كلية الحاسبات و الذكاء الاصطناعى
جامعة القاهرة

جامعة القاهرة

[Auto parking]

1 _ Mahmoud Ayman Abdelhamid	20200494
2_ shreen Osama Abdelrahman	20200263
3 _SEIF-ALDEEN TAREK MOHAMED	20210600
4_ Ahmed Yasser Ahmed mesherf	20200061
5_ ABDELRAHMAN AHMED	20200280
6_ YOUSSEF MOHAMED	20200661

Abstract:

This project presents an innovative Autonomous Car Parking System (ACPS) designed to revolutionize traditional parking infrastructure by integrating ESP-based technology. The ACPS leverages ESP32/ESP8266 modules to enable vehicles to autonomously navigate and park within designated parking areas, streamlining the parking process and enhancing user convenience.

The system integrates ultrasonic sensors and ESP microcontrollers to facilitate real-time data acquisition and processing. By harnessing the ESP's computational capabilities, the ACPS interprets sensor data to detect and analyze parking space availability, guiding vehicles to suitable parking spots with precision and efficiency.

Key objectives of the project include reducing parking congestion, optimizing parking space utilization, and minimizing human intervention in parking maneuvers. Through seamless

Key objectives of the project include reducing parking congestion, optimizing parking space utilization, and minimizing human intervention in parking maneuvers. Through seamless communication between vehicles and the ACPS, cars can autonomously navigate, park, and exit parking areas, enhancing overall traffic flow, and reducing the time spent searching for parking.

Background:

Our robotics project specifically in auto parking car we have challenges relative to software, hardware and combining them both to do the project well.

Then we explain our project components in each system:

1 –Mechanical system: A system concerned in car motions and kinematics containing Dc motors, wheels (Actuator).

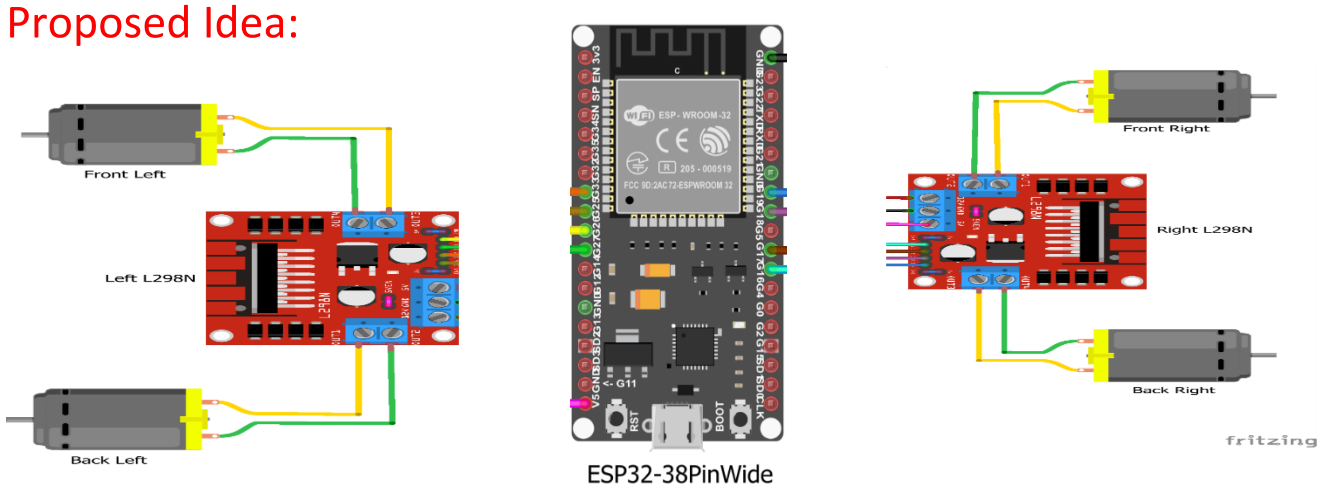
2 – Electrical system: control the voltages and the pulses in the electrical devices containing two l298n (motor driver), batteries (22V).

3 – sensors: read and sense signals in the surrounding environment, we used three ultrasonic (sr04).

4 – microcomputer module: it’s a platform that has microcontroller and pins to interface with another hardware chips, Esp32.

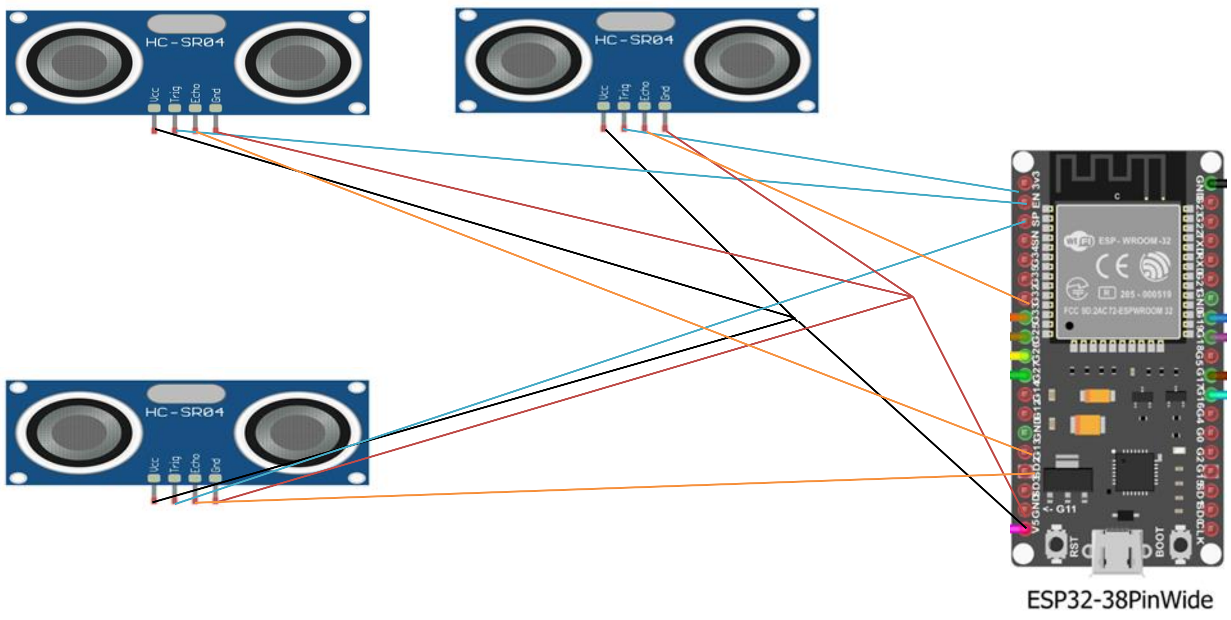
5 – software.

Proposed Idea:

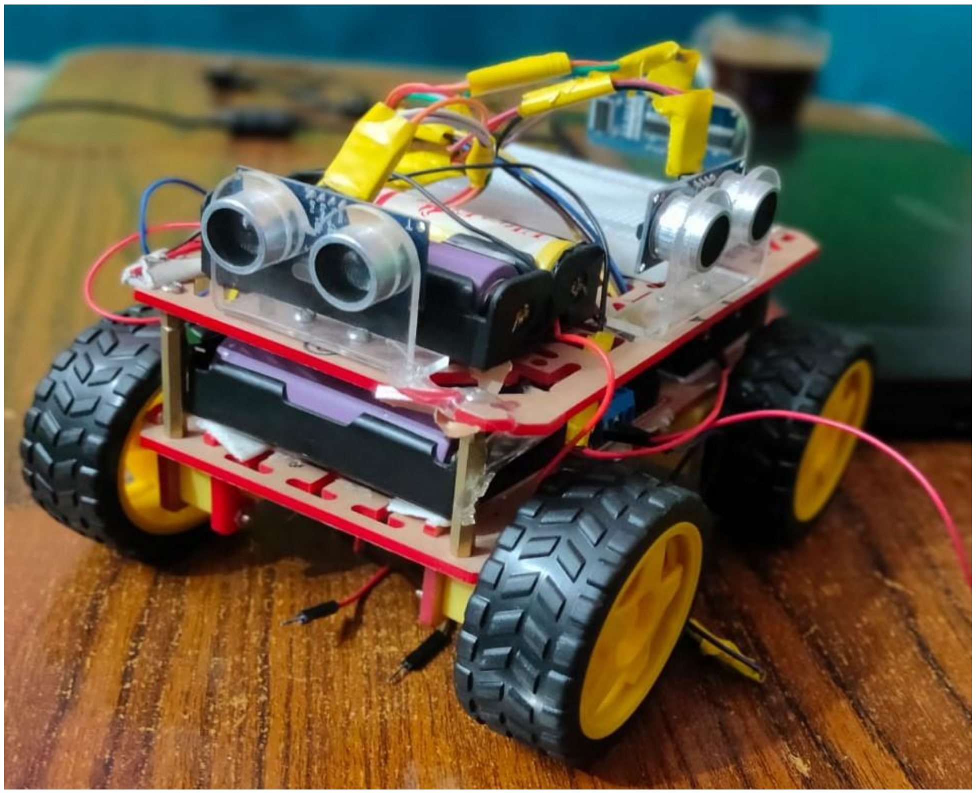


1 –Motors interfacing: we follow a differential steering approach which concerns in steering the four wheels to turn left and turn right so we used two motor drivers each one control a side using three batteries for each one powering 12V.

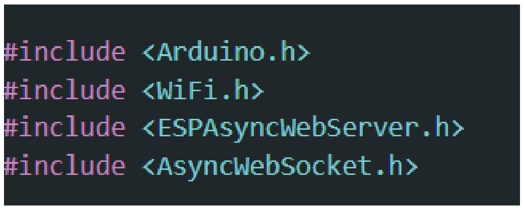
2 – ultrasonics interface: we are using three ultrasonics, the first one forward, the second one right, the third one backward, we put three constraint to all sensors in software to auto parking successfully.



Therfore, The final result ☺



software:



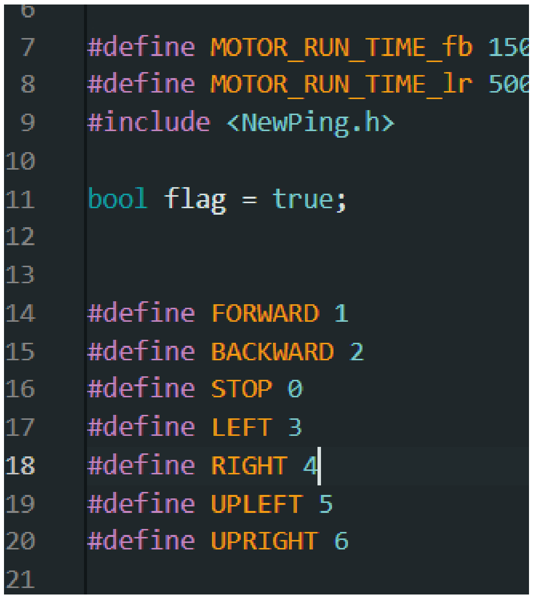
```
#include <Arduino.h>
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <AsyncWebSocket.h>
```

'Arduino.h': The core Arduino library, providing essential functions and definitions for Arduino programming.

'WiFi.h': This library allows the Arduino board to connect to a Wi-Fi network, enabling it to communicate over a wireless network.

'ESPAsyncWebServer.h': An asynchronous web server library designed for the ESP8266 and ESP32 boards. It handles HTTP requests and responses asynchronously, which can be more efficient in handling multiple connections compared to synchronous servers.

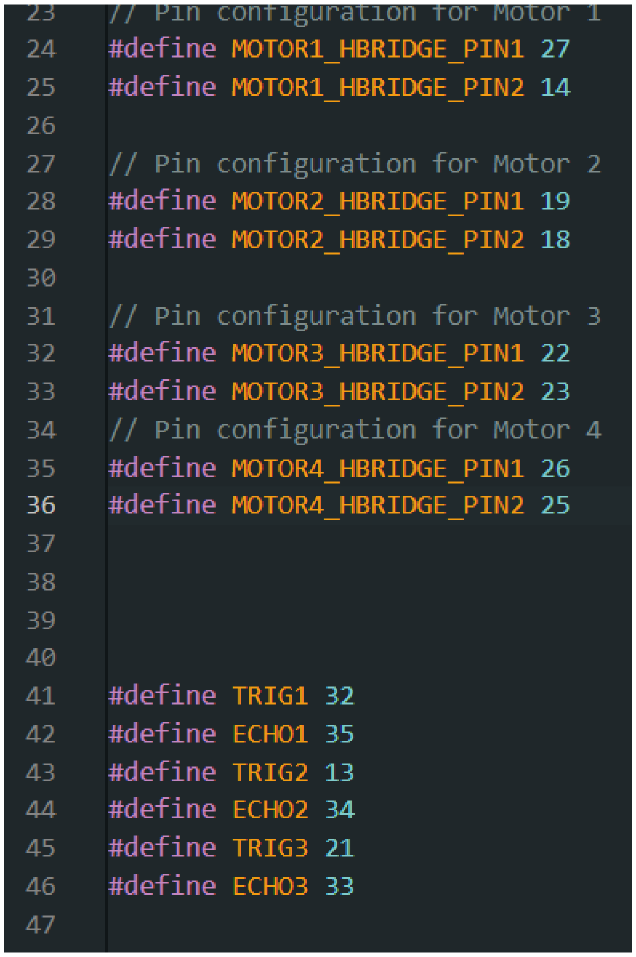
'AsyncWebSocket.h': This library adds support for WebSocket communication to the ESPAsyncWebServer. WebSocket allow for full-duplex communication channels over a single, long-lived connection, often used for real-time applications.



```
#define MOTOR_RUN_TIME_fb 15
#define MOTOR_RUN_TIME_lr 50
#include <NewPing.h>

bool flag = true;


#define FORWARD 1
#define BACKWARD 2
#define STOP 0
#define LEFT 3
#define RIGHT 4
#define UPLEFT 5
#define UPRIGHT 6

```

MOTOR_RUN_TIME_fb and MOTOR_RUN_TIME_lr: These constants are likely used to define the duration for which the motors will run when moving forward and when turning left or right.

Directional Constants (FORWARD, BACKWARD, STOP, LEFT, RIGHT, UPLEFT, UPRIGHT): These constants likely represent different directions or actions for controlling motors or indicating movement.



```
// Pin configuration for Motor 1
#define MOTOR1_HBRIDGE_PIN1 27
#define MOTOR1_HBRIDGE_PIN2 14

// Pin configuration for Motor 2
#define MOTOR2_HBRIDGE_PIN1 19
#define MOTOR2_HBRIDGE_PIN2 18

// Pin configuration for Motor 3
#define MOTOR3_HBRIDGE_PIN1 22
#define MOTOR3_HBRIDGE_PIN2 23
// Pin configuration for Motor 4
#define MOTOR4_HBRIDGE_PIN1 26
#define MOTOR4_HBRIDGE_PIN2 25




#define TRIG1 32
#define ECHO1 35
#define TRIG2 13
#define ECHO2 34
#define TRIG3 21
#define ECHO3 33

```

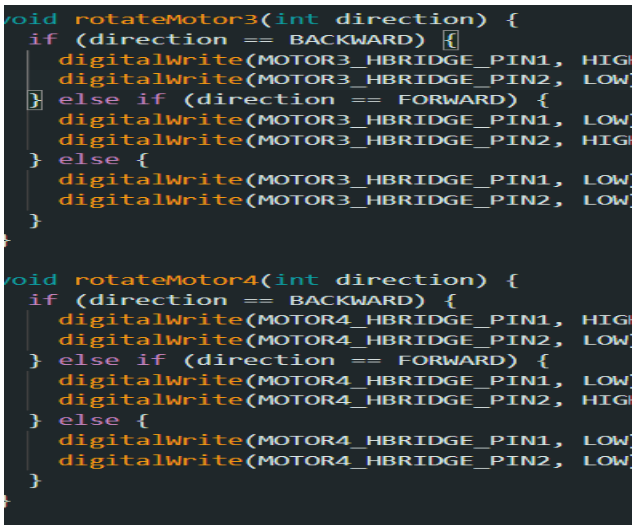
MOTOR1_HBRIDGE_PIN1 and MOTOR1_HBRIDGE_PIN2: These constants likely represent the pins connected to the H-bridge for controlling Motor 1 in a dual-H-bridge motor driver setup. They correspond to the input pins (IN1 and IN2) for controlling the direction and speed of the motor.

Motor 2, 3, 4: Like Motor 1, the code assigns specific pins (IN1 and IN2) for Motor 2, 3, and 4, assuming they're connected to an H-bridge driver for motor control.

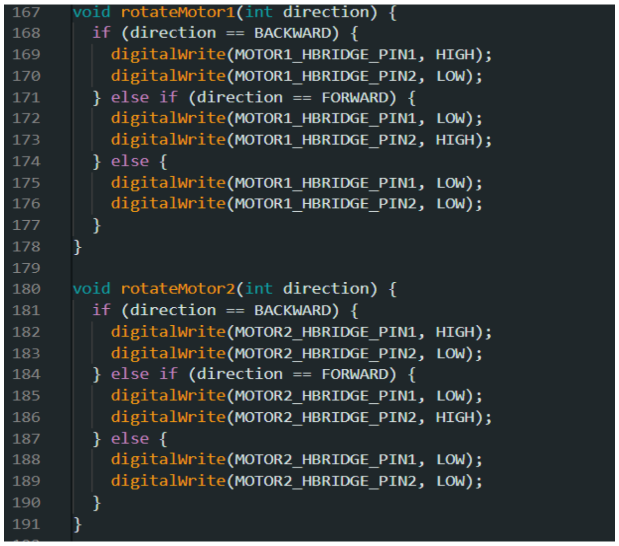
Ultrasonic Sensor Pin Configuration:

Ultrasonic Sensor 1 (TRIG1 and ECHO1): TRIG1 and ECHO1 likely correspond to the trigger and echo pins of an ultrasonic distance sensor (such as an HC-SR04 or similar). TRIG1 is likely the trigger pin used to send ultrasonic pulses, and ECHO1 is likely the pin used to receive the echo and calculate the distance.

Ultrasonic Sensor 2 ,3(TRIG2 and ECHO2): Like Ultrasonic Sensor 1, these constants represent the trigger and echo pins for a second ultrasonic sensor.



```
oid rotateMotor3(int direction) {
  if (direction == BACKWARD) {
    digitalWrite(MOTOR3_HBRIDGE_PIN1, HIG
    digitalWrite(MOTOR3_HBRIDGE_PIN2, LOW
  } else if (direction == FORWARD) {
    digitalWrite(MOTOR3_HBRIDGE_PIN1, LOW
    digitalWrite(MOTOR3_HBRIDGE_PIN2, HIG
  } else {
    digitalWrite(MOTOR3_HBRIDGE_PIN1, LOW
    digitalWrite(MOTOR3_HBRIDGE_PIN2, LOW
  }

oid rotateMotor4(int direction) {
  if (direction == BACKWARD) {
    digitalWrite(MOTOR4_HBRIDGE_PIN1, HIG
    digitalWrite(MOTOR4_HBRIDGE_PIN2, LOW
  } else if (direction == FORWARD) {
    digitalWrite(MOTOR4_HBRIDGE_PIN1, LOW
    digitalWrite(MOTOR4_HBRIDGE_PIN2, HIG
  } else {
    digitalWrite(MOTOR4_HBRIDGE_PIN1, LOW
    digitalWrite(MOTOR4_HBRIDGE_PIN2, LOW
  }
```



```
void rotateMotor1(int direction) {
  if (direction == BACKWARD) {
    digitalWrite(MOTOR1_HBRIDGE_PIN1, HIGH);
    digitalWrite(MOTOR1_HBRIDGE_PIN2, LOW);
  } else if (direction == FORWARD) {
    digitalWrite(MOTOR1_HBRIDGE_PIN1, LOW);
    digitalWrite(MOTOR1_HBRIDGE_PIN2, HIGH);
  } else {
    digitalWrite(MOTOR1_HBRIDGE_PIN1, LOW);
    digitalWrite(MOTOR1_HBRIDGE_PIN2, LOW);
  }
}

void rotateMotor2(int direction) {
  if (direction == BACKWARD) {
    digitalWrite(MOTOR2_HBRIDGE_PIN1, HIGH);
    digitalWrite(MOTOR2_HBRIDGE_PIN2, LOW);
  } else if (direction == FORWARD) {
    digitalWrite(MOTOR2_HBRIDGE_PIN1, LOW);
    digitalWrite(MOTOR2_HBRIDGE_PIN2, HIGH);
  } else {
    digitalWrite(MOTOR2_HBRIDGE_PIN1, LOW);
    digitalWrite(MOTOR2_HBRIDGE_PIN2, LOW);
  }
}
```

These functions rotateMotor1(), rotateMotor2(), rotateMotor3(), and rotateMotor4() appear to control the direction of four different motors based on the provided direction parameter. Here's a breakdown of each function:

Motor Control Logic:

direction == BACKWARD:

If the provided direction is BACKWARD, the function sets the MOTOR1_HBRIDGE_PIN1 to HIGH and MOTOR1_HBRIDGE_PIN2 to LOW.

This configuration in an H-bridge typically causes the motor to rotate in the backward direction by applying a specific voltage to the motor terminals.

direction == FORWARD:

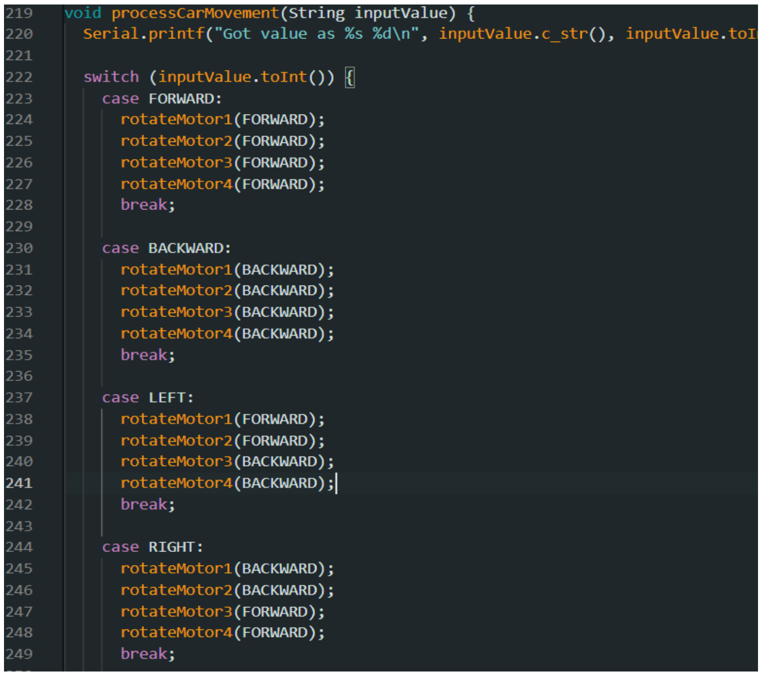
If the provided direction is FORWARD, it sets MOTOR1_HBRIDGE_PIN1 to LOW and MOTOR1_HBRIDGE_PIN2 to HIGH.

This setting usually makes the motor rotate in the forward direction by reversing the voltage polarity across the motor terminals.

Other Directions or STOP:

If the provided direction is neither FORWARD nor BACKWARD, it sets both MOTOR1_HBRIDGE_PIN1 and MOTOR1_HBRIDGE_PIN2 to LOW.

This state typically stops the motor by removing power from both sides of the motor.



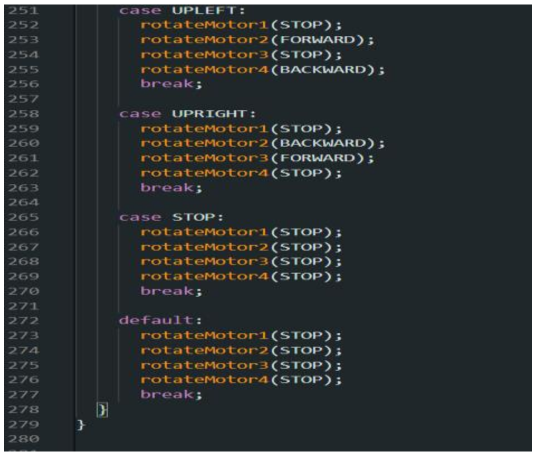
```
void processCarMovement(String inputValue) {
  Serial.printf("Got value as %s %d\n", inputValue.c_str(), inputValue.toI

  switch (inputValue.toInt()) {
    case FORWARD:
      rotateMotor1(FORWARD);
      rotateMotor2(FORWARD);
      rotateMotor3(FORWARD);
      rotateMotor4(FORWARD);
      break;

    case BACKWARD:
      rotateMotor1(BACKWARD);
      rotateMotor2(BACKWARD);
      rotateMotor3(BACKWARD);
      rotateMotor4(BACKWARD);
      break;

    case LEFT:
      rotateMotor1(FORWARD);
      rotateMotor2(FORWARD);
      rotateMotor3(BACKWARD);
      rotateMotor4(BACKWARD);
      break;

    case RIGHT:
      rotateMotor1(BACKWARD);
      rotateMotor2(BACKWARD);
      rotateMotor3(FORWARD);
      rotateMotor4(FORWARD);
      break;
```

FORWARD: Sets all four motors (rotateMotor1() to rotateMotor4()) to move forward.

BACKWARD: Sets all four motors to move backward.

LEFT: Configures the motors to turn left by causing motors 1 and 2 to move forward and motors 3 and 4 to move backward.

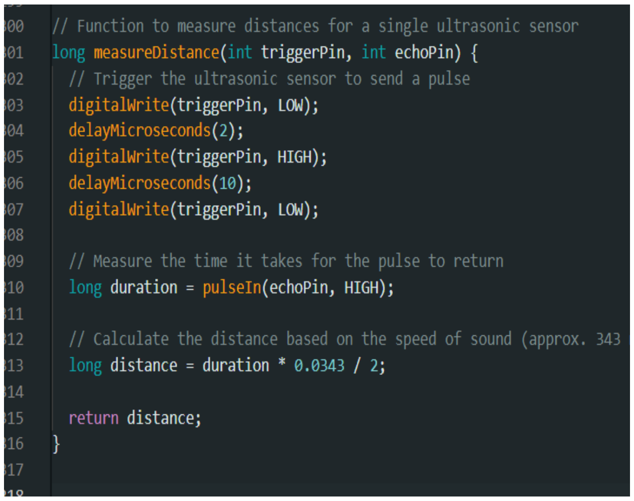
RIGHT: Configures the motors to turn right by causing motors 1 and 2 to move backward and motors 3 and 4 to move forward.



```
    case UPLEFT:
      rotateMotor1(STOP);
      rotateMotor2(FORWARD);
      rotateMotor3(STOP);
      rotateMotor4(BACKWARD);
      break;

    case UPRIGHT:
      rotateMotor1(STOP);
      rotateMotor2(BACKWARD);
      rotateMotor3(FORWARD);
      rotateMotor4(STOP);
      break;

    case STOP:
      rotateMotor1(STOP);
      rotateMotor2(STOP);
      rotateMotor3(STOP);
      rotateMotor4(STOP);
      break;

    default:
      rotateMotor1(STOP);
      rotateMotor2(STOP);
      rotateMotor3(STOP);
      rotateMotor4(STOP);
      break;
  }
}
```

UPLEFT: Stops Motor 1, makes Motor 2 move FORWARD, stops Motor 3, and makes Motor 4 move BACKWARD. This configuration seems to execute a movement pattern resembling a forward-left turn.

UPRIGHT: Stops Motor 1, makes Motor 2 move BACKWARD, makes Motor 3 move FORWARD, and stops Motor 4. This configuration appears to execute a movement pattern resembling a forward-right turn.



```
// Function to measure distances for a single ultrasonic sensor
long measureDistance(int triggerPin, int echoPin) {
  // Trigger the ultrasonic sensor to send a pulse
  digitalWrite(triggerPin, LOW);
  delayMicroseconds(2);
  digitalWrite(triggerPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(triggerPin, LOW);

  // Measure the time it takes for the pulse to return
  long duration = pulseIn(echoPin, HIGH);

  // Calculate the distance based on the speed of sound (approx. 343
  long distance = duration * 0.0343 / 2;

  return distance;
}
```

Distance Measurement Process

Triggering Ultrasonic Sensor:

Sets the trigger Pin to LOW, provides a small delay of 2 microseconds, then triggers the sensor by setting trigger Pin to HIGH for 10 microseconds and immediately setting it back to LOW.

This sequence prompts the sensor to send out an ultrasonic pulse.

Measuring Pulse Return:

Utilizes pulse in (echo Pin, HIGH) to measure the duration for which the echoPin remains HIGH.

pulse in () measures the time taken for the ultrasonic pulse to bounce off an object and return to the sensor.

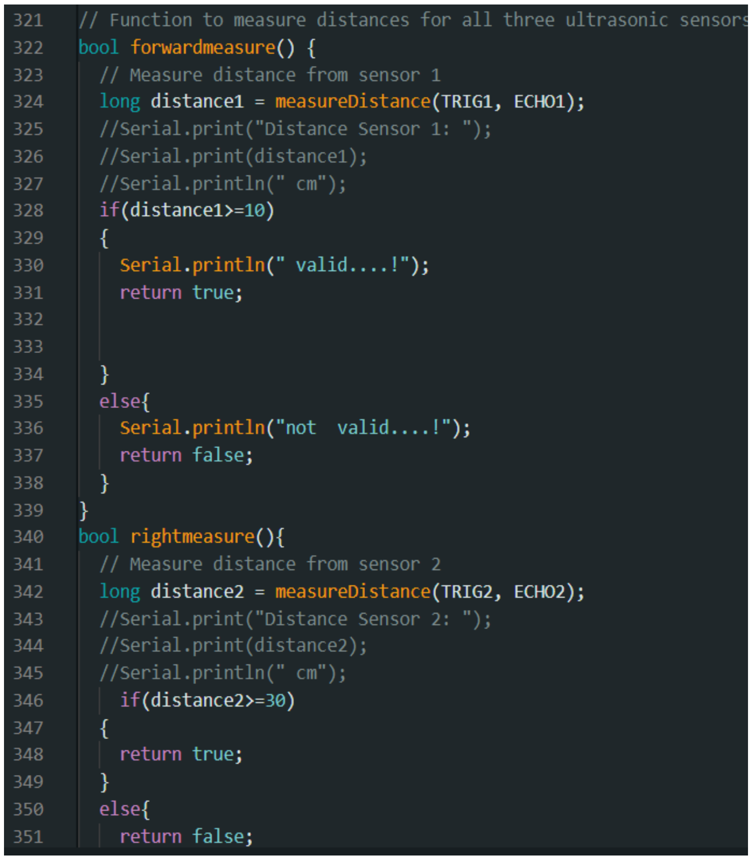
Distance Calculation:

Calculates the distance based on the time taken by the pulse to return.

Uses the formula: distance = duration * speed Of Sound / 2, where speed Of Sound is the speed of sound in air (approximately 343 meters/second).

Dividing by 2 because the pulse travels to the object and back, so the calculated duration corresponds to the one-way travel time.

Return Value: Returns the calculated distance in centimeters.



```
// Function to measure distances for all three ultrasonic sensors
bool forwardmeasure() {
  // Measure distance from sensor 1
  long distance1 = measureDistance(TRIG1, ECHO1);
  //Serial.print("Distance Sensor 1: ");
  //Serial.print(distance1);
  //Serial.println(" cm");
  if(distance1>=10)
  {
    Serial.println(" valid....!");
    return true;


  }
  else{
    Serial.println("not  valid....!");
    return false;
  }
}
bool rightmeasure(){
  // Measure distance from sensor 2
  long distance2 = measureDistance(TRIG2, ECHO2);
  //Serial.print("Distance Sensor 2: ");
  //Serial.print(distance2);
  //Serial.println(" cm");
    if(distance2>=30)
  {
    return true;
  }
  else{
    return false;
```

forward measure (): Sensor 1 Measurement

Uses measure Distance () with TRIG1 and ECHO1 to measure the distance from sensor 1.

Distance Check:

If the distance measured (distance1) is greater than or equal to 10 centimeters, it prints "valid" to the serial monitor and returns true. This suggests that there is enough space in the forward direction.

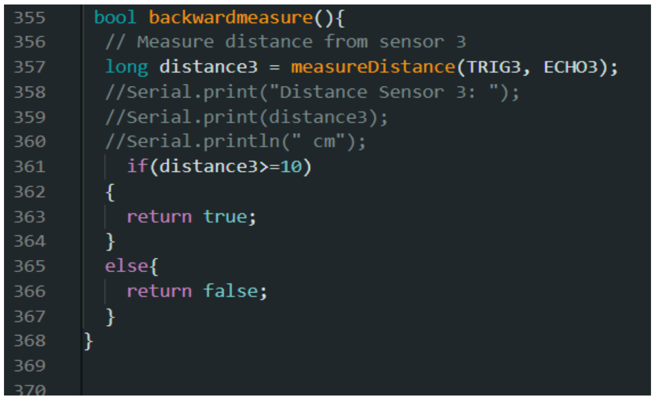
Otherwise, it prints "not valid" to the serial monitor and returns false, indicating that the space ahead might be too close or obstructed.

right measure () Sensor 2 Measurement:

Uses measure Distance () with TRIG2 and ECHO2 to measure the distance from sensor 2.

Distance Check:

If the distance measured (distance2) is greater than or equal to 30 centimeters, it returns true. This suggests that there is sufficient space to the right, Otherwise, it returns false, indicating that the space to the right might be too close or obstructed.



```
bool backwardmeasure(){
 // Measure distance from sensor 3
 long distance3 = measureDistance(TRIG3, ECHO3);
 //Serial.print("Distance Sensor 3: ");
 //Serial.print(distance3);
 //Serial.println(" cm");
   if(distance3>=10)
 {
   return true;
 }
 else{
   return false;
 }
}
```

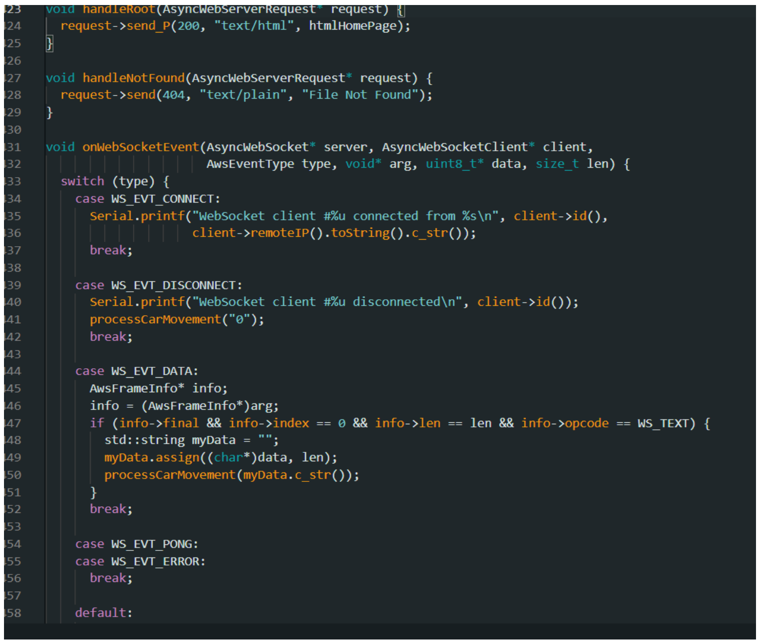
backward measure () Sensor 3 Measurement:

Uses measure Distance () with TRIG3 and ECHO3 to measure the distance from sensor 3.

Distance Check:

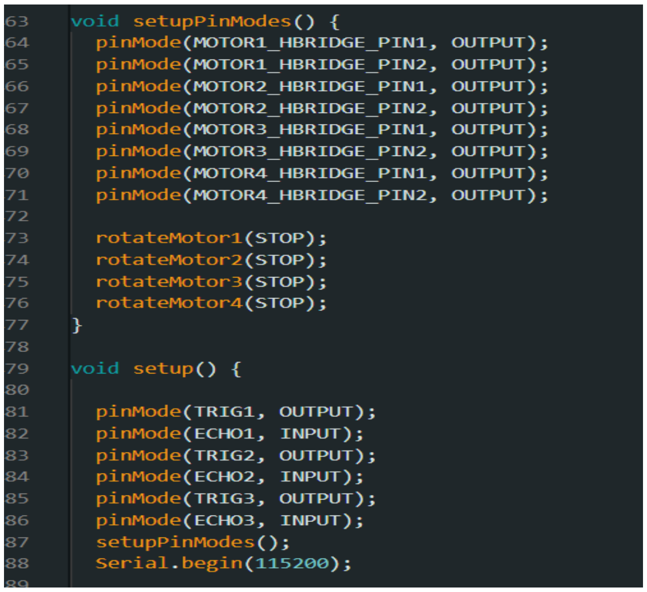
If the measured distance (distance3) is greater than or equal to 10 centimeters, it returns true. This suggests that there is sufficient space behind (in the backward direction).

If the distance is less than 10 centimeters, it returns false, indicating that the space behind might be too close or obstructed.



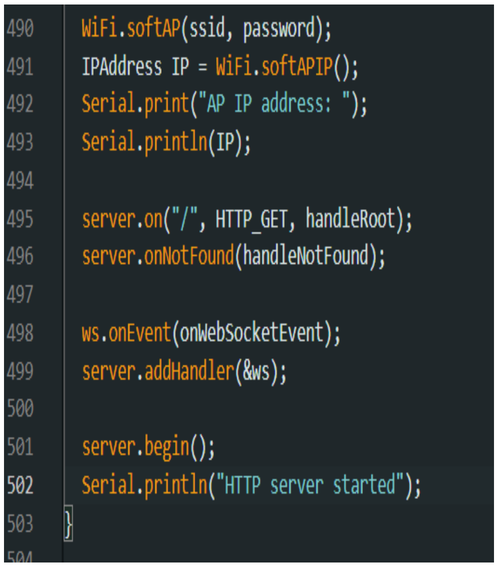
```cpp
void handleRoot(AsyncWebServerRequest* request) {
  request->send_P(200, "text/html", htmlHomePage);
}

void handleNotFound(AsyncWebServerRequest* request) {
  request->send(404, "text/plain", "File Not Found");
}

void onWebSocketEvent(AsyncWebSocket* server, AsyncWebSocketClient* client,
                      AwsEventType type, void* arg, uint8_t* data, size_t len) {
  switch (type) {
    case WS_EVT_CONNECT:
      Serial.printf("WebSocket client #%u connected from %s\n", client->id(),
                    client->remoteIP().toString().c_str());
      break;

    case WS_EVT_DISCONNECT:
      Serial.printf("WebSocket client #%u disconnected\n", client->id());
      processCarMovement("0");
      break;

    case WS_EVT_DATA:
      AwsFrameInfo* info;
      info = (AwsFrameInfo*)arg;
      if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        std::string myData = "";
        myData.assign((char*)data, len);
        processCarMovement(myData.c_str());
      }
      break;

    case WS_EVT_PONG:
    case WS_EVT_ERROR:
      break;

    default:
```

These functions are part of a WebSocket-based server, with handle Root() serving an HTML page, handleNotFound() handling missing resources, and onWebSocketEvent() managing WebSocket events like client connections, disconnections, and incoming data, allowing control of car movements based on received commands over the WebSocket connection.



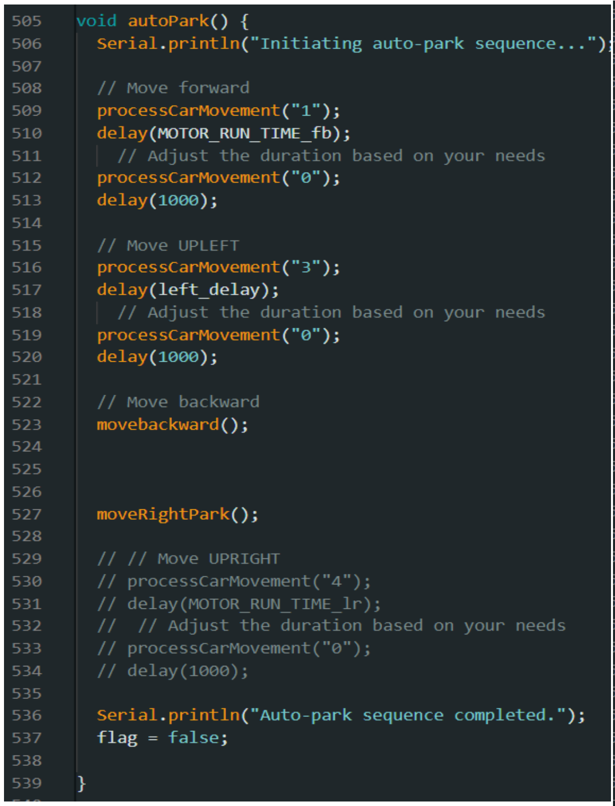
```cpp
void setupPinModes() {
  pinMode(MOTOR1_HBRIDGE_PIN1, OUTPUT);
  pinMode(MOTOR1_HBRIDGE_PIN2, OUTPUT);
  pinMode(MOTOR2_HBRIDGE_PIN1, OUTPUT);
  pinMode(MOTOR2_HBRIDGE_PIN2, OUTPUT);
  pinMode(MOTOR3_HBRIDGE_PIN1, OUTPUT);
  pinMode(MOTOR3_HBRIDGE_PIN2, OUTPUT);
  pinMode(MOTOR4_HBRIDGE_PIN1, OUTPUT);
  pinMode(MOTOR4_HBRIDGE_PIN2, OUTPUT);

  rotateMotor1(STOP);
  rotateMotor2(STOP);
  rotateMotor3(STOP);
  rotateMotor4(STOP);
}

void setup() {

  pinMode(TRIG1, OUTPUT);
  pinMode(ECHO1, INPUT);
  pinMode(TRIG2, OUTPUT);
  pinMode(ECHO2, INPUT);
  pinMode(TRIG3, OUTPUT);
  pinMode(ECHO3, INPUT);
  setupPinModes();
  Serial.begin(115200);
```

These functions collectively initialize the necessary pins for ultrasonic sensors and motor control. The setup () function ensures proper initialization of serial communication, ultrasonic sensor pins, and then calls setupPinModes () to set up the motor control pins and ensure all motors start in a stopped state before the main code execution begins.

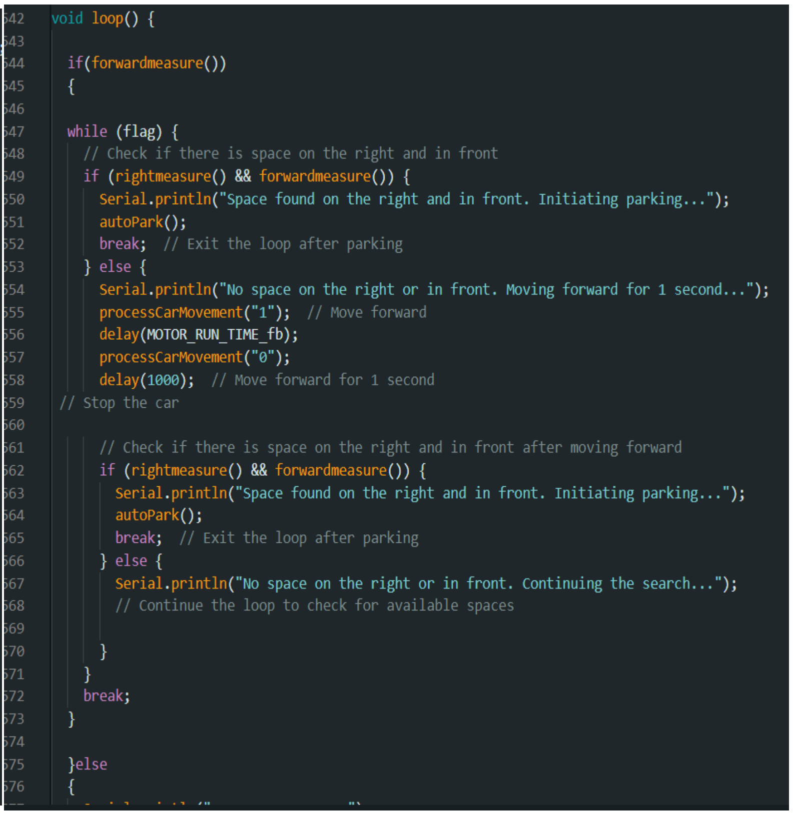


```
  WiFi.softAP(ssid, password);
  IPAddress IP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(IP);

  server.on("/", HTTP_GET, handleRoot);
  server.onNotFound(handleNotFound);

  ws.onEvent(onWebSocketEvent);
  server.addHandler(&ws);

  server.begin();
  Serial.println("HTTP server started");
}
```

This code segment initializes a Soft AP, sets up a basic HTTP server to handle root requests and not-found cases, configures a WebSocket event handler, and starts the server, allowing clients to connect to the ESP8266/ESP32 via a web interface and WebSocket communication. The Soft AP mode enables the device to act as an access point, and clients can interact with it through HTTP and WebSocket protocols.



```
void autoPark() {
  Serial.println("Initiating auto-park sequence...");

  // Move forward
  processCarMovement("1");
  delay(MOTOR_RUN_TIME_fb);
    // Adjust the duration based on your needs
  processCarMovement("0");
  delay(1000);

  // Move UPLEFT
  processCarMovement("3");
  delay(left_delay);
    // Adjust the duration based on your needs
  processCarMovement("0");
  delay(1000);

  // Move backward
  movebackward();



  moveRightPark();

  // // Move UPRIGHT
  // processCarMovement("4");
  // delay(MOTOR_RUN_TIME_lr);
  //  // Adjust the duration based on your needs
  // processCarMovement("0");
  // delay(1000);

  Serial.println("Auto-park sequence completed.");
  flag = false;

}
```



```
void loop() {

  if(forwardmeasure())
  {

  while (flag) {
    // Check if there is space on the right and in front
    if (rightmeasure() && forwardmeasure()) {
      Serial.println("Space found on the right and in front. Initiating parking...");
      autoPark();
      break;  // Exit the loop after parking
    } else {
      Serial.println("No space on the right or in front. Moving forward for 1 second...");
      processCarMovement("1");  // Move forward
      delay(MOTOR_RUN_TIME_fb);
      processCarMovement("0");
      delay(1000);  // Move forward for 1 second
  // Stop the car

      // Check if there is space on the right and in front after moving forward
      if (rightmeasure() && forwardmeasure()) {
        Serial.println("Space found on the right and in front. Initiating parking...");
        autoPark();
        break;  // Exit the loop after parking
      } else {
        Serial.println("No space on the right or in front. Continuing the search...");
        // Continue the loop to check for available spaces

      }
    }
    break;
  }

  }else
  {
```

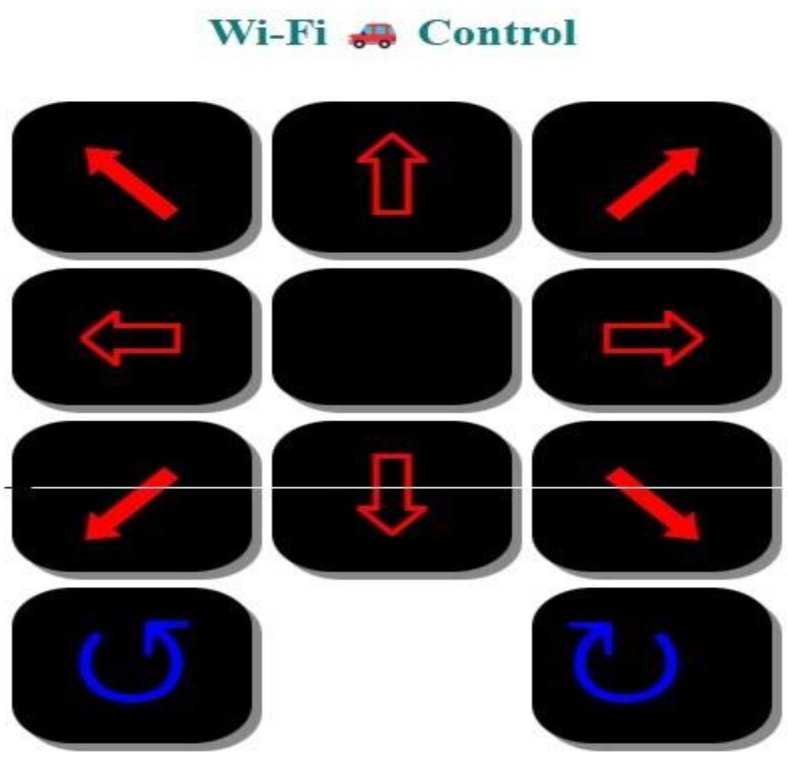
In the function auto parking the mechanism of the park is printing (Initiating auto-park sequence…!) , then move forward with motor run time fb , then stop for 1 second and when the delay finished car will move to the left by the left run time then stop for 1 second and when the delay finished the car move backward with the function move backward() then the car move to the right with function moveRightPark(),and print “Auto-park sequence completed.” And give the flag false value.

In the void loop we use all functions we build for the application and test the auto parking on the car by doing the next steps:

check the distance at the forward of the car by the function forward measure(), if it give a true value you will go the while loop and check the flag if the flag equal true you will go the next if condition if the right measure() function and the forward measure() function equals true you will call the auto park() function and the car will parking the break and give the flag false value the void loop will break. If the one of the two functions right measure() function and the forwardmeasure() function equal false ,the car will go to else then will move forward with the motor run time fl then stop and delay 1 second and then check the right measure() function and the forwardmeasure() function equals true will call the function auto-park ad repeat the all above sequence to park if the car find the forward measure function equal false it will go to the else and stop.

Application to control the auto parking:

We use the HTML for control the auto parking:



HTML Section:

Document Type Declaration: Specifies the document type as HTML5.

Viewport Meta Tag: Sets the initial viewport scale and disables user scaling, This tag controls how the webpage is displayed on different devices, ensuring it initially scales to fit the device's width without allowing user scaling.

Internal CSS: Defines styling rules for various elements on the webpage, including arrows, table cells, and touch behavior, Defines the visual appearance of elements in the webpage,

, arrows and circular Arrows classes are used to style directional and circular arrow icons, respectively.

‘td’ styling includes a black background, rounded corners, and a shadow effect. The’: active’ pseudo-class removes the shadow when a ‘td’ element is touched.

Body Section:

Header: Displays the title and subtitles for the webpage.

Table: Contains a layout of directional buttons within cells.

The buttons (<td> elements) are touch-sensitive, triggering onTouchStartAndEnd () JavaScript function calls upon touch events.

These buttons likely represent directional controls (forward, backward, left, right) and possibly other functionalities.

JavaScript Section:

WebSocket Initialization: Defines a WebSocket URL based on the current host and attempts to establish a WebSocket connection, WebSocket URL is constructed based on the current hostname, presuming a WebSocket server running on the same host, and attempts to establish a WebSocket connection.

Functions:

initWebSocket (): Initializes the WebSocket connection and handles open/close events.

onTouchStartAndEnd(value): Sends a specified value over the WebSocket connection when a touch event occurs on the buttons.

Event Listeners:

Attaches a touch-end event listener to the ‘mainTable’ element to prevent default touch behavior, ensuring that touch events on the table do not trigger default actions, such as scrolling or zooming.

Localization:

Title: Introduction to Localization and RSSI

Localization in Wireless Communication:

Localization refers to determining the physical position of an object or entity in a given space. In wireless communication, this process involves various techniques to pinpoint the location of devices or assets.

Understanding RSSI: RSSI, or Received Signal Strength Indication, is a measure used to assess the power level of a received signal in a wireless communication system. It provides an estimate of the signal strength between devices, Localization Techniques and Methods

Classification of Localization Techniques:

Various methods, including GPS-based, proximity-based, and fingerprinting, are employed for localization purposes. Each method utilizes different technologies and algorithms.

RSSI-Based Localization:

RSSI plays a crucial role in wireless localization, especially in techniques using signal strength for determining device proximity. However, factors like environmental interference and hardware limitations can affect its accuracy.

Title: Wi-Fi-Based Robot Localization Using RSSI

Overview:

This code implements a Wi-Fi-based localization system for a mobile robot. It utilizes Received Signal Strength Indication (RSSI) to estimate distances between the mobile robot and stationary robots within a network. The objective is to calculate the mobile robot's position relative to the stationary robots by analyzing signal strengths.

2_ Code Structure and Initialization:

the code includes Wi-Fi and math libraries, defines network credentials, and structures data for stationary robots, each with a unique SSID, an estimated distance, and a position.

Setup Function:

Initializes serial communication for debugging purposes.

Sets the Wi-Fi mode to station (WIFI_STA) and disconnects from any previous networks.

3: Scanning and Distance Estimation:

Loop Function:

Scans for available Wi-Fi networks and iterates through the list of scanned SSIDs.

Compares scanned SSIDs with the predefined SSIDs of stationary robots.

If a match is found, the RSSI is obtained, and a function converts RSSI to estimated distance using a predefined formula.

Displays the calculated distances to each stationary robot.

RSSI to Distance Conversion:

Utilizes a formula based on RSSI to estimate distance, applying a logarithmic function.

Position Calculation

Trilateration Process:

Uses trilateration, a geometric method, to determine the mobile robot's position based on distances obtained from stationary robots.

The calculate Position () function attempts to calculate the mobile robot's position using the trilateration method.

Mathematical calculations for trilateration are initiated but require completion for accurate position determination.

Looping and Soft Access Point

Loop Continuity:

The code continuously loops, periodically scanning for nearby robots and calculating distances and positions.

After calculations, it sets up a Wi-Fi soft access point using specific SSID and password.

Summary:

The code implements a fundamental framework for Wi-Fi-based robot localization using RSSI measurements. It scans for nearby Wi-Fi networks, estimates distances based on RSSI, attempts to calculate the mobile robot's position using trilateration, and periodically updates its Wi-Fi setup.