Table of Contents

List of Figures

List of Tables

1 Acknowledgements

Special thanks go out to

Jo Arve Alfredsen, for providing guidance throughout this project's execution and, of course, for agreeing to supervise it in the first place;

Aksel Lunde Aase, for proofreading and providing feedback on the structure of this report so that I could make it even longer;

Friends and colleagues at Omega Verksted, for tips and insights into the wonders of PCB design as well as interpreting oscilloscope squiggles (no, I still haven't tested the capacitance of my lab); and

Omega Verksted itself, for sponsoring the majority of this project!

2 Abstract

Embedded systems engineering is a field which lends itself to a multitude of applications. Whether clothes are being washed or an unstable jet is kept in level flight – in the modern world, most processess are to some extent handled by computers. The multitudes of hardware and software systems developed for these applications are often both cheap and readily available to hobbyists, further expanding the application set: curious projects which have little tangible value beyond the sense of accomplishment they invoke in the projects' proprietors.

This project sought to restore functionality to a defunct 6DOF robotic arm donated to a student driven workshop at NTNU, Omega Verksted, which specialises in curious hobby projects. Inspired by the plethora of robotic bartenders present in establishments across the world, the goal set for this project was that the arm should be able to move with such a strength and accuracy that it may aid users in the mixing of refreshments, and that the control system be simple enough to use by someone without a background in robotics. Additionally, the software should be flexible enough that it may be expanded to other use cases.

Three hardware control units were designed around the STM32F303 microcontroller such that they may each interface with two DC motors and relative encoders, as well as collect and process inertial data from LSM6DSM inertial measurement units by I2C. Units communicate internally via CAN bus, and externally via RS232/UART.

Software modules were developed to run on the microcontrollers, implementing a PID controller for robotic joint positional control via motor voltage. Development heuristics such as a system requirement specification and the SOLID principles were utilised in order to achieve coherence and maintainability. Kinematics is handled by MoveIt, a subsidiary of the Robotic Operating System, which runs on a desktop computer communicating with the arm control units via UART.

Results indicate that the system is marginally compatible with the intended use case. Objects with a short axis length of up to 75 mm and mass of up to 0.8 kg may be manipulated to a precision error of approximately 10 mm/10°. The system did not reach the desired degree of autonomy. While simple path planning is available (i.e. undertaking a pose change $A \rightarrow B$), task planning (i.e. $A \rightarrow B \rightarrow C$) is not. However, the system appears to be readily expandable for such functions. Additionally, substantial issues were had with the use of inertial data; both hardware and software will need further work if strength and precision are to be increased.

3 Oppsummering

Innvevde datasystemer er et ingeniørfaglig felt med stort bruksområde. Enten man vasker klær eller holder et ellers ustabilt jetfly horisontalt innebærer de fleste prosesser i dagens verden bruk av en datamaskin. De mange maskin- og programvareløsningene som utvikles til disse prosessene er ofte både billige og lett tilgjengelige for hobbymakere, noe som utvider feltets bruksområde ytterligere: artige prosjekter uten åpenbar verdi ut over det læringsutbyttet de tilfører prosjektenes innehavere.

Dette prosjektet tilstrebet å reparere en 6DOF-robotarm som ble donert til et studentdrevet verksted på NTNU, Omega Verksted, hvis spesialitet kan sies å være artige hobbyprosjekter. Inspirert av de mange robotbartenderne som finnes i barer verden over ble det satt som mål at armen skulle kunne bevege seg med slik styrke og nøyaktighet at den ville behjelpelig til å blande drinker, og at kontrollsystemet skulle være enkelt nok til at det kunne tas i bruk av folk uten en faglig bakgrunn i robotikk.

Tre kontrollenheter ble utviklet rundt mikrokontrolleren STM32F303, med grensesnitt slik at hver enhet er i stand til å styre to likestrømsmotorer med tilhørende relative enkodere, samt innhente og prosessere data fra treghetsmålere av typen LSM6DSM over I2C. Enhetene kommuniserer med hverandre over CAN-buss, og med eksterne enheter via RS232/UART.

Programvareenheter som implementerer en PID-regulator for styring av leddposisjoner via motorspenning, ble utviklet for å kjøre på mikrokontrollerne. Under uviklingen ble verktøy slik som SOLID-prinsippene og systemkravspesifikasjon tatt i bruk med formål om å oppnå vedlikeholdbarhet og standardisering på tvers av systemmodulene. Kinematikk håndteres av MoveIt, et program under Robotic Operating System-paraplyen, og kjører på en ekstern datamaskin som kommuniserer med armen over UART.

Resultatene indikerer at systemet er marginalt kompatibelt med det forespeilede bruksområdet. Objekter med en lengde opp mot 80 mm langs den korte aksen og med en masse opp mot 0.8 kg kan manipuleres til en presisjon på 10 mm/10°. Systemet ble ikke funnet å ha oppnådd den ønskede graden av autonomi. Planlegging av enkle bevegelser er tilgjengelig (altså en positurendring $A \rightarrow B$) men oppgaveplanlegging (altså $A \rightarrow B \rightarrow C$) er det ikke. Systemet viser derimot stort potensiale for å kunne utvides med slik funksjonalitet. I var det store problemer tilknyttet bruken av treghetsdata, og både maskin- og programvare behøver videre arbeid dersom presisjon og styrke skal kunne forbedres.

4 Abbreviations and glossary

4.1 Abbreviations

Abbreviations are presented in the order of appearance.

Abbreviation	Meaning
OV	Omega Verksted
DOF	Degrees of freedom
DC	Direct current Alt.: Don't Care
ORCA	Optimised Robot for Chemical Analysis
IC	Integrated Circuit
PCB	Printed Circuit Board
ROS	Robotic Operating System
IMU	Inertial Measurement Module
ADC	Analog to Digital Converter
MCU	MicroController Unit
PWM	Pulse Width Modulation
ST	STMicroelectronics N.V.
CAN	Controller Area Network
RM	Reference Manual
IDE	IDentifier Extension Alt.: Integrated Development Environment
DLC	Data Length Code
RTR	Remote Transmission Request
pt	Part
cp	Chapter
UM	User Manual
PID	Proportional Integral Derivative
OS	Operating System
Distro	Distribution
API	Abstract Programming Interface
URDF	Unified Robot Description Language
CAD	Computer Aided Design
OOP	Object Oriented Programming
SRP	Single Responsibility Principle
OCP	Open-Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Principle
DIP	Dependency Inversion Principle
	cont. next column

	cont. from prev. column
UART	Universal Asynchronous Reciever-Transmitter
USB	Universal Serial Bus
TVS	Transient Voltage Supressor
I2C	Inter-Integrated Circuit
GPIO	General Purpose Input-Output
VBUS	Voltage Bus
DP	Data Plus
DM	Data minus
AN	Application note
CANL	CAN Low
CANH	CAN High
LED	Light Emitting Diode
SDA	Signal DAta
SCL	Signal CLock
HAL	Hardware Abstraction Layer
DAC	Digital to Analog Converter
DMA	Direct Memory Access
NVIC	Nested Vector Interrupt Controller
AL	Abstraction Layer
EXTI	EXTernal Interrupt
HSE	High Speed External
PLL	Phase-Locked Loop
AR	Architectural Requirement
GUI	Graphical User Interface
HMI	Human-Machine Interface
LSB	Least Significant Bit(s)
MSB	Most Significant Bit(s)
ZN	Ziegler-Nichols
LTI	Linear Time Invariant
LGA	Land Grid Array
CG	Centre of Gravity

Table 1: Table of abbreviations

4.2 Glossary

Glossary items are presented in the order of appearance

Phrase	Meaning
Embedded system	A computational system tailored for interaction with a mechanical system
(Timer) Count	The value of a timer peripheral's TIM_CNT register
N-N	Comm. bus: Multi-master, multi-slave
1-N	Comm. bus: Single-master, multi-slave
Message	Information packet, usually CAN frame
Python	Interpreted object oriented programming language
C++	Compiled object oriented programming language
Node	Unit in a network, usually ROS
Interface	The set of pathways through which a module may be interacted with
C	Compiled procedural programming language
Module	Part of the system responsible for a defined set of tasks
(The) System	All software and hardware modules involved with control of the arm
Joint	Mechanical connection between two sections of the arm
Non-expert	Person without intimate knowledge of the system
Expert	Person with intimate knowledge of the system
Mk1.1	PCBs developed for this project
Mk1	PCBs developed for the specialisation project
Duty cycle	Percentage of PWM period in which the signal is high
Optical sensor	Optical switch, triggered when aperture is blocked
Assembly	Hardware module; Circuit associated with one (type of) IC
ST-LINK	Hardware unit for flashing and debugging STM MCUs
STM32F303RE	MCU model which this project is centred around
FreeRTOS	An open source Real Time Operating System for MCUs
Pinout	The mapping between an IC's pins and its functions
PuTTy	Serial user interface program
Doxygen	Code documentation generation software
Pattern	A set of software development heuristics
Resolution	The relationship between encoder count and joint movement distance

Table 2: Glossary

4.3 Operational notes

There are operational notes spread around the report **relevant to usage** of the system. These are graded by levels of severity, and may be recognised by bold, capital letters followed by a colon.

NOTE: Item is relevant to correct system usage, and a failure to heed may result in unexpected system behaviour.

CAUTION: Failure to heed may result in moderate to severe damage to the system.

WARNING: Failure to heed may result in moderate to severe damage to the system, and/or may result in moderate to severe injury to personnel.

5 Introduction

5.1 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.

5.2 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 project was conceptualised in 2021, when a defunct Beckmann Coulter ORCA robotic arm was donated to OV. As is often the case with such donations, very little context or information was available beyond the markings on the arm and its associated control computer. Research indicated that it was from the mid 1990s, and originally controlled by software intended to run from Windows 3. Various attempts were made at contacting the arm, but none were successful. At one point it was concluded that the control computer was faulty. Thus, an idea formed: why not redesign the control hardware with modern components, using the old hardware as a "blueprint"? ICs found in the old units were mapped to gain an understanding of how the old control loop would have worked, and it was found to be (at least superficially) relatively simple. The arm was equipped with load cells, inertial measurement units and six DC motors with relative encoders – one for each joint. Additionally, a two-wire bus ran the length of the arm connecting processing units. Time went by without much progress, but during the spring semester of 2023 it was decided that the project was appropriate for a specialisation project, and, if results were promising, likely a master's project. The specialisation project culminated in 7 PCB designs to match the functionality and form factor of the orginials.

As with the specialisation project, the motivation behind this master's project is the desire to combine embedded systems development elements in a wholesome, cohesive engineering project with a specific real-world use case. Where the specialisation project focused on hardware development, this project focuses on software development and to some extent robotic control.

Development of software for this project builds on experiences made from previous projects in both hobby and professional settings. For instance, the choice to rely on open source solutions as far as possible stems from experiences with licenced software as unintuitive and with limited options for support outside of specialised fora – in addition to licenses tending to expire or be locked to individual user accounts. This is impractical in the setting of a student driven workshop catering to a wide set of experience levels and preferences. The hope for this project is that it will be a long lived and fun addition to OV's portfolio, with a barrier to entry as low as reasonably practicable.

5.3 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 motivating this goal is the mixing of beverages during informal meetings at Omega Verksted (OV), that is, meetings with no specific scholastic

or administrative goal. 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.

Scope 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.

5.3.1 Project phases

The project may be divided in three phases, outlining the structure of this report.

5.3.1.1 Hardware improvement: Implementation of the improvement specification from the specialisation project report. This phase includes a separate verification process, the results of which lay the foundation for software implementation. Covers scope item 1. Relevant sections are:

7.2 Hardware description

9 Hardware design/implementation

9.9 Hardware verification

5.3.1.2 Software development: Development and implementation of software in adherance to the specification presented in section 7.3, covers scope items 2 and 3. Software may be categorised according to four abstraction levels:

AL1: MCU peripheral configuration, section 10

AL2: Hardware drivers, section 12

AL3: Control system, section 13

AL4: Kinematics (ROS), section 14

5.3.1.3 Test and verification: Evaluation of the implemented system, covers scope item 4. Relevant sections are:

16 Tests

17 Results

6 Theory

6.1 Circuit design

6.2 Angular measurements

An LSM6DSM inertial measurement unit (IMU) carrying an accelerometer was used to make joint angle estimates for one joint. Angles were measured using output from a single axis parallel with the horisontal plane. Acceleration measurements are output as a 16 bit signed integer[45] such that minimum and maximum values represent the measurement range of the accelerometer, here $\pm 2G$. Assuming that a joint is standing still while the measurement is made, the angle relates to measured acceleration by the following equation:

$$\alpha = sin^{-1}(\frac{a_m + O}{a_M}), \tag{1}$$

where a_m is the measured output value, a_M is the output value which would equate to $1G$ of acceleration, and O is a calibration offset value accounting for imperfections in IMU installation.

Relative encoders were used to make joint angle estimates for all joints. Joint angles were measured using the following relation:

$$\alpha = \frac{n_e}{R}, \tag{2}$$

where n_e is the number of encoder clicks registered ("counted") since the count was last set to zero, and R is the "resolution", a conversion value unique to each joint with the unit $\frac{clicks}{rad}$ for rotational joints and $\frac{clicks}{mm}$ for linear joints.

6.3 ADC voltage conversion

Analog to Digital Converters (ADC) were used to measure current consumed by motors via a proportional current output pin on the DRV8251A motor drivers. This proportional current (I_P) was diverted through a resistor (R_{IP}), and ADCs were set to measure the resultant voltage (V_{IP}). The proportional current is a constant A_{IP} relation with the unit $\frac{\mu A}{A}$. This yields the following relations:

$$I_P = I_m A_{IP} \xrightarrow[law]{Ohm's} V_{IP} = R_{IP} I_P \implies I_m = \frac{V_{IP}}{R_{IP} A_{IP}}, \tag{3}$$

where I_m is the current consumed by the motor. Furthermore, the converted ADC value is proportional to the microcontroller (MCU) analog reference voltage V_{REF}, resulting in the following relation between ADC output, measured voltage and motor current:

$$V_{IP} = V_{REF} \frac{n_{ADC}}{N_{ADC}}, \tag{4}$$

where n_{ADC} is the converted ADC value and N_{ADC} is the maximum ADC output value. Finally, inserting equation 4 into 3, the motor current is given as:

$$I_m = \frac{n_{ADC}}{N_{ADC}} \frac{V_{REF}}{R_{IP} A_{IP}} \tag{5}$$

The ADC reaches saturation ($n_{ADC} = N_{ADC}$) when the measured voltage is equal to V_{REF}. The resistor value R_{IP} was chosen such that the highest measurable current I_{SAT} would equal that of the motor driver's rated maximum current (datasheet[55] cp 7.5):

$$R_{IP} = \frac{V_{REF}}{I_{SAT} A_{IP}} = \frac{3.3V}{4.1A \cdot 1575 \frac{\mu A}{A}} = 511\Omega \tag{6}$$

Incidentally, as the ADC is 12 bit (RM0316[46] cp 15) and therefore $N_{ADC} = 4096$, current consumption per ADC value is almost precisely 1mA.

6.4 Timer configuration

Timers are used for pulse width modulated (PWM) signal generation, encoder input registration and the generation of interrupts at regular intervals. Information presented here is collated from a tutorial written by an ST employee[34], ST RM0316 cp 21.3.9 and 20.3.21[46], and a ControllersTech blog post[53]. All timers are configured to count upwards.

There are four registers relevant to timer configuration in this context:

- `TIMx_CNT`: Timer counter value, indicator of how much time has passed since the count started.
- `TIMx_PSC`: Prescaler, the rate at which the timer count increases relative to the system clock.
- `TIMx_ARR`: Auto reload, the count value at which the count is reset.
- `TIMx_CCRx`: Capture compare register, the value at which some action is taken. The x in CCRx denotes channel number of TIMx.

"Timer frequency", i.e. the frequency at which the timer count is updated is given by the following equation:

$$f_{TIM} = \frac{clk}{TIMx\_PSC + 1}, \tag{7}$$

where `clk` is the system clock frequency. Unless a particularly slow timer is needed, `TIMx_PSC` is generally set to zero.

6.4.1 PWM generation

PWM frequency is decided by the `TIMx_ARR` value:

$$f_{PWM} = \frac{f_{TIM}}{TIMx\_ARR} \tag{8}$$

PWM duty cycle for a given channel is then determined as such:

$$DT = \frac{TIMx\_CCRx}{TIMx\_ARR} \tag{9}$$

In the case of PWM output, channel output will be high until `TIMx_CNT` reaches the value of `TIMx_CCR`. This triggers the output to go low until `TIMx_CNT` reaches `TIMx_ARR`, at which point the count is reset and the process starts again.

PWM is used to control motor drivers by outputting generated signals from two channels. Channels are set reciprocally to obtain the desired power setting:

$$DT_{ch2} = 1 - DT_{ch1} \tag{10}$$

6.4.2 Interrupt generation

Interrupt frequency is determined by setting the `TIMx_ARR` and `TIMx_PSC` registers, similar to equation 8:

$$f_{INT} = \frac{clk}{(TIMx\_PSC + 1)TIMx\_ARR} \tag{11}$$

The interrupt will be generated when `TIMx_CNT` reaches the value of `TIMx_ARR`, provided that a global interrupt has been enabled for the relevant channel.

6.4.3 Encoder input

When in encoder mode, `TIMx_CNT` is driven by the input of two MCU pins, upwards/downwards count determined by the input state (RM0316 table 120). `TIMx_CNT` will be set to zero when it passes `TIMx_ARR` counting upwards, and to `TIMx_ARR` when it passes zero counting downwards. `TIMx_ARR` is set to its maximum value (16 or 32 bit) in this mode.

6.5 Communication protocols

6.5.1 Controller Area Network (CAN)

The STM32F303 MCU supports the CAN standard versions 2.0A and B (RM0316 cp 31). The CAN protocol is an N-N bus centred around the transmission of data frames consisting of a header with message ID, meta information such as number of data bytes and ID format, and a payload of up to eight data bytes. Arbitration is handled on the hardware level such that a message with a low ID number is prioritised over one with a high ID. CAN version 2.0A defines the message ID as 11 bits wide, while version 2.0B defines it as 29 bits wide. The CAN frame is illustrated in figure 1. Fields relevant to this project are:

- Identifier: Message ID, may be 29 or 11 bits;
- IDE: Identifier extension bit, determines whether the ID is 29 or 11 bit;
- DLC: Data length code, determines the number of data (payload) bytes; and
- RTR: Remote transmission request bit, determines whether the message contains data (0) or is a request for data (1).

In this project, all messages are defined with a data length of 8 bytes ($DLC = 8, RTR = 0$) and the standard message identifier width of 11 bits ($IDE = 0$).

The CAN peripheral is configured such that it will reject incoming messages with an ID not matching its configuration. This is handled by the `ID` and `Mask` fields, where a zero bit in the mask field will result in the corresponding bit being treated as irrelevant (DC) when an incoming ID is compared with the ID field. In the example configuration in figure 1, the unit will accept any combination of bits in the six least significant bits, so long as the five most significant bits match the value $0b00001$, for instance.

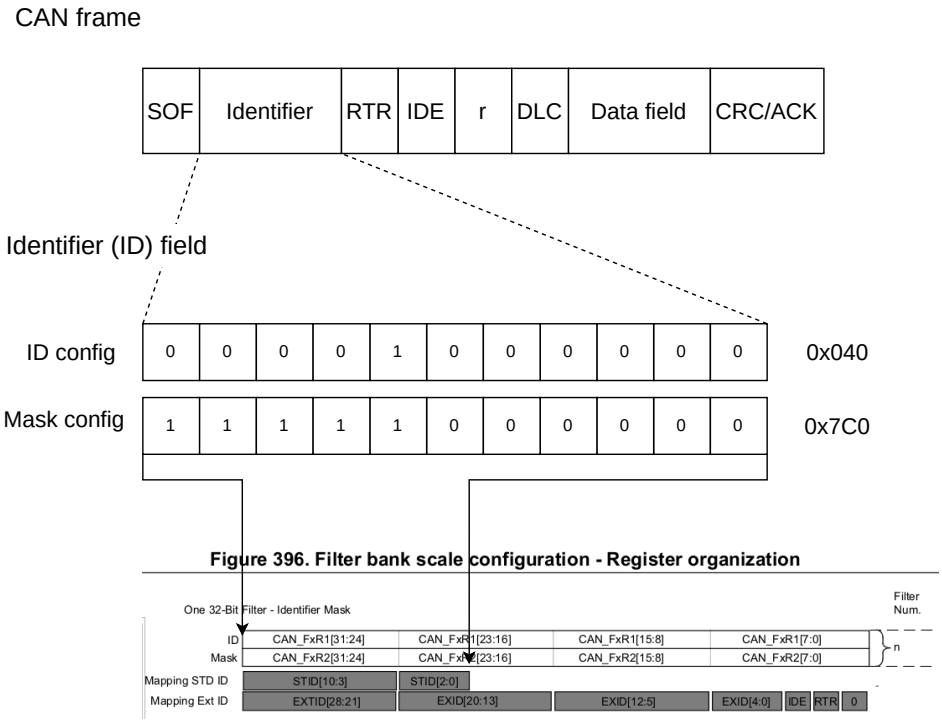


Figure 1: CAN frame breakdown

CAN bus uses bit stuffing for bit stream encoding (CAN standard pt A cp 5[13]), where a complemetary bit value is inserted wherever five consecutive bits of the same value are present upon transmission. For a standard ID message, this results in a theoretical maximum frame length of 135 bits. At the maximum transmission rate of 1Mbps, the maximum frame transmission rate is approximately 7.4kFps.

6.5.2 Inter-integrated circuits (I2C)

The STM32F303 supports the I2C standard revision 3 (RM0316 cp 28), including Fast Mode transmission rates up to 400kbps and 7 bit slave addresses. I2C is a 1-N bidirectional bus centered around transmission of byte data between ICs. A key concept is that the receiving unit (i.e. master or slave) acknowledges reception of each byte by pulling the SDA line low (NXP UM10204 cp 3.1.6[31]).

Implementation of the protocol is to some extent constrained by the slave unit. The LSM6DSM IMU has a set 6 bit address, where the seventh bit may be set by pulling one of its pins (`SA0`) high or low so as to allow for two units on the same bus. When communicating with the unit, this 7 bit address is transmitted and immediately followed by a bit indicating whether the slave is being written or read. In the case of a write, the following byte indicates which slave register data is being written to. In the case of a read, the IMU will output the data of a register previously determined by a write operation. See figure 2, snipped from the LSM6DSM datasheet[45].

Table 14. SAD+Read/Write patterns

Command	SAD[6:1]	SAD[0] = SA0	R/W	SAD+R/W
Read	110101	0	1	11010101 (D5h)
Write	110101	0	0	11010100 (D4h)
Read	110101	1	1	11010111 (D7h)
Write	110101	1	0	11010110 (D6h)

Table 15. Transfer when master is writing one byte to slave

Master	ST	SAD + W		SUB		DATA		SP
Slave			SAK		SAK		SAK	

Table 16. Transfer when master is writing multiple bytes to slave

Master	ST	SAD + W		SUB		DATA		DATA		SP
Slave			SAK		SAK		SAK		SAK	

Table 17. Transfer when master is receiving (reading) one byte of data from slave

Master	ST	SAD + W		SUB		SR	SAD + R			NMAK	SP
Slave			SAK		SAK			SAK	DATA		

Table 18. Transfer when master is receiving (reading) multiple bytes of data from slave

Master	ST	SAD+W		SUB		SR	SAD+R			MAK		MAK		NMAK	SP
Slave			SAK		SAK			SAK	DATA		DAT A		DATA		

Figure 2: The implementation of the I2C protocol is to some extent constrained by the slave, here an LSM6DSM IMU

6.6 PID control

Joint control was implemented as a discretised variant of the well-known Proportional-Integral-Derivative (PID) controller[30], presented in equation 12, where a control variable u is set with respect to an observed error e in a process variable setpoint. In this system, joint actuator motor voltage is set with respect to a joint's positional setpoint.

$$u = K_p e + \frac{K_p}{T_i} \int_{t_0}^{t} e d\tau + K_p T_d \frac{de}{dt} \tag{12}$$

6.6.1 Proportional term

The first term of the equation, $K_p e$, states that the control variable should be directly proportional to the setpoint error. A high value of the proportional gain K_p results in the controller reaching its setpoint quickly, before tending to overshoot due to inertia in the system under control. A proportional term alone is not sufficient to achieve exact accuracy; for a given value of K_p, some steady-state error will be present due to physical limitations in the system. For instance, a joint will stop moving when the torque produced by the motor is insufficient to overcome mechanical load on the joint. Due to the gradual decrease in controller output as the error decreases, this point will necessarily be reached before the joint reaches its setpoint.

6.6.2 Integral term

The second term, $\frac{K_p}{T_i}\int_{t_0}^{t} e d\tau$, states that the control variable should be proportional to the accumulated error over time, and the goal is to reduce or eliminate steady-state error. Where the proportional term output abates when error decreases, the integral term output will increase so long as the setpoint has not been reached. The integral time T_i signifies the system's tolerance to error, and a low value tends to result in setpoint overshoot and oscillations around the setpoint. The introduction of integral action will result in zero steady-state error so long as the system is capable of reaching its setpoint with maximum power output. For instance, so long as the load on a joint is small enough that its motor is capable of overcoming it, integral action will eventually result in a motor torque sufficient to maintain a joint's position at its setpint.

6.6.3 Derivative term

The final term, $K_p T_d \frac{de}{dt}$, states that control variable should be proportional to the time derivative of the error. The goal is to predict error development based on past error measurements, and may be understood as accounting for inertia in the system by reducing control output as the system nears its setpoint. The derivative time T_d acts as a proxy for this parameter, and a high value will dampen oscillations resulting from the proportional and integral terms while increasing the system's settling time.

6.7 ROS

The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source. – Open Robotics, ROS homepage[35].

ROS is not an operating system (OS) as such, but rather a set of software libraries organised as distributions (distros) similar to an OS. ROS runs as a set of programs on top of an OS, forming a virtual network in which each node is responsible for a narrow set of tasks. It is compatible with most UNIX OSes, including real time OSes like FreeRTOS[11]. ROS libraries are available as APIs in Python or C++[36]. There are two major versions, ROS and ROS2, with minor versions, i.e. distros, named alphabetically. This project uses the ROS2 Iron Irwin distro and C++.

6.7.1 Nodes

Nodes in a ROS network are responsible for singular tasks such as a hardware interface, a specific bit of data processing, or data storage. A node may run as an idependent program (by which is meant a compiled binary in the context of C++) or a thread in a larger program. Nodes communicate by publishing and subscribing to "topics" over which "messages" are sent adhering to a format specific to that topic. At a programming level, messages are similar to C language structs in that they define a set of fields of specified data types. A node is typically naïve to the existence of other nodes publishing/subscribing to a topic it itself publishes/subscribes to, and unaware of topics it does not.

6.7.2 Packages

ROS code is organised in packages with a singular CMake file (in the case of C++) as its product. A package typically defines a set of nodes or message types as well as dependencies.

6.7.3 Universal Robotic Definition Format

URDF (Unified Robot Description Format) is a file format for specifying the geometry and organization of robots in ROS. - ROS2 Documentation[38]

URDF is an XML file format in which robotic links and joints are described with their physical parameters such as shape and inertial matrices as well as joint extension/velocity/acceleration limits. URDF files may be written manually, or generated based on 3D model (CAD) files.

6.8 Software development

6.9 SOLID principles

The SOLID principles for object oriented programming (OOP) served as a guide for various implementation elements of this project as they to some extent may be generalised to apply to procedural programming, system architecture in general, and hardware design.

6.9.1 Single responsibility principle

The Single Responsibility Principle (SRP) states that a module should be responsible for only one thing; that "there should never be more than one reason for a class to change"[24]. Class may be generalised to "module" in the case of the C language/systems design. The motivation is to reduce probability of common cause failures[18]: that a system breaks due to unforeseen, and/or hard to trace, dependencies between modules which are not obviously related. In the context of an embedded system, a module could be responsible for functionality pertaining to a single hardware unit such as a sensor. The "thing" for which a module is responsible, then, is a physical object.

6.9.2 Open-closed principle

The Open-Closed Principle (OCP) states that "entities should be open for extension, but closed for modification"[23]. In essence, this means that once functionality has been added to a module, that functionality should never be changed. The motivation is to avoid cascading changes, i.e. having to change dependent modules when a module is changed. Adherance to this principle requires extensive testing of functionality before its addition to a module is committed, in order to ensure that the module is essentially "perfect" upon integration with the larger system.

This is hard to achieve in practice, but may generally be enforced by abstraction. For C++, this could be enforced by encapsulating basic functionality in base classes which are unlikely to change, from which child classes derive their properties. C does not provide that option, but abstraction and encapsulation may still be achieved by the implementation of sufficiently narrow modules – as per the SRP.

6.9.3 Liskov substitution principle

The Liskov Substitution Principle (LSP) states that "functions that use [...] base classes must be able to use objects of derived classes without knowing it"[22]. Plainly, that replacing an object with a child object should not break the program, and a base object should not need to know about child objects. This principle has little direct applicability outside of OOP, but could be stretched to mean that an entity should be naïve to the origin of a piece of information.

As a concrete example, consider that the MCUs used in this project may need to process IMU data, and that they communicate via CAN bus. This data may originate from an IMU controlled by the relevant MCU, or it may have been collected from another via CAN bus. The entity responsible for processing the data (in this case most likely a function) should not need to be informed which

one is the case; data from a local IMU may be "substituted" for data from a remote IMU without breaking the program.

6.9.4 Interface segregation principle

The Interface Segregation Principle (ISP) states that modules should not have "fat" interfaces, that is, interfaces which may be broken into groups of member functions[21] such that each group serves a different set of clients. In this context, and 'interface' is the group of functions relating a derived class to its bases, and an alternative formulation is that "clients should not be forced to depend on interfaces that they do not use". The key idea is to not include unnecessary base classes in a derived class on the basis that its children may need the provided functionality. As with the LSP, this is a structural principle relating to class inheritance and thus is not directly applicable to this project.

In the C context, an interface could be understood to be the set of public functions which a module provides. A "narrow" interface could simply be an interface which provides the minimum necessary functionality to interact with it, implemented such that they are safe to use in any context.

6.9.5 Dependency inversion principle

The Dependency Inversion Principle (DIP) states that "abstractions should not depend upon details; but details should depend upon abstractions"[20]. In the context of OOP, this means that a base class should provide abstract functionality, and that derived classes should be specialised instances of those abstractions. Not adhering to this principle tends to result in violations of the ISP, where a derived class may be affected by changes to a parent class it is not itself using.

Generally, it could be said that modules which depend on concrete functionality become "rigid"; difficult to maintain as the code base expands, and hard to reuse. Reuse may be of limited concern in an embedded systems context where code may tend to be optimised for a specific use case or hardware item (which is not likely to change). Modules should still be developed with generality in mind, however.

Returning to the example of IMU data processing: It was said that the responsible unit should be naïve to the origin of the data. Still, it may need to take into account unique details of the IMU such as calibration parameters. The processor should be general enough to take into account calibration parameters from any *specific* IMU. That is, the implementation is *independent* of any concrete IMU.

7 System specification

This section provides a high level description of the robotic system(7.1), a description of the hardware developed during the specialisation project(7.2), and a requirement specification for the software developed in this master project(7.3). The latter elaborates on scope items 2 and 3 from section 5.3.

7.1 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 3, 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 4.

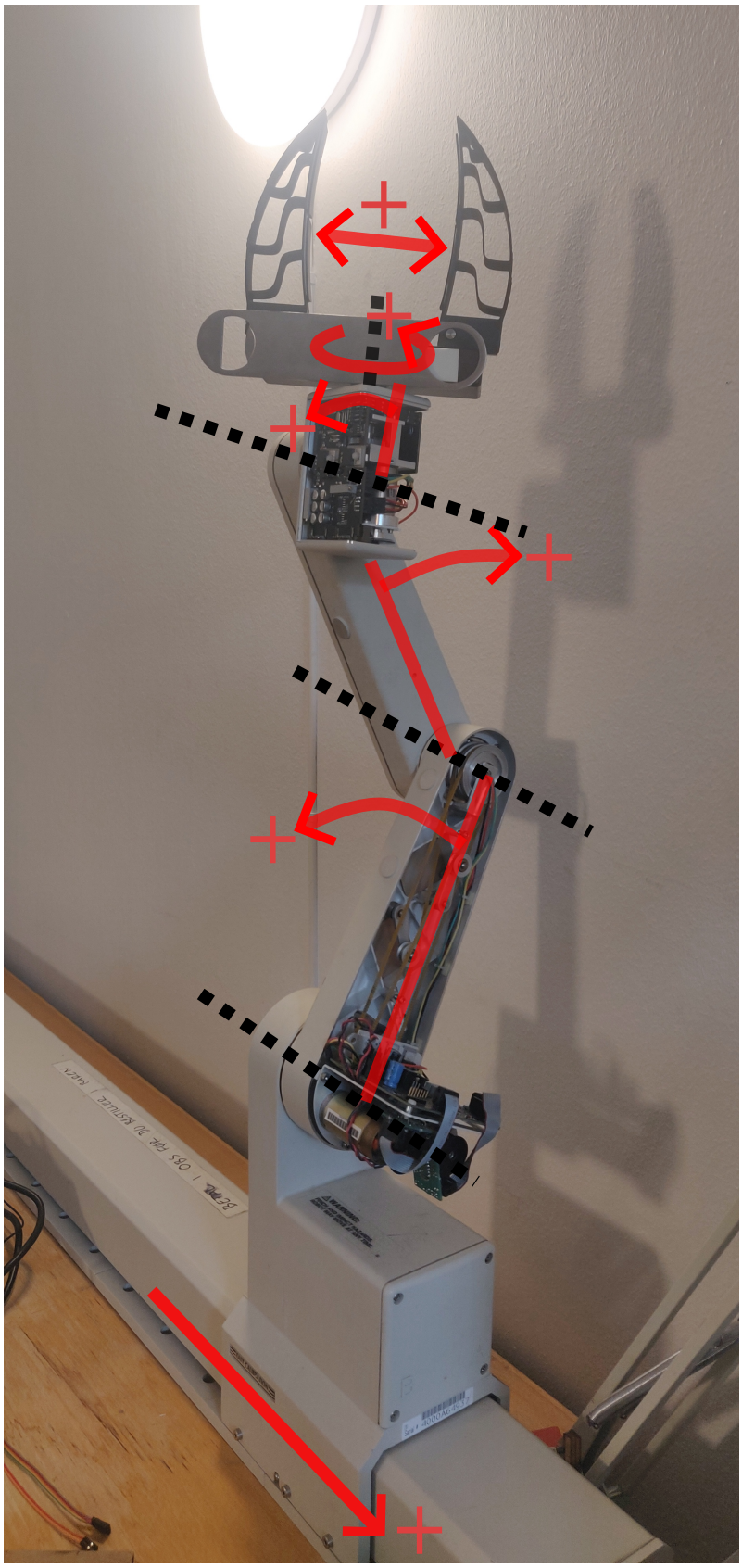


Figure 3: The ORCA arm, illustrated with joints and their directions. From the bottom up, joints are named rail, shoulder, elbow, wrist, twist and pinch.

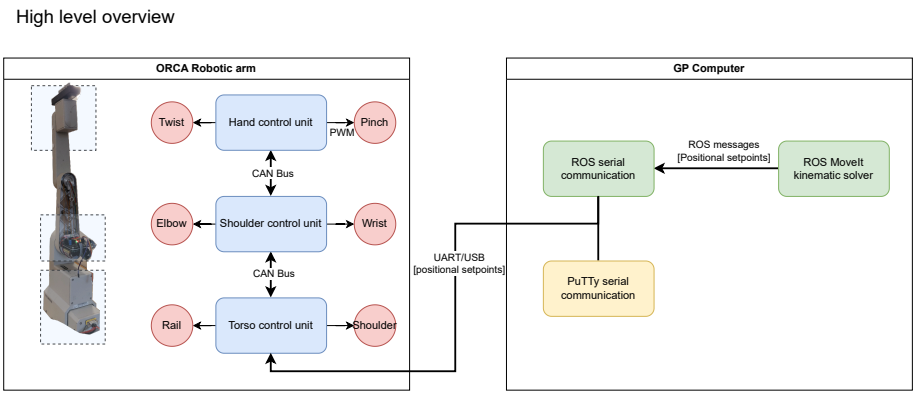


Figure 4: An overview of the robotic system

7.2 Hardware

As mentioned in section 5.2, all mechanical and some electromechanical parts were kept during the specialisation project in order to limit its scope. The parts are presented in table 3, 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 5. Figure 6 provides a detailed description of the arm, and was originally presented in the specialisation project report[3].

Table 3: Original parts kept in the project

Part name	**Description**	**Part no.**
Rail motor	Pittman 40mm 30.3V BDC motor	DC040B-2[27]
Shoulder motor	Pittman 54mm 30.3VDC BDC motor	DC054B-5[28]
Elbow motor	Pittman 40mm 24V BDC motor	DC040B-3
Wrist motor	Pittman 30mm 30.3V BDC motor	DC030B-3[26]
Twist motor	Escap 23mm 23LT2R, 12V BDC motor	23LT2R[12]
Pinch motor	Escap 23mm 23LT2R, 12V BDC motor	23LT2R
Wrist rotational sensor	TT Electronics Photologic Slotted Optical Switch	OPB971N51[10]
End switch	TT Electronics Photologic Slotted Optical Switch	OPB971N51
Motor encoder x6	Optical quadrature encoder 500CPR	HEDS-9100[54]

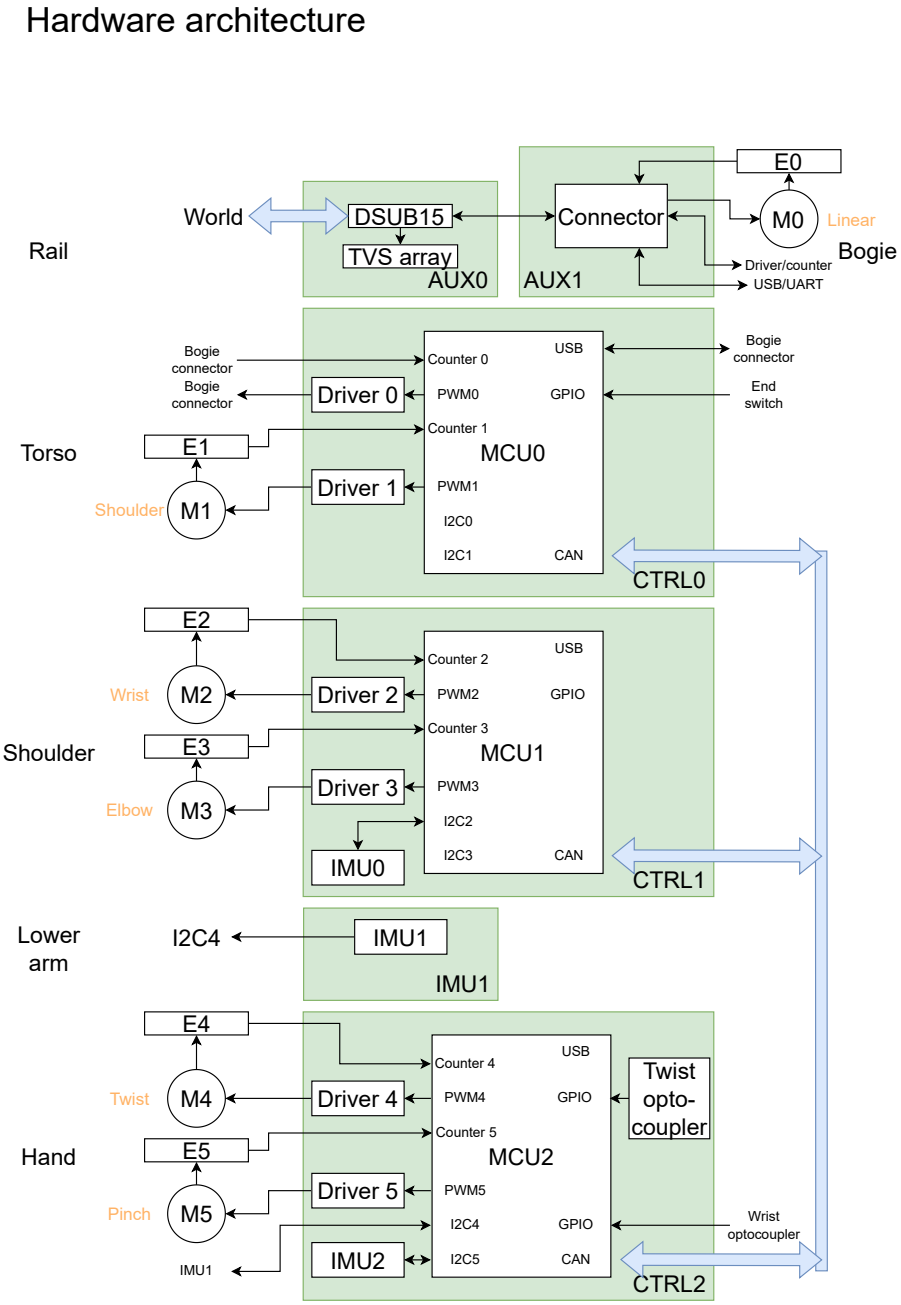
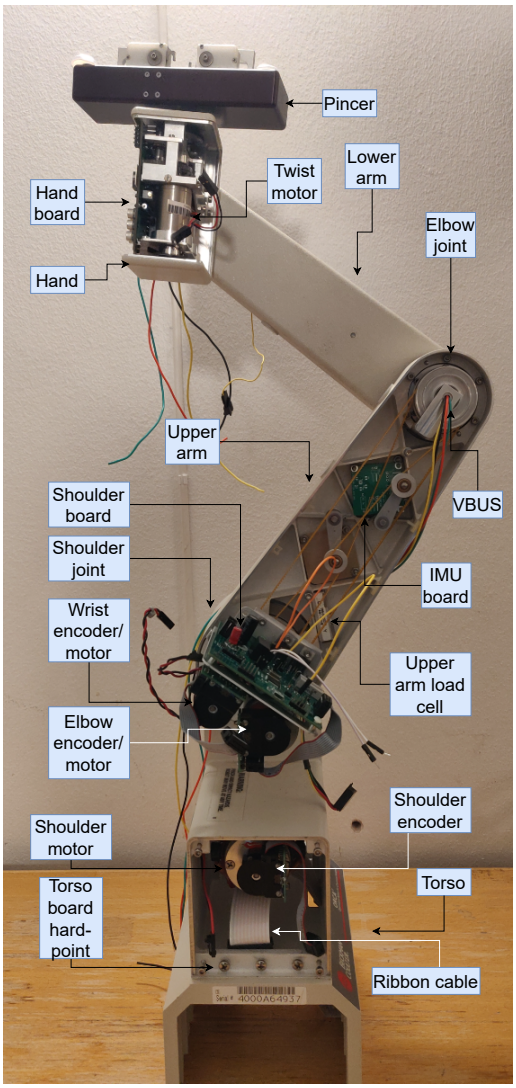
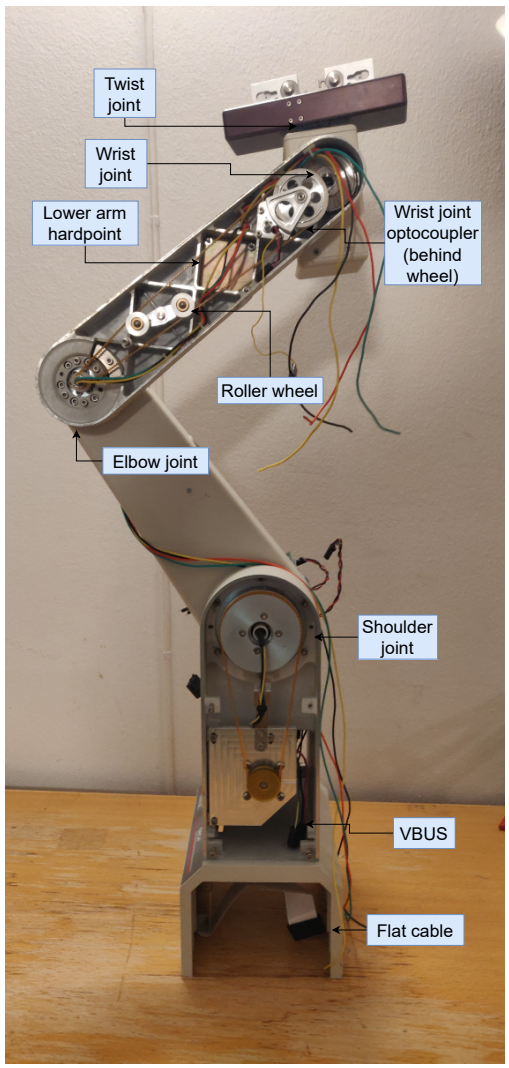


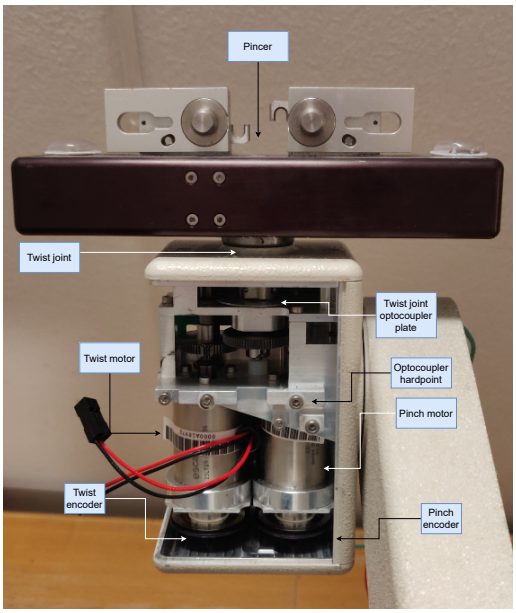
Figure 5: Hardware architecture. Green squares represent a PCB designed in the specialisation project, with key components as white squares. Left hand keywords indicate PCB mount location inside the arm. Originally presented in the specialisation report. Mn = Motor number, En = Encoder number



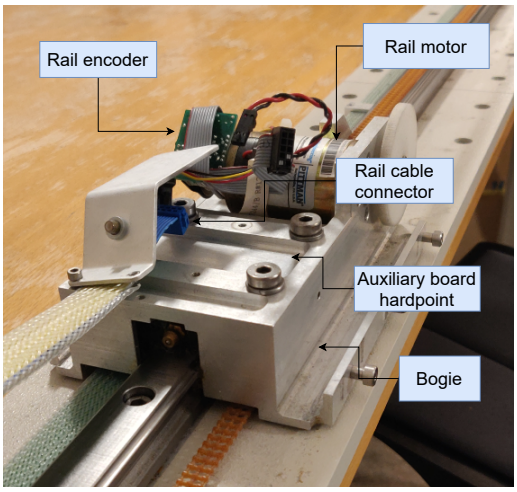
Arm viewed from the front.



Arm from the back.



A more detailed image of the hand and pincer.



The bogie on which the torso is mounted.

Figure 6: The arm with annotations. Orginally presented in the specialisation report

7.2.1 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(7.2.2). Additionally, it protects the USB and UART data lines from voltage spikes via its TVS array CDSC706[7]. The TVS array is clamped at 5V sourced from the torso control unit(7.2.3).

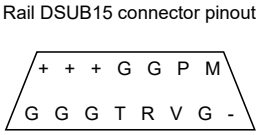


Figure 7: DSUB15 pinout

Table 4: Legend for figure 7, 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

7.2.2 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(7.2.3).

7.2.3 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 (7.2.4), 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 5 for a presentation of the symbols in figure 5.

7.2.4 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 (7.2.6) 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 5.

Table 5: 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 3
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 3
Motor 1 (M1)	Shoulder joint motor	See table 3

7.2.5 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.

7.2.6 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 5.

7.3 Software

The hardware presented in section 7.2 provides constraints for the software architecture. As mentioned, the software covers scope points 2 and 3 from section 5.3. 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

7.3.1 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.

7.3.2 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

7.3.3 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

8 Tools and workflow

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

8.1 Software

8.1.1 Tools

Git was used during all phases of the project, and the project's repository may be found at Github[5].

KiCad 7.0.11 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[19]. When necessary, component footprints were downloaded from the UltraLibrarian website when available, or otherwise drawn in KiCad based on component datasheets. These footprints may be found in the directory `PCBs/lib` in the GitHub repository.

STMCubeMX[48] 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. 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[29] with ST extension[8] was the primary IDE for this project. The extension *stm32-for-vscode* was used to build and flash binaries for the MCUs.

PuTTy[52] was used for serial communication with and debugging of the MCUs.

ROS2 Iron[37] was used for interaction with the robotic system in the later stages of development. ROS2 is an open source robotic development ecosystem, see section 6.7 for more information.

MoveIt[40] was used as a GUI and kinematic solver for high level control of the robotic system, see section 6.7 for more.

8.1.2 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 kit8.2.

8.2 Electronic/Hardware

Reflow oven: A SEF 548.04G reflow oven[41] was used for the soldering of PCBs.

Oscilloscope: A Rigol DS1054Z oscilloscope[16] was used for inspection of produced PCBs, debugging, et cetera.

9 Hardware design

This section discusses the hardware presented in section 7.2 in further detail. Hardware produced for this project is an iteration of hardware produced for the specialisation project[3], and is dubbed Mk1.1. The material presented overlaps with sections 7 and 8 of the specialisation project report[3], 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 17.

9.1 Improvement matrix

Table 6 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 6: A summary of further work

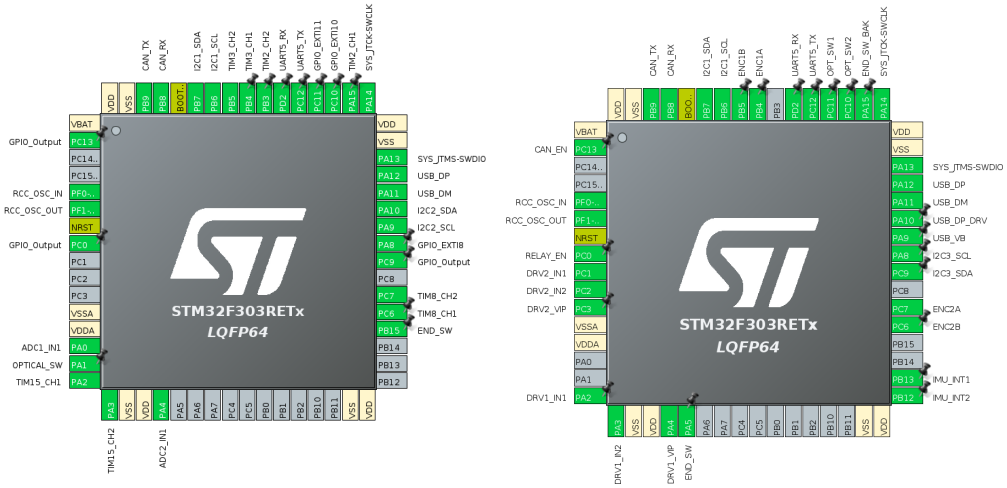
Item/affects unit	**Hand**	**Shoulder**	**Torso**	**Bogie**	**Rail**	**IMU (board)**	**MCU**
USB VBUS voltage divider		X	X				
USB VBUS pin		X	X				X
USB pullup on DM		X	X				
USB pullup transistor		X	X				
IMU 5V and 3.3V lines	X	X				X	
IMU/OPT header design	X	X				X	X
IMU header placement		X					
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		

9.2 MCU pinout

The MCU pinout lays the foundation for PCB layout, and was changed first. The pinout is presented in figure 8.

- USB VBUS detection was set to pin PA9 in accordance with AN4879 chapter 2.6[44].
- 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[49].
- 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[3]. 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 10.8.



MCU pinout configuration prior to hardware improvements

MCU pinout configuration after hardware improvements

Figure 8: Comparison of the old and new MCU pinout configuration

9.3 IMU board

As mentioned in section 7.2, 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 6. 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[45] 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 appendix A.

The OPB971 optical sensor requires an input voltage of minimum 4.5V[10], while the LSM6DSM IMU has a maximum input voltage of 3.6V[45]. 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[15], was intruduced as a replacement for a simple voltage divider in Mk1.

Figure 9 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[3]). 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:

$$I_{I2C\_PWR1} = (1_{INT\_SEL1} \oplus 2_{INT\_SEL1}) \cdot (I_{INT\_EN1} \oplus O_{INT\_EN1}) \tag{13}$$

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

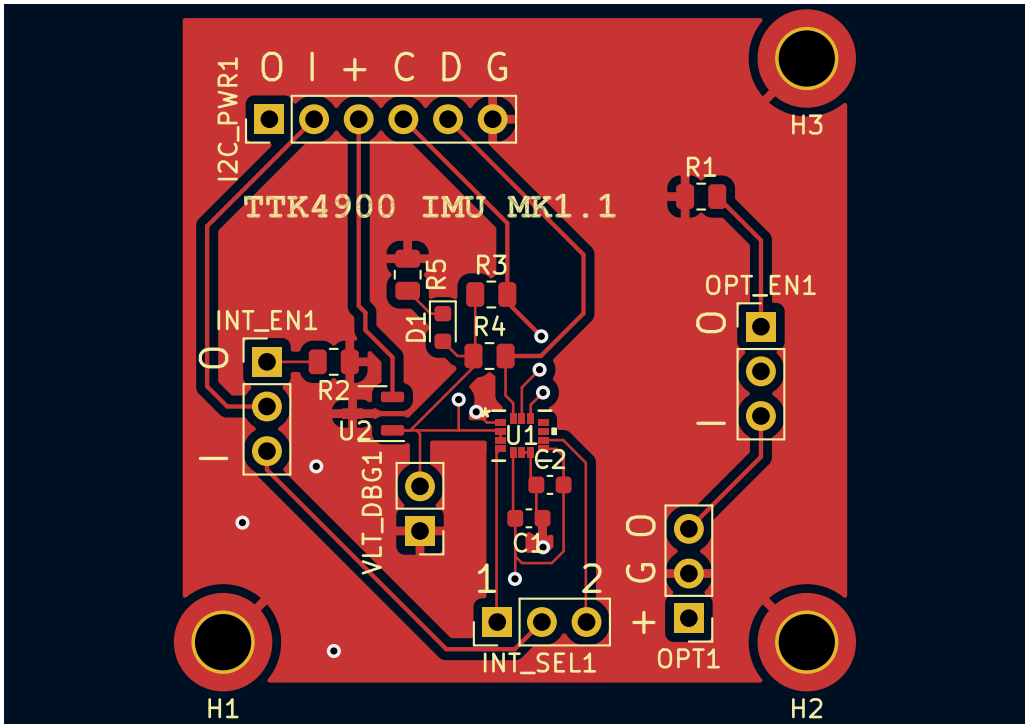


Figure 9: Front copper and silk layers of the improved IMU board

9.4 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 appendix A.

Table 8 explains the silk symbols on the 16 pin connector socket of the rail board as seen in figure 10, which shows the back copper and silk of the rail board. The silk should be interpreted as an

Table 7: 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 10 is connected to the DSUB15 connector presented in section 7.2, silk symbols explained in table 4. 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 8: 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

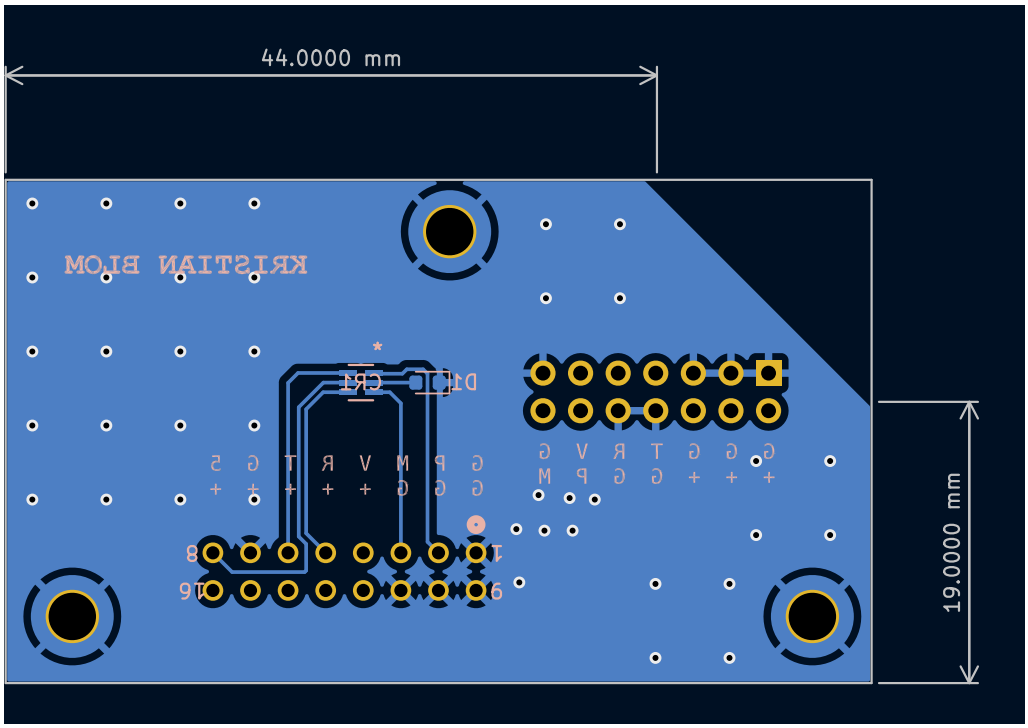


Figure 10: Back copper and silk layers of the improved rail board

9.5 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 11, and the silk symbols for the 20 pin connector socket is presented in table 9. The silk symbols for the 16 pin connector socket is presented in table 8. The circuit is presented in appendix A.

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

Table 9: 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		

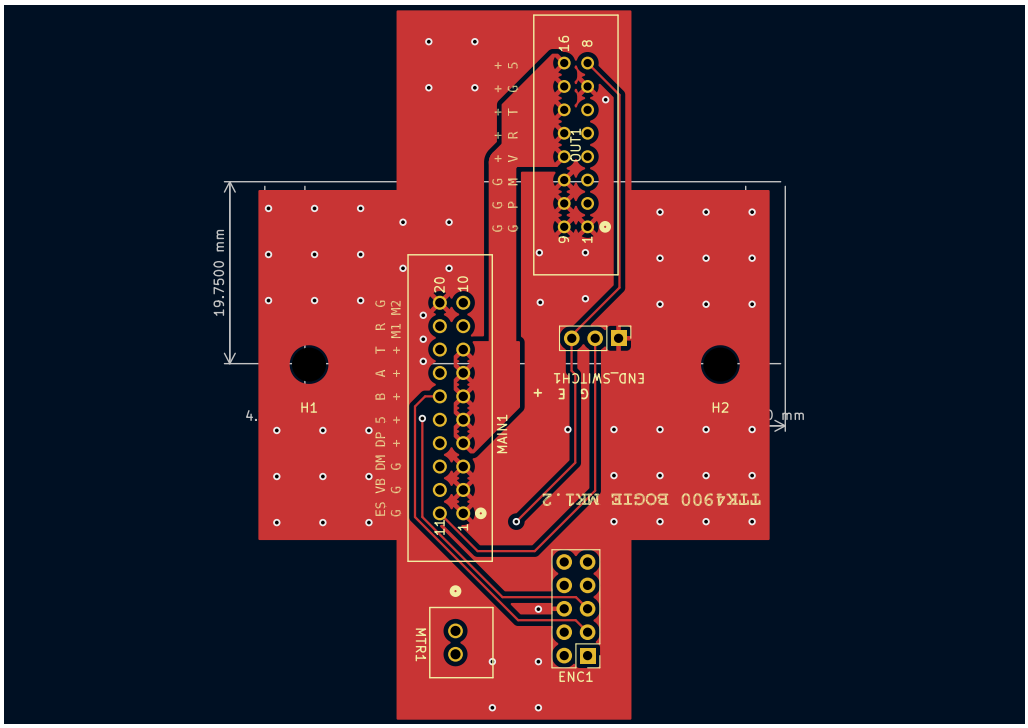


Figure 11: Front copper and silk layers of the improved bogie board

9.6 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 13 presents the improved torso control unit. Legends for various pin headers are presented in 10. The torso circuit is presented in appendix A.

9.6.1 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 5, 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 appendix A.

The circuit was redesigned in accordance with chapter 2.6 and figure 5 of AN4879[44], see figure 12. 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**

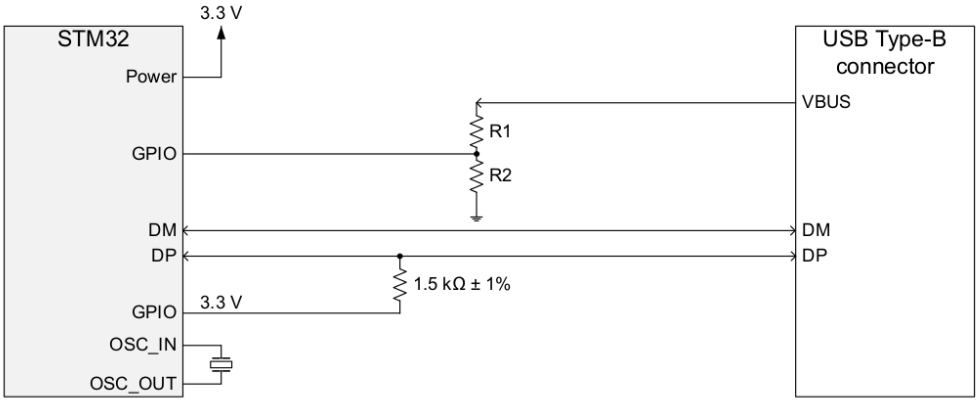


Figure 12: Figure 5 from ST AN4879, illustrating a valid USB circuit

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.

9.6.2 CAN bus assembly

The CAN bus assembly is present in all three control units, and consists of the TJA1057BT[32] 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 appendix A.

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[32] chapter 6, pin 5).

9.6.3 Motor driver assembly

The motor driver assembly is present in all three control units, and consists of two DRV8251A motor drivers[55], 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 appendix A.

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 9.2 for Mk1.1.

9.6.4 Voltage regulator assembly

The voltage regulator assembly is present in all three control units, and consists of two adjustable LM317HV linear voltage regulators[56] 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 appendix A.

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.

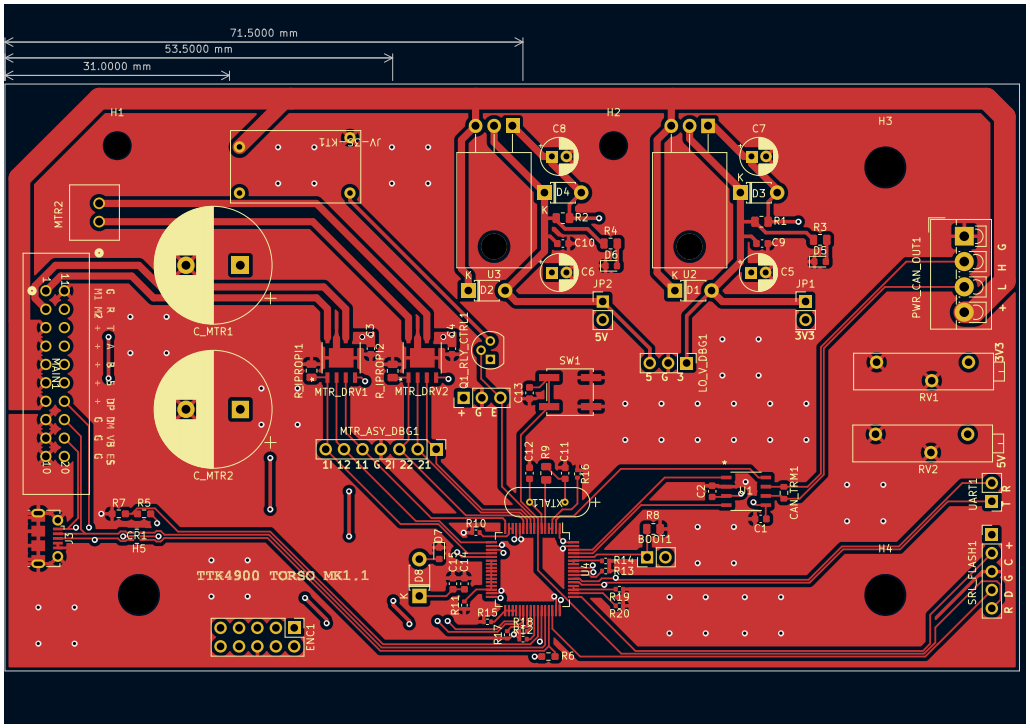


Figure 13: Front copper and silk layers of the improved torso board

9.7 Shoulder

As shown in table 6, 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 figure 15, and the shoulder circuit is presented in appendix A.

[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 10: 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 UM1724[50].

9.7.1 Mount points

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

9.7.2 IMU assembly

The IMU assembly is present in the shoulder and hand control units, and consists of the LSM6DSM IMU[45] and its power supply circuit. A modified version is also present in the IMU board, and the circuit is presented in appendix A. The SDA and SCL lines are pulled to I2C port 3 of the hand and shoulder MCUs.

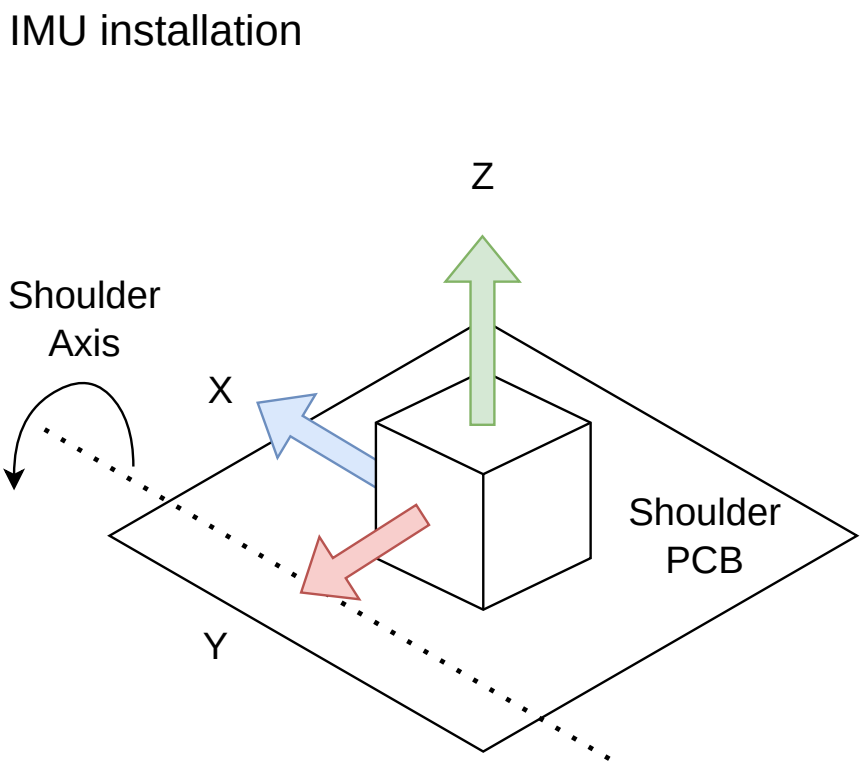


Figure 14: Shoulder IMU installation relative to shoulder joint axis

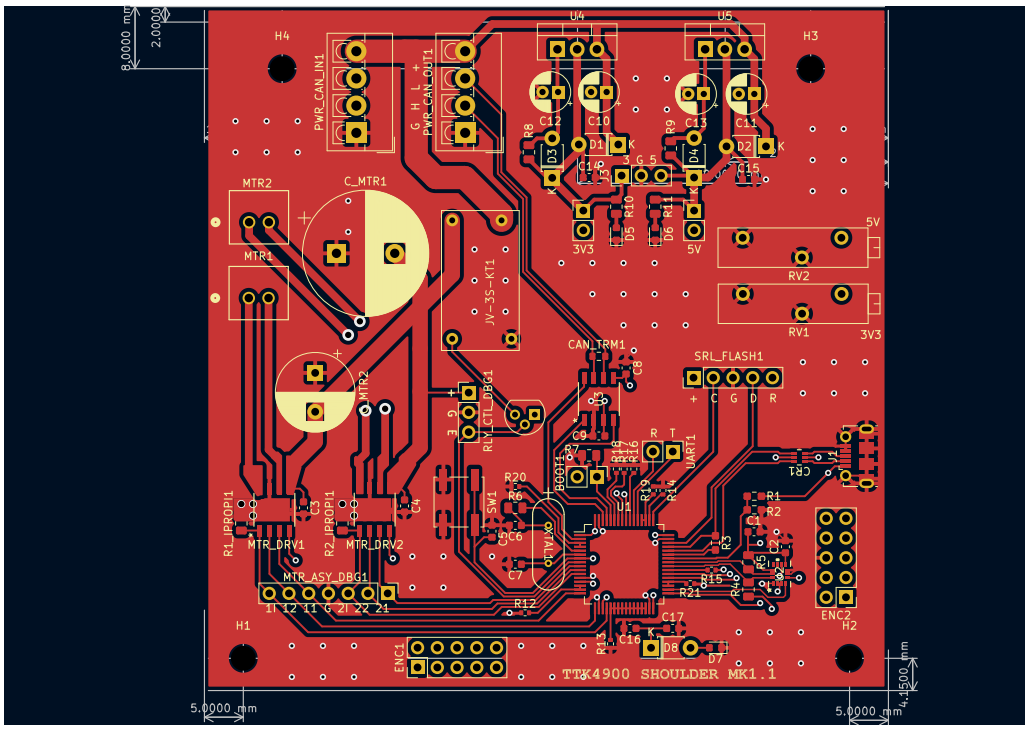


Figure 15: Front copper and silk layers of the improved shoulder board

9.8 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 11, and boards are presented in figures 16 and 17.

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 appendix A.

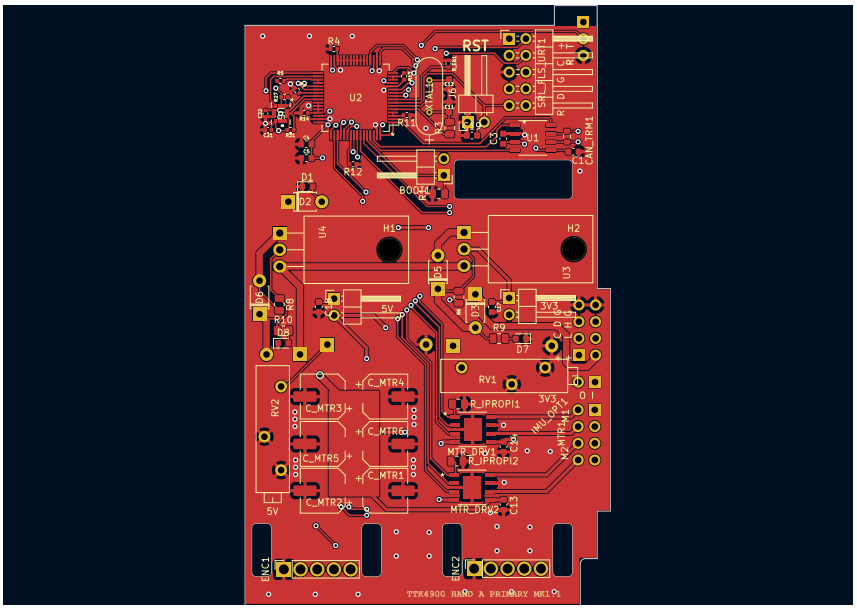
9.8.1 Hand B: Optical sensor

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 appendix A.

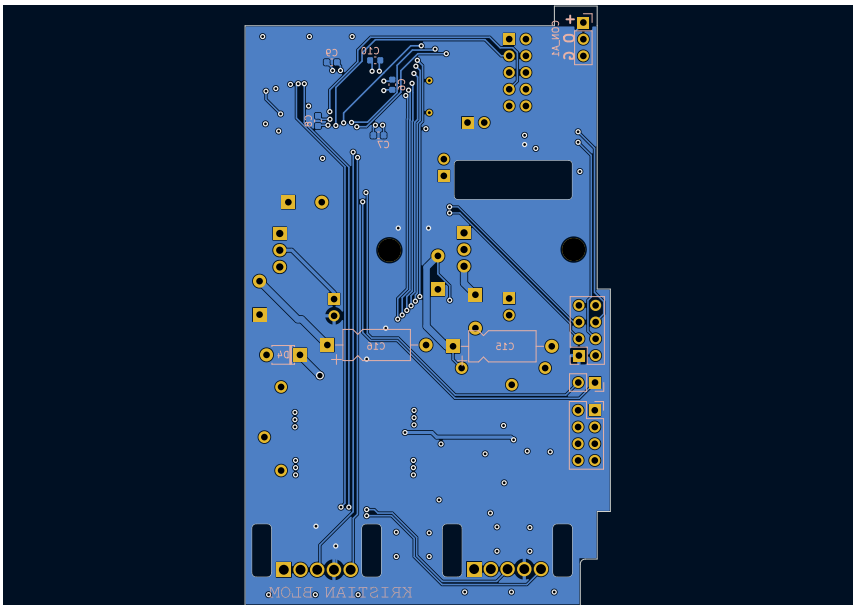
The OPB971 optical sensor activates when its aperture is obstructed, such that a short circuit occurs between the output and ground pins. A voltage sensor placed between the output and voltage input pins will then sense the differential. A breadboard test was conducted around this concept with an LED 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 11: 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 16: Front and back copper layers of the improved hand unit

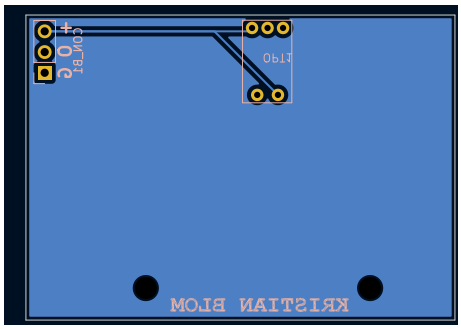


Figure 17: Back copper and silk layers of the hand B board

9.9 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.

9.9.1 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`. 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.

9.9.2 MCU programming

The previously generated MCU pinout was compiled using VSCode and the ST extension (see section 8.1), 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.

9.9.3 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 10.5), 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.

9.9.4 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 10.8) 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[55] chapter 8.4.1. This test applies to all control units.

PWM output was verified by probing the `MTR_ASY_DBG1` header 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

9.9.5 UART data transmission

UART transmission was tested by sending the character string `HELLO, WORLD!` using configuration parameters discussed in section 10.9 to the external computer via the Nucleo development kit's UART-to-USB adapter (see chapter 6.8 of UM1724[50]). 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.

9.9.6 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 9.8, 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[49]). 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.

9.9.7 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 9.9.6, this is above the five volt pin activation threshold, and the sensors were deemed to be operational.

9.9.8 I2C data transfer

I2C was tested by attempting to read the `WHO_AM_I` register of the three LSM6DSM IMU units[45] using a program developed for the specialisation project, and displaying the value via UART. The program utilises the STM32 HAL library discussed in section 10.1, 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, 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 12.

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 12: 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 6.5.2) should have appeared. This indicates that the circuit is correctly designed, and that I2C1 is correctly configured (see 10.6). Figure 18 illustrates this. Here, the hand I2C1 peripheral has been configured for 100kHz transmission in standard mode. The spikes and relatively long rise time of SCL and SDA could indicate high line capacitance due to the measurement setup, and/or that the signal was somehow malformed by the MCU.

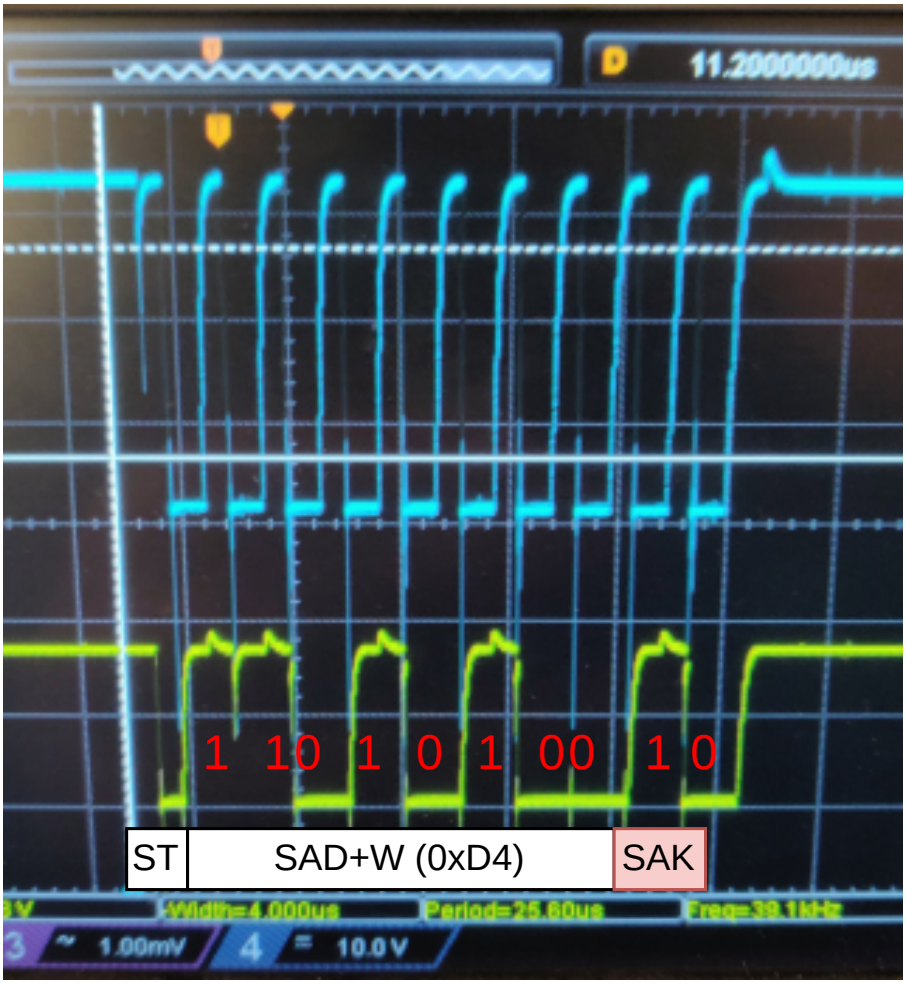


Figure 18: I2C failure with hand I2C1 and IMU board unit. The write sequence appears to be sent correctly, but the Slave Acknowledge bit is high. Note that SCL and SDA are at the same voltage level, but oscilloscope readout has been adjusted for SCL to better see the waveform shape.

However, 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, 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 and the IMU assembly is that both interrupt pins are never grounded at the same time, see equation 13. 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[45]. 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 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 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[1]. While the STM32F303 is configurable for repeated start condition (chapter 25.2.1 of UM1786[47]), 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.

9.9.9 USB data transfer

In order to verify the USB circuit, the shoulder control unit was programmed to act as a USB device by enabling the USB peripheral. The USB was then plugged into the external computer in the hopes that it would register as a device. This did not happen, and due to significant time having been spent on I2C verification; the UART interface being operational; and uncertainty regarding correct use of the USB HAL library, further debugging was not attempted.

USB connectivity is not operational.

9.9.10 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

[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

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.

9.9.11 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.

9.10 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.

9.11 Circuit diagrams

Circuit diagrams are presented in appendix A.

10 Implementation AL1: Peripheral configuration

This section presents the configuration of peripherals and certain core functions of the STM32F303 microprocessor. Peripheral usage lays the foundation for the software architecture presented in section 11, and configuration was done in STM32CubeMX, see section 10.2.

10.1 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[49], reference manual RM0316 for the STMF3 family[46] and user manual UM1786 for the Harware Abstraction Layer (HAL) software library[47].

10.2 STM32CubeMX

As mentioned in section 8.1, 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;
```

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

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 of figure 19, while others are opaque. All generated files have several sections dedicated to user code.

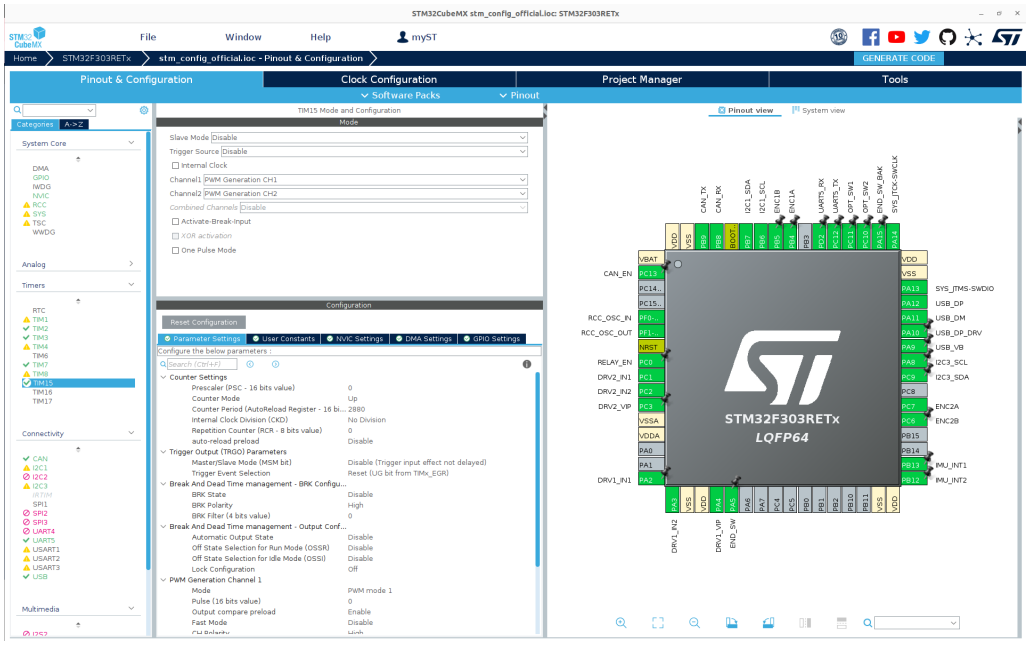


Figure 19: An example of peripheral configuration in STMCubeMX. Peripherals are selected in the left column, and configured in the middle. Mapping to pins may be selected from the MCU representation on the right, where relevant.

10.3 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. The ADC peripheral is described in chapter 15 of RM0316[46].

10.3.1 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

10.3.2 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 12.6) 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 7.3. It will trigger an interrupt if the ADC conversion result is above 4000, corresponding to a motor current draw of approximately 4A (see section 6.3), which then may be acted upon.

10.4 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 12.5 and 11.6. The CAN bus peripheral is described in chapter 31 of RM0316.

10.4.1 Parameters

Parameters not mentioned are left to their default values

- Prescaler: 9
- Time quanta in bit segment 1: 2 times
- Bitrate: 1Mbps (function of prescaler/time quanta)
- NVIC settings: `CAN_RX0` interrupts: Enabled

The enabled interrupt triggers when a CAN message is received.

10.4.2 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.

10.5 GPIO

GPIO is discussed in section 9.2, and concerns the configuration of all MCU pins.

10.5.1 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

10.5.2 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.

10.6 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.

10.6.1 Parameters

Parameters not mentioned are left to their default values.

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

10.6.2 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[45]), 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 9.9.8. As the one functional unit (shoulder) remained operational at the highest possible frequency (400kHz), this setting was kept.

10.7 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.

10.7.1 Parameters

Parameters not mentioned are left to their default values.

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

10.7.2 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 20.

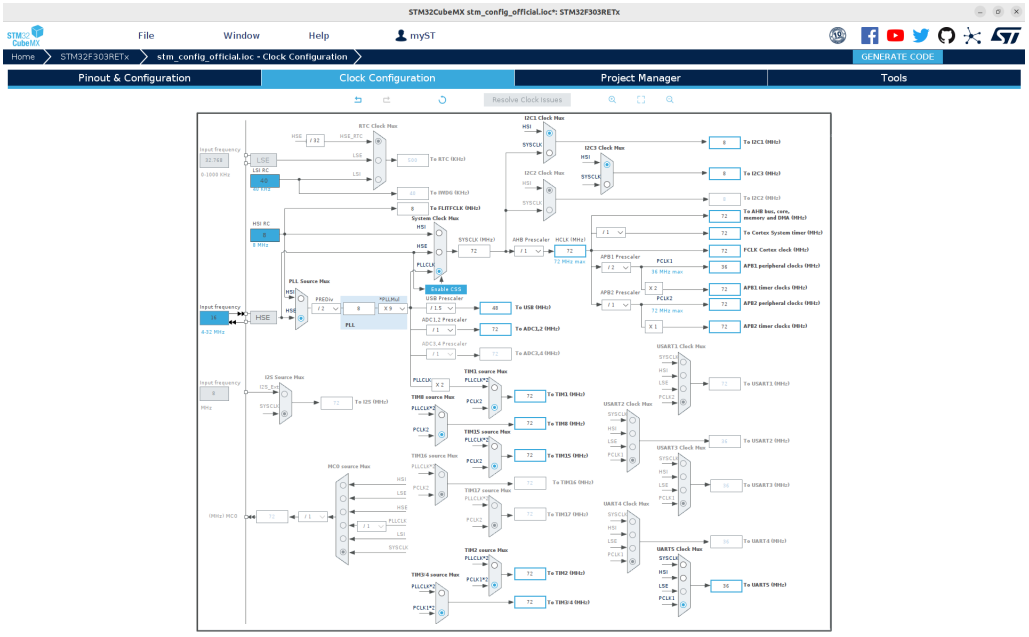


Figure 20: Clock configuration in CubeMX. The GUI presents a number of configration options to the user for tuning clock frequencies of various functions. Prescaler values are generally calculated automatically, and will yield error messages if the user attempts to set an invalid configuration.

10.8 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 11.8. The timer peripherals are described in chapters 20, 21 and 22 of RM0316.

10.8.1 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.

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

- Combined channels: encoder mode
- Counter period: 65535

10.8.1.2 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.

10.8.2 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.

10.8.2.1 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

10.8.2.2 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 6.4. Note that the auto reload register `TIMx_ARR` is referred to as "counter period" in CubeMX.

10.8.3 Interrupt timers: TIM2, TIM4, TIM7

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

10.8.3.1 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

10.8.3.2 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 6.4, and the reasoning behind the target frequencies is presented in 11.8.

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

10.9 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 13). USART/UART is described in chapter 29 of RM0316.

10.9.1 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

10.9.2 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.

10.10 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[44].

10.10.1 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)

10.10.2 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.

11 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 12. Implementation details are mentioned here if they were relevant to the system's overall structure. Figures summarising various aspects of the architeture may be found in section 11.3, details are presented in subsequent subsections and chapters.

11.1 Implementation of Architectural Requirements

See section 7.3 for the complete list of, and motivation behind, architectural requirements.

11.1.1 Naming conventions

11.1.1.1 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.

11.1.1.2 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.

11.1.1.3 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>`.

11.1.2 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.

11.1.2.1 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 11.1.1, 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.

11.1.2.2 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.

11.1.2.3 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 while preserving interface safety according to AR1.3, structs are typically grouped in lists of pointers to the relevant structs.

Again using the motor driver as an example in the following C-like pseudocode and including `interface` usage:

```
//We want to avoid the following:
motor_driver_set_power(int motor, int power):
    if(motor == 1):
        &motor1.power_setting = power;
    else if(motor == 2):
        &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 6.9, 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).

11.1.2.4 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
```

11.2 Code organisation

All code is managed by a single git repository, available via GitHub[5], and placed in one of two directories: code pertaining to ROS nodes (`ros`), and code pertaining to the MCUs (`stm_config_official`). See example structure below.

Code pertaining to the MCUs is structured according to the description given in section 10.2. Code written as part of this project can be found in the directory `TTK4900_drivers`, under 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 12.

Code pertaining to ROS nodes is structured according to best practice for ROS packages[39]. Each node is given a dedicated directory with the typical `src/inc` structure, and placed in the `ros/src` directory.

```
//TTK4900 folder under stm_config_official//ROS package structure
|- stm_config_official                    |- ros
    |- .mxproject                             |- src
    |- Core                                       |- pkg_name
    |- Makefile                                       |- inc
    |- Drivers                                            |- node1.h
    |- TTK4900_drivers                                |- src
        |- Inc                                            |- node1.cpp
            module1.h                           |- build
        |- Src                                  |- install
            module1.c
    |- Middlewares
    |- USB_DEVICE
```

11.3 Architectural overview

This section presents an overview of the implemented architecture. Elements of presented figures are discussed in subsequent sections.

Figure 21 summarises the most important interactions between the arm's system modules (AL[1..3]), 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.

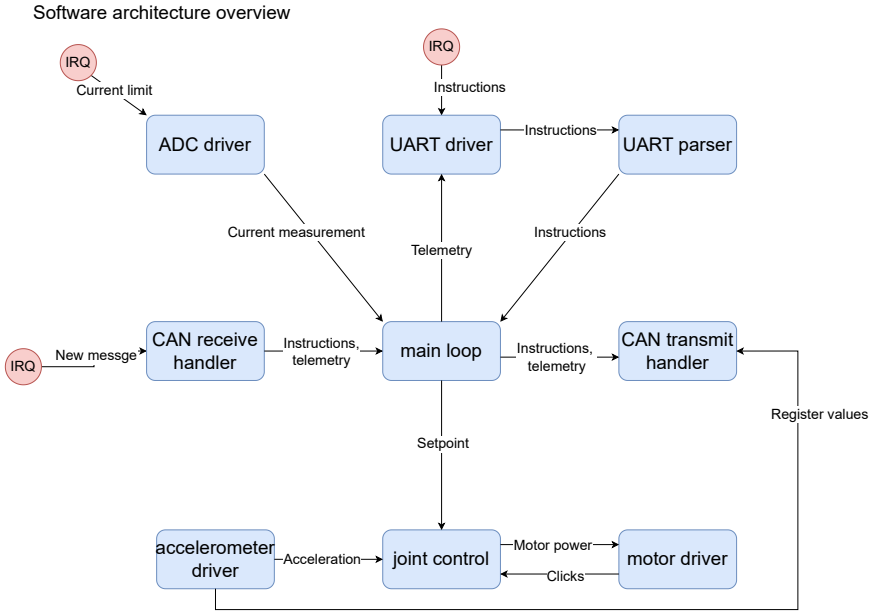


Figure 21: An overview of the central components of the arm's onboard software

Figure 22 illustrates relations between entities on the external computer as described in previous sections as well as the MoveIt motion planning GUI (AL4). Note that the XOR gate is normative; no exclusive access to the computer's serial interface is enforced.

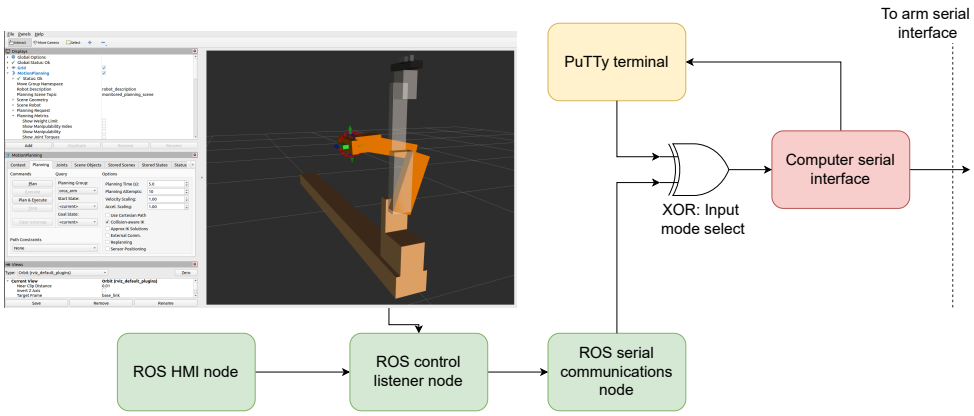


Figure 22: An overview of the central components of software running on the external control computer

Figure 23 is a reference figure presenting developed modules by how they relate to one another across abstraction levels.

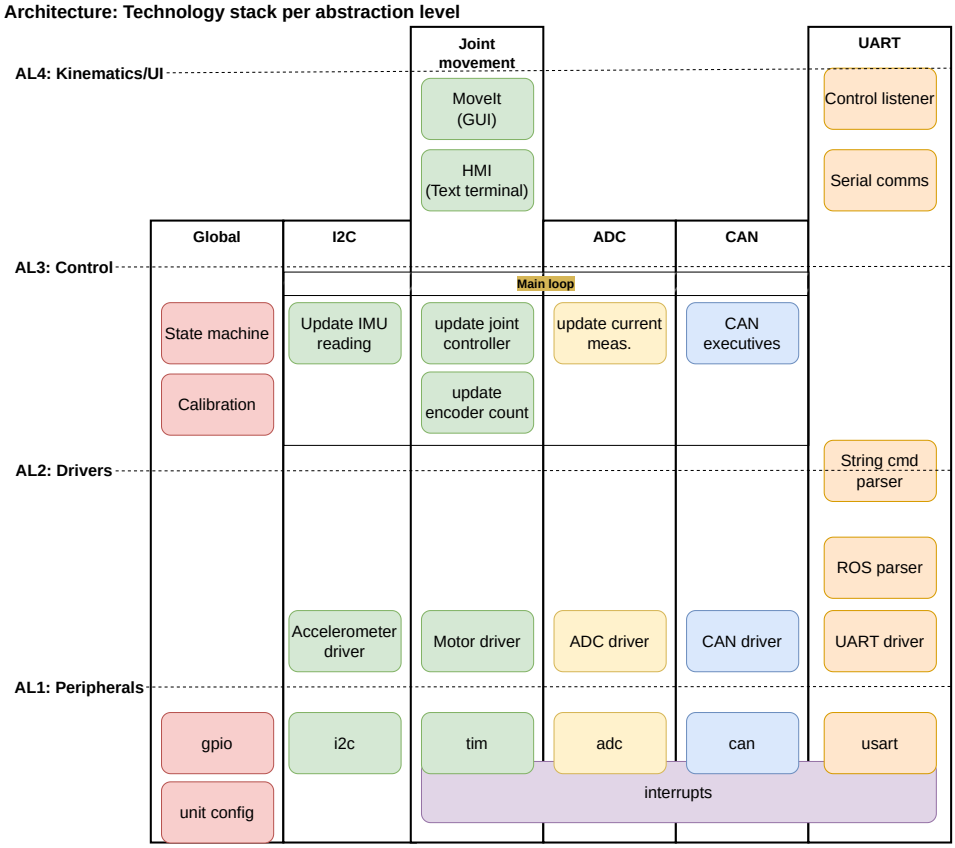


Figure 23: An overview of developed modules sorted by functionality group and abstraction level

11.4 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 12.

11.5 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 6.5.1), a master/slave structure would also need to be implemented on top of the CAN bus protocol, increasing overall system complexity.

11.6 CAN bus message ID structure

As discussed in section 7.2, 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 6.5.1). 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 12.5.

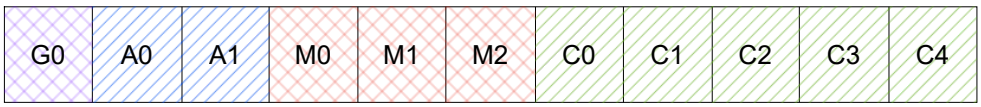


Figure 24: CAN bus message ID structure

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 24, 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.

11.6.1 Motor messages

Motor messages are for joint control and motor telemetry requests. 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.

11.6.2 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.

11.6.3 Global messages

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

11.7 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.

11.7.1 ROS vs Terminal

As the robotic arm may be controlled either automatically through MoveIt (see 11.9) or through a terminal with human-readable commands, two modules were written for the parsing of serial input data: `ros_uart_parser`[10] and `string_cmd_parser`. A preprocessor statement is used to enable one or the other.

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

11.8 Interrupts

As mentioned in section 10.8, 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.

11.8.1 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.

11.8.2 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 17.2.3.1 for the full calculation.

11.8.3 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 17.2.3.1 for the full calculation.

This section concludes the discussion of arm onboard modules.

11.9 ROS nodes and the external computer

As illustrated in figure 4, 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 8.1, 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 14.

11.9.1 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.

11.9.2 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 14) 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.

11.9.3 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 12.8). 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.

12 Implementation AL2: Hardware drivers

This section presents the hardware drivers written for the project, which include most of the modules defined in the `TTK4900_drivers` directory. Higher level control modules, such as the joint controller and main loop are discussed in section 13. The github repository for the project may be found at https://github.com/kristblo/TTK4900_Masteroppgave[5]. Descriptions in this section are meant to provide an explanation for the driver's key structure and functionality, not a complete understanding of the code base. The reader is encouraged to download the generated Doxygen/HTML documentation where most struct fields, functions et cetera are described in detail. Alternatively, the PDF is available under the name `Doxygen_documentation`, also in the github repository.

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

12.1 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[11]. The module's structure is illustrated in figure 25 along with the string parser and ROS input parser modules.

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

12.1.1 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`

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

12.1.2 Key functionality: Human input mode

The module provides a text based interface with the arm via the string parser module (12.2), 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):
    //'enter' key registered
    if(uartRxChar == '\r'):
        //Hand buffer over to string processor
        string_cmd_processor(uartRxHoldingBuffer);

        //Erase content and reset position counter
        clear_buffer();

    //Actual implementation has additional clause for backspace key.
    //Any other key registered:
    else:
        uartRxHoldingBuffer[bufferPosition] = uartRxChar;

        //Simple overflow protection
        *bufferPosition = (++(*bufferPosition))%64;

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

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.

12.1.3 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.

12.2 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 25 along with the UART module.

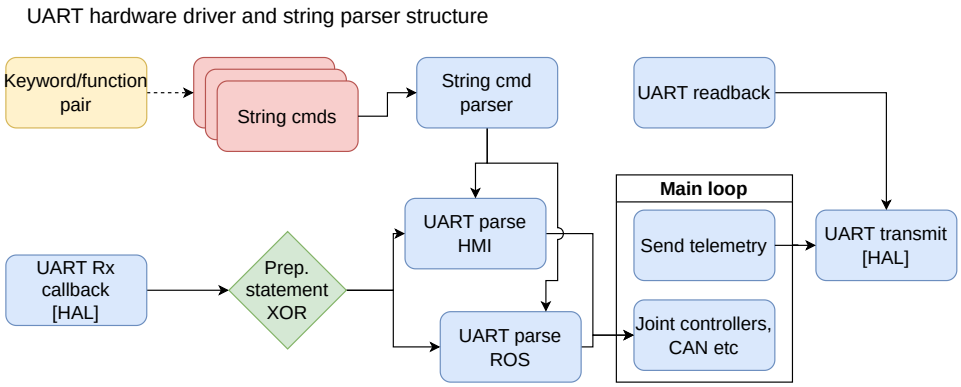


Figure 25: Structural diagram of the UART driver and string parser modules

12.2.1 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 (13.2).

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[12]. 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.

12.2.2 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:

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

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

//...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 SRP (6.9.1):

- 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.

12.2.3 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 was inspired by a thread on StackOverflow[2].

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 SRP. 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.

12.3 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 26.

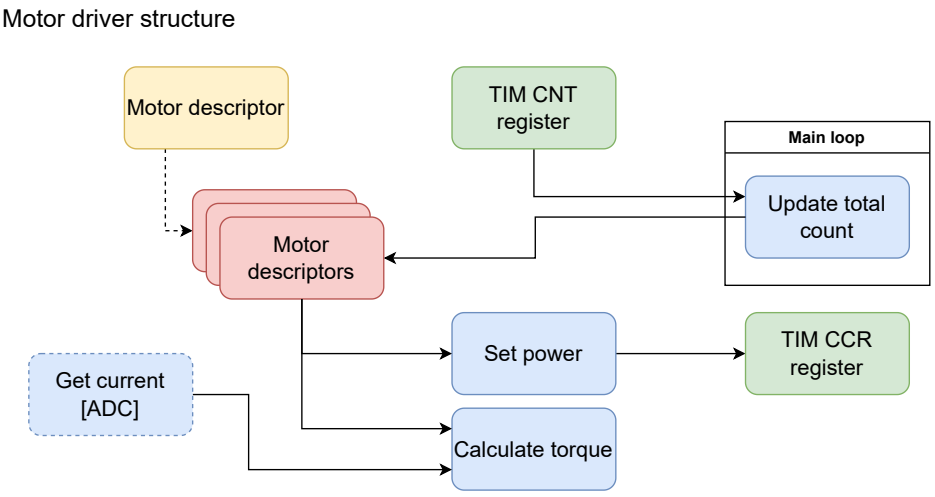


Figure 26: Structural diagram of the motor driver

12.3.1 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

12.3.2 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 13 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);
```

12.3.3 Key functionality: Motor power setting

As mentioned in section 7.2, 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[55], 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.

12.3.4 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 15.1). 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 26. 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.

12.4 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[45] 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 7.2, 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[45]). The resolution is 16bits, and converted measurements are stored in two adjacent single-byte addresses.

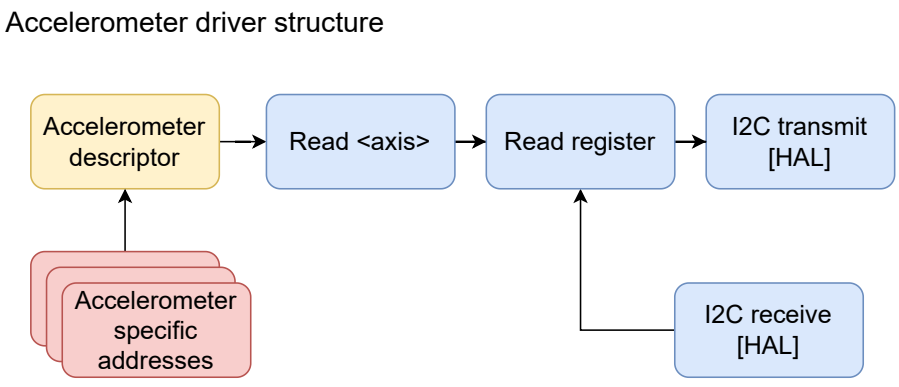


Figure 27: Structural diagram of the accelerometer driver

12.4.1 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

12.4.2 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 9.9.8, much time had been spent debugging the I2C/IMU hardware with little success. Consequently, optimising I2C bus access was not prioritised.

12.5 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 11.6, 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 28.

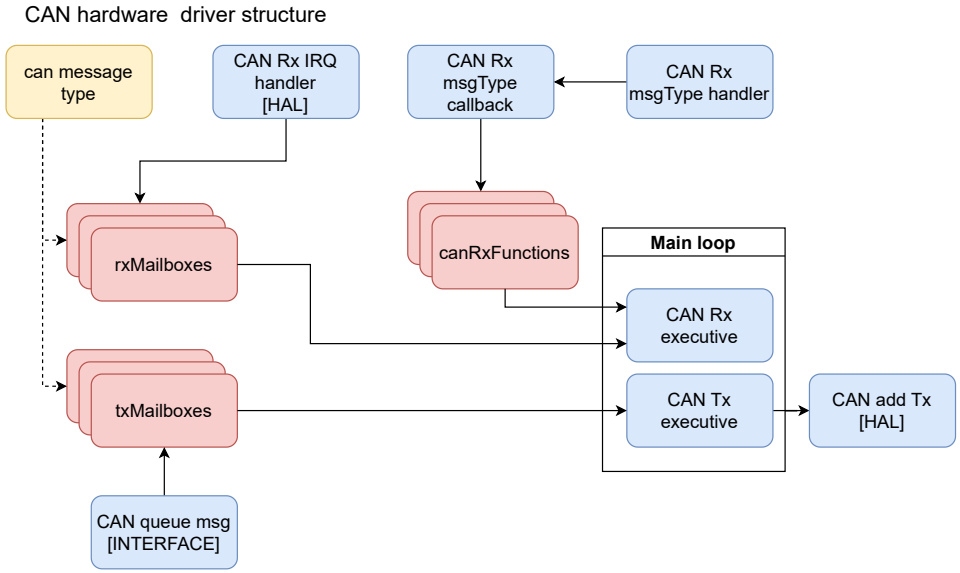


Figure 28: Structural diagram of the CAN driver

12.5.1 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.

12.5.2 Component: Message types, mailboxes and receive callbacks

Valid CAN message types are defined in an enumerated list forming the backbone of the driver. Message types correspond to the five LSBs of a CAN message ID discussed in section 11.6. A `can_mailbox` struct 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, 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 (12.2). 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
```

12.5.3 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;
```

12.5.4 Key functionality: Using ID bits to select hardware

As mentioned in section 11.6, 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 13.1), 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 (i.e. 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.

12.5.5 CAN message ID filter configuration

A discussion of CAN filter configuration was relegated here from section 10.4 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 category 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 13.

Parameter	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 13: CAN peripheral filter bank configuration

12.5.6 Comment: The number of supported message types

As mentioned in section 11.6, 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 24), 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.

12.6 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.

12.6.1 HAL interactions

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

12.6.2 Key functionality: Convert ADC values

As mentioned in section 10.3, 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 6.3).

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.

12.7 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.

12.8 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 29 illustrates the module.

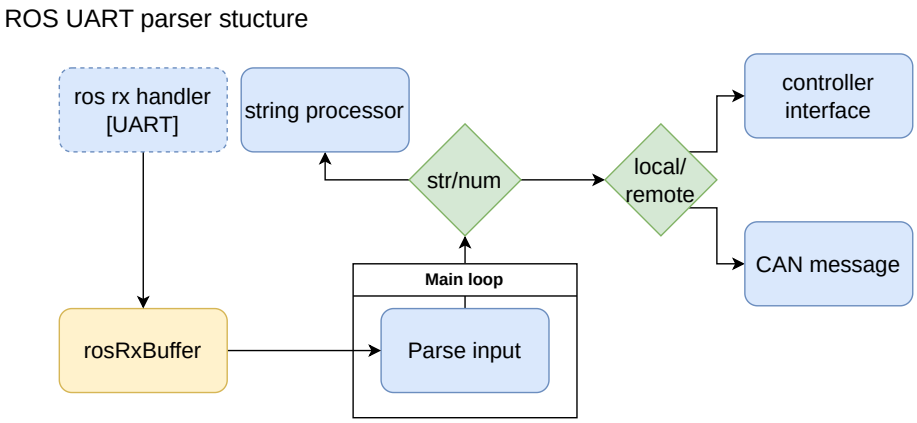


Figure 29: Structural diagram of the ROS UART driver

12.8.1 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 12.2. 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.

12.8.2 Key functionality: Parsing messages

12.8.2.1 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 13.1), 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.

12.8.2.2 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.

12.8.3 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.

12.9 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[13]. 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.

[13]For Motor, Accelerometer and Global, respectively

13 Implementation AL3: Control system

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

13.1 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.

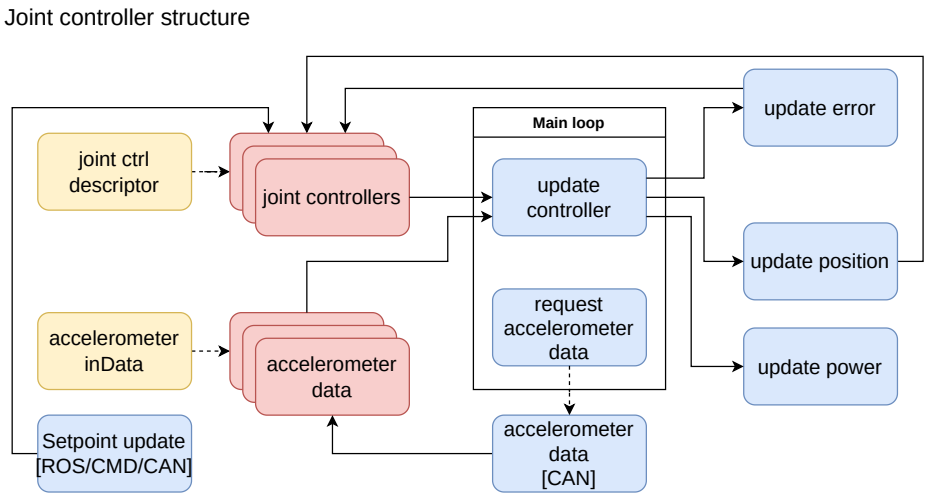


Figure 30: Structural diagram of the joint controller

13.1.1 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.

13.1.2 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 18.

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.

13.1.3 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 13.2), will be either whatever position the joint was in at powerup or its calibrated home position (see section 15.2). 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 3.

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.

13.1.4 Key functionality: Flags and joint state update

As described in section 10.8, 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.

13.1.4.1 joint controller update position: A joint's position is primarily calculated by converting encoder clicks to radians or meters. Encoder clicks are counted from a calibrated zero position (see section 15), and the function uses a conversion constant stored in the associated motor descriptor struct, `resolution` defined as clicks per radian. In the case of the shoulder joint, the function will use the latest accelerometer data when new data is available. 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.

13.1.4.2 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.

13.1.5 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, a discretisation of equation 12:

$$P = K_p E_t + \sum_{t_0}^{t} K_{pti} E + K_d (E_t - E_{t-1}), \tag{14}$$

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

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

13.1.5.2 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 large 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:

$$G_{kpti} = 1 - \frac{exp(20|E| - 5)}{1 + exp(20|E| - 5)}, \tag{15}$$

where G_{kpti} is the sigmoid gain. Numerical values 20 and 5 were set experimentally (see section 15) and ensure that the gain slope is appropriately shaped. The final expression for one iteration of the integral error is as follows:

$$I = K_{pti} G_{kpti} E, \tag{16}$$

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

13.1.5.3 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 $100\mu s$ to complete.

13.2 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 31. 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.

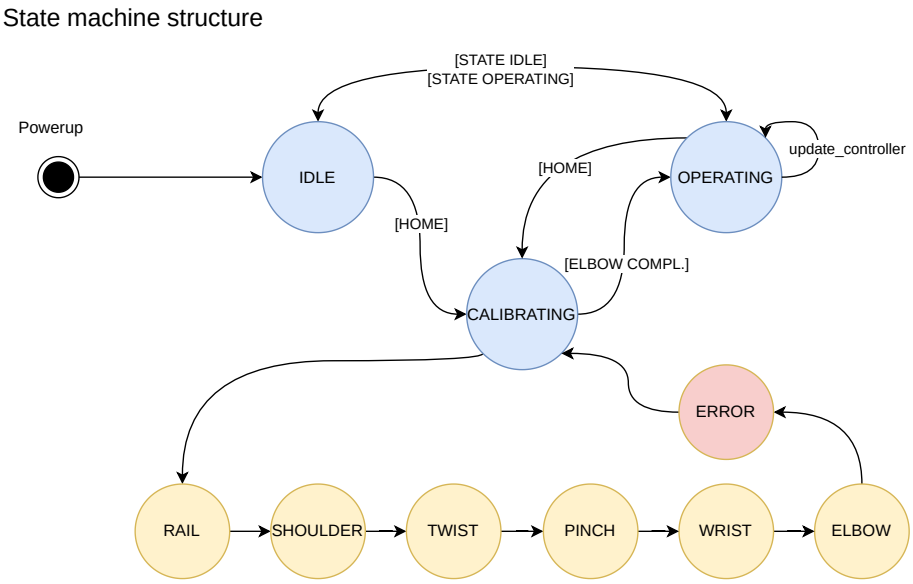


Figure 31: Structural diagram of the state machine. The error state is more accurately a "calibration complete" state. No error handling was implemented, and the state was repurposed without changing its name

13.2.1 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

13.2.1.1 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.

13.2.1.2 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.

13.2.1.3 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 15.2.

13.2.2 Key functionality: Broadcasting states

As discussed in section 11.6, 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 31).

13.2.3 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 15.2 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 31. 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`.

13.3 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 32. Blue bubbles represent a function or set of functions, and arrows represent the order in which (sets of) functions are called.

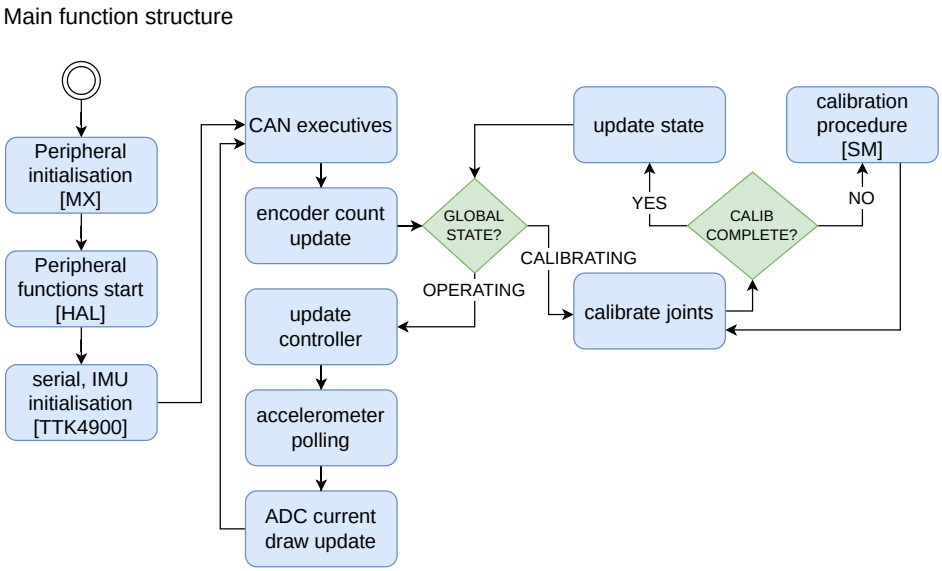


Figure 32: Structural diagram of the main function

13.3.1 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 10. 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 by referencing UM1786[47]. 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 12.4 using the relevant driver interface functions.

13.3.2 Loop

Loop items are primarily discussed in their respective subsections of sections 12 and 13. 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 32 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(calib_state == CS_WRIST):
        //...

    else if(calib_state == CS_ELBOW):
        //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
    //The following 'else' catches other evaluations of calib_state
    //for every control unit
    else:
        //Maintain joint position while others are calibrating.
        //By this point, joints' "0" positions will
        //be their calibrated home positions.
        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.

14 Implementation AL4: ROS nodes

As outlined in chapter 11.9, this project uses the Robotic Operating System (ROS) for high level control of the arm. ROS was integrated into the system at the expense of telemetric output from the arm, the consequences of which are further discussed in section 18. This section describes the individual programs, or nodes, which make up the ROS portion of the project, similar to the discussion of hardware drivers (12) and control system (13).

14.1 Summary

14.1.1 Nodes

This project defines four nodes as indicated by figure 21, one of which is the MoveIt kinematic solver and graphical user interface which motivated the use of ROS at the outset of the project's development. Remaining nodes fulfill a minimal set of functionality to pass MoveIt output to the arm and preserve string command functionality.

14.1.2 Packages

This project defines three packages: `control_listener`, which defines the three non-MoveIt nodes; `orca_ctrl_msgs`, which defines a message type exchanged by the same nodes; and `orca_moveit_config`, a configuration package which makes MoveIt compatible with the arm. Additionally, but not as a package per definition, a Universal Robotic Definition Format (URDF) file was written to provide MoveIt with a physical description of the arm. All code may be found under the `code/ros/src` directory.

14.2 MoveIt: Kinematics and GUI

MoveIt is a robotic control framework which runs on top of ROS and provides kinematic solvers, interactive 3D simulation and path/motion planning tools. When using the ROS input mode, movement of the arm is done via the 3D simulation module. The GUI lets the user drag the gripper to a desired position/orientation, then plan and execute the movement. Upon execution, provided that the other nodes are running, the arm will move to the planned position. The initial goal of the project was to develop this functionality to a point where higher level movement tasks, i.e. combinations of multiple movements in series, would be available, but this was not achieved. MoveIt is illustrated in figure 33.

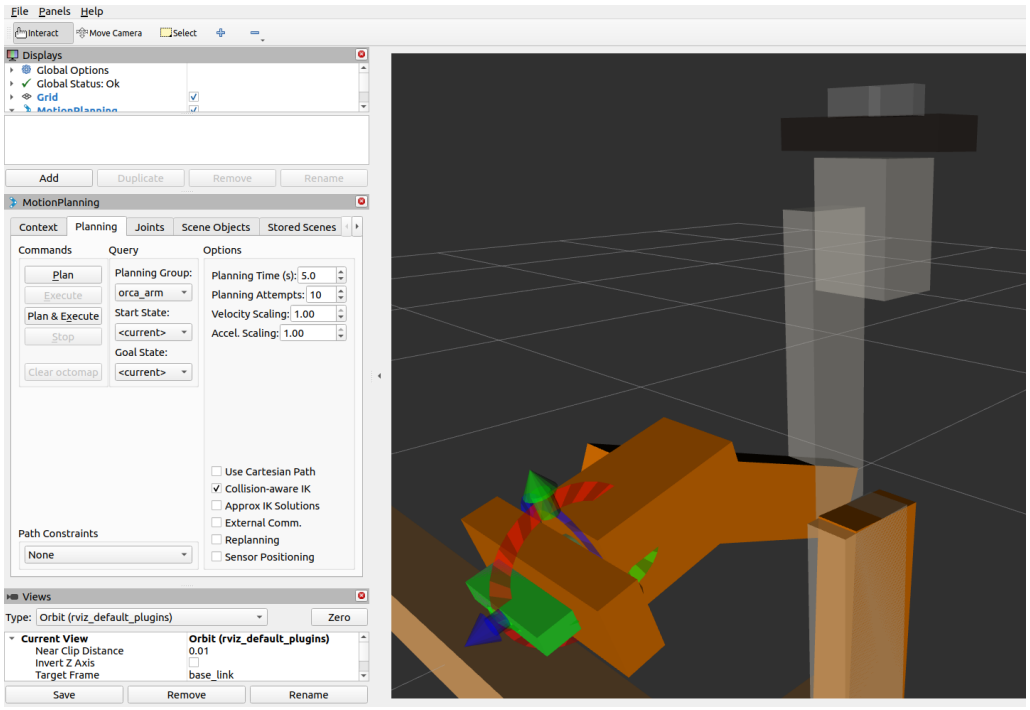


Figure 33: Example of gripper manipulation in MoveIt. The left hand context menu lets the user plan and execute a movement when arm model has been positioned

14.2.1 URDF: Robotic description

ROS systems use the URDF file format to describe physical properties of a robotic system. These may vary in fidelity, but key elements in a URDF file are *links* which describe individual components of a robot, and *joints* which describe how links are connected. The URDF file written for this project approximates all links as rectangular boxes for simplicity. Joints are defined with movement range, velocity and acceleration limits. Velocity limitation is an important safety precaution in the context of this project. While fast movements are generally desirable, the arm must be slow enough for a human operator to detect and respond to erroneous behaviour. Limits were set experimentally as a compromise between efficient movement and predictability, and are as follows:

- Rail: 0.1m/s
- Shoulder: 0.17rad/s
- Elbow: 0.17rad/s
- Wrist: 0.17rad/s
- Twist: 3rad/s
- Pinch: 0.01m/s

14.2.2 MoveIt setup assistant: Configuration wizard

MoveIt was configured using its built-in setup wizard, which lets the user import a URDF file and configure joint planning groups. An unresolved error led to the misconfiguration of the pinch joint, disabling MoveIt control for this joint. The wizard is illustrated in figure 34.

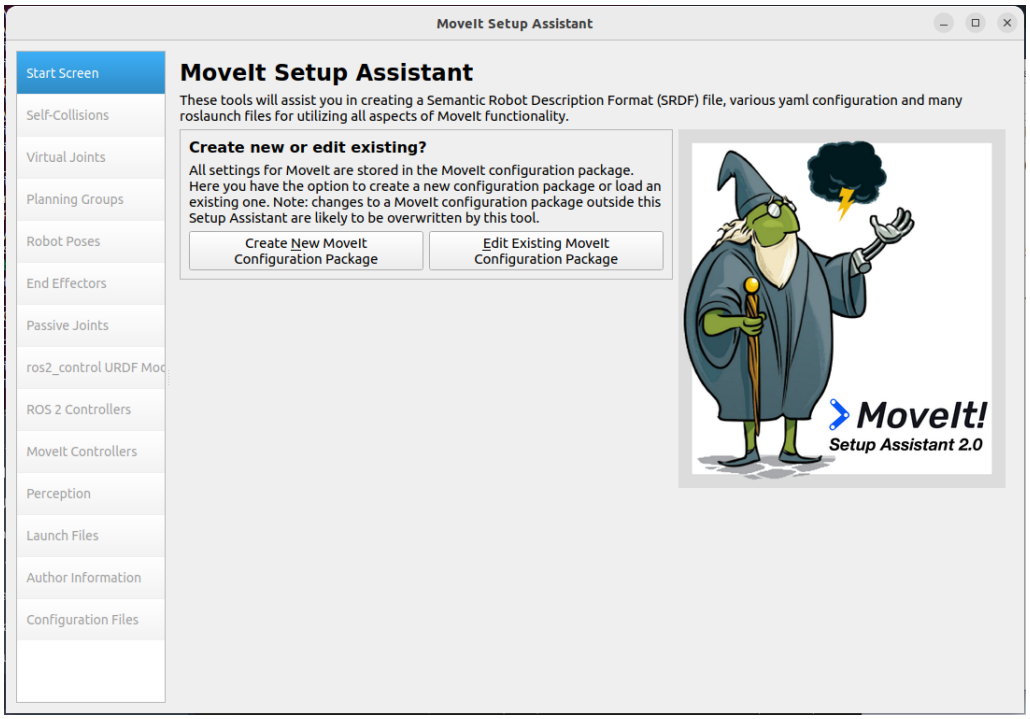


Figure 34: The setup assistant guides the user through configuration of MoveIt

14.3 Control listener: Parsing MoveIt

During the execution step of a movement, MoveIt will publish messages to a topic called `controller_state`. The control listener node was written to subscribe to this topic, filter out setpoints of the five successfully configured joints, and publish these on a topic `orca_ctrl` using the defined `Ctrl` message type at a frequency of 100Hz. This frequency limitation was imposed to prevent UART bus overload. At a UART bitrate of 115200 bps, the Control Listener contributes to a bus load of approximately 22%. In addition to joint setpoints, the `Ctrl` message defines the field `msgtype`. This field is a string, which the control listener fills with "num" for parsing by the ROS input module aboard the arm. The node is illustrated in figure 35, and the `Ctrl` message format is described in table 14.

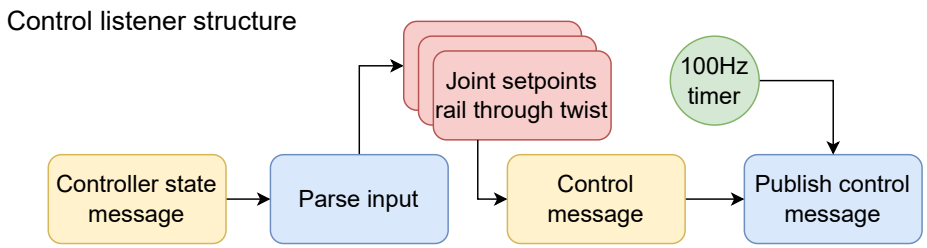


Figure 35: Structural diagram of the Control Listener ROS node

14.4 HMI: String command interface

The Human/Machine Interface (HMI) node was introduced to enable control of the pinch joint. It was determined that the shortest path to that goal was to include the string command parser in the arm ROS input module, and consequently the HMI node is simply a string input module to run in a terminal window. It publishes a `Ctrl` message on the `orca_control` topic, where the `msgtype`

Field	Type	Description
msgtype	string	Message type
strcmd	string	String command
pos_rail	float32	Rail setpoint
pos_shoulder	float32	Shoulder setpoint
pos_elbow	float32	Elbow setpoint
pos_wrist	float32	Wrist setpoint
pos_twist	float32	Twist setpoint

Table 14: Description of the ROS Control message format

field is set to "str". The user input string is stored in the `strcmd` field. No processing is done by this node: When the user presses the enter key, the preceding string is inserted in message and published. Strings must adhere to the patterns discussed in section 12.2. The node is illustrated in figure 36.

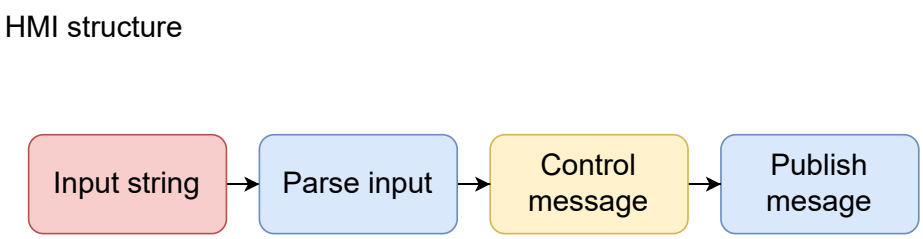


Figure 36: Structural diagram of the HMI ROS node

14.5 Serial comms: UART transmitter

The Serial Comms node bridges other nodes with the control computer's serial interface, passing data received via the `orca_control` topic onto the interface. It is only capable of transmitting, as a cursory attempt at receiving was not successful. When started, this node will configure the control computer's serial interface with parameters similar to those discussed in section 10.9. This process entails hardware register manipulation, and the node is likely only compatible with the Ubuntu operating system as a consequence. Configuration was done by modifying example code presented at the online blog Mbedded Ninja[14]. The node is illustrated in figure 37.

When a message is received, it will check whether the `msgtype` field is "num" (numerical, coming from the Control Listener node) or "str" (string, coming from the HMI node). In either case, message type and payload will be copied into a 32 byte array and sent to the serial interface by reading the relevant message fields and converting numerical values to bytes using C++ library functions. The UART transmitter node is naive to the message frequency of 100Hz, and will output messages as soon as they are received.

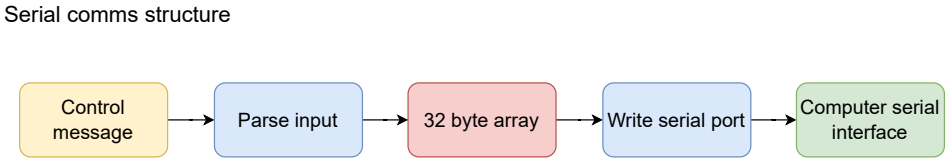


Figure 37: Structural diagram of the Serial Comms ROS node

15 Calibration

This section discusses various aspects of calibrating the arm for interaction with the real world. This included finding the relationship between encoder click counts and joint movement, devising calibration sequences for each joint as outlined in section 13.2 and, perhaps most importantly, tuning PID controllers for safe and reliable control of the joints.

15.1 Encoder clicks per movement, and motor polarity

In order for encoder data to be used effectively, the relationship between encoder clicks and joint movement had to be established for each joint. This size is defined in the `resolution` field of the motor descriptor (see section 12.3). For linear joints, the unit is clicks per millimeter. For rotational joints, the unit is clicks per radian. Resolution was estimated for each joint by moving them 10.000 clicks using string commands, and measuring their movement using a ruler (linear joints) or a digital level (rotational joints).

Figure 3 indicates positive directions for each joint. These are a consequence of hardware design: a joint's positive, or forward, direction is the direction of movement for which increments to its encoder count are positive. As positive/negative counts are a result of the phase between rising flanks reaching the encoder counter timer peripheral pins, the positive direction for a given joint is random in the sense that it was not taken into account in the hardware design phase.

The relationship between joint direction and motor direction motivated the introduction of the `motorPolarity` field. Depending on the installation of a motor's power connector plug, a call to `motor_interface_set_power` with a positive power setting may result in the joint moving backward. This is undesirable, and the callees of `motor_interface_set_power` were modified to take polarity into account. If a motor is installed such that a positive power setting would nominally make the joint move forward, then its polarity was set to 1. Conversely, if a positive power setting would nominally make the joint move backward, then its polarity was set to -1. The result is that a positive power setting always makes the joint move forward, that is, cause a positive increment to its encoder counter. Table 15 presents resolutions and polarities.

Joint	**Resolution [ec/rad][ec/mm]**	**Polarity**
Rail	99	-1
Shoulder	19022	-1
Elbow	36097	1
Wrist	17700	1
Twist	8881	-1
Pinch	460	1

Table 15: Joint resolution and polarity

15.2 Joint position calibration

A key aspect of autonomous operation of the arm is its ability to estimate its position relative to the environment; its pose. With one IMU in each arm section (upper arm, lower arm, hand), estimating the arm's pose would be straightforward: measure acceleration in relevant axes while the arm is standing still to obtain the orientation of the IMUs relative to gravity, and compare to known physical parameters of the arm. With only one accelerometer operational, pose estimation must rely almost entirely on encoder data. The system's encoders are relative, meaning that they start at 0 when the arm is powered ("powerup"). The arm's pose cannot be assumed to be known on powerup, and it is therefore essential to operation of the arm to find a certain "home", or zero, position for each joint from which setpoints may be calculated. This section describes the calibration sequence outlined in section 13.2 in detail.

15.2.1 Overview of home positions

Rail : Activation of the end switch, defined as 0 m.

Shoulder : Accelerometer reading of 0g in the Y axis, i.e. vertical, defined as 0 rad.

Elbow : Upper and lower arm at 180°, i.e. vertical, defined as 0 rad.

Wrist : Hand flush with lower arm, pincer pointing away from arm, i.e. vertical, defined as 0 rad.

Twist : Pincer mount aligned with (the physical) rail, defined as 0 rad.

Pinch : Pincer closed, defined as 0 m.

15.2.2 Calibration procedure: Rail joint

Calibration happens entirely within the function `state_calibrate_rail`, called by the main loop. Joint moves at 20% power until the end swith interrupt is triggered. The power setting ensures the joint moves with the minimum velocity/torque necessary to trigger the switch once at a system voltage of 25V. No debounce of the switch has been implemented, and higher power settings were found to activate the switch multiple times, disturbing the calibration sequence. Both the joint controller field `posCurrent` and motor descriptor field `encoderTotalCount` are set to 0 immediately after switch activation. The joint then moves at -10% power for one second, or approximately 20 mm, in order to ensure that subsequent calibration steps do not trigger the end switch. Joint position is finally updated to correspond with the last movement, and calibration state is set to `CS_SHOULDER` in the main loop.

15.2.3 Calibration procedure: Shoulder joint

Calibration happens partially in the main loop, and partially in the function `state_calibrate_shoulder`. Correct calibration of the shoulder joint is critical to calibration of subsequent joints, and the goal is to set the upper arm in a vertical position. Calibration starts with a joint setpoint of 0, and the main loop will poll the shoulder IMU as described in section 13.3 between calls to `state_calibrate_shoulder`. This function calculates the joint's position, error and power setting based on the latest accelerometer data. In the main loop, `encoderTotalCount` and `posCurrent` are set to 0 when the absolute error is less than 0.04 rad. Calibration state is finally set to `CS_TWIST`, handing control over to the hand control unit.

NOTE: The shoulder IMU was disabled before a public demonstration of the arm due to stability issues by setting the joint descriptor field `hasAccelerometer` to 0. In this configuration, it is the user's responsibility to move the arm to a vertical position before powerup.

WARNING: The arm may act erratically, unpredictably and violently when the IMU is active. DO NOT USE in public until properly debugged and verified.

15.2.4 Calibration procedure: Twist joint

Calibration happens entirely within the function `state_calibrate_twist`. Twist calibration has to happen after shoulder calibration as wrist calibration is physically dependent on the pincer mount being perpendicular to the rail. The goal for the calibration is for the pincer mount to be parallel with the rail; "horisontal". Similar to rail joint calibration, the twist joint is moved at 10% power until the twist sensor is triggered. Motor encoder count is set to zero immediately after. The sensor is approximately 30° off horisontal, corresponding to -10900 encoder clicks, and the joint is moved at -10% power until this encoder count is reached. The joint's position is then set to zero along with the motor encoder count. Finally, the joint is given a setpoint of 1.5708 rad or 90° to accommodate wrist and elbow calibration, and the calibration state is set to `CS_PINCH`.

15.2.5 Calibration procedure: Pinch joint

Calibration happens entirely within the function `state_calibrate_pinch`. The goal is to find the pincer's fully closed position. The pinch joint has no sensor indicating its absolute position. Instead, it relies on the joint physically stopping while under power. Power is set to -10%, causing it to close at a rate of approximately 10mm/s. Power is cut when the motor descriptor field `mostRecentDelta`, defined as the difference between the most and second to most recent encoder counts, reaches 0. Joint position and encoder count are set to 0 immediately after. The calibration state is finally set to `CS_WRIST`, handing control over to the shoulder control unit.

15.2.6 Calibration procedure: Wrist joint

Calibration happens entirely within the `state_calibrate_wrist` function. The goal is for the hand to be flush with the lower arm, pincer mount pointing away from the arm. This would originally be accomplished using the hand accelerometer. With this being defunct, the wrist joint instead uses the same method as the pinch joint. The joint will stop mechanically at 185°, referring to figure 3 and 0° as described. Joint power is set to 15% until it stops, i.e. `mostRecentDelta == 0`. After stopping, the joint will move backwards at -15% power for 55600 encoder clicks. Neither movement would be physically possible had the twist joint not previously been set to 90°. At this point the hand will be flush with the lower arm, and the joint position and encoder count are set to 0. Calibration state is set to `CS_ELBOW`.

15.2.7 Calibration procedure: Elbow joint

Calibration happens entirely within the `state_calibrate_elbow` function, and follows the same pattern as pinch and wrist. The goal is for the lower arm to end in a vertical position. Provided that the shoulder, wrist and twist are at 0°, 0° and 90° respectively, the joint will be able to move freely until the lower arm collides with the torso. At that point the elbow angle will be $-165°$, or 103952 encoder clicks, relative to its 0 at vertical, again provided that the shoulder is vertical. The joint is set to 15% power, which at 25V input voltage is minimally sufficient for it to reach the vertical position. Joint position and encoder clicks are set to 0 when the position has been reached. Finally, calibration state is set to `CS_ERROR`, and global state is set to `GS_OPERATING`.

15.3 IMU axis offset

During testing of the IMU it was discovered that it had not been installed perfectly level relative to the upper arm chassis. While the upper arm was measured to be vertical using a digital level, the IMU would output a Y axis measurement of 2300, which for a $\pm 2g$ range at 16 bits of resolution corresponds to approximately $-8°$. Consequently, an offset of 2300 is subtracted from all IMU Y axis readings. This bias may probably be attributed to uneven application of solder paste during assembly of the shoulder control unit, and/or uneven installation in the arm chassis.

15.4 PID tuning

15.4.1 Overview of implemented PID controllers

The PID controllers implemented in this project control the torque of individual joints, with power setting (P) as the controlled variable. Power setting is defined as percentage of system input voltage, which relates to developed torque (T) via the torque constants (K_t) of individual motors. The process variable is the joint's position (p) as measured via encoder or accelerometer. The relationship between power setting, torque and position may be described as follows:

$$T = K_t \frac{V_i}{R_m} P, \dot{\omega} = T - T_f \implies p = \int T - T_f dt^2, \tag{17}$$

where T_f is an unknown dynamic friction individual to the given joint, V_i is the input voltage and R_m is the real impedance of the motor under control. While K_t, R_m and T_f are presumed constant, V_i may vary depending on the available power supply. A set of PID gains is only valid for a single value of V_i, as is implied by equation 17. The power setting controller is implemented as described in section 13.1.5.

One weakness of this controller implementation is the lack of direct velocity control. Other control schema were considered, such as a cascaded controller[25] where the inner loop controls velocity and the outer loop controls position. Velocity ($\dot{p}$) could be measured using a timer to measure the time passed since last update to the joint controller, and calculating the distance travelled (Δp) since then. This could not be implemented in due time. As a compromise, the PID controller was tuned for small increments in setpoint ($\leq 0.3rad$) with no expectation of stability for larger increments, and velocity control was relegated to MoveIt.

15.4.2 Tuning methods

In a 2010 whitepaper, Shamsuzzoha and Skogestad[42] describe a method of tuning PID gains by a method they name the "Setpoint overshoot method" as a safer, more robust alternative to the classical Ziegler-Nichols (ZN) method. In brief, this method involves measuring step response in the system using a P controller with K_p sufficient to yield in a single setpoint overshoot which would then be analysed to retrieve appropriate PID gain values. Unlike ZN, this method does not involve bringing the system to its stability limit. While interesting, using it was opted against for multiple reasons:

- Time: ROS had been implemented in favour of telemetry sufficiently sophisticated to use this method.
- Setting: The lab in which the system was developed was too small and primitive to safely conduct the necessary experiments.
- Friction: All joints, possibly except for the elbow, have too high dynamic friction to yield overshoot at the voltage levels available.
- (Non-)Linearity: properties of the elbow and shoulder joints are mutually dependent on the other's state. Treating this system as LTI with static PID gains is arguably inappropriate. Carefully following a tuning scheme under these circumstances seemed not worthwile.

ZN was discarded for the same reasons, with particular emphasis on the second and last points. In the end, each joint was tuned qualitatively: time to stabilise was measured using a stopwatch to time the interval between passing of a setpoint via string command and stabilisation of the joint at the setpoint.

15.4.3 Tuning

Tuning was initially planned for 20V, 25V and 30V as collecting response data from multiple voltage levels may be interesting with regards to characterising the system. However, the CAN bus was found to be intermittently unresponsive at 30V (see section 18), and several problems were found at 25V. Solving these problems took priority over tuning at 20V. Thus, only one set of PID gains were found, at 25V.

As mentioned in section 13.1.5, the controller implements a sigmoid gain as a countermeasure to integral windup. The joints generally have high mechanical friction, and initial attempts at tuning found that aggressive tuning, i.e. high values for K_p and/or K_{pti} were needed to overcome the static friction of the rail and wrist joints in particular. However, lower dynamic friction and high inertia of the arm's links made aggressively tuned controllers overshoot their setpoints considerably. Lower values of K_p would lower the risk of overshoot, but this would lead to the integral term winding up at the beginning of a motion during the static phase – essentially, joints would only

start moving when the value of the integral term was sufficiently high, and no value of K_d would prevent significant overshoot. Thus, the sigmoid was introduced on the following premise: *A high K_p is needed to overcome static friction, a moderate K_d is needed to prevent overshoot, and the integral term should be minimally active until a joint is close to its setpoint in order to counteract damping by the derivative term.* A discrete approach where the integral term was either active or inactive depending on the absolute size of the error was tested, but found to induce jolts as the power setting would suddenly increase near the setpoint.

The sigmoid is illustrated in figure 38, where the use of the absolute value of E makes it look deceptively like a bell curve. For simplicity, the shape is the same for all joints. This means that the integral term is essentially inactive for the linear joints until they are less than 0.5 mm away from a setpoint, and 0.5 rad for the rotational joints. As sub-millimeter accuracy was not attainable, the sigmoid function was eventually disabled for linear joints. Final PID gain values are presented in table 16, and the tuning process is described in section 16.

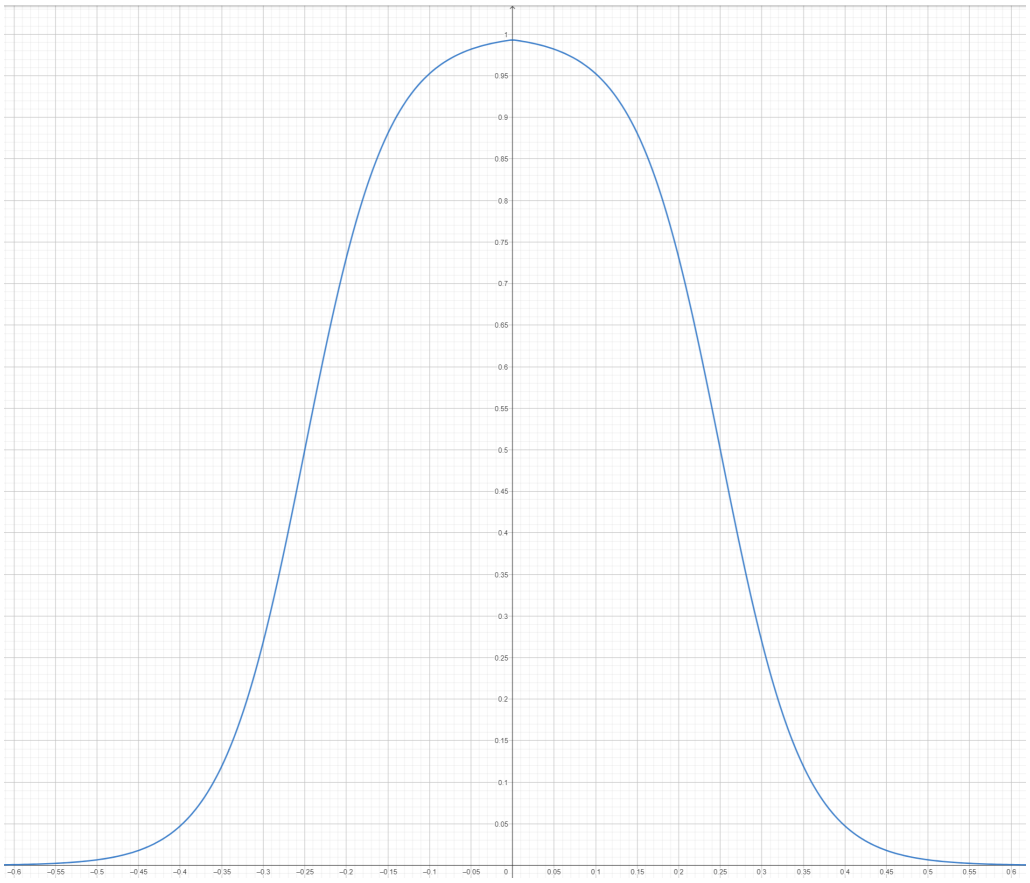


Figure 38: "Sigmoid" used to modify PID I term gain, first presented in equation 15. Y: gain value. X: Error in rad or mm

Table 16: Final PID gain values

Joint	**Kp**	**KpTi**	**Kd**
Rail	0.5	0.0002	0.1
Shoulder	60	0.02	10
Elbow	50	0.005	30
Wrist	50	0.07	20
Twist	40	0.01	7
Pinch	20	0.01	0

16 Tests

This section describes tests done to evaluate various aspects of the system's performance. Due to the limited time available for development of telemetry readouts and availability of diagnosis tools such as a debugger, tests were often ad hoc and qualitative; no time was dedicated to the systematic development of tests and/or success criteria. Test were instead devised as needed, usually with a specific improvement in mind, and often stemming from the result of a previous test. This section therefore also presents certain intermediate results, with section 17 acting as a summary. Tests, results and the impact of this ad hoc approach are discussed in section 18.

16.1 Motor overcurrent

In order to prevent overcurrent in the motor drivers, ADCs were configured to trigger an interrupt if motor current exceeds 4A, the rated limit of the motor drivers[55]. The interrupt handler was then configured to disable motor relays present on the torso and shoulder control units. In order to test this functionality, the ADC was configured to trigger at approximately 100mA as this was typically observed during operation of individual joints via readouts from the power supply. Wrist and shoulder joints were tested by manipulating them away from a setpoint of 0. In both cases, the power was cut shortly after the joints started compensating for manipulation. This was considered a positive result, and the current limit was increased to 4A. No test could be devised to safely test this limit explicitly.

16.2 PID tuning

Joints were tuned individually by testing their stabilisation times at a range of setpoint deltas, as mentioned in section 15.4, and presented in ascending order per figure 3. Movement characteristics of the shoulder and elbow joints are mutually dependent: moment of inertia acting on the shoulder joint depends on the angle of the elbow joint, while the load acting on the elbow joint for a given angle depends on the shoulder joint angle. For consistency and simplicity, both joints were tuned while the other was at 0°, vertical. Tests described here were undertaken at an input voltage of 25V. Generally, joints were subjected to setpoint deltas of 0.3, 0.1 and 0.01 rad (or meters), and "time to stabilise" was measured between the passing of a setpoint and no movement was visible using a stopwatch. Some remarks are made about the joints' friction. These are entirely qualitative, and were "measured" only by casual observation.

Stabilisation times at small deltas are likely more relevant to performance than large deltas. Recall that ROS was configured to a setpoint refresh rate of 100Hz, and that movement velocity limits were set to 0.17rad/s for most joints. While ROS outputs absolute setpoints, the refresh rate implies that deltas will be on the order of 0.002rad or 0.1°. This is too small a delta to evaluate by observation, hence the lowest test delta of 0.01rad.

After an initial round of tuning, a bottle of approximately 750g was attached to the gripper and PID gains were reevaluated. Initial tuning was done with ROS/MoveIt disabled.

16.2.1 Initial tuning

16.2.1.1 Rail: Test values are displayed in table 17. Values to the left of the vertical separator display gain values tested on a set of deltas, and time values to the right are the time to stabilise for the set of gains and value of delta. For instance, a gain set of $Kp = 0.25, Kpti = 0.0001, Kd = 0.25$ resulted in 17s to stabilise at a delta of 0.3m, 30s at 0.1m and infinite time at 0.01m. Kd was initially set equal to Kp on the hypothesis that it would prevent overshoot by quickly reducing the power setting when the rail joint started moving.

The first test indicates how high static friction in the rail joint impacts the controller as a smaller delta resulted in higher time to stabilise: For high deltas, the proportional term yields a high initial

power setting and immediate movement, giving the integral term little time to wind up before overshoot and eventual wind-down, causing the typical decaying oscillation around the setpoint before settling. At the smallest delta of 0.01m, however, the proportional term was likely too low for the controller to overcome static friction, resulting in a long delay before movement while the integral error accumulates and eventually caused a jerking motion overshooting the setpoint and repeating the process. The middle result of 30s at 0.1m delta is likely a mix of those extrema, where a high Kd (relative to Kp) also contributes to a longer time to settle compared to 0.3m delta.

In the second test, Kpti was increased to 0.0002 to test the friction hypothesis further, resulting in slightly stronger oscillations at 0.01m delta. During this test, the integral windup was readily audible: A high-pitched noise of increasing amplitude could be heard from the controller in the moments before movement was observed. Kd was increased to 0.5 in the third test, resulting in a stabilisation at the setpoint after approximately one second with minimal overshoot. Time to stabilise was not recorded for larger deltas during the second and third tests.

Test no.	Test parameters				Time[s]
	Kp	Kpti	Kd	Delta [m]	
1	0.25	0.0001	0.25	0.3	17
				0.1	30
			0.01	inf	
2	0.25	0.0002	0.25	0.3	-
				0.1	-
			0.01	inf	
3	0.50	0.0002	0.25	0.3	-
				0.1	-
			0.01	1	

Table 17: Initial rail test values

16.2.1.2 Shoulder: Test values are displayed in table 18, note that deltas are in rad. Initial values were selected arbitrarily as $K_{kpti} = K_p \frac{1}{20} = 0.2, K_d = K_p \frac{1}{4} = 10$ and tested at 30V. This resulted in 25s of decaying oscillation for a delta of 0.3 rad, and Kpti was immediately set to 0.02. This resulted in significantly improved performance, 15s at 0.3 rad delta. At this point, voltage was finally reduced to 25V as the CAN bus failed repeatedly at 30V. The decrease in voltage resulted in a marginal reduction in stabilisation time, from 15s to 12s. The observed oscillations were high in amplitude and low in frequency, indicating a low Kp. Kp was increased to 60 in the last test, which yielded a response time of less than a second with no visible overshoot for a delta of 0.01rad.

16.2.1.3 Elbow: Test values are displayed in table 19. Initial values were chosen similarly to the shoulder, but with a lower Kpti from the outset of testing. Some oscillation was observed, and the joint stabilised after approximately 6 seconds. No movement was observed at 0.01 rad delta, which motivated the increase in Kp in the second test. This lead to a significant decrease in performance for 0.1 rad delta, from 1s to 5s due to overshoot and oscillation. Kp was reduced back to the inital value, and Kd was increased on the hypothesis that higher damping would decrease oscillation. This was observed, but at the cost of marginally longer time to stabilise at 0.3 rad delta compared to the first test. Kd was reduced to 0 in the final test with no apparent impact on performance.

Gains were restored to their initial values, generally because they resulted in the best performance, and specifically Kd because it was assumed that some degree of damping would be beneficial: Testing was done with the joint near the verical position, where the fact that the joint may support itself unpowered implies that gravity load is negligible compared to joint friction. Gravity load would be comparatively significant in operational scenarios where the elbow will need to support the lower arm and a payload, likely necessetating some damping.

Test no.	Test parameters				Time[s]
	Kp	Kpti	Kd	Delta [rad]	
1	40	0.2	10	0.3	25
				0.1	17
	@30V			0.01	7
2	40	0.02	10	0.3	15
				0.1	10
	@30V			0.01	2
3	40	0.02	10	0.3	12
				0.1	-
				0.01	1
4	50	0.02	10	0.3	10
				0.1	-
				0.01	-
5	60	0.02	10	0.3	8
				0.1	5
				0.01	1

Table 18: Initial shoulder test values

Test no.	Test parameters				Time[s]
	Kp	Kpti	Kd	Delta [rad]	
1	50	0.01	12	0.3	6
				0.1	1
				0.01	inf
2	60	0.01	12	0.3	6
				0.1	5
				0.01	inf
3	50	0.01	20	0.3	7
				0.1	-
				0.01	-
4	50	0.01	0	0.3	7
				0.1	-
				0.01	-

Table 19: Initial elbow test values

16.2.1.4 Wrist: Test values are displayed in table 20. Initial values were chosen similarly to the shoulder, but with a lower Kpti from the outset of testing. Similar to the rail, the wrist joint appears to have high static and/or dynamic friction. The first test resulted in some overshoot before stabilising after seven seconds. Learning from tuning of the rail joint, Kp and Kd were increased in an attempt at reducing overshoot, with no change in behaviour. Kpti was then increased to 0.2, which yielding a similar result as for the rail: Improved stabilisation time for larger deltas, and worsened for the smallest delta.

The wrist joint will need to support varying payload weights without "drooping', and thus needs to be resistant to perturbations. An attempt at manually turning the joint while powered and using the gain values of the fourth test indicated that it would react slowly to an increase in payload. The joint did move back to its original position, but only after several seconds (exact time was not recorded due to the author being busy manipulating the powered arm). In an attempt at improving resistance, Kpti was increased. This made the joint significantly more resistant to manipulation, but, as can be seen from the fifth test, also made it unstable for small deltas. Kpti was reduced to its final value as a compromise and on the assumption that it would be sufficiently stable when carrying a payload.

Test no.	Test parameters				Time[s]
	Kp	Kpti	Kd	Delta [rad]	
1	40	0.01	20	0.3	7
				0.1	-
				0.01	-
2	50	0.01	20	0.3	7
				0.1	-
				0.01	-
3	50	0.01	25	0.3	7
				0.1	-
				0.01	-
4	50	0.02	25	0.3	3
				0.1	1
				0.01	5
5	50	0.04	25	0.3	5
				0.1	3
				0.01	inf
6	50	0.03	25	0.3	-
				0.1	-
				0.01	-

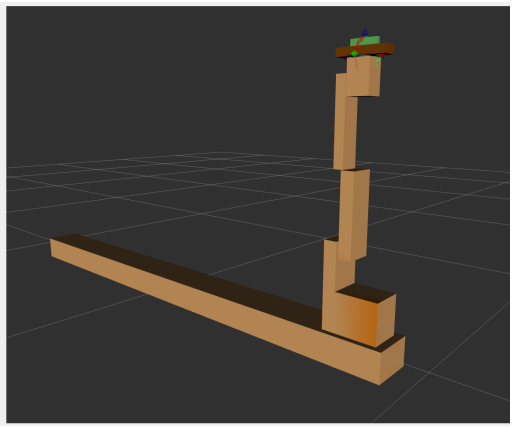
Table 20: Initial wrist test values

16.2.1.5 Twist: $Kp = 40, Kpti = 0.01, Kd = 7$, resulting in 3 seconds to stabilise at a delta of 1.57 rads with minimal overshoot and previous tests indicating no decreased performance at smaller deltas.

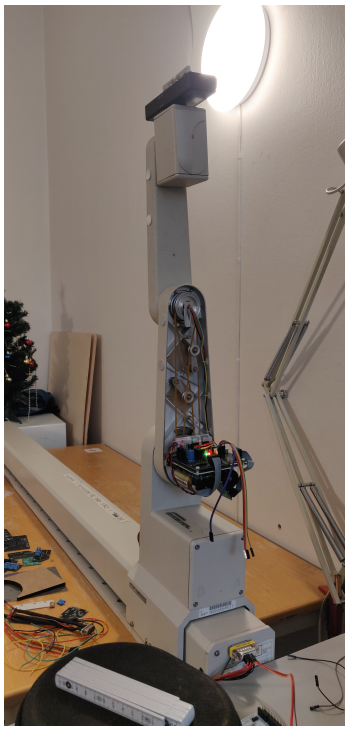
16.2.1.6 Pinch: Dynamic friction in the pinch joint is high enough compared to pincer inertia that a P controller would be sufficient, and no safe input voltage would result in "overshoot", i.e. continued movement after power cutoff. Thus Kd is set to 0. Maintaining grip strength while carrying a payload is essential, which motivated the introduction of a nonzero Kpti. If the pincer is given a setpoint of 0 while an item is placed between the gripper fingers (see figure 3), the integral term would ensure that the fingers are gripping with maximum strength shortly after the fingers close around the object. While this may be harmful to fragile objects, given the arm's primary use case, guaranteeing a strong grip seems like the safer option. $Kp = 20, Kpti = 0.01, Kd = 0$.

16.3 MoveIt, bottle carrying

Following initial tuning, ROS/MoveIt was enabled to evaluate tuning parameters in an operational setting. MoveIt was configured to store the "calibrated" position of the arm with all joints at 0 except for the rail joint at -0.03m. When the ROS serial bridge is activated, the twist joint is immediately moved from its position of 1.57 rad (its final position after its calibration procedure) to 0. The result of this syncronisation is displayed in figure 39. Note that the gripper fingers visible in figure 3 had not yet been produced when these tests were undertaken.



The "calibrated" position in MoveIt

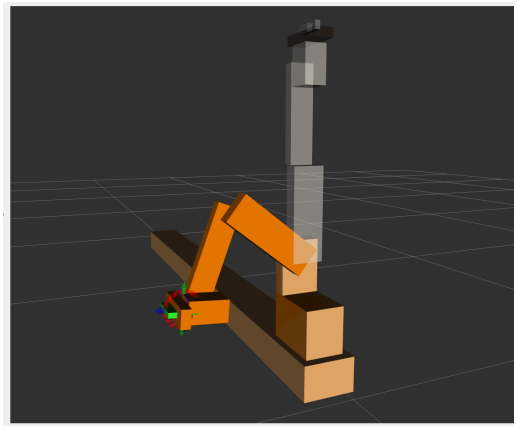


The arm after calibration and syncronisation with MoveIt

Figure 39: Arm calibrated, syncronised and ready for operation with MoveIt

An initial test of movement planning and execution using MoveIt was uploaded to a YouTube playlist TTK4900[6] under the name *Simple movement test with MoveIt*. See section 18.3.1 for discussion.

Figure 40 illustrates movement planning in MoveIt, where the gray shadow represents the current position and the orange solid represents the planned end pose. Execution of this movement is also displayed. Some overshoot was observed in the shoulder and elbow joints, indicating that PID gains were acceptable but suboptimal. The arm's pose after stabilisation is quite similar to the planned pose, indicating that encoder resolution measurements are generally accurate. The shoulder joint setpoint is approximately 1.3 rad. When the arm stabilised in this position, a movement of the rail joint was planned. The shoulder joint slammed downwards at very high velocity immediately after execution, indicating an error relating to the IMU. This failure mode was observed during multiple tests, and debugging is discussed in section 16.5.



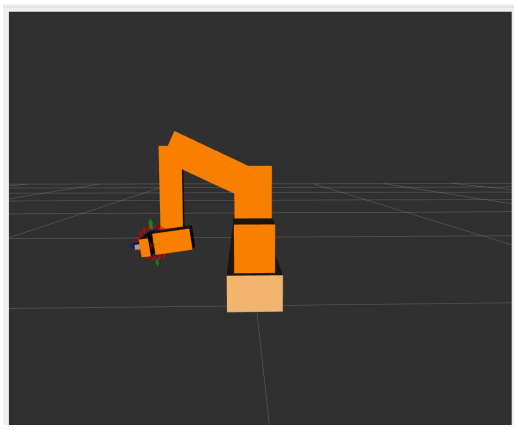
Arm movement planning in MoveIt

Execution of the planned movement from the calibrated state

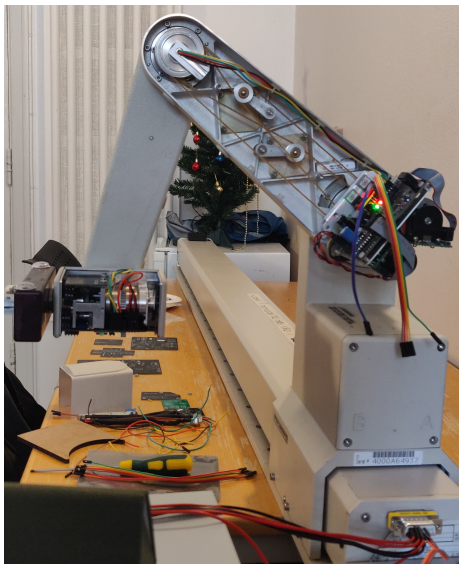
Figure 40: Movement planning and execution using MoveIt, performance test of initial PID tuning

Figure 41 illustrates movement between a pose close to its maximum movement range before self-collision and a pose where the hand is raised along the near-vertical (green) axis relative to the initial pose. In the initial position, the actual angle achieved by the elbow joint is visibly lower than the planned angle. This could indicate that the resolution measured for the elbow joint encoder is too low; more encoder clicks should have been counted in order to acheieve the desired joint angle. However, calibration was successful using the measured resolution, indicating that it is correct. An alternative explanation could be that there is some degree of mechanical coupling between the shoulder and elbow joints. Such a coupling was previously observed between the twist and pinch joints, where the pincer would move approximately 1cm per full rotation of the twist joint. No test could be devised to test this hypothesis, but the observation highlights a weakness of relying purely on encoder data. Had there been an accelerometer in the lower arm, data from that unit could have been used to more accurately estimate the elbow angle.

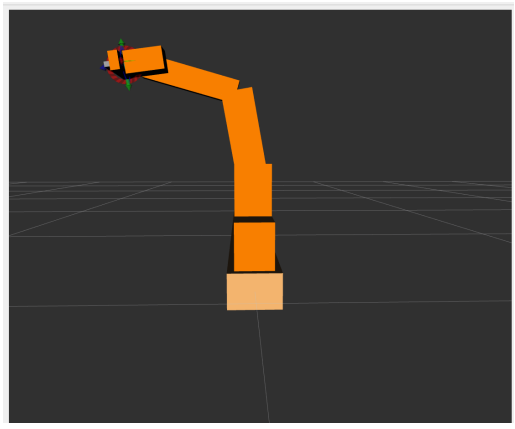
The second appears to have been achieved with high accuracy, indicating that the previous inaccuracy is related to the joints being near their maxima. As in the previous test, some oscillation was observed as the joints neared their final setpoints.



Initial position before second movement test in MoveIt



Actual initial position before second movement test



Planned end position of second movement test

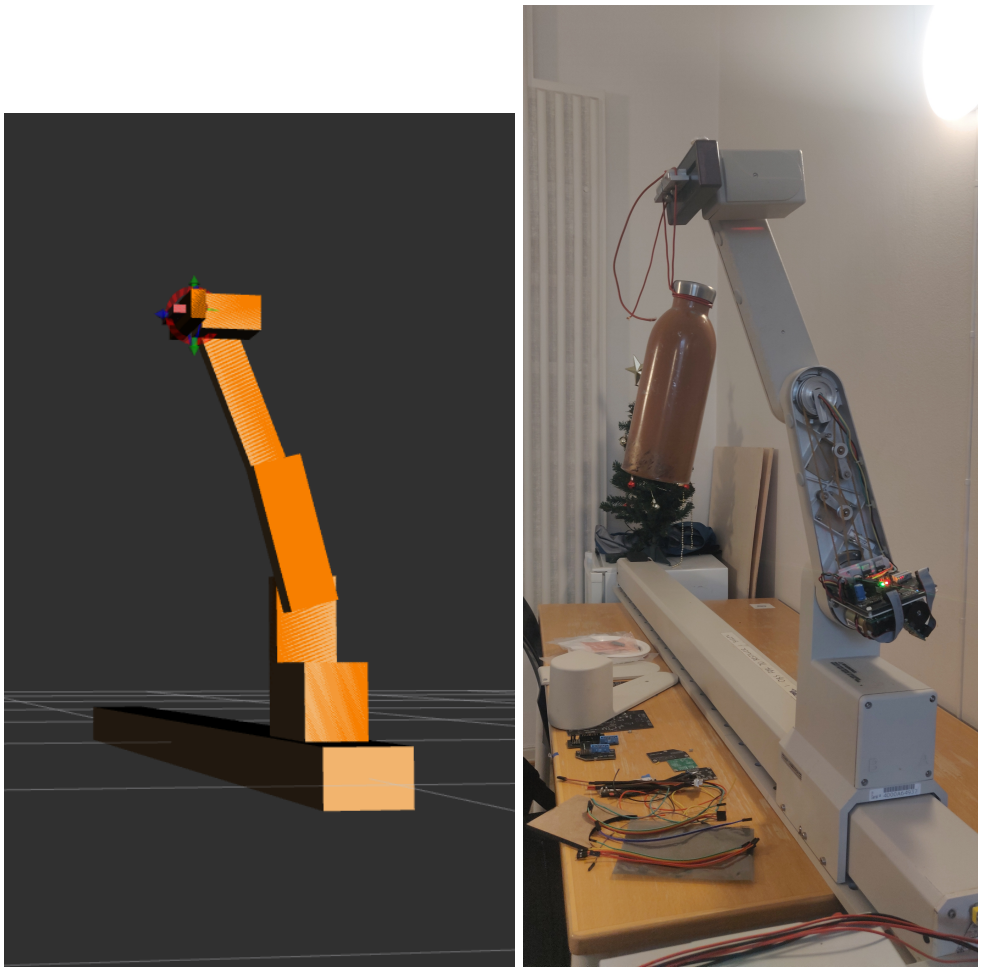
Actual end position after second movement test

Figure 41: The second movement test spawned two hypotheses: Mechanical coupling between joints, or erroneous measurment of joint encoder resolutions

The next test seeked to evaluate performance while carrying a payload. A bottle was filled with water for a total weight of 750g, and attached to the pincer stubs by wire. The bottle was then raised to the position indicated by figure 42, approximately 25cm above the table. The position appears to be accurate, with some "droop" visible in the wrist joint, likely due to load. No oscillation was observed during this test, but the bottle was lowered faster than it was raised despite movement rate implied by MoveIt setpoint updates was the same for either motion. These observations indicate that higher gain values may be beneficial compared to a no-load scenario, in particular an increase in Kd for the elbow to counteract gravity load.

After raising the bottle, the arm was moved to the end of the rail and the bottle was lowered to

the table. During travel, the twist joint was observed successfully counteracting swing movement induced by the hanging bottle, indicating appropriate gain values. The test was then repeated. During the second attempt at lowering the bottle, the shoulder joint slammed downwards immediately after the bottle made contact with the table. While this could be indicative of a lack of robustness to perturbation in the controller, it seems improbable as no signs of sudden or violent motion was observed in other joints at any point. As a result of these tests, elbow Kd was increased to 30 from 12.



Planned movement while carrying bottle Acheived position while carrying bottle

Figure 42: First bottle carry test

16.4 Sigmoid gain

Previous tests indicated that load may be significant to performance, in particular that finding Kpti values appropriate to avoid droop under load while also not inducing oscillation with no load would be difficult. This motivated introduction of the sigmoid gain: integral action would be low for high errors preventing oscillations induced by windup, and high for low errors preventing droop even under load. Three sigmoid gains were tested, illustrated in figure 43.

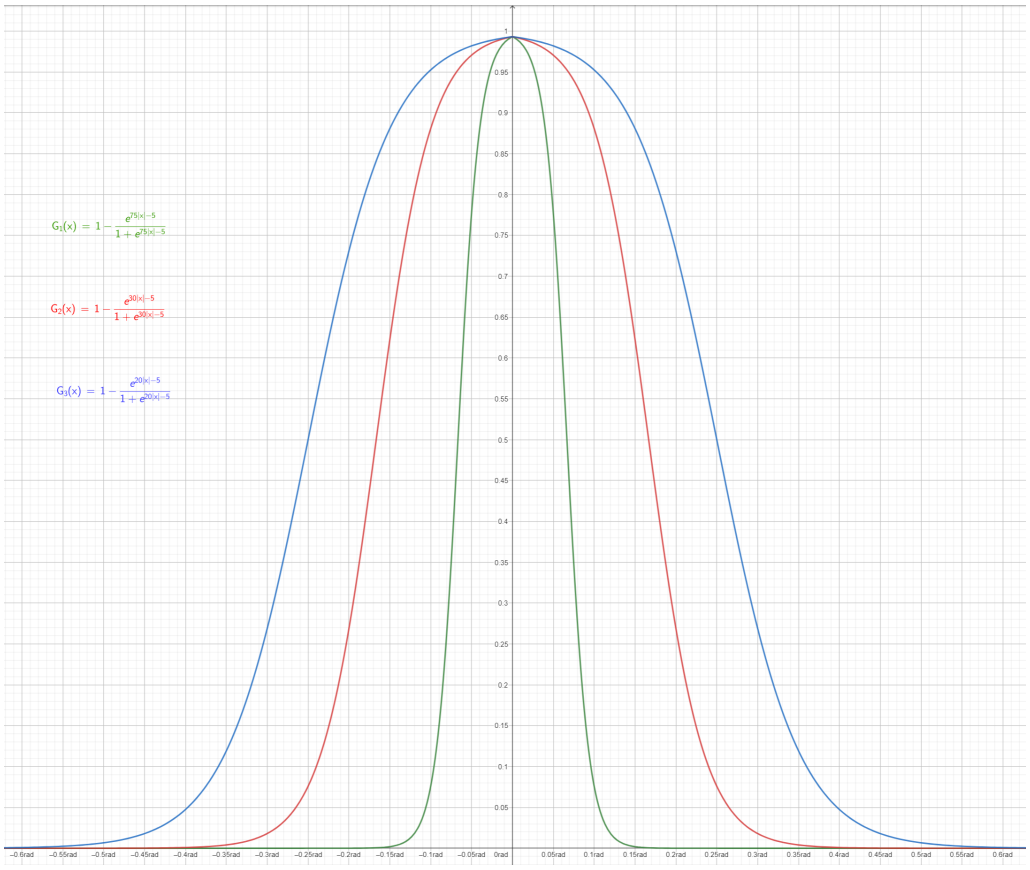


Figure 43: Three sigmoid gain shapes were tested, and the widest G_3 was accepted

16.4.0.1 G_1, shape argument of 75: For this shape, integral action becomes active ($G > 0.1$) at an error smaller than approximately 0.09rad. Shoulder, elbow and wrist were tested without payload, at deltas of 0.3 rad. Introducing the sigmoid gain appeared to remove overshoot for all joints, but accuracy was reduced compared to previous tests. This may indicate that integral action was insufficiently applied, that is, that the shape argument was too high.

16.4.0.2 G_2, shape argument of 30: Integral action active for errors less than 0.24rad. The test was repeated and resulted in improved accuracy with no overshoot for no load. However, testing under load yielded visible droop again indicating that integral action was insufficiently applied. The bottle was raised between the two positions illustrated in 44, and subfigure a) illustrates the droop that was present during the raise: the wrist was pulled down as the bottle left the ground and reached the lower position in subfigure b), and did not reach its goal position of fully horisontal by the end of the movement in subfigure a). The elbow was also observed not to reach its goal position, although accurate notes were not taken.

16.4.0.3 G_3, shape argument of 20: This reduction in shape argument yielded a horisontal hand position througout the raise movement, as seen in subfigure b) of figure 44. The wrist quickly corrected as load increased when the bottle left the ground, and stayed horisontal from start (b) to end (a) after the initial correction. G_3 was accepted as it appeared to improve overall system performance, lending credibility to the hypothesis that gradually activating integral action is beneficial to both stability and accuracy in this system.

a) Sigmoid shape G_2 yielded significant droop under load

b) Sigmoid shape G_3 yielded less droop, and was accepted

Figure 44: Sigmoid shape impact on droop while under load

16.5 IMU debug

As mentioned in section 16.2.1, the shoulder joint was observed to slam violently downwards under certain conditions. In an attempt at debugging this issue, as well as test rudimentary telemetry capabilities, a slam event was triggered and recorded as shown in figure 51. The graph shows measured shoulder position, shoulder setpoint and measured current in the moments leading up to the event. The event was triggered by dropping a bottle from the pincer fingers while the shoulder was at an angle of approximately 1.2 rad. Due to difficulty in converting non-integer values to character strings aboard the MCU, retrieved values are in milliradians ([mrad]) and milliampere ([mA]). The graph is presented and discussed in section 17.3.2.

16.6 Operational test: Pour

As mentioned in section 5.3, the primary use case of the system is beverage mixing at the venue of the project's commissioner and sponsor, Omega Verksted (OV). The arm and control computer were installed at OV for public demonstration and operational tests. As preparation, the IMU had been disabled and shoulder range was limited to 1.2rad for improved safety, and the gripper fingers first shown in figure 3 were produced. Tests presented here were all performed by manipulating the gripper in the MoveIt GUI. These tests may be understood as a "factory acceptance test", that is, a set of tests to evaluate the system's performance while operating in its intended setting.

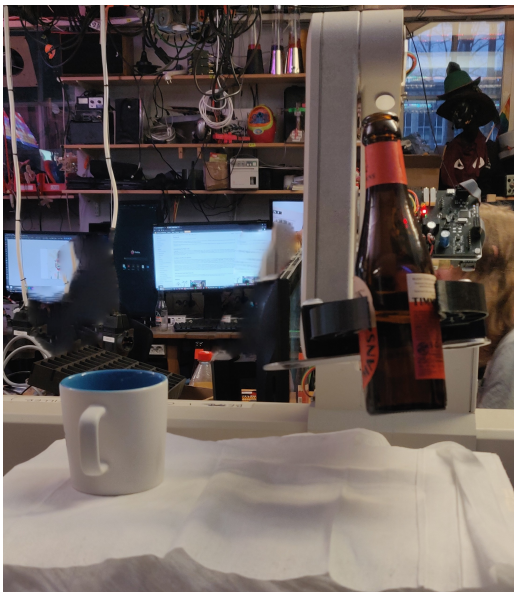
An initial operational test, the arm was manipulated to lift an empty coffee cup from a table, perform a "pour", and place the cup back on the table. A video of the test was uploaded to the youtube playlist TTK4900[6] titled *"Cup lift and simulated pour test"*. Throughout the test, the hand was planned to be horisontal. During the raise between the 10s and 25s marks, the wrist joint can be seen gradually correcting the hand towards a horisontal attitude, but an offset is clearly visible while the hand close to the table. This is likely due to the error mode discussed in section 16.3, where the elbow fails to reach its setpoint for high shoulder angles. The video also highlights

an issue with performing a controlled pour from the 17s mark. As the twist joint must be instructed to rotate the cup horisontally (17s) and then vertically (26s) in separate planning/execution operations, pouring small amounts of liquid precisely is difficult.

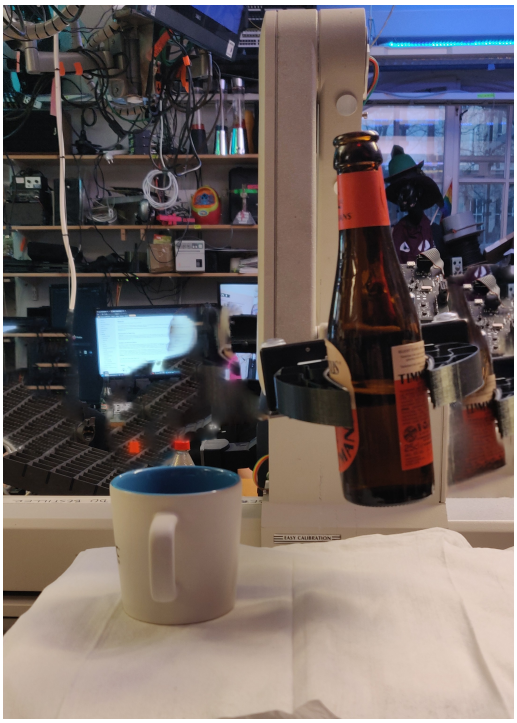
In another, not illustrated, test, a 500ml glass bottle was filled with water for a total mass of 0.9kg. The bottle was lifted, but the arm was found to struggle under this load. Despite attempts at closing the gripper around centre of gravity (CG) of the bottle, the twist joint was unable to turn the bottle horisontally. This is well below the load of a standard 750ml beverage bottle, and indicates that the arm is not compatible with standard mixing equipment.

Figure 45 illustrates the sequence of a successful pour. A 250ml glass bottle was filled with 1dl of water, for a total mass of approximately 0.5kg, and placed on an elevated surface next to a cup. In subfigure a), the hand is positioned such that the gripper is horisontal and close to bottle CG. This positioning indicated that it is possible to translate the gripper with an accuracy and precision of approximately 1cm, as multiple minor adjustments were needed to reach this position. In subfigure b), the bottles is raised approximately 10cm above the surface. While the gripper was instructed to remain horisontal, the bottle can be seen leaning inward in subfigure b) due to wrist angle not sufficiently decreasing during the lift. In subfigure c) the bottle has been moved close to the cup, and a movement of the twist joint finally pours the water into the cup in subfigure d).

a) Hand is positioned such that grippers are close to bottle CG



b) Hand is raised. Note that the bottle leans inwards



c) Inward lean is compensated by wrist action, bottle positioned for pour

d) Water is poured into cup by twist action, approx 25° past horisontal

Figure 45: Sequence of moves to pour a bottle filled with water into a cup

The test recorded in the video *Syrup pour test* evaluates the system's operability by non-expert personnel (5.3), and demonstrates a controlled pour of a spirit into a standard 4cl shot glass using the same bottle as in the previous test. A colleague at OV was given an introduction to the MoveIt GUI and left to familiarise himself with the system over the course of approximately 10 minutes. He then repeated the movement pattern described in figure 45 without further instruction. While some spillage can be seen at the 2s mark as the beam of liquid marginally overshoots the glass, this test demonstrates that the system is useable by a non-expert.

16.7 Main loop time

As discussed in section 13.1.5, accurate PID control requires regular updates of integral and derivative terms. A timer interrupt was set to raise a flag at a frequency of 10kHz, and the main loop was programmed to enter the control loop if the flag was raised. The frequency was arbitrarily chosen, on the assumption that the microcontroller operating at 72MHz would complete the main loop at a frequency "much higher" than 10kHz, and that 10kHz is sufficient for stable control of the joints. In order to test this assumption, a timer was set to count MCU cycles between the start and end of one iteration of the main loop. In order to avoid data contamination by the sending of telemetry, this section of the main loop was excluded from the cycle count. This test was performed after operational tests, and its results were not incorporated into tuning or otherwise accounted for in previous evaluations.

Main loop cycles were counted under three separate conditions: In the `GS_IDLE` state with no input from MoveIt, to provide a baseline reading; in the `GS_IDLE` state with input from MoveIt to measure impact of the handling of setpoint updates and; in the `GS_OPERATING` state with input from MoveIt to measure the impact of control loop execution. In the first test, CAN bus activity is limited to accelerometer data/requests between the torso and shoulder control units. In the second and third tests, CAN bus activity also includes distribution of received setpoints. Thus, the difference between the first and second test acts as an estimate of CAN bus and UART handling impact, while the difference between the second and third test estimates the impact of the control loop. All tests were performed on the torso control unit. Results are presented in table 22, and indicate that the controller is updated at a frequency of 5.6kHz. See section 17.2 for discussion.

16.8 Shoulder movement tests

Telemetry data was gathered from the shoulder joint, with the elbow joint in three configurations, in an effort to investigate to what extent movement characteristics are dependent on pose. It has previously been hypothesised that performance of the elbow and shoulder are mutually dependent, such as by moment of inertia experienced by the shoulder joint being dependent on elbow joint position. Figure 46 shows telemetry from the shoulder while it moves from 0 (vertical) to 0.6 rad while the elbow joint is at 0; figure 47 from 0 to 0.8 rad while elbow is at -1.57 rad; figure 48 from 0 to 0.8 rad while the elbow is also moving from 0 to -1.57 rad and; figure 49 shows a comparison of the error development in all three tests.

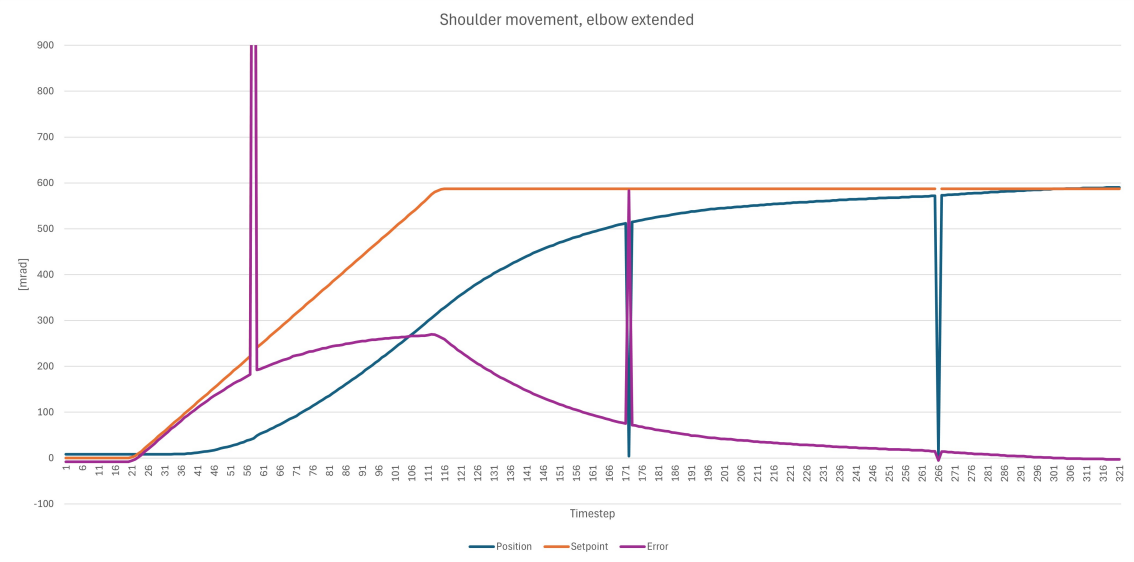


Figure 46: Shoulder movement with the elbow joint fully extended/vertical. Setpoint change rate: 6mrad/ts

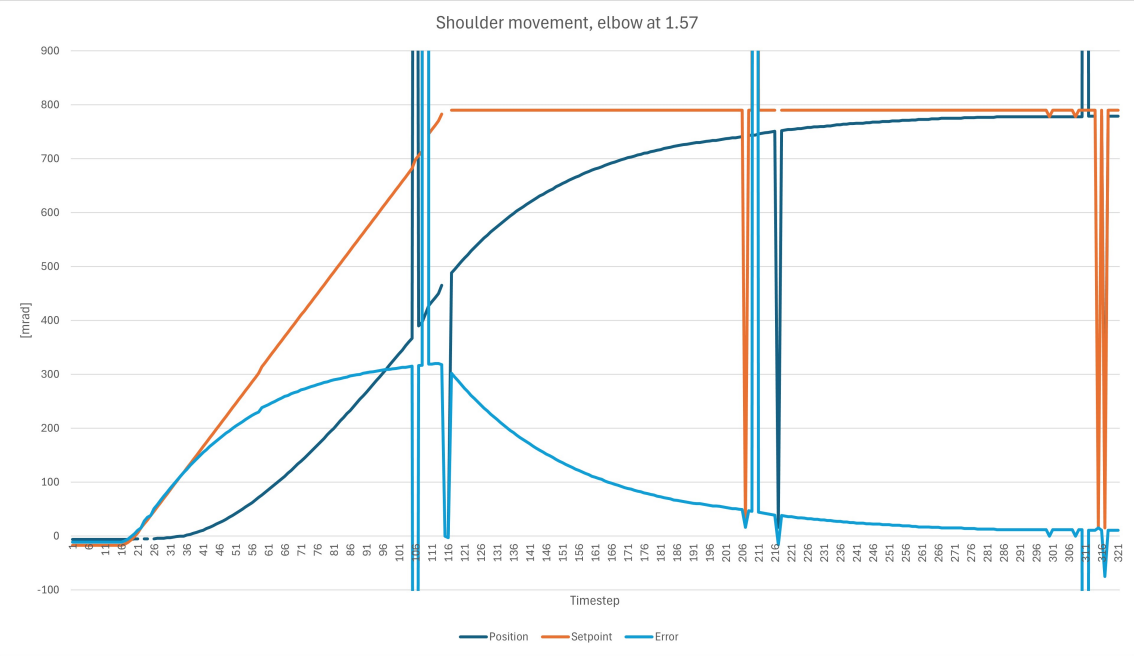


Figure 47: Shoulder movement with the elbow joint at 90 degrees. Setpoint change rate: 8mrad/ts

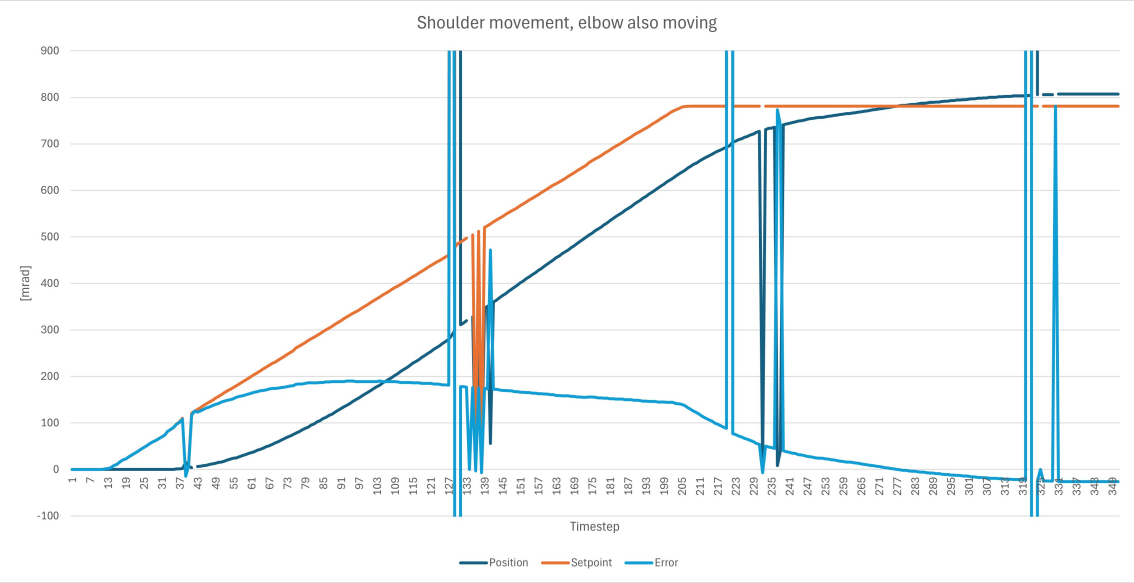


Figure 48: Shoulder movement with the elbow joint moving. Setpoint change rate: 4mrad/ts

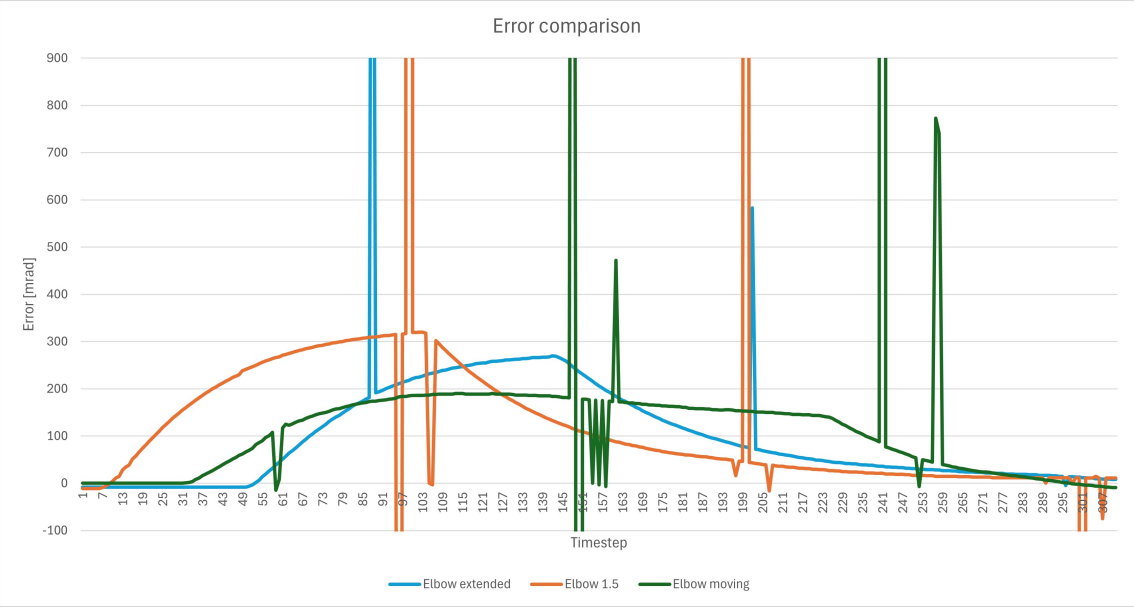


Figure 49: Comparison of error developement from previous tests

17 Results

This section presents project results, categorised by the three project phases of hardware development, software development and operations.

17.1 Hardware

17.1.1 Production

The following list summarises verification of produced hardware. See section 9.9 for a detailed discussion.

- 6 PCBs were produced on the basis of the specialisation project, named Rail, Bogie, Torso, Shoulder, IMU board and Hand.
- Of the 6 PCBs, 3 are control units: Torso, Shoulder and Hand. Their design is centered around the STM32F303 MCU.
- Interfaces with all 6 pre-existing encoders are functional; 2 per control unit.
- Interfaces with all 6 pre-existing motors are functional; 2 per control unit.
- Interface with pre-existing rail joint end stop sensor is functional.
- Interface with pre-existing wrist joint position sensor is **not** functional.
- Interface with pre-existing DSUB15 connector is functional.
- DSUB15 connection supports input voltage, UART, USB signals.
- 5V and 3.3V IC/sensor power supply circuits are functional; 1 set per control unit.
- UART serial interface is operational for all control units.
- UART serial interface may be accessed via DSUB15 connector; Torso.
- USB serial interface is **not** functional; Torso and Shoulder.
- CAN bus is functional for system supply voltages **up to 25V**.
- IMU unit embedded in Shoulder is **partially** functional.
- IMU unit embedded in IMU board is **not** functional.
- IMU unit embedded in Hand is **not** functional.
- Off-board I2C interface of Hand is functional.
- Motor power relays are operational; Torso and Shoulder.
- Twist joint position sensor was produced and is functional; Hand.
- Metal fastening bolts were replaced with nylon fastening bolts for all PCBs.

17.1.2 Other findings

The following list summarises findings pertaining to hardware made at various stages of the project.

- Voltage regulators produce significant heat when system input voltage is above 25V.
- CAN bus intermittently drops out when system supply voltage is above 25V, even though CAN transcievers are powered from 5V voltage regulators.
- Motor controllers produce an audible and uncomfortable noise when PWM frequency is in audible range, amplitude proportional to motor power setting.
- IMUs are largely non-functional, and precise cause could not be established. Several explanations were suggested, but lack of I2C debug options made investigation difficult. In the case of the "IMU board" unit, most likely explanation seems to be floating interrupt output pins.
- Signs of inappropriate/insufficient grounding in the system as a whole were observed. Minor discharges were frequently experienced when touching any part of the arm when powered/shortly after disabling power.
- A precise reason for USB failure could not be established, and investigation was not prioritised in favor of IMU/I2C investigation. Misconfiguration of the hardware driver, specifically VBUS detection interrupt, seems likely.

17.2 Software

17.2.1 Developed modules

Software modules were produced on four abstraction levels: Peripheral configuration, hardware drivers, control system and kinematics.

17.2.1.1 Peripheral configuration: The following list summarises software modules generated in CubeMX, responsible for configuration of the MCU's hardware peripherals and making available the relevant sections of the HAL library.

- `adc`: ADC1 and ADC2.
- `can`: CAN bus, including message ID filter banks.
- `gpio`: IMU interrupt input, CAN enable, relay enable, USB vbus detect and DP driver, twist and end stop interrupt input.
- `i2c`: I2C1 and I2C3.
- `stm32f3xx_it`: Declaration of interrupt handlers for ADCs, CAN, GPIO, UART and TIM.
- `tim`: TIM1, TIM2, TIM3, TIM4, TIM7, TIM8 and TIM15.
- `usart`: UART5

17.2.1.2 Hardware drivers: The following list summarises software modules responsible for interacting with the HAL library, developed in accordance with requirements described in section 7.3.

- `accelerometer_driver`: Controls the I2C peripheral and provide abstrations relevant to communication with/usage of the LSM6DSM IMU.
- `adc_driver`: Receives readings from the ADC peripheral, calculate current.

- `can_driver`: Controls the CAN peripheral, define framework for CAN communication.
- `gpio_driver`: Defines handlers for switch interrupts.
- `motor_driver`: Controls motor driver and encoder TIM peripherals, define interface for motor control.
- `ros_uart_parser`: Handles UART input when ROS communication mode is active.
- `string_cmd_parser`: Handles human readable string command input, defines set of string commands.
- `uart_driver`: Controls the UART peripheral, handle input via parsers.
- `unit_config`: List of global system configuration variables. Crucially, which control unit to compile for.

17.2.1.3 Control system The following list summarises software modules written for higher level control functions, developed in accordance with requirements described in section 7.3.

- `joint_controller`: Implements PID controller via motor and accelerometer drivers.
- `state_machine`: Defines a set of global states for calibration of individual joints.
- `main`: Main function, collects all previously listed functionality.

17.2.1.4 Kinematics The following list summarises the set of ROS nodes written and/or configured to enable arm control via GUI, in accordance with the primary goal of the project.

- MoveIt: Kinematics solver, movement planning by GUI. Control of all joints except pinch.
- URDF: Description of physical paramters of the arm, used by MoveIt.
- `control_listener`: Collects and parses output from MoveIt.
- `hmi`: Enables use of the string command interface while ROS is active, in particular pinch control.
- `serial_comms`: Collates information from other nodes, sends to arm via UART.

17.2.2 Evaluation with regard to system specification

The following list compares developed software to relevant system specification items presented in section 7.3. Acceptance levels indicate to what extent specification requirements were adhered to. See section 18.2 for elaboration.

- **AR1.1 interface definition:** High acceptance. All relevant modules define a set of `interface` functions through which they ought to be used.
- **AR1.2 scope limitation:** Low acceptance. Severe violation in `joint_controller`, moderate violation in `can_driver`. Other modules have precise scope, but often violate SRP(6.9.1) to some extent.
- **AR1.3 information passing:** Medium acceptance. While usage of non-interface functions outside of modules is possible, it will result in compilation and/or runtime errors in most cases.
- **AR1.4 naming conventions:** High acceptance. The entity naming schemes presented in section 11.1.1 was strictly adhered to.

- **AR1 clear module definitions:** Medium acceptance, see previous points.
- **DR1.1 unique documentation:** High acceptance. While some non-interface functions do not have unique docstrings, the vast majority of entities across the system are documented.
- **DR1 module documentation:** High acceptance, see previous point.
- **DR2 documentation availability:** High acceptance. Doxygen HTML and PDF build output is available from the project github[5].
- **DR3.1 documentation language:** High acceptance. Like entity naming conventions, documentation is formulated similarly across modules.
- **DR3 documentation conventions:** High acceptance, see previous point. Documentation conventions are not discussed in this report.

17.2.3 Other findings

17.2.3.1 Bus load estimates: Bus load estimates were calculated in an effort to estimate the relevance of the timers configured to limit bus load. Timers were configured based on estimates of bus usage before or during implementation of the relevant driver modules. Estimates were revised after implementation, and are presented in table 21.

UART: ROS sends 24 byte of position setpoints at 100Hz. When telemetry is active, such as during the tests described in section 16.8, the torso control unit outputs 14 byte telemety packets at 50Hz. In total, the bus load is approximately 22%. Conclusion: Telemetry may safely be sent more frequently, or packets may contain more data.

CAN bus: One CAN bus message frame (F) contains 135 bits (worst case, see section6.5.1); all messages are configured for a payload of 8 Byte. Setpoints received from ROS are distributed as they arrive and packaged in two CAN messages, consuming 27kbps. IMU data is requested at 344Hz, which, assuming requests and transmissions are symmetric consumes 92kpbs. In total, CAN bus load is 12%. Were all three IMUs functional, and data from all three rotational and accelerational axes used in pose estimation, load would be between 58% and 86% depending on how data were packaged. In alternative 1, one CAN frame contains raw IMU data (2 bytes) from all three axes of either rotation or acceleration. In alternative 2, one frame contains converted data (float, 4 bytes) from one axis of rotation and acceleration. Restructuring the system to forego request messages would roughly halve either estimate. Conclusion: the current solution could support all three accelerometers as well as improved telemetry with significant margin.

I2C: Y axis acceleration is read at 344Hz, that is, when a CAN message requesting data is handled. As multi-byte reads were not implemented, reading a 2 byte register results in 8 bytes of traffic. Consequently, the current implementation consumes 6% of I2C bus capacity. Were all axes of acceleration and rotation utilised, bus load would be 33%. Conclusion: accelerometer data polling rate could be tripled without changes to the driver.

Summary conclusion: Data buses are not at capacity in the current implementation, and additional desired functionality, such as improved telemetry and IMU data transmission, could be implemented with no or insignificant changes to existing drivers.

Bus	Item	Details	Usage	
UART				
	Capacity	Total	115 200	b/s
	ROS	6x4B @100Hz	19 200	b/s
	TEL	14B@50Hz	5 600	b/s
	Load	**Total**	**22**	%
CAN				
	Capacity	Total	1 000 000	b/s
	IMU	2F@344Hz	92 880	b/s
	Setpoints	2F@100Hz	27 000	b/s
	IMU alt 1	12F@344Hz	557 280	b/s
	IMU alt 2	18F@344Hz	835 920	b/s
	Load	alt 1	58	%
	Load	alt 2	86	%
	Load	**Total**	**12**	%
I2C				
	Capacity	Total	400 000	b/s
	Y Accel.	2x4B@344Hz	22 013	b/s
	X/Y/Z acc/rot	12x4B@344Hz	132 096	b/s
	Load	all axes	33	%
	Load	**Total**	**6**	%

Table 21: Summary of bus load estimates. "Total" loads are based on the system as is, other loads are hypothetical secenarios. For CAN: estimates are based on the largest possible frame size of 135 bits

17.2.3.2 Main loop estimates As mentioned in section 16.7, main loop execution time was estimated. Table 22 presents data from the test, collected over at least one minute, or at least 3000 data points, for each test state. Results indicate that the joint controller by a wide margin consumes most MCU core capacity, as the difference between test 1 (IDLE, NOP) and 2 (IDLE, RECV) is insignificant: handling incoming ROS setpoints decreased main loop execution frequency by approximately 50Hz, from 36.20kHz to 36.15kHz. Updating the joint controller results in an execution frequency of approximately 5.6kHz, with a coefficient of variance of approximately 2.3%. This is significantly below the target frequency of 10kHz, but it is regular.

State\results	Avg [μs]	STDEV	%RMS	Avg [kHz]
IDLE, NOP	27.62	2.7422	9.928	36.20
IDLE, RECV	27.66	2.7789	10.04	36.15
OPER, RECV	178.7	4.1365	2.315	5.596

Table 22: Test results from main loop timing

17.2.3.3 Module coupling: As a tool to investigate coupling between modules, the illustration in figure 50 was generated using a static analysis tool in Python[4]. The tool parses C code to find `#include` statements in a project, listing `.c` and `.h` files as shown, with modules assigned a large node if they have (relatively) many incoming or outgoing include statements.

`.c` files should only include the corresponding `.h` file. This is violated by `can_driver`, `gpio_driver`, and `string_cmd_parser`, for which the `.c` file includes the `uart_driver` module. Generally, more complex modules have more includes as they need access to more parts of the system than less complex modules, such as `state_machine`, `can_driver` and `joint_controller`. The opposite may also be true; `uart_driver` has a high score (large node) because it is itself included in most modules in order to enable error message output.

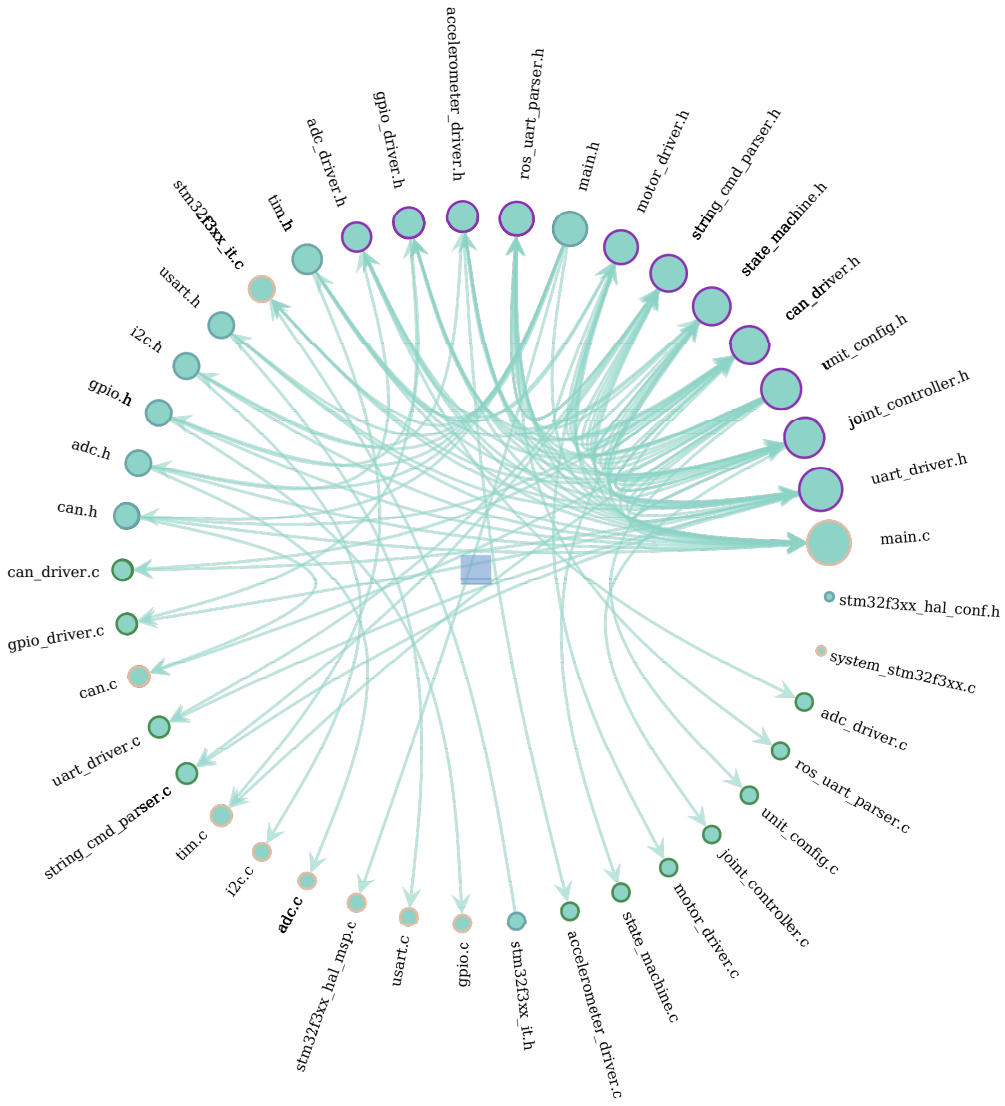


Figure 50: A graph of module inclusions to illustrate coupling. Arrows imply that a module includes another, node sizes indicate number of incoming/outgoing include statements, node outline color indicate files belonging to the same folder

17.2.3.4 State machine/calibration: The calibration phase, governed by the state machine, may only run successfully once after powerup. If triggered a second time after completion, the twist joint will be calibrated to an arbitrary zero position. No cause has been found for this behaviour.

17.3 Operations

The primary goal of this project has been to develop a robotic software system [...], 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 (OV) [...]. – Section 5.3

Operational tests indicate that the system does achieve the primary goal, and is to some extent compatible with the primary use case. The desired level of automation was not achieved, this was primarily due to time constraints; ROS/MoveIt enables automated movement, and the arm onboard software system is generally compatible with ROS input.

17.3.1 Evaluation with regard to system specification

The following list compares developed software to relevant system specification items presented in section 7.3. Acceptance levels indicate to what extent specification requirements were adhered to. See section 18.3 for elaboration.

- **F1.1 joint control:** High acceptance. All joints may be controlled concurrently.
- **F1.2 positional setpoints:** High acceptance. The arm consistently responds to setpoints received from the external control computer.
- **F1 pincer accuracy:** High acceptance. While there is some discrepancy between planned and achieved pose, pincer position and rotation may be manually controlled with an precision of approximately 1cm/10° in any direction.
- **F2.1 motion jerk:** Medium acceptance. Provided that the shoulder IMU is disabled, the arm generally moves smoothly at the velocities defined in the URDF file when under control from MoveIt.
- **F2.2 motion predictability:** High acceptance. While MoveIt occasionally outputs operationally invalid paths, the arm will follow given instructions upon execution of planned movements.
- **F2 general predictability:** Medium acceptance, see previous points.
- **F3 safety parameters detection:** Low acceptance. While power to motors will be cut if detected current is above 4A, no other sensors have been configured to detect operationally unsafe states. 4A is a hardware limitation[55], and unsafe movement was observed at well under 500mA.
- **F4 user friendly interface:** Medium acceptance. While the MoveIt GUI was found easy to use by one non-expert, the entire user interface is spread across three modules: MoveIt, HMI and PuTTy (or similar serial interface).

17.3.2 Other findings

17.3.2.1 Shoulder movement and telemetry: Graphs presented in section 16.8 compare movement of the shoulder joint with the elbow in various configurations. Telemetry was collected via the UART interface, and spikes represent malformed datapoints. These appear to occur semi-regularly, which warrants investigation of the UART driver. Figure 46 and 47 indicate that movement characteristics are similar while the elbow joint is not moving; in both cases the shoulder joint reaches it setpoint approximately 200 timesteps, or 4 seconds[14], after the final setpoint change. Figure 48 shows the shoulder reaching its final setpoint after approximately 150 timesteps, or 3 seconds. Error developments are compared in figure 49, and indicate that the shoulder joint is generally closer to its setpoints while setpoints are changing in the case where the elbow is moving. This could be a consequence of setpoints changing at a lower rate for this case, that is, that MoveIt planned a lower shoulder joint velocity for this movement.

17.3.2.2 Shoulder slam test: This paragraph discusses figure 51, result of the test presented in section 16.5. The minor spike in current around timestep 231 (purple) likely represents the botte drop, where the controller rapidly attempts to correct for the sudden change in load. This jolt is experienced by the accelerometer, possibly recorded as a miscalculated position (red). The yellow dotted trend line indicates a slight increase in motor current compared to the moments leading up to the spike, but no apparent movement until power spikes around timestep 306 followed by position increasing to approximately 2.1 rad. Movement stops there as the upper arm collides with the torso. Large spikes in the data set are likely caused by bad telemetry packets, such as around

[14]Provided that telemetry is in fact updated at 50Hz

timesteps 11 and 31. Small, semiregular spikes are likely a result of IMU readings being used to calculate position rather than encoder data. Spikes at timestep 311 and 317 (not fully shown) indicate that motor current briefly exceeded 1A, followed by a jolt in position reading at timestep 320. This test did not lead to any conclusions about the cause of the slam, and other telemetry items such as power setting and excplicit IMU readings are probably necessary.

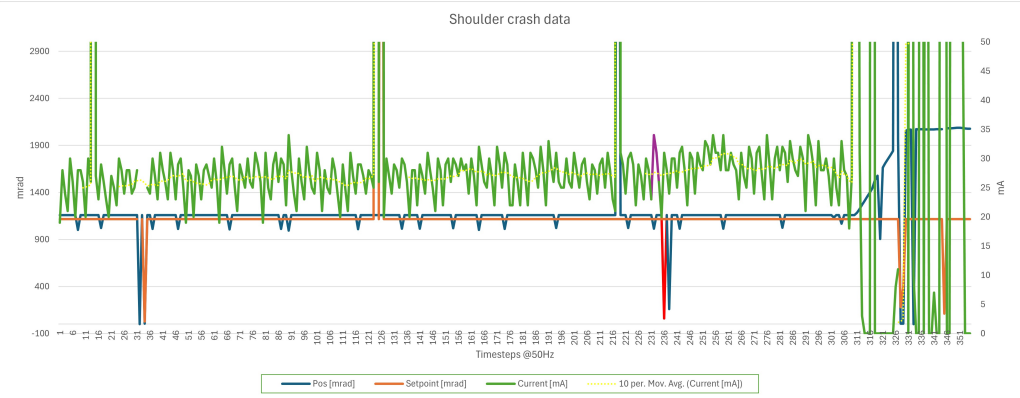


Figure 51: Telemetry readouts from the IMU debug test

17.3.2.3 Primary use case: Operational tests during a public demonstration at OV indicate that the system works according to the stated primary goal, but is not a functionally useful tool for beverage mixing. The system was operated by a non-expert after approximately 10 minutes of instruction, who succeeded in pouring a 4cl shot. In particular, it was found that relying on pure twist joint action when pouring frequently resulted in liquid overshooting the target cup/glass even for low angles. This demonstration was only possible after disabling the shoulder accelerometer, which removed the previously discovered failure mode of the shoulder joint reacting violently to perturbations.

18 Discussion

This section presents qualitative discussions about project results, and forms the basis for section 20.

18.1 Hardware

18.1.1 Indicator LEDs

During the hardware verification phase, motor control relays were used to check that MCU flashing was successful by listening for the relay contactor switch. While convenient, the test easily could have resulted in a false negative as the switch produces no audible sound when input voltage is above $\tilde{2}$0V and is in any case not valid for the hand control unit, which has no relay. Designing the control units with controllable LEDs on unused GPIO ports would have made debugging easier both during this initial phase by reducing reliance on UART output. For instance, while testing the CAN bus, LEDs could have been activated when sending/recieving a message. This would have made the testing process significantly faster, as moving the UART/USB adapter between control units to verify input/output in each instance was tedious. There are several unused ports on all MCUs, and LEDs are generally small and cheap enough that installing a number of them would have been feasible for all control units.

18.1.2 Hand optical sensor

The problem of the hand optical sensor initially producing a useless output before modification with an external voltage divider likely could have been avoided by careful study of the data sheet[10] as well as the original sensor circuit. In addition to producing an output voltage consistent with division (i.e. 2.7V for a 5V input), the circuit appears to have two resistors mounted close to the sensor. Plainly, the original sensor circuit seems to have been replicated by accident in the modified circuit, and replicating in on purpose would have saved some time.

18.1.3 IMU debug

The decision to place the upper arm IMU on the shoulder control unit seemed pertinent: it would reduce wiring inside the arm, reduce bus length between MCU and IMU, and remove the need for a PCB the sole purpose of which was to mount the IMU. In hindsight, this made it very difficult to debug with the available equipment and, perhaps more importantly, practically impossible to replace. Debugging would have been significantly easier if a debug header were added, similar to the motor driver debug header. Replacing the IMU or changing the IMU assembly circuit effectively requires reassembly of the entire shoulder control module, which is expensive and time consuming. Furthermore, the LSM6DSM IMU unit is small enough that manual assembly presents an error source. The combination of a 0.5mm pitch in an LGA package with 14 pads made it difficult to apply solder paste consistently as well as inspect the finished solder.

Furthermore, the complete circuit design should have been tested more thoroughly before commiting to production, particularly with respect to leaving the interrupt output pins floating. No reference to this error mode could be found in the data sheet[45], but it would likely have been discovered by breadboarding the design. The breadboard design which was tested was based on the specialisation project, where both interrupt pins were grounded. This error mode does not fully explain why IMUs on the shoulder and hand control units did not work, however. In these designs, both interrupts are connected to GPIO pins on the MCU, which appears to be valid. Conclusions about exact error modes are further occluded by the shoulder accelerometer being partially functional. The data sheet does not indicate that individual axes need to be activated for output, and breadboard tests with the LSM6DSM were successful without any such configuration.

It seems plausible that the shoulder IMU suffers from hardware faults, but unlikely that all IMUs are faulty.

18.1.4 Regulator heat

Voltage regulators were found to produce significant heat for input voltages above 20V. This is to be expected, considering that they are linear/dissipative[56], but at 30V the heat is intense enough to produce a smell of burnt plastic. Operational tests indicate that system voltage should be increased above 25V in order to support loads of more than 1kg, implying that the regulator circuit needs redesign. The LM317HV unit was chosen due to its high current capacity of 1.5A and simple voltage adjustment circuit. However, no indication of currents exceeding a few hundred milliampere per regulator were observed: When not moving, the arm was observed to pull approximately 1A in total via the power supply display. It is believed that most of this went to the relays, which act as a short circuit when powered. Selecting a switching (i.e. non-dissipative) voltage regulator with these parameters in mind should be possible, and would enable usage of a higher system input voltage.

18.1.5 Grounding

PCBs were isolated from the arm chassis using a combination of nylon fastening bolts, silicone pads and electrical (PVC) tape, emulating the original design without critical analysis. With the exception of the rail PCB, all PCBs are isolated. This may have been a bad design choice, as shocks were frequently experienced when touching the arm. The source of the shocks was not investigated, but probably should be as they represent a moderate hazard to equipment and a low hazard to personnel.

18.1.6 CAN bus voltage

During the PID tuning phase, it was discovered that the CAN bus would occasionally drop out if the system input voltage was above 25V. The exact failure mode could not be estblished, but the Torso MCU would output an error message indicating a failure to send the relevant CAN message. Whether the failure was in the MCU CAN peripheral or the CAN transciever is unknown. This failure mode is particularly strange, as system input voltage should only be seen by the voltage regulators. Along with the grounding issues discussed, this problem necessitates further investigation if higher system voltages are to be used.

18.1.7 USB and voltage clamping

Little time was spent trying to investigate the USB functionality. There was uncertainty about circuit design, peripheral configuration and driver module. Combined with the UART being operational and more than a week having been spent investigating the IMU/I2C problems, it did not seem worthwile. However, it was discovered that while section 2.3 of AN4879[44] recommends a separate TVS diode for VBUS detection, this line was connected to the same TVS array (USBLC6[51]) as the data lines such that VBUS acts as the voltage clamp of that array. Furthermore, USB data lines along with UART lines were protected by an array on the rail PCB with a constant 5V clamp. An LED was connected in series with the ground pin, reasoning that it would alight in the event of overvoltage in the array. However, the LED would always be lit while the arm was powered. What impact, if any, this had on the USB assembly or previously discussed grounding issues was not established and warrants further investigation.

18.1.8 UART5

UART5 was chosen for no particular reason beyond being avaliable after other peripherals had been configured, along a favorable edge of the MCU. This port has the fewest features of all U(S)ART ports, missing features like flow control, DMA transfer and automatic baud rate detection (RM0316 chapter 29.4[46]). Considering that UART bitrates above 115kpbs were not achieved, having had access to these features may have been useful. USART1, for instance, has all features and is available from unused pins.

18.2 Software

18.2.1 General considerations

18.2.1.1 Project basis: Development of software for this project builds on experiences made during several previous projects in both hobby and professional settings. For instance, the choice to rely on open source solutions as far as possible stems from experiences with licenced software as unintuitive and with limited options for support outside of specialised fora – in addition to licenses tending to expire or be locked to individual user accounts. The hope for this project is that it will be long lived and usable by people with varying degrees of experience with low or high level programming at Omega Verksted. In that context, it is important that the code base is as open as possible, and that it may be altered with commonly used tools. ST's IDE STM32CubeIDE, which for instance offers debugging tools not immediately available in VSCode, was specifically elected against on this basis.

18.2.1.2 Conventions: Similarly, the conventions discussed in section 11.1.1 were established on the basis that using standardised language makes the code base easier to understand. Some function names, for instance, are very long, but more descriptive than a shorter name might be: Parsing the name `controller_interface_update_controller` requires less effort than `ctl_itf_upd_ctl` when reading code, and the convention of "write words fully" is easer to use than "select stressed letters from each syllable of the original word to make a three letter abbreviation" when writing code. The actor/verb/noun convention is arguably heavyset, but provides predictability across a complex project.

18.2.1.3 CubeMX: In a departure from the "open source" directive, CubeMX was used for peripheral and pinout configuration, and generally found to be immensely helpful in that regard. Combing AN4206[43] for clues to which registers to configure for every peripheral would have severely hampered project progress, and it is doubtful that ROS implementation would have been feasible at all.

18.2.1.4 Debugging: While the VSCode extension *STM32 for VSCode* claims to support debugging[8] via UART, this was not successfully configured. It was not prioritised on the basis that debugging of a real-time system is best handled by print statements. In hindsight, prioritising setup of the debugger may have saved some time, in particular during configuration of the CAN bus. There were signs that the RX interrupt handler either did not set a flag in the correct register after handling, or would set it twice. The behaviour was eventually attributed to the RX interrupt handler being long enough for it to be preempted by another interrupt handler, but having access to direct register readouts and breakpoints would have been helpful in that determination.[15]

18.2.1.5 Bare metal vs OS: This project was solved "bare metal", but the STM32F303 is compatible with lightweight real time operating systems such as FreeRTOS. Considering that the main loop runs several processes which effectively function as threads (CAN executives, for

[15] Plainly: This project was, regretfully, solved without the use of a debugger

instance) and that CubeMX enables configuration of FreeRTOS, it would have been interesting to implement this. It was briefly considered, but elected against so as to reduce project scope. As FreeRTOS may run ROS nodes, enabling it could allow for tighter coupling between the arm and MoveIt/other ROS nodes in the future.

18.2.2 Adherance to requirements, SOLID

The architectural requirements presented in section 7.3, and the SOLID principles presented in section 6.9 acted as guidelines when implementing software modules. While the SOLID principles were originally intended for object oriented programming languages, they arguably have general applicability to architecture design. In particular, the single responsibility and interface segregation principles generally state that modules and interfaces should be narrow. One interpretation of "narrow" is that a module concerns no more or less than what is implied by its name. The motivation for this view is to ensure maintainability of the code base as it expands, possibly beyond the creators' original intention.

In the context of an embedded system, where software is tied to the hardware it controls, adhering strictly to these principles may not be necessary or even useful. The ADC driver, for instance, is really rather a motor current calculator. However, the function of current calculation is innately tied to ADC usage, and, importantly, ADCs are not likely to be used for anything other than current measurements in this system as there are no other analog measurements available to the MCU in the current hardware configuration.

In other cases, such as for the motor driver, it may have been useful to separate motor driver functionality (that is, control of the DRV8251 units) from functionality pertaining to encoder reading. When this was not done, it was because there is a one-to-one relationship between encoders and motors which is not subject to change, and motors are, it was reasoned, innately coupled to their encoders by the "resolution" trait. The motor drivers were written before the joint controllers were conceptualised, and in hindsight it may have been more intuitive to store the resolution trait in that module, in turn making a separation of encoder functionality a natural next step.

Finally, a more banal reason for not adhering more strictly to narrowness is that it tends to result in a high number of modules. This has an overhead with regards to compiler configuration, documentation generation et cetera, which takes some time and mental overhead to manage. Neither were abundant resources in this project, which undoubtedly resulted in some shortcuts being taken.

One C convention which was omitted from the project was that of declaring and defining private functions in the source file, preferably as `static`, as opposed to declaring them in the header file. As discussed, the keyword `interface` instead serves as an indicator that a function is public. It was thought that collecting all functions in the header file makes information flow and documentation easier to follow, with the added benefit that dependent functions do not need to be defined in the order of depenency. While this is arguably true, this structure also opens up the possibility of accessing private functions outside of a module simply because the code editor suggests functions alphabetically by default (`driver` < `interface`). The programmer then needs to remember the full name of most `interface` functions rather than relying on suggestions, increasing mental overhead while programming. This system should be overhauled with `static` private functions, and is the reason for the medium score on **AR1.3**.

Adherance to architectural requirements was given an overall medium score as indicated by the evaluation in section 17.2, for which poor scope limitation was the main reason. The truly severe violations of the `joint_controller` and `can_driver` modules are discussed in dedicated subsections.

18.2.3 Joint controller

The `joint_controller` module was conseptualised as "the thing which controls joints", and should thus have access to "all information relevant to the control of joints". This includes encoder and accelerometer data, which it would use to estimate a joint's position. Encoder data is stored in the `motor_driver` module, which is local to the controller itself – that is, it exists within the same MCU. Accelerometer data, however, is accessed from another MCU via the CAN bus. This is true for all joints for which accelerometer data was originally supposed to be available for, assuming data from only one accelerometer would be used for each joint: the torso control unit gets its shoulder joint data from the shoulder control unit, and the shoulder control unit gets its elbow and wrist data from the hand control unit (see figure 5).

This necessitates a structure which holds and processes incoming/non-local accelerometer data, separate from the `accelerometer_driver` module which handles communication with local accelerometers. As non-local accelerometer data would be used exclusively by the joint controller, this structure was implemented as a struct and set of functions in the `joint_controller` module. The struct has separate fields for acceleration and rotation data in three axes, as well as flags to indicate whether or not each data field is new (i.e. not stale). The set of functions provide get/set/clear for each of the fields, resulting in a very high number of functions. Additionally, the `joint_controller` module was made responsible for storing, and providing an interface to, the timer flags which moderate controller update and accelerometer polling frequency. This effectively makes the `joint_controller` three modules in one: accelerometer data handler, timer watchdog and joint controller. The module should *have access to* "all information relevant to the control of joints", but not *be responsible* for it.

In addition to being in the wrong module, all accelerometer data would not need separate fields in the struct. Instead, a number of arrays, for instance matching the associated CAN message formats, would suffice, reducing the number of associated functions. A module called `accelerometer_controller` could then be responsible for storing and processing raw values to angles/angular rotation and make data available to the joint controller through its interface.

Timer handlers could have been in a `schedule_controller` module, with the status of polling flags available through its interface. Such a module would also emphasise the motivation behind the timers, namely to act as a rudimentary scheduler in lieau of an operating system to handle this functionality.

The `joint_controller` module is the main reason for the low evaluation of **AR1.2**.

18.2.4 CAN driver

The `can_driver` module suffers from a similar lack of specialisation as the `joint_controller`, but not to the same extent. For this module it may have been pertinent to move the CAN message format specification, i.e. `can_message_type` enumeration and handler functions, to a separate module. The reasoning for the current implementation was similar as for the joint controller; the module should be responsible for everything relating to CAN communication. Limiting it to hardware interactions and message scheduling may have improved readability, however. The structure could have been similar to the `uart_driver` and `string_cmd_parser` modules, where the former is responsible for hardware interactions and the latter is responsible for parsing.

The structure of the CAN message ID (fig 24) is also flawed. The reasoning when establishing the protocol was to treat each hardware unit uniquely rather than as a subsidiary of the relevant control unit. Plainly, that one would access "accelerometer number 1" rather than "hand control unit, accelerometer number 0", for instance. This wastes one bit in the ID field, which could otherwise have been used for message type identification. The reasoning also breaks with a general expectation in bus addressing schemes, where units are accessed with increasing precision for lower bit positions. A more intuitive approach would have been to reserve two bits for control unit selection, one bit for accelerometer/motor selection, and one bit for unit number. Considering that two bits could "fit" four control units, this scheme could save an additional bit by foregoing

"global" messages as a separate category indicated by the most significant ID bit and instead letting one "control unit address" signify a global message. This would require an overhaul of CAN filter bank configurations and bit selection logic in the message handler functions, but would allow for more message types.

18.2.5 State machine

Rather than being a cornerstone in architectural design, the state machine was only introduced after it became apparent that it would be needed for calibration. Functionality for zeroeing joint positions is effectively hard coded to the calibration procedures, and the calibration procedure producing different results if run a second time after powerup indicates that some information is kept between state transitions even though the intention was for the state machine to be memoryless. While the exact cause could not be established, it is believed to be a consequence of tight coupling between the `joint_controller` and `motor_driver` modules. If lower level functionality were implemented with the state machine in mind, this may have been avoided.

18.2.6 Peripheral interconnect, DMA

The STM32F3 MCU family has a peripheral interconnect matrix (RM0316,cp8) which supports direct transfer of information between peripherals as well as memory (DMA) (RM0316,cp13) rather than via the MCU core. These hardware functions were not made use of, as they seemed to have low benefit compared to implementation overhead. In light of the main loop taking well over twice as long to complete as assumed (5.6kHz vs >10kHz), DMA/interconnect usage should be reconsidered.

18.2.7 Expandability

As mentioned in section 5.3, the secondary goal of the project is for software to be expandable to other use cases requiring higher movement precision/accuracy than beverage mixing, and incorporating more advanced sensors such as robotic vision. While the arm onboard software clearly has room for improvement, ROS/MoveIt shows great promise with regards to expansion. Adding nodes to the network was found to be as simple as adding one line of code to determine a topic for subscription/publishing, and MoveIt supports advanced movement planning through the concept of Tasks[9].

18.2.8 Error handling

Very little effort was put into error handling. HAL library functions will output `HAL_ERROR` if they fail under certain circumstances, which in some cases such as for bus messages was opted to propagate as UART messages for debugging. There is no error handling in the system beyond that, and generally no checking for safe or even physically meaningful sensor values. This is clearly detrimental to system stability, as was seen in the case of the shoulder slamming downward.

18.3 Operations

18.3.1 General considerations

Despite accelerometers being entirely disabled for most of the operational tests, the arm was found to be precise enough to fulfill its purpose with regards to the primary use case. That the arm is not strong enough to lift a standard 70 cl bottle massing approximately 1.5 kg is unfortunate, but may likely be solved by increasing system voltage. Characteristics of the twist joint motor[12] should be investigated further to determine whether it is strong enough to tilt a standard bottle

at various fill levels. If it is dependent on always gripping the bottle precisely at CG for a given bottle fill level, creating standardised movement tasks for lifting and pouring would be difficult.

During the pour tests, it was generally found very difficult to tilt the bottle such that liquid did not overshoot the cup/glass. This could likely be amended to some extent by reducing the twist joint movement rate limit, which is currently 3rad/s. However, the primary issue was found in the pour movement itself. A human pouring from a bottle will typically center the bottle tilt around the bottle mouth rather than CG, with the bottle mouth placed close to the glass throughout the movement. Emulating this pattern in the arm is difficult, as MoveIt has not been configured to center a movement around an object which it may be carrying but rather around the gripper itself. Pour tests also revealed that limiting the shoulder joint to an extension of 1.2 rad is impractical as the arm struggles to reach objects at table level – hence the need for an elevated surface as seen in figure 45.

As mentioned in section 16.3, a video was uploaded to a YouTube playlist TTK4900[6] under the name *Simple movement test with MoveIt.* The video shows the arm moving from the calibrated pose to an arbitrary "crouched" pose along the rail. While movement appears smooth, the path planned by MoveIt is clearly not the most efficient or least complex possible path to the end pose. Moreover, this path would likely not be valid for an operational scenario where the arm is carrying an object. MoveIt would occasionally plan paths like these throughout the testing phase. The cause could not be found, but may relate to a lack of constraints on movement in the MoveIt configuration. This mechanism was minimally detrimental to further testing, as an invalid movement could simply be discarded and the movement replanned. Towards the end of the video, the rail joint can be seen oscillating around its setpoint, likely as a consequence of high static friction.

18.3.2 Causes for shoulder slam; accelerometer behaviour

The precise reason for violent shoulder joint behaviour while the accelerometer was active could not be established, but some speculation is due. First, I2C bus failure may likely be discarded. While there were considerable issues with accelerometer data, telemetry tests performed on the shoulder control unit in the verification phase (not previously discussed) indicated that Y axis readings were consistent and correct. It seems more likely that a lack of filtering and/or combination with encoder data, as well as a long path from data request to reception could explain the behaviour.

18.3.2.1 Data path: Torso queues a CAN message requesting accelerometer data, which, considering that CAN executors run immediately after, is probably sent within $25\mu s$ based on results in table 22. Depending on where the shoulder control unit is in its main loop when receiving the request, it may take $200\mu s$ before the request is handled, and $160\mu s$ to read accelerometer data based on I2C bitrate. While data would be sent back to the torso within few microseconds, the torso may in turn not use it for another $200\mu s$ depending on its position in the main loop upon reception. In total, accelerometer data may be approximately 0.6ms out of date by the time it is used.

18.3.2.2 Data filtering: In addition to time delay causing inaccurate positional estimates, acceleration data may only be directly translated to joint angle while the joint is not moving. During movement, some combination of filtering (i.e. averaging to compensate for outliers or sudden jolts) and blending with rotation and/or encoder data would be necessary to purposefully use IMU (accelerometer) data in pose estimation. Studying figure 51, minor spikes can be seen along the position measurement data series. Recalling that telemetry is refreshed at 50Hz, and that accelerometer data is updated at 344Hz, these spikes likely represent telemetry readouts coinciding with accelerometer updates. The spikes are not present in the shoulder movement test results (such as fig 47), for which the IMU was disabled. The spikes indicate an offset between joint position estimates based on accelerometer versus encoder data, which could be explained by data path delay, resolution estimate error, and/or accelerometer offset error. Each spike would induce a step response in the controller, which during steady state operation clearly is not enough to destabilise

the system. However, when a perturbation such as the drop of a bottle causes an acceleration measurement which would indicate a large position error, the controller is destabilised. In the case of figure 51, the error is momentarily increased to well over 1 rad at timestep 236. Stress tests (not previously discussed) indicated that the controller would struggle to maintain stability for setpoint steps greater than 0.7 rad, so that a sudden "step" of more than one 1 rad could destabilise the controller does not seem implausible even though subsequent encoder readings should help restabilise it.

This is speculation, and will require more sophisticated telemetry data to confirm. IMU data filtering capabilities further motivate the introduction of an `accelerometer_controller` module.

18.3.2.3 Single axis measurement: Section 13.1.4 mentions that only the accelerometer Y axis is used for position estimation (see figure 14), with the result that the estimate is guaranteed to be wrong if the joint passes the horisontal position in either direction. For a valid estimate of all angles physically obtainable by the shoulder joint, both Y and Z axes are necessary. Unfortunately, no data could be extracted from the Z axis.

18.3.3 Main loop timing

Table 22 indicates that the main loop runs at approximately 5.6kHz, indicating that the joint controller is updated at roughly half of the target frequency of 10kHz. This result was unexpected, but is not necessarily problematic. The motivation behind the timer was ensuring that the controller was updated regularly. So long as the frequency is high enough to provide stability, regularity is more important than frequency is it ensures that integration and derivative intervals are constant. Table 22 also presents the coefficient of variation (%RMS), which was calculated to be approximately 2.3%, equating to a difference of approximately $8\mu s$ between the longest and shortest execution time. Empirically, this is acceptable: A mistake made early in the development phase had the shoulder joint controller update at approximately 50Hz, which resulted in stable operation for a system voltage of approximately 20V. Nevertheless, the update frequency should probably be lowered such that the main loop frequency is at least twice as high[17]. Alternatively, the controller function could be investigated for possible optimisations.

Knowing the update frequency may simplify analytical assessment of the implemented controller, as it gives meaning to the $\frac{1}{T_i}$ and $\frac{dE}{dt}$ parameters of the PID equation. The main reason for presenting $\frac{K_p}{T_i}$ as a single constant, `Kpti`, was the lack of meaningful physical interpretation of $\frac{1}{T_i}$ at the time of implementation, when main loop execution time was not yet known.

18.3.4 Bad telemetry data

Graphs presented in section 16.8 include spikes of malformed data received from the arm. A cursory investigation of the full data sets (see GitHub[5]) indicate that these spikes appear regularly. If that the case, it could be indicative of a bug in the `uart_driver` module.

18.3.5 PID implementation

The implementation of the PID controller presented in section 13.1.5.3 has a negative sign on the derivative term. This is unintuitive, as the power output for one iteration of the derivative term would increase rather than decrease if the error has decreased: The derivative term is defined as $D = K_d(E_t - E_{t-1})$, which when subtracted from the power setting should result in an unstable controller for all cases except while a joint is overshooting its setpoint (i.e. increasing error). However, the opposite was found. The controller was consistently unstable when the derivative term was added rather than subtracted. Code was inspected for sign errors which might explain this, but none were found. This could be an indication of entirely mistuned/misimplemented PID controller, which just happens to be stable for the gain values found.

One error was found in the PID implementation which supports the hypothesis of an incidentally stable PID controller: While inspecting for sign errors in the D term calculation, it was found that the implmentation of the sigmoid gain effectively squares the `Kpti` gain (see `joint_controller_update_power`in the repository[5]), implying that integral action is very small. This, in combination with the strange derivative term, casts doubt on the validity of the PID controller as a whole.

18.3.6 Joint accuracy

Section 16.3 suggested that mechanical coupling between joints may explain the poor accuracy of the elbow and wrist joints near extrema, as illustrated in figures 41 and 45. An inspection of the arm revealed no reason to believe there is coupling between the shoulder and other joints, but the wrist joint is indeed coupled with the elbow. When the elbow is moved from the vertical position to its negative extreme, the wrist moves approximately 10° in the positive direction. This is opposite of the inaccuracy illustrated in figure 45, so it cannot be the sole explanation – and does not explain the elbow inaccuracy at all.

The next plausible culprit is the encoder timer. If the timer misses encoder clicks, thereby failing to update the encoder counter register, measured joint position would drift over time. However, the timer is asynchronous, register updates triggered by the encoder itself. Assuming its highest possible "sampling rate" is at least equal to the system clock of 72 MHz, this is not likely to be an error source. The wrist[26] and elbow[27] motors are rated for approximately 6000 rpm, which for an encoder resolution of 500 cpc[54] would require a "sampling rate" of 50 kHz. Furthermore, drift was not observed. Even after several joint movements, the arm would correctly find the vertical position when ordered to do so via MoveIt.

Finally, the explanation could be that the controller is not accurate enough. It was tuned for lower joint angles, and thus different loads, and that this invalidates the controller for higher angles cannot be excluded as an explanation. In particular, the sigmoid gain squaring `Kpti` implies that integral action is far lower than intended, and that the controller may effectively act as a P or PD controller. The impact of this on performance outside of the tuning range could not be ascertained, but seems like a probable error source.

18.4 Verification vs development

As is clear from previous subsections in this chapter, a lack of rigorous testing schema led to many qualitative findings which have no clear clear conclusion. This should be considered a failure to achieve scope item 4 from section 5.3. A myriad of minor "tests" were performed throughout the project in order to *verify* details as they were developed, but little time was set aside to *evaluate* performance of modules, or interplay between modules. The specialisation project actively employed the V-model[33] to gradually improve hardware modules, while this project tended to begin development of a new module as soon as positive results were seen from the current one. Conventions established early in the development phase were relied on to prevent "spaghetti"[57] code, in particular by strictly relying on safe interfaces to keep chaos contained to a single module.

To some extent, this was a deliberate choice. At one point it was decided that implementation of ROS features, and thereby the ability to control the arm not just by giving setpoints to single joints at a time, should be prioritised over development of telemetric capabilities; The ability to evaluate the arm was sacrificed for the benefit of controlling it. This was valuable with regards to achieving the project's main goal, but arguably affected its quality as a Master's project in embedded software development.

The telemetric data presented in section 16.8, for instance, was collected by making the torso unit output its received setpoints, calculated position and error directly via the UART interface. Were similar data to be collected from other units, the CAN module would have to be configured for transmission and reception of the same data. By all likelyhood, this would warrant the development of a separate telemery module with capabilities for both local and non-local data extraction, as well

as robust logging capabilities on the control computer. This likely would have taken about as much time as it took to implement the ROS modules: one week and change. Listing all the little problems which a rigorous telemetry module potentially could have solved is futile, but such a module would certainly have been valuable and should be highly prioritised in further development.

19 Conclusion

Six PCBs were produced according to improvement specifications carried over from the specialisation project. Most improvements were implemented successfully. Most notably, an error mode where one motor driver was disabled by interference between a timer and an ADC peripheral was resolved. Critical functions such as motor control and inter-MCU communication via CAN bus are operational. However, I2C and IMU functionality is largely non-functional. No clear cause could be established, but a foundation was laid for rigorous future testing. IMU functionality is critical to the secondary goal of the project. An error mode where CAN bus is intermittently disabled for system input voltages above 25V was discovered, and the cause could not be established. Furthermore, significant heat is emitted from voltage regulators for input voltages above 25V, possibly causing damage to nearby hardware elements.

A software system of 15 modules were developed for the control of the previously produced hardware, based on architectural requirements established early in the development phase and using the SOLID principles as guidelines. With the exception of functionality hampered by defunct hardware, the system largely achieves the project's primary and secondary goals. In particular, the Robotic Operating System holds good promise for future feature development as per the secondary goal. Some severe violations of architectural requirements were committed, but these may be rectified without significant changes to software structure.

A PID controller was implemented for the control of six linear and rotational joints, and tuned manually. Lack of telemetric capabilities paired with a lack of structured approach to testing made the final product difficult to evaluate. However, operational tests indicate that the system is at least marginally compatible with the primary use case. Further development of ROS functionality is necessary for autonomous beverage mixing, but this may be accomplished by manual operation of the arm in its current state. While difficult, pouring accurate amounts of liquid is possible. The arm does not appear to be able to lift objects massing more than approximately 0.9 kg, which may likely be attributed to the low system voltage of 25V, falling short of the mass of a typical 70 cl bottle by approximately 0.5 kg.

This project was a large undertaking for a single person, spanning a wide range of topics and abstraction levels. Considering the starting point of a list of hardware improvement points and outlines for hardware drivers left from the specialisation project, the implementation of a functional and expandable robotic system over the span of four months should be considered a success. Although some modules need improvement, the system shows great potential for further development and could become an interesting addition to Omega Verksted's portfolio of home-brew hardware projects.

20 Further work

Items presented in this section are based on sections 17 and 18, and are generally presented in an order of decreasing priority/increasing dependency on previous items.

20.1 Hardware

20.1.1 IMU

- Investigate cause for IMU failure further
- Breadboard IMU/I2C circuit design and verify
- Consider reestablishing shoulder IMU as a separate PCB
- Consider establishing hand IMU as a separate PCB by utilising unused hand mount points
- Ensure debug headers are available for all relevant signals

20.1.2 CAN bus

- Find cause for CAN bus failure above 25V
- Investigate possibility of causal relationship with high voltage regulator temperature
- Investigate poor grounding as a failure cause
- Investigate interference from power supply bus as a failure cause
- Twist CAN bus cable pair

20.1.3 LEDs

- Enable 3-5 unused MCU pins as GPIO and mount indicator LEDs

20.1.4 UART

- Move UART connection to the UART1 peripheral

20.1.5 USB

- Investigate viability of the USB circuit in light of AN4879[44]
- Prototype circuit using STM32F303 Nucleo devkit and breadboard
- Consider using the STM32F103 BluePill schematic as a basis

20.1.6 Encoder timers

- Investigate possibility of changing encoder timers to 32 bit timers (TIM2/3/4) to avoid overflow handling.

20.2 Software

20.2.1 FreeRTOS

- Consider introducing FreeRTOS
- Consider restructuring all main loop functions to run as ROS nodes

20.2.2 Joint controller

- Move IMU/accelerometer functionality to a separate module
- Move timer functionality to a separate module
- Investigate reason for control loop drastically increasing main loop execution time
- Consider introducing DMA and/or interrupts to decrease execution time
- Consider using more sensors for pose estimation: IMU rotation rate, torque calculation, multi-sensor blending

20.2.3 CAN driver

- Reconfigure message IDs to an addressing system based around control units
- Consider making use of the remote transmission request (RTR) bit for data request message types
- Move message definition and handlers to separate module

20.2.4 Motor driver

- Move encoder functionality to a separate module

20.2.5 Telemetry

- Create a telemetry module
- Implement local telemetry readouts
- Implement non-local telemetry readouts in conjunction with CAN message handlers
- Implement improved telemetry logging options in conjunction with ROS
- Consider outputting telemtry in raw byte formats rather than as text symbols to save bandwidth

20.2.6 Error handling

- Create an error handling module to provide a standardised error handling interface
- Implement safe fail states for safety critical functions
- Overhaul all modules to include error handling and error message output
- Implement error output and logging in conjunction with ROS

20.2.7 ROS

- Wrap all existing ROS nodes in a single program using a thread pool
- Expand serial communication node to include reading; dispense of PuTTy
- Expand HMI node capabilities such that it may act as a central node for ROS operations
- Implement telemetry logging capabilities

20.2.8 USB driver

- Assuming a viable USB circuit design is confirmed, create a driver
- Investigate possibility of misconfiguration of the VBUS detection interrupt as the cause for failure

20.3 Operations

20.3.1 State machine

- Investigate reason for inconsistent twist calibration
- Implement robustness to interruption, options to safely interact with the arm during calibration
- Implement a global state `GS_ERROR` in conjunction with error handling module
- Reimplement calibration state `CS_ERROR` as a meaningful error state

20.3.2 PID tuning/implementation

- Investigate cause for stability even with sign error in power setting calculation
- Reduce PID controller timer to less than half of the frequency of the main loop
- Remove squaring of `Kpti` in sigmoid gain calculation, retune controller
- Consider implementing a cascaded PID controller for control of both velocity and position
- Consider implementing a more sophisticated control scheme, such as MPC

20.3.3 MoveIt

- Reconfigure for use with pincer
- Implement task groups, in particular so that a pour may be centered more closely to a bottle's mouth
- Consider implementing feedback for real-time control in conjunction with telemetry module and/or FreeRTOS/ROS

Bibliography

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

[2] bk. *C default arguments*. URL: https://stackoverflow.com/questions/1472138/c-default-arguments (visited on 5th June 2024).

[3] 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.

[4] Kristian Blom. *Project Codeviz*. URL: https://github.com/kristblo/codeviz (visited on 30th May 2024).

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

[6] Kristian Blom. *Youtube: TTK4900 playlist*. URL: https://youtube.com/playlist?list=PLj8pbju16P5-arPH3SvBNVudJiaaosiVx&si=CbHY0YM3KSBwjJPh (visited on 29th May 2024).

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

[8] Bureau Moeilijke Dingen. *stm32-for-vscode*. URL: https://marketplace.visualstudio.com/items?itemName=bmd.stm32-for-vscode (visited on 31st May 2024).

[9] MoveIt documentation. *Pick and Place with MoveIt Task Constructor*. URL: https://moveit.picknik.ai/main/doc/tutorials/pick_and_place_with_moveit_task_constructor/pick_and_place_with_moveit_task_constructor.html (visited on 1st June 2024).

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

[11] eProsima. *micro-Ros overview: Features and Architecture*. URL: https://micro.ros.org/docs/overview/rtos/ (visited on 26th May 2024).

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

[13] Robert Bosch GmbH. *CAN specification Version 2.0*. 1991.

[14] Geoffrey Hunter. *Linux Serial Ports Using C/C++*. URL: https://blog.mbedded.ninja/programming/operating-systems/linux/linux-serial-ports-using-c-cpp/ (visited on 26th May 2024).

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

[16] RIGOL INCORPORATED. *1000Z Series Mixed Signal Oscilloscopes*. URL: https://www.rigolna.com/products/digital-oscilloscopes/1000z/ (visited on 5th June 2024).

[17] National Instruments. *Acquiring an Analog Signal: Bandwidth, Nyquist Sampling Theorem, and Aliasing*. URL: https://www.ni.com/en/shop/data-acquisition/measurement-fundamentals/analog-fundamentals/acquiring-an-analog-signal--bandwidth--nyquist-sampling-theorem-.html (visited on 1st June 2024).

[18] Harry W. Jones. *Common Cause Failures and Ultra Reliability*. NASA Ames Research Center, 2016.

[19] Kicad. *ROS2 Documentation: URDF*. URL: https://www.kicad.org/about/kicad/ (visited on 5th June 2024).

[20] Robert Martin. 'The Dependency Inversion principle'. In: *The C++ report* (1996).

[21] Robert Martin. 'The Interface Segregation Principle'. In: *The C++ report* (1996).

[22] Robert Martin. 'The Liskov Substitution Principle'. In: *The C++ report* (1996).

[23] Robert Martin. 'The Open-Closed Principle'. In: *The C++ report* (1996).

[24] Robert Martin. 'The Single Responsibility Principle'. In: *The C++ report* (1996).

[25] Matlab. *Designing Cascade Control System with PI Controllers*. URL: https://www.mathworks.com/help/control/ug/designing-cascade-control-system-with-pi-controllers.html (visited on 26th May 2024).

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

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

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

[29] Microsoft. *Visual Studio Code*. URL: https://code.visualstudio.com/ (visited on 5th June 2024).

[30] Nikoloas Minorsky. 'Directional Stability of Automatically Steered Bodies'. In: *Journal of the American Society for Naval Engineers* (1922).

[31] NXP. *NXP UM10204: I2C-bus specification and user manual*. 2021.

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

[33] Artem Oppermann. *builtIn: What Is the V-Model in Software Development?* URL: https://builtin.com/software-engineering-perspectives/v-model (visited on 1st June 2024).

[34] Laurids Petersen. *How do I generate a one second interrupt using an STM32 Timer?* URL: https://community.st.com/t5/stm32-mcus/how-to-generate-a-one-second-interrupt-using-an-stm32-timer/ta-p/49858 (visited on 2nd June 2024).

[35] Open Robotics. *ROS home page*. URL: https://www.ros.org/ (visited on 24th May 2024).

[36] Open Robotics. *ROS2 Documentation: Client libraries*. URL: https://docs.ros.org/en/iron/Concepts/Basic/About-Client-Libraries.html (visited on 3rd June 2024).

[37] Open Robotics. *ROS2 Documentation: Iron*. URL: https://docs.ros.org/en/iron/index.html (visited on 5th June 2024).

[38] Open Robotics. *ROS2 Documentation: URDF*. URL: https://docs.ros.org/en/iron/Tutorials/Intermediate/URDF/URDF-Main.html (visited on 3rd June 2024).

[39] Open Robotics. *ROS2 Tutorial: Creating a package*. URL: http://docs.ros.org/en/iron/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html#id4 (visited on 6th June 2024).

[40] PickNik Robotics. *MoveIt home page*. URL: https://moveit.ros.org/ (visited on 5th June 2024).

[41] Sigurd Rosland. *Reflow Oven*. URL: https://confluence.omegav.no/display/OV/Reflow+Oven (visited on 5th June 2024).

[42] Sigurd Skogestad. 'The setpoint overshoot method: A simple and fast closed-loop approach for PID tuning'. In: *DYCOPS* (2010).

[43] ST. *AN4206: Getting started with STM32F303 series hardware development*. 2015.

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

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

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

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

[48] ST. *STM32CubeMX*. URL: https://www.st.com/content/st_com/en/stm32cubemx.html (visited on 5th June 2024).

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

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

[51] ST. *USBLC6-2 Very low capacitance ESD protection*. 2020.

[52] Simon Tatham. *PuTTy*. URL: https://www.putty.org/ (visited on 5th June 2024).

[53] Controllers Tech. *PWM (Pulse Width Mod) in STM32*. URL: https://controllerstech.com/pwm-in-stm32/ (visited on 2nd June 2024).

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

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

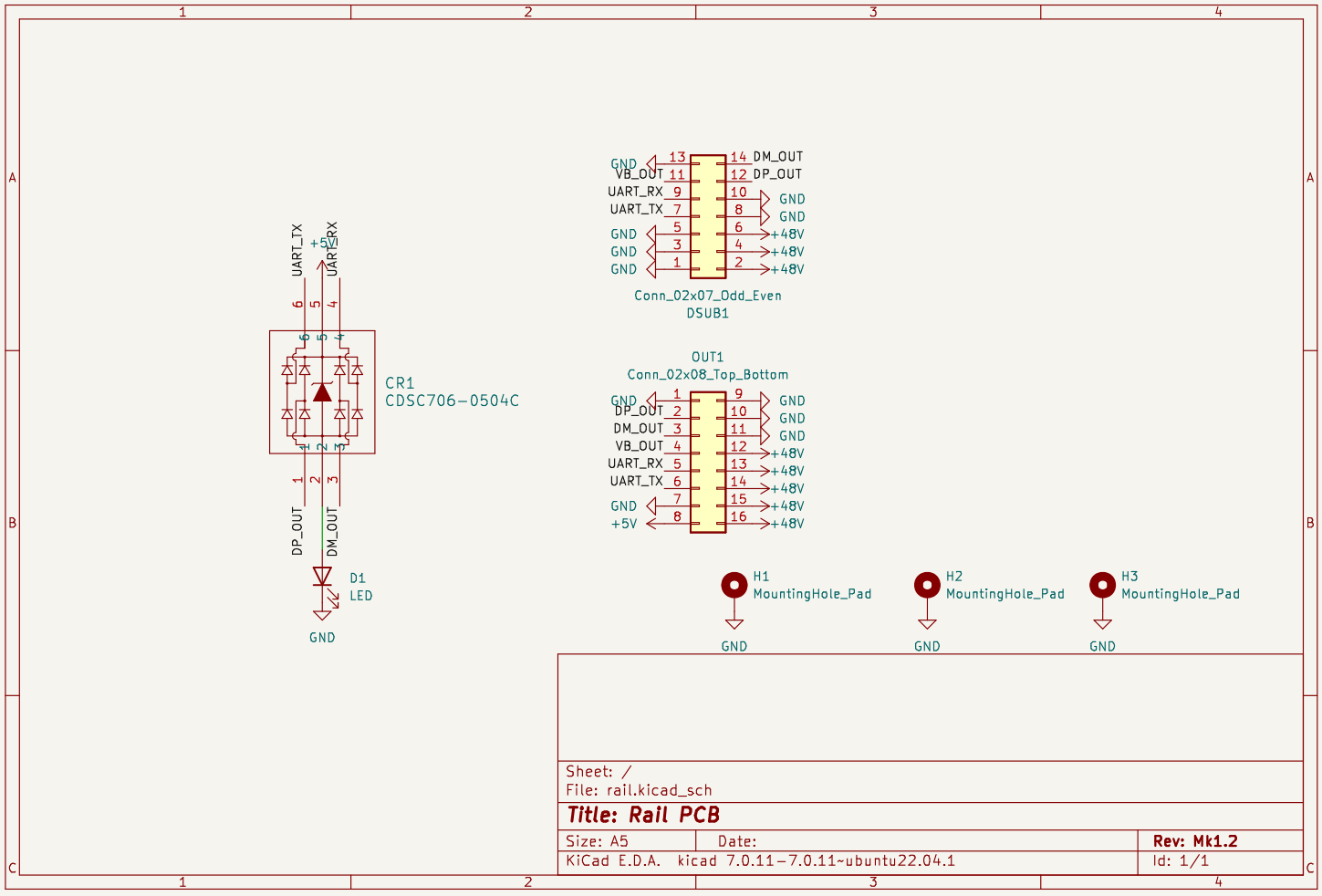
[56] TexasInstruments. *LMx17HV High Voltage Three-Terminal Adjustable Regulator With Overload Protection*. 2015.

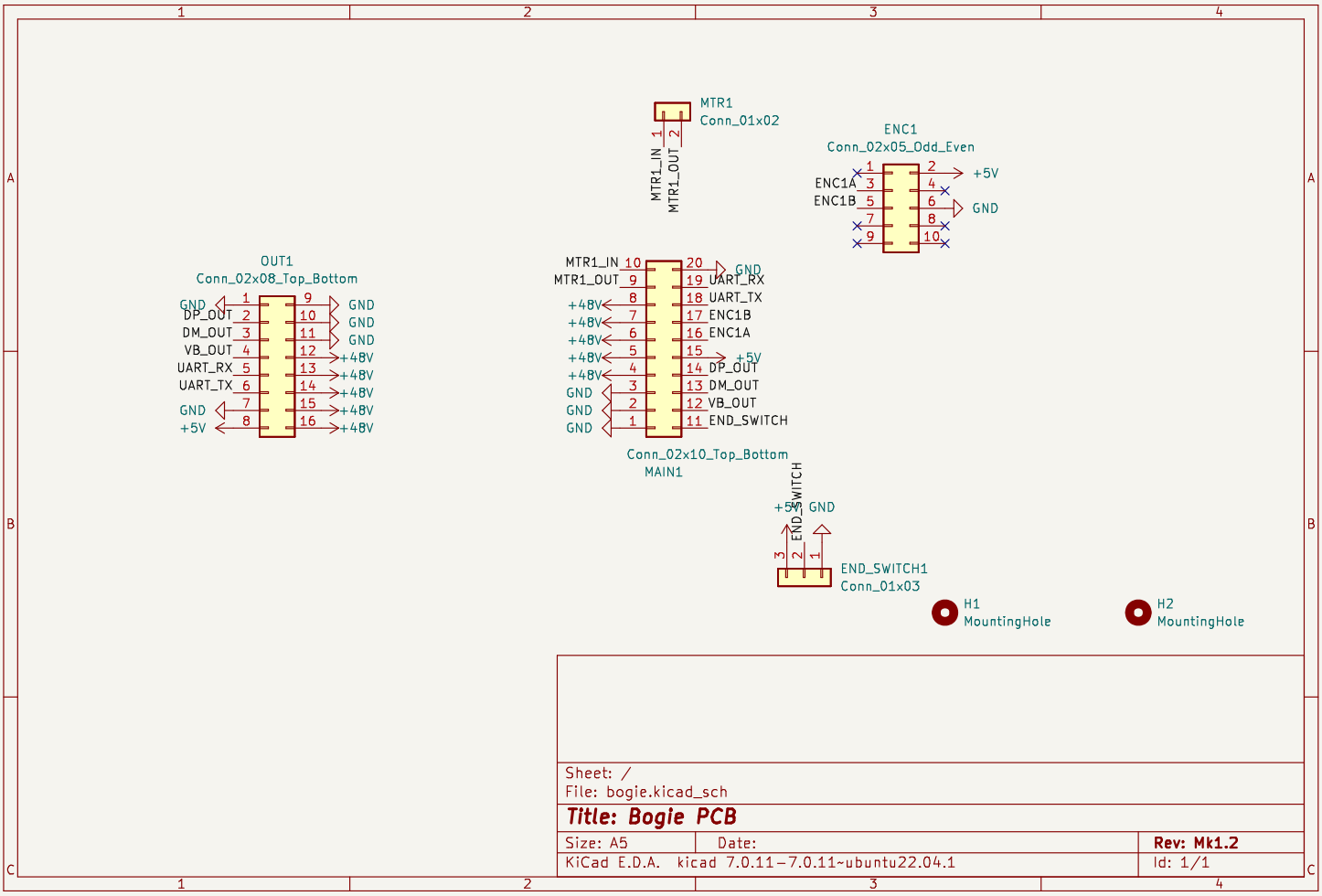
[57] Stephen Watts. *bmcBlog: What is Spaghetti Code (And Why You Should Avoid It)*. URL: https://www.bmc.com/blogs/spaghetti-code/ (visited on 1st June 2024).

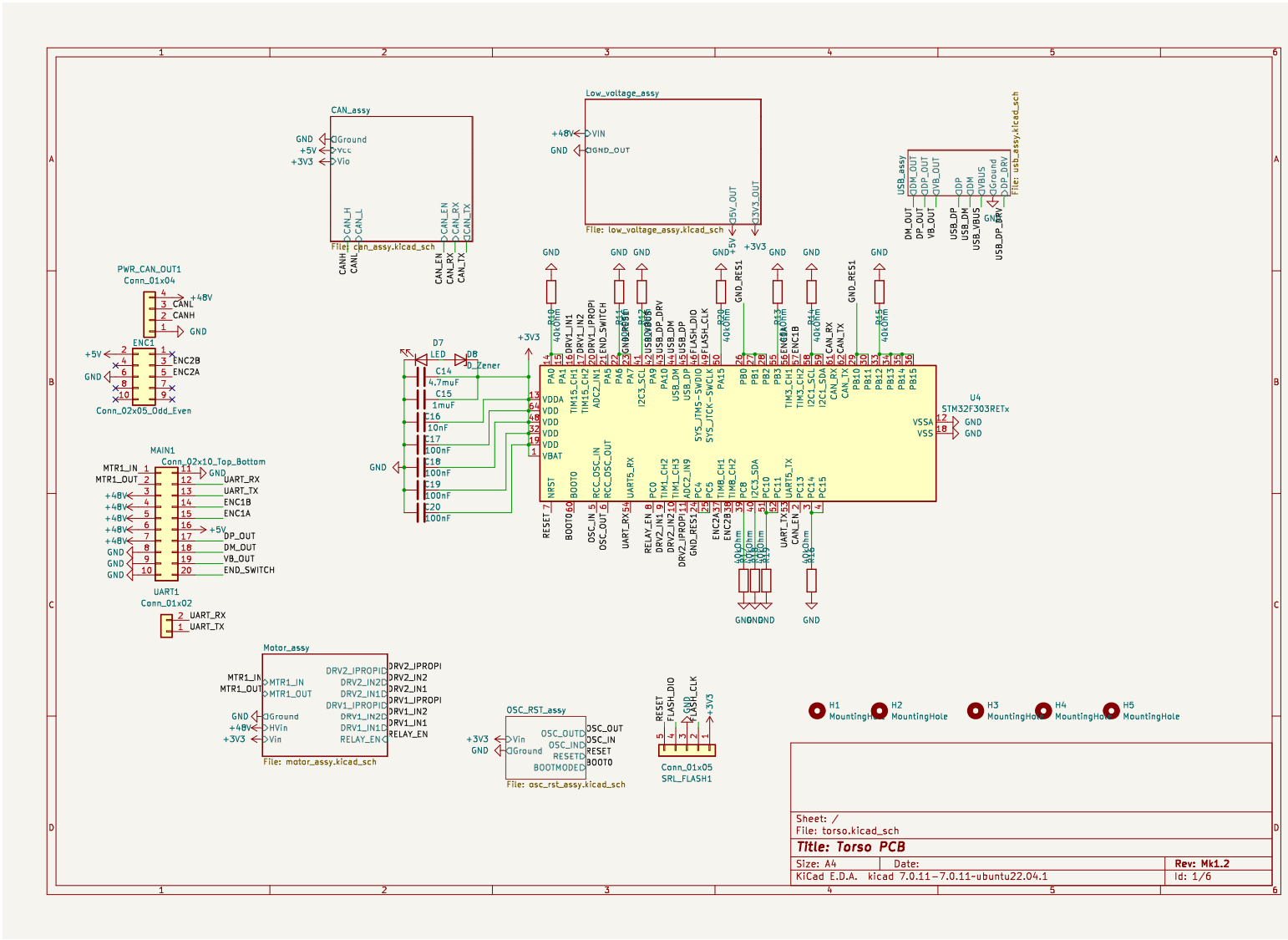
Appendix

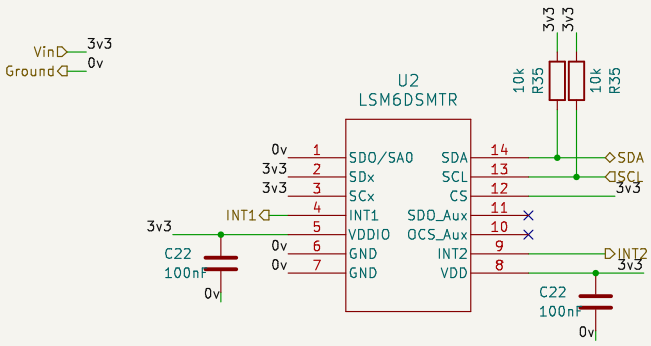
A Circuit diagrams

The following pages present circuit diagrams of the produced PCBs.

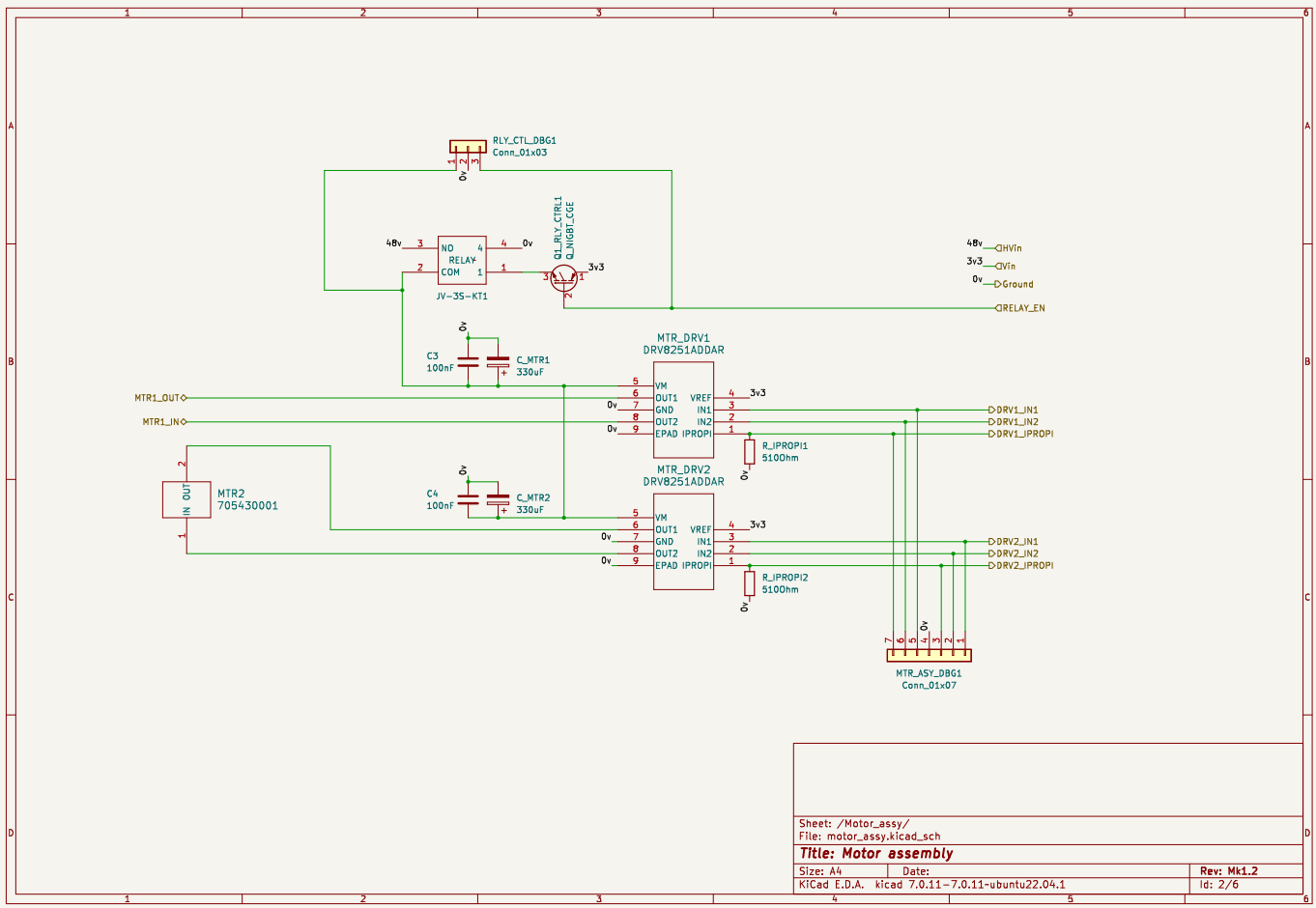

UART_TX
+5V
UART_RX
6
5
4
CR1
CDSC706-0504C
1
2
3
DP_OUT
DM_OUT
D1
LED
GND
GND 13
VB_OUT 11
UART_RX 9
UART_TX 7
GND 5
GND 3
GND 1
14 DM_OUT
12 DP_OUT
10 GND
8 GND
6 +48V
4 +48V
2 +48V
Conn_02x07_Odd_Even
DSUB1
OUT1
Conn_02x08_Top_Bottom
GND 1
DP_OUT 2
DM_OUT 3
VB_OUT 4
UART_RX 5
UART_TX 6
7
GND 8
+5V
9 GND
10 GND
11 GND
12 +48V
13 +48V
14 +48V
15 +48V
16 +48V
H1
MountingHole_Pad
GND
H2
MountingHole_Pad
GND
H3
MountingHole_Pad
GND
Sheet: /
File: rail.kicad_sch
Title: Rail PCB
Size: A5
Date:
Rev: Mk1.2
KiCad E.D.A. kicad 7.0.11-7.0.11~ubuntu22.04.1
Id: 1/1

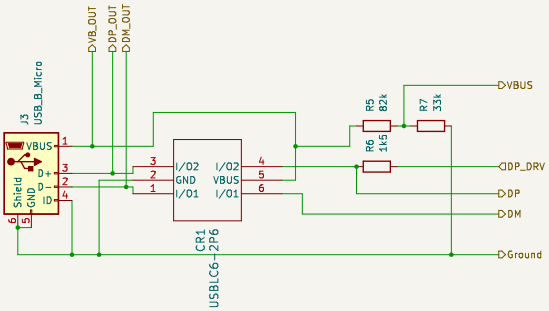

MTR1
Conn_01x02
MTR1_IN
MTR1_OUT
ENC1
Conn_02x05_Odd_Even
ENC1A
ENC1B
+5V
GND
OUT1
Conn_02x08_Top_Bottom
DP_OUT
DM_OUT
VB_OUT
UART_RX
UART_TX
+48V
MAIN1
Conn_02x10_Top_Bottom
UART_RX
UART_TX
ENC1B
ENC1A
DP_OUT
DM_OUT
VB_OUT
END_SWITCH
END_SWITCH1
Conn_01x03
H1
MountingHole
H2
MountingHole
Sheet: /
File: bogie.kicad_sch
Title: Bogie PCB
Size: A5
Date:
Rev: Mk1.2
KiCad E.D.A. kicad 7.0.11-7.0.11~ubuntu22.04.1
Id: 1/1


CAN_assy
Low_voltage_assy
File: can_assy.kicad_sch
File: low_voltage_assy.kicad_sch
File: usb_assy.kicad_sch
PWR_CAN_OUT1
Conn_01x04
ENC1
Conn_02x05_Odd_Even
MAIN1
Conn_02x10_Top_Bottom
UART1
Conn_01x02
Motor_assy
File: motor_assy.kicad_sch
OSC_RST_assy
File: osc_rst_assy.kicad_sch
Conn_01x05
SRL_FLASH1
U4
STM32F303RETx
H1
MountingHole
H2
MountingHole
H3
MountingHole
H4
MountingHole
H5
MountingHole
Sheet: /
File: torso.kicad_sch
Title: Torso PCB
Size: A4
Date:
Rev: Mk1.2
KiCad E.D.A. kicad 7.0.11-7.0.11~ubuntu22.04.1
Id: 1/6

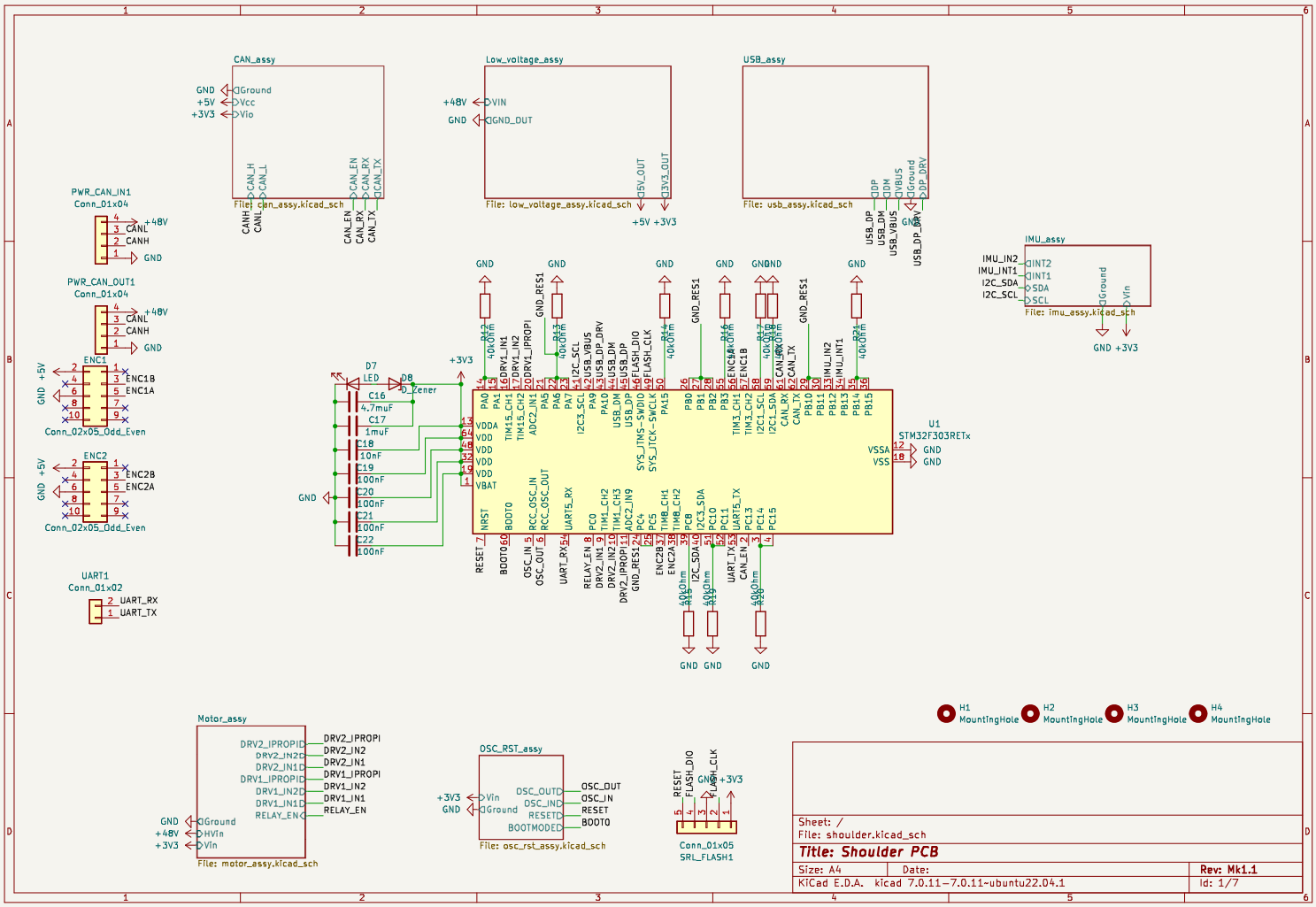


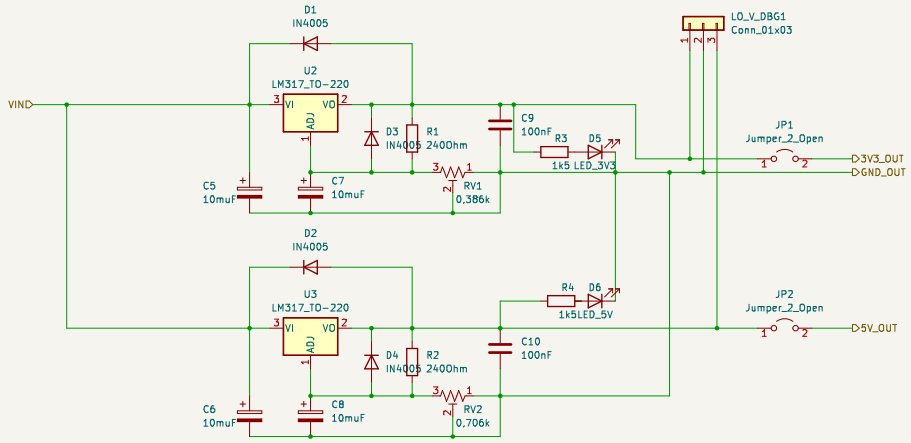
Sheet: /		
File: imu_assy.kicad_sch		
Title: IMU assembly		
Size: A5	Date:	**Rev: 1.1**
KiCad E.D.A. eeschema 7.0.11-7.0.11~ubuntu22.04.1		Id: 1/1


RLY_CTL_DBG1
Conn_01x03
0v
48v
NO
RELAY
COM
JV-3S-KT1
Q1_RLY_CTRL1
Q_NIGBT_CGE
3v3
0v
48v
HVin
3v3
Vin
0v
Ground
RELAY_EN
C3
100nF
C_MTR1
330uF
MTR_DRV1
DRV8251ADDAR
VM
OUT1
GND
OUT2
EPAD
VREF
IN1
IN2
IPROPI
MTR1_OUT
MTR1_IN
R_IPROPI1
5100hm
DRV1_IN1
DRV1_IN2
DRV1_IPROPI
MTR2
705430001
IN
OUT
C4
100nF
C_MTR2
330uF
MTR_DRV2
DRV8251ADDAR
R_IPROPI2
5100hm
DRV2_IN1
DRV2_IN2
DRV2_IPROPI
MTR_ASY_DBG1
Conn_01x07
Sheet: /Motor_assy/
File: motor_assy.kicad_sch
Title: Motor assembly
Size: A4
Date:
Rev: Mk1.2
KiCad E.D.A. kicad 7.0.11-7.0.11-ubuntu22.04.1
Id: 2/6

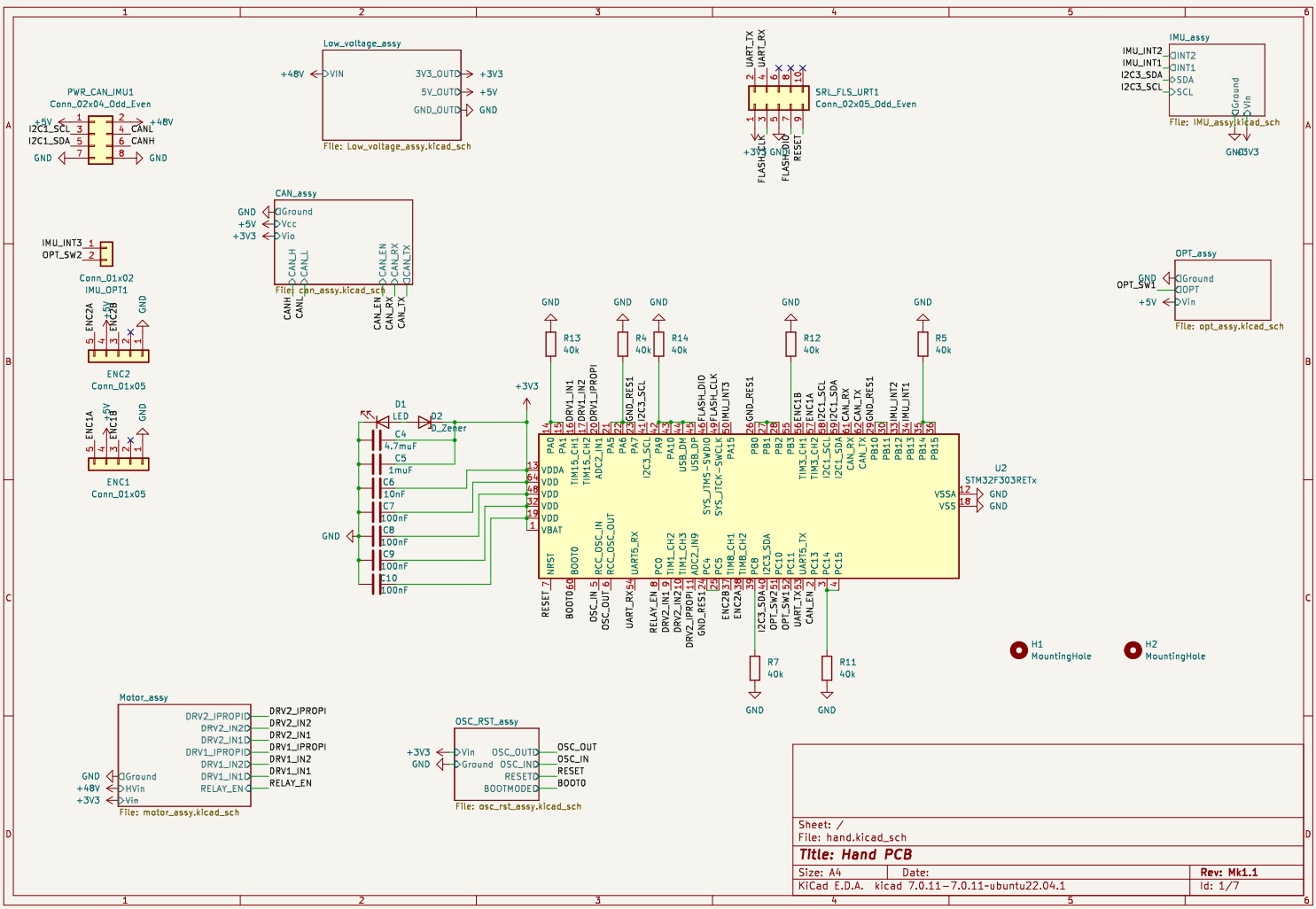


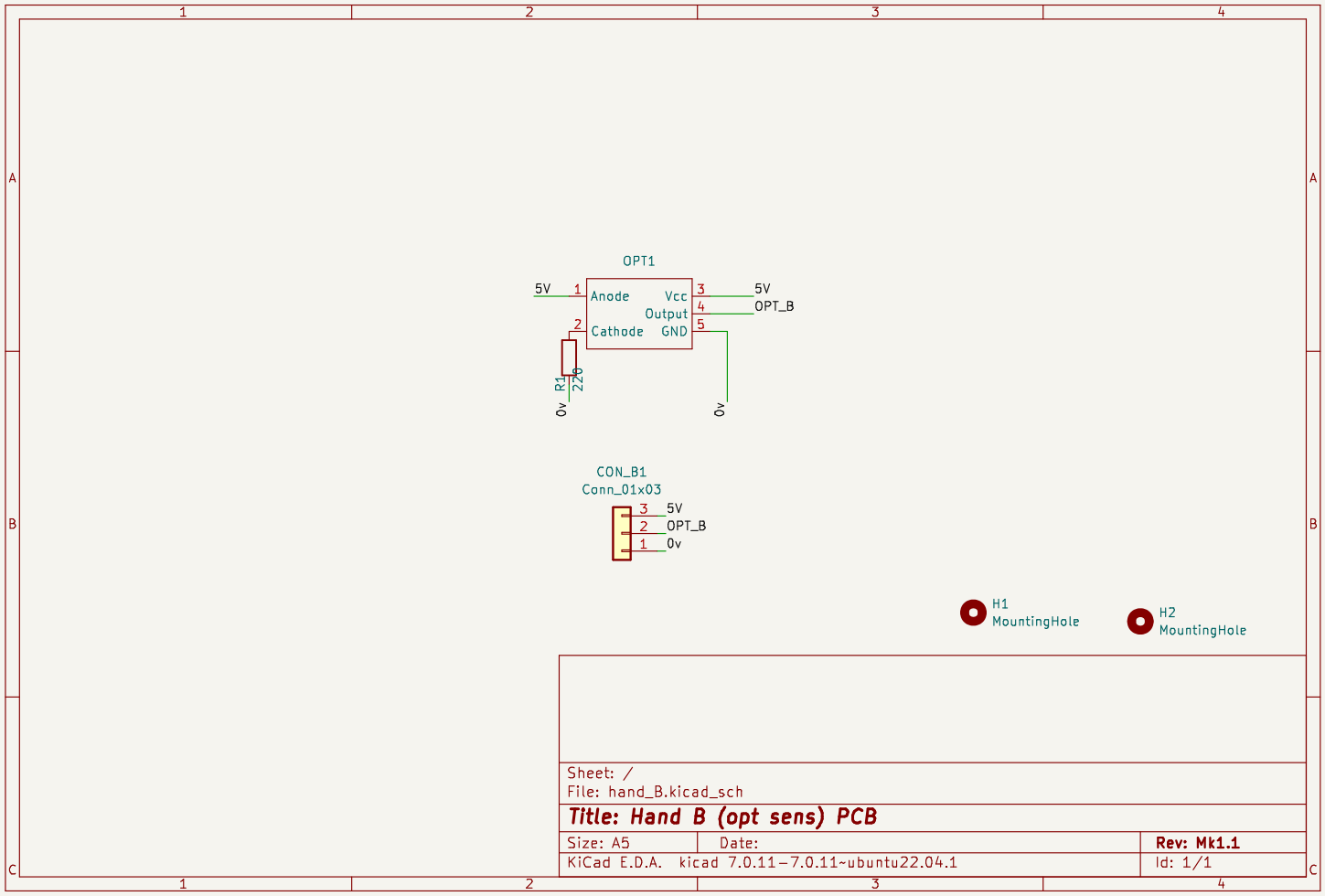
Sheet: /		
File: usb_assy.kicad_sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. eeschema 7.0.11-7.0.11~ubuntu22.04.1		Id: 1/1

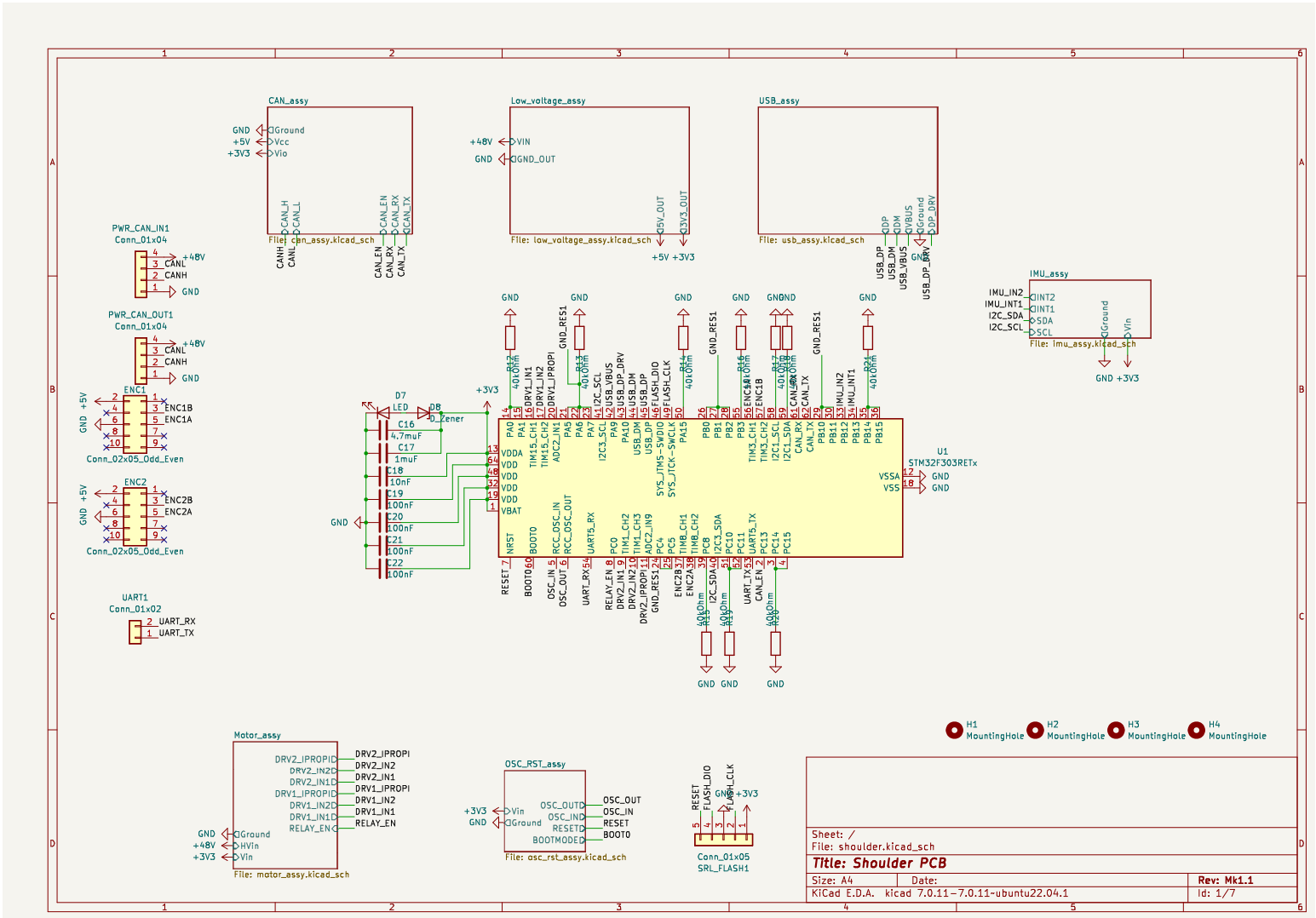

CAN_assy
Low_voltage_assy
USB_assy
IMU_assy
PWR_CAN_IN1
PWR_CAN_OUT1
ENC1
ENC2
UART1
U1
STM32F303RETx
Motor_assy
OSC_RST_assy
Conn_01x05
SRI_FLASH1
H1 MountingHole
H2 MountingHole
H3 MountingHole
H4 MountingHole
Sheet: /
File: shoulder.kicad_sch
Title: Shoulder PCB
Size: A4
Date:
Rev: Mk1.1
KiCad E.D.A. kicad 7.0.11-7.0.11~ubuntu22.04.1
Id: 1/7

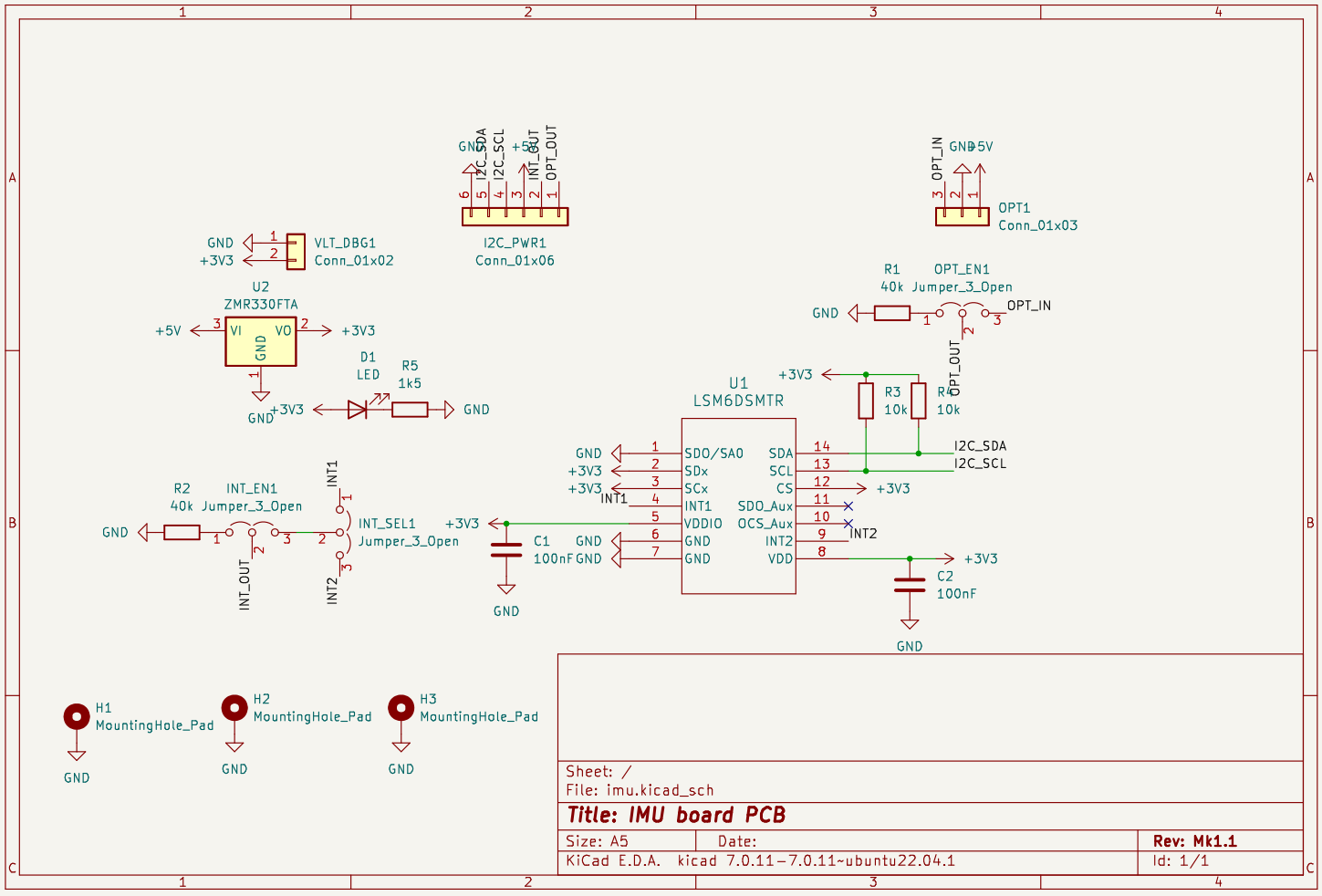


Sheet: /		
File: low_voltage_assy.kicad_sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. eeschema 7.0.11-7.0.11~ubuntu22.04.1		Id: 1/1


Low_voltage_assy
File: Low_voltage_assy.kicad_sch
PWR_CAN_IMU1
Conn_02x04_Odd_Even
SRL_FLS_URT1
Conn_02x05_Odd_Even
IMU_assy
File: IMU_assy.kicad_sch
CAN_assy
File: can_assy.kicad_sch
IMU_OPT1
Conn_01x02
ENC2
Conn_01x05
ENC1
Conn_01x05
OPT_assy
File: opt_assy.kicad_sch
U2
STM32F303RETx
H1
MountingHole
H2
MountingHole
Motor_assy
File: motor_assy.kicad_sch
OSC_RST_assy
File: osc_rst_assy.kicad_sch
Sheet: /
File: hand.kicad_sch
Title: Hand PCB
Size: A4
Date:
Rev: Mk1.1
KiCad E.D.A. kicad 7.0.11-7.0.11~ubuntu22.04.1
Id: 1/7


OPT1
5V
1
Anode
Vcc
3
5V
Output
4
OPT_B
2
Cathode
GND
5
R1
220
0v
0v
CON_B1
Conn_01x03
3
5V
2
OPT_B
1
0v
H1
MountingHole
H2
MountingHole
Sheet: /
File: hand_B.kicad_sch
Title: Hand B (opt sens) PCB
Size: A5
Date:
Rev: Mk1.1
KiCad E.D.A. kicad 7.0.11-7.0.11~ubuntu22.04.1
Id: 1/1


CAN_assy
Low_voltage_assy
USB_assy
IMU_assy
PWR_CAN_IN1
PWR_CAN_OUT1
ENC1
ENC2
UART1
Motor_assy
OSC_RST_assy
U1
STM32F303RETx
H1 MountingHole
H2 MountingHole
H3 MountingHole
H4 MountingHole
Sheet: /
File: shoulder.kicad_sch
Title: Shoulder PCB
Size: A4
Date:
Rev: Mk1.1
KiCad E.D.A. kicad 7.0.11-7.0.11-ubuntu22.04.1
Id: 1/7


GND
I2C_SDA
I2C_SCL
+5V
INT_OUT
OPT_OUT
I2C_PWR1
Conn_01x06
OPT_IN
OPT1
Conn_01x03
VLT_DBG1
Conn_01x02
U2
ZMR330FTA
VI
VO
+5V
+3V3
D1
LED
R5
1k5
R1
40k
OPT_EN1
Jumper_3_Open
R3
10k
R4
10k
U1
LSM6DSMTR
SDO/SA0
SDx
SCx
INT1
VDDIO
GND
SDA
SCL
CS
SDO_Aux
OCS_Aux
INT2
VDD
I2C_SDA
I2C_SCL
C1
100nF
C2
100nF
R2
40k
INT_EN1
INT_SEL1
Jumper_3_Open
INT_OUT
H1
MountingHole_Pad
H2
MountingHole_Pad
H3
MountingHole_Pad
Sheet: /
File: imu.kicad_sch
Title: IMU board PCB
Size: A5
Date:
Rev: Mk1.1
KiCad E.D.A. kicad 7.0.11-7.0.11~ubuntu22.04.1
Id: 1/1