SCHOOL OF SCIENCE AND ENGINEERING

DEPARTMENT OF ENGINEERING

EGG 409-01: Senior Design Project I & II

Advisor: Wafi Danesh

Co-Advisor: Kerry Ford

4/26/25

Course Project:

Autonomous Robotic Delivery

Student Name	Major
Alexander Krupinski	Mechanical Engineering
Owen McGarrity	Mechanical Engineering
Mark Camitan	Electrical Engineering
Benjamin Weisfeld	Electrical Engineering
Ryan Schubert	Computer Engineering
Ryan Zhang	Computer Engineering

Abstract

The autonomous robotic delivery project focuses on designing and developing a Global Positioning System (GPS) guided remote controlled vehicle capable of autonomous food delivery on a college campus. This system addresses the need for improved time management and convenience among students and faculty by enabling users to schedule food pickups and deliveries in advance through the Delivery App coded and campus Grubhub. The vehicle will operate autonomously, navigating through a nodal network while avoiding obstacles to ensure efficiency and timely delivery. A multidisciplinary team of mechanical, electrical, and computer engineering students will collaborate on the design and integration of the car's physical frame, navigation software, and hardware systems. The Fall 2024 semester concentrated on planning, designing, and prototyping the project while the Spring 2025 semester focused on finalizing the prototype to engineer a fully functional project. The project emphasizes cross-disciplinary collaboration, real-world problem solving, and the implementation of autonomous systems in everyday scenarios.

Table of Contents

Abstract.....2

Introduction / Scope of Work.....4

Theory.....5

Procedure.....7

Results.....17

Conclusion.....19

References.....20

Appendix.....21

Concurrence Table.....56

Introduction / Scope of Work

This project presents the design and implementation of a GPS / Nodal Network remote controlled automobile that was engineered by New Paltz Engineering program students to address challenges in time management and food delivery logistics. Leveraging satellite navigation and onboard sensors, the vehicle will accurately determine the car's position, calculate optimal routes, and adapt to environmental factors in real time. By focusing on the projects scope of work, here is a detailed breakdown on the contributions per engineering discipline on the project:

- Mechanical Engineering
    - Utilize SolidWorks Dassault to design the vehicle’s chassis and body.
    - Kinematics of the Vehicle (wheels, axles, and suspension).
    - Conducted detailed measurements of material geometry to ensure accurate component specifications.
- Electrical Engineering
    - Integrated and wired components which included ESP32 and I2C for Artificial Intelligence (AI) integration.
    - Utilized DC voltage systems, power, and Ohm’s Law to support vehicle operations and functionality.
- Computer Engineering
    - Developed and utilized software to support and navigate the vehicle.
    - Engineered obstacle detection systems through AI to autonomously avoid obstacles and navigate around blockage.
    - Implemented Bluetooth technology to provide path instructions and enable manual override capabilities.

By combining these efforts, Team 13 will deliver a functional working design capable of autonomous food delivery with real time positioning, dynamic routing, and reliable obstacle avoidance.

Theory

Modern day automobiles use GPS operated movement through location calculation by measuring time signals taken to travel between satellites and the GPS receiver. Throughout the automobile industry, precision has improved throughout the years due to technological improvements in Integrated Circuit (IC) chips, code algorithms, and frequency signals. The system integrates with digital maps—often preloaded—and calculates movement using both satellite data and internal sensors. A Nodal Network is a system of interconnected devices using Nodes that create, receive, and store timing signals and map data, performing calculations to compute and share precise position fixes, exchanging real time traffic updates for dynamic rerouting, and falling back on dead reckoning and multi–Global Navigation Satellite System (GNSS) integration to maintain continuous and reliable navigation.

From a Mechanical Engineering standpoint, the chassis and steering system must deliver both structural rigidity and precise wheel control to support reliable navigation. The team used FEA analysis to identify and eliminate flex points in the frame, then performed detailed geometric measurements and material property evaluations to select lightweight yet stiff materials that will hold the GPS module and other components firmly in place. Furthermore, the suspension and drive train layouts were modeled to ensure consistent wheel-speed encoder readings across various terrains. Crucially, Team 13 also implemented Ackermann steering for direction

manipulation. Ackermann steering is a type of front wheel movement that during a turn, the inner and outer wheels trace concentric circles about a common center, thereby minimizing tire scrub and maintaining pure rolling contact. This type of steering optimizes vehicle wheel grip and handling across diverse conditions, while also having the ability to take sharp turns with precision. To complement the vehicle's autonomy, we also developed a mobile application that allows users to manually override the car's automated routes, select pickup and drop-off locations among SUNY New Paltz's on-campus dining options, and command the vehicle's movements in real time. This app communicates wirelessly with the car's control nodes via Bluetooth and Wi-Fi enabling seamless transitions between autonomous delivery mode and manual control as needed.

On the Electrical Engineering side, achieving reliable navigation and control requires diverse and meticulous circuit and power system design. The usage of microcontrollers such as ESP32, motor drivers, servo, and sensor conditioning electronics were utilized to control and operate the vehicle. Battery management circuits and DC-DC regulators then deliver stable voltage rails to all modules, while DC steppers and analog to digital converters preserve the fidelity of the wheel speed, inertia, and ultrasonic range signals. A combined wired CAN bus and wireless network protocols ties these nodes together, enabling the dynamic rerouting and manual override capabilities that make the autonomous food delivery robot both practical and robust.

Procedure

To transform this project into a functional food-delivery vehicle, multiple phases of research and prototyping were necessary. Once the initial scope of the project was established, the car was designed using SolidWorks software to precisely model and shape the physical chassis, which

was subsequently fabricated through 3D printing with appropriate filaments. Figure 1 below illustrates the SolidWorks prototype of the chassis, developed after careful consideration of mechanical measurements, finite element analysis, and research of the kinematics involved with gears, axles, and the overall structural frame.

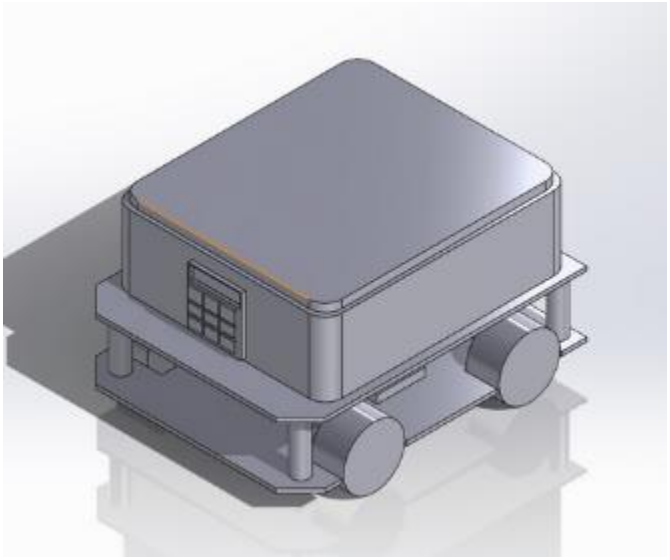


Figure 1: SolidWorks Model of Chassis Model

After developing and presenting the chassis, the team selected the Ackermann steering mechanism based on its advantages outweighing any disadvantages, notably rapid steering angle accessibility and practical feasibility for integration within the chassis. The Ackermann mechanism was meticulously modeled in SolidWorks using mating functions, enabling precise rotation of gears along a defined guide rail, with clamps strategically positioned to prevent unintended steering angles. Figure 2 below illustrates the 3D-printed Ackermann steering model.

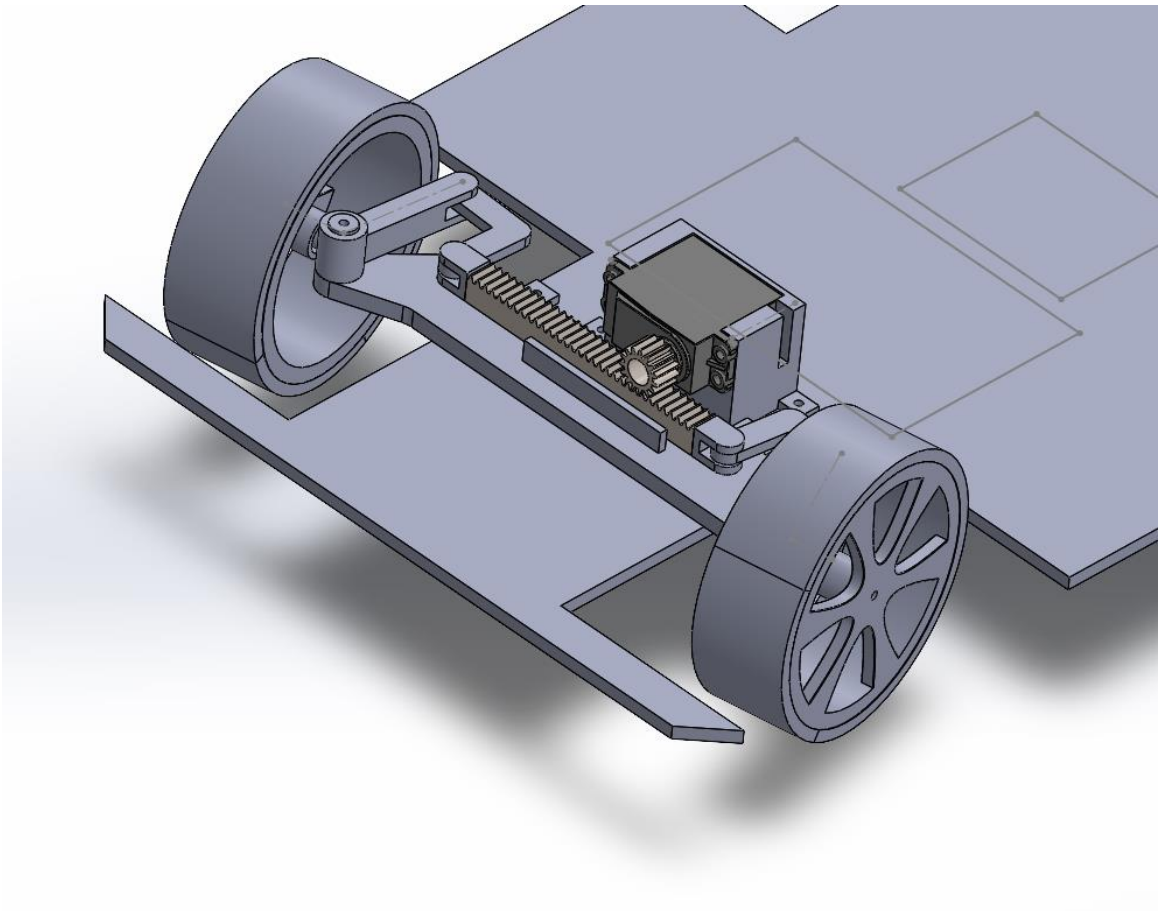


Figure 2: 3D Print of Ackermann Engineering

After completing the chassis and steering systems, the team procured acrylic sheets to construct multiple layered platforms atop the vehicle. These layers were designed with distinct purposes: the bottom layer houses electronic hardware components, such as the ESP32 microcontroller and breadboard; the second layer securely holds the battery pack; and the third, uppermost layer serves as the storage area for food and beverages intended for delivery. To assemble these layers, the acrylic sheets were carefully measured, cut, and precisely positioned on the vehicle, ensuring optimal placement, functionality, and structural integrity.

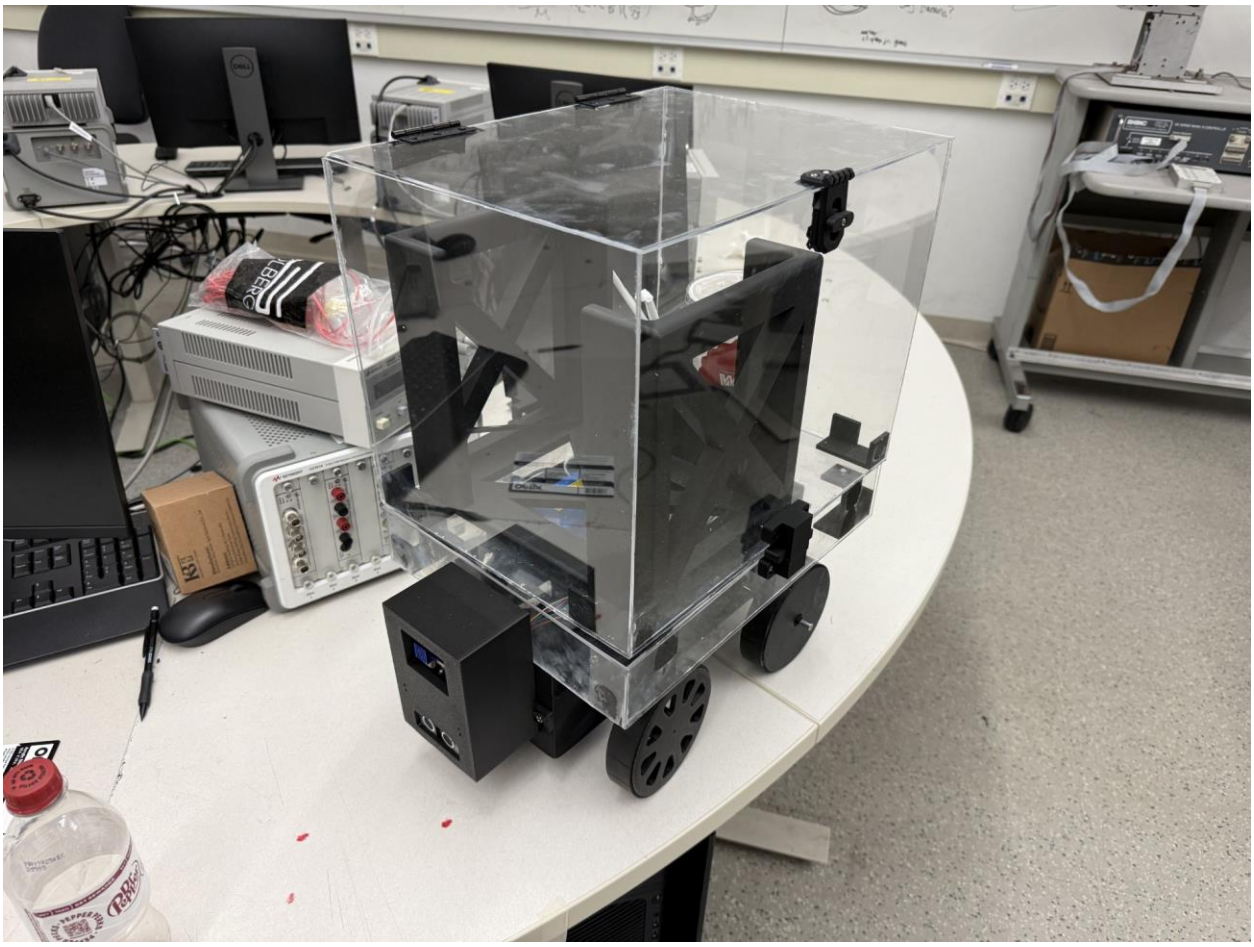


Figure 3: Final Design of Vehicle

During the physical engineering phase, the electrical and computer engineering team concurrently developed the software and electronic hardware systems supporting the vehicle. An ESP32 microcontroller was integrated onto a breadboard to facilitate Bluetooth connectivity, enabling communication with an Android application designed as a remote-control interface for manual override functionality. Figures 4a and 4b illustrate the wired ESP32 configuration on the breadboard and the corresponding Android app interface, respectively. Additionally, Wi-Fi connectivity was implemented to support and power the GPS module, ensuring continued nodal network functionality and navigation capability in scenarios where Bluetooth via the ESP32

might become unavailable or non-functional. The code for nodal network and Wi-Fi is included in the appendix

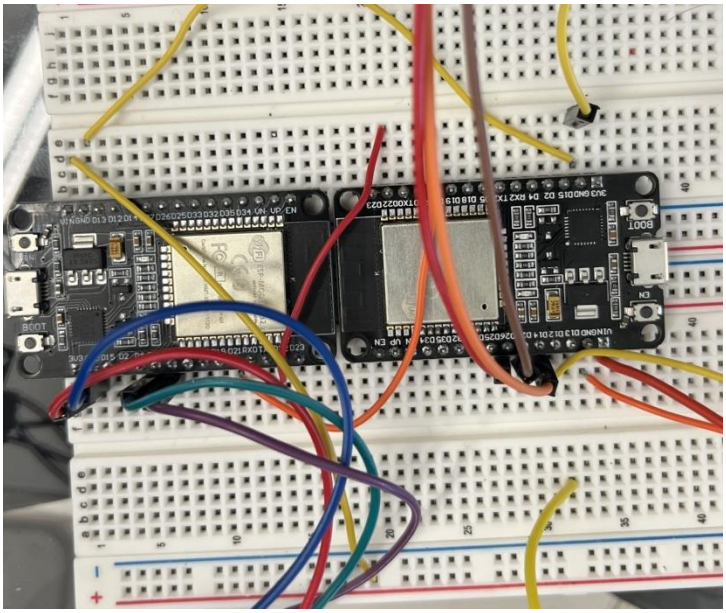


Figure 4a: ESP32 Connected to Breadboard

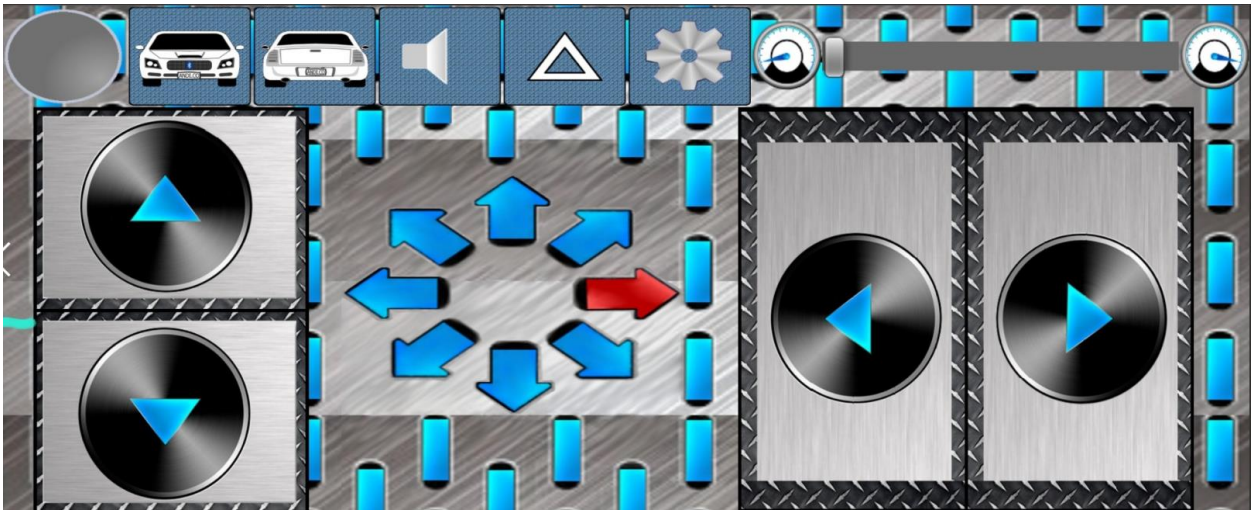


Figure 4b: Car Control App

To effectively detect obstacles along the delivery route, the team incorporated the DFROBOT HuskyLens, an AI-powered camera module designed for real-time object detection and human/object recognition. The HuskyLens AI module underwent extensive training over several months to reliably identify faces, humans, and common obstacles, enabling responsive behavior when obstructions are detected in the vehicle's path. Initially, AI was trained using publicly sourced images to recognize basic objects such as trees and people. With advanced training and iterative refinement, the module progressively improved, gaining the capability to swiftly and accurately identify and differentiate between various objects in real-time scenarios. Figures 5 show the AI module attached to the final product.

Figure 5: HuskyLens Attached

To regulate and maintain appropriate voltage and current levels throughout the system, a DC-DC step-down voltage regulator and an L298N DC motor driver module were incorporated into the

project. The DC-DC step-down regulator was configured to ensure stable voltage and current distribution, particularly during high-speed operation when current draw significantly increases, thus preventing potential damage to sensitive electronic components. Additionally, the L298N motor driver was integrated to precisely control and modulate voltage and current delivered to the motor, safeguarding the motor hardware from overheating, voltage spikes, or electrical damage during varying operational conditions.

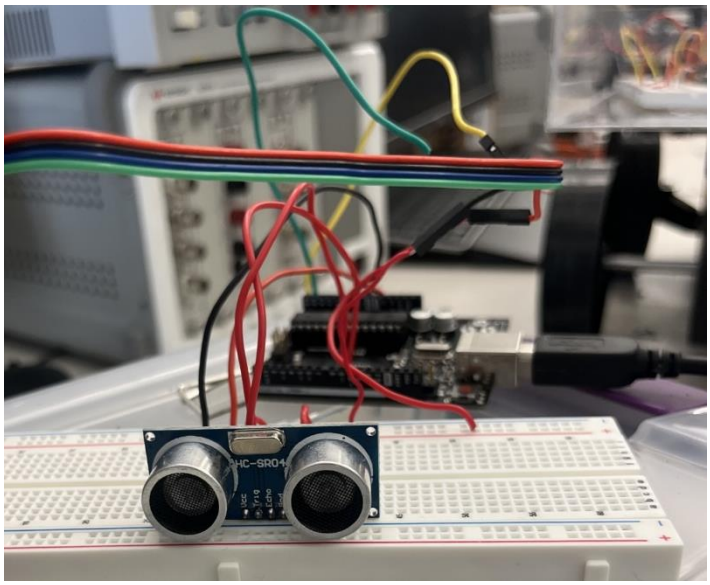


Figure 5: DC-DC Step Down Voltage

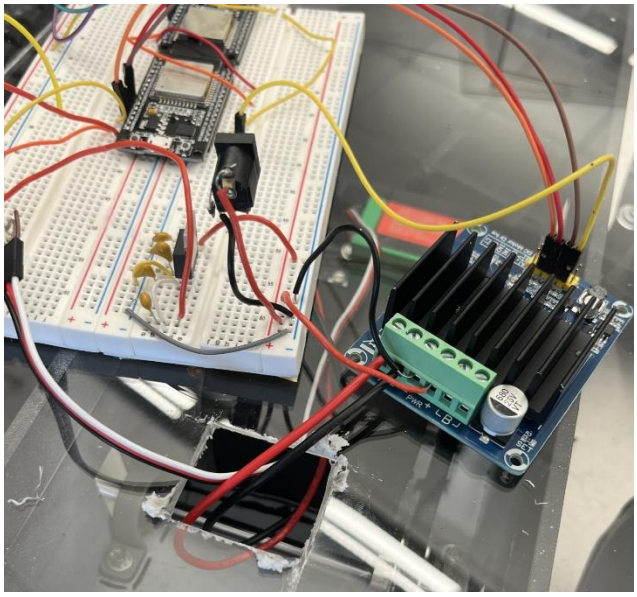


Figure 6: L298N DC Motor Driver Module

To enhance customer service and improve user convenience, a food-selection application was developed. The application was created using HTML (**HyperText Markup Language**) on CodePen, then subsequently converted into an APK file for deployment on an Android test device. The primary functionality of the app requires users to enter a 4-digit password to gain access and then select the dining location from which their food was ordered (e.g., Element 93 at SUNY New Paltz). If the user enters an incorrect password five consecutive times, the app locks, triggering a manual override prompt. Should the manual override code also be entered incorrectly five times, the application will permanently disable access and display the message, "PLEASE CONTACT CUSTOMER SUPPORT." The implementation of this security feature addresses contemporary concerns regarding theft in robotic food delivery services. The complete source code for this application is included in the Appendix below.

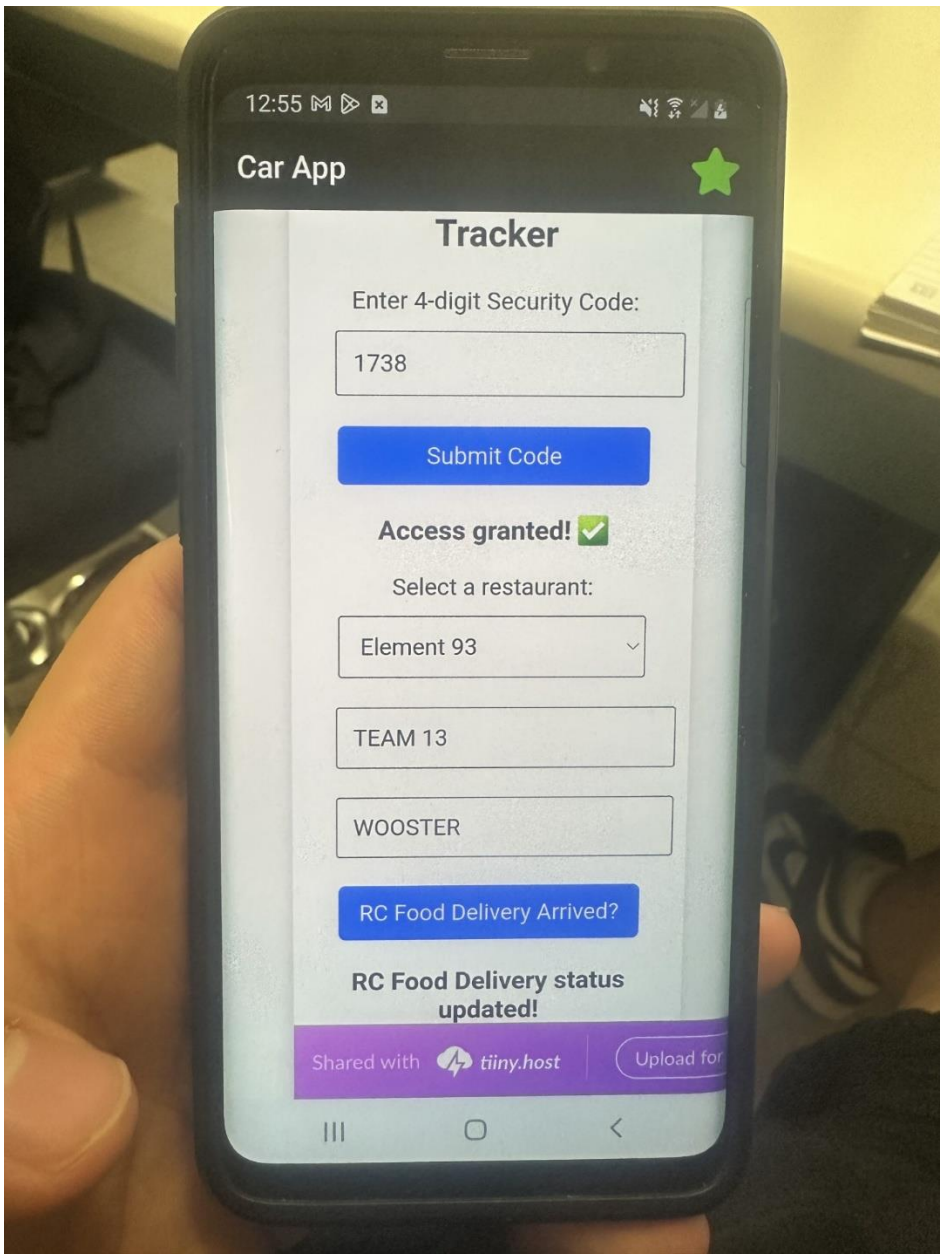


Figure 7a: Basic App Function

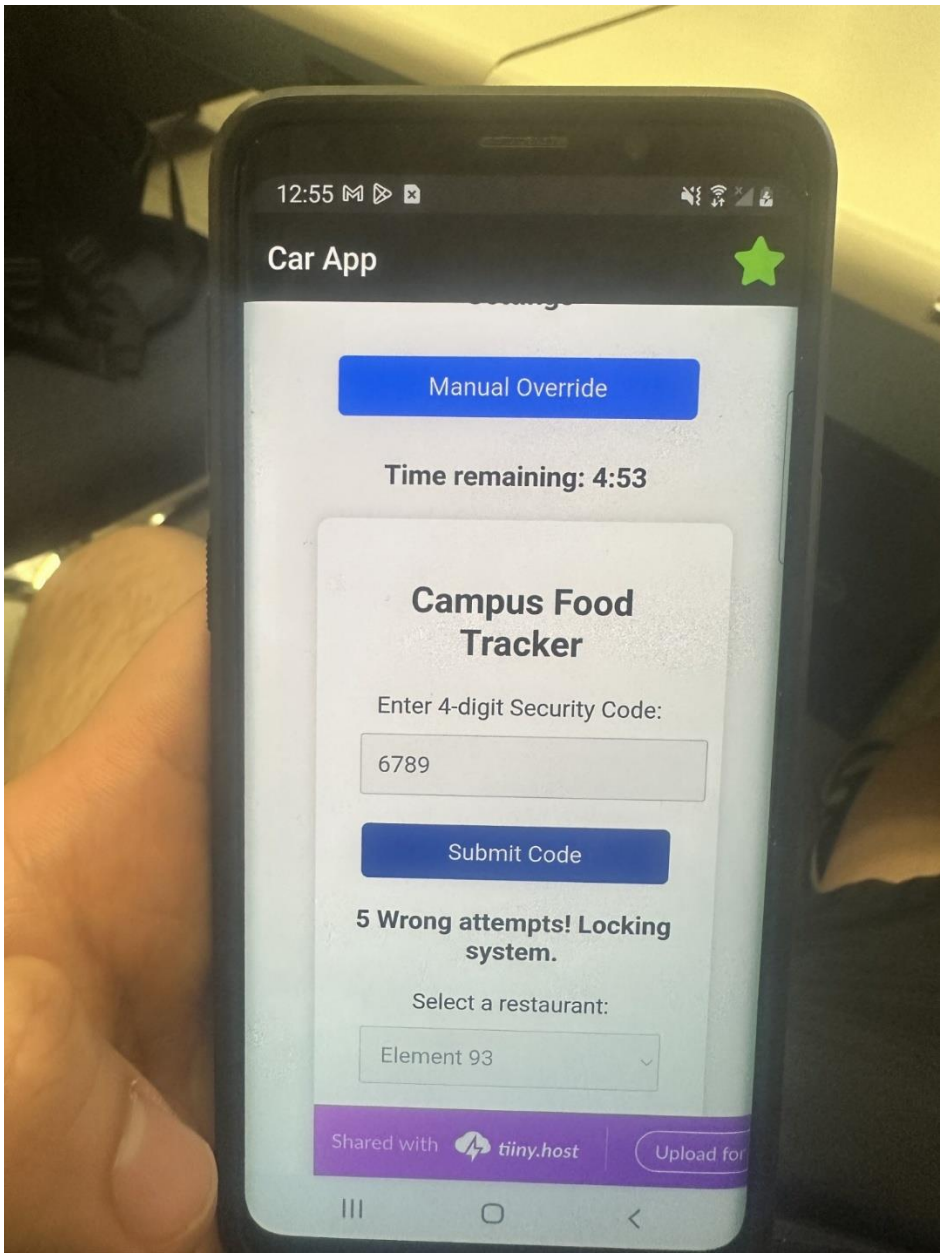


Figure 7b: Locked App

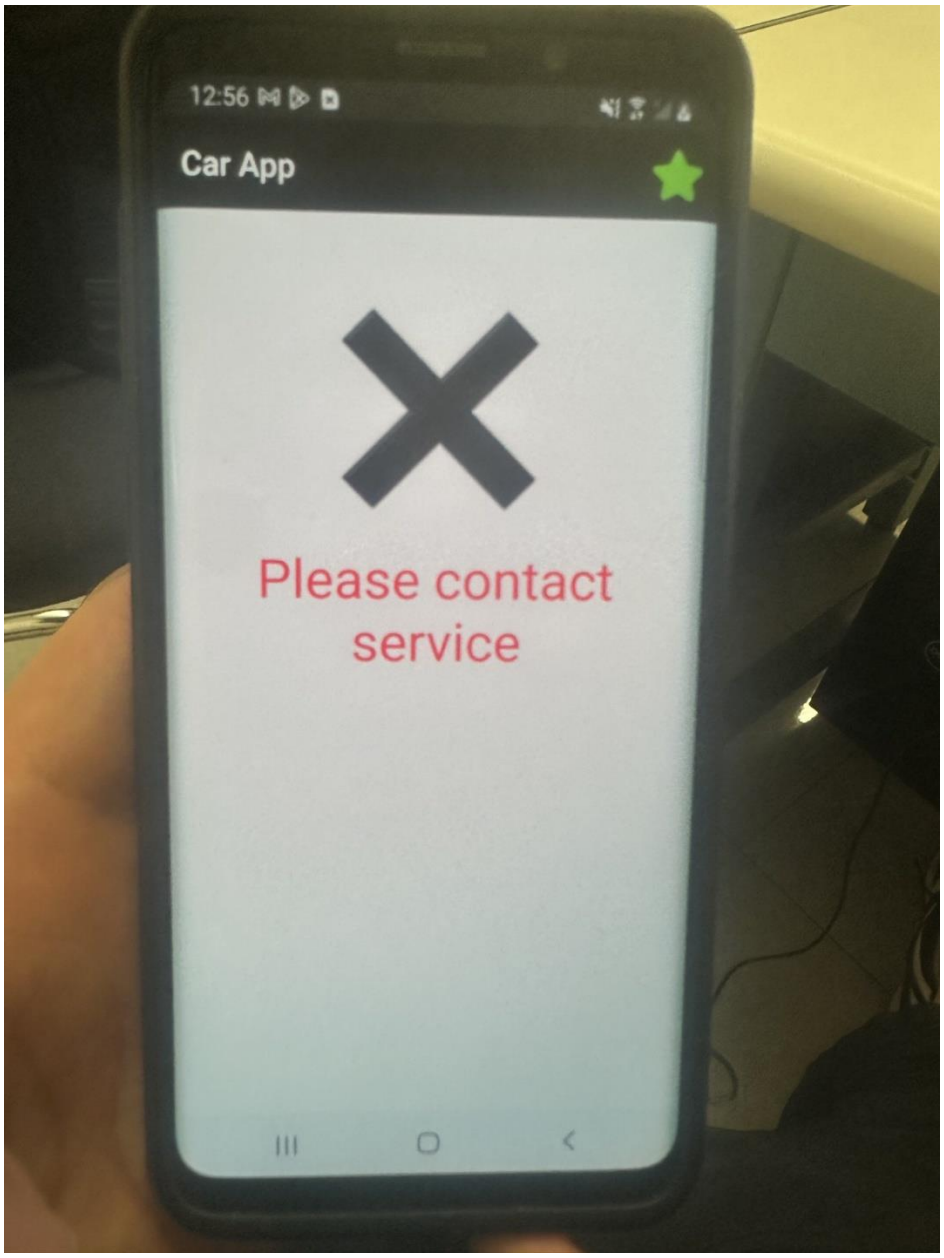


Figure 7c: Manual Override and Error Message

The entire RC car is powered by a 5V battery bank, located on the second layer of the vehicle, which supplies power to all onboard components and wiring necessary for full system functionality.

Results

The goal of this project was to be able to provide food deliveries across campus while using AI powered self-navigation without user input. This task involves many different components that all need to work and function to create a working car. Our results can be summarized into a few main parts.

Mechanical Performance:

The car's mechanics were successfully able to get it from the starting point to its destination. The design is able to withstand the weight of multiple meal containers provided from dining services, along with its own hardware and bodyweight. The car used the Ackermann steering design controlled by a servo to make turns and avoid obstacles. The car is limited to solid surfaces for the most part due to no suspension being in the design. Even with this it can operate a little off-road, but it's not recommended. The original design was to create a triple wheel design to navigate elevation changes and stairs. This was not realistic due to the time frame and budget, so we had to move to a standard wheel and axle design. The wheels were printed out of acrylic and then lined with rubber to deal with cracks and indifferences in the surface giving the wheels a longer lifespan as well. For the material selection group 13 decided to go with 3d printed acrylic since it is weather resistant to outdoor conditions, strong and inexpensive to print. This also came in handy when printing multiple parts for the prototype. Using an inexpensive but effective material allows for the group to stay under budget while being able to print multiple part designs. To support the placement of the HuskyLens and Ultrasonic sensor, a new 3D printed mount/enclosure was placed on front of the car to maximize performance and material

placement. Additionally, a clip lock was added on layer 2 and layer 3 to prevent unauthorized access to malicious actors.

AI Navigation System:

The AI navigation was implemented but did not work to its full extent. For AI training we used the Huskylens built in AI functions for obstacle detection. The built-in cameras had AI detection where it can recognize objects such as trees and a person to avoid collision or readjust the current path to avoid collision. The AI module automatically recognizes the distance between itself and the obstacle and updates it in real time. The original design was to train the module with stock images and real-life situations to train it to avoid obstacles that could appear on a college campus. However, the Maixcam was not compatible with Arduino microcontroller and struggled communicating with each other. Because of this, the car needs to be manually controlled through an app. Due to the Maixcam complications, Team 13 decided to implement the Huskylens instead due to convenience, capability, and functionality.

Electrical System Performance:

To send a full 8.4V to power the servo to turn and maximize Ackermann performance with food weight load, an MP1584 buck converter is used to deliver a stable 8.4V supply to the servo, ensuring maximum torque and responsiveness for Ackermann steering performance under food weight load. An MP1584 is a DC-DC buck converter, which means it steps down a higher input voltage to a lower, stable output voltage. All connections to power the car were successfully implemented to make the car operational and wired all components such as motor and microcontrollers.

App Performance:

The app's main goal was to provide security for the person's ordered food. The app successfully generated a user specific code for each delivery to ensure the correct person received their food. The app provides proper feedback on the codes and will lock you out after the fifth wrong code. This works the same way on the manual override key meaning that a hard reset will have to be done by one of the developers to get the vehicle working again. The app that manual controls the car is used as a manual override function to the AI detection. Since the AI detection does not fully work during vehicle operation, the car was driven using the controls app.

Conclusion

In conclusion, the autonomous robotic delivery project has demonstrated the feasibility and practical value of integrating a GPS guided navigation, obstacle avoidance, and a user-friendly scheduling interface to meet the on-campus needs of students and faculty. By moving from initial concept and multidisciplinary planning in Fall 2024 to a working prototype in Spring 2025, the team validated both the mechanical design and the end-to-end software architecture. This effort was a valid attempt to meet the scope of the project and functionality but held back by a few obstacles along the path due to AI coding issues and GPS module coding difficulties. Besides these issues, all other engineering deliverables were delivered. This effort not only underscored the importance of cross-disciplinary collaboration in tackling real-world challenges but also laid the groundwork for scalable deployment of autonomous delivery solutions in similar environments. Future work will focus on perfectly implementing AI detection while moving, GPS self-navigation, rigorous field testing, refining routing algorithms for dynamic campus conditions, and expanding

the platform's integration with existing food-service networks to enhance reliability and user satisfaction.

Bibliography

Sarasota, J. (2022, July 13th). *How GPS Works in a Car: The Complete Guide*. Retrieved from Jaguar Sarasota: https://www.jaguarsarasota.com/how-gps-works-in-a-car-the-complete-guide/

SolarWinds. (n.d.). *What Is a Network Node?* Austin, Texas: SolarWinds.

Vogel, J. (2021, April 6). *Tech Explained: Ackermann Sttering Geometery*. Retrieved from Racecar Engineering: https://www.racecar-engineering.com/articles/tech-explained-ackermann-steering-geometry/.

Appendix

MandSobjectTrackingSlave

```
#include <Wire.h> #include "BluetoothSerial.h" #include <ESP32Servo.h> #include
"esp_system.h"

BluetoothSerial SerialBT; Servo myServo;

// Diagnostic LED (on-board usually GPIO 2) const int diagLedPin = 2;

// I2C slave address #define I2C_SLAVE_ADDR 0x08

// Received I2C command volatile uint8_t i2cCommand = 0; volatile bool i2cNew = false;

// Motor pins const int motorPin1 = 27; const int motorPin2 = 26; const int enablePin = 14;

// Servo parameters const int servoPin = 12; int curAngle = 90; const int straightAngle = 90;
const int turnAngle = 40; // degrees to turn for left/right

// Tracking speed const int trackSpeed = 20; // % of max

// PWM properties const int freq = 60000; const int pwmChannel = 0; const int resolution = 10;

// ===== I2C receive event ===== void IRAM_ATTR onI2CReceive(int numBytes) { while
(Wire.available()) { i2cCommand = Wire.read(); i2cNew = true; } }

// ===== Drive & Steer ===== void drive(int speedPct, bool forward) { int duty = (speedPct *
((1 << resolution) - 1)) / 100; digitalWrite(motorPin1, forward ? HIGH : LOW);
digitalWrite(motorPin2, forward ? LOW : HIGH); ledcWrite(enablePin, duty); }
```

void stopDrive() { // immediate brake digitalWrite(motorPin1, LOW); digitalWrite(motorPin2, LOW); ledcWrite(enablePin, 0); }

int steerTo(int angle,int curAngle) { myServo.write(angle); // 3) wait just long enough for the horn to move int delta = abs(angle - curAngle); //float msPerDeg = 3.9; float msPerDeg = 15; delay(delta * msPerDeg + 10);

curAngle = angle; return curAngle; }

// ===== Tracking routines ===== void trackForward() { stopDrive(); digitalWrite(diagLedPin, HIGH); steerTo(straightAngle,90); drive(trackSpeed, true); }

void turnRight() { stopDrive(); digitalWrite(diagLedPin, HIGH); steerTo(straightAngle + turnAngle,90); drive(trackSpeed, true); }

void turnLeft() { stopDrive(); digitalWrite(diagLedPin, HIGH); steerTo(straightAngle - turnAngle,90); drive(trackSpeed, true); delay(50); }

void stopMovement() { digitalWrite(diagLedPin, LOW); stopDrive(); }

// ===== Setup ===== void setup() { // Diagnostic LED pinMode(diagLedPin, OUTPUT); digitalWrite(diagLedPin, LOW);

// Motor pins pinMode(motorPin1, OUTPUT); pinMode(motorPin2, OUTPUT); pinMode(enablePin, OUTPUT); ledcAttachChannel(enablePin, freq, resolution, pwmChannel);

// Servo myServo.attach(servoPin); myServo.write(straightAngle); curAngle = straightAngle;

// I2C slave init Wire.begin(I2C_SLAVE_ADDR); Wire.onReceive(onI2CReceive);

// Serial & BLE Serial.begin(115200); SerialBT.begin("ESP32TrackSlave");

SerialBT.println("BLE & I2C Tracking Slave Ready"); }

// ===== Main loop ===== void loop() { if (i2cNew) { uint8_t cmd = i2cCommand; i2cNew =

false; SerialBT.printf("[I2C] Cmd=%d\n", cmd);

```
switch (cmd) {
  case 3: trackForward();  break;  // forward
  case 4: turnRight();     break;
  case 5: turnLeft();      break;
  case 6: stopMovement();  break;
  default:
    SerialBT.println("[I2C] Unknown tracking cmd");
    stopMovement();
    break;
}
```

} }

HuskyLensObjectTrackerMaster

#include <HUSKYLENS.h> #include <Wire.h> #include <BluetoothSerial.h>

HUSKYLENS huskylens; BluetoothSerial SerialBT;

// Ultrasonic sensor pins const int trigPin = 18; const int echoPin = 19;

// I2C address of the motor/servo ESP32 slave const uint8_t SLAVE_ADDR = 0x08;

// Command bytes for the slave ESP32 #define CMD_TRACK_FORWARD 3 #define CMD_TURN_RIGHT 4 #define CMD_TURN_LEFT 5 #define CMD_STOP 6

// HuskyLens frame geometry const int FRAME_WIDTH = 320; const int CENTER_X = FRAME_WIDTH / 2; const int TOLERANCE = 40; // ±px around center const int DIST_THRESHOLD = 60; // cm

long duration;

// Measure distance via ultrasonic sensor int getUltrasonicDistance() { digitalWrite(trigPin, LOW); delayMicroseconds(2); digitalWrite(trigPin, HIGH); delayMicroseconds(10); digitalWrite(trigPin, LOW);

duration = pulseIn(echoPin, HIGH); //return duration * 0.034 / 2; return 70; }

// Send a single-byte command over I2C to the slave void sendCommand(uint8_t cmd) { Wire.beginTransmission(SLAVE_ADDR); Wire.write(cmd); Wire.endTransmission(); Serial.printf("Sent command %d\n", cmd); SerialBT.printf("Sent command %d\n", cmd); }

void setup() { // Serial + BLE for debug Serial.begin(115200); while (!Serial); SerialBT.begin("HuskyMaster"); Serial.println("Serial & BLE started");

// Ultrasonic pins pinMode(trigPin, OUTPUT); pinMode(echoPin, INPUT);

// I2C for HuskyLens + slave Wire.begin(); // SDA=21, SCL=22

// Initialize HuskyLens Serial.println("Connecting to HuskyLens..."); while (!huskylens.begin(Wire)) { Serial.println("HuskyLens begin failed! Check wiring and protocol."); delay(500); } Serial.println("HuskyLens connected.");

// Switch to object-tracking algorithm //huskylens.writeAlgorithm(ALGORITHM_OBJECT_TRACKING); Serial.println("HuskyLens set to OBJECT_TRACKING mode."); }

void loop() { // Request new data if (!huskylens.request()) { Serial.println("HuskyLens request failed."); sendCommand(CMD_STOP); delay(200); return; }

// If an object is learned and available if (huskylens.isLearned() && huskylens.available()) { HUSKYLENSResult obj = huskylens.read(); int x = obj.xCenter; //int distance = getUltrasonicDistance(); int distance = 70; Serial.printf("Object at x=%d, dist=%dcm\n", x, distance); SerialBT.printf("Object at x=%d, dist=%dcm\n", x, distance);

```
if (distance > DIST_THRESHOLD) {
  // Object far enough: decide turn vs forward
  if (abs(x - CENTER_X) <= TOLERANCE) {
    sendCommand(CMD_TRACK_FORWARD);
  }
  else if (x > CENTER_X) {
    sendCommand(CMD_TURN_RIGHT);
  }
  else {
    sendCommand(CMD_TURN_LEFT);
  }
} else {
  // Too close: stop
  sendCommand(CMD_STOP);
}
```

} else { // No object on-screen: stop sendCommand(CMD_STOP); }

delay(200); }

MandSObjectAvoidanceWHsuky

#include <Wire.h> #include "BluetoothSerial.h" #include <ESP32Servo.h> #include "esp_system.h"

```
BluetoothSerial SerialBT; Servo myServo;

// Diagnostic LED (on-board usually GPIO 2) const int diagLedPin = 2;

// I2C slave address (choose unused 7-bit address) #define I2C_SLAVE_ADDR 0x08

// Received I2C command code volatile uint8_t i2cCommand = 0; volatile bool i2cNew = false;

// Motor pins const int motorPin1 = 27; const int motorPin2 = 26; const int enablePin = 14;

// Servo parameters const int servoPin = 12; int curAngle = 90; const int straightAngle = 90;
const int avoidAngle = 60;

// PWM properties const int freq = 60000; const int pwmChannel = 0; const int resolution = 10;

// Movement timing (ms per degree at 3.3V) const float msPerDeg = 4.0;

// Avoidance params const int avoidSpeed = 15; // % of max const int avoidTime = 1000; // ms
const int turnTime = 1000; // ms

// Software "interrupt" flag volatile bool stopRequested = false;

// ===== I2C receive event ===== void IRAM_ATTR onI2CReceive(int numBytes) { while
(Wire.available()) { i2cCommand = Wire.read(); i2cNew = true; } }

// ===== Drive & Steer ===== void drive(int speedPct, bool forward) { int duty = (speedPct *
((1 << resolution) - 1)) / 100; digitalWrite(motorPin1, forward ? HIGH : LOW);
digitalWrite(motorPin2, forward ? LOW : HIGH); ledcWrite(enablePin, duty); }

void stopDrive() { // quick brake then coast steerAngle(straightAngle); drive(20, false);
delay(200); ledcWrite(enablePin, 0); }
```

```
int steerAngle(int angle,int curAngle) { myServo.write(angle); // 3) wait just long enough for the horn to move int delta = abs(angle - curAngle); //float msPerDeg = 3.9; float msPerDeg = 15; delay(delta * msPerDeg + 10);

curAngle = angle; return curAngle; }

int steerAngle(int angle) { myServo.write(angle); // 3) wait just long enough for the horn to move

float msPerDeg = 3.9; //delay(delta * msPerDeg + 10); return 1; }

// ===== check for BL E / stop ===== bool checkStop() { // poll BLE for '0' while (SerialBT.available()) { char c = SerialBT.read(); if (c == '0') { stopRequested = true; } } if (stopRequested) { stopDrive(); stopRequested = false; return true; } return false; }

// ===== Avoidance routines ===== void avoidLeft() { if (checkStop()) return; curAngle = steerAngle(straightAngle - avoidAngle,curAngle);//Turn Left if (checkStop()) return; drive(avoidSpeed, true);// Drive if (checkStop()) return; //delay(avoidTime); if (checkStop()) return; curAngle = steerAngle(straightAngle + avoidAngle,curAngle);//right if (checkStop()) return; //delay(avoidTime);//delay if (checkStop()) return; curAngle = steerAngle(straightAngle,curAngle); if (checkStop()) return; // delay(avoidTime);//delay curAngle = steerAngle(straightAngle + avoidAngle,curAngle); if (checkStop()) return; // delay(avoidTime); curAngle = steerAngle(straightAngle - avoidAngle,curAngle); if (checkStop()) return; // delay(avoidTime); curAngle = steerAngle(straightAngle,curAngle); }
```

```
void avoidRight() { SerialBT.println("[I2C] avoidRight triggered"); if (checkStop()) return; steerAngle(straightAngle + avoidAngle); if (checkStop()) return; drive(avoidSpeed, true); unsigned long end = millis() + avoidTime; while (millis() < end) { if (checkStop()) return; delay(10); } steerAngle(straightAngle - avoidAngle); if (checkStop()) return; end = millis() + avoidTime * 1.3; while (millis() < end) { if (checkStop()) return; delay(10); } steerAngle(straightAngle); }

void backupLeft() { SerialBT.println("[I2C] backupLeft triggered"); if (checkStop()) return; drive(avoidSpeed, false); unsigned long end = millis() + avoidTime; while (millis() < end) { if (checkStop()) return; delay(10); } steerAngle(straightAngle - avoidAngle); if (checkStop()) return; drive(avoidSpeed, true); end = millis() + turnTime; while (millis() < end) { if (checkStop()) return; delay(10); } steerAngle(straightAngle); }

void backupRight() { SerialBT.println("[I2C] backupRight triggered"); if (checkStop()) return; drive(avoidSpeed, false); unsigned long end = millis() + avoidTime; while (millis() < end) { if (checkStop()) return; delay(10); } steerAngle(straightAngle + avoidAngle); if (checkStop()) return; drive(avoidSpeed, true); end = millis() + turnTime; while (millis() < end) { if (checkStop()) return; delay(10); } steerAngle(straightAngle); }

void setup() { pinMode(diagLedPin, OUTPUT); digitalWrite(diagLedPin, LOW);

// Motor setup pinMode(motorPin1, OUTPUT); pinMode(motorPin2, OUTPUT); pinMode(enablePin, OUTPUT); ledcAttachChannel(enablePin, freq, resolution, pwmChannel);

// Servo myServo.attach(servoPin); myServo.write(straightAngle); curAngle = straightAngle;

// I2C slave init Wire.begin(I2C_SLAVE_ADDR); Wire.onReceive(onI2CReceive);
```

```
// Serial & BLE init Serial.begin(115200); SerialBT.begin("ESP32testSlave");
SerialBT.println("BLE & I2C Slave Ready"); }

void loop() { // Check for new I2C command if (i2cNew) { uint8_t cmd = i2cCommand; i2cNew
= false; SerialBT.print("[I2C] Cmd="); SerialBT.println(cmd); switch (cmd) { case 1:
avoidLeft(); break; case 2: avoidRight(); break; case 3: backupLeft(); break; case 4:
backupRight(); break; case 0: stopRequested = true; break; default: SerialBT.println("[I2C]
Unknown cmd"); break; } } if (SerialBT.available()) { char c = SerialBT.read();
Serial.print("Cmd: "); Serial.println(c);
```

```
switch (c) {
  case '0':
    stopDrive();
    while(c != '7'){
    if (SerialBT.available()) {
      c = SerialBT.read();
      SerialBT.println("C is " + c);
    }}
    break;
  case '1':
    drive(15, true);
    break;
  case '2':
    avoidLeft();
    break;
  case '3':
    avoidRight();
    break;
  case '4':
    backupLeft();
    break;
  case '5':
    backupRight();
```

```
    break;
  default:
    Serial.println("Use 0-5");
    break;
}
```

}}

HuskyLensI2C1Obstacle

#include <HUSKYLENS.h> #include "Wire.h" #include "BluetoothSerial.h"

HUSKYLENS huskylens; BluetoothSerial SerialBT;

const int trigPin = 18; const int echoPin = 19;

long duration; int distance = 30;

// I2C address of your motor/servo ESP32 slave const uint8_t SLAVE_ADDR = 0x08;

int getUltrasonicDistance() { digitalWrite(trigPin, LOW); delayMicroseconds(2); digitalWrite(trigPin, HIGH); delayMicroseconds(10); digitalWrite(trigPin, LOW);

duration = pulseIn(echoPin, HIGH); return duration * 0.034 / 2; }

void setup() { // USB serial + BLE Serial.begin(115200); while (!Serial); SerialBT.begin("HuskyMaster"); Serial.println("Serial & BLE started"); SerialBT.println("BLE ready");

// Ultrasonic pins pinMode(trigPin, OUTPUT); pinMode(echoPin, INPUT);

// I2C bus for HuskyLens + slave Wire.begin(); // SDA=21, SCL=22

// HuskyLens init Serial.println("Connecting to HuskyLens..."); SerialBT.println("Connecting to HuskyLens..."); while (!huskylens.begin(Wire)) { Serial.println("HuskyLens begin failed!"); SerialBT.println("HuskyLens begin failed!"); delay(1000); } Serial.println("HuskyLens connected!"); SerialBT.println("HuskyLens connected!"); delay(100);

// Name your trained IDs huskylens.setCustomName("Brick", 1); huskylens.setCustomName("Sidewalk", 2); huskylens.setCustomName("Rocks", 3); huskylens.setCustomName("Pothole", 4); huskylens.setCustomName("Shoes", 5);

Serial.println("Setup complete. Ready to detect."); SerialBT.println("Setup complete. Ready to detect."); }

void loop() { // Request a new frame if (huskylens.request()) { int count = huskylens.countBlocks(); Serial.print("Objects detected: "); Serial.println(count); SerialBT.printf("Objects detected: %d\n", count);

```
// Iterate through detections
for (int i = 0; i < count; i++) {
  HUSKYLENSResult result = huskylens.getBlock(i);
  int id = result.ID;
  Serial.printf("Detected ID: %d\n", id);
  SerialBT.printf("Detected ID: %d\n", id);

  // If it’s a Shoes (ID==5) check distance and then alert slave
  if (id == 5) {
    distance = getUltrasonicDistance();
    Serial.printf("Distance to Shoes: %d cm\n", distance);
    SerialBT.printf("Distance to Shoes: %d cm\n", distance);

    // Only send the I2C command if distance within threshold
    if (distance <= 90) {
      // Send byte 2 to slave
      Wire.beginTransmission(SLAVE_ADDR);
      Wire.write((uint8_t)2);
      Wire.endTransmission();

      Serial.println("Sent command 2 to slave (avoidShoes)");
      SerialBT.println("Sent command 2 to slave (avoidShoes)");
    } else {
```

```
      Serial.println("Shoes   too   far,   not   sending   command");
      SerialBT.println("Shoes   too   far,   not   sending   command");
    }
  }
}
```

} else { Serial.println("No object detected this frame."); SerialBT.println("No object detected this frame."); }

delay(500); }

MandSI2CGPS1

#include <Wire.h> #include <ESP32Servo.h> #include <BluetoothSerial.h>

#define SLAVE_ADDRESS 0x08

BluetoothSerial SerialBT;

// Helper to print to both Serial Monitor & BLE console, unconditionally void debug(const String &msg) { Serial.println(msg); SerialBT.println(msg); }

Servo myServo; const int servoPin = 13; const int motor1Pin1 = 27; const int motor1Pin2 = 26; const int enable1Pin = 14;

const int freq = 60000; const int pwmChannel = 0; const int resolution = 10; const int maxPWM = 1024;

// Servo positions const int servoCenter = 90; const int servoLeft = 45; const int servoRight = 135;

void setup() { Serial.begin(115200); SerialBT.begin("Motor_ESP32_Slave"); // Bluetooth SPP device name

// Motor driver setup pinMode(motor1Pin1, OUTPUT); pinMode(motor1Pin2, OUTPUT); pinMode(enable1Pin, OUTPUT); pinMode(enable1Pin, OUTPUT); ledcAttachChannel(enable1Pin, freq, resolution, pwmChannel);

// Servo setup myServo.attach(servoPin); myServo.write(servoCenter);

// I²C slave setup Wire.begin(SLAVE_ADDRESS); Wire.onReceive(receiveEvent);

debug("✅ Motor ESP32 Slave ready (I2C + BLE)"); }

void loop() { static bool wasConnected = false; bool isConnected = SerialBT.hasClient();

// Detect new connections / disconnections if (isConnected && !wasConnected) { debug("🔗 BLE client connected"); wasConnected = true; } else if (!isConnected && wasConnected) { debug("❌ BLE client disconnected"); wasConnected = false; }

// Nothing else to do in loop — all I²C work happens in receiveEvent delay(10); }

// I²C receive handler: print distance & bearing to Serial + BLE void receiveEvent(int numBytes) { if (numBytes >= 2) { uint8_t dist = Wire.read(); uint8_t angle = Wire.read(); debug("I2C RX → Distance: " + String(dist) + " m"); debug("I2C RX → Bearing: " + String(angle) + " deg");

```
// Flush any extra bytes
while (Wire.available()) {
  uint8_t extra = Wire.read();
  debug("I2C RX extra byte flushed: " + String(extra));
}
```

} else { debug("❌ I2C RX error: expected 2 bytes, got " + String(numBytes)); while (Wire.available()) { uint8_t extra = Wire.read(); debug("I2C RX flushed byte: " + String(extra)); } } }

// Placeholder: manual motor control via BLE console void setMotorSpeed(int percent) { int pwm = map(percent, 0, 100, 0, maxPWM); digitalWrite(motor1Pin1, HIGH); digitalWrite(motor1Pin2, LOW); ledcWrite(pwmChannel, pwm); debug("🚗 Motor " + String(percent) + "% (PWM " + String(pwm) + ")"); }

// Placeholder: manual servo control via BLE console void turnServo(char dir) { if (dir == 'l') { myServo.write(servoLeft); debug("↩️ Servo LEFT"); } else if (dir == 'r') { myServo.write(servoRight); debug("↪️ Servo RIGHT"); } }

GPSI2CNodes

#include <Wire.h> #include <BluetoothSerial.h> #include <TinyGPSPlus.h> #include

#define MOTOR_SLAVE_ADDRESS 0x08 // I²C address of Motor ESP32

BluetoothSerial SerialBT; HardwareSerial mySerial(1); TinyGPSPlus gps;

```
// === YOUR 13 MAIN NODES === std::vector<std::pair<double,double>> pathNodes =
{ {41.742878348761664, -74.08066534735731}, // 0 Start {41.742857, -74.080647}, // 1
{41.742744, -74.080793}, // 2 {41.742752, -74.080884}, // 3 {41.742734, -74.081047}, // 4
{41.742795, -74.081254}, // 5 {41.742844, -74.081413}, // 6 {41.742884, -74.081534}, // 7
{41.742960, -74.081858}, // 8 {41.743000, -74.082100}, // 9 {41.743085, -74.082025}, // 10
{41.743110, -74.082700}, // 11 {41.743100, -74.083200} // 12 End };

int currentNode = 0; bool tripStarted = false; const double reachThreshold = 2.0; // meters

// helper to print to Serial Monitor AND BLE console void debug(const String &msg)
{ Serial.println(msg); SerialBT.println(msg); }

void sendI2C(uint8_t dist, uint8_t angle)
{ Wire.beginTransmission(MOTOR_SLAVE_ADDRESS); Wire.write(dist); Wire.write(angle);
Wire.endTransmission(); debug("→ I²C send D=" + String(dist) + " A=" + String(angle)); }

void setup() { Serial.begin(115200); SerialBT.begin("GPS_ESP32_Master"); // BLE debug
always on Wire.begin(21, 22); // SDA, SCL mySerial.begin(9600, SERIAL_8N1, 16, 17);

debug("GPS Master booted; GPS starting..."); }

void loop() { // feed raw NMEA to TinyGPS bool any = false; while (mySerial.available()) { any
= true; gps.encode(mySerial.read()); } if (any) debug("NMEA fed to parser");

// wait for valid fix if (!gps.location.isValid()) { debug("Waiting for GPS fix..."); delay(1000);
return; }

// once we get an update if (gps.location.isUpdated()) { double lat = gps.location.lat(); double lng
= gps.location.lng(); int sats = gps.satellites.value(); double hdop = gps.hdop.hdop();
```

```
debug("GPS Fix: lat=" + String(lat,6) +
      " lng=" + String(lng,6) +
      " sats=" + String(sats) +
      " hdop=" + String(hdop,1));

// compute bearing & distance to the current target node
auto [tlat, tlng] = pathNodes[currentNode];
double dist    = TinyGPSPlus::distanceBetween(lat, lng, tlat, tlng);
double bearing = TinyGPSPlus::courseTo(       lat, lng, tlat, tlng);
uint8_t dByte = dist    > 255 ? 255 : (uint8_t)dist;
uint8_t aByte = bearing > 255 ? 255 : (uint8_t)bearing;

// always send distance & bearing
debug("→ Target Node#" + String(currentNode) +
      " Dist=" + String(dist,2) + "m" +
      " Brg="  + String(bearing,1) + "°");
sendI2C(dByte, aByte);

if (!tripStarted) {
  // Pre-start: check if close enough to start node
  if (sats >= 3) {
    debug("Dist→start node: " + String(dist,2) + " m");
    if (dist < reachThreshold) {
```

```
      tripStarted = true;
      debug("Trip started!");
    } else {
      debug("Move closer to start node...");
    }
  } else {
    debug("Waiting for at least 3 satellites...");
  }
} else {
  // In-transit: advance when within threshold
  if (dist < reachThreshold) {
    currentNode++;
    if (currentNode >= pathNodes.size()) {
      debug("Destination reached!");
    } else {
      debug("Advancing to node #" + String(currentNode));
    }
  }
}


}
```

delay(1000); // 1 Hz update }

AppControlledCar

```
#include "BluetoothSerial.h" #include <ESP32Servo.h> #include "esp_system.h"

BluetoothSerial SerialBT;

// Diagnostic LED (on-board usually GPIO 2) const int diagLedPin = 2;

// Motor A pins const int motor1Pin1 = 27; const int motor1Pin2 = 26; const int enable1Pin = 14;

// Servo setup Servo myServo; const int servoPin = 12; int servoAngle = 90; const int tickStep = 5;

// PWM properties const int freq = 60000; const int pwmChannel = 0; const int resolution = 10;

// Speed variables int currentSpeedPercentage = 0; bool goingForward = true; const int maxSpeed = 1024;

// Queues QueueHandle_t motorQueue; QueueHandle_t servoQueue;

// Diagnostics state bool wasConnected = false; unsigned long lastHeartbeat = 0;

// Convert reset reason to string const char* resetReasonToString(esp_reset_reason_t r) { switch (r) { case ESP_RST_UNKNOWN: return "Unknown"; case ESP_RST_POWERON: return "Power-on"; case ESP_RST_EXT: return "External reset"; case ESP_RST_SW: return "Software reset"; case ESP_RST_PANIC: return "Panic"; case ESP_RST_INT_WDT: return "Interrupt WDT"; case ESP_RST_TASK_WDT: return "Task WDT"; case ESP_RST_WDT: return "Other WDT"; case ESP_RST_DEEPSLEEP: return "Deep Sleep"; case ESP_RST_BROWNOUT: return "Brownout"; case ESP_RST_SDIO: return "SDIO"; default: return "??"; } }
```

```
// Servo turning void turnServo(const String &direction) { if (direction == "rr") { if (servoAngle < 180) { servoAngle += tickStep; myServo.write(servoAngle); } Serial.print("Turning right. Servo angle: "); Serial.println(servoAngle); } else if (direction == "ll") { if (servoAngle > 0) { servoAngle -= tickStep; myServo.write(servoAngle); } Serial.print("Turning left. Servo angle: "); Serial.println(servoAngle); } else { Serial.println("Invalid input for servo."); } }

// Motor movement void moveMotor(int speedPercentage, const String &direction) { int dutyCycle = (speedPercentage * maxSpeed) / 100;

if (direction == "forward") { digitalWrite(motor1Pin1, HIGH); digitalWrite(motor1Pin2, LOW); } else { digitalWrite(motor1Pin1, LOW); digitalWrite(motor1Pin2, HIGH); }

ledcWrite(enable1Pin, dutyCycle);

Serial.print("Moving "); Serial.print(direction); Serial.print(" at "); Serial.print(speedPercentage); Serial.print("% ("); Serial.print(dutyCycle); Serial.println(" duty cycle)"); }

// Motor control task void motorControlTask(void *parameter) { String motorCommand; unsigned long lastPedalTime = 0; const int decelInterval = 150; const int noPressTimeout = 400;

while (true) { if (!SerialBT.hasClient()) { ledcWrite(enable1Pin, 0); currentSpeedPercentage = 0; vTaskDelay(100 / portTICK_PERIOD_MS); continue; }
```

```
if (xQueueReceive(motorQueue, &motorCommand, 10 / portTICK_PERIOD_MS))
{
  if (motorCommand == "ff") {
    goingForward = true;
    lastPedalTime = millis();
    currentSpeedPercentage = min(currentSpeedPercentage + 5, 100);
    moveMotor(currentSpeedPercentage, "forward");
  }
  else if (motorCommand == "bb") {
    goingForward = false;
    lastPedalTime = millis();
    currentSpeedPercentage = min(currentSpeedPercentage + 5, 100);
    moveMotor(currentSpeedPercentage, "backward");
  }
  else if (motorCommand == "ss") {
    lastPedalTime = 0;
  }
}

if (millis() - lastPedalTime > noPressTimeout &&
currentSpeedPercentage > 0) {
  static unsigned long lastDecel = 0;
  if (millis() - lastDecel >= decelInterval) {
```

```
    currentSpeedPercentage = max(currentSpeedPercentage - 30, 0);
    moveMotor(currentSpeedPercentage, goingForward ? "forward" :
"backward");
    lastDecel = millis();
  }
}

vTaskDelay(10 / portTICK_PERIOD_MS);


} }
```

// Servo control task void servoControlTask(void *parameter) { String servoCommand; while (true) { if (xQueueReceive(servoQueue, &servoCommand, portMAX_DELAY)) { turnServo(servoCommand); } } }

void setup() { // Diagnostic LED pinMode(diagLedPin, OUTPUT); digitalWrite(diagLedPin, LOW);

// Motor pins pinMode(motor1Pin1, OUTPUT); pinMode(motor1Pin2, OUTPUT); pinMode(enable1Pin, OUTPUT); ledcAttachChannel(enable1Pin, freq, resolution, pwmChannel);

// Servo myServo.attach(servoPin); myServo.write(servoAngle);

// Serial Serial.begin(115200); delay(500);

// Print reset reason esp_reset_reason_t reason = esp_reset_reason(); Serial.print("🔄 Boot! Reset reason: "); Serial.println(resetReasonToString(reason));

// Start Bluetooth SPP (no PIN) if (!SerialBT.begin("ESP32test")) { Serial.println("❌ BT init failed!"); while (1) { delay(100); } } Serial.println("✅ Bluetooth SPP started as "ESP32test"");

// Create queues and tasks motorQueue = xQueueCreate(10, sizeof(String)); servoQueue = xQueueCreate(10, sizeof(String)); xTaskCreatePinnedToCore(motorControlTask, "Motor Task", 10000, NULL, 1, NULL, 0); xTaskCreatePinnedToCore(servoControlTask, "Servo Task", 10000, NULL, 1, NULL, 1); }

void loop() { // Connection diagnostics bool isConnected = SerialBT.hasClient(); if (isConnected != wasConnected) { Serial.println(isConnected ? "🔗 Client connected" : "❌ Client disconnected"); digitalWrite(diagLedPin, isConnected ? HIGH : LOW); wasConnected = isConnected; }

// Heartbeat if (millis() - lastHeartbeat >= 5000) { lastHeartbeat = millis(); Serial.print("⏱️ Uptime: "); Serial.print(millis() / 1000); Serial.print(" s, Free heap: "); Serial.println(ESP.getFreeHeap()); }

// Handle incoming BLE commands if (SerialBT.available()) { String input; while (SerialBT.available()) { char c = SerialBT.read(); if (c == '\n' || c == '\r') break; input += c; } input.trim(); input.toLowerCase();

```
// Normalize single-letter to double-letter
if      (input == "f") input = "ff";
else if (input == "b") input = "bb";
else if (input == "s") input = "ss";
else if (input == "r") input = "rr";
else if (input == "l") input = "ll";
else if (input == "i") input = "ii";
else if (input == "g") input = "gg";
else if (input == "h") input = "hh";
else if (input == "j") input = "jj";

Serial.print("Received input: ");
Serial.println(input);

// Dispatch commands
if      (input == "ii") {
  String m = "ff", s = "rr";
  xQueueSend(motorQueue, &m, portMAX_DELAY);
  xQueueSend(servoQueue, &s, portMAX_DELAY);
}
else if (input == "gg") {
  String m = "ff", s = "ll";
  xQueueSend(motorQueue, &m, portMAX_DELAY);
```

```
  xQueueSend(servoQueue, &s, portMAX_DELAY);
}
else if (input == "hh") {
  String m = "bb", s = "ll";
  xQueueSend(motorQueue, &m, portMAX_DELAY);
  xQueueSend(servoQueue, &s, portMAX_DELAY);
}
else if (input == "jj") {
  String m = "bb", s = "rr";
  xQueueSend(motorQueue, &m, portMAX_DELAY);
  xQueueSend(servoQueue, &s, portMAX_DELAY);
}
else if (input == "ff" || input == "bb" || input == "ss") {
  xQueueSend(motorQueue, &input, portMAX_DELAY);
}
else if (input == "rr" || input == "ll") {
  xQueueSend(servoQueue, &input, portMAX_DELAY);
}



}
```

delay(10); }

HTML Code for Food App

```
#include "BluetoothSerial.h" #include <ESP32Servo.h> #include "esp_system.h"

BluetoothSerial SerialBT;

// Diagnostic LED (on-board usually GPIO 2) const int diagLedPin = 2;

// Motor A pins const int motor1Pin1 = 27; const int motor1Pin2 = 26; const int enable1Pin = 14;

// Servo setup Servo myServo; const int servoPin = 12; int servoAngle = 90; const int tickStep = 5;

// PWM properties const int freq = 60000; const int pwmChannel = 0; const int resolution = 10;

// Speed variables int currentSpeedPercentage = 0; bool goingForward = true; const int maxSpeed = 1024;

// Queues QueueHandle_t motorQueue; QueueHandle_t servoQueue;

// Diagnostics state bool wasConnected = false; unsigned long lastHeartbeat = 0;

// Convert reset reason to string const char* resetReasonToString(esp_reset_reason_t r) { switch (r) { case ESP_RST_UNKNOWN: return "Unknown"; case ESP_RST_POWERON: return "Power-on"; case ESP_RST_EXT: return "External reset"; case ESP_RST_SW: return "Software reset"; case ESP_RST_PANIC: return "Panic"; case ESP_RST_INT_WDT: return "Interrupt WDT"; case ESP_RST_TASK_WDT: return "Task WDT"; case ESP_RST_WDT: return "Other WDT"; case ESP_RST_DEEPSLEEP: return "Deep Sleep"; case ESP_RST_BROWNOUT: return "Brownout"; case ESP_RST_SDIO: return "SDIO"; default: return "??"; } }
```

```
// Servo turning void turnServo(const String &direction) { if (direction == "rr") { if (servoAngle < 180) { servoAngle += tickStep; myServo.write(servoAngle); } Serial.print("Turning right. Servo angle: "); Serial.println(servoAngle); } else if (direction == "ll") { if (servoAngle > 0) { servoAngle -= tickStep; myServo.write(servoAngle); } Serial.print("Turning left. Servo angle: "); Serial.println(servoAngle); } else { Serial.println("Invalid input for servo."); } }

// Motor movement void moveMotor(int speedPercentage, const String &direction) { int dutyCycle = (speedPercentage * maxSpeed) / 100;

if (direction == "forward") { digitalWrite(motor1Pin1, HIGH); digitalWrite(motor1Pin2, LOW); } else { digitalWrite(motor1Pin1, LOW); digitalWrite(motor1Pin2, HIGH); }

ledcWrite(enable1Pin, dutyCycle);

Serial.print("Moving "); Serial.print(direction); Serial.print(" at "); Serial.print(speedPercentage); Serial.print("% ("); Serial.print(dutyCycle); Serial.println(" duty cycle)"); }

// Motor control task void motorControlTask(void *parameter) { String motorCommand; unsigned long lastPedalTime = 0; const int decelInterval = 150; const int noPressTimeout = 400;

while (true) { if (!SerialBT.hasClient()) { ledcWrite(enable1Pin, 0); currentSpeedPercentage = 0; vTaskDelay(100 / portTICK_PERIOD_MS); continue; }
```

```
if (xQueueReceive(motorQueue, &motorCommand, 10 / portTICK_PERIOD_MS))
{
  if (motorCommand == "ff") {
    goingForward = true;
    lastPedalTime = millis();
    currentSpeedPercentage = min(currentSpeedPercentage + 5, 100);
    moveMotor(currentSpeedPercentage, "forward");
  }
  else if (motorCommand == "bb") {
    goingForward = false;
    lastPedalTime = millis();
    currentSpeedPercentage = min(currentSpeedPercentage + 5, 100);
    moveMotor(currentSpeedPercentage, "backward");
  }
  else if (motorCommand == "ss") {
    lastPedalTime = 0;
  }
}

if (millis() - lastPedalTime > noPressTimeout &&
currentSpeedPercentage > 0) {
  static unsigned long lastDecel = 0;
  if (millis() - lastDecel >= decelInterval) {
```

```
    currentSpeedPercentage = max(currentSpeedPercentage - 30, 0);
    moveMotor(currentSpeedPercentage, goingForward ? "forward" :
"backward");
    lastDecel = millis();
  }
}

vTaskDelay(10 / portTICK_PERIOD_MS);


} }
```

// Servo control task void servoControlTask(void *parameter) { String servoCommand; while (true) { if (xQueueReceive(servoQueue, &servoCommand, portMAX_DELAY)) { turnServo(servoCommand); } } }

void setup() { // Diagnostic LED pinMode(diagLedPin, OUTPUT); digitalWrite(diagLedPin, LOW);

// Motor pins pinMode(motor1Pin1, OUTPUT); pinMode(motor1Pin2, OUTPUT); pinMode(enable1Pin, OUTPUT); ledcAttachChannel(enable1Pin, freq, resolution, pwmChannel);

// Servo myServo.attach(servoPin); myServo.write(servoAngle);

// Serial Serial.begin(115200); delay(500);

// Print reset reason esp_reset_reason_t reason = esp_reset_reason(); Serial.print("🔄 Boot! Reset reason: "); Serial.println(resetReasonToString(reason));

// Start Bluetooth SPP (no PIN) if (!SerialBT.begin("ESP32test")) { Serial.println("❌ BT init failed!"); while (1) { delay(100); } } Serial.println("✅ Bluetooth SPP started as "ESP32test"");

// Create queues and tasks motorQueue = xQueueCreate(10, sizeof(String)); servoQueue = xQueueCreate(10, sizeof(String)); xTaskCreatePinnedToCore(motorControlTask, "Motor Task", 10000, NULL, 1, NULL, 0); xTaskCreatePinnedToCore(servoControlTask, "Servo Task", 10000, NULL, 1, NULL, 1); }

void loop() { // Connection diagnostics bool isConnected = SerialBT.hasClient(); if (isConnected != wasConnected) { Serial.println(isConnected ? "🔗 Client connected" : "❌ Client disconnected"); digitalWrite(diagLedPin, isConnected ? HIGH : LOW); wasConnected = isConnected; }

// Heartbeat if (millis() - lastHeartbeat >= 5000) { lastHeartbeat = millis(); Serial.print("⏱️ Uptime: "); Serial.print(millis() / 1000); Serial.print(" s, Free heap: "); Serial.println(ESP.getFreeHeap()); }

// Handle incoming BLE commands if (SerialBT.available()) { String input; while (SerialBT.available()) { char c = SerialBT.read(); if (c == '\n' || c == '\r') break; input += c; } input.trim(); input.toLowerCase();

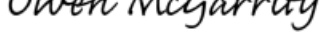
```
// Normalize single-letter to double-letter
if      (input == "f") input = "ff";
else if (input == "b") input = "bb";
else if (input == "s") input = "ss";
else if (input == "r") input = "rr";
else if (input == "l") input = "ll";
else if (input == "i") input = "ii";
else if (input == "g") input = "gg";
else if (input == "h") input = "hh";
else if (input == "j") input = "jj";

Serial.print("Received input: ");
Serial.println(input);

// Dispatch commands
if      (input == "ii") {
  String m = "ff", s = "rr";
  xQueueSend(motorQueue, &m, portMAX_DELAY);
  xQueueSend(servoQueue, &s, portMAX_DELAY);
}
else if (input == "gg") {
  String m = "ff", s = "ll";
  xQueueSend(motorQueue, &m, portMAX_DELAY);
```

```
  xQueueSend(servoQueue, &s, portMAX_DELAY);
}
else if (input == "hh") {
  String m = "bb", s = "ll";
  xQueueSend(motorQueue, &m, portMAX_DELAY);
  xQueueSend(servoQueue, &s, portMAX_DELAY);
}
else if (input == "jj") {
  String m = "bb", s = "rr";
  xQueueSend(motorQueue, &m, portMAX_DELAY);
  xQueueSend(servoQueue, &s, portMAX_DELAY);
}
else if (input == "ff" || input == "bb" || input == "ss") {
  xQueueSend(motorQueue, &input, portMAX_DELAY);
}
else if (input == "rr" || input == "ll") {
  xQueueSend(servoQueue, &input, portMAX_DELAY);
}



}
```

delay(10); }

Concurrence Table

Name	Role	Signature / Electronic Signature
Alexander Krupinski	Author/Peer Review	

Owen McGarrity	Author/Peer Review	
Mark Camitan	Author/Peer Review	
Benjamin Weisfeld	Author/Peer Review	
Ryan Schubert	Author/Peer Review	
Ryan Zhang	Author/Peer Review	
Wafi Danesh	Advisor/Advisor Review	
Kerry Ford	Co-Advisor/Co-Advisor Review