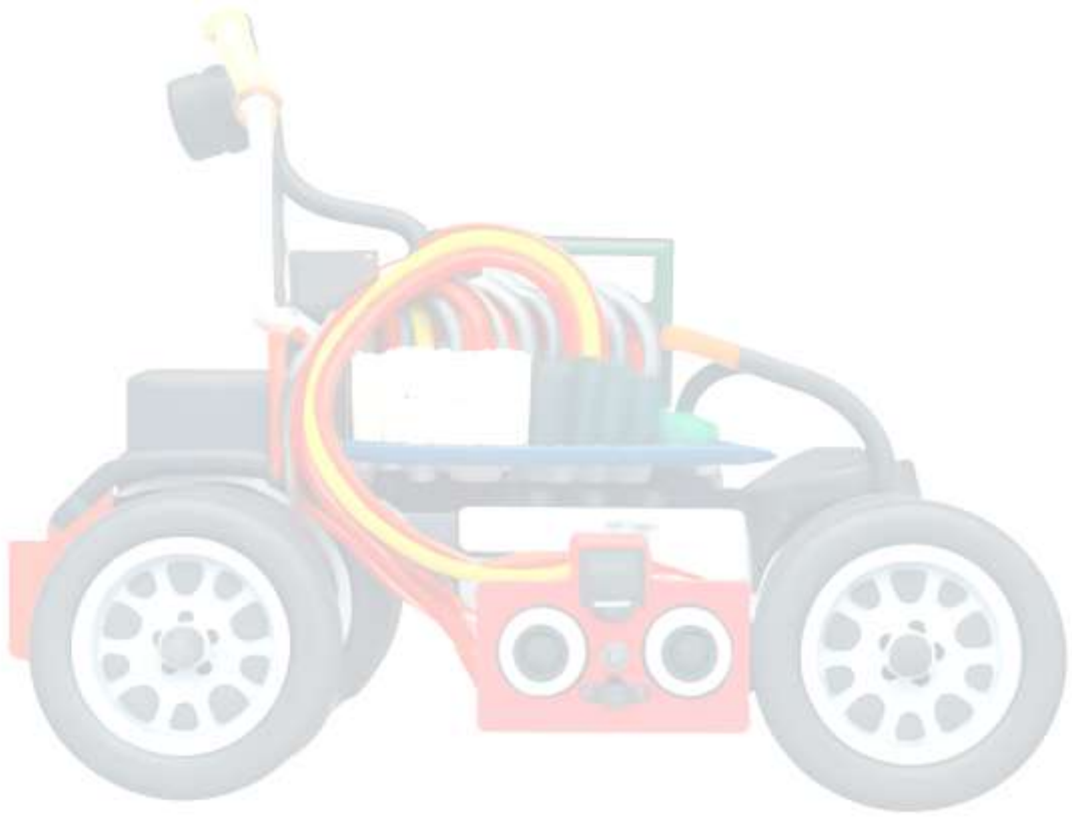
Viva La Vida — WRO FE Obstacle Challenge

VIVA ΛΑ VIΔΑ

IN A ROBOTICS/ENGINEERING CONTEXT, IT IS A FUN, PLAYFUL SLOGAN MEANING “LONG LIVE THE SCREW!”, CELEBRATING THE SIMPLE BUT ESSENTIAL MECHANICAL COMPONENT!

Table of Contents

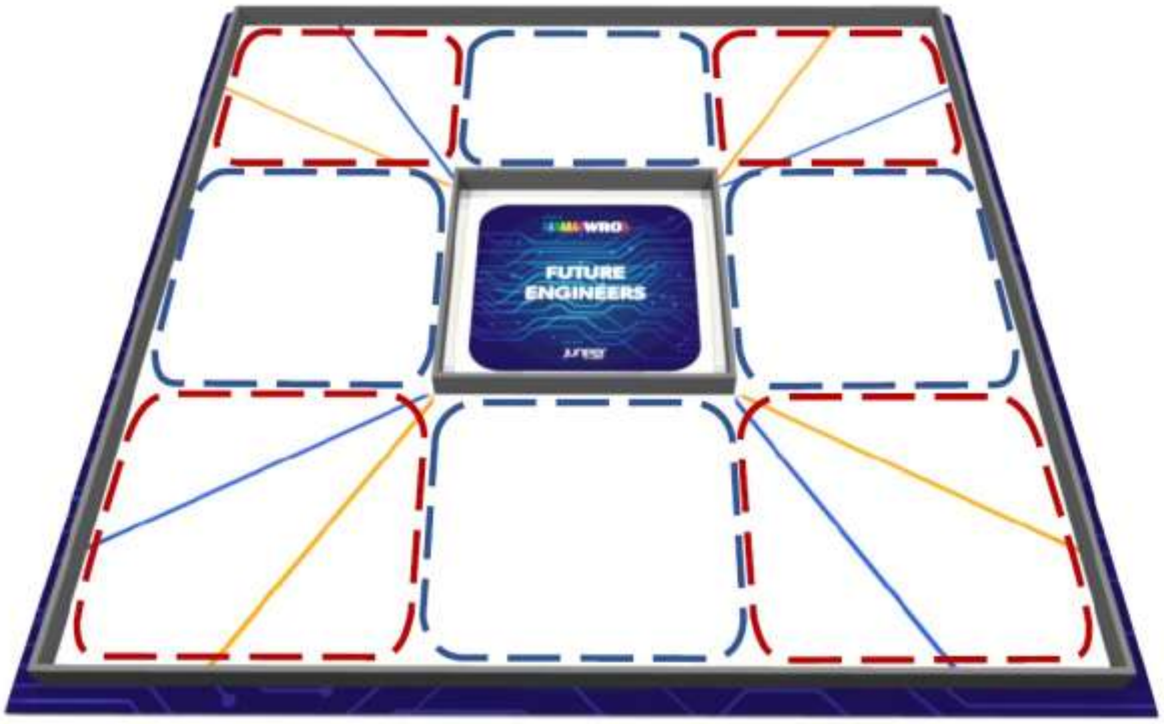


1 Overview & Scope

Mission: Autonomously complete three (3) laps on the official Obstacle Challenge track while respecting randomly placed red and green traffic signs and finishing with a parallel parking manoeuvre inside the parking lot. The robot must decide on which side of the lane to drive based on the colour of the pillars and must avoid moving them. After three laps, it has to locate the magenta parking lot and perform a controlled parallel park.

The game field is a fixed-width racetrack divided into straight sections and corners. At predefined "seats" along the straights, red or green traffic signs (50×50×100 mm pillars) can be placed before each round. A red pillar instructs the vehicle to keep to the right side of the lane, while a green pillar instructs it to keep to the left side. The vehicle must not knock over, push, or significantly displace these signs during its run.

After the third lap, the objective changes from lap-time to precision: the robot needs to detect the magenta parking lot limitations and execute a parallel parking manoeuvre so that the car is fully inside the parking area and aligned with the parking borders.



Our solution reuses the same core hardware platform as in the 1st mission (Raspberry Pi, IMU, ultrasonic/ToF distance sensors, steering servo, DC motor), but adds a camera-based perception pipeline. Computer vision is responsible for lane and traffic-sign detection; the distance sensors and IMU provide safety, collision avoidance, and yaw-based heading control. A finite state machine (FSM) coordinates all behaviours: lap driving, sign handling, obstacle avoidance, parking search, and the parking manoeuvre itself.

2 The robot - Hardware & Wiring

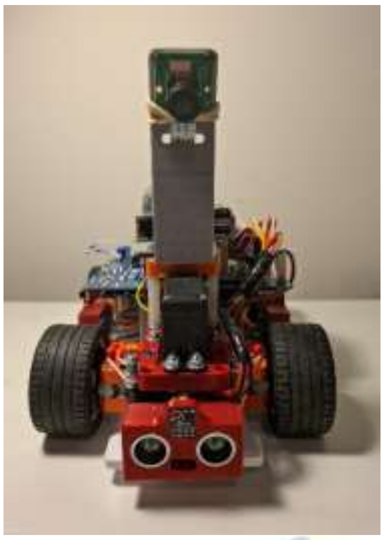
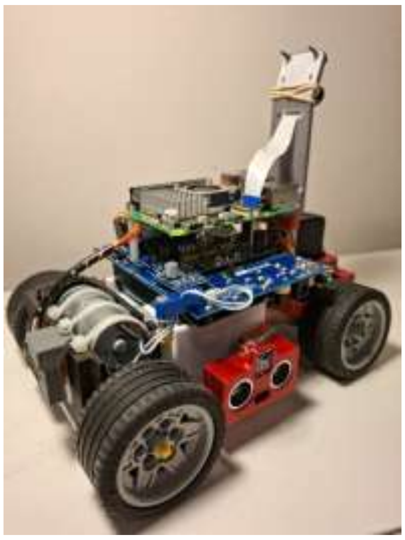
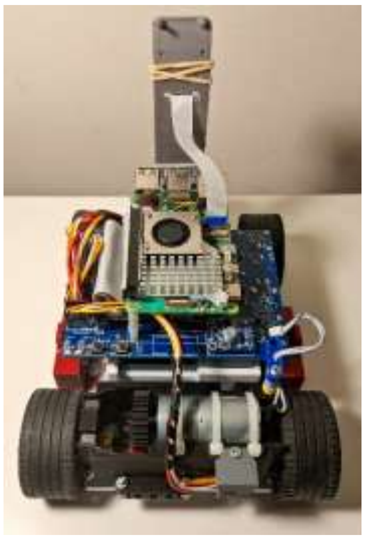
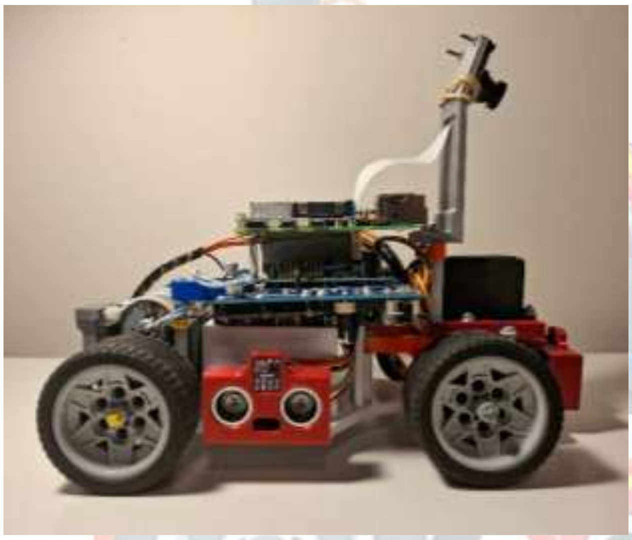
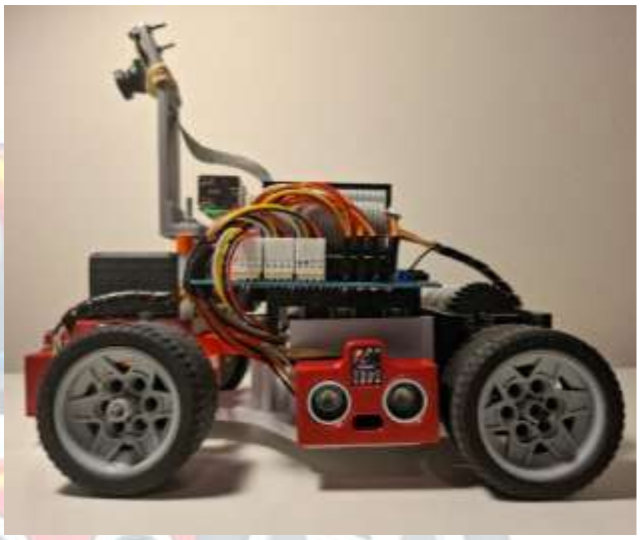
2.1 Bill of Materials

Component	Role	Interface/Notes
Raspberry Pi	Main controller	GPIO/I²C
PCA9685	Servo + motor PWM	50 Hz PWM; I²C
MPU6050	Gyro-Z for yaw	I²C
Ultrasonic x3	Front/Left/Right	Trig/Echo via gpiozero
VL53L0X (opt.)	ToF sensors	I²C + XSHUT addressing
Steering Servo	Single	1000–2000 µs pulse
Motor + Driver/ESC	Single	PWM ch1/ch2 (FWD/REV)
Start Button	Headless start	GPIO20
Status LEDs	Green / Red	GPIO19 / GPIO13
Battery/Power	5–6 V servo; 5 V logic	Common ground

2.2 Pin Map

Subsystem	Signal	Pin/Channel	Dir	Notes
Start Button	START_BTN	GPIO 20	IN	Pull-up; active LOW
LEDs	GREEN/RED	GPIO 19 / 13	OUT	Status/ready
US Front	TRIG / ECHO	22 / 23	OUT / IN	max_distance, queue_len
US Left	TRIG / ECHO	27 / 17	OUT / IN	max_distance, queue_len
US Right	TRIG / ECHO	5 / 6	OUT / IN	max_distance, queue_len
Servo	SERVO_CHANNEL	PCA ch 0	PWM	Steering
Motor	MOTOR_FWD / REV	PCA ch 1 / 2	PWM	Duty 0–100%
I²C	SCL / SDA	board.SCL / board.SDA	—	PCA/IMU/ToF
ToF XSHUT	L/R/F/B/LF/LR	D16 / D25 / D26 / D24 / D8 / D7	OUT	Addr 0x30/31/32/33/34/35

2.3 Robot photos

3 System Architecture

The Obstacle Challenge solution is implemented as a single Python program running on a Raspberry Pi. It builds on the same electromechanical stack as the 1st mission (PCA9685 for servo/motor PWM, MPU6050 IMU for yaw, ultrasonic/ToF for distances), and extends it with a camera module for lane and traffic-sign detection.

At a high level, the architecture is divided into three layers:

1. **Perception Layer**
   - **Camera & Computer Vision**
     - PiCamera2 captures RGB frames at a fixed resolution and frame rate.
     - OpenCV converts frames to HSV colour space and extracts:
       - Blue/orange lane markings on the mat.
       - Red and green traffic signs (pillars).
       - Magenta parking lot limitations.
   - **Range & Pose Sensing**
     - Ultrasonic and/or VL53L0X ToF sensors measure front, left, and right distances.

        - An MPU6050 IMU provides gyro-Z data, integrated into **yaw** (relative heading) for turn and heading control.

2. **Decision & Control Layer**
    - A **finite state machine (FSM)** maintains the current high-level mode:
        - Lap driving (following lane and obeying traffic signs).
        - Obstacle handling (stop / slow-down in front of unexpected objects).
        - Parking search (after lap 3).
        - Parking alignment and execution.
    - The FSM uses:
        - **Vision features** (lane centre offset, sign type/position, parking markers).
        - **Distance thresholds** (safety margins and stop distance).
        - **Yaw targets** (for consistent cornering and straight-line recovery).
    - The controller outputs:
        - Steering commands (servo angle).
        - Speed commands (PWM duty to the drive motor).
3. **Infrastructure Layer**
    - **Hardware abstraction** (functions or a small controller class) wraps:
        - Servo control (angle → microsecond pulse → PCA9685 duty cycle).
        - Motor control (forward/backward duty mapping).
        - Raw sensor reads (ultrasonic, ToF, IMU).
    - **Configuration & logging**
        - Constants for thresholds and speeds are loaded from code and optional JSON files.
        - Runtime logs (print statements or optional CSV export) record timestamps, state, yaw, distances, and detection results for offline analysis and tuning.

I
n calibration/debug mode, the same program can run with more verbose logging and lower speeds; in competition mode, it uses a fixed configuration and headless start button, respecting the WRO start-up rules (power switch + separate start button).

3.1 Code Structure

3.1.1 Imports

3.1.1.1 External Libraries

- cv2, numpy: computer vision (colour conversion, masks, contours, geometry).
- picamera2: frame capture from the Raspberry Pi camera.
- adafruit_pca9685: PWM driver for motor and steering servo.
- adafruit_mpu6050: IMU (gyro-Z readings).
- adafruit_vl53l0x and/or gpiozero.DistanceSensor: distance sensors (front, left, right).
- threading, time, collections.deque: concurrency and timing.

3.1.2 Global Configuration

- **Camera & vision**
    - Camera resolution, frame rate, and rotation.
    - HSV ranges for blue/orange lane lines, red/green traffic signs, and magenta parking markers.

- Cropping percentages (top/bottom/left/right) to restrict processing to relevant regions of interest.

- **Motion & safety**
  - Speed setpoints for normal lap driving, sign-approach slowdown, obstacle approach, parking search, and parking manoeuvre.
  - Distance thresholds for “safe to go”, “slow down”, and “must stop”.
  - Yaw-related constants for turning and for recovering the straight heading after obstacle avoidance.

Configuration is initialised with sensible defaults and can optionally be overridden by loading a JSON file so that competition tuning does not require editing the code.

3.1.3 Perception Helpers

- **Lane detection**
  - preprocess_lane(frame): crop, blur, convert to HSV, and threshold for blue/orange.
  - find_lane_center(mask): compute the lane centre (cx) within a horizontal ROI; return an offset from the image centre.
- **Traffic sign detection**
  - detect_signs(frame): threshold for red and green in a near-front ROI, extract contours, filter by area/aspect ratio, and classify each detection as “GREEN_LEFT” or “RED_RIGHT”.
  - The closest valid detection is converted into a discrete **lane side** command that remains active until the next sign.
- **Parking detection**
  - detect_parking(frame): search for magenta regions near the right-hand border (or according to starting direction), and output a Boolean “parking_slot_visible” plus approximate geometry.

3.2 Control & FSM

- A small enum or set of constants describes the FSM states (e.g. IDLE, LAP_DRIVE, SIGN_HANDLE, OBSTACLE_STOP, PARK_SEARCH, PARK_ALIGN, PARK_EXECUTE, FINISHED).
- The main loop:
  1. Grabs a new frame from the camera.
  2. Reads distance sensors and IMU (gyro-Z).
  3. Updates yaw by integrating gyro-Z.

4. Runs perception helpers to update lane centre, active lane side, sign presence, and parking visibility.

5. Calls the FSM step function, which decides:

    - Current speed setpoint.
    - Steering angle.
    - State transitions (e.g., after three laps, when a sign is recognised, or when the parking lot comes into view).

All actuation goes through a small abstraction layer so that the FSM code remains readable and declarative (“set speed to PARK_SLOW”, “steer to lane_center_angle”) rather than dealing directly with PWM counts.

4 Decision-Making Architecture

The decision-making logic separates **perception**, **state-based intent**, and **low-level actuation**. Each step uses explicit thresholds, timeouts, and hysteresis to avoid unstable behaviour.

4.1 Perception Pipeline

1. **Lane model**

    - The bottom part of the image (road near the front of the robot) is cropped and thresholded for blue/orange to obtain a binary mask of lane markings.
    - The mask is collapsed horizontally to estimate a **lane centre** and its offset relative to the image centre.
    - This offset is converted into a desired steering angle, with a dead-band around zero to avoid constant micro-corrections.

2. **Traffic sign model**

    - A forward ROI is scanned for red and green blobs with the expected pillar size and aspect ratio.
    - If a green pillar is detected, the robot sets a **mode flag** lane_side = LEFT. If a red pillar is detected, it sets lane_side = RIGHT.
    - Once set, lane_side remains active for the current section until another valid sign is detected, mirroring the rule that signs indicate which side to keep rather than a one-frame command.

3. **Parking lot model**

    - When the “laps completed” condition is satisfied, the perception pipeline starts looking for magenta rectangles corresponding to parking lot limitations.

- As the robot approaches, the relative size and position of the magenta blobs are used to align the vehicle beside the slot and to trigger the parking manoeuvre.

4. **Range & heading**
    - Front distance: used to slow down/stop before walls, traffic signs, or unexpected obstacles.
    - Left/right distances: used for additional wall-distance safety and as a fallback estimate when lane markings are momentarily lost.
    - Yaw (integrated gyro-Z): used to stabilise turns, recover original heading after obstacle avoidance, and keep the vehicle aligned during straight segments.

4.2 Behavioural Logic

- **Lane following under sign constraints**
    - The controller computes a **desired lateral offset** based on lane_side:
        - If lane_side = LEFT, target the left half of the lane.
        - If lane_side = RIGHT, target the right half of the lane.
    - The lane-centre offset is combined with this target to produce a steering command.
    - If lane markings are temporarily lost, the controller falls back to yaw (keep last heading) and side distances (keep a safe margin to the visible wall).
- **Obstacle handling**
    - If the front distance falls below a safety threshold, the robot enters an obstacle/stop behaviour:
        - Speed is ramped down to zero.
        - The robot waits for a short period and checks again.
        - If the path becomes clear (e.g., a false positive or transient reflection), the robot resumes lane following.
        - If the obstacle remains, the FSM may choose a slow “creep” or fail-safe to avoid pushing traffic signs.
- **Lap counting**
    - Laps are counted either by yaw-based corner counting (similar to the 1st mission) or by crossing detection of a specific lane pattern near the start section.
    - Once three laps are completed, all sign-based lane decisions are ignored, and the FSM transitions to the **parking phase**, while still maintaining safety distance rules.
- **Parking**

- In the parking phase, decisions are dominated by parking-related detections:
  - The robot searches for magenta parking markers.
  - When detected, it aligns longitudinally and laterally using a combination of lane centre, yaw, and distance sensors.
  - The parallel parking manoeuvre is executed as a sequence of controlled reverse/forward segments with predefined steering angles, finalising when the car is inside and aligned.

All decisions rely on clear numeric thresholds defined in the configuration (e.g. minimum sign area, distance to start slowing, yaw error tolerance). This makes the behaviour reproducible and tunable between rounds.

5 Finite State Machine (FSM)

The Obstacle Challenge controller uses a finite state machine to structure behavior into clear phases. Below is a conceptual FSM; the exact constant names in the code can follow this mapping.

5.1 States

State	Enter	Actions	Exit
IDLE	Program started; waiting for Start button	Motors off; centre steering; zero lap counter and yaw bias	Start button pressed → **LAP_DRIVE**
LAP_DRIVE	Start pressed; laps < 3	Lane following under current lane_side; obey sign updates; normal speed	Front too close → **OBSTACLE_STOP**; laps = 3 → **PARK_SEARCH**
OBSTACLE_STOP	Front distance < STOP_DIST (any time)	Ramp speed to 0; keep heading; wait OBSTACLE_WAIT_TIME and re-check	Path clear → back to previous (usually **LAP_DRIVE**); hard timeout → **FINISHED** (fail-safe)
PARK_SEARCH	Laps completed (≥3)	Ignore new sign instructions; drive slowly; search for parking markers	Parking lot detected → **PARK_ALIGN**
PARK_ALIGN	Parking markers visible	Align alongside slot using lane, magenta markers, yaw, and side distances	Alignment constraints met → **PARK_EXECUTE**

5.2 Sate actions

- **IDLE**
  - On entry: all counters cleared, yaw integration reset, and any sign/lane flags reset.

- Guard: only the physical Start button or an explicit software command can leave this state.

- **CRUISE**
  - Normal driving mode for the three laps.
  - Uses lane detection plus lane_side (from sign detection) to compute steering.
  - Speeds and accelerations are tuned for lap time while staying safely below distance and yaw limits.
  - Transition to **OBSTACLE_STOP** if d_front < STOP_THRESHOLD.
  - Lap detection increments lap_count; when lap_count >= 3, the state transitions to **PARK_SEARCH**.

- **OBSTACLE_AVOID**
  - Purpose: honour the rule that traffic signs must not be pushed or moved.
  - Immediate cut of motor PWM and brake by coasting to stop.
  - Waits for OBSTACLE_WAIT_TIME; if the obstacle is still too close, the robot either remains stopped or executes a very cautious low-speed creep, but never applies enough force to move pillars.

- **PARK_SEARCH**
  - Lateral behaviour is simplified (e.g. keep right side of lane, independent of previous sign rules).
  - The vision pipeline prioritises magenta detection; other detections are ignored except for safety.
  - Speed is reduced to improve detection reliability and parking precision.

- **PARK_ALIGN**
  - Robot positions itself parallel to the parking slot, at a known offset:
    - Yaw is corrected so car faces along the lane.
    - Distance from the parking boundary is adjusted using side sensors.
  - When alignment error (in yaw and lateral distance) is below configurable thresholds, transition to **PARK_EXECUTE**.

- **PARK_EXECUTE**
  - Parallel parking sequence:
    - Reverse with steering into the slot until a distance or time threshold.

- Counter-steer forward slightly to centre the vehicle.

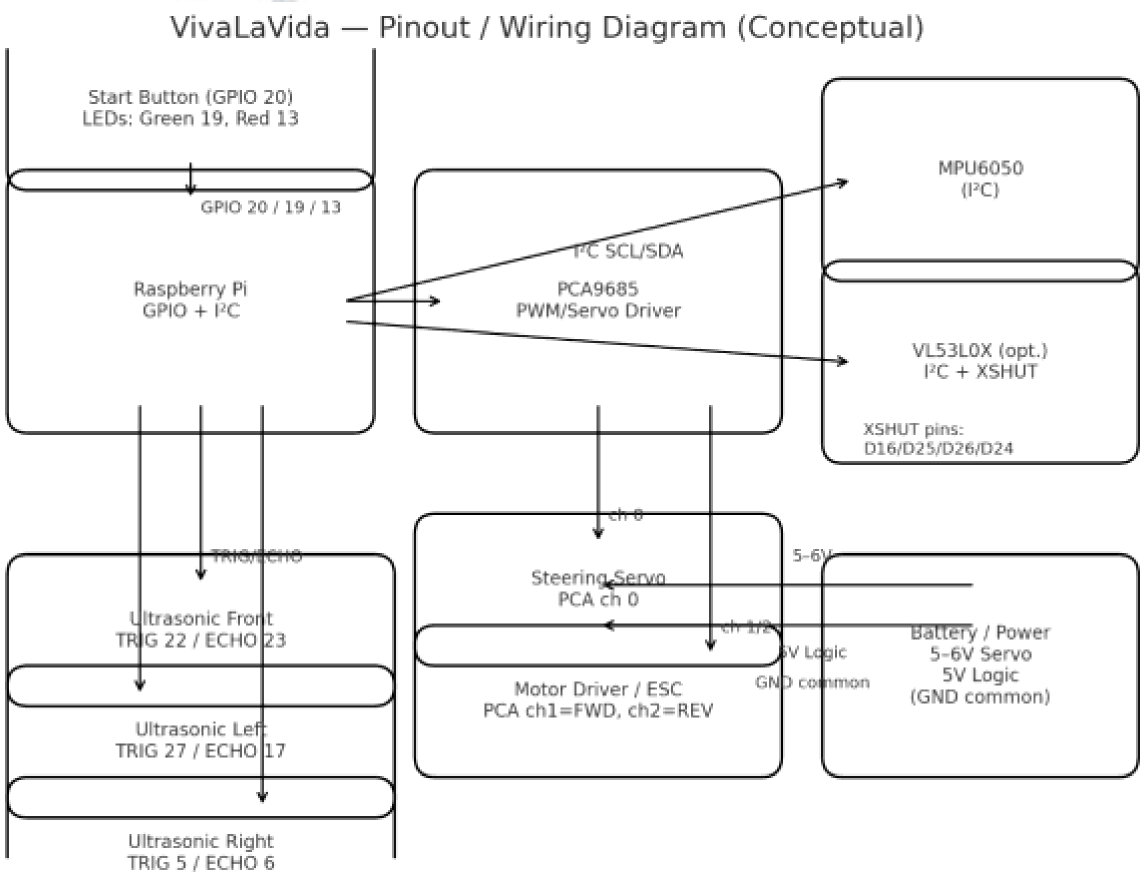
  - During the manoeuvre, yaw and distances are monitored to avoid collisions with parking boundaries.

- **FINISHED**
  - Terminal state for a successful run (three laps + parking).
  - Motors remain off; the robot awaits manual retrieval or a reset command.

This FSM keeps the mission logic explicit and easy to explain to judges: every behaviour change is tied to a specific sensor condition or mission milestone (lap count, sign detection, parking detection).

6 Diagrams

6.1 Pinout / Wiring (Conceptual)



7 Appendix A - Variables

Notes

- Units:
    - Distances in centimetres (cm).
    - Time in seconds (s).
    - Angles in degrees (°).
    - Speeds as 0–100 % motor PWM.
- Many variables are shared with the 1st mission (speeds, basic safety thresholds, yaw filtering). The 2nd mission adds a new group of **vision-related** and **parking-related** variables.

Below is a non-exhaustive list of the most important variables grouped by function.

- **Initialization & Mode**
- **USE_CAMERA**
  Enables/disables the camera and vision pipeline (for testing with only distance sensors).
  Values: 0 = disabled, 1 = enabled.
- **START_DIRECTION**
  Encodes whether the run is clockwise or counter-clockwise, affecting lane-side interpretation and parking search direction.
  Values: "CW", "CCW" (or 0/1).
- **USE_TOF_FRONT / USE_TOF_SIDES**
  Select ToF or ultrasonic sensors for front and side measurements.
  Values: 0 = ultrasonic, 1 = ToF.
- **Lane & Sign Detection**
- **HSV_BLUE_LOW/HIGH, HSV_ORANGE_LOW/HIGH**
  HSV ranges used to detect lane markings on the mat (blue and orange lines).
  Notes: Tuned once under competition lighting, kept stable thereafter.
- **HSV_RED_LOW/HIGH, HSV_GREEN_LOW/HIGH**
  HSV ranges used to detect red and green traffic signs. Based on the official RGB values of the pillars; slightly expanded to allow for lighting changes.
- **SIGN_MIN_AREA**
  Minimum contour area in pixels for a candidate to be considered a valid traffic sign (filters out noise).
- **SIGN_MAX_ASPECT**
  Maximum allowed width/height ratio to still be considered a “pillar”. Very wide blobs are rejected.
- **SIGN_PERSIST_FRAMES**
  Number of consecutive frames a detection must appear before lane_side is updated. Helps avoid flicker.
- **LANE_ROI_TOP, LANE_ROI_BOTTOM (percent)**
  Vertical cropping percentages (0–1) defining the lane-detection band near the bottom of the image.
- **LANE_DEAD_BAND**
  Range of lane centre offsets (in pixels or normalised units) considered “good enough” to drive straight.
- **LANE_KP, LANE_MAX_CORR**
  Proportional gain and maximum allowed steering correction from lane offset to servo angle.
- **Parking Detection & Manoeuvre**
- **HSV_MAGENTA_LOW/HIGH**
  HSV ranges used to detect the magenta parking lot limitations.

- **PARK_MIN_AREA**
  Minimum contour area for a magenta region to be treated as a parking marker.
- **PARK_EDGE_MARGIN**
  How far from the image borders magenta blobs are expected when the robot is correctly aligned with the slot.
- **PARK_ALIGN_LAT_TOL**
  Allowed lateral distance error (cm) between the car and the parking boundary before the manoeuvre starts.
- **PARK_ALIGN_YAW_TOL**
  Allowed yaw error (°) before starting the parallel parking sequence.
- **PARK_BACK_DIST**, **PARK_FORWARD_DIST**
  Target reverse and forward travel distances (or equivalent times) during the parking manoeuvre.
- **PARK_SPEED**
  Speed used during parking; typically much lower than lap speed for precision.
- **Speeds & Safety (shared with 1st mission, re-used here)**
- **SPEED_LAP / SPEED_SIGN_APPROACH / SPEED_PARK_SEARCH / SPEED_PARK_EXECUTE**
  Main speed setpoints for each phase of the mission (see Section 6).
- **STOP_THRESHOLD**
  Front distance (cm) below which the robot must stop immediately to avoid collisions or moving traffic signs.
- **SLOWDOWN_DISTANCE**
  Distance at which the robot starts reducing speed near obstacles or signs.
- **SIDE_SAFE_DISTANCE**
  Minimum acceptable distance to side walls or pillars; used as an extra guard when lane information is noisy.
- **Yaw & Filtering**
- **YAW_FILTER_ALPHA**
  Exponential moving average factor applied to gyro-Z before integration (0–1; higher = less smoothing).
- **YAW_STRAIGHT_TOL**
  Allowed yaw deviation from the target “straight” heading before corrections are applied.
- **YAW_PARK_ALIGN_TOL**
  Stricter yaw tolerance used in the parking alignment phase.
- **N_READINGS**
  Window size for median filtering of distance sensors.
- **FILTER_ALPHA_DIST, FILTER_JUMP_DIST**
  EMA smoothing factor and maximum jump for distance sensors; large jumps are considered spikes and ignored.
- **Timing & Mission Management**
- **OBSTACLE_WAIT_TIME**
  How long to wait in **OBSTACLE_STOP** before re-checking if the path is clear.
- **FRAME_INTERVAL**
  Desired time between camera processing frames; can be increased to reduce CPU load.
- **MAX_LAPS**
  Number of laps to drive before switching to parking logic (3 for the official Obstacle Challenge).
- **TIMEOUT_PARK_SEARCH**
  Maximum time allowed in **PARK_SEARCH** before failing safely in **FINISHED** if the slot is never found.