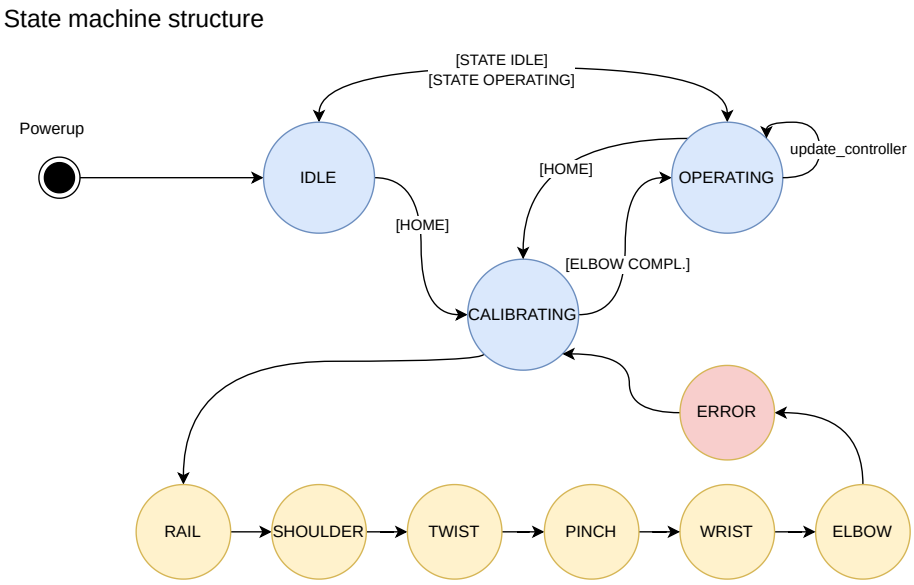


Figure 38: Structural diagram of the state machine

11.2.1 Component: States

- `GS_IDLE`: Idle state. Text interface command: `state idle` or `s`
- `GS_CALIBRATING`: Calibration state. Text interface command: `state calibrate` or `home`
- `GS_OPERATING`: Operational state: Text interface command: `state operate`
- `GS_ERROR`: Global error state. Not implemented
- `CS_ERROR`: Calibration error state invalid/irrelevant
- `CS_RAIL`: Rail joint calibration
- `CS_SHOULDER`: Shoulder joint calibration
- `CS_ELBOW`: Elbow joint calibration
- `CS_WRIST`: Wrist joint calibration
- `CS_TWIST`: Twist joint calibration
- `CS_PINCH`: Pinch joint calibration

11.2.1.1 Global state: IDLE

In the idle state, the arm will not update the joint controller, but is otherwise operational: It will update encoder counts and accelerometer readings, and the string command interface is available. Importantly, it is possible to update setpoints for the controller, i.e. set them to something other than the default of 0. If a transition directly to the operating state is forced after setpoints have been updated, the arm will immediately seek these setpoints. The same will happen if the arm has been manipulated manually before state transition is forced; it's default setpoint is 0, and the controller will discover that it has been moved away from this position.

11.2.1.2 Global state: OPERATING

The operating state is the nominal state in which regular operation should occur. The meaningful difference from the idle state is that the joint controller update function is active in the main loop.

11.2.1.3 Global state: CALIBRATING

In the calibration state, the arm will perform an automated calibration sequence, moving through the calibration states until all joints are in a known position. The control units will exit the main loop during calibration of its own joints, and are thus unavailable for interaction: CAN executors will not be running, and string command/ROS input may only be partially handled. It is therefore not recommended to interact with the arm during calibration, and care should be taken to disable the ROS serial communication node before calibration. The only safe way to interrupt the calibration procedure is to disable the arm's power supply.

Calibration states are discussed in section 13.1.

11.2.2 Key functionality: Broadcasting states

As discussed in section 9.6, control units are peers. Any control unit may set the system's global or calibration state via the state machine interface functions
`state_interface_broadcast_global_state` and
`state_interface_broadcast_calibration_state`. These functions queue a CAN message type `GBL_ST_SET`, where the message payload may contain one or both of the global and calibration states.

In practice, most global state updates will be handled by the torso unit as the user may opt to set the global state manually. During the calibration phase/states, however, a control unit will calibrate its own joints before setting the calibration state to the next joint in the calibration sequence (see fig 38).

11.2.3 Key functionality: Enabling calibration

As mentioned, the purpose of the state machine is to calibrate the joints before allowing operation. When the system enters the global `GS_CALIBRATING` state, calibration state will immedeately be set to `CS_RAIL`. Calibration procedures have been written for each joint in functions called `state_calibrate_<joint>`, and generally involve moving them to a known position (see section 13.1 for details). In the case of the rail joint, it will move until the end stop is triggered.

While a joint is calibrating, other joints will remain still. When a joint has finished its calibration procedure, the controller sets the calibration state to the next joint in the sequence outlined in figure 38. When calibration of the elbow is complete, the system is set to calibration state `CS_ERROR`, and the global state is set to `GS_OPERATING`.

11.3 Main file

This section describes the `main.c` file which can be found in the `stm_config_official/Core/Src` directory. The file was initially generated by CubeMX, but has been modified to fit the project. In addition to the main function, it contains CubeMX generated functions such as `SystemClock_Config`. They are defined below the main function, and have not been altered or intentionally used in this project. As is typical, the main function is divided in two sections: system initialisation to run once, and the "main loop" which repeats indefinitely. The main function is illustrated in figure 39. Blue bubbles represent a function or set of functions, and arrows represent the order in which (sets of) functions are called.

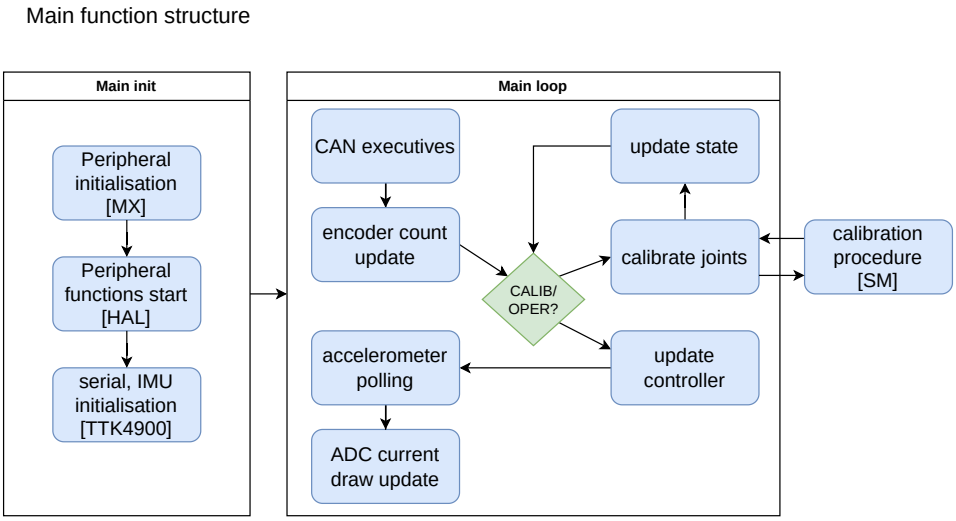


Figure 39: Structural diagram of the main function

11.3.1 Initialisation

Initialisation happens in three stages. First, a set of CubeMX generated functions on the format `MX_<peripheral>_Init` initialise peripherals according to the configuration settings described in section 8. These have not been altered, with the exception of `MX_CAN_Init` in which the CAN filter banks were configured. Second, a set of HAL library functions are called to begin peripheral operations, such as `HAL_TIM_Encoder_Start` which triggers the called encoder timer to begin counting encoder ticks. These calls were added manually, usually found using online sources and confirmed in UM1786[15]. Finally, the serial interface is initialised to depend either on ROS or pure string commands, and in the case of the shoulder control unit the accelerometers are initialised as discussed in section 10.4 using the relevant driver interface functions.

11.3.2 Loop

Loop items are primarily discussed in their respective subsections of sections 10 and 11. The order of operations inside the loop should have little impact on its operation, as a key principle in the development of the system was that modules should be independent. At worst, information will be one loop iteration "behind" by the time it reaches a given step. For instance, updating the encoder count after immediately after, rather than immediately before, a controller update would ensure that encoder information is one full loop out of date by the time it is used in the joint controller.

The green square in figure 39 represents the check to determine system state. If the state is `GS_IDLE`, none of the actions succeeding it will occur, but the loop instead jumps directly to accelerometer polling. Calls to calibration procedures make up the bulk of the main loop, and is

divided into one section for each control unit, selected by the `ACTIVE_UNIT` preprocessor flag. The pseudocode below presents a condensed version of this. Note the difference between preprocessor and C language if statements, and that elbow is in fact the last joint to be calibrated. The order of presentation, i.e. TORSO, SHOULDER, HAND, is chosen because it is physically correct with with regards to the controllers' position within the arm.

```
if(global_state == GS_CALIBRATING):
#if ACTIVE_UNIT == TORSO
    if(calib_state == CS_RAIL):
        state_calibrate_rail();
        state_interface_set_calibration_state(CS_SHOULDER);
        state_interface_broadcast_calibration_state();

    else if(calib_state == CS_SHOULDER):
        //calib shoulder, set+broadcast TWIST
#elif ACTIVE_UNIT == SHOULDER
    //repeat pattern for elbow and wrist joints
    if(...):
        //elbow calibrated -> calibration complete
        state_interface_set_global_state(GS_OPERATING);
        state_interface_broadcast_global_state();
#elif ACTIVE_UNIT == HAND
    //repeat torso pattern for twist and pinch joints
#endif
    else:
        //MAINTAIN joint position while others are calibrating.
        //By this point, "0" will have been set to calibrated
        //home positions for the controller's joints.
        controller_interface_update_controller();

else if(global_state == GS_OPERATING):
    controller_interface_update_controller();
```

The `controller_interface_update_controller` function is called every loop, but has an internal check on the flag raised by the 10kHz timer interrupt.

Accelerometer polling happens towards the end of the loop, and is handled by the torso. If the 344Hz CAN bus flag has been raised, it will queue a request for the shoulder control unit's accelerometer Y axis. The request will be sent near the beginning of the next loop by the CAN TX executor, and handled by the shoulder control unit's CAN RX executor.

Not present in figure 39[15] is a code snippet checked by the 50Hz UART telemetry flag, just below accelerometer polling. An attempt was made at making the torso control unit output the position, setpoint and current of the shoulder joint, in the interest of aquiring quantitative test data. It was moderately successful, and is briefly discussed in section 14.

[15]..and really only relevant if the reader is actually concerned with code

12 ROS nodes

URDF

Velocity limitation as a safety precaution.

13 Calibration

Method of calibration, usage in the state machine.

13.1 Joint position calibration

13.2 PID tuning

PID tuning: voltage levels, agressive and shaky vs slow and steady, stability/convergence for a given deltaP, and how that relates to the velocity limits imposed on MoveIt, didn't use Jo's suggested method as it was impractical, Ziegler-Nichols barely relevant for a system as nonlinear as this and couldn't really be applied due to space at the lab,

13.3 Encoder clicks per radian and motor polarity

13.4 IMU axis offset

14 Tests

Message round trip times: CAN, I2C, accelerometers=(CAN+I2C)

PID tuning: Sigmoid vs other methods such as zeroing I term when passing setpoint.

Bottle carrying.

My one quantitative data point: triggered a crash.

OV robodranks

15 Results

Note: ROS functionality was implemented at the expense of telemetry output from the arm,

Regret not putting indicator LEDs on GPIO, would have made basic testing easier. The optical sensor shenanigans could have been avoided if the original sensors had been studied more carefully. State machine is whack, inconsistent calibration for twist for whatever reason Accelerometers are whack, should have had proper debug headers System works well overall when accelerometers are disabled ROS is really powerful, and can probably be expanded readily CAN bus craps out for voltages above 25V

The decision to place the upper arm IMU on the shoulder control unit PCB made it impossible to debug, and may be considered a violation of the project's second goal (or requirement number NUMBER) as the whole board will need to be replaced in order to repair/replace the IMU.

USB non-functional. Could it be due to the clamping voltage in the rail TVS? Or not implementing VBUS ESD protection properly, see an4879 2.3.

Input voltage no more than 25V, far less than the suggested 48V. Heat, motor control, CAN bus.

Low coupling between modules? Not really, look at the include graph

Were the ARs adhered to? More or less

I have no way to know to what extent the interval scheduling actually works as intended. Had I had a debugger I might've tried something like setting breakpoints in the interrupt handler and main loop and see which triggers first, and done that a couple times to test the hypothesis that "the mcu goes real damn fast compared to everything else".

Impact of main loop organisation on accelerometer data delay

16 Discussion

Pros and cons of using youtube as a source: bootstrap large project, making other people's mistakes.

Not having access to a debugger was frustrating

DMA and peripheral interconnect: could using them have increased i2c or can message rates?

FreeRTOS: would be cool, and relevant with the amount of nonsynchronous stuff going on.

CubeMX, despite landing solidly in the category of "corporate shitware" was easy to use and saved a lot of time - not a chance I would have made it to ROS if I had had to flip all them bits.

Choosing UART5 instead of literally any other USART port may have limited communication options.

Everything relating to handling of the accelerometer inData structs in the joint controller module should have been in a separate accelerometer controller module. This is a gross violation of AR1.2. Additionally, the acc inData could have just been a number of lists with known offsets instead of 12 individual fields. Might have also reduced the number of aux functions – which is ridiculous anyway. A separate acc controller module may also have made it easier to implement functionality to allow for multiple accs to be used when estimating position of a single joint. Something something kalman filter. FURTHERMORE, the timer interrupts are basically defined as "limit accelerometer/motor polling frequency" interrupts rather than "limit CAN bus usage" which is what they're actually for. Should also have been its own module.

Cause for bad accelerometer behaviour: not only is the data updated at 1/30th the rate of the joint controller, there is probably a non-insignificant delay between measurement and processing. measurement -¿ CAN tx queue -¿ CAN trx -¿ CAN rx queue -¿ acc mailbox -¿ pos update. A separate module would have made it easier to ensure data was recent, maybe. Also jolt, or any non-stationary reading, will affect the angle measurement.

CAN message IDs are actually wasting one bit on determining recipient MCU for both accelerometer and motor messages. Had the system followed an adress system instead of being centered on a specific piece of equipment, two bits could have determined the recipient MCU, one bit could have determined the equipment type, and one bit could have determined equipment number. This would require a bit more logic when parsing a message ID in the CAN driver than the current implementation, but much more efficient and expandable. Accelerometer data polls are essentially triggered by the torso. This must be insanely inefficient compared to making the shoulder poll every so often and simply send its latest (provided it is not old/stale) upon request?

17 Conclusion

18 Operations

18.1 PCB installation

18.2 MAIN connector usage

18.3 Gripper

18.3.1 Installation

18.3.2 Manufacture

18.4 Control software usage

Bibliography

[1] Adafruit. *Adafruit MMA8451 Accelerometer Breakout*. 2023.

[2] Kristian Blom. 'From hardware to control algorithms: Retrofitting a legacy robot using open source solutions'. MA thesis. The Norwegian University of Science and Technology (NTNU), 2023.

[3] Kristian Blom. *TTK4900 project repository*. URL: https://github.com/kristblo/TTK4900_Masteroppgave (visited on 23rd May 2024).

[4] Bourns. *CDSC706-0504C - Surface Mount TVS Diode Array*. 2015.

[5] TT Electronics. *Photologic® Slotted Optical Switch OPB960, OPB970, OPB980, OPB990 Series*. 2019.

[6] Escap. *escap 23LT12 graphite/copper commutation systems*. 2024.

[7] DIODES incorporated. *ZMR series: fixed 2.5, 3.3 and 5 volt miniature voltage regulators*. 2013.

[8] Pittman Metek. *Brush commutated DC motors DC030B Series*. 2024.

[9] Pittman Metek. *Brush commutated DC motors DC040B Series*. 2024.

[10] Pittman Metek. *Brush commutated DC motors DC054B Series*. 2024.

[11] NXP. *TJA1057 High-speed CAN transceiver*. 2023.

[12] ST. *AN4879: Introduction to USB hardware and PCB guidelines using STM32 MCUs*. 2023.

[13] ST. *LSM6DSM iNEMO inertial module: always-on 3D accelerometer and 3D gyroscope*. 2017.

[14] ST. *ST RM0316: Reference manual STM32F303xB/C/D/E, STM32F303x6/8, STM32F328x8, STM32F358xC, STM32F398xE advanced Arm®-based MCUs*. 2024.

[15] ST. *ST UM1786: Description of STM32F3 HAL and low-layer drivers*. 2020.

[16] ST. *STM32F303xD STM32F303xE*. 2016.

[17] ST. *UM1724: User manual STM32 Nucleo-64 boards (MB1136)*. 2020.

[18] Avago Technologies. *HEDS-9000/9100 Two Channel Optical Incremental Encoder Modules*. 2016.

[19] TexasInstruments. *DRV8251A 4.1-A Brushed DC Motor Driver with Integrated Current Sense and Regulation*. 2022.

[20] TexasInstruments. *LMx17HV High Voltage Three-Terminal Adjustable Regulator With Overload Protection*. 2015.

Appendix

A Hello World Example

```cpp
int main {
    // This is a comment
    std::cout << "Hello World from C++!" << std::endl;
    std::cout << "I am using the default style to print this code in beautiful
    ↪  colors. Since the text is so long I have to include the 'breaklines'
    ↪  option as well" << std::endl;
    return 0;
}
```

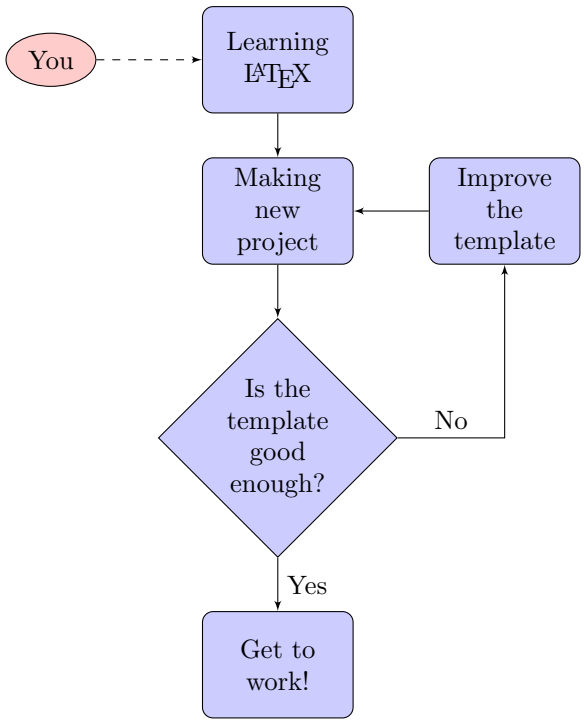
⋮

```python
# This is a comment
print('Hello world from Python!')
print('I am using the "rrt" style to print this code in beautiful colors')
```

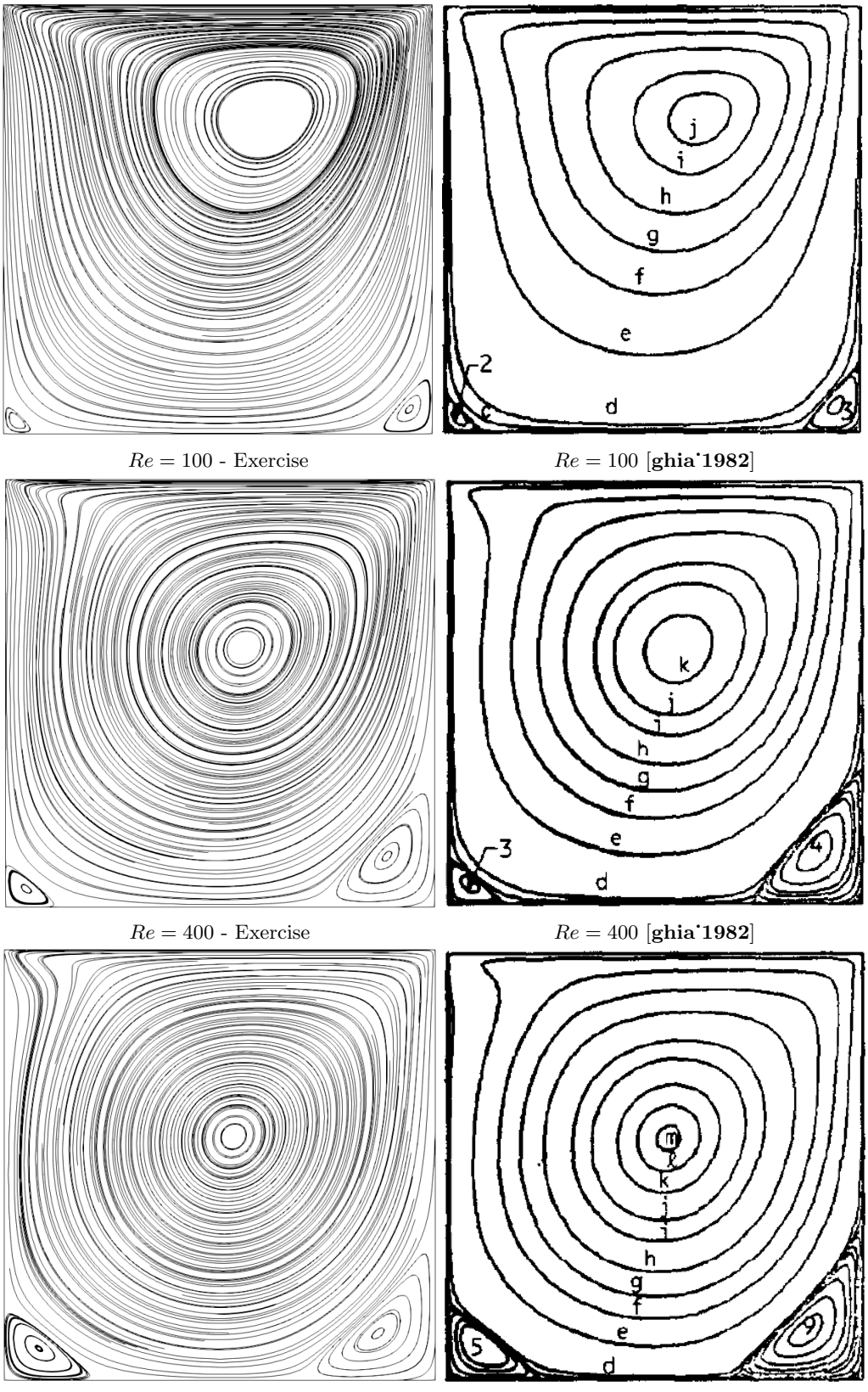
⋮

```matlab
% This is a comment
disp("Hello World from MATLAB!");
disp("I am using the "tango" style to print this code in beautiful colors");
```

B Flow Chart Example



C Sub-figures Example



$Re = 100$ - Exercise $Re = 100$ [**ghia1982**]

$Re = 400$ - Exercise $Re = 400$ [**ghia1982**]

$Re = 1000$ - Exercise $Re = 1000$ [**ghia1982**]

Figure 40: Streamlines for the problem of a lid-driven cavity.