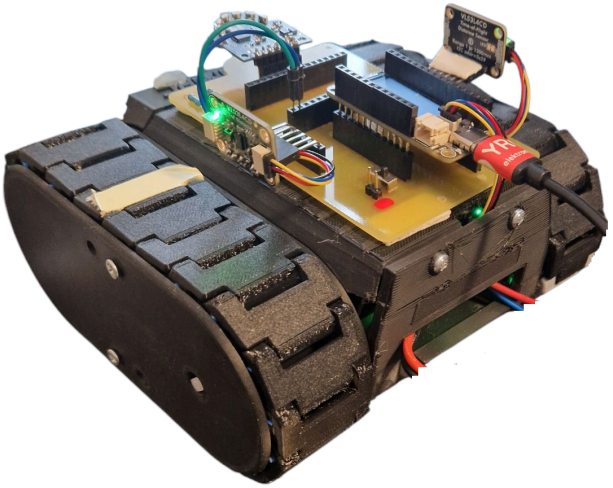
Students::
Niklas Brodén, Milad Makdesi,

Ghost Division



YRGO

Summary

This project was part of the Elektronikprojekt course at Yrgo. The goal was to build an autonomous robot that could drive around a track. We created two vehicles: OmniX, our main robot, and Panzer, a simpler backup made from spare parts.

OmniX used an ESP32 S3 microcontroller and several sensors. We originally planned to use AI for steering, but due to technical problems, we switched to a rule-based system that was more reliable. We also aimed to reuse the same logic unit for both vehicles but focused on OmniX as it became our priority.

We faced issues like I2C congestion and power spikes that shut down the motor shield. These were fixed by using two motor shields, improving the power setup, and adjusting the software. We ran stress and current spike tests to make sure the system was stable.

The team followed agile methods using Jira for task management and GitHub for version control. Two sprints were interrupted—one due to missing parts and another due to PCB delays.

In the end, both robots could drive on their own. We learned a lot about embedded systems, teamwork, and solving practical problems.

1. Introduction

This project was completed as part of the course *Elektronikprojekt* in the *Electronics Engineering – Autonomous Systems* program at Yrgo [1]. The assignment was to work in a team to plan, build, and deliver an autonomous vehicle capable of independently navigating a predefined track in the Yrgo Grand Prix competition.

The work included both theoretical and practical components. The theoretical side involved applying agile methods, working in sprints, using Jira for task management, and presenting progress during sprint demos. On the practical side, we developed and tested two vehicles:

- **OmniX**, our main robot focused on speed, control, and advanced sensor handling
- **Panzer**, a simpler backup robot built using leftover components

The project involved custom PCB design, embedded programming, real-time sensor integration, and system testing. We used GitHub for version control and Google Drive for collaboration in the early stages.

The main goal was not only to build functional robots, but also to strengthen our skills in electronics design, embedded systems, teamwork, and structured project planning. This report outlines our development process, key design decisions, the challenges we encountered, and what we learned from building and testing both platforms.

2. Risk analysis

We used a risk matrix to plan for possible problems during the project. The biggest concerns were system crashes, power issues, and delays that could affect testing. We also tracked risks like hardware failures, budget overruns, and poor team coordination. This helped us focus on the most critical areas and prepare backups where needed.

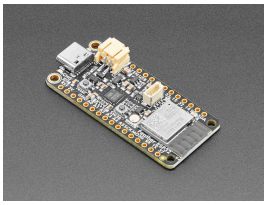
Low Risk - High Impact • Critical Hardware failure or malfunction during final demo. • Not meeting rules or project requirements. • Budget overrun • Data loss	High Risk - High Impact • Power issues • Severe Software bugs • Excessive tests leading to component failure • Software module incompatibility
Low Risk - Low Impact • Minor hardware changes • Team workflow issues	High Risk - Low Impact • Delays in component delivery

3. Hardware Overview

This section outlines the hardware components used across both vehicles, OmniX and Panzer. While many parts were shared between the two platforms, each robot also included unique modules tailored to its design and purpose. The selection of components was based on functionality, availability, and ease of integration. Below is a breakdown of shared and platform-specific hardware.

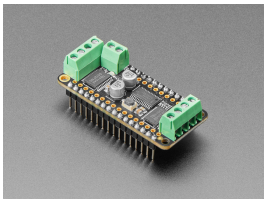
3.1 Shared Components (OmniX and Panzer):

Microcontroller – ESP32-S3 TFT Feather



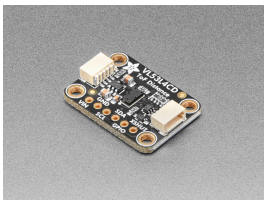
Used in both vehicles as the final controller. Offers high processing speed, Wi-Fi, and TensorFlow Lite support. The built-in display was useful for debugging and status feedback.

Motor Driver – Adafruit Motor Shield V2



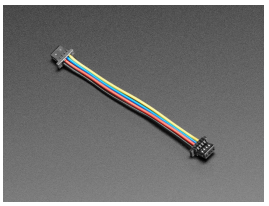
One Motor FeatherWings can drive up to four DC motors. Each board supports 1.2 A continuous and 3.2 A peak per side. They connect via I²C and stack directly on the ESP32 Feather.

Sensors - VL53L4CD



These Time-of-Flight sensors provide accurate readings up to 1300 mm at 100 Hz with an 18° field of view. Each sensor communicates via I²C and handles its own processing, reducing the load on the ESP32.

Cables – STEMMA QT / Qwiic



A variety of STEMMA QT cables were used to connect sensors to the ESP32. Their plug-and-play design simplified I²C wiring and helped maintain a clean, modular setup.

Battery – 3.7 V LiPo 500 mAh

A 3.7 V 500 mAh LiPo battery powered the ESP32 and charged automatically via USB through the onboard circuit.

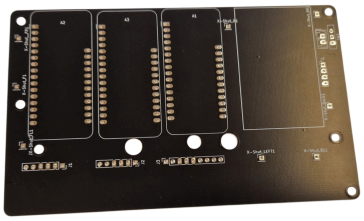
DC Motor – 1:48 Gear Ratio, 230 RPM @ 6 V

We used low-cost DC motors with a 1:48 gearbox, rated at 230 RPM @ 6 V. They were run at up to 12 V for higher speed during testing. Their low price made them ideal for development, as they were easy to replace if damaged.

PCB's - A base for the logic units and peripherals

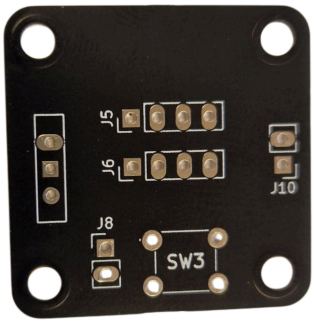
Two PCB's were designed, one for each vehicle. The intention was to have a sturdy mount for the various components, but also reduce the wiring and bad connections.

Features that proved important and also makes managing the vehicles easier, are main switches and battery connections.



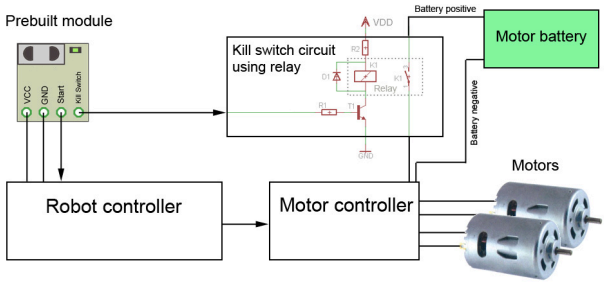
PCB with prepared for assembly of an ESP-32, 2 Motor-Shields and 2 INA219 and 1 LSM6DSOX. Connections for the external Carrier plate, Logic's Battery and 8 GPIO's placed at optimal positions.

The Carrier-Plate. This was to be an out-board mount for easy access in regard to the Logic Main switch, Reset Button and mount for the mandatory Kill Switch.



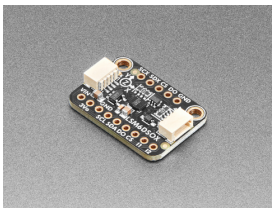
Kill Switch

The mandatory Kill Switch was provided by the Faculty. [1]



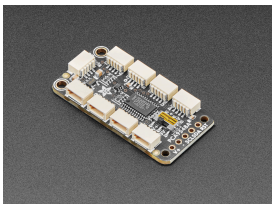
3.2 OmniX-Specific Components

IMU – LSM6DSOX



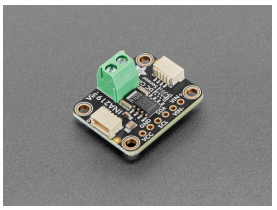
The LSM6DSOX combines a 3-axis accelerometer and gyroscope. It connects over I²C and supplies motion and orientation data used in AI navigation.

I²C Multiplexer – PCA9548A



To avoid address conflicts among the 8 identical VL53L4CD sensors, a PCA9548A multiplexer was used. It allows selecting one sensor at a time via I²C, supports 8 channels, and eliminates the need for XSHUT-based management.

Current Sensor – INA219



INA219 monitors both current and bus voltage from each motor shield via a shunt resistor and I²C. It provides real-time power data, helping detect overloads and analyze motor performance.

Power – Buck Converter and 7.4 V Li-ion batteries

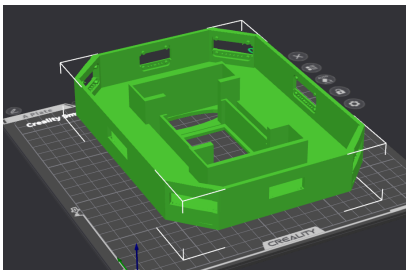
We used the XL4016 buck converter to step down voltage from two 7.4 V Li-ion batteries in series to 12 V. The output was split and fed into each motor shield to provide stable power for all four motors.

Wheels – Mecanum

The robot uses Mecanum wheels for full omni-directional movement, allowing it to move forward, sideways, and rotate in place. This enabled precise control and flexible navigation in tight spaces.

Mounts – Motors, Wheels, and Platform

The chassis included metal brackets for mounting the DC motors and Mecanum wheels. All other components—ESP32, sensors, wiring, power modules, and a custom PCB—were integrated into a 3D-printed top platform, creating a compact and organized setup.

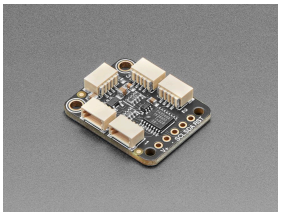


3.3 Panzer-Specific Components

Sensor - Pololu OPT3101

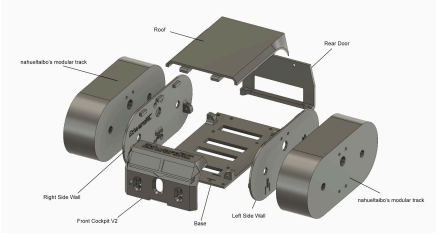
This board is a 3-channel time-of-flight proximity and distance sensor module based on the OPT3101 IC from Texas Instruments. It emits infrared light in one of three selectable directions with its six integrated LEDs and measures distance by measuring the time delay of the reflected signal. Distance measurements can be read through a digital I²C interface. The combined field of view of the three sensing zones is almost 180 degrees, and the maximum range is about one meter.

I2C Multiplexer - PCA9546 4-Channel



Based on the same chip as the 8 channel multiplexer used for the Omnix. This one is a bit smaller in size and handles 4 channels to direct traffic through I2C. Switching from one component to another through channel select, thus avoiding conflicts on the I2C bus.

Panzer - Platform



Fully 3d-printed chassis and caterpillar-tracks. Propulsion is provided by two DC-motors. one for each track. The chassis was printed in PLA and the tracks were printed in TPU, a softer material for better traction.

12V Li-Ion 2200mAh - Battery

Packning a lot of power. This was intentionally meant to be used, or at least tested on the OmniX. However, when the time came, it ended up in the Panzer instead. With deadlines just around the corner it was more than well suited to power the Dual motorized vehicle.

4. Bill of Materials and Budget Overview

Description	Datasheet Link or Product page	Quantity	Cost(SEK)
K036-B - RoverC Pro Robot Kit without M5StickC	https://drive.google.com/file/d/1b2hhEkuM56H0nMTga7DYlkJySmzQrnOx/view?usp=drive_link	1	622.15
Step Down Converter 3-pack	https://drive.google.com/file/d/1h2xfD3BGcCHJC_UatQv3dfHRDUJdiJ6k/view?usp=sharing	1	259
2-Pack Li-ion Battery 7.4V 2000mAh	Yangers Set med 2 uppladdningsbara batterier Li-ION 7,4 V 2000 mAh	2	579.98
EEMB 3.7V Lipo Battery 750mAh 403048 Lithium Polymer Ion Battery	Batteri LiPo 3.7V 750mAh	1	106.09
Mecanum Wheel Smart Robot Chassis Black	MC100 4WD Mecanum Wheel Car Chassis	1	588.52
Adafruit VL53L4CD Time of Flight Distance Sensor -1 to 1300mm - STEMMA QT / Qwiic	https://drive.google.com/file/d/1PxcT7UWGe5pBz1wQUa3RLsYKIfSBkgUB/view?usp=drive_link	8	1,278.80
Adafruit LSM6DSOX 6 DoF Accelerometer and Gyroscope - STEMMA QT / Qwiic	https://drive.google.com/file/d/1XgdywUMPakZXrJ3tgltQXDkEkLLazut2/view?usp=sharing	1	127.77
Assembled DC Motor + Stepper FeatherWing Add-on	https://drive.google.com/file/d/1gyc1GhMOVHmwuwg4DpKB-D3yMrwJ3d8i/view?usp=drive_link	1	230
Adafruit ESP32-S3 TFT Feather - 4MB Flash 2MB PSRAM STEMMA QT	https://drive.google.com/file/d/14n239eX5Ng0weeAf4D5G7RgwZXF2Vf4M/view?usp=sharing	1	266.8
Adafruit ESP32 Feather V2 - 8MB Flash + 2 MB PSRAM - STEMMA QT	https://drive.google.com/file/d/1gn5KPkuPtiD2LDpWcZPvSd_jO6-hXlm7/view?usp=sharing	1	213.33
STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long	https://drive.google.com/file/d/1batPVD2Dx57AelLxmWnUrqz5hQ7gED7h/view?usp=sharing	10	101.7
STEMMA QT / Qwiic JST SH 4-pin Cable - 100mm Long	https://drive.google.com/file/d/1pi0F7eCOmWcd-klvUYV5x2-UiTQ6-ogE/view?usp=sharing	10	101.7
FeatherWing OLED - 128x64 OLED Add-on For Feather - STEMMA QT / Qwiic	https://drive.google.com/file/d/1YbQUr_UWcsF_7wQAYzPqulFRo_UpmD24/view?usp=drive_link	1	159.85
STEMMA JST PH 2mm 3-Pin to Female Socket	https://drive.google.com/file/d/1GXs	2	26.68

Cable - 200mm	ASVDAJVmtx412rSu65lTkN009tPc0/view?usp=sharing		
STEMMA JST PH 2mm 3-Pin to Male Header Cable - 200mm	https://drive.google.com/file/d/1GXsASVDAJVmtx412rSu65lTkN009tPc0/view?usp=sharing	2	26.68
STEMMA QT / Qwiic JST SH 4-Pin Cable - 200mm Long	https://drive.google.com/file/d/131fmbCn26Tg7JjI9D_cCTzexVlUljevw/view?usp=sharing	1	13.34
SparkFun STEMMA QT / Qwiic Breadboard Breakout Adapter	https://drive.google.com/file/d/1Jp78VtLWkWrBvmSjOWK9ScAYCc67JkRg/view?usp=sharing	1	27.03
FeatherWing Doubler - Prototyping Add-on For All Feather Boards	https://drive.google.com/file/d/1Z79k46t-Uz9Lviss50w-FrRjbI93rL0d/view?usp=sharing	1	80.27
Adafruit PCA9548 8-Channel STEMMA QT / Qwiic I2C Multiplexer	https://drive.google.com/file/d/1kqalJfDoLsUuuL_-KyI_MeR6rSRguuh0/view?usp=sharing	2	150.19
DC-motor med kuggväxel 1:48 230rpm 6V	DC-motor med kuggväxel 1:48 230rpm 6V till rätt pris @ electrokit	4	185.6
Adapterkabel QWIIC till/från Grove 100mm	QWIIC till/från Grove 100mm	4	124.8
GUUZI 8st TT DC växellåda Motor dubbelaxel 3-6V växelmotor för Arduino Smart bilrobot	GUUZI 8st TT DC växellåda Motor dubbelaxel 3-6V	1	97.18
AFTERTECH 12V 7000mAh 7Ah LITIULADDNINGSBART BATTERIPACK 75x60x40 mm F3C2	Talentcell 12V Rechargeable Lithium ion Battery Pack	1	669.62
STEMMA QT/QWIIC CABLE 300MM	https://drive.google.com/file/d/19NYMR5dFD0eGxY1_ZPGxmQMM45fZj14a/view?usp=sharing	5	63.2
			6100.28

5. OmniX

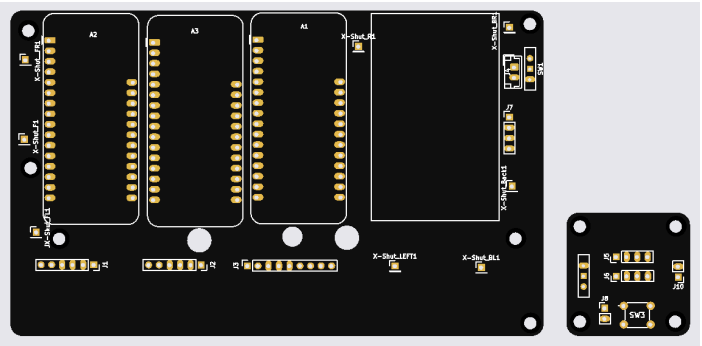
5.1 Construction

The platform of the OmniX lacked fittings for peripherals. This had been taken into account already. Still, this didn't make our lives easier. No one in our team possessed any 3D-cad skills. Although one of the team members was in possession of a 3D-printer, so we had a given "volunteer".

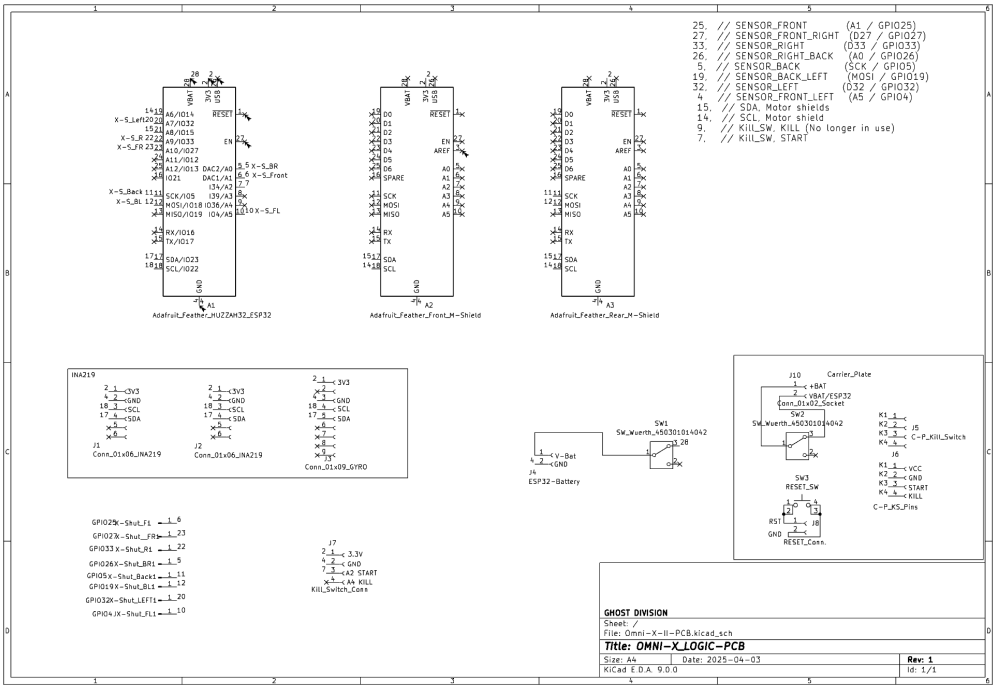
The first build was intended as a mockup to get a feeling for the final build. It was a basic design, with a simple goal. To harbor the batteries and logic units, and to some degree start testing.

First build. Two Battery bays on each side of the step down converter bay in the centre, cut-outs for 8 VL53L4CD sensors.

The second iteration was also intended as a version 0.XX but became the final build. This one was quite a bit bulkier and designed to meet maximum specs. The first build helped us to better figure out optimal component placement in regard to sensors, but also the main power source with its peripherals and logic components. The need to have everything neat and tight, except from the visual aspect was quite important from assembly point of view and also to minimize risk of faulty connections. This granted the development of a PCB.



OmniX with a carrier plate for external assembly.



OmniX PCB-schematic

The design layout was not that complicated since we would mainly use the PCB as a holder for the MCU and Motor Shields. During the design process we decided to incorporate connections for GPIO pins on the VL53L4CD sensors. The first iteration, without the GPIO pins, was milled in-house on a single-layer PCB. This one was discarded due to faults in the milling process. This however, gave us the opportunity to add the GPIO pins for the sensors. When the faults reiterated themselves a second and third time. We decided to order the PCB from JLCPCB in China.

Once the PCB had arrived the assembly was promptly finalized and more parts 3D-printed, such as a cover and logic module housing. These were later discarded due to weight regulations.

5.2 Software Overview

The software was written in C++ and runs on the ESP32 using the FreeRTOS and Arduino framework. It is modular and divided into separate files, each responsible for a specific subsystem such as AI steering, controller input, sensor handling, motor control, and wireless communication.

Core Features

- **FreeRTOS task management**: Separate tasks for sensors, motors, telemetry, and AI logic.
- **Manual and autonomous modes**: Switchable between Xbox controller input and AI control.
- **Real-time telemetry**: Sensor and system data are sent via Wi-Fi (UDP) to a PC.
- **Live parameter tuning**: Parameters can be updated over Wi-Fi during runtime.
- **Sensor filtering**: Filters invalid or noisy VL53L4CD readings to improve reliability.

File Structure

- **ai.cpp / ai.h** – Rule-based autonomous steering logic (no ML)
- **controller.cpp / controller.h** – Xbox controller input via Bluetooth.
- **sensors.cpp / sensors.h** – VL53L4CD sensor reading and filtering.
- **motors.cpp / motors.h** – Motor speed and direction control.
- **params.cpp / params.h** – Adjustable steering and filtering parameters.
- **wifi_setup.cpp / wifi_setup.h** – Wi-Fi and UDP initialization.
- **udp_handler.cpp / udp_comm.h** – Parses incoming parameter updates.
- **udp_sender.cpp** – Sends telemetry data over Wi-Fi.
- **imu.cpp / imu.h** – Reads motion data from the LSM6DSOX IMU.
- **tasks.cpp / tasks.h** – Starts and manages FreeRTOS tasks.
- **mux.cpp / mux.h** – Controls the PCA9548A I^2C multiplexer.
- **xshut.cpp / xshut.h** – Optional sensor reset logic (unused in final build).
- **globals.h** – Shared constants and global state variables.

5.3 Python Program – Visualization and Control

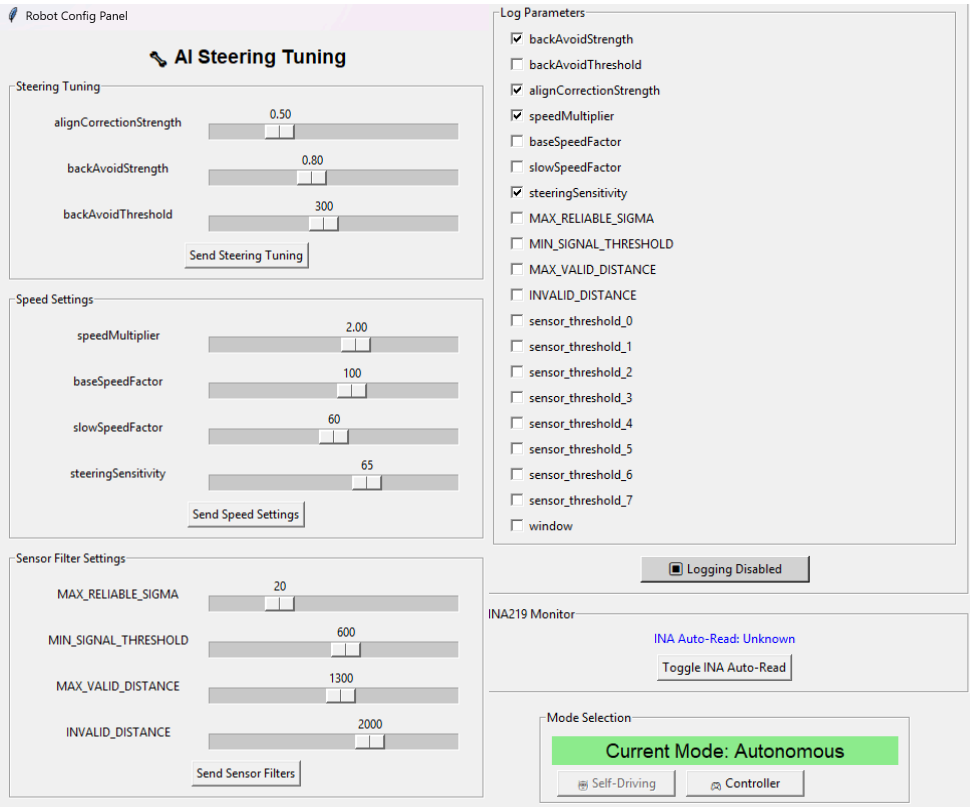
A custom Python program was developed to support debugging, visualization, and live parameter tuning during development. It communicates with the ESP32 over Wi-Fi using UDP and provides a graphical interface for real-time monitoring.

File Structure

- **main.py** – Entry point, launches UI and visualizer.
- **ui.py** – Tkinter UI with parameter sliders and mode controls.
- **visualizer.py** – Pygame-based display of sensor and robot state.
- **udp_comm.py** – Manages UDP communication.
- **logger.py** – Logs data to CSV.
- **config.py** – Stores IPs, ports, and default parameters.

Key Features

- **Graphical UI**: Tkinter-based interface for adjusting steering, speed, and filtering parameters.
- **Mode switching**: Allows toggling between manual and autonomous driving.

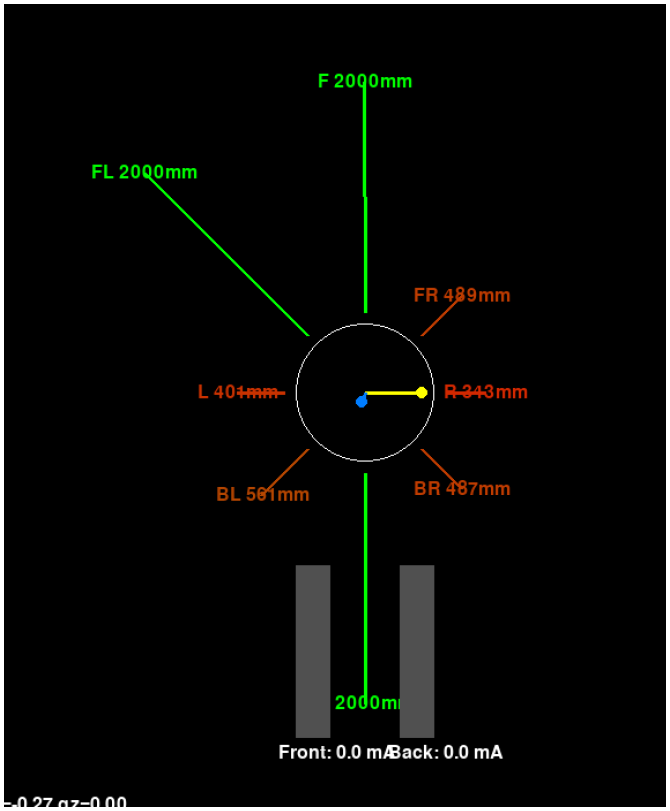


- **Telemetry logging**: Logs system and sensor data to CSV for analysis or AI training.

logs > robot_log_2025-05-08_13-15-03.csv > data

```
1  Timestamp,UnixTime,F,FL,FR,L,R,BL,BR,B,M1_Speed,M1_Dir,M2_Speed,M2_Dir,M3_Speed,M3_Dir,M4_Speed,M4_Dir
2  2025-05-08 13:15:03,1746702903.695842,1600,1600,1600,1600,1600,1600,1600,1051,30,1,30,1,30,1,30,1
3  2025-05-08 13:15:03,1746702903.9997272,1600,1600,1600,1600,1600,1600,1600,1061,30,1,30,1,30,1,30,1
4  2025-05-08 13:15:04,1746702904.303294,1600,1600,1600,1600,1600,1600,1600,1060,30,1,30,1,30,1,30,1
5  2025-05-08 13:15:04,1746702904.5074635,1600,1600,1600,1600,1600,1600,1600,1059,30,1,30,1,30,1,30,1
```

- **INA219 monitoring**: Displays live current and power usage per motor shield.
- **Live sensor visualization**: Displays real-time distance readings and IMU data using Pygame.

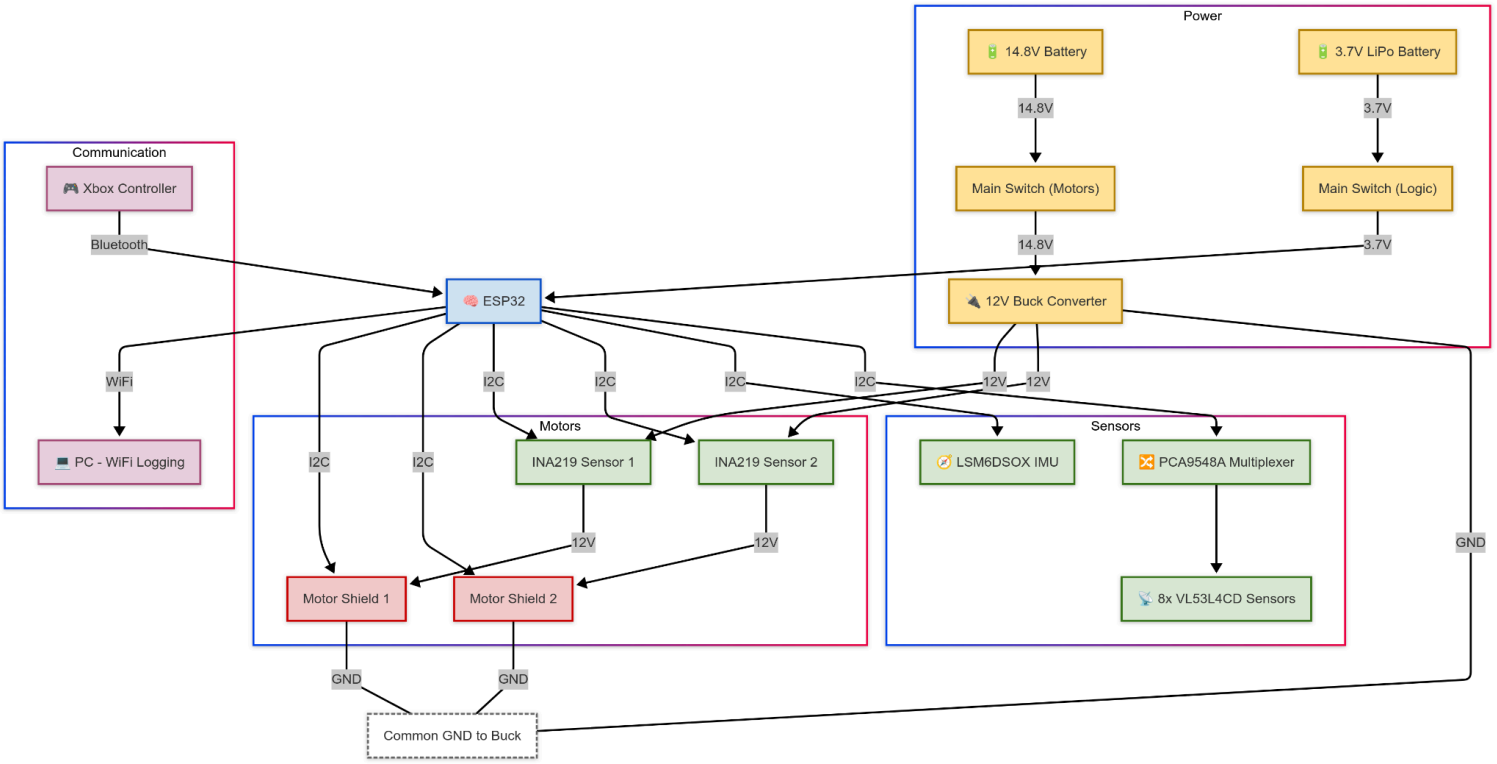


5.4 Power and Communication Architecture

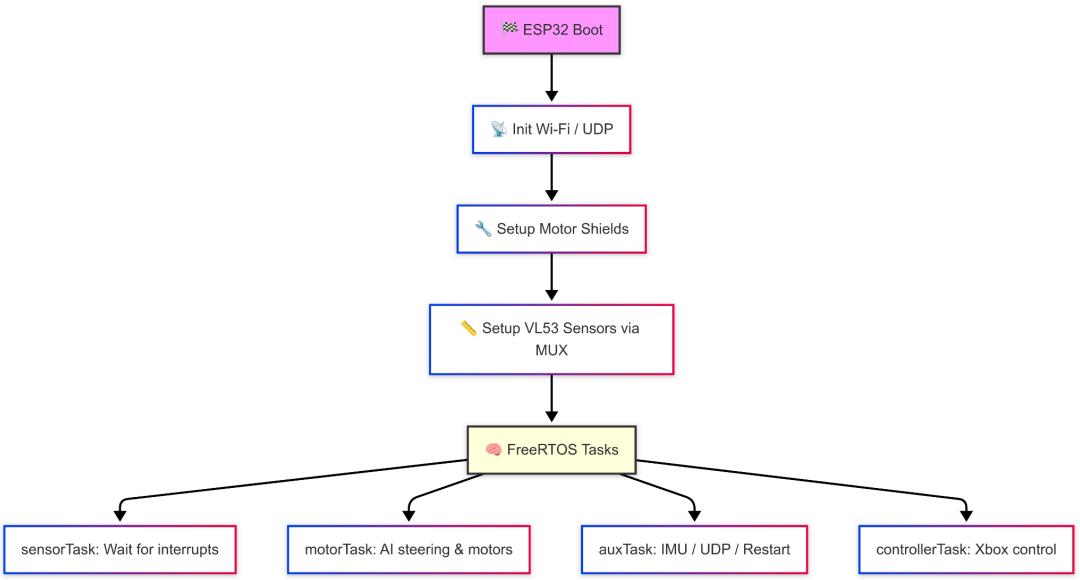
The diagram shows the OmniX robot's power and communication layout. A 14.8 V LiPo battery powers the motors, stepped down to 12 V via a buck converter, while a 3.7 V battery powers the ESP32 and logic. Both lines are controlled by separate main switches.

The ESP32 manages I²C communication with motor shields, INA219 current sensors, and a set of environmental sensors. These sensors connect to the ESP32 using STEMMA QT cables, which provide I²C (SDA, SCL), 3.3 V power, and GND in a compact 4-pin connector.

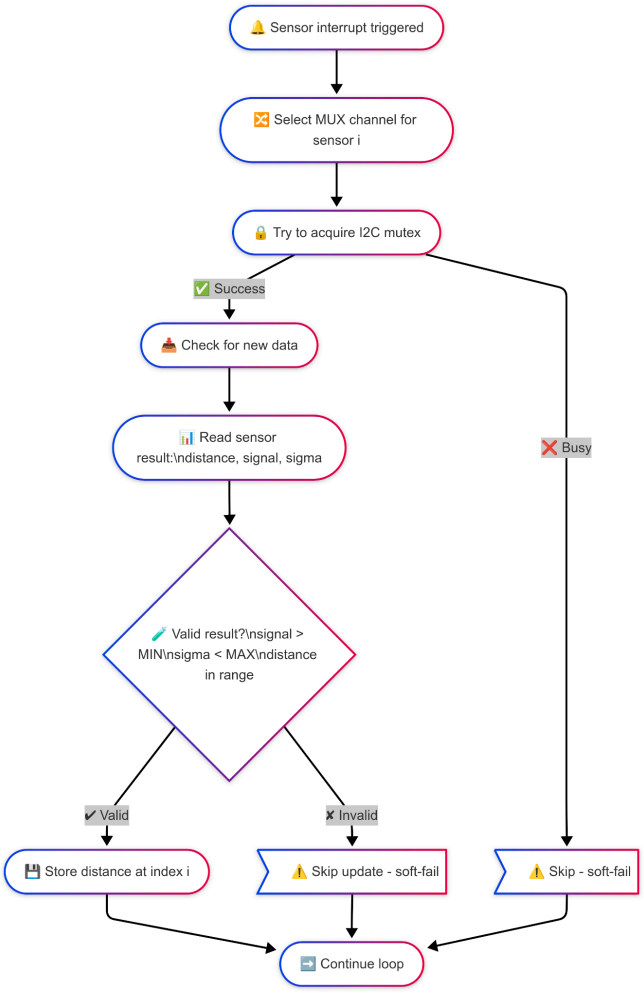
Telemetry is sent over Wi-Fi to a PC, and the robot can be controlled manually via Bluetooth and an Xbox controller. Sensor data is acquired via a PCA9548A I²C multiplexer and includes eight VL53L4CD ToF sensors and an LSM6DSOX IMU. A shared ground reference ensures electrical stability across the system.



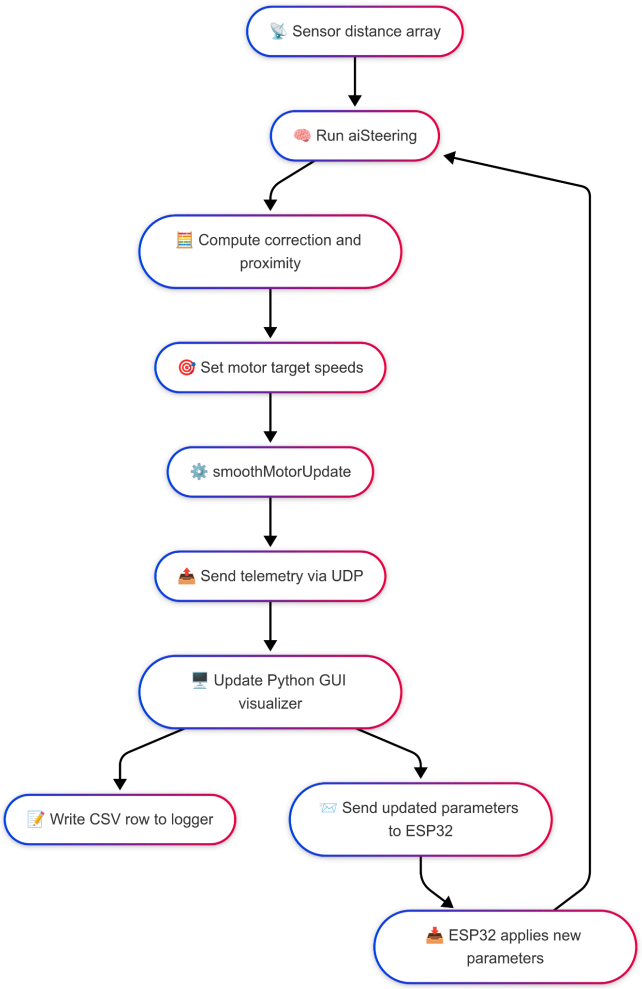
5.5 System Overview: Initialization, Sensor Processing, and Steering Control Loop



Sensor control



Steering Control



5.6 Original Machine Learning Plan

The original plan was to use AI to control the robot's steering. We wanted to run a small neural network on the ESP32-S3 using TensorFlow Lite. The model would take input from 8 VL53L4CD distance sensors and the LSM6DSOX IMU to decide how the robot should move.

To train the model, we planned to drive the robot manually using an Xbox controller. The ESP32 would send sensor and motor data over Wi-Fi to a laptop, where we would save the data to CSV files. After training, the model would be converted to TensorFlow Lite and run directly on the ESP32.

This would let the robot learn how to drive in different situations instead of just following fixed rules.

5.7 Limitations and Technical Challenges

We faced several technical issues that made it hard to finish the machine learning system. Because of that, we focused on building a rule-based system that worked more reliably.

ESP32 resource conflicts: The ESP32 couldn’t handle Bluetooth, Wi-Fi, motor control, and sensor reading at the same time. This caused delays and crashes when everything ran together.

I²C congestion: All 8 distance sensors and 2 current sensors shared the same I²C bus. This caused slowdowns. We fixed it by moving the sensors and motor shields to separate buses.

Polling delays: At first, the sensors were checked in a loop, which slowed things down. We changed to an interrupt-based system, so each sensor was only checked when new data was ready.

FreeRTOS migration: We switched to FreeRTOS to split the code into separate tasks. This helped, but it didn’t fully solve the timing problems when many tasks were running at once.

Semaphores and delays: We used semaphores to stop tasks from accessing the I²C bus at the same time. We also shortened delays where we could, but some short delays were still needed—for example, when switching channels on the I²C multiplexer.

Wi-Fi overhead: Sending data over Wi-Fi caused more delays. We considered logging data to an SD card instead, but didn’t have time to add it.

5.8 Controller Integration and Input Delay

We used the Bluepad32 library to connect an Xbox controller via Bluetooth. The setup worked well and was easy to integrate. When Wi-Fi logging was turned off and the system had fewer tasks running, the robot responded instantly and was easy to control.

However, once we enabled Wi-Fi telemetry, added motor control, and started reading all eight distance sensors, we noticed input delays. The combined load from Bluetooth input, Wi-Fi communication, sensor polling, and motor commands overwhelmed the ESP32.

This showed the limits of the hardware when handling many real-time tasks at once. It helped us understand that we needed to reduce the number of active processes or offload some tasks if we wanted faster response times.

5.9 Steering Development and Regression

Our first steering version was made early in the project when we started driving on the track. It used the front-left and front-right sensors for direction, the six side sensors for alignment and curve detection, and the front sensor to reverse when needed. This worked well and allowed stable driving at 70% speed.

Later, we ran into sensor failures caused by too many I^2C commands. We first added XSHUT control to reset sensors, then switched to using interrupt pins to only read sensors when data was ready. This improved performance and stability.

We also added live parameter tuning over Wi-Fi, which helped us adjust sensor and steering settings while the robot was on the track.

After a major rebuild, the old steering code no longer worked properly. We rewrote it, but delays and hardware problems left little time for tuning. In the end, we had to limit speed to 40%, with a temporary boost on straight paths.

5.10 Software Development Workflow

Another challenge was our limited experience with version control tools like Git. In early stages, we had no reliable way to revert to previous versions of the code. Later in the project, we learned how to use branches and commits more effectively, which helped avoid further setbacks and made the workflow smoother in the final phase.

---

5. 11 Hardware Failures

Faulty PCB
The first PCB for OmniX had several issues caused by unreliable in-house milling. The board suffered from short circuits and broken traces, most likely due to worn-out milling tools. Since OmniX was our main priority for the race, we decided to order a professionally manufactured PCB from China. The fast delivery allowed us to get back on track quickly, and the board was assembled in time for final setup and testing.

Motor Shield Failure
Although the motor shield was rated to handle up to four DC motors and 3 A of current, we damaged the first one during initial testing. While we had calculated the average current draw, we didn't take into account the short current spikes that occur during rapid changes in speed or direction. These brief surges overwhelmed the board and caused it to fail.

After this, we ordered two new motor shields and restructured the setup so that each board powered only two motors. This reduced the load on each shield and improved overall reliability. We also introduced short delays before changing direction to reduce the risk of current spikes. The damaged motor shield was later reused in Panzer, where only one functional side channel was needed

Missing Batteries
At one point, both of OmniX's main batteries and one step-down converter went missing. The converter was not a major issue as we had spares, but the battery loss caused a delay of around two weeks. This occurred early in the project when our understanding of electric vehicle systems was still developing, and every missed day impacted our schedule significantly.

5.12 Electrical Stability Issues

During testing, we noticed strange behavior from the sensors and the I²C bus. Using an oscilloscope, we found that electrical noise was affecting the system, especially when the motors were running.

The problem came from the custom PCB:

Ground path problems: The ground traces between the ESP32 and the motor shields were too thin and too close to signal lines.

No ground separation: There was no dedicated ground layer to separate the power system from the signal system. This allowed noise from the motors to spread across the board.

This noise likely caused some of the earlier sensor problems. If we had more time, we would redesign the PCB with:

- Wider ground traces between the motor shields and power supply
- A separate ground plane to protect signals
- Filtering capacitors near the motor shields

5.13 Sensor Filtering and Configuration

A lot of time was spent making the VL53L4CD sensors more reliable. They are rated for up to 1.3 m, we found that the sensors often gave random or unstable readings beyond that.

To fix this, we tested different methods:

Built-in configuration: We tried the sensor's own settings, like range timing and thresholds for signal and sigma. These helped a little, but the results were not stable at longer distances.

Software filtering: We got better results by reading the raw signal and sigma values from the sensor and filtering them in our own code. If the signal was too weak or the noise too high, we ignored the reading. This gave us more control than using the built-in filters.

Signal-and-threshold strategy: In the end, we used both signal/sigma checks and custom distance thresholds depending on the sensor's position. This made the readings much more stable and helped the robot drive more smoothly.

---

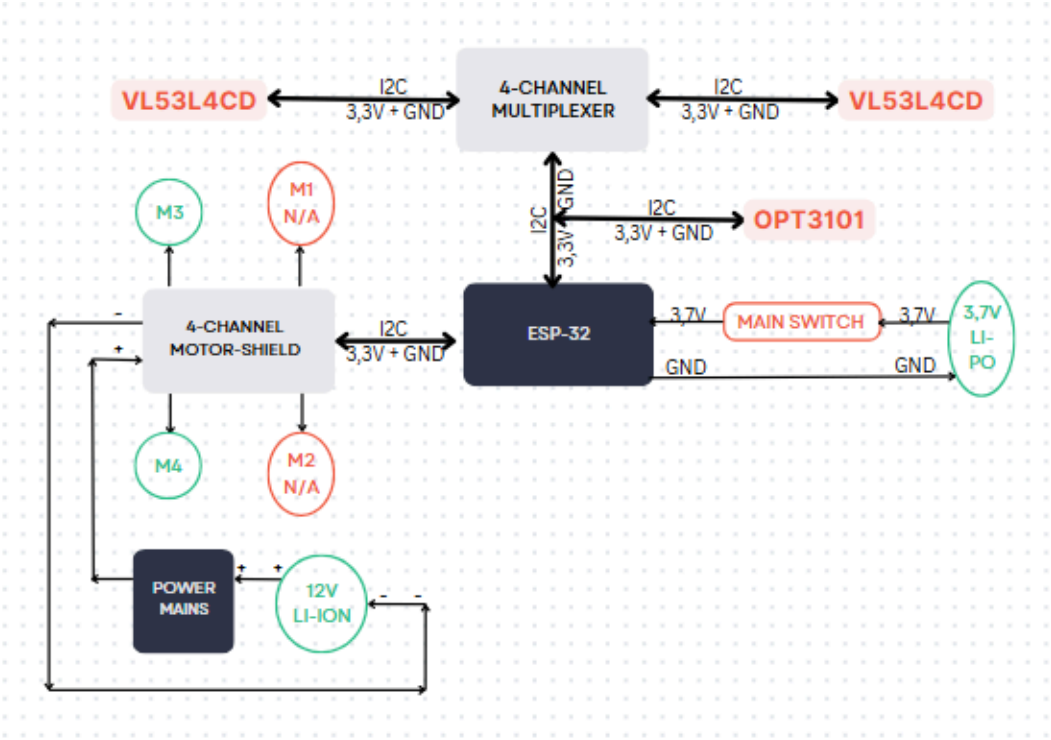
6. Panzer

6.1 Construction

We decided to start with a partial test print of the Panzer—just to check print quality and get a feel for whether the chassis was worth pursuing. It turned out okay, though some tuning of the printer settings was needed. Shortly after, the Panzer project was put on hold for several weeks in favor of OmniX.

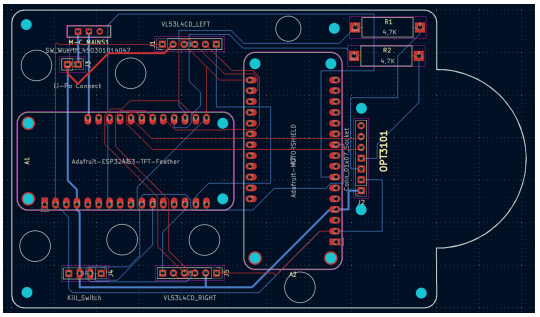
The design was chosen mainly because the chassis met the size requirements and used caterpillar tracks. Everybody loves caterpillars. Assembly was straightforward—some screws and a healthy dose of hot glue did the trick. The tracks were printed separately in a softer filament for better grip.

The chassis provided ample space for both power and logic batteries, the motor shield, the multiplexer and wiring. Logic components and sensors were mounted on the roof for better field of view and easier ESP32 access.



Connection layout for the Panzer. 2 Power switches, one for the logic units and one for the motors.

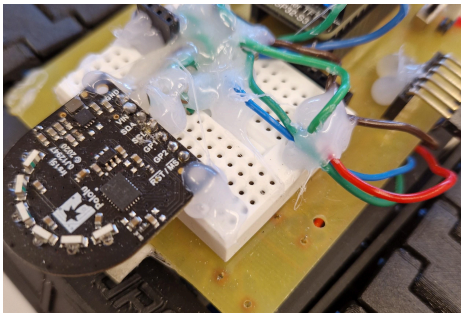
A PCB was necessary here, although simpler in design than the one used for OmniX. Unfortunately, we ran into the same milling issues—short circuits and broken traces caused by worn-out tools. However, due to its simpler layout, the board was easier to correct. We reused working power and I²C lines and bypassed the rest using STEMMA QT cables and a small breadboard glued directly onto the PCB. The full setup was mounted on the Panzer's chassis and remained stable through the project.



This was the 2 layer design had we ordered from JLCPCB.

One downside was friction. The 3D-printed design lacked bearings for the drive wheels and tracks. This led to motor strain and heat buildup during extended runs, which caused some wear on the printed parts.

A breadboard, some wires and again, a healthy amount of hot glue did the trick.



6.2 Software

The software was written in C++ and runs on the ESP32-S3 TFT. It is written in modules for an easier overview and configuration. Where each module is purposed for a certain task, or in some cases multiple, linked tasks.

Core Features

- **Autonomous Navigation:** Based on sensor data, the vehicle navigates around and avoids obstacles.
- **Multi-Sensor Integration:** two different sensor modules, work is handled by the system.
- **Obstacle Avoidance with Adaptive Logic:** Depending on the obstacle / situation, the vehicle navigates around obstacles and when unavoidable, reverse and calculate a new course.
- **Real-Time Course Correction:** When “free line of sight” the course is adjusted for minor corrections / deviations with a “boost” - function when route is clear.
- **Visual Feedback via TFT Display:** This was mainly used during development and omitted later on to decrease response time.
- **Modular and Scalable Codebase:** Written in “modules” to easier track where adjustments and fine tuning is required.
- **OTA updates**: a side project to enable wireless firmware uploads. The framework was also used to some extent to test motors and hardware functionality. This was also omitted in the final project.

In retrospect, more time on software development would have helped in the Panzer project. A lot of the information and experiences gained from the tests for the OmniX project could be applied here. Such as sensor behaviour, motor-shield issues and so on. The steering algorithm was developed separately and was an issue for further development had there been more time.

The initial idea was that as long as there were no obstacles detected(within set range), DRIVE. No course correction required. This would be done by either using interrupts or software. We went with the latter and were not coherent enough to revisit the other alternative. Actually, time was also a factor here. This worked to some extent, however Obstacle Avoidance wasn’t fast enough and the Real-Time Course Correction was sub par. These two subroutines were designed to work at different ranges, responding to different scenarios and in some cases in collaboration with each other.

File Structure

ODT.ino – Main entry point. Initializes sensors, TFT display, and I2C buses. Continuously reads sensor data, calls steering logic, and updates the display. The Kill Switch is also implemented here.

steering.cpp / steering.h – High-level navigation logic. Performs course correction or delegates to evade.cpp when obstacles are detected.

evade.cpp / evade.h – Obstacle avoidance module. Handles reversing and turning maneuvers based on proximity data.

mux.cpp / mux.h – Manages VL53L4CD time-of-flight sensors through an I2C multiplexer. Includes sensor setup and critical left and right distance checks.

opt.cpp / opt.h – Reads and filters data from the OPT3101 distance sensor (3-channel setup). Provides front, front-left, and front-right distance readings.

tft.cpp / tft.h – Manages the TFT display for visual debugging. Renders sensor data and field-of-view sketch during development.

6.3 Development Notes

Leftover Components

Nearly all components used in Panzer were leftovers from OmniX. The only unique part was the OPT3101 Sensor, which was discarded by another team. This influenced many design decisions and made the build more challenging but also more resourceful.

Sensor-Based Navigation

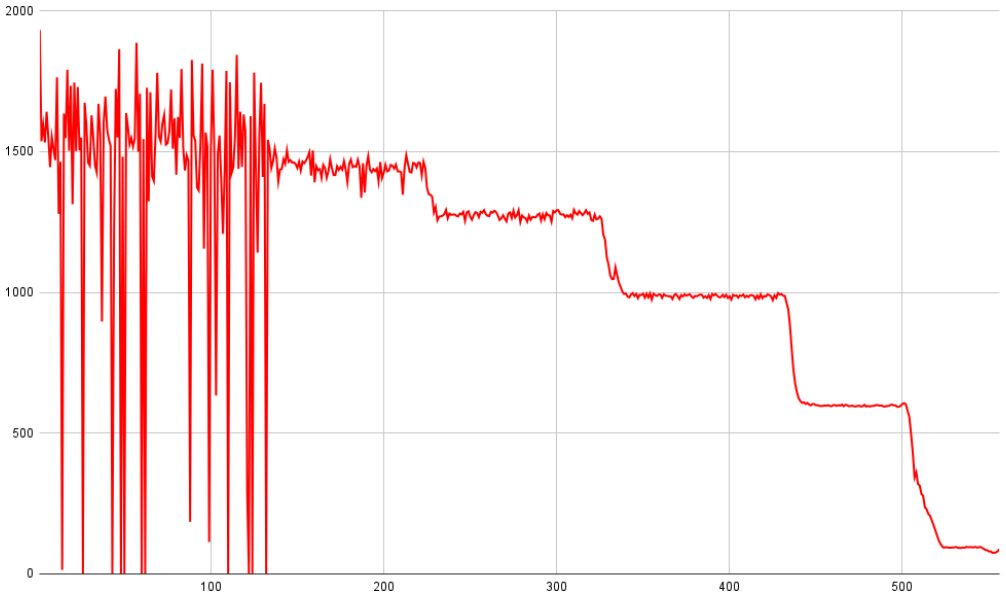
Our steering logic used distance to obstacles to decide on adjustments—smaller corrections for far obstacles, larger ones when close. A boost was used on open paths, and reverse logic kicked in when directly blocked. While this worked at times, it was not robust enough for racing, and late changes only made things worse.

---

7. Testing

7.1 Distance Reliability Test – VL53L4CD

We conducted a test to examine how the VL53L4CD sensor behaves at longer distances. The robot remained stationary while a flat target was moved closer in steps. The red line in the graph shows the raw distance values reported by the sensor over time.



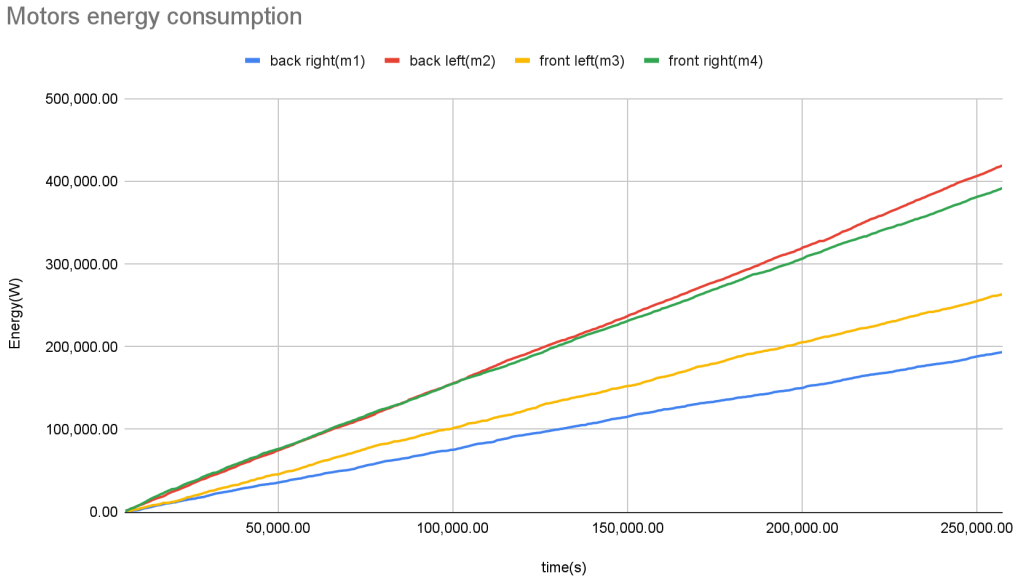
At distances beyond 1.3 meters, the sensor started to return unstable and random values, with readings dropping to zero or spiking unpredictably. This confirmed the sensor's practical reliability limit, even though it's technically rated for up to 1300 mm. The unstable data made it clear that additional filtering was needed to ensure consistent behavior.

This test supported our decision to:

- Use signal strength and sigma for software filtering
- Ignore all readings above a set threshold (e.g. 1300 mm)
- Build logic to reject low-confidence data, especially for navigation

7.2 Motor Energy Consumption Test

To measure the power usage of each motor, we ran a test where all four motors were driven with the same PWM value while the robot was powered by batteries and remained stationary. Current draw was logged using INA219 sensors to calculate energy use over time



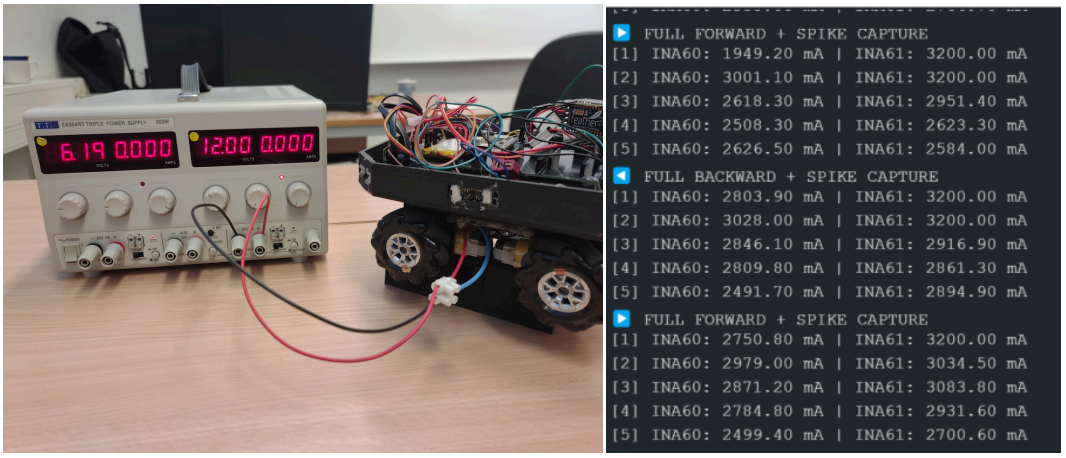
The results showed significant differences in energy consumption between motors, even though the control signals were identical. Motor 2 (back left) and Motor 4 (front right) used the most energy, while Motor 1 (back right) used the least.

This imbalance is likely due to the behavior of the motor shields. Each shield drives two motors, and we suspect they may prioritize one motor channel over the other. This could explain why motors on the same shield showed different performance, even under the same conditions.

7.3 Stress and Current Spike Testing

To evaluate OmniX's durability and current handling, we carried out multiple tests under different load conditions, both before and after the robot was rebuilt.

The first test focused on voltage endurance and took place before the rebuild. We ran the robot continuously, starting at 6 V for 10 minutes, then gradually increased the voltage in steps. At 12 V, which was the intended maximum for the system, one of the motors began to emit smoke and eventually failed. The test was stopped immediately. This showed that while the motors performed well at lower voltages, extended use at maximum voltage exceeded their thermal and electrical limits.



Picture show the stress test using a power supply and the output from the spike test

After rebuilding the robot with two motor shields and INA219 current sensors, we performed a second test to examine current spikes during sudden movement. In this test, we ramped the motors from zero to full speed and then reversed direction. Current measurements were logged from both motor shields. One shield (INA61) reached the INA219's maximum measurable current of 3200 mA, while the other (INA60) also recorded sharp spikes. These results confirmed that running all four motors on a single shield could lead to unsafe current levels, and validated our decision to split the load between two shields.

Together, these tests highlighted the importance of:

- Distributing motor load across two shields
- Avoiding abrupt direction changes without short delays
- Monitoring current to protect the system and prevent hardware failure

8. Discussion

This project taught us a lot about designing and building an autonomous robot from scratch. From the start, our goal was to create something unique. We planned to use machine learning for steering, and much of the system was designed around that idea. If our only goal had been to win the race, we could have chosen a simpler and more proven design. But we wanted to explore new concepts and challenge ourselves technically.

Eventually, we had to move away from machine learning and instead build a rule-based steering system. The ESP32 could not handle Bluetooth input, Wi-Fi logging, sensor reading, and motor control all at once without problems. By simplifying the logic, we were able to create a more stable and reliable system that we could tune and improve during development.

We also found that real hardware can behave differently from what is written in the datasheet. The motor shields were supposed to handle 3 amps, but quick direction changes caused power spikes that made the shields shut down. By adding a second motor shield and splitting the load, we avoided this issue and improved performance.

Our distance sensors also had their limits. Although rated for up to 1.3 meters, they gave unstable readings beyond that range. This was true even on clean white walls, which should have been ideal. To make the system more reliable, we added filtering based on signal strength and noise level. For future projects, we may consider using sensors with longer range or better consistency.

We also learned the value of good PCB design. Our first boards had very thin ground lines and no separation between power and signal paths. This caused electrical noise that made sensor readings less stable. A better board layout would have saved us time and made the system easier to work with.

Even though the project was complex, our modular design and debugging tools helped a lot. FreeRTOS allowed us to divide the software into tasks so we could find and fix issues more easily. The Python GUI made it simple to monitor live data and adjust settings while the robot was running.

In the end, we did not choose the simplest path, but we learned a great deal. That experience will help us in future builds, especially when we want to focus on performance or competition.

9. Conclusion

This project resulted in two working autonomous vehicles with modular hardware and configurable software. Although machine learning was not fully implemented, the rule-based system proved reliable after extensive testing and tuning. The project helped us gain practical experience in real-time systems, embedded development, and hardware integration. We also learned valuable lessons about debugging, current management, and sensor filtering.

---

10. Source code

Link to entire project on Github

11. References

[1] P1r.se, *Kill switch documentation: Kill Switch Relay – Robot Start Module.* [Online]. Available: https://p1r.se/robots/start-module. [Accessed: May 23, 2025].

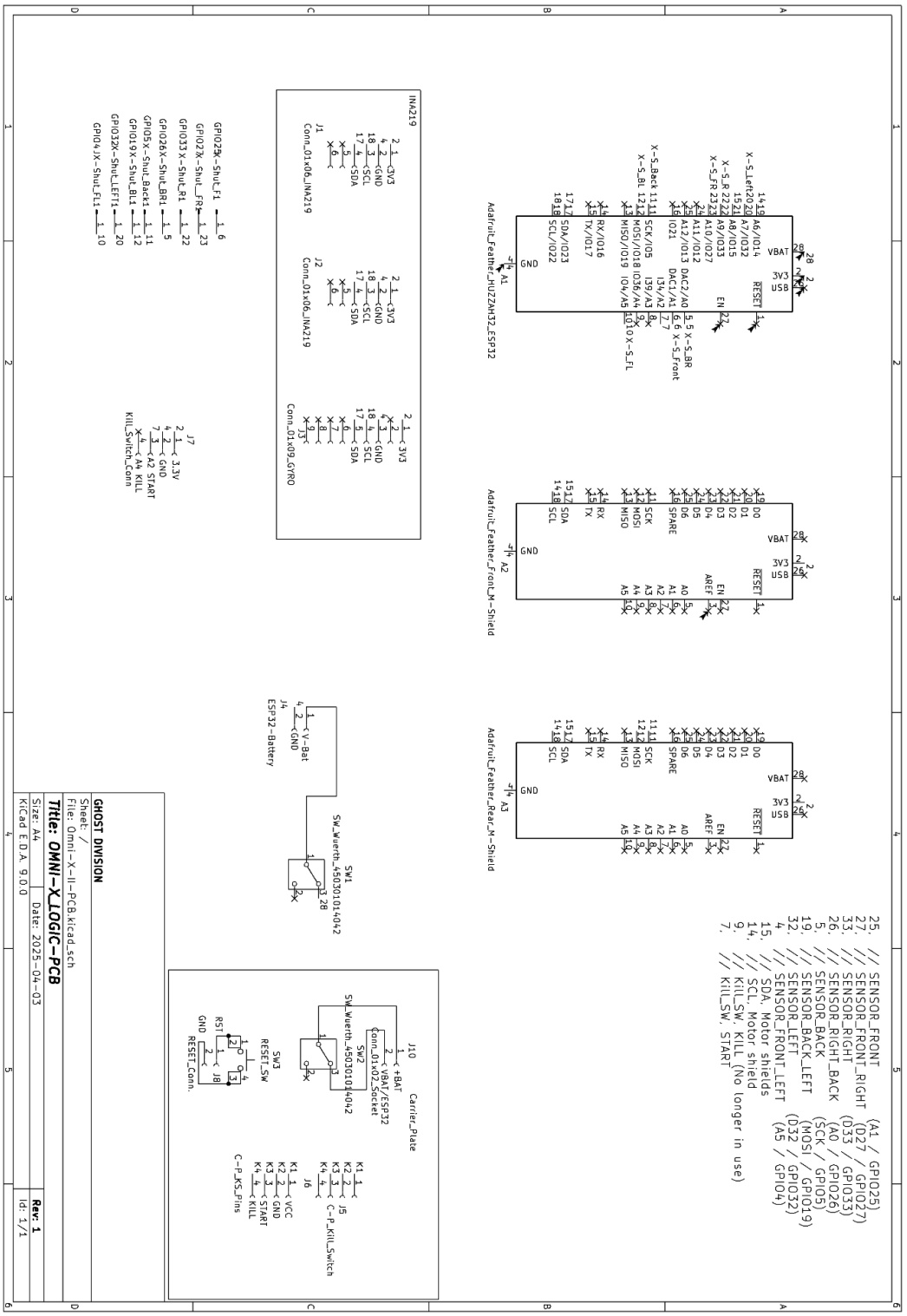
[2] Yrgo, *Project Guidelines and Regulations: Klassarbete för Elektronikprojekt Ex24.* [Online]. Available: Studieplan Elektronikprojekt - Google Docs[Accessed: May 23, 2025].

[3] Yrgo, *Rules Addendum: Tillägg regler - Google Dokument.* [Online]. Available: Tillägg regler - Google Docs[Accessed: May 23, 2025].

[4] STMicroelectronics, *VL53L1X Datasheet – Long Distance Ranging Time-of-Flight Sensor.* [Online]. Available: https://www.st.com/resource/en/datasheet/vl53l1x.pdf. [Accessed: May 23, 2025].

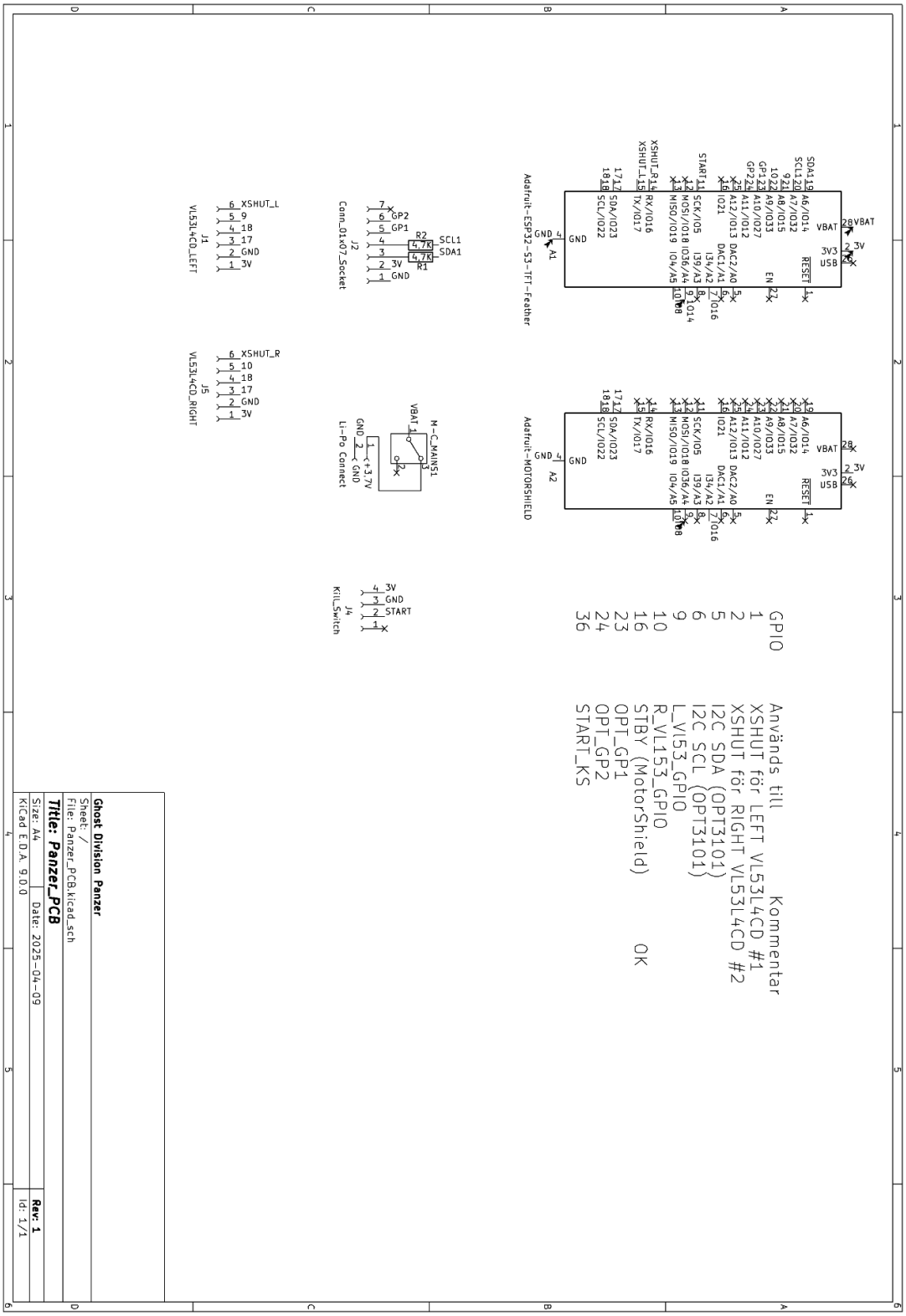
Note: Additional Datasheets and references to components are available in the Bill of Materials and Budget List.

12. Appendix



A.1.

OmniX PCB Schematic



A.2.

Panzer PCB Schematic.