CTRL+ALT+ DEFEAT

Welcome to the repository for our entry in the WRO Future Engineers 2025 competition! We are a team of 2 aspiring engineers building an autonomous robotic car using Raspberry Pi 5 & RP2040, equipped with various sensors and custom designs.

About the Team

Adbhut Patil: Adbhut is studying in 10th grade, under NIOS. He is interested in robotics, programming, and aviation.

Pranav Kiran Rajarathna: Pranav is currently studying in the 11th grade(PCMC combination) and has been interested in Robotics for several years . He has participated in WRO in other categories in the past years. He was also part of Coding club in his school and built a few robotics projects for school events. His other interests include maths, physics, numismatics and history.

Team Photo

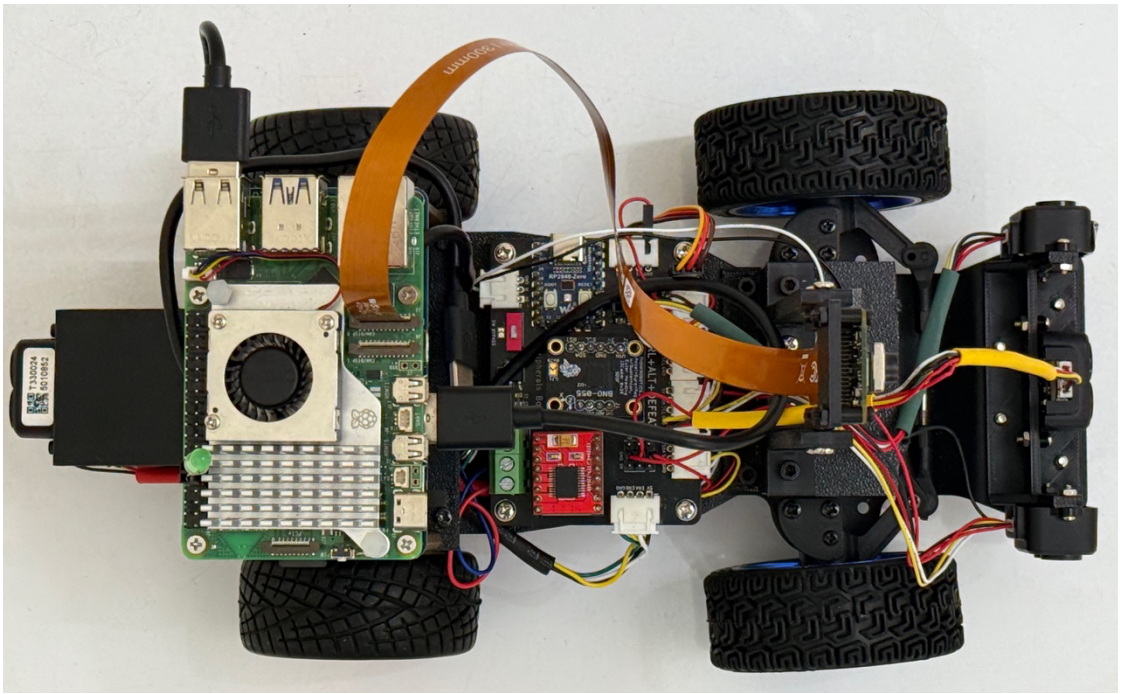
Picture: Pranav (Left) and Adbhut (Right)

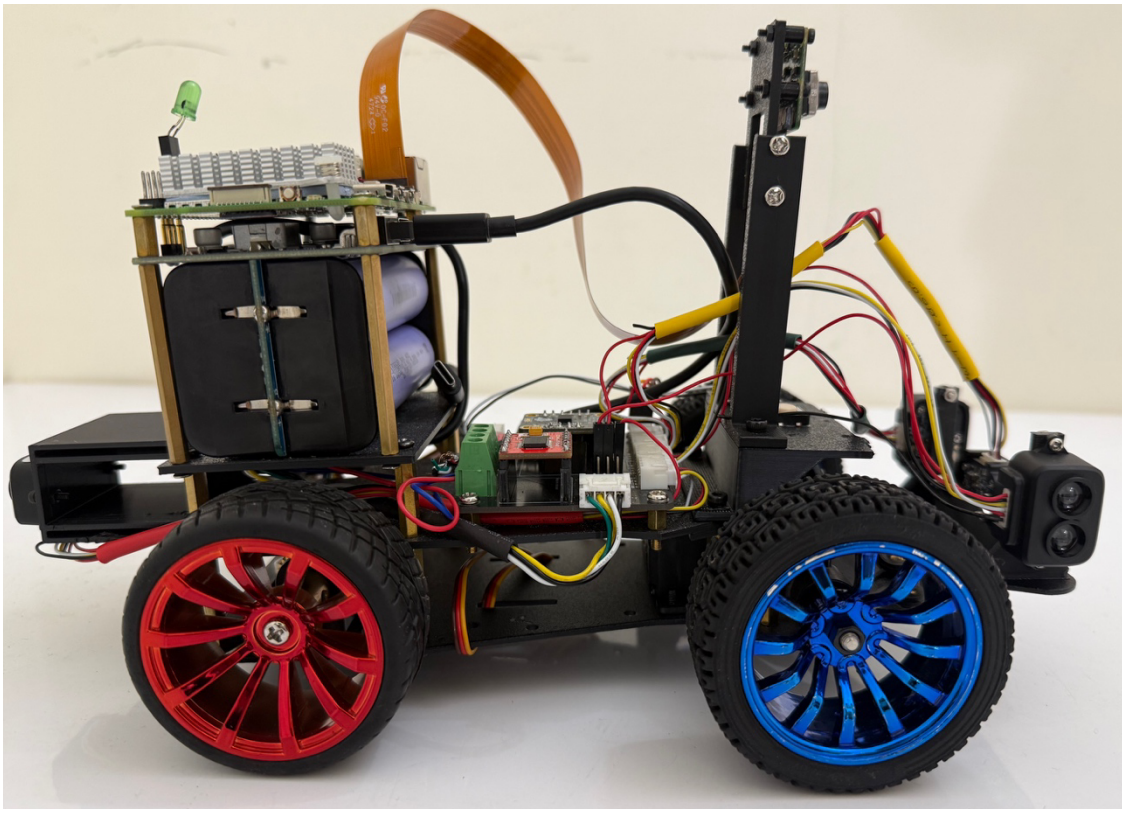
Project Overview

This project is our official entry for the Future Engineers category at the World Robot Olympiad 2025. Our goal is to construct a self-driving vehicle capable of navigating the track in both the Open and Obstacle Rounds. The programs are written in C++ and Python in the VSCode IDE, with the PlatformIO extension. OpenCV is used with the Raspberry Pi for object detection and navigation. This repository contains the programs, hardware description, schematics and design files of our solution.

Note: The whole readme file is massive. See this section for more details about the peripherals interface board, and this section for details about the RPi 5 and its setup. System architecture includes information about the robot and mobility, power system, sensors, obstacle management and parking and unparking.

Some photos of the robot -





All photos may be found here

Hardware Components

- **Compute:** Raspberry Pi 5 (main computer), Raspberry Pi 2040 (Waveshare RP2040-Zero, real-time control)
- **Sensors:** 1D LiDAR, IMU, rotary encoders, Picamera
- **Actuators:** N20 geared DC motor with encoder, MG996R 180° servo
- **Chassis:** Commercially available base with 3D-printable additions (3D-design files included)
- **Electronics:** Custom peripherals interface PCB for reliable connections to sensors

Repository Structure

- design-files : Contains the Bill of Materials, 3D-printable design files and hardware.
- initial-tests : Intial tests on various sensors to ensure accurate data collection and basic object detection algorithm.
- hw : Contains schematic and PCB files for the peripherals interface board.
- sw : Contains PlatformIO project for the peripherals interface board.

- open-round : It contains the program files for the open round over various iterations and hardware setups. The final open round program is found in `sw/peripherals-board/src/divers`
- repo-assets : It contains pictures and description of the robot and its components. All other photos in the repo are also found here.
- obstacle-round : Programs for the obstacle round. These are categorised in order of working alphabetically. The related videos and log files are also present in these folders.
- test-data-recordings/open-round : It contains data logs from various open round tests

Note

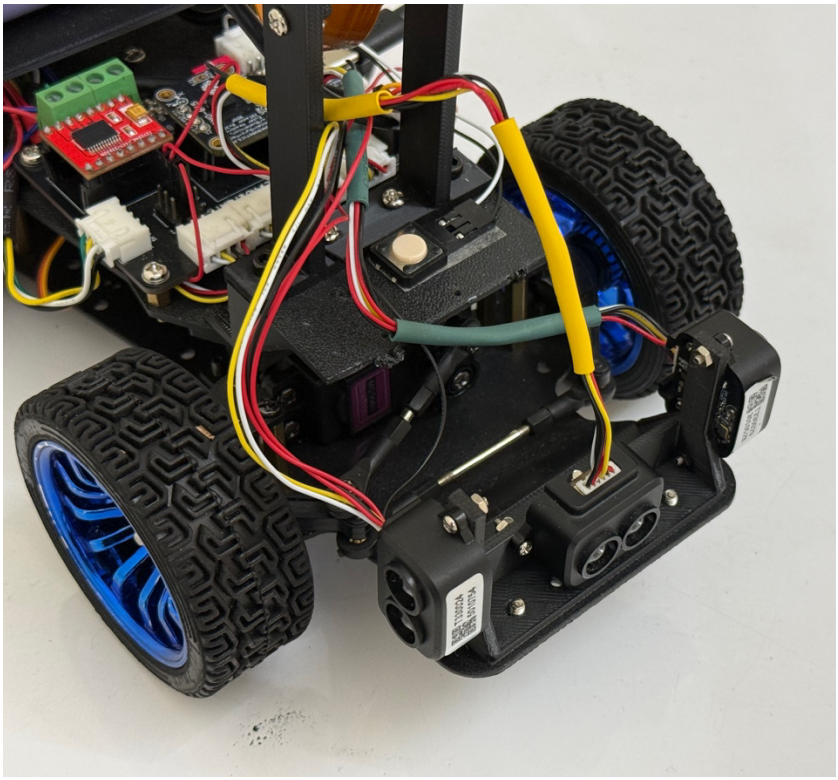
Random files called `.lgd-nfy0` may be seen sometimes in varioues places. These were probably created by a typo a long time ago, and despite repeated deletion, they keep spawning everywhere in the Pi's cone of the repo! Just ignore them, if they occur. `.DS_Store` are some system files used on MacOS. These too may appear throughout the repo, and may be ignored.

System Architecture

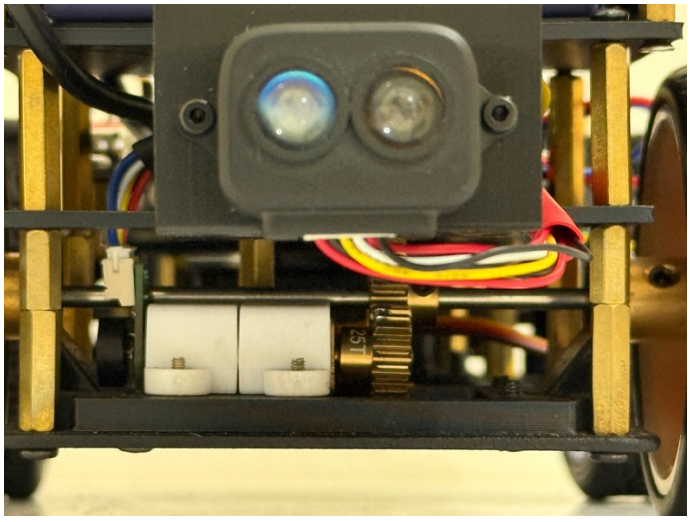
- Raspberry Pi 5: It uses ROS2 and handles more complex tasks like object and colour detection, route planning, and the 2D LiDAR
- RP2040 (Pi Pico): Handles real-time tasks (motor PID, encoder feedback, servo control, IMU, 1D LiDAR)
- LiDARs: Used for detecting turn direction and executing turns and in parking
- IMU: Gives yaw (and other orientations) of the robot in [0,360) clockwise
- Motor/Servos: PWM and PI controls respectively for movement using sensor data, coordinated by the RP2040

Mobility

- A commercially available metal chassis base has been used with custom designed and printed parts. This chassis was selected mainly for its good steering system, mobility and customisabilty (there are lots of through holes through the base-frame). While the steering was retained, the remaining parts were custom designed.
- A link based steering system is used in the robot with a link-rod between the front wheel joints and another between a wheel joint and the servo. This steering system has been taken from the chassis without much modification due to its precision. A MG996R 180° servo has been used for the steering due to its high torque and accuracy, which improve mobility. The turn radius is around 32-33cm.



- A single axle rear drive train is used for driving is used. A 200RPM N20 motor is used due to its speed and decent torque. A 1:1 gear ratio is used with a pair of brass gears. Bearings are used on the rear axle for smooth movement. A custom rear drive holder is used to house all these parts.
- Both the servo and N20 motors are connected to the peripherals board. The servo is controlled using the standard library, while PWM is used for speed control of the motor.



- This robot offers realistic car-like dynamics, ideal for FE challenge simulation. Design files contains all the other custom 3D-Design files (created on TinkerCAD) for mounting various compnents and systems onto the base chassis and one another. Robot assembly instructions are below.

- Improvements:

1. A uniform steering system. In the current model, the servo has to turn a greater angle left than right for the same angles of the wheels in each direction. This complicates driving algorithms to some extent. A better, but equally smooth steering system, maybe a rack-and-pinion, will solve these issues.
2. Better gear train. The current gears, though made of brass wear down quite easily for some unidentified reason. This causes slipping if there is more wear.
3. Reducing the turning radius. The current bot, while quite manoeuvrable has a large turn radius (almost 33-34cm). Reducing this would make taking tighter turns possible, improving error margins in the obstacle round. This can be done by bringing the drive train forward.

Power

- A Waveshare UPS Hat E is used for powering the robot. It uses 4 4000mAH 3.6V 21700 Li-ion batteries. (Different battery capacities may be used). All outputs are at 5V.
- The Pi is powered at a stable 5V via the POGO pins on the UPS Hat. The PiCamera and RPLidar are power from the Pi.
- The RP2040 and all connected sensors are powered through a USB A-C cable from the Pi to it.
- The total current draw for the obstacle round is around 600-630mA, and around 250-300mA for idle with the Pi.
- The peripherals board gets power directly from the UPS Hat through a USB-A cable that has been cut to get the power wires. This powers only the rear motor and steering servo.
- Improvements:

1. The UPS Hat that we are using has a weird anomaly. Although the batteries will have good voltage output and detected capacity, it will show the net remaining capacity to be very low. We have edited the code of the UPS library to not allow it run `sudo poweroff` when it misdetects such a situation. Additiionally, charging of batteries when the Pi is off does not seem to be detected.

Sensors

- **TFLuna LiDAR:** These are 1D-LiDARs that are used to measure distances to the front, left, right (for driving) and back(parking) of the robot. They are connected to the RP2040. A separate section has been designed at the front of the robot to house all four LiDARs. Each LiDAR has 5 wires, grouped into 4+1 (I2C + GND) going into the

peripherals board. These LiDARs were selected for their accuracy, small size and ease of use.

- **BNO055:** This is the IMU we are using for orientation due to its reliability, accuracy and simplicity of use. It is connected to the RP2040, and is directly slotted into the peripherals board.
- **nRF24L01:** This is the wireless module used for wireless communication during testing and debugging with the RP2040. Another custom module (telemetry board) is made for receiving the data. (Note : This is not plugged in for actual rounds). It is directly slotted into the peripherals board.
- **Motor encoders:** The encoders give ticks each time they are trigerred by rotation of the motor shaft. Data from it is used for distance calculations. In our case, about 43 ticks corresponds to 1cm. The 4 encoder wires are grouped together and connected to the peripherals board.
- **Raspberry Pi Camera Module 3 Wide:** It is used with the Pi for detecting the an obstacles. The camera is secured at four points for greater stability. With only two connection points, the camera could be easily turned due to the material properties of the Camera to Pi 5 connection wire. The higher mounting location allows it to detect obstacles from a large distance (almost 2m!) and plan accordingly. The Wide lens allows for a greater field of view. The NoIR version used to give odd red patches due to the lack of the IR filter in some test cases.
- Improvements:

1. The 1D LiDARs are quite expensive. Ultrasonic sensors would have done for the sides.
2. The BNO055 does clock stretching, which causes issues on the I2C buses of most microcontrollers (thankfully not the RP2040). It also has a slight drifting tendency of a few degrees after a round. A IMU that does not have these issues and is easy to use will be more accurate and more flexible in use.

The Bill of Materials contains all necessary parts/components and their sources.

Obstacle Management

- This is done mainly using the camera feed from the PiCamera 3 Wide. The NoIR version tends to give red spot in places with lots of light due to lack of IR filter, which made it incompatible for this case.
- The obstacle management sections were not able to be integrated with one another in time, but the various sections do work independently.
- The OpenCV library (in Python3) is used for all the image manipulation and data extrapolation. HSV mode has been used for all the detecting, while images arer displayed in RGB. Sometimes, red may appear as blue due to use of BGR format used by camera to record. This is of no consequence, since it does not affect detection itself.

- Initially, the colour ranges for the obstacles, parking walls and side walls are identified. Interestingly, in our case, the 'magenta' parking walls show up as pure red.
- First, the region of interest (ROI) is taken as the bottom half of the feed (this is for our camera placement. May vary for other setups).
- Then, coloured masks (one each for red and green) are applied, followed by OpenCV's controur detection to detect surfaces. Data like position, height and width of the contours may be obtained from the other OpenCV methods.
- If the contours detected meet the required criteria (like size, height-width ratio), they are classified as obstacles. For our benefit, rectangles are drawn around the obstacles.
- The basic idea is to treat both possible positions in a row of the 2x3 grid as the same (calculations use the actual values). This greatly simplifies tthe program, since only the outer and inner lanes are used for the driving, while the middle lane is used only for switching between the other two.
- Using the data available, including obstacle position, yaw and distance travelled, the required steering is calculated to pass the obstacle on the correct side, when not at a turn (Red - right, green - left) on a linear scale
- For turns:

1. Starting closer to the inner wall: The seen obstacle before the turn is considered. If turn must to taken sticking to the inner wall, it is immediately taken. If the turn must be taken to get closer to the outer wall, a curve(like a question-mark) is followed. If there is another obstacle of the opposite colour, then the robot the goes back some distance to be able to do the manoeuvre.
2. Starting closer to the outer wall: A turn is taken, making it align with the middle lane. Following this, the robot manoeuvres based on the obstacle/s present.
3. In case of the parking wall being there on every 4th turn, the turning and linear coefficients are changed to get the robot to turn to the adjusted lanes and obstacle position.

Improvements:

1. The RPLidar might have been used effectively with better drivers with the standard Raspbian OS.
2. A smaller robot would have been more agile, making both turning and straight section navigation much easier.
3. Having the axles closer would enable for tighter turns.

Getting out of the parking lot

Getting out of the parking lot (fondly referred to as "unparking") is handled by the peripherals interface board. The vehicle takes three turns to orient itself so it is perpendicular to the parking lot, and then backs up into the parking lot. It then transfers control to the Raspberry Pi.

Parallel parking

Parking is also handled by the peripherals interface board. The Raspberry Pi hands over control after the last obstacle. The algorithm takes the following steps:

- Move backwards until the vehicle is abeam the parking lot
- Turn and move backwards and move into the parking lot
- Move forwards so that the next turn can be made
- Turn and move backwards into the parking lot
- Move forwards and turn until the vehicle is parallel to the parking lot

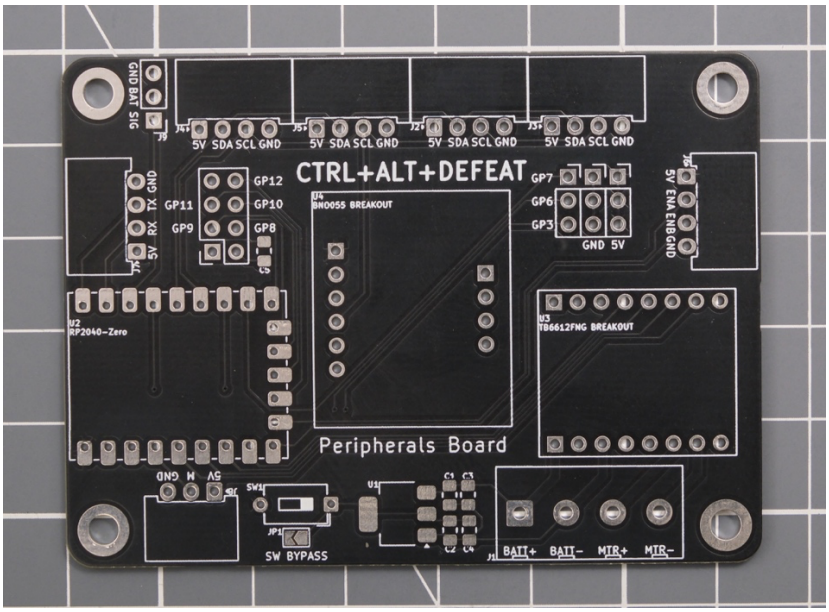
Robot Assembly

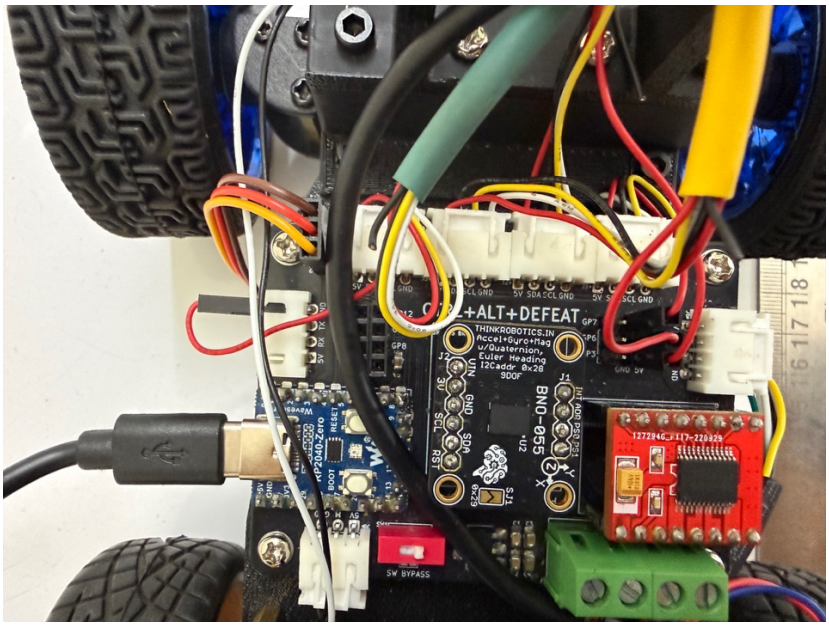
It will be helpful to refer to the pictures of the construction while building. Most screws used are of 3M type (3mm dia.). At some places in our robot, internal hex screws are used in places where plus(+) screws of the right length were not available.

1. Add the front wheels and steering link to the robot
2. Attach the servo motor and connect it to the steering link
3. Print all design files
4. Attach the N20 motor onto the rear drive with brackets. Fit a gear onto the motor.
5. Cut the rear axle to the appropriate length (around 96mm) and attach the rear axle with wheels and gear onto the rear drive.
6. Mount the rear drive onto the chassis.
7. Attach 6 pillar screws of length 26mm in the back and 2 of length 22mm in the front (adjacent to the servo)
8. Attach the middle plate at the 6 rear points, but not at the front. The peripherals board may be screwed on using 4 pillar screws.
9. Attach the TFLuna LiDARs to the front LiDAR holder section.
10. Attach the LiDAR holder section to the front of the chassis.

11. Attach the additional support section (this used to be used when the RPLidar was also attached to the model) to the 2 front middle plate points and 2 rear RPLidar points.
12. Attach the two PiCamera holder parts to the rear of the support section, and screw in the camera (2M screws and nuts are needed)
13. Attach the top-most Raspberry Pi and UPS platform to the rear with screw pillars.
14. Attach the rear lidar holder with the rear lidar attached to it
15. Attach the Raspberry Pi with the UPS Hat to the top.
16. Make all wired connections by referring to schematic files.

Peripherals Interface Board





The peripherals interface board goes between the Raspberry Pi and vehicle hardware. The board talks to three 1d TFLuna LiDARs, a 9-axis BNO055 IMU, the motor encoder, the drive motor, and the steering servo. It talks to the Pi over USB.

Hardware

The board is a two-layer, 1.6 mm thick PCB designed in KiCad and manufactured by Robu.
TODO: Expand

Software

The interface board was programmed in C++, with the C++ SDK from Raspberry Pi for the RP2040 chip. Code was compiled and flashed with the PlatformIO extension for Visual Studio Code.

The codebase is designed to be modular, with easily swappable drivers for hardware abstraction. Algorithms do not need to know about specific hardware protocols or constraints

`main.cpp` loads in drivers, which implement virtual interface classes. `main.cpp` then initialises everything, and in a loop, gets data from the `SensoorManager`, passes that data to our drive algorithm, and outputs it to the `TargetController`, which commands the motor and steering servo. Drivers are in `/src/drivers`, Interfaces are in `/src/interfaces/`, and utilities are in `/src/utils/`.

Code Development Standards

To ensure easily testable, upgradeable, and clean code, the following standards were decided for writing code for the interface board.

Note

Due to time constraints, not all of these could be adhered to during final development

- `main.cpp` must be as clean as possible.
- Logic and code flow must be easy to understand.
- All hardware-specific things should be done inside a driver.
- Drivers must take in a standard unit as their input.
- Drivers must not take in magic numbers.

Drivers

Note

Hardware revision 1 drivers are incomplete, as the peripherals baord codebase was partly written when the switch from the prototype interface and the PCB occured, and drivers from that point on were only for the PCB (hardware revision 2).
A list of implemented drivers:

- `hwrev2_imu` : Implements `ISensor`. Driver for the BNO055. Gets orientation data processed by the BNO055's internal microcontroller.
- `hwrev2_lidar` : Implements `ISensor`. Driver for the TFLuna LiDARs. Returns a [0, 259] of distance data. The three LiDARs are mapped to `0`, `90`, and `270`. (This is a remnant of the original plan of connecting the RPLidar to the peripherals board, instead of to the Pi.)
- `hwrev2_motor_driver` : Implements `IMotorDriver`. Driver for the TB6612FNG dual-channel H-bridge motor driver. The driver only supports driving a single channel on the T66612FNG.
- `hwrev2_rf24_communication` : Implements `ICommunication`. Driver for the nRF24L01+ radio used during development. Transmits vehicle data and gets commands from another nRF24L01+, connected to a computer.
- `hwrev2_single_lidar_open_round` : Implements `IDriveAlgorithm`. A port of our open round test algorithm. The name is misleading, the algorithm uses all three TFLunas.
- `hwrev2_steering_driver` : Implements `ISteeringDriver`. Driver for the MG996 steering servo. Takes in a degree input and coverts it to a 0-180 command with a conversion constant.
- `hwrev2_target_control` : Implements `ITargetControl`. Takes in a `VehicleCommand`, and either passes it directly through to the `ISteeringDriver` and `IMotorDriver`, or passes the command through a PID controller first.
- `hwrev2_uart_logger` : Implements `ILogger`, Takes in a sender, message type, and a message string
- `hwrev2_vehicle_speed` : Implements `ISensor`. Gets vehicle distance from the motor encoder. Calculates speed using time between pulses. TODO: unparking and parking drivers

Interfaces

Interfaces are virtual classes which define certain functions. Drivers implement these classes. Code calls functions defined in the virtual interface class.

- `IDriveAlgorithm` : Takes in a `VehicleData` object, and returns a `VehicleCommand` object.

- `ILogger` : Constructs a message and prints it out on a UART bus. Takes in a sender string, a log type (information, warning, error) and the message.
- `IMotorDriver` : Takes in a speed value and outputs to a motor driver.
- `ISensor` : A generic sensor object. Returns a vector of `SensorData` objects.
- `ISteeringDriver` : Takes in a wheel steering angle, outputs to a steering mechanism.
- `ITargetControl`: Takes in a `VehicleCommand`, outputs to a motor and steering driver.

Managers

Managers are classes that handle certain aspects of the vehicle. Certain structs have their header files in the `/src/managers` folder, even though they are not classes.

- `VehicleConfig` : A struct containing information about the vehicle.
  A `VehicleConfig` struct is passed to all drivers asnd managers.
- `SensorData` : A struct containing fields for all data that can be collected by the vehicle, and an enum which tells the caller what fields are being used.
- `SensorManager` : A class which takes in an arbitrary number of sensors, collects data from them, and processes them into a `VehicleData` struct.
- `SensorStatus` : A (currently unused) enumeration which defines keys for states a sensor can be in.
- `Vec3f` : A struct of three double-precisionn floating point numbers, used to represent three-axis values.
- `VehicleCommand` : A struct defining a target speed and yaw value for the vehicle. Returned by a drive algorithm, passed to a `ITargetController`.
- `VehicleData` : A struct containing processed vehicle data. Returned by a `SensorManager`.

Utilities

- `SchedulerTask` : A class which calls a function periodically.
- `Schedular` : A class which takes in an arbitrary number of `SchedulerTask`s, and updates them.

Raspberry Pi System

The Raspberry Pi system handles most tasks for the obstacle round, including unparking, obstacle detection and navigation. TODO: it doesn't handle unparking, update this section

Raspbian Setup

We are using Raspbian Bookworm on the Raspberry Pi (latest at the time of this project)
Run the following commands in the terminal

- $`sudo chmod 0700 /run/user/1000`
- $`sudo apt install software-properties-common`
- $`sudo apt install python3-launchpadlib`
- $`sudo apt install code`
- $`sudo apt upgrade code`
- $`sudo apt install idle3`
- $`sudo apt install python3-opencv`
- $`sudo apt install -y python3-libcamera python3-pyqt5 python3-picamera2`
- Test camera library installation with $`rpicam-hello`
- Refer here to install UPS Hat library to see details about the batteries etc.
- If the Pi is accidentally shut down, and git is not working, run the following commands:

1. $`find .git/objects/ -type f -empty | xargs rm` - Cleaning
2. $`git fetch -p` - Restore missing objects
3. $`git fsck --full` - Verify

- It is advised to have RPiConnect for easy connection to the Pi.
- This repositttory is cloned on the Pi and the obstacle round program is added to the startup sequence.

Adding Programs to Startup

The method used here involves running executables on startup. Refer `initial-tests/startup-test/` for the code.

1. Create the python file
2. Create a launcher script, like the one in the initial test file, that ends in `.sh`. Ensure the launcher is working with `./(launcher-name).sh` in its directory.
3. In the launcher script, navigate to the directory of the python file and run it with `sudo python3 (file-name).py`. Then return to user directory
4. In the directory with launcher script, make it executable by running `sudo chmod 755 (launcher-name).sh`
5. Run `sudo crontab -e` (you can use nano to edit it), and add `@reboot sh /home/(user)/(path to launcher)/(launcher-name).sh` at the end.
6. Restart the Pi (`sudo reboot`) to check if it works.

PiCamera 3

The Raspberry Pi Camera Module 3 Wide is used for obstacle detection. It is placed high up to have a good view. (We actually only use the bottom half of the frame!) It is connected to port 0.

Serial Communication

The Pi communicates with the RP2040 over the serial. When the Pi sends a command (speed and steering), the RP2040 sends back data (like yaw, lidar distance, turn direction etc.) The RP2040 is connected to a USB port of the Raspberry Pi and is recognised by its ID.

Open Round

Refer to `sw/peripherals-board/src/drivers/hwrev2/hwrev2_single_lidar_open_round.cpp and .hpp` files for program. The idea to stick to the outer wall, so even with randomisation, the extended inner wall is not hit.

Videos of the open round may be found here

- Uses front LiDAR to detect when to start a turn (if obstacle is close).
- Decides turn direction (left/right) based on side LiDARs (First turn only)
- Turns are started considering factors of distance available, yaw, current section and distance travelled.
- Adjusts steering (servo position) during turn based on yaw and for greater smoothness.
- Target yaw and threshold distance may be altered for better accuracy and counter drifting.
- When not turning, PI controller is used with yaw to follow a straight path accurately.
- Counts turns; after 12 turns (3 rounds), slows down and stops at the start section.

Flow

1. Start moving straight.
2. Decide turn direction using side LiDARs. (For first turn only)
3. Detect distance to wall ahead with front LiDAR.
4. Execute turn when condition are met, adjust steering and target yaw.
5. After turn, resume straight movement with PI control.
6. Repeat for 3 rounds (12 turns).
7. After final round, slow down and stop in the start section.

Acknowledgements

We would like to thank

- YoLabs Team: We thank the YoLabs Team for their guidance and mentorship provided, which helped us improve algorithms and robot design.
- Raspberry Pi Foundation: Gratitude is owed to the Raspberry Pi Foundation for making excellent hardware, enabling development of this project.
- OpenCV: Appreciation goes to the OpenCV community for supplying the computer vision tools and extensive documentation, which we have used and are crucial to the solution for the obstacle round.
- GitHub: Recognition is given to GitHub for providing a collaborative platform that streamlined code management and teamwork, ensuring efficient development.