MEAM 5100 Final Report

Team 11(Zhixin Yin/ Yishen Zhang/Kunjie Xu)

Functionality

Our team designed a mecanum-wheeled mobile robot driven by four motors. The robot is organized into three functional layers. The bottom layer is responsible for motion and drive control. Two motor drivers (L298N) are used to independently control the rotation direction and speed of the front two wheels and the rear two wheels.

The second layer focuses on control and sensing. A single ESP32-S3 board is used as the main controller to handle all manual and autonomous tasks. Two ToF distance sensors are installed to measure the distances between the robot and surrounding obstacles, which are used to assist the task of navigating around the arena.

The top layer integrates localization and attacking modules. Two Vive sensors are used to determine the robot's two-dimensional position and orientation within the field. A servo motor is mounted at the center of the front of the robot and connected to a straight arm, enabling a 180-degree sweeping motion to attack opposing vehicles.

We chose a mecanum-wheel platform primarily for its flexibility. In addition to forward motion and turning, the robot can translate laterally and diagonally while maintaining a fixed heading. It can also perform fast and accurate in-place rotation and backward motion. Each wheel is driven by an independent motor, providing both high maneuverability and sufficient power to complete all manual tasks and rapidly strike moving targets.

Another key advantage of the mecanum-wheel design is its suitability for autonomous navigation. In autonomous tasks such as capturing towers, attacking the nexus, and moving to specified coordinates, the robot maintains a constant heading and relies on lateral and longitudinal translations. As long as the Vive position measurements are accurate and stable, the robot can reliably reach target locations and execute tasks. For the task of circling the wall, a clockwise motion strategy was adopted.

Two ToF sensors are installed between the second and third layers of the robot to measure distances to obstacles in front of the robot and on its left side. During wall-following, the robot continuously adjusts wheel speeds based on the left-side distance to maintain a desired clearance from the wall. When an obstacle is detected within a predefined safety distance in front of the robot, it performs an in-place right turn until the path is clear, after which it resumes wall following.

Ultimately, the robot successfully completed all manual and autonomous tasks. However, during the final check-off demonstration, increased noise and fluctuations in Vive position measurements near the edges of the arena caused the final turn in the wall-following task to be imperfect, and the autonomous uphill attack on the high tower did not achieve the expected performance. Under stable Vive measurement conditions in the lab, all functionalities were verified to work as intended.

Mechanical Design

1. Chassis architecture and structural choices

Material and fabrication

The chassis uses 1/8 in acrylic, which is 3.175 mm thick. All plates were manufactured by laser cutting. The structure is a three-layer stacked frame. Each plate acts as a rigid deck that supports a specific subsystem. Layer 1 supports 4*N20 motor, 2*L298N motor diver and 4*18650 battery. Layer 2 supports 2*TOF400C, wiring and ESP32-S3-WROOM. Layer 3 supports the attack mechanism, 2*vive tracker and 4*AA battery.

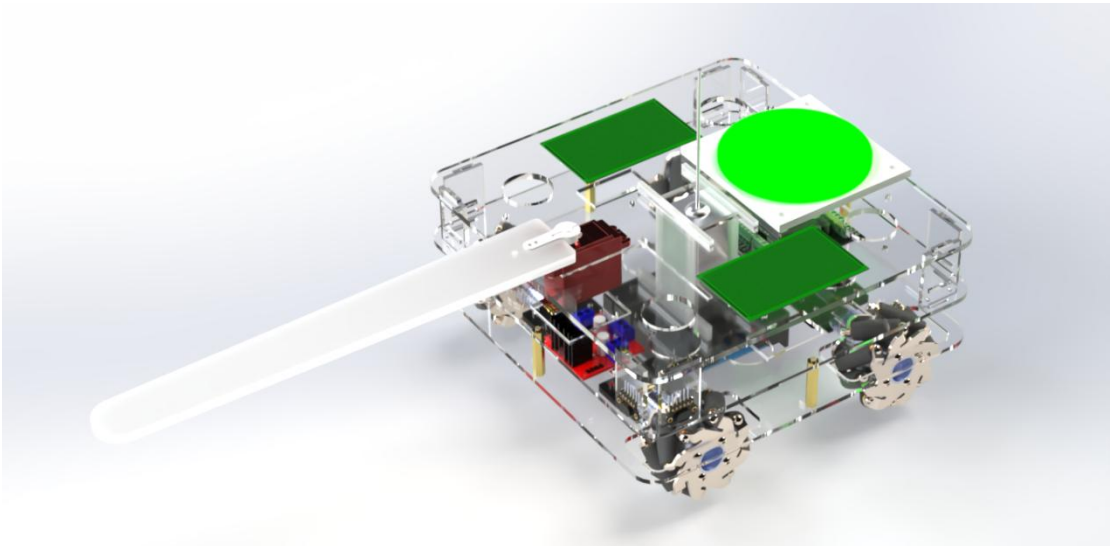


Figure 2.1 CAD design

Layer-to-layer connection

The three plates are separated and supported by hex standoffs. The standoffs are bolted through the acrylic. This creates a lightweight box-like frame that is fast to assemble and easy to service. Individual layers can be removed without damaging the chassis.

Reinforcement using interlocking splines

Acrylic is stiff but brittle, so it can crack near fasteners under shock loads. To increase stiffness and reduce torsional flex, we added interlocking tab-and-slot splines between Layer 2 and Layer 3.

Structural reinforcement. The splines increase the effective shear area between layers, so the stack resists twisting and compression better than screw clamp friction alone.

Functional integration. The same spline geometry provides a repeatable locating feature for the time-of-flight sensor mount. This reduces part count and improves repeatability after reassembly.

2. Top hat mounting

Our design emphasizes fast removal without tools, repeatable alignment, and strong in-plane constraint so the top hat does not shift during impacts or aggressive driving. While HP display is mounted to Layer 3 using bolts and screws.

Sandwich clamp module

The top hat has four mounting holes. we laser cut two acrylic plates, 70 mm by 70 mm, with matching hole spacing. The top hat is captured between these two plates. Instead of screws, we inserted four wooden dowel pins through the aligned holes. The pins act as keys, so they prevent relative sliding and rotation between the top hat and the clamp plates.

To prevent the dowels from backing out, we covered the hole openings with tape. we left a peel tab on the tape edge. During a match, we can remove the tape and invert the assembly so the dowels drop out. This reduces top hat removal time from minutes to seconds.

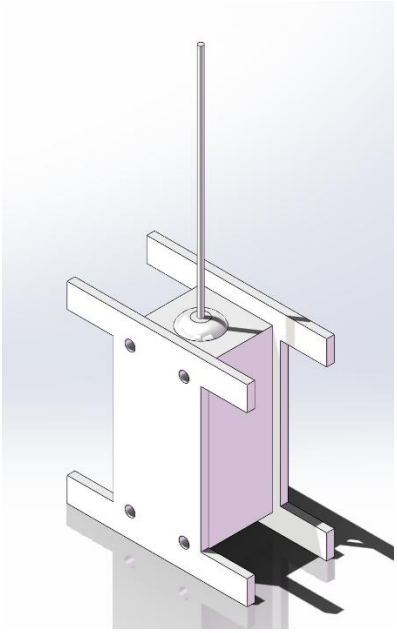


Figure 2.2 Assembled top hat

Drawer-style chassis interface

The clamp module does not attach to the chassis using screws. Layer 2 and Layer 3 each include an I-shaped slot. The clamp module slides into the slots like a drawer. Once inserted, the slot walls constrain translation in the plane and constrain yaw. The module intentionally remains free in the vertical direction so it can be extracted quickly for debugging. This design makes alignment depend on laser cut geometry rather than fastener preload, which improves repeatability across many install cycles.

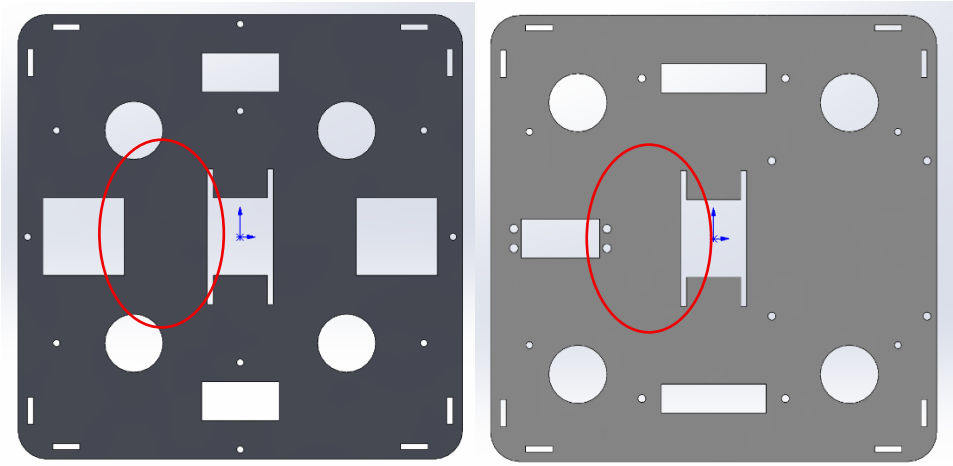


Figure 2.3 I-shaped slot on Layer 2 and Layer 3

3. Actuator mounting

Drive motors

The drivetrain uses four 150:1 N20 gearmotors with a specified stall torque of 6.8 kgf·cm. Each motor is installed in a custom 3D printed motor hub. The motor body is seated in a pocket and fixed using hot glue. This speeds up assembly, holds alignment for small motors, and adds some vibration damping. Each hub is bolted to the Layer 1 plate using machine screws and nuts. This creates a stiff and serviceable mount. A failed motor can be replaced by removing one hub rather than disassembling the full chassis.

Attack servo and bat end effector

The attack mechanism is mounted on Layer 3 and raised using standoffs. This provides clearance for rotation and reduces collision risk with wiring and sensor mounts. The end effector is a bat-style acrylic part with an effective length of 280 mm. It is mounted to the servo horn using a three-screw pattern, which improves impact resistance and reduces loosening risk.

The servo rated output torque is converted as follows:

$$\tau_s = 24.5\ \text{kgf}\cdot\text{cm} \times 0.09807\ \frac{\text{N}\cdot\text{m}}{\text{kgf}\cdot\text{cm}} \approx 2.40\text{N}\cdot\text{m}$$

A useful check is the equivalent tangential force at the servo horn radius:

$$F_t = \frac{\tau_s}{r_h}$$

Using an effective horn radius of $r_h = 10\ \text{mm} = 0.01\ \text{m}$,

$$F_t \approx \frac{2.40\ \text{N}\cdot\text{m}}{0.01\ \text{m}} \approx 240\ \text{N}$$

4. Sensor placement and interference management

vive tracker placement

The vive tracker is mounted on the top of Layer 3 and elevated using standoffs. The goal is an unobstructed line of sight and no interference with the attack mechanism. The bat sweep envelope was kept below the tracker height, and the swing plane was offset so the bat clears the tracker throughout its motion.

Time of flight sensor placement

The time of flight sensor is placed at the lowest position on Layer 2. The initial plan was to place it on Layer 1, but performance was highly sensitive to height. If the sensor is too low, it detects the ramp early and can trigger premature turning logic. If the sensor is too high, it can miss the wall and the robot can remain in a forward drive state. Layer 2 provided a stable compromise. we also designed a compatible key-slot interface on Layer 1 so the same mounting piece can be moved between layers with minimal redesign.

Cable routing and battery mounting

Each layer includes cable pass-through holes for vertical routing. After assembly, wiring is bundled with zip ties and electrical tape to avoid snags with mecanum rollers and the swinging bat. The battery pack is mounted under Layer 1 near the geometric center. This lowers the center of gravity and improves weight distribution across wheels, which matters for mecanum traction.

Drivetrain sizing and ramp feasibility estimate

The mecanum wheel diameter is 68 mm, so the wheel radius is $r_w = 34\ \text{mm} = 0.034$ m. A component-based estimate places robot mass between 0.8 kg and 1.0 kg. Four 18650 cells contribute about 180 g. Four AA cells contribute about 90 g to 110 g. Two L298N driver boards contribute about 50 g. The DS3225 servo contributes about 60 g. Four N20 motors contribute about 48 g. The chassis, wheels, fasteners, and miscellaneous parts contribute about 350 g to 500 g.

On a ramp with angle θ, the downslope component of gravity is:

$$F_{\parallel} = \text{mgsin}\,\theta$$

Assuming the load is shared across four wheels, the required torque per wheel is:

$$\tau_w = \frac{F_{\parallel} r_w}{4} = \frac{mg\sin\,\theta}{4} r_w$$

Using $\theta = 20°,\ \sin\,20° \approx 0.342,\ g = 9.81\ \text{m/s}^2$, and $r_w = 0.034$ m,

$$\tau_w \approx \frac{m \cdot 9.81 \cdot 0.342}{4} \cdot 0.034 \approx 0.0285m\ \text{N} \cdot \text{m}$$

For representative masses:

$$\tau_w(m = 0.9\ \text{kg}) \approx 0.0257\ \text{N} \cdot \text{m} \approx 0.262\ \text{kgf} \cdot \text{cm}$$
$$\tau_w(m = 1.0\ \text{kg}) \approx 0.0285\ \text{N} \cdot \text{m} \approx 0.291\ \text{kgf} \cdot \text{cm}$$

Compared to a motor stall torque of 6.8 kgf·cm, this suggests the drivetrain is torque-capable in an ideal model. In practice, ramp climbing can still fail due to reduced traction from mecanum rollers, voltage drop in the motor driver, and battery sag under load.

Mecanum kinematics and square layout rationale

The wheelbase is 135 mm and the four motors are placed in a square configuration. This symmetry reduces directional bias and simplifies calibration. It also improves yaw control because wheel speed magnitudes become more uniform when the effective half-length $L$and half-width W are similar.

A standard mapping from body velocities to wheel angular speeds is:

$$\begin{bmatrix}\omega_{FL}\\ \omega_{FR}\\ \omega_{RL}\\ \omega_{RR}\end{bmatrix}=\frac{1}{r}\begin{bmatrix}1 & -1 & -(L+W)\\ 1 & 1 & (L+W)\\ 1 & 1 & -(L+W)\\ 1 & -1 & (L+W)\end{bmatrix}\begin{bmatrix}v_x\\ v_y\\ \omega_z\end{bmatrix}$$

This supports translation, strafing for alignment, and rotation for aiming the attack mechanism.

5. Intended and actual mechanical performance

The intended performance was a rigid three-layer acrylic frame with good service access, rapid top hat removal with repeatable alignment, stable mecanum mobility, and a reliable bat attack without vive interference.

In testing, the frame remained rigid during normal driving, and the stacked structure maintained good squareness after repeated handling. The interlocking splines with screw clamping reduced upper-deck twist during impacts, so the vive tracker and the bat mount stayed aligned after multiple collisions. The drawer-style top hat mount preserved repeatable alignment across multiple removals and reduced maintenance time significantly. However, because tolerance was not fully considered in the first iteration, the insertion and extraction friction was higher than expected. This made top hat removal feel stiff and inconsistent, especially after small edge wear from repeated use. Applying a small amount of lubricant reduced friction and improved repeatability, but it also revealed that a designed clearance and lead-in chamfer would be a more robust solution.

Mecanum mobility worked well on flat surfaces, with reliable omnidirectional motion and stable in-place rotation. Actual performance on the ramp was less consistent than the ideal torque estimate suggested. When the motor PWM command was low, the robot often struggled to climb, even though the motors were theoretically torque-capable. The main observation is that low PWM reduces average motor voltage, which reduces available torque and increases sensitivity to static friction and battery sag. Under these conditions, the mecanum rollers also reduced effective traction, so the wheels could slip rather than convert torque into uphill motion. As a result, ramp success depended on using a higher PWM during the climb and maintaining a stable normal force distribution across all four wheels. This behavior was not fully expected during early design because the simplified model did not include traction limits, voltage drop in the driver, and the nonlinear effect of static friction at low speed.

The bat mechanism stayed secure under repeated swings due to the three-screw mount, and the servo mount did not crack the acrylic during normal operation. However, peak impacts sometimes caused small transient vibrations in the top deck, which could slightly change the perceived swing timing relative to the robot motion. This suggests that even when the structure

is globally stiff, local compliance at fasteners and standoffs can still influence dynamic behavior. The vive tracker placement avoided occlusion and collisions throughout the swing envelope, but at aggressive acceleration the top deck experienced small oscillations, which likely added noise to pose estimation. Overall, the structure met the core goals of rigidity and serviceability, but ramp climbing reliability, low PWM behavior, and top hat insertion tolerance were the main gaps between intended and actual performance.

6. Iteration process and lessons learned

Time of flight height tuning did not generalize across ramp and wall-follow behaviors, and the root cause is geometric rather than purely algorithmic. The ToF measures distance along its line of sight. When the robot approaches a ramp, the first surface it can see is the ramp plane, which is effectively a sloped line in a 2D cross section. Changing sensor height changes the intersection point between the ToF beam and the ramp, so the measured distance changes even if the robot does not move.

A simple right triangle model explains this clearly. Consider the ramp cross section as the hypotenuse of a right triangle. The floor is the long horizontal leg, and the vertical rise to the ramp is the short vertical leg. Place the ToF at a point on the short vertical leg at height h. If the ToF beam is approximately horizontal, then the distance from the sensor to the ramp along the beam corresponds to the horizontal distance from that point to the hypotenuse. For a ramp with angle θ relative to the ground, the ramp line satisfies

$$y = x\tan\ \theta$$

At sensor height h, the intersection occurs when $y = h$, which gives

$$h = \mathrm{xtan}\,\theta \Rightarrow x = \frac{h}{\tan\theta} = \mathrm{hcot}\,\theta$$

This x is exactly the measured distance to the ramp along a horizontal beam. The key result is that the distance is proportional to sensor height:

$$d_{\mathrm{ramp}} \propto h$$

Therefore, increasing the ToF height increases the measured distance to the ramp, even when the robot is stationary. This explains why the same control threshold can trigger too early at one height and too late at another height.

This geometric coupling created a practical conflict between ramp handling and wall-follow behavior. A height that is good for stable wall distance estimation may shift ramp detection farther away, which can trigger ramp logic prematurely and cause early turns. A height that delays ramp detection may improve ramp approach stability, but it can reduce wall detection reliability, especially if the wall is partially outside the ToF field or the beam is more likely to miss reflective surfaces. The most important lesson is that ToF height is not a minor mounting detail. It changes the meaning of the sensor measurement in a structured way, and that change propagates directly into decision thresholds and state transitions.

Because of this, we eventually decided to avoid combining wall-follow and uphill motion. The ramp detection distance was not consistent across small mounting changes and small variations

in robot pitch near the ramp entrance. Even a few millimeters of height change or a small change in tilt can shift the intersection geometry, which makes a fixed threshold unreliable. This was one major reason we abandoned wall-follow up the ramp and instead performed a more careful wall-follow search along the lower region of the map, where the surface geometry is closer to planar and the ToF measurement is more interpretable.

We did not fully solve this issue. A robust solution likely requires more sensing information than a single ToF can provide in this configuration. One approach is to add additional ToF sensors at different heights or angles so the algorithm can estimate ramp presence using multiple rays rather than one distance value. This would improve geometric observability, but it would also increase algorithm complexity. It would require sensor fusion, filtering, and logic to handle disagreement between sensors, as well as calibration for mounting angles and offsets. In our design constraints, we accepted the tradeoff and simplified the mission strategy rather than increasing sensor count and software complexity.

An additional lesson came from ramp performance at low PWM, which appeared counterintuitive at first. In an ideal DC motor model, lowering speed is often associated with higher available torque. However, our actual system did not behave like the ideal model because the limiting factor was not only motor torque. At low PWM, the average applied voltage is reduced, so the motor operates closer to the static friction region and can fail to initiate motion. At the same time, battery sag and driver voltage drop further reduce the effective voltage seen by the motor. This can push the system into a regime where the motor cannot overcome stiction, even though stall torque on a datasheet seems sufficient.

Traction was also a major limiting factor. Mecanum wheels trade efficiency for omnidirectional motion because a significant component of contact force is redirected through angled rollers. On an incline, the required uphill force increases, but the normal force distribution can become uneven across wheels. When combined with low PWM, this makes slip more likely than steady climbing. The key insight is that the drivetrain can be torque-capable yet traction-limited, and the traction limit becomes more visible at low speed where small disturbances, roller compliance, and surface friction variability dominate. In practice, we observed that increasing PWM improved ramp success not because higher speed is inherently better, but because it increased the probability of overcoming static friction and reduced the time spent in the unstable stick-slip regime.

This analysis changed our strategy. Instead of forcing a ramp climb while wall-following, we removed the uphill segment from the wall-follow routine. We shifted to a more controlled search along the lower part of the map. In that region, low PWM was beneficial because it reduced overshoot and improved the stability of distance regulation, so our wall-follow algorithm performed very well. This tradeoff reflects a broader lesson in mechatronic design. A control strategy that is optimal for precision near walls can be incompatible with the requirements for climbing, where the dominant constraints are traction, voltage headroom, and stiction rather than kinematic accuracy.

Top hat mounting with conventional screws became a maintenance bottleneck under competition time pressure. The dowel-based clamp and drawer-style slot demonstrated that serviceability must be designed early. It also highlighted the importance of mechanical

tolerance design. A structure can be strong and repeatable in theory, but if the fit is too tight, repeated insertion increases wear and increases variability in alignment force. This experience reinforced that repeatability is not only a geometric problem. It is also a friction and tolerance problem that must be considered as part of the mechanical design.

Electrical Design

1. System overview and design goals

The robot uses an ESP32-S3-WROOM as the main controller. The ESP32 reads localization and sensing inputs including Vive, time of flight sensors, and wheel encoders. It commands two L298N dual H-bridge driver boards to run four N20 DC gear motors and drives one DS3225 servo using PWM. The key design goals were:

1. Reliable motor drive for four N20 gear motors (two motors per L298N).
2. Clean sensor/logic signals despite switching noise from brushed DC motors.
3. Safe power distribution with appropriate voltage rails for: motors (12V), ESP32/sensors (5 V / 3.3 V), and servo (4.8–6.8 V).
4. Compatibility with top hat communication and power constraints.

2. Battery selection rationale

Four brushed DC motors can draw large transient current during acceleration, rapid corrections, and collisions. Using the N20 stall current estimate, the worst-case peak is

$$I_{\text{motors,stall}} = 4 \times 0.7\ \text{A} = 2.8\ \text{A}$$

This requires a supply that can tolerate multi-amp peaks without large voltage sag. A 4-cell 18650 pack was selected because it provides better transient behavior than AA cells under similar conditions due to lower internal resistance, and it offers higher energy density for match runtime. The higher bus voltage also helps maintain usable motor voltage under load despite wiring loss and driver loss.

Power domains

Motor power domain: A 4-cell 18650 battery pack powers both L298N boards directly. Each L298N drives one left and right motor pair. This domain is treated as the highest-current and noisiest region, so it is routed and placed to reduce EMI coupling into analog and logic circuits.

Logic power domain: The ESP32 is powered from the 5 V output of the L298N onboard regulator. The ESP32 then generates 3.3 V locally for sensors and digital IO. This approach reduces wiring complexity, but it can couple motor transients into the logic rail, so grounding and decoupling become critical to prevent brownouts and sensor glitches.

Servo power domain: The DS3225 servo is powered from a separate 4×AA battery pack. The servo can draw up to about 1.9 A stall at 5 V and about 2.3 A at 6.8 V, so isolating its supply

reduces voltage droop and ground bounce on the MCU and sensor rails. The PWM signal is generated by the ESP32, so the servo ground and ESP32 ground must be tied together, ideally at a single point, to preserve a consistent logic reference.

Top hat power and I2C reference: The top hat remains on the 18650 domain. The ESP32 connects to the top hat over I2C, so the grounds are tied to ensure valid logic levels. The robot reports WiFi usage to the top hat at 2 Hz.

3. Motor Drive Subsystem

Driver choice and electrical limits

The drivetrain uses two L298N dual full-bridge drivers. The L298N is specified for a motor supply up to 46 V and a total DC current up to 4 A across the device. For a single bridge, the datasheet rating is 2 A in DC operation, with higher short peaks allowed under specific duty conditions.

A key practical constraint is the internal saturation drop of the bipolar output stages. At $I_L = 1$ A, the typical total bridge drop is about 1.80 V. At $I_L = 2$ A, the typical total drop is about 4.9 V. This drop reduces the effective voltage delivered to the motor under load and increases driver heating.

PWM control and commutation bandwidth

Speed is controlled by PWM on the enable pins, while direction is set by the input pins. The L298N supports both input chopping and enable chopping. When PWM is applied, the current ripple and switching loss depend on frequency and load. The datasheet provides a commutation frequency range on the order of tens of kilohertz for $I_L = 2$ A, which indicates the silicon can switch quickly enough for common PWM choices used in robotics.

In our implementation, low PWM duty cycles were not transferable across tasks such as climbing ramps. This is expected from a DC motor model. Let D be the PWM duty cycle, V_S the driver supply, and $V_{\text{drop}}(I)$ the L298 bridge drop. The motor sees an approximate average voltage

$$V_{\text{avg}} \approx D(V_S - V_{\text{drop}}(I))$$

At low duty, V_{avg} may not exceed the threshold needed to overcome static friction and gravity on an incline. At near-stall conditions,

$$I \approx \frac{V_{\text{avg}}}{R_m}\tau \approx K_t I$$

so reducing D reduces stall current and stall torque almost linearly for a fixed motor resistance R_m. The large V_{drop} of L298N at higher current further compresses the usable torque margin, which makes “too-low PWM” failures more likely during climbing or pushing.

Thermal behavior and efficiency implication

Because the driver drop is large at higher current, power dissipation inside the L298N can become significant:

$$P_{\mathrm{drv}} \approx I \cdot V_{\mathrm{drop}}(I)$$

Using typical values, at $I = 1$ Aand $V_{\mathrm{drop}} \approx 1.8$ V, the dissipation is about 1.8 Wper active bridge. At $I = 2$ Aand $V_{\mathrm{drop}} \approx 4.9$ V, the dissipation is about 9.8 W, which can quickly drive the package temperature upward.

Thermally, the Multiwatt15 package has a junction-to-ambient thermal resistance of about $35\ ℃/W$ under the stated mounting condition. This indicates that sustained high current requires heatsinking and good airflow, otherwise thermal shutdown risk increases.

Module guides often state a maximum board power consumption around 20 W at $T = 75℃$, which is consistent with the need to treat high-current operation as time-limited unless thermal design is strong

Protection, wiring, and module-level caveats

We applied local decoupling. A non-inductive capacitor of about 100 nF is required from both V_S and $V_{SS}$to ground, placed as close as possible to the IC ground. If the bulk capacitor is far away, an additional smaller capacitor should be placed near the device.

4. Feedback Sensors and Signal Interfaces

Time of Flight sensors on I2C

The ToF400C module is commonly based on the ST VL53L1X family and communicates over I2C. Typical module-level characteristics include up to 4 m ranging capability, up to 50 Hz update rate, and a full field-of-view around 27 degrees.

Because identical ToF sensors often boot with the same default I2C address, multi-sensor operation requires controlled startup. The XSHUT pin can hold a sensor in shutdown so that sensors can be enabled one at a time and assigned unique addresses during initialization. A practical constraint is that the updated address is typically not persistent across power cycles, so the address assignment procedure must run at every boot.

Encoder inputs and pull-up design

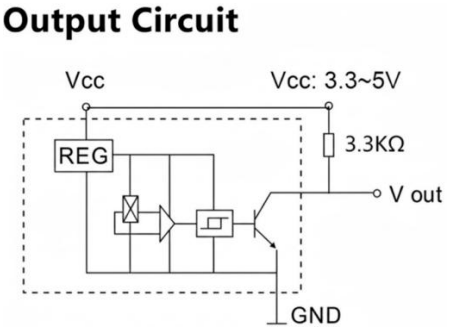


Figure 3.1 Encoder output circuit

Each wheel encoder is powered by the ESP32's 3.3 V rail. The encoder output stage is effectively an open-collector/open-drain transistor output that requires a 3.3 kΩ pull-up resistor. In our implementation, the encoder outputs are pulled up to 3.3 V so the ESP32 can read them directly without level shifting.

With open-collector outputs, the rising edge quality depends on pull-up value and wiring capacitance. Keeping runs short and using a reasonable pull-up helped reduce missed counts at higher wheel speeds.

Vive front end signal conditioning

The vive photodiode front end is sensitive to EMI and wiring parasitics because it processes small, fast optical pulses. In our circuit, a key active component is the TLV272 operational amplifier. It supports a wide supply range from 2.7 V to 16 V, has a typical bandwidth of 3 MHz, and a typical slew rate of 2.4 V per microsecond. These parameters are consistent with amplifying and shaping fast transient signals while staying within a low-power design envelope.

Given motor-induced noise, we treated the vive path as an analog subsystem. We minimized loop area, kept photodiode wiring short, routed motor currents away from the amplifier input, and ensured local decoupling near the analog components. These choices improved timing stability and reduced false triggers during aggressive motor maneuvers.

5. Servo Control

Electrical characteristics and control timing

The steering servo is a DS3225-class high-torque digital servo. Vendor datasheets for DS3225 commonly specify an operating voltage around 5.0 V to 6.8 V, stall torque up to about $24.5\ \mathrm{kg \cdot cm}$ at 6.8 V, and stall current up to about $2.3\ \mathrm{A}$ at 6.8 V.

Control is standard PWM position command. A typical specification is a pulse width range of $500\ \mu\mathrm{s}$ to $2500\ \mu\mathrm{s}$ with neutral near $1500\ \mu\mathrm{s}$. Some DS3225 datasheets also specify an operating command frequency range of about 50 Hz to 330 Hz and a deadband around $3\ \mu\mathrm{s}$.

Power integrity implication

The servo is a highly dynamic load. Near stall or during fast steering transients, its current can approach the datasheet stall current. If the servo shares a supply rail with the MCU and sensitive sensors, this step load can create voltage droop and ground bounce that causes resets or timing errors. The measured instability we observed when powering the servo from the shared battery rail is consistent with the magnitude of these current spikes.

For this reason, the final architecture used a dedicated servo supply while keeping a common ground reference for the PWM signal. This separation reduced the coupling of high di and dt events into logic rails and improved overall control stability.

6. Estimated Current Draw and Power Budget

We estimate total current by summing domain loads:

$$I_{\text{total}} = \sum_{k=1}^{n} I_k \; P = VI$$

Motor rail: With $N = 4$motors,

$$I_{\text{motors}} = N I_{\text{per motor}}$$

$$I_{\text{motors,NL}} = 4 \times 0.03\ \text{A} = 0.12\ \text{A}$$
$$I_{\text{motors,L}} = 4 \times 0.09\ \text{A} = 0.36\ \text{A}$$
$$I_{\text{motors,stall}} = 4 \times 0.7\ \text{A} = 2.8\ \text{A}$$

Because the L298N dissipates power internally, battery current under load can exceed the ideal motor current. This is one reason the 18650 pack was chosen with transient margin.

pasted

Servo rail: Using a stall current of about 2.3 A, a peak electrical power estimate is

$$I_{\text{servo,max}} \approx 2.3\ \text{A} P_{\text{servo,max}} \approx 5\ \text{V} \times 2.3\ \text{A} = 11.5\ \text{W}$$

Logic rail: The 5 V load can be expressed as

$$I_{5\text{V}} = I_{\text{ESP32}} + I_{\text{ToF}} + I_{\text{Vive}} + I_{\text{enc}} + I_{\text{misc}}$$

A conservative margin factor accounts for WiFi transmit peaks and sensor startup current:

$$I_{\text{5V,budget}} = \gamma I_{\text{5V,avg}} \gamma \in [1.5,2]$$

7. Intended and Observed Performance

Power Delivery and System Stability

Expected: The 4-cell 18650 pack was expected to handle the current spikes from four N20 motors during acceleration, turning, and collisions. The ESP32 and sensors were expected to remain stable, even when the motors changed direction quickly.

Because the servo was designed as an isolated power domain, steering commands were expected to have minimal impact on the MCU power rail.

Observed: Motor transients still affected system stability. Because the ESP32 was powered through the 5 V output on the L298N module, aggressive motor events sometimes introduced behavior consistent with voltage droop and ground noise. This did not meet the original stability expectation.

Servo behavior matched the expectation only after iteration. Early prototypes powered the servo from the same rail as the controller. Under sudden steering loads, the MCU could brown out, and the servo could become unstable after long operation. After moving the servo to a dedicated 4×AA supply, these failures disappeared and control became reliable. The final result matched the intent, but the single-battery plan did not.

Motor Drive Performance with L298N

Expected: PWM speed control was expected to be consistent and monotonic. Higher PWM should always produce stronger motion. Similar commands should produce similar behavior on the left and right sides.

Observed: The L298N introduced noticeable voltage loss and heating under higher load. On flat ground, many PWM settings behaved as expected. On ramps or during pushing, low PWM values that worked on flat terrain could fail to start motion or stall early. This was repeatable and indicated limited voltage headroom under load. This outcome was explainable, but it did

not match the expectation that the same PWM tuning would transfer across tasks.

Feedback Sensors and Signal Interfaces

Encoders

We expected clean quadrature signals and stable counting at different speeds. After adding pull-up resistors and improving wiring, encoder reading was generally reliable. Remaining errors were mostly due to mechanical slip rather than signal integrity. This matched expectations.

Time of Flight sensors on I2C

We expected stable I2C communication and consistent startup by sequencing sensors with SHUT pins. In practice, the I2C link was stable when wiring was short and grounding was solid. The main limitation was not bus failure but sensitivity to mechanical placement and environment, which could change how distance readings affected control. Electrically, this mostly matched expectations.

Vive front end

We expected robust pulse detection and stable localization across the full map. Vive was the most noise-sensitive subsystem. Early layouts produced false triggers and missed pulses under motor noise. After separating motor wiring from the Vive circuit and improving ground return paths, the signal quality improved, which matched expectation for an analog front end.

However, localization was still less stable near the edges of the map. This did not match the assumption of uniform performance across the workspace.

8. Iteration Process and Lessons Learned

The electrical system was originally intended to run from a single 4-cell 18650 pack that powered the motor drivers, the ESP32 and sensors, the TopHat, and the steering servo. In theory this simplifies wiring and reduces weight. In practice, this architecture did not meet the stability requirements, and we could not fully fix it within the project constraints. The most critical mismatch came from the servo behaving as a highly dynamic load. During rapid steering changes and near-stall conditions, the servo current increased sharply. Because the power path from the battery to the electronics had nonzero impedance from wiring, connectors, and shared return paths, these current steps produced voltage droop and ground reference movement. The expected failure mode can be summarized by

$$\Delta V \approx I_{\text{step}} \cdot R_{\text{path}}$$

where R_{path}includes battery internal resistance, cabling, module traces, and shared ground impedance. The observed symptoms were repeatable: the ESP32 could reset or briefly lose responsiveness, sensor readings could jump during combined motor and steering events, and the servo could become unstable after extended operation when it shared the same supply rail. We attempted typical mitigations such as adding bulk capacitance and improving wiring layout, but these measures only reduced high-frequency ripple and did not eliminate the large step-induced droop. A deeper limitation was that the ESP32 logic rail was sourced from the 5 V

output on the L298N module, which is not designed as a low-noise, high-transient regulator for mixed-signal loads. As a result, we did not achieve the intended single-battery design. The final, reliable solution was to change the architecture by powering the servo from a separate 4×AA pack while tying the grounds together to preserve a valid PWM reference. This improved stability significantly, but it increased mass and reduced overall mechanical efficiency. The key lesson is that "one battery for everything" requires an intentional power tree and grounding strategy, not only sufficient battery capacity.

A second mismatch was the lack of transferable PWM tuning across load conditions when using L298N driver modules. We expected PWM duty cycle to map predictably to wheel speed and tractive effort, so that values tuned on flat ground would remain usable for ramps and pushing. On flat terrain, the drivetrain generally behaved as expected. Under higher load, low PWM values that worked on flat ground could fail to start motion on a ramp or stall early. This behavior was consistent and indicated insufficient voltage headroom at the motor terminals during high current. With the L298N, the effective motor voltage is reduced by the driver voltage drop, and the average motor voltage under PWM can be approximated as

$$V_{\text{motor,avg}} \approx D(V_S - V_{\text{drop}}(I))$$

As the robot climbed a ramp, the required torque increased, motor current increased, and the L298N drop increased, further reducing the voltage applied to the motor. This created a practical dead zone at low duty cycle where the motor could not overcome static friction and gravitational load, so the system would not initiate motion. The driver also dissipates more power as current increases, which further degrades performance through heating:

$$P_{\text{drv}} \approx I \cdot V_{\text{drop}}(I)$$

We could not remove this limitation through tuning alone, because it originates from the driver and supply headroom. Instead of continuing to search for a universal PWM mapping, we changed the control strategy. We used higher minimum duty values for ramp conditions, introduced startup bias, and adjusted behavior planning to avoid operating in sustained high-load low-duty regimes. This was a workaround rather than a full fix, but it produced reliable execution under competition time pressure.

The third major mismatch was Vive localization reliability near the edges of the map. The initial expectation was approximately uniform performance across the workspace. In reality, pose readings became noticeably less stable near boundaries. We did improve failure modes that were clearly electrical. Early prototypes showed false triggers and missed pulses under motor noise. By separating motor wiring from the Vive analog front end, tightening ground return paths, and improving local decoupling, we reduced EMI coupling and improved pulse detection. However, the edge-of-map instability persisted and appeared to be dominated by signal margin rather than wiring noise alone. Near boundaries, the optical geometry can be less favorable due to incidence angle and partial occlusion, and the usable signal amplitude can approach the detection threshold. In that regime, small disturbances translate into larger timing errors and higher pose jitter. We did not have enough time to build a complete boundary-specific calibration and filtering pipeline, and we could not change the physical environment. As a result, we shifted our navigation plan to reduce dependence on high-confidence Vive readings at the

map edges, which again was an architectural workaround rather than a root-cause correction.

Overall, several outcomes did not match the original intent, and the final system relied on deliberate design changes rather than incremental tuning. The single-battery power goal failed due to servo-induced transients and shared-rail sensitivity, so we isolated the servo supply. The expectation of transferable PWM tuning failed under high load due to limited voltage headroom and driver losses, so we changed the control strategy for ramps. The expectation of uniform Vive performance failed near boundaries, so we modified navigation behavior to avoid edge-sensitive trajectories. These mismatches reinforced a central lesson: in mobile robots, electrical power integrity, driver headroom, and sensing margin are system-level constraints that often dominate controller tuning.

Processor architecture and code architecture:

1. Robot control logic:

The overall software architecture of the robot is divided into several functional modules. These include the main program, which handles autonomous driving logic, switching between manual and autonomous modes, and receiving web-based commands; motor control code with PID-based wheel speed regulation; ToF distance sensing hardware code; Vive signal acquisition code; and the web-based HTML interface. This modular design makes the code structure clearer and more compact, improves readability, and significantly reduces the difficulty of debugging.

In the final version of the system, a single ESP32-S3 board is used to implement all vehicle-side control functions. Two ToF distance sensors communicate with the ESP32-S3 via the I²C bus, while two Vive sensor reading modules are directly connected to the ESP32-S3 through GPIO interfaces. In addition, the Tophat module, which contains its own onboard MCU, communicates with the vehicle's ESP32-S3 via I²C. The ESP32-C3 is used to count the number of data packets received from the web interface within a fixed time window and transmits this information to the Tophat MCU, enabling the computation and visualization of the vehicle's health value. Figure 4.1 shows a block diagram illustrating how the MCUs and hardware components are interconnected.

We first introduce the code logic for manual control of the robot's motion via the web interface. The direction of wheel rotation and the emergency braking function are controlled using high and low digital signals (as summarized in Diagram 4.2), while the wheel speed is controlled by adjusting the PWM duty cycle.

Each motor is driven using three GPIO pins, referred to as p_forward, p_backward, and p_pwm. The p_forward and p_backward pins determine the rotation direction of the motor as well as braking behavior, while the p_pwm pin outputs a PWM signal to control the motor speed. As the PWM duty cycle increases, the absolute value of the wheel speed increases accordingly.

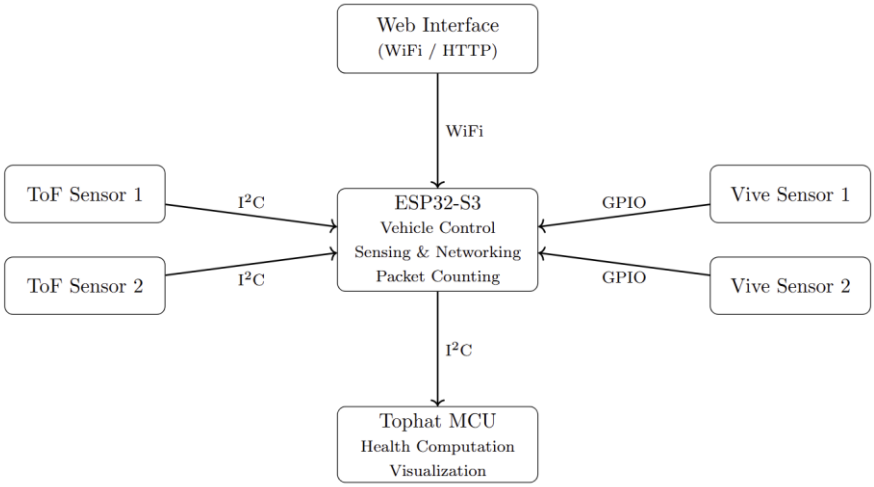


Figure 4.1 Block diagram for MCUs and sensors connection.

In this system, the PWM frequency is set to 5000 Hz, and the PWM resolution is configured to 8 bits, providing a suitable balance between speed control resolution and stable motor operation.

P_forward	P_backward	Mode
High	Low	Forward
Low	High	Backward
High	High	Break

Diagram 4.2 Logic-level combinations for controlling the rotation direction and emergency braking of a single wheel.

Considering the friction between the field surface and the wheels, the additional resistance introduced by obstacles, and the component of gravitational force acting along the slope during uphill motion, the wheel speed can be significantly disturbed. As a result, a fixed PWM command cannot always achieve the desired wheel speed. To address this issue, we designed a PID controller with feedforward compensation to dynamically adjust the PWM duty cycle, allowing the wheel speed to adaptively track and maintain a specified target value in real time.

We have applied Hall sensor to detect the rotation of the wheel and to compute the velocity. A Hall-effect sensor measures wheel speed by detecting periodic changes in a magnetic field produced by a rotating magnet or encoder attached to the wheel or motor shaft. Each magnetic transition generates a pulse, and by counting the number of pulses within a fixed time interval, the wheel's rotational speed can be calculated and converted to wheel speed using the wheel radius and gear ratio. When two Hall sensor channels are arranged with a fixed phase offset (typically 90 degrees), they form a quadrature encoder. By comparing the leading and lagging relationship between the two pulse signals, the system can determine not only the rotational speed but also the direction of rotation, enabling reliable detection of forward and reverse motion.

Each rising edge on the two encoder channels triggers an interrupt, temporarily suspending the main program to ensure accurate pulse counting. By comparing the logic levels of the two channels, the direction of wheel rotation can be determined: when one channel is high and the

other is low, the motor is rotating forward and the encoder count is incremented; when both channels are high, the motor is rotating backward and the encoder count is decremented.

The wheel speed is then computed by measuring the change in encoder counts over a fixed time window of 50 ms. Using this count difference and the effective pulses per wheel revolution, the rotational speed of the wheel is calculated according to the formula given below.

$$PPR_{wheel} = PPR * 4 * reductio\_ratio$$

$$RPM = \frac{count\_current - count\_last}{PPR_{wheel}} * \frac{60}{\Delta t}$$

PPR_{wheel} is the effective number of pulses per wheel revolution, with the factor of 4 accounting for quadrature decoding. Reduction radio of the motor is 150. Δt is the sampling time in seconds.

A feedforward-augmented PID controller is used to regulate the PWM duty cycle and control the motor output in order to achieve stable wheel speed. Through extensive experimental tuning, the final controller parameters were selected as $k_{ff} = 0.5$, $k_p = 0.6$, $k_i = 1.0$, and $k_d = 0.02$. This set of parameters enables fast and stable convergence at medium to high speeds and allows rapid speed transitions. When operating at low speeds (RPM < 50), a small amount of oscillation is observed during the initial transient phase. The duty cycle is computed using the following equation:

$$Error = RPM\_target - RPM\_current$$

$$Error\_d = \frac{Error\_current - Error\_last}{\Delta t}$$

$$Error\_i = \int_{t}^{t+\Delta t} Error\, d_t$$

$$Duty\_cycle = k_{ff} * RPM\_target + k_p * Error + k_d * Error\_d + k_i * Error\_i$$

Mecanum wheels are capable of generating force components in both the longitudinal and lateral directions. As a result, different combinations of rotation directions of the four wheels enable the robot to move forward and backward, translate left and right, and perform in-place clockwise and counterclockwise rotations. The specific correspondence between wheel rotation directions and motion modes is summarized in Table 4.3, where "+" denotes forward rotation and "−" denotes reverse rotation.

With this design, all vehicle motion control functionalities are fully implemented, enabling the robot to successfully complete all manual control tasks.

Mode	Left front	Right front	Left back	Right back
Forward	+	+	+	+
Backward	-	-	-	-
Left	-	+	+	-
Right	+	-	-	+
Rotate(clockwise)	+	-	+	-

Rotate(counter clockwise)	-	+	-	+

Diagram 4.3 Relationship between wheel rotation directions and vehicle motion.

2. Code for sensor:

For the autonomous driving component, two Vive sensors and two ToF sensors are used. The code related to the Vive sensors is based on the implementation provided by the course. A Time-of-Flight (ToF) distance sensor measures distance by emitting short infrared laser pulses toward a target and measuring the time taken for the reflected light to return to the sensor. Since the speed of light is known, the distance can be calculated from the measured round-trip time.

Two VL53L1X Time-of-Flight sensors are interfaced with the ESP32-S3 using a shared I²C bus. Since both sensors have the same default I²C address, their XSHUT (shutdown) pins are controlled by GPIOs to enable sequential initialization. At system startup, both sensors are first held in shutdown by pulling their XSHUT pins low, ensuring that no address conflict occurs on the I²C bus. Each sensor is then powered on individually by setting its corresponding XSHUT pin high. After power-up, the sensor is initialized, a communication timeout is configured, and a unique I²C address is assigned to the sensor. This process allows multiple identical ToF sensors to coexist on the same I²C bus. The sensors are subsequently configured with a medium distance mode and a fixed measurement timing budget, which provides a compromise between measurement range, accuracy, and update rate. During normal operation, distance measurements are obtained in millimeters using single-shot ranging. A small constant offset is applied to each sensor reading to compensate for systematic errors introduced by sensor mounting position and mechanical tolerances. The resulting distance values are stored and made available to the rest of the control system for obstacle detection and navigation.

3. Autonomous Task Logic of the Robot:

- **Task: Wall following**

The wall-following behavior is implemented as a two-state finite-state controller driven by two ToF distance measurements: a left-facing sensor used to regulate lateral clearance to the wall, and a front-facing sensor used for corner/obstacle detection. The controller switches between two modes:

- FOLLOW_WALL_S: normal wall tracking using a PD controller on the left distance error.
- TURN_CORNER_S: in-place turning to negotiate corners or avoid a frontal obstacle.

This separation simplifies behavior: one mode focuses on continuous tracking, while the other handles discrete corner events.

In the FOLLOW_WALL_S state, a nominal wheel speed $\mathrm{RPM}_{\mathrm{base}}$is first defined for straight-line motion, and the desired distance to the wall is set to $d_{\mathrm{target}} = 110\,\mathrm{mm}$. The robot continuously computes the error between the measured distance to the wall and the target distance. A PD controller (with $k_p = 0.5$and $k_d = 0.15$) is then used to adjust the steering of the robot in order to maintain this desired clearance. The calculation formula is given below:

$$Error = d\_left - d\_target$$

$$Error\_derivative = (Error\_current - Error\_last)/\Delta t$$

$$RPM\_turn = kp * Error + kd * Error\_derivative$$

$$RPM\_left = RPM\_base - RPM\_turn;\ RPM\_right = RPM\_base + RPM\_turn$$

The resulting control output is converted into left and right wheel speed commands, RPM_{left} and RPM_{right}, through the wheel speed controller, enabling stable straight-line motion at an approximately constant distance from the wall.

When the distance to a front wall or obstacle falls below a predefined threshold ($d_f \leq 230$ mm), the state machine transitions to the TURN_CORNER_S state. In this mode, a fixed rotational speed ($RPM_{clockwise} = 65$) is applied to perform an in-place clockwise rotation. The robot continues rotating until the front distance exceeds the clearance threshold ($d_f > 230$ mm), at which point the controller returns to the FOLLOW_WALL_S state and resumes wall following.

The main challenge in this implementation lies in tuning the PD controller gains and selecting an appropriate base wheel speed. Due to the arena layout, a sharp turn is required between the nexus and the ramp. If the base speed is set too high, the turning radius increases, preventing the robot from successfully navigating this corner. Conversely, if the base speed is set too low, the robot lacks sufficient traction to climb the ramp due to gravitational effects. To address this trade-off, a relatively low base speed was selected, and the ramp was treated as an obstacle to be avoided rather than traversed, allowing the robot to bypass the ramp and maintain robust wall-following performance.

- **Basic logic of the rest autonomous tasks:**

For the remaining autonomous driving tasks, the overall control strategy is based on real-time heading regulation, ensuring that the robot's heading remains aligned with the positive Y-direction at all times. Taking advantage of the unique capabilities of the mecanum-wheel platform, the planned path is decomposed into multiple intermediate waypoints. The robot reaches each intermediate waypoint and the final target by executing pure translations along the X-axis and Y-axis, after which it either performs an emergency stop or initiates an attack maneuver.

The readings from the two Vive sensors are denoted as (vive1_x, vive1_y) and (vive2_x, vive2_y). Using these measurements, the robot's current position and orientation can be computed as follows:

$$Position(x, y) = (\frac{vive1\_x + vive2\_x}{2}, \quad \frac{vive1\_y + vive2\_y}{2})$$

$$Orientation = arctan(\frac{vive2\_y - vive1\_y}{vive2\_x - vive1\_x})$$

Similar to the principle used for in-place rotation, a separate fixed rotational component, denoted as RPM_w, is introduced to continuously regulate the robot's heading. As a result, the

robot's motion is decomposed into three independent components: translation along the Y-axis, translation along the X-axis, and rotational motion for heading correction.

Based on the wheel rotation sign conventions corresponding to each motion mode shown in Diagram 4.3, the final wheel speeds are determined using the principle of superposition, where the contributions from translational and rotational components are combined to generate the velocity commands for all four wheels.

Left front	Right front	Left back	Right back
Rpm_y + Rpm_x – Rpm_w	Rpm_y - Rpm_x + Rpm_w	Rpm_y - Rpm_x – Rpm_w	Rpm_y + Rpm_x + Rpm_w

Diagram 4.4 Computation of the four wheel speeds while maintaining the robot's heading.

The above control logic is encapsulated into a helper function *setMecanumSpeed(rpm_y, rpm_x, rpm_w)*. When the robot moves along the Y-direction, the wheel speed command is set to (rpm_y, 0, rpm_x); when the robot moves along the X-direction, the wheel speed command is set to (0, rpm_x, rpm_y) The three velocity components are predefined constants, with rpm_y = 100, rpm_x = 100, and rpm_w = 10.

- **Task：Autonomous cover 3 vive positions**

For the task of navigating to three specified target coordinates, a fixed starting point is defined at the lower-left corner of the arena. The arena is partitioned into multiple regions, enabling the robot to automatically perform trajectory planning based on the location of the target point. Figure 4.5 illustrates the arena zoning and the corresponding path-planning strategies.

The boundaries between regions were pre-measured using the Vive sensors. Once a target coordinate is provided, the program automatically determines which zone the target lies in. The robot always starts from the same initial position $(x_{\text{start}}, y_{\text{start}})$. When the target lies in Zone 1, the controller generates an intermediate waypoint Point $1\_1 = (x_{\text{start}}, y_{\text{target}})$. The robot first translates along the positive Y-direction to this intermediate point and then moves along the negative X-direction to reach the target point $(x_{\text{start}}, y_{\text{target}})$, while continuously maintaining its heading aligned with the positive Y-direction.

For Zones 2, 3, and 4, a simple Y-then-X motion pattern cannot be directly applied due to obstructions caused by the nexus and tower, which block L-shaped paths. Therefore, appropriate intermediate waypoints were predefined based on arena geometry and experimental measurements.

For Zone 2, the robot first moves along the positive Y-direction to an intermediate waypoint Point 2_1, then translates along the negative X-direction to a second waypoint Point 2_2, and finally moves along the positive Y-direction to reach the target point. The motion strategies for Zones 3 and 4 follow a similar multi-waypoint approach and are illustrated in detail in Figure 4.5.

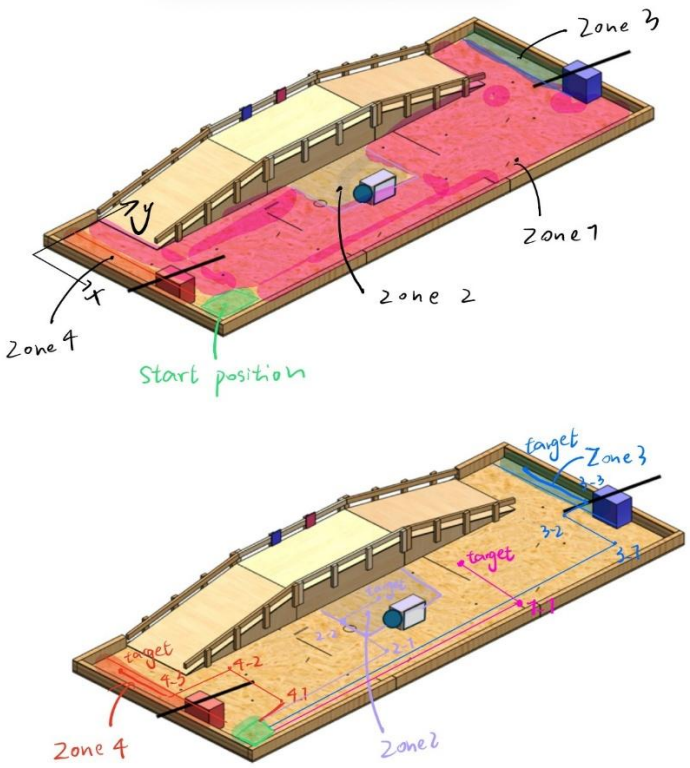


Figure 4.5 Arena partitioning and corresponding trajectory planning strategies.

- **Task: Autonomous capture low/High tower**

In **Figure 4.6**, the blue trajectory represents the path planning for the **capture low tower** task. The robot starts from the initial position and first moves along the positive Y-axis to a predefined intermediate waypoint. After reaching this point, the motion mode switches to translation along the negative X-direction, while the robot continuously regulates its heading. Upon reaching the target position, a brake mode is activated to stop the robot while facing directly toward the tower. The capture procedure is then initiated.

During the capture phase, the wheel speeds are set to a low-speed mode with $\mathrm{RPM} = 40$. The robot automatically moves forward for 6.5 seconds to ensure the button is fully pressed, after which the brake mode is engaged to hold the button for 10 seconds. Finally, the robot reverses at the same speed to return to the original target position.

The **capture high tower** task, shown as the red trajectory in Figure 4.6, follows a different strategy. This task uses two intermediate waypoints, and the robot approaches the final target by moving backward from the second waypoint. During the capture phase, the robot translates leftward for 6.5 seconds to press the button, holds the position for 10 seconds, and then moves rightward at the same speed to return to its original position.

- **Task：Autonomous attack nexus**

As shown in Figure 4.6, the robot first moves along the positive Y-direction to an intermediate

waypoint, and then translates along the negative X-direction to position itself directly in front of the bottom of the nexus. One attack cycle is defined as moving forward at 40 RPM for 6.5 s, followed by moving backward at 40 RPM for 4.5 s. This cycle is repeated four times to complete the attack task.

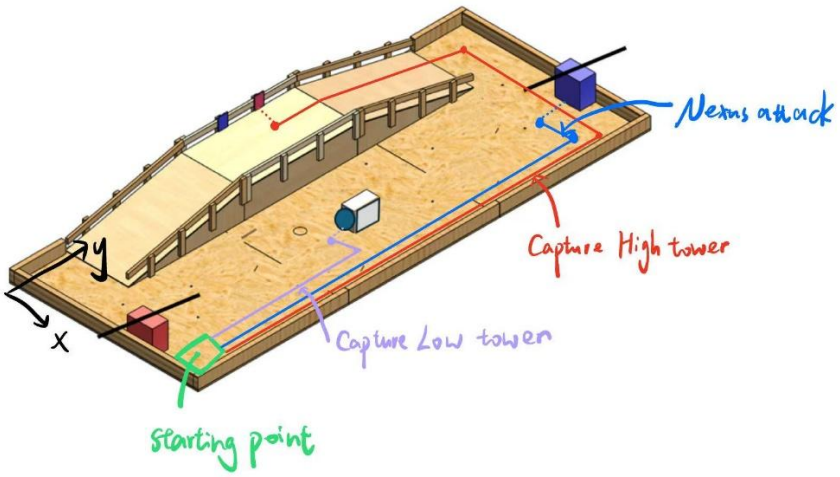


Figure 4.6 Autonomous capture low/high tower and autonomous attack nexus task logic.

Due to time constraints, the task of attacking both towers and the nexus within 45 seconds was not fully implemented. However, following the same logic as the tasks described above, this task should also be achievable with appropriate planning of intermediate waypoints and trajectories.

- **Code for servo control:**

The servo is controlled using a fixed-period PWM signal in which the desired angular position is encoded in the width of the high-level pulse rather than in the duty cycle. The servo's internal controller interprets this pulse width as a target position and drives the motor to the commanded angle using internal position feedback.

In this system, PWM signal generation is handled by the Servo library. When a target angle is specified through attackServo.write(angle), the library maps the angle command to the corresponding pulse width and outputs the appropriate control signal on the servo pin. This abstraction allows the application code to specify servo position directly in degrees without managing low-level timing.

The application logic determines how the target angle changes over time. In manual mode, the target angle is set directly by user commands, while in automatic mode it is updated periodically to produce a sweeping motion within predefined limits. The function applyServo() ensures that servo commands are updated only when the target angle changes, improving stability and reducing unnecessary signal updates.

4. Difficulty and Discussion

During the final demonstration, some of the autonomous tasks were not successfully completed, such as the high tower attack. This was primarily due to the fact that this task relied entirely on navigating to predefined Vive target points. Near the edges of the arena, the Vive position

measurements exhibited larger errors and increased instability. This instability caused unexpected robot motions when operating close to the arena boundaries, ultimately leading to task failure.

After reflection, a potential solution was identified: instead of relying solely on point-to-point navigation using Vive measurements, the robot should leverage the previously implemented wall-following logic when approaching and attacking the high tower. This strategy would allow the robot to navigate along the wall and climb the ramp with reduced sensitivity to localization noise.

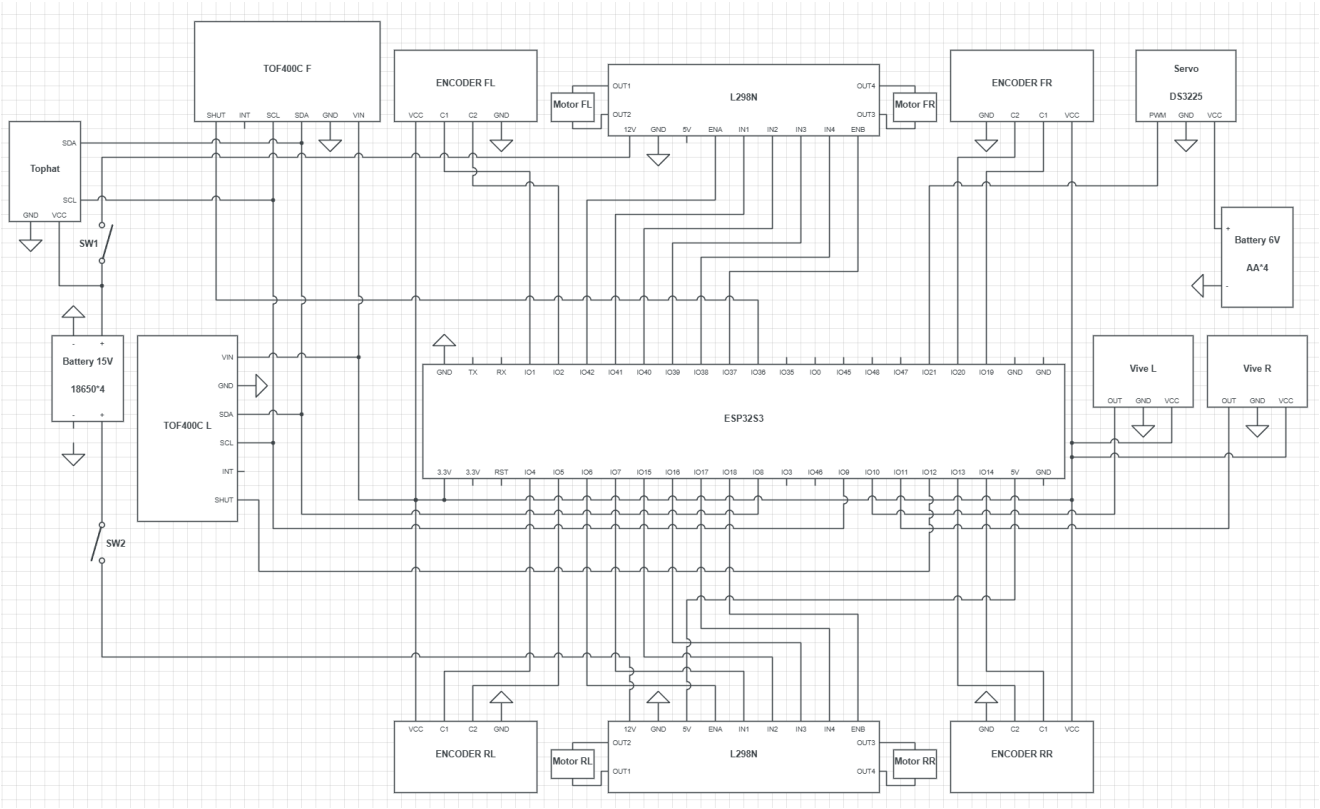
Another issue observed was that the wheel-speed PID controller did not perform well under large and rapid speed changes, resulting in continuous oscillations and occasional trajectory deviations. Further refinement of the wheel-speed controller parameters is therefore essential and is expected to significantly improve the overall performance and robustness of the robot.

Appendix

1. All CAD files

https://drive.google.com/drive/folders/1lfubCLB-lKDKERPPgyZRUV1WlLUvvosw?usp=drive_link

2. Whole robot circuit



3. N20 motor specification table

Motor Specification Table								
Model		GA12-N20 (GA12-B1215)			Rated Voltage	DC12V		
Type		DC Brushed Gear Motor			Test Voltage	DC12V		
No-Load		Rated Load				Stall		Reduction Ratio
Speed (RPM)	Current (A)	Speed (RPM)	Torque (kgf.cm)	Current (A)	Power (W)	Torque (kgf.cm)	Current (A)	
30	0.03	24	2.5	0.09	1.08	16	0.7	1000
50	0.03	40	1.6	0.075	0.9	9.8	0.58	298
60	0.03	48	1.7	0.08	0.96	11.7	0.62	298
80	0.03	64	1.8	0.085	1.02	13.6	0.66	298
100	0.03	80	1.9	0.09	1.08	15.5	0.7	298
120	0.03	96	1.6	0.09	1.08	14.6	0.7	250
140	0.03	112	0.68	0.09	1.08	11.7	0.7	200
160	0.03	128	1.4	0.09	1.08	9.5	0.7	200
200	0.03	160	0.96	0.09	1.08	6.8	0.7	150
240	0.03	192	0.46	0.08	0.96	3.6	0.67	100
300	0.03	240	0.5	0.09	1.08	4	0.7	100
400	0.03	320	0.54	0.1	1.2	4.3	0.72	100
500	0.03	400	0.56	0.11	1.32	4.6	0.74	100
600	0.03	480	0.4	0.09	1.08	3.2	0.7	50
800	0.03	640	0.48	0.1	1.2	3.6	0.73	50
1000	0.03	800	0.3	0.09	1.08	2.4	0.7	30
1200	0.03	960	0.32	0.1	1.2	2.6	0.72	30
1400	0.03	1120	0.34	0.11	1.32	2.8	0.74	30
1600	0.03	1280	0.36	0.12	1.44	3	0.76	30
2000	0.03	1600	0.38	0.13	1.56	3.2	0.78	30
2400	0.03	1920	0.12	0.07	0.84	1	0.6	10
3000	0.03	2400	0.14	0.08	0.96	1.1	0.65	10
4000	0.03	3200	0.16	0.09	1.08	1.2	0.7	10

4. DS3225 servo datasheet

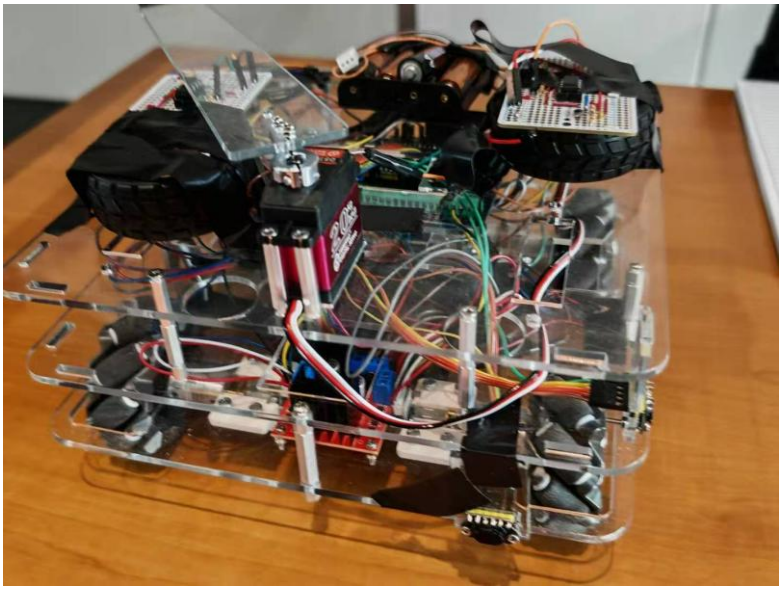
3. 电气特性 Electrical Specification

No.	工作电压 Operating Voltage	5V	6.8V
3-1	待机电流 Idle current(at stopped)	4mA	5mA
3-2	空载转速 Operating speed (at no load)	0.15 sec/60°	0.13sec/60°
3-3	堵转扭矩 Stall torque (at locked)	21 kg-cm	24.5 kg-cm
3-4	堵转电流 Stall current (at locked)	1.9A	2.3A

4. 控制特性 Control Specification

No.	Item	Specification
4-1	驱动方式 Control System	PWM(Pulse width modification)
4-2	脉宽范围 Pulse width range	500～2500μsec
4-3	中点位置 Neutral position	1500μsec
4-4	控制角度 Running degree	180° or 270° (when 500～2500 μ sec)
4-5	控制精度 Dead band width	3 μsec
4-6	控制频率 Operating frequency	50-330Hz
4-7	旋转方向 Rotating direction	Counterclockwise (when 500～2500 μsec)

5. Photo of the final mechatronic car



6. Videos

Wall following:
https://drive.google.com/file/d/1CEG6RLvStv5I9KM0bOrT0CX6WA5Tb3nV/view?usp=sharing

Vive for certain points 1 & 2:
https://drive.google.com/file/d/1jT8L_whc_qZl__XnMaFfk11uJFNiH8hB/view?usp=sharing

https://drive.google.com/file/d/1wXOrLmGSxkHy-DL_l2w9c1cxLt2o9KnS/view?usp=sharing

Attack Nexus:
https://drive.google.com/file/d/1iGlxBPty_loqXKcx_AA_0jOoG1VEoIya/view?usp=sharing

Competition Video:
https://drive.google.com/file/d/14C3Y_I_lL51oyoHde9w5WEJCiBCYxDIm/view?usp=sharing

7. BOM (Bill of Materials)

Bill of Materials

Materials	Cost
4 × N20 Encoder Motors	$47.98 (2 quantities for 4 pcs)
2 × ESP 32 S3	$21.99
2 × L298N driver	$6 (share 2 pieces with Team 7)
4 × 3.7 Volt 18650 battery and charger set	$19.99
1 × 18650 battery holder	$4.99 (1/2/3/4 slots)
4 × Mecanum wheel	$15.59
1 × 20KG Digital Servo	$13.99
4 × TOF400C	$20.89 (4pcs)
4 × 3mm to 8mm Shaft Coupling	$12.98 (2 quantity for 4 pcs)
Total Cost	$164.4 (without tax)

8. Code

Google drive link: All_set_1 (2) - Google Drive

Main.ino

```
#include <Arduino.h>
#include <WiFi.h>
#include <WebServer.h>
#include <ESP32Servo.h>
#include <Wire.h>

#include "vive_tracking.h"
#include "motor_control.h"
#include "webpage_vive_points.h"
#include "tof_sensors.h" // ⭐ Include ToF sensors

// ============================================================
```

```
//                          WiFi Access Point Configuration
// ============================================================
const char* AP_SSID = "Robot-Car-Final-Ultimate";
const char* AP_PASS = "12345678";
WebServer server(80);

// ============================================================
//                    I2C / TopHat Health System
// ============================================================
#define I2C_SLAVE_ADDR 0x28
#define SDA_PIN 8
#define SCL_PIN 9
// ⭐ Must remain at 40 kHz to match the TopHat MCU
#define I2C_FREQ 40000
const unsigned long I2C_PERIOD_MS = 500;
unsigned long lastI2CTime = 0;
uint8_t wifi_packets = 0;
uint8_t tophat_health = 255;
bool isDead = false;
bool is_blind_forward = false;                  // Whether the robot is in 1.5 s blind-forward
mode
unsigned long blind_forward_start = 0;    // Timestamp when blind-forward mode starts
const unsigned long BLIND_FORWARD_MS = 1500; // Blind forward duration: 1.5 s

// ========================== I2C Helper Functions
==========================
void send_I2C_byte(uint8_t data) {
  Wire.beginTransmission(I2C_SLAVE_ADDR);
  Wire.write(data);
  Wire.endTransmission();
}

uint8_t receive_I2C_byte() {
  uint8_t count = Wire.requestFrom(I2C_SLAVE_ADDR, (uint8_t)1);
  if (count > 0 && Wire.available()) {
    return Wire.read();
  } else {
    return tophat_health; // Keep previous value if no data is received
  }
}

// ============================================================
//                          Navigation Parameters
// ============================================================
```

```
float FINAL_X = 0.0, FINAL_Y = 0.0;
float TARGET_X = 0.0, TARGET_Y = 0.0;
float TARGET_HEADING = 0.0f;

const float POS_TOL      = 50.0f;
const float ANGLE_TOL = 0.04f;
const int MOVE_RPM              = 100;
const int CORRECTION_GAIN = 10;

// ============================================================
//                        Obstacle and Target Parameters
// ============================================================

const float TOWER_X_MIN=4280.0, TOWER_X_MAX=4988.0;
const float TOWER_Y_MIN=3612.0, TOWER_Y_MAX=4369.0,
TOWER_BYPASS_X=5226.0;
const float NEXUS_Y_LOW_MAX=1659.0, NEXUS_Y_HIGH_MIN=6415.0;
const float LOWER_RAMP_X_MAX=3697.0, LOWER_RAMP_Y_MAX=1930.0;
const float UPPER_RAMP_X_MAX=3697.0, UPPER_RAMP_Y_MIN=6210.0;
const float MID_X=4071.0, LOWER_MID_Y=1782.0, UPPER_MID_Y=6415.0;
const float RED_NEXUS_X=4530.0, RED_NEXUS_Y=1954.0;
const float BLUE_NEXUS_X=4553.0, BLUE_NEXUS_Y=6276.0;
const float LOW_TOWER_BLUE_X=4618.0, LOW_TOWER_BLUE_Y=3560.0;
const float LOW_TOWER_RED_X=4618.0, LOW_TOWER_RED_Y=4640.0;
const float HIGH_TOWER_BLUE_X=3200.0, HIGH_TOWER_BLUE_Y=3423.0;
const float HIGH_TOWER_RED_X=3200.0, HIGH_TOWER_RED_Y=4597.0;

// ============================================================
//                        State Machines and Tasks
// ============================================================
enum State { STATE_ALIGN_Y, STATE_ALIGN_X, STATE_STOP };
enum PathMode { PATH_Y_THEN_X, PATH_X_THEN_Y, PATH_LOWER_RAMP,
PATH_UPPER_RAMP };
enum HighLevelTask {
  TASK_NONE, TASK_GOTO_POINT,
  TASK_ATTACK_RED, TASK_ATTACK_BLUE,
  TASK_CAPTURE_LOW_TOWER_BLUE, TASK_CAPTURE_LOW_TOWER_RED,
  TASK_CAPTURE_HIGH_TOWER_BLUE,
TASK_CAPTURE_HIGH_TOWER_RED,
  TASK_SEQUENCE_ATTACK_ALL,
  TASK_WALL_FOLLOW // ⭐ Newly added
};

State currentState = STATE_STOP;
```

```
PathMode currentPath = PATH_Y_THEN_X;
HighLevelTask currentTask = TASK_NONE;

bool path_decided = false;
bool brake_released = false;
bool has_target = false;
int rampPhase = 0;
bool towerBypassActive = false;
bool is_intermediate_move = false;
int sequenceStep = 0;

// ============================================================
//                        Attack & Capture States
// ============================================================
enum AttackState { ATTACK_IDLE, ATTACK_FORWARD, ATTACK_BACKWARD,
ATTACK_PAUSE, ATTACK_DONE };
AttackState attackState = ATTACK_IDLE;
int attackCount = 0;
int targetAttackHits = 4;
unsigned long attackPhaseStart = 0;

const int ATTACK_RPM = 40;
const unsigned long ATTACK_FORWARD_TIME_RED    = 650;
const unsigned long ATTACK_BACKWARD_TIME_RED = 450;
const unsigned long ATTACK_FORWARD_TIME_BLUE = 650;
const unsigned long ATTACK_BACKWARD_TIME_BLUE = 450;
const unsigned long ATTACK_PAUSE_TIME      = 1000;
const int                    ATTACK_HITS_TOTAL      = 4;

enum CaptureState { CAPTURE_IDLE, CAPTURE_FORWARD, CAPTURE_HOLD,
CAPTURE_BACKWARD, CAPTURE_DONE };
enum CaptureAxis { CAPTURE_AXIS_Y, CAPTURE_AXIS_X };
CaptureState captureState = CAPTURE_IDLE;
CaptureAxis captureAxis = CAPTURE_AXIS_Y;
int captureDirToward = 0;
unsigned long capturePhaseStart = 0;

const int CAPTURE_RPM = 40;
const unsigned long CAPTURE_FORWARD_TIME    = 4000;
const unsigned long CAPTURE_BACKWARD_TIME = 450;
const unsigned long CAPTURE_HOLD_TIME         = 15000;

// ============================================================
//                        Wall-Following Parameters
```

```
// ============================================================
const int LEFT_TOF_INDEX = 1;
const int FRONT_TOF_INDEX = 0;
const float D_TARGET_MM = 110.0f;
const float BASE_DUTY = 40.0f;
const float KP_WALL = 0.5f;
const float KD_WALL = 0.15f;
const float MAX_TURN_DUTY = 30.0f;
const float MIN_FORWARD_DUTY = 20.0f;
const float FRONT_ENTER_THRESH_MM = 230.0f;
const float FRONT_CLEAR_THRESH_MM = 230.0f;
const float TURN_IN_PLACE_DUTY = 65.0f;

enum WallState { FOLLOW_WALL_S, TURN_CORNER_S };
WallState wallState = FOLLOW_WALL_S;
float prevError_WALL = 0.0f;
unsigned long lastFollowTime = 0;

// ============================================================
//                        Servo & Manual Control
// ============================================================
const int SERVO_PIN = 36;
const int SERVO_MIN_DEG = 30;
const int SERVO_MAX_DEG = 150;
const int SERVO_STEP_DEG = 5;
const unsigned long SERVO_INTERVAL_MS = 30;

Servo attackServo;
enum ServoMode { SERVO_OFF, SERVO_MANUAL, SERVO_AUTO };
ServoMode servoMode = SERVO_OFF;
float servoCurDeg = 90.0f;
int     lastServoDeg = -1; // Store last servo angle to avoid redundant updates
unsigned long lastServoUpdate = 0;
int servoDir = +1;

String curMode = "stop";
int curRPM = 0;
bool manualMode = false;

static inline bool ieq(const String& a, const String& b) { return a.equalsIgnoreCase(b); }

// ============================================================
//                        Helper Functions
// ============================================================
```

```
float angNorm(float a) {
  while (a > 3.1415926f) a -= 6.2831852f;
  while (a < -3.1415926f) a += 6.2831852f;
  return a;
}
float computeHeading() { return angNorm(atan2(vive2_y - vive1_y, vive2_x -
vive1_x)); }
float getRobotX() { return (vive1_x + vive2_x) / 2.0f; }
float getRobotY() { return (vive1_y + vive2_y) / 2.0f; }

void setMecanumSpeed(int vx, int vy, int w) {
  setTargetRPM(vy + vx - w, vy - vx + w, vy - vx - w, vy + vx + w);
}

// ---------------- Tower Collision Checking ----------------
bool verticalHitsTower(float x, float y1, float y2) {
  if (x < TOWER_X_MIN || x > TOWER_X_MAX) return false;
  float ym = min(y1, y2), yM = max(y1, y2);
  return !(yM < TOWER_Y_MIN || ym > TOWER_Y_MAX);
}
bool horizontalHitsTower(float y, float x1, float x2) {
  if (y < TOWER_Y_MIN || y > TOWER_Y_MAX) return false;
  float xm = min(x1, x2), xM = max(x1, x2);
  return !(xM < TOWER_X_MIN || xm > TOWER_X_MAX);
}
bool pathYthenX_hitsTower(float x0, float y0, float xt, float yt) {
  return verticalHitsTower(x0, y0, yt) || horizontalHitsTower(yt, x0, xt);
}
bool pathXthenY_hitsTower(float x0, float y0, float xt, float yt) {
  return horizontalHitsTower(y0, x0, xt) || verticalHitsTower(xt, y0, yt);
}
bool targetInNexusBand() { return (FINAL_Y <= NEXUS_Y_LOW_MAX) ||
(FINAL_Y >= NEXUS_Y_HIGH_MIN); }
bool lowerRampTarget() { return (FINAL_X < LOWER_RAMP_X_MAX) &&
(FINAL_Y < LOWER_RAMP_Y_MAX); }
bool upperRampTarget() { return (FINAL_X < UPPER_RAMP_X_MAX) &&
(FINAL_Y > UPPER_RAMP_Y_MIN); }
bool inTowerBandX(float x) { return (x > TOWER_X_MIN) && (x <
TOWER_X_MAX); }
bool crossTowerY(float y1, float y2) { return (y1 < TOWER_Y_MIN && y2 >
TOWER_Y_MAX) || (y1 > TOWER_Y_MAX && y2 < TOWER_Y_MIN); }

void resetNavigation(float x, float y) {
  FINAL_X = x; FINAL_Y = y;
```

```
  TARGET_X = x; TARGET_Y = y;
  has_target = true; path_decided = false; is_intermediate_move = false;
  rampPhase = 0; towerBypassActive = false;
  currentState = STATE_ALIGN_Y;
  brakeAll(); brake_released = false;
  is_blind_forward = true;
  blind_forward_start = millis();
}

// ============================================================
//                        Servo Control Logic
// ============================================================
void applyServo() {
  int target = (int)servoCurDeg;
  if (target != lastServoDeg) {
    attackServo.write(target);
    lastServoDeg = target;
  }
}

void updateServoAuto() {
  if (servoMode != SERVO_AUTO) return;
  unsigned long now = millis();
  if (now - lastServoUpdate < SERVO_INTERVAL_MS) return;
  lastServoUpdate = now;

  servoCurDeg += servoDir * SERVO_STEP_DEG;
  if (servoCurDeg >= SERVO_MAX_DEG) { servoCurDeg = SERVO_MAX_DEG;
servoDir = -1; }
  else if (servoCurDeg <= SERVO_MIN_DEG) { servoCurDeg = SERVO_MIN_DEG;
servoDir = +1; }

  applyServo();
}

void handleServo() {
  wifi_packets++;
  String mode = server.arg("mode");
  if (ieq(mode, "manual")) {
    int ang = server.arg("angle").toInt();
    servoCurDeg = constrain(ang, 0, 180);
    servoMode = SERVO_MANUAL;
    applyServo();
  } else if (ieq(mode, "auto")) {
```

```
    servoMode = SERVO_AUTO;
  } else if (ieq(mode, "stop")) {
    servoMode = SERVO_OFF;
  }
  server.send(200, "text/plain", "OK");
}

// ============================================================
//                      HTTP Handlers (Manual & Tasks)
// ============================================================
void setTargetsFromModeRPM(const String& mode, int val) {
  curMode = mode; curRPM = val;
  if (ieq(mode, "stop")) {
    brakeAll();
    setTargetRPM(0,0,0,0);
    manualMode = false;
    usePID = true; // ⭐ Restore PID when stopping
    return;
  }
  manualMode = true;
  usePID = true; // ⭐ Enable PID in manual mode
  releaseBrake();
  float rpm = constrain(val, 0, 200);
  if (ieq(mode, "forward")) setTargetRPM(rpm,rpm,rpm,rpm);
  else if (ieq(mode, "back")) setTargetRPM(-rpm,-rpm,-rpm,-rpm);
  else if (ieq(mode, "left")) setTargetRPM(-rpm,rpm,rpm,-rpm);
  else if (ieq(mode, "right")) setTargetRPM(rpm,-rpm,-rpm,rpm);
  else if (ieq(mode, "cw")) setTargetRPM(rpm,-rpm,rpm,-rpm);
  else if (ieq(mode, "ccw")) setTargetRPM(-rpm,rpm,-rpm,rpm);
}

void handleCmd() {
  wifi_packets++;
  setTargetsFromModeRPM(server.arg("mode"), server.arg("rpm").toInt());
  server.send(200, "text/plain", "OK");
}

void handleSetTarget() {
  wifi_packets++;
  currentTask = TASK_GOTO_POINT;
  usePID = true; manualMode = false;
  resetNavigation(server.arg("x").toFloat(), server.arg("y").toFloat());
  server.send(200, "text/plain", "Target Set");
}
```

```
void handleAttackRed() {
  wifi_packets++; currentTask = TASK_ATTACK_RED; targetAttackHits = 4;
  attackState = ATTACK_IDLE; attackCount = 0; manualMode = false; usePID = true;
  resetNavigation(RED_NEXUS_X, RED_NEXUS_Y);
  server.send(200, "text/plain", "Attack RED");
}

void handleAttackBlue() {
  wifi_packets++; currentTask = TASK_ATTACK_BLUE; targetAttackHits = 4;
  attackState = ATTACK_IDLE; attackCount = 0; manualMode = false; usePID = true;
  resetNavigation(BLUE_NEXUS_X, BLUE_NEXUS_Y);
  server.send(200, "text/plain", "Attack BLUE");
}

void handleAttackSequence() {
  wifi_packets++; currentTask = TASK_SEQUENCE_ATTACK_ALL; manualMode =
false; usePID = true;
  sequenceStep = 1; targetAttackHits = 1; attackState = ATTACK_IDLE; attackCount =
0;
  resetNavigation(LOW_TOWER_BLUE_X, LOW_TOWER_BLUE_Y);
  server.send(200, "text/plain", "Seq Started");
}

void handleCaptureLowBlue() {
  wifi_packets++; currentTask = TASK_CAPTURE_LOW_TOWER_BLUE;
captureState = CAPTURE_IDLE; manualMode = false; usePID = true;
  resetNavigation(LOW_TOWER_BLUE_X, LOW_TOWER_BLUE_Y);
  server.send(200, "text/plain", "Cap Low Blue");
}

void handleCaptureLowRed() {
  wifi_packets++; currentTask = TASK_CAPTURE_LOW_TOWER_RED; captureState
= CAPTURE_IDLE; manualMode = false; usePID = true;
  resetNavigation(LOW_TOWER_RED_X, LOW_TOWER_RED_Y);
  server.send(200, "text/plain", "Cap Low Red");
}

void handleCaptureHighBlue() {
  wifi_packets++; currentTask = TASK_CAPTURE_HIGH_TOWER_BLUE;
captureState = CAPTURE_IDLE; manualMode = false; usePID = true;
  resetNavigation(HIGH_TOWER_BLUE_X, HIGH_TOWER_BLUE_Y);
  server.send(200, "text/plain", "Cap High Blue");
}
```

```
void handleCaptureHighRed() {
  wifi_packets++; currentTask = TASK_CAPTURE_HIGH_TOWER_RED; captureState
= CAPTURE_IDLE; manualMode = false; usePID = true;
  resetNavigation(HIGH_TOWER_RED_X, HIGH_TOWER_RED_Y);
  server.send(200, "text/plain", "Cap High Red");
}

// ⭐ Newly added wall-following route
void handleWallFollow() {
  wifi_packets++;
  currentTask = TASK_WALL_FOLLOW;
  usePID = false; // ⭐ Disable PID and switch to open-loop control
  manualMode = false; has_target = false;

  wallState = FOLLOW_WALL_S;
  lastFollowTime = millis();
  prevError_WALL = 0.0f;

  releaseBrake(); brake_released = true;
  server.send(200, "text/plain", "Wall Follow Started");
}

void handleStop() {
  wifi_packets++; brakeAll(); setTargetRPM(0,0,0,0);
  brake_released = false; has_target = false; path_decided = false;
  towerBypassActive = false; currentState = STATE_STOP; currentTask =
TASK_NONE;
  attackState = ATTACK_IDLE; captureState = CAPTURE_IDLE; manualMode = false;
  usePID = true; // ⭐ Restore PID when stopping
  server.send(200, "text/plain", "STOP OK");
}

// ============================================================
//                        Logic Loops
// ============================================================
void updateAttack() {
  if (attackState == ATTACK_IDLE || attackState == ATTACK_DONE) return;
  unsigned long now = millis();
  bool isRed = (currentTask == TASK_ATTACK_RED || currentTask ==
TASK_CAPTURE_LOW_TOWER_RED || currentTask ==
TASK_CAPTURE_HIGH_TOWER_RED);
  if (currentTask == TASK_SEQUENCE_ATTACK_ALL) isRed = false;
  int vy = isRed ? -ATTACK_RPM : ATTACK_RPM;
```

```
  unsigned long fwdT = (currentTask == TASK_ATTACK_RED) ?
ATTACK_FORWARD_TIME_RED : ATTACK_FORWARD_TIME_BLUE;
  unsigned long bwdT = (currentTask == TASK_ATTACK_RED) ?
ATTACK_BACKWARD_TIME_RED : ATTACK_BACKWARD_TIME_BLUE;

  switch (attackState) {
    case ATTACK_FORWARD:
      if (!brake_released) { releaseBrake(); brake_released = true; }
      setMecanumSpeed(0, vy, 0);
      if (now - attackPhaseStart >= fwdT) { attackPhaseStart = now; attackState =
ATTACK_BACKWARD; }
      break;
    case ATTACK_BACKWARD:
      if (!brake_released) { releaseBrake(); brake_released = true; }
      setMecanumSpeed(0, -vy, 0);
      if (now - attackPhaseStart >= bwdT) { attackPhaseStart = now; attackState =
ATTACK_PAUSE; attackCount++; brakeAll(); brake_released = false; }
      break;
    case ATTACK_PAUSE:
      brakeAll(); brake_released = false;
      if (attackCount >= targetAttackHits) {
        attackState = ATTACK_DONE;
        if (currentTask != TASK_SEQUENCE_ATTACK_ALL) { currentTask =
TASK_NONE; has_target = false; }
      } else if (now - attackPhaseStart >= ATTACK_PAUSE_TIME) {
        attackPhaseStart = now; attackState = ATTACK_FORWARD;
      }
      break;
    default: brakeAll(); break;
  }
}

void updateCapture() {
  if (captureState == CAPTURE_IDLE || captureState == CAPTURE_DONE) return;
  unsigned long now = millis();
  int vx=0, vy=0;
  if(captureAxis==CAPTURE_AXIS_Y) vy = captureDirToward*CAPTURE_RPM;
  else vx = captureDirToward*CAPTURE_RPM;

  switch (captureState) {
    case CAPTURE_FORWARD:
      if(!brake_released){releaseBrake();brake_released=true;}
      setMecanumSpeed(vx, vy, 0);
      if(now - capturePhaseStart >= CAPTURE_FORWARD_TIME) {
```

```
            brakeAll(); brake_released=false; capturePhaseStart=now;
captureState=CAPTURE_HOLD;
          }
          break;
        case CAPTURE_HOLD:
          brakeAll(); brake_released=false;
          if(now - capturePhaseStart >= CAPTURE_HOLD_TIME) {
            capturePhaseStart=now; captureState=CAPTURE_BACKWARD;
          }
          break;
        case CAPTURE_BACKWARD:
          if(!brake_released){releaseBrake();brake_released=true;}
          setMecanumSpeed(-vx, -vy, 0);
          if(now - capturePhaseStart >= CAPTURE_BACKWARD_TIME) {
            brakeAll(); brake_released=false; captureState=CAPTURE_DONE;
currentTask=TASK_NONE; has_target=false;
          }
          break;
        default: brakeAll(); break;
    }
}

// ⭐ Wall-following logic
void updateWallFollow() {
  updateToF(); // Read distance measurements
  float d_left = tof[LEFT_TOF_INDEX];
  float d_front = tof[FRONT_TOF_INDEX];

  switch (wallState) {
    case FOLLOW_WALL_S: {
      unsigned long now = millis();
      float dt = (now - lastFollowTime) / 1000.0f;
      lastFollowTime = now;

      // Obstacle detected in front -> turn corner
      if (d_front > 0 && d_front < FRONT_ENTER_THRESH_MM) {
        wallState = TURN_CORNER_S;
        prevError_WALL = 0.0f;
        break;
      }

      // PD control
      float error = d_left - D_TARGET_MM;
      float d_term = 0.0f;
```

```
        if (dt > 0) d_term = KD_WALL * ((error - prevError_WALL) / dt);
        prevError_WALL = error;

        float turn = KP_WALL * error + d_term;
        if (turn > MAX_TURN_DUTY) turn = MAX_TURN_DUTY;
        if (turn < -MAX_TURN_DUTY) turn = -MAX_TURN_DUTY;

        float dL = BASE_DUTY - turn;
        float dR = BASE_DUTY + turn;

        // Clamp duty range
        if (dL >= 0) dL = max(dL, MIN_FORWARD_DUTY);
        if (dR >= 0) dR = max(dR, MIN_FORWARD_DUTY);
        dL = constrain(dL, -100.0f, 100.0f);
        dR = constrain(dR, -100.0f, 100.0f);

        // Write to motor targets (used as duty values here)
        targetL = dL; targetL2 = dL;
        targetR = dR; targetR2 = dR;
        break;
      }
      case TURN_CORNER_S: {
        targetL = TURN_IN_PLACE_DUTY; targetL2 = TURN_IN_PLACE_DUTY;
        targetR = -TURN_IN_PLACE_DUTY; targetR2 = -TURN_IN_PLACE_DUTY;

        if (d_front > FRONT_CLEAR_THRESH_MM) {
          wallState = FOLLOW_WALL_S;
          lastFollowTime = millis();
        }
        break;
      }
    }
}

// ============================================================
//                        SETUP & LOOP
// ============================================================
void setup() {
  Serial.begin(115200);
  delay(300);

  // Initialize servo
  ESP32PWM::allocateTimer(3);
  attackServo.setPeriodHertz(50);
```

```
  attackServo.attach(SERVO_PIN, 500, 2500);
  attackServo.write(90);
  lastServoDeg = 90;

  // ⭐ Initialize I2C (40 kHz)
  Wire.begin(SDA_PIN, SCL_PIN, I2C_FREQ);

  initVives();
  initToF(); // Wire.begin is NOT called internally
  initMotors();
  brakeAll();

  WiFi.mode(WIFI_AP);
  WiFi.softAP(AP_SSID, AP_PASS);

  server.on("/", []{ server.send_P(200, "text/html", PAGE_VIVE_POINTS); });
  server.on("/set_target", handleSetTarget);
  server.on("/attack_red", handleAttackRed);
  server.on("/attack_blue", handleAttackBlue);
  server.on("/attack_sequence", handleAttackSequence);
  server.on("/wall_follow", handleWallFollow); // ⭐ Register route
  server.on("/capture_low_blue", handleCaptureLowBlue);
  server.on("/capture_low_red", handleCaptureLowRed);
  server.on("/capture_high_blue", handleCaptureHighBlue);
  server.on("/capture_high_red", handleCaptureHighRed);
  server.on("/stop", handleStop);
  server.on("/cmd", handleCmd);
  server.on("/servo", handleServo);
  server.begin();

  Serial.println("System Ready.");
}

void loop() {
  server.handleClient();

  // ==================== I2C Health Check ======================
  if (millis() - lastI2CTime >= I2C_PERIOD_MS) {
    lastI2CTime = millis();
    send_I2C_byte(wifi_packets);
    tophat_health = receive_I2C_byte();
    wifi_packets = 0;

    // 💀 Death detection
```

```
    if (tophat_health == 0) {
        if (!isDead) {
            Serial.println("💀 DEAD!");
            isDead = true;
            brakeAll(); brake_released = false; servoMode = SERVO_OFF;
            currentTask = TASK_NONE; currentState = STATE_STOP;
            manualMode = false; has_target = false;
            attackState = ATTACK_IDLE; captureState = CAPTURE_IDLE;
            usePID = true; // Restore PID on death
        }
    } else {
        if (isDead) { Serial.println("✨ ALIVE!"); isDead = false; }
    }
}

// ⭐ Death lock
if (isDead) {
    brakeAll();
    return;
}

// ==================== Normal Control ======================
updateVives();
updateMotorControl();
updateServoAuto();

if (manualMode) return;

if (is_blind_forward) {
    // Check if still within the 1.5 s blind-forward period
    if (millis() - blind_forward_start < BLIND_FORWARD_MS) {
        // 1. Release brake
        if (!brake_released) { releaseBrake(); brake_released = true; }

        // 2. Force forward motion (using MOVE_RPM = 100)
        // Order: FL, FR, RL, RR
        setTargetRPM(MOVE_RPM, MOVE_RPM, MOVE_RPM, MOVE_RPM);

        // 3. Update motor control
        updateMotorControl();
        updateVives(); // Update pose data

        return; // ⛔ Skip navigation during blind-forward mode
    } else {
```

```
            // Blind-forward period ends
            brakeAll();
            brake_released = false;
            is_blind_forward = false; // Disable blind-forward flag
        }
    }

    // Core logic: select behavior based on currentTask
    if (currentTask == TASK_WALL_FOLLOW) {
        updateWallFollow();
        return; // Skip Vive-based navigation in wall-follow mode
    }

    if (attackState != ATTACK_IDLE && attackState != ATTACK_DONE)
{ updateAttack(); return; }
    if (captureState != CAPTURE_IDLE && captureState != CAPTURE_DONE)
{ updateCapture(); return; }

    if (!has_target) { brakeAll(); return; }

    float cur_x = getRobotX();
    float cur_y = getRobotY();
    float cur_theta = computeHeading();

    if (fabsf(cur_x) < 1.0f && fabsf(cur_y) < 1.0f) return;

    // ==========================================
    // ⭐ Core logic: path planning and preprocessing
    // ==========================================
    if (!path_decided) {
        is_intermediate_move = false;
        TARGET_X = FINAL_X; TARGET_Y = FINAL_Y;

        // 1. Escape from lower area
        if (cur_y < 1883.0f) {
            TARGET_X = cur_x; TARGET_Y = 2100.0f;
            currentState = STATE_ALIGN_Y; is_intermediate_move = true;
        }
        // 2. Prevent horizontal crossing in tower zone
        else if ((cur_y >= 3766.0f && cur_y <= 4518.0f) && (FINAL_Y >= 3766.0f &&
FINAL_Y <= 4518.0f)) {
            TARGET_X = cur_x; TARGET_Y = 3600.0f;
            currentState = STATE_ALIGN_Y; is_intermediate_move = true;
        }
```

```
    // 3. General path planning
    else {
      if (!towerBypassActive && inTowerBandX(cur_x) &&
inTowerBandX(FINAL_X) && crossTowerY(cur_y, FINAL_Y)) {
        towerBypassActive = true; currentState = STATE_ALIGN_X; currentPath =
PATH_X_THEN_Y;
      } else if (upperRampTarget() || currentTask ==
TASK_CAPTURE_HIGH_TOWER_BLUE || currentTask ==
TASK_CAPTURE_HIGH_TOWER_RED) {
        currentPath = PATH_UPPER_RAMP; currentState = STATE_ALIGN_X;
      } else if (lowerRampTarget()) {
        currentPath = PATH_LOWER_RAMP; currentState = STATE_ALIGN_X;
      } else {
        if (targetInNexusBand()) {
          if (!pathXthenY_hitsTower(cur_x, cur_y, FINAL_X, FINAL_Y)) {
            currentPath = PATH_X_THEN_Y; currentState = STATE_ALIGN_X;
          } else {
            currentPath = PATH_Y_THEN_X; currentState = STATE_ALIGN_Y;
          }
        } else {
          if (!pathYthenX_hitsTower(cur_x, cur_y, FINAL_X, FINAL_Y)) {
            currentPath = PATH_Y_THEN_X; currentState = STATE_ALIGN_Y;
          } else {
            currentPath = PATH_X_THEN_Y; currentState = STATE_ALIGN_X;
          }
        }
      }
    }
    path_decided = true; rampPhase = 0;
  }

  // State-machine-based motion execution
  float theta_err = angNorm(TARGET_HEADING - cur_theta);
  int w_correction = (fabsf(theta_err) > ANGLE_TOL) ?
                     (theta_err > 0 ? CORRECTION_GAIN : -
CORRECTION_GAIN) : 0;

  switch(currentState) {
    case STATE_ALIGN_Y: {
      float y_dest = TARGET_Y;
      if (!is_intermediate_move) {
        if
((currentPath==PATH_LOWER_RAMP||currentPath==PATH_UPPER_RAMP)) {
          if(rampPhase==1)
```

```
y_dest=(currentPath==PATH_LOWER_RAMP)?LOWER_MID_Y:UPPER_MID_Y;
                else if(rampPhase==3) y_dest=TARGET_Y;
            }
        }
        float dy = y_dest - cur_y;
        if (fabsf(dy) < POS_TOL) {
          brakeAll(); brake_released=false; delay(50);
          if(is_intermediate_move) { path_decided=false; is_intermediate_move=false; }
          else if (currentPath==PATH_Y_THEN_X) currentState=STATE_ALIGN_X;
          else if
(currentPath==PATH_LOWER_RAMP||currentPath==PATH_UPPER_RAMP) {
              if(rampPhase==1) {rampPhase=2; currentState=STATE_ALIGN_X;}
              else if(rampPhase==3) currentState=STATE_STOP;
          } else currentState=STATE_STOP;
        } else {
          if(!brake_released){releaseBrake();brake_released=true;}
          setMecanumSpeed(0, (dy>0)?MOVE_RPM:-MOVE_RPM, w_correction);
        }
    } break;

    case STATE_ALIGN_X: {
        float x_dest = TARGET_X;
        if (!is_intermediate_move) {
            if
(currentPath==PATH_LOWER_RAMP||currentPath==PATH_UPPER_RAMP) {
              if(rampPhase==0) x_dest=MID_X;
              else if(rampPhase==2) x_dest=TARGET_X;
            }
            if (towerBypassActive) x_dest=TOWER_BYPASS_X;
        }
        float dx = x_dest - cur_x;
        if (fabsf(dx) < POS_TOL) {
          brakeAll(); brake_released=false; delay(50);
          if(towerBypassActive) {towerBypassActive=false; path_decided=false;}
          else
if(currentPath==PATH_LOWER_RAMP||currentPath==PATH_UPPER_RAMP) {
              if(rampPhase==0) {rampPhase=1; currentState=STATE_ALIGN_Y;}
              else if(rampPhase==2) {rampPhase=3; currentState=STATE_ALIGN_Y;}
          } else {
              if (currentPath==PATH_X_THEN_Y) currentState=STATE_ALIGN_Y;
              else currentState=STATE_STOP;
          }
        } else {
          if(!brake_released){releaseBrake();brake_released=true;}
```

```
            setMecanumSpeed((dx>0)?MOVE_RPM:-MOVE_RPM, 0, w_correction);
        }
    } break;

    case STATE_STOP:
        brakeAll(); setTargetRPM(0,0,0,0);
        brake_released=false; has_target=false; path_decided=false;

        // Trigger tasks
        if (currentTask == TASK_ATTACK_RED || currentTask ==
TASK_ATTACK_BLUE) {
            attackState = ATTACK_FORWARD; attackPhaseStart = millis();
        }
        else if (currentTask == TASK_SEQUENCE_ATTACK_ALL) {
            if (attackState == ATTACK_IDLE) {
                attackState = ATTACK_FORWARD; attackPhaseStart = millis();
            }
            else if (attackState == ATTACK_DONE) {
                if (sequenceStep == 1) {
                    sequenceStep=2; targetAttackHits=1;
                    attackState=ATTACK_IDLE; resetNavigation(BLUE_NEXUS_X,
BLUE_NEXUS_Y);
                }
                else if (sequenceStep == 2) {
                    sequenceStep=3; targetAttackHits=1;
                    attackState=ATTACK_IDLE;
resetNavigation(HIGH_TOWER_BLUE_X, HIGH_TOWER_BLUE_Y);
                }
                else if (sequenceStep == 3) {
                    currentTask=TASK_NONE; attackState=ATTACK_IDLE;
                }
            }
        }
        else if (currentTask >= TASK_CAPTURE_LOW_TOWER_BLUE &&
currentTask <= TASK_CAPTURE_HIGH_TOWER_RED) {
            if(currentTask==TASK_CAPTURE_LOW_TOWER_BLUE ||
currentTask==TASK_CAPTURE_LOW_TOWER_RED) {
                captureAxis=CAPTURE_AXIS_Y;
                captureDirToward=(currentTask==TASK_CAPTURE_LOW_TOWER_BL
UE)?1:-1;
            } else {
                captureAxis=CAPTURE_AXIS_X; captureDirToward=-1;
            }
            captureState=CAPTURE_FORWARD; capturePhaseStart=millis();
```

```
        } else currentTask=TASK_NONE;
        break;
  }
}
```

Motor_control.h

```
#pragma once
#include <Arduino.h>

struct MotorPins {
  int pinPWM;
  int pinF;
  int pinB;
  int pinA;
  int pinB_enc;
};

extern MotorPins FL_pins;
extern MotorPins FR_pins;
extern MotorPins RL_pins;
extern MotorPins RR_pins;

extern bool usePID;

extern volatile long L_encoderCount, R_encoderCount, L2_encoderCount,
R2_encoderCount;
extern float targetL, targetR, targetL2, targetR2;
extern float rpmL, rpmR, rpmL2, rpmR2;

void initMotors();
void updateMotorControl();

void setTargetRPM(float fl, float fr, float rl, float rr);
void brakeAll();
void releaseBrake();
```

Motor_control.cpp

```
#include "motor_control.h"

// ================== Motor pin definitions ==================
MotorPins FL_pins = {42, 41, 40, 2, 1};
MotorPins FR_pins = {37, 39, 38, 20, 19};
MotorPins RL_pins = {6, 7, 15, 5, 4};
MotorPins RR_pins = {18, 16, 17, 14, 13};
```

```
// ================== PWM parameters ==================
static const int pwm_freq = 5000;
static const int pwm_res    = 8;

// ================== Encoder parameters ==================
static const int PPR = 7;
static const int reduction_ratio = 150;
static const int PPR_WHEEL = PPR * reduction_ratio * 4;

// ================== Debounce ==================
static const unsigned long debounce_us = 0;

// ================== PID enable / disable ( ⭐ only new addition)
==================
bool usePID = true;

// ================== Encoder variables ==================
volatile long L_encoderCount = 0;
volatile long R_encoderCount = 0;
volatile long L2_encoderCount = 0;
volatile long R2_encoderCount = 0;

volatile unsigned long L_lastPulse = 0;
volatile unsigned long R_lastPulse = 0;
volatile unsigned long L2_lastPulse = 0;
volatile unsigned long R2_lastPulse = 0;

// ================== Target wheel speeds ==================
float targetL    = 0.0, targetR    = 0.0;
float targetL2 = 0.0, targetR2 = 0.0;

// ================== Measured wheel speeds ==================
float rpmL    = 0.0, rpmR    = 0.0;
float rpmL2 = 0.0, rpmR2 = 0.0;

// ================== PID intermediate variables ==================
float errorL = 0, errorR = 0;
float errorL2 = 0, errorR2 = 0;

float prevErrorL = 0, prevErrorR = 0;
float prevErrorL2 = 0, prevErrorR2 = 0;

float errorSumL = 0, errorSumR = 0;
float errorSumL2 = 0, errorSumR2 = 0;
```

```
int dutyL = 0, dutyR = 0;
int dutyL2 = 0, dutyR2 = 0;

const float ERROR_SUM_MAX = 30.0;
float Kp = 0.6, Ki = 1.0, Kd = 0.02;

// ================== Timing ==================
unsigned long lastPIDTime = 0;
unsigned long lastRPMTime = 0;

// ISR declarations
void IRAM_ATTR L_ISR_A(); void IRAM_ATTR L_ISR_B();
void IRAM_ATTR R_ISR_A(); void IRAM_ATTR R_ISR_B();
void IRAM_ATTR L2_ISR_A(); void IRAM_ATTR L2_ISR_B();
void IRAM_ATTR R2_ISR_A(); void IRAM_ATTR R2_ISR_B();

// ============================================================
//    Low-level motor output (internal use only)
// ============================================================
static void applyMotorRaw(int dutyL, int dutyR, int dutyL2, int dutyR2)
{
    int maxDuty = (1 << pwm_res) - 1;
    int pwmL    = map(abs(dutyL),    0, 100, 0, maxDuty);
    int pwmR    = map(abs(dutyR),    0, 100, 0, maxDuty);
    int pwmL2 = map(abs(dutyL2), 0, 100, 0, maxDuty);
    int pwmR2 = map(abs(dutyR2), 0, 100, 0, maxDuty);

    // === Front-left motor ===
    if (dutyL > 0) {
        digitalWrite(FL_pins.pinF, HIGH); digitalWrite(FL_pins.pinB, LOW);
    } else if (dutyL < 0) {
        digitalWrite(FL_pins.pinF, LOW); digitalWrite(FL_pins.pinB, HIGH);
    } else {
        digitalWrite(FL_pins.pinF, LOW); digitalWrite(FL_pins.pinB, LOW);
    }
    ledcWrite(FL_pins.pinPWM, pwmL);

    // === Front-right motor ===
    if (dutyR > 0) {
        digitalWrite(FR_pins.pinF, HIGH); digitalWrite(FR_pins.pinB, LOW);
    } else if (dutyR < 0) {
        digitalWrite(FR_pins.pinF, LOW); digitalWrite(FR_pins.pinB, HIGH);
    } else {
```

```
        digitalWrite(FR_pins.pinF, LOW); digitalWrite(FR_pins.pinB, LOW);
    }
    ledcWrite(FR_pins.pinPWM, pwmR);

    // === Rear-left motor ===
    if (dutyL2 > 0) {
        digitalWrite(RL_pins.pinF, HIGH); digitalWrite(RL_pins.pinB, LOW);
    } else if (dutyL2 < 0) {
        digitalWrite(RL_pins.pinF, LOW); digitalWrite(RL_pins.pinB, HIGH);
    } else {
        digitalWrite(RL_pins.pinF, LOW); digitalWrite(RL_pins.pinB, LOW);
    }
    ledcWrite(RL_pins.pinPWM, pwmL2);

    // === Rear-right motor ===
    if (dutyR2 > 0) {
        digitalWrite(RR_pins.pinF, HIGH); digitalWrite(RR_pins.pinB, LOW);
    } else if (dutyR2 < 0) {
        digitalWrite(RR_pins.pinF, LOW); digitalWrite(RR_pins.pinB, HIGH);
    } else {
        digitalWrite(RR_pins.pinF, LOW); digitalWrite(RR_pins.pinB, LOW);
    }
    ledcWrite(RR_pins.pinPWM, pwmR2);
}

// ============================================================
//   Initialization
// ============================================================
void initMotors()
{
    MotorPins* M[4] = { &FL_pins, &FR_pins, &RL_pins, &RR_pins };

    for (int i = 0; i < 4; i++)
    {
        pinMode(M[i]->pinF, OUTPUT);
        pinMode(M[i]->pinB, OUTPUT);
        pinMode(M[i]->pinA, INPUT_PULLUP);
        pinMode(M[i]->pinB_enc, INPUT_PULLUP);
        ledcAttach(M[i]->pinPWM, pwm_freq, pwm_res);
    }

    attachInterrupt(digitalPinToInterrupt(FL_pins.pinA), L_ISR_A, CHANGE);
    attachInterrupt(digitalPinToInterrupt(FL_pins.pinB_enc), L_ISR_B, CHANGE);
    attachInterrupt(digitalPinToInterrupt(FR_pins.pinA), R_ISR_A, CHANGE);
```

```
    attachInterrupt(digitalPinToInterrupt(FR_pins.pinB_enc), R_ISR_B, CHANGE);
    attachInterrupt(digitalPinToInterrupt(RL_pins.pinA), L2_ISR_A, CHANGE);
    attachInterrupt(digitalPinToInterrupt(RL_pins.pinB_enc), L2_ISR_B, CHANGE);
    attachInterrupt(digitalPinToInterrupt(RR_pins.pinA), R2_ISR_A, CHANGE);
    attachInterrupt(digitalPinToInterrupt(RR_pins.pinB_enc), R2_ISR_B, CHANGE);
}

// ============================================================
//   STOP: active electronic braking
// ============================================================
void brakeAll()
{
    // ⭐ To prevent PID integral windup during braking,
    // the integrators are cleared here.
    // PID is NOT permanently disabled; releaseBrake() restores normal operation.

    digitalWrite(FL_pins.pinF, HIGH); digitalWrite(FL_pins.pinB, HIGH);
    digitalWrite(FR_pins.pinF, HIGH); digitalWrite(FR_pins.pinB, HIGH);
    digitalWrite(RL_pins.pinF, HIGH); digitalWrite(RL_pins.pinB, HIGH);
    digitalWrite(RR_pins.pinF, HIGH); digitalWrite(RR_pins.pinB, HIGH);

    ledcWrite(FL_pins.pinPWM, 0);
    ledcWrite(FR_pins.pinPWM, 0);
    ledcWrite(RL_pins.pinPWM, 0);
    ledcWrite(RR_pins.pinPWM, 0);

    targetL = targetR = targetL2 = targetR2 = 0;
    errorSumL = errorSumR = errorSumL2 = errorSumR2 = 0;
}

// ============================================================
//   Release STOP (brake)
// ============================================================
void releaseBrake()
{
    // Restore neutral state; actual motion is handled by updateMotorControl()
    digitalWrite(FL_pins.pinF, LOW); digitalWrite(FL_pins.pinB, LOW);
    digitalWrite(FR_pins.pinF, LOW); digitalWrite(FR_pins.pinB, LOW);
    digitalWrite(RL_pins.pinF, LOW); digitalWrite(RL_pins.pinB, LOW);
    digitalWrite(RR_pins.pinF, LOW); digitalWrite(RR_pins.pinB, LOW);
}

// ============================================================
//   Set target RPM
```

```
// ============================================================
void setTargetRPM(float fl, float fr, float rl, float rr)
{
    targetL  = fl;
    targetR  = fr;
    targetL2 = rl;
    targetR2 = rr;
}

// ============================================================
//  Main motor control update
//  (⭐ key modification: PID / open-loop switch)
// ============================================================
void updateMotorControl()
{
    // ⭐ Mode B: open-loop control (wall following)
    if (!usePID) {
        // Interpret target values directly as duty cycles (-100 ~ 100)
        dutyL  = constrain((int)targetL,  -100, 100);
        dutyR  = constrain((int)targetR,  -100, 100);
        dutyL2 = constrain((int)targetL2, -100, 100);
        dutyR2 = constrain((int)targetR2, -100, 100);

        applyMotorRaw(dutyL, dutyR, dutyL2, dutyR2);
        return;
    }

    // ⭐ Mode A: closed-loop PID control (original logic preserved)
    unsigned long now = millis();

    // ======== Update RPM (every 200 ms) ========
    if (now - lastRPMTime >= 200)
    {
        float dt = (now - lastRPMTime) / 1000.0;
        lastRPMTime = now;

        static long lastCountL = 0, lastCountR = 0, lastCountL2 = 0, lastCountR2 = 0;

        noInterrupts();
        long cL = L_encoderCount;
        long cR = R_encoderCount;
        long cL2 = L2_encoderCount;
        long cR2 = R2_encoderCount;
        interrupts();
```

```
        rpmL  = ((cL  - lastCountL ) / (float)PPR_WHEEL) * (60.0 / dt);
        rpmR  = ((cR  - lastCountR ) / (float)PPR_WHEEL) * (60.0 / dt);
        rpmL2 = ((cL2 - lastCountL2) / (float)PPR_WHEEL) * (60.0 / dt);
        rpmR2 = ((cR2 - lastCountR2) / (float)PPR_WHEEL) * (60.0 / dt);

        lastCountL  = cL;
        lastCountR  = cR;
        lastCountL2 = cL2;
        lastCountR2 = cR2;
    }

    // ======== PID update (every 50 ms) ========
    if (now - lastPIDTime >= 50)
    {
        float dt = (now - lastPIDTime) / 1000.0;
        lastPIDTime = now;

        errorL  = targetL  - rpmL;
        errorR  = targetR  - rpmR;
        errorL2 = targetL2 - rpmL2;
        errorR2 = targetR2 - rpmR2;

        const float Kff = 0.5;
        float uLff  = Kff * targetL;
        float uRff  = Kff * targetR;
        float uL2ff = Kff * targetL2;
        float uR2ff = Kff * targetR2;

        errorSumL  = constrain(errorSumL  + errorL  * dt, -ERROR_SUM_MAX,
ERROR_SUM_MAX);
        errorSumR  = constrain(errorSumR  + errorR  * dt, -ERROR_SUM_MAX,
ERROR_SUM_MAX);
        errorSumL2 = constrain(errorSumL2 + errorL2 * dt, -ERROR_SUM_MAX,
ERROR_SUM_MAX);
        errorSumR2 = constrain(errorSumR2 + errorR2 * dt, -ERROR_SUM_MAX,
ERROR_SUM_MAX);

        float dL  = (errorL  - prevErrorL)  / dt;
        float dR  = (errorR  - prevErrorR)  / dt;
        float dL2 = (errorL2 - prevErrorL2) / dt;
        float dR2 = (errorR2 - prevErrorR2) / dt;

        prevErrorL  = errorL;
```

```
            prevErrorR    = errorR;
            prevErrorL2 = errorL2;
            prevErrorR2 = errorR2;

            float uL    = uLff    + Kp * errorL    + Ki * errorSumL    + Kd * dL;
            float uR    = uRff    + Kp * errorR    + Ki * errorSumR    + Kd * dR;
            float uL2 = uL2ff + Kp * errorL2 + Ki * errorSumL2 + Kd * dL2;
            float uR2 = uR2ff + Kp * errorR2 + Ki * errorSumR2 + Kd * dR2;

            dutyL    = constrain((int)uL,    -100, 100);
            dutyR    = constrain((int)uR,    -100, 100);
            dutyL2 = constrain((int)uL2, -100, 100);
            dutyR2 = constrain((int)uR2, -100, 100);

            applyMotorRaw(dutyL, dutyR, dutyL2, dutyR2);
        }
}

// ================== ISR section (unchanged) ==================
void IRAM_ATTR L_ISR_A() {
    unsigned long now = micros(); if (now - L_lastPulse < debounce_us) return;
    L_lastPulse = now;
    int a = digitalRead(FL_pins.pinA);
    int b = digitalRead(FL_pins.pinB_enc);
    if (a == b) L_encoderCount++; else L_encoderCount--;
}
void IRAM_ATTR L_ISR_B() {
    unsigned long now = micros(); if (now - L_lastPulse < debounce_us) return;
    L_lastPulse = now;
    int a = digitalRead(FL_pins.pinA);
    int b = digitalRead(FL_pins.pinB_enc);
    if (a != b) L_encoderCount++; else L_encoderCount--;
}

void IRAM_ATTR R_ISR_A() {
    unsigned long now = micros(); if (now - R_lastPulse < debounce_us) return;
    R_lastPulse = now;
    int a = digitalRead(FR_pins.pinA);
    int b = digitalRead(FR_pins.pinB_enc);
    if (a == b) R_encoderCount++; else R_encoderCount--;
}
void IRAM_ATTR R_ISR_B() {
    unsigned long now = micros(); if (now - R_lastPulse < debounce_us) return;
    R_lastPulse = now;
```

```
    int a = digitalRead(FR_pins.pinA);
    int b = digitalRead(FR_pins.pinB_enc);
    if (a != b) R_encoderCount++; else R_encoderCount--;
}

void IRAM_ATTR L2_ISR_A() {
    unsigned long now = micros(); if (now - L2_lastPulse < debounce_us) return;
    L2_lastPulse = now;
    int a = digitalRead(RL_pins.pinA);
    int b = digitalRead(RL_pins.pinB_enc);
    if (a == b) L2_encoderCount++; else L2_encoderCount--;
}
void IRAM_ATTR L2_ISR_B() {
    unsigned long now = micros(); if (now - L2_lastPulse < debounce_us) return;
    L2_lastPulse = now;
    int a = digitalRead(RL_pins.pinA);
    int b = digitalRead(RL_pins.pinB_enc);
    if (a != b) L2_encoderCount++; else L2_encoderCount--;
}

void IRAM_ATTR R2_ISR_A() {
    unsigned long now = micros(); if (now - R2_lastPulse < debounce_us) return;
    R2_lastPulse = now;
    int a = digitalRead(RR_pins.pinA);
    int b = digitalRead(RR_pins.pinB_enc);
    if (a == b) R2_encoderCount++; else R2_encoderCount--;
}
void IRAM_ATTR R2_ISR_B() {
    unsigned long now = micros(); if (now - R2_lastPulse < debounce_us) return;
    R2_lastPulse = now;
    int a = digitalRead(RR_pins.pinA);
    int b = digitalRead(RR_pins.pinB_enc);
    if (a != b) R2_encoderCount++; else R2_encoderCount--;
}
```

tof_sensors.h

```
#pragma once
#include <Wire.h>
#include "VL53L1X.h"

void initToF();
void updateToF();

extern int tof[2];      // tof[0] ~ tof[2]
```

tof_sensors.cpp

```
#include "tof_sensors.h"
#include <Wire.h>

VL53L1X tof1, tof2;

#define XSHUT1 47
#define XSHUT2 21

int tof[2] = {0, 0};

void initXShutPins() {
  pinMode(XSHUT1, OUTPUT);
  pinMode(XSHUT2, OUTPUT);
}

void allOff() {
  digitalWrite(XSHUT1, LOW);
  digitalWrite(XSHUT2, LOW);
}

bool initOne(VL53L1X &sensor, int xshutPin, uint8_t newAddr) {
  digitalWrite(xshutPin, HIGH);
  delay(10);

  sensor.setTimeout(500);
  if (!sensor.init()) {
     Serial.println("VL53L1X init failed!");
     return false;
  }

  sensor.setAddress(newAddr);

  sensor.setDistanceMode(VL53L1X::Medium);
  sensor.setMeasurementTimingBudget(50000);
  return true;
}

void initToF() {

  initXShutPins();
  allOff();
```

```
  delay(50);

  if (!initOne(tof1, XSHUT1, 0x2A)) return;
  if (!initOne(tof2, XSHUT2, 0x2B)) return;

  Serial.println("All 2 ToF sensors initialized.");
}

void updateToF() {
  tof[0] = tof1.readRangeSingleMillimeters()-16;
  tof[1] = tof2.readRangeSingleMillimeters()-10;
}
```

Vive_510.cpp

```
/*
 * MEAM510 hacks for Vive Interface V2
 * Dec 2021
 * Use at your own risk
 *
 * Mark Yim
 * University of Pennsylvania
 * copyright (c) 2021 All Rights Reserved
 */

#include "vive510.h"
//#define DEBUG
//#define DEBUG2

portMUX_TYPE mux = portMUX_INITIALIZER_UNLOCKED;

// need global interrupt Arduino attachInterrupt won't take member function as interrupt
// need to pass THIS as arg since all vive object interrupts call same global interrupt
void IRAM_ATTR m_pulseISR(void *v) {
  portENTER_CRITICAL_ISR(&mux);
  static_cast<Vive510 *>(v)->pulseISR(micros());
  portEXIT_CRITICAL_ISR(&mux);
}

Vive510::Vive510(int pin) {
  m_pin = pin;
}

void IRAM_ATTR Vive510::pulseISR(uint32_t us) {
    if (digitalRead(m_pin)==HIGH) {
```

```
        m_usRising = us;
      }
      else {
        m_usFalling = us;
      }
      if (m_vivestatus == VIVE_RECEIVING)      processPulse();
}

uint16_t Vive510::xCoord(){
  return m_xCoord;
}

uint16_t Vive510::yCoord(){
  return m_yCoord;
}

int Vive510::isKPulse(uint32_t pulsewidth){
  if (pulsewidth < 75 ||
        (pulsewidth > 85 && pulsewidth < 95) ||
        (pulsewidth > 106 && pulsewidth < 117) ||
        (pulsewidth > 127 && pulsewidth < 137)
  ) return false;
  else return true;
}

int Vive510::isJPulse(uint32_t pulsewidth){
  if (pulsewidth < 75 ||
        (pulsewidth > 85 && pulsewidth < 95) ||
        (pulsewidth > 106 && pulsewidth < 117) ||
        (pulsewidth > 127 && pulsewidth < 137)
  ) return true;
  else return false;
}

// move checkflag to be backwards checking...think about whether tu link w/spuriuos
void Vive510::processPulse() {
 // static int checkflag=0;

  if (m_lastFalling != m_usFalling) {
     int pulsewidth = m_usFalling-m_usRising;

     if (pulsewidth > m_sweepWidth) {
        if (pulsewidth > 140) { //pulse too long. bad pulse
#ifdef DEBUG2
```

```
            ets_printf("P%d Spur %d width:%d \n",m_pin,m_spurious,pulsewidth);
#endif
            m_pulseType = 0;
          }
          else if (isKPulse(pulsewidth)) {
            m_pulseType = KTYPE;
#ifdef DEBUG
  ets_printf("\nKPin%d width=%d ", m_pin, pulsewidth);
#endif
          }
          else if (isJPulse(pulsewidth)) {
            m_pulseType = JTYPE;
#ifdef DEBUG
  ets_printf("\tJPin%d width=%d", m_pin, pulsewidth);
#endif
          }
      }
      else { // x sweep or y sweep
#ifdef DEBUG
          ets_printf("Pin%d:%d    r=%d", m_pin,pulsewidth,m_usRising-m_lastFalling);
#endif
          if (m_pulseType == JTYPE)    m_yCoord = m_usRising-m_lastFalling;
          if (m_pulseType == KTYPE)    m_xCoord = m_usRising-m_lastFalling;

           m_spurious = 0;
      }

      if (m_spurious++ > 60) m_vivestatus=VIVE_SYNC_ONLY;

      m_lastFalling = m_usFalling;
  }
}

void Vive510::start() {
  // use ESP32 version of attachInterrupt to allow THIS argument
  attachInterruptArg(digitalPinToInterrupt(m_pin), m_pulseISR,    this,    CHANGE);
}

void Vive510::begin() {
  pinMode(m_pin, INPUT);
  start();
}

void Vive510::begin(int pin) {
```

```
  m_pin = pin;
  pinMode(m_pin, INPUT);
  start();
}

void Vive510::stop() {
  detachInterrupt(digitalPinToInterrupt(m_pin));
}

int Vive510::status() {
  return m_vivestatus;
}

uint32_t    Vive510::sync(int reps){
  int i=0;
  uint32_t m_lastFalling;
  uint32_t startms = millis();
  m_lastFalling = m_usFalling;

  while (millis()-startms < (reps+1)*1000/120) { // count    pulses
     if (m_lastFalling != m_usFalling) {
        m_lastFalling = m_usFalling;
        i++;
     }
     yield();
  }

  if (i == 0) {
     m_vivestatus = VIVE_NO_SIGNAL; // just for debugging info
#ifdef DEBUG2
     ets_printf("no signal ");
#endif
  }
  else if (i < 2*reps) {
     m_vivestatus = VIVE_SYNC_ONLY; // just for debugging info
#ifdef DEBUG2
 ets_printf("missing some pulses %d/%d ",i,2*reps);
#endif
  }
  else {
     m_vivestatus = VIVE_RECEIVING;
#ifdef DEBUG2
  ets_printf(" Sweep and Sync received ");
#endif
```

```
  }

  return m_vivestatus;
}
```

Vive_510.h

```
/*
 * header for MEAM510 hacks for vive interface
 * May 2021
 * Use at your own risk
 *
 */

#ifndef VIVE510
#define VIVE510

#include <arduino.h>

// vive status errors
#define VIVE_NO_SIGNAL 0
#define VIVE_SYNC_ONLY 1
#define VIVE_RECEIVING 2

#define KTYPE 2
#define JTYPE 1

class Vive510
{
private:

  volatile uint32_t m_usRising ; // updated by interrupts
  volatile uint32_t m_usFalling ;
  uint16_t m_xCoord;
  uint16_t m_yCoord;
  int m_vivestatus = 0;
  int m_pin;                    // signal input pin
  int m_sweepWidth=50;
  int m_pulseType; // 1 is J, 2 is K

  uint32_t m_lastFalling;
  int m_spurious;

  int isJPulse(uint32_t pulsewidth);
  int isKPulse(uint32_t pulsewidth);
```

```
  void processPulse();

public:
  Vive510(int pin);
  uint16_t xCoord();
  uint16_t yCoord();
  uint32_t sync(int);
  int status();
  void stop();
  void start();
  void begin();
  void begin(int);

    void    pulseISR(uint32_t); // need public for global interrupt Arduino hack
};

#endif
```

vive_tracking.h

```
#pragma once
#include "vive510.h"

void initVives();
void updateVives();

extern uint16_t vive1_x, vive1_y;
extern uint16_t vive2_x, vive2_y;
```

vive_tracking.cpp

```
#include "vive_tracking.h"

#define VIVE1_PIN 10
#define VIVE2_PIN 11

Vive510 vive1(VIVE1_PIN);
Vive510 vive2(VIVE2_PIN);

uint16_t vive1_x = 0, vive1_y = 0;
uint16_t vive2_x = 0, vive2_y = 0;

static uint16_t
  x10, x11, x12, y10, y11, y12,
  x20, x21, x22, y20, y21, y22;
```

```
uint16_t med3(uint16_t a, uint16_t b, uint16_t c) {
  if (a <= b && a <= c) return (b <= c ? b : c);
  else if (b <= a && b <= c) return (a <= c ? a : c);
  else return (a <= b ? a : b);
}

void initVives() {
  vive1.begin();
  vive2.begin();
}

void processVive(Vive510 &v,
                 uint16_t &x0, uint16_t &x1, uint16_t &x2,
                 uint16_t &y0, uint16_t &y1, uint16_t &y2,
                 uint16_t &outX, uint16_t &outY)
{
  if (v.status() == VIVE_RECEIVING) {
    x2 = x1;   x1 = x0;   x0 = v.xCoord();
    y2 = y1;   y1 = y0;   y0 = v.yCoord();
    outX = med3(x0, x1, x2);
    outY = med3(y0, y1, y2);
  }
  else {
    v.sync(5);
    outX = outY = 0;
  }
}

void updateVives() {
  processVive(vive1, x10,x11,x12, y10,y11,y12, vive1_x, vive1_y);
  processVive(vive2, x20,x21,x22, y20,y21,y22, vive2_x, vive2_y);
}
```

webpage_vive_points.h

```
#pragma once

const char PAGE_VIVE_POINTS[] PROGMEM = R"rawliteral(
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Robot-Car Controller</title>
```

```
<style>
  body{
     background:#0c1222; color:#fff;
     font-family:sans-serif; margin:0; padding:16px; text-align:center;
  }
  .card{
     background:#141b33; border-radius:12px; padding:12px; margin-bottom:16px; text-
align:left;
  }
  .row{ display:flex; flex-wrap:wrap; gap:8px; margin-bottom:8px; }
  .btn{
     padding:10px 12px; border:none; border-radius:8px; cursor:pointer; color:#fff; flex-
grow:1; font-size:15px;
  }
  .btn-go{ background:#4caf50; flex-grow:0; }
  .btn-red{ background:#d32f2f; }
  .btn-blue{ background:#1976d2; }
  .btn-stop{ background:#000; width:100%; border:1px solid red; font-weight:bold; }

  .btn-seq { background: linear-gradient(90deg, #FF9800 0%, #FF5722 100%); font-
weight:bold; width:100%; }
  .btn-wall { background: linear-gradient(90deg, #9C27B0 0%, #673AB7 100%); font-
weight:bold; width:100%; }

  input[type=number]{ padding:8px; background:#111827; color:#fff; border:1px solid
#4b5563; width:80px; }



  .slider-container { margin: 10px 0; width: 100%; }
  input[type=range] { width: 100%; }
  label { font-size: 14px; color: #ccc; }

  .grid-drive{ display:grid; grid-template-columns:1fr 1fr 1fr; gap:6px; margin-
top:10px; }
  .drive{ height:50px; border-radius:8px; border:none; background:#374151; color:#fff;
font-size:16px; }
  .drive:active { background: #4b5563; }
</style>
</head>
<body>

<h1>Mission Control</h1>

<div class="card">
```

```
  <h2>Auto Tasks</h2>

  <div class="row">
    <button class="btn btn-seq" onclick="attackSequence()">⚡ ATTACK SEQUENCE ⚡</button>
  </div>

  <div class="row">
    <button class="btn btn-wall" onclick="startWallFollow()">🧱 WALL FOLLOW (Left) 🧱</button>
  </div>

  <div style="height:10px"></div>

  <div class="row">
    <button class="btn btn-red" onclick="attackRed()">Attack Red</button>
    <button class="btn btn-blue" onclick="attackBlue()">Attack Blue</button>
  </div>
  <div class="row">
    <button class="btn btn-blue" onclick="captureLowBlue()">Cap Low Blue</button>
    <button class="btn btn-red" onclick="captureLowRed()">Cap Low Red</button>
  </div>
  <div class="row">
    <button class="btn btn-blue" onclick="captureHighBlue()">Cap High Blue</button>
    <button class="btn btn-red" onclick="captureHighRed()">Cap High Red</button>
  </div>

  <hr style="border-color:#333">

  <div class="row">
    <input type="number" id="xval" placeholder="X">
    <input type="number" id="yval" placeholder="Y">
    <button class="btn btn-go" onclick="sendTarget()">GO</button>
  </div>

  <div class="row">
    <button class="btn btn-stop" onclick="stopRobot()">STOP ALL</button>
  </div>
  <div id="status">Ready</div>
</div>

<div class="card">
  <h2>Manual & Servo</h2>
```

```
  <div class="row">
    <button class="btn" style="background:#555"
onclick="fetch('/servo?mode=auto')">Servo Auto</button>
    <button class="btn" style="background:#333"
onclick="fetch('/servo?mode=stop')">Servo Stop</button>
  </div>

  <div class="slider-container">
    <label>Servo Angle: <span id="servoVal">90</span>°</label>
    <input type="range" id="servoRange" min="0" max="180" value="90">
  </div>

  <hr style="border-color:#333">

  <div class="slider-container">
    <label>Manual Speed: <span id="rpmVal">80</span> RPM</label>
    <input type="range" id="rpmRange" min="0" max="200" value="80">
  </div>

  <div class="grid-drive">
    <button class="drive" onclick="sendCmd('cw')">CW</button>
    <button class="drive" onclick="sendCmd('forward')">FWD</button>
    <button class="drive" onclick="sendCmd('ccw')">CCW</button>
    <button class="drive" onclick="sendCmd('left')">LEFT</button>
    <button class="drive" onclick="sendCmd('stop')"
style="background:#b91c1c">STOP</button>
    <button class="drive" onclick="sendCmd('right')">RIGHT</button>
    <div></div>
    <button class="drive" onclick="sendCmd('back')">BACK</button>
    <div></div>
  </div>
</div>

<script>
const BASE = "";
function setStatus(t){ document.getElementById("status").innerText=t; }

// Auto Tasks
function
attackSequence(){ fetch(BASE+"/attack_sequence").then(r=>r.text()).then(setStatus); }
function
startWallFollow(){ fetch(BASE+"/wall_follow").then(r=>r.text()).then(setStatus); }
function stopRobot(){ fetch(BASE+"/stop").then(r=>r.text()).then(setStatus); }
```

```
function sendTarget(){
  let x=document.getElementById("xval").value;
  let y=document.getElementById("yval").value;
  fetch(`${BASE}/set_target?x=${x}&y=${y}`).then(r=>r.text()).then(setStatus);
}
function attackRed(){ fetch(BASE+"/attack_red").then(r=>r.text()).then(setStatus); }
function attackBlue(){ fetch(BASE+"/attack_blue").then(r=>r.text()).then(setStatus); }
function
captureLowBlue(){ fetch(BASE+"/capture_low_blue").then(r=>r.text()).then(setStatus); }
function
captureLowRed(){ fetch(BASE+"/capture_low_red").then(r=>r.text()).then(setStatus); }
function
captureHighBlue(){ fetch(BASE+"/capture_high_blue").then(r=>r.text()).then(setStatus);
}
function
captureHighRed(){ fetch(BASE+"/capture_high_red").then(r=>r.text()).then(setStatus); }

let currentRpm = 80;
document.getElementById("rpmRange").oninput = function(e) {
  currentRpm = e.target.value;
  document.getElementById("rpmVal").innerText = currentRpm;
  // sendCmd(lastMode);
};

let lastMode = "stop";
function sendCmd(m){
  lastMode = m;
  fetch(`${BASE}/cmd?mode=${m}&rpm=${currentRpm}`);
}

document.getElementById("servoRange").oninput = function(e) {
  let val = e.target.value;
  document.getElementById("servoVal").innerText = val;
  fetch(`${BASE}/servo?mode=manual&angle=${val}`);
};

</script>
</body>
</html>
)rawliteral";
```