Abstract :

This technical documentation provides a detailed insight into the design, development, and implementation of an Autonomous Parking System (APS) for a vehicle equipped with four DC motors, an L298N motor driver, four lithium batteries, a voltage step-down module, two ultrasonic sensors, a servo motor, and an ESP32-WROOM-32 microcontroller. The document outlines the system architecture, key components, and operational principles of the APS, offering a comprehensive guide for engineers, developers, and enthusiasts interested in autonomous vehicle technologies.

The documentation begins by elucidating the hardware components, including the four DC motors responsible for vehicle propulsion, the L298N motor driver facilitating motor control, and the lithium batteries configured in parallel to power the system. Details on the voltage step-down module and its role in maintaining optimal power supply are also provided, ensuring efficient utilization of the battery resources.

The integration of two ultrasonic sensors and a servo motor for environmental perception and steering control is thoroughly explained. The ESP32-WROOM-32 microcontroller serves as the brain of the system, orchestrating sensor inputs, decision-making algorithms, and motor commands to enable autonomous parking maneuvers.

A comprehensive overview of the software architecture follows, detailing the programming logic, control algorithms, and communication protocols employed by the ESP32 microcontroller. The document delves into the integration of the ultrasonic sensors for obstacle detection, the servo motor for precise steering control, and the coordination of the four DC motors for smooth and accurate parking maneuvers.

Practical considerations, such as user interfaces, feedback mechanisms, and emergency overrides, are addressed to enhance the user experience and ensure the safety of the autonomous parking process.

Furthermore, the documentation covers power management strategies, addressing the optimization of battery usage, charging considerations, and the implementation of fail-safe mechanisms to prevent unexpected system failures.

In conclusion, this documentation serves as a comprehensive resource for individuals involved in the development and understanding of an Autonomous Parking System utilizing four DC motors, an L298N motor driver, lithium batteries, ultrasonic sensors, a servo motor, and an ESP32-WROOM-32 microcontroller. It provides a thorough exploration of both the hardware and software aspects, facilitating a deeper understanding of the technology and encouraging further advancements in autonomous vehicle systems.

Background:

Modern advancements in autonomous vehicle technology have led to the integration of sophisticated hardware components to enable autonomous parking systems (APS) in automobiles. The hardware architecture designed for the autonomous parking car described in this documentation leverages a combination of advanced electronic components, sensors, and microcontrollers to achieve precise and reliable autonomous parking maneuvers.

1. DC Motors and L298N Motor Driver:

- Four DC motors are strategically positioned in the vehicle to control its movement during parking.
- The L298N motor driver acts as the interface between the microcontroller and the DC motors, providing precise control over motor speed and direction.
- This motor configuration allows for omnidirectional movement, facilitating agile and accurate adjustments during parking.

2. Power Supply System:

- The car relies on a power supply system consisting of four lithium batteries, each producing 3.7 volts.
- A voltage step-down module is incorporated to regulate and optimize the voltage supplied to various components, ensuring stable and efficient operation.

3. Ultrasonic Sensors:

- Two ultrasonic sensors are strategically placed on the vehicle to provide real-time environmental data.
- These sensors play a crucial role in obstacle detection and avoidance, contributing to the safety and precision of the autonomous parking process.

4. Servo Motor:

- A servo motor is employed for precise control of the vehicle's steering mechanism during parking.
- This component enhances the system's ability to navigate tight spaces and execute accurate parking maneuvers.

5. Microcontroller (ESP32-WROOM-32):

- The ESP32-WROOM-32 serves as the central processing unit, responsible for orchestrating the entire autonomous parking system.

- Its processing capabilities, coupled with built-in wireless communication features, make it well-suited for real-time decision-making and coordination of the various hardware components.

6. Sensors and Perception:

- The integration of ultrasonic sensors allows the vehicle to perceive its environment accurately.

- The data from these sensors are processed in real-time by the microcontroller to make informed decisions regarding obstacle detection, distance estimation, and parking trajectory planning.

7. Safety Mechanisms:

- Fail-safe mechanisms are incorporated to ensure the safety of the system, preventing unintended behaviors or malfunctions.

- Emergency overrides and user interfaces are implemented to allow manual intervention when necessary, enhancing the overall safety of the autonomous parking system.

This hardware architecture combines precision, adaptability, and safety, creating a robust foundation for the implementation of an efficient and reliable Autonomous Parking System in modern vehicles. The integration of these components contributes to the seamless orchestration of motor control, environmental perception, and decision-making required for autonomous parking maneuvers.

Mechanism:

The advent of autonomous vehicles has ushered in a new era of innovation, transforming the way we perceive and interact with transportation. Among the myriad capabilities of these cutting-edge vehicles, the autonomous parking mechanism stands out as a testament to the seamless integration of hardware, sensors, and intelligent algorithms. In this essay, we explore the intricate mechanism that powers a car's autonomous parking, from its initial approach to the precise alignment within a parking slot.

1. Initial Approach:

The autonomous parking journey commences with the vehicle moving forward until it senses the presence of a nearby wall using ultrasonic sensors. This strategic approach sets the stage for the subsequent maneuvers, positioning the vehicle in proximity to a potential parking space.

2. Rotation for Parallel Alignment:

Upon reaching the wall, the car executes a meticulous rotation, aligning itself parallel to the detected structure. This rotation is not merely a functional adjustment but a calibrated movement designed for optimal parking space utilization and subsequent maneuvers.

3. Forward Movement and Sensor Detection:

The vehicle then embarks on a forward trajectory, guided by continuous input from ultrasonic sensors. These sensors serve as the vehicle's eyes, dynamically assessing the environment and detecting adequate parking space. This phase requires real-time analysis, ensuring that the car navigates with precision and caution.

4. Indication of Available Parking Slot:

Once a suitable parking slot is identified, the system communicates its readiness to the user through visual or auditory indicators. This seamless interaction between the vehicle and its user marks a pivotal moment in the autonomous parking process, ensuring user awareness and cooperation.

5. Rotation for Optimal Alignment:

Prior to entering the parking slot, the car executes another calculated rotation, aligning itself with the wall that was initially behind it. This deliberate movement ensures a direct and unobstructed path into the identified parking slot.

6. Controlled Entry into the Parking Slot:

The vehicle cautiously advances into the parking slot, its movements orchestrated with precision to avoid collisions and ensure a smooth entry. This controlled approach is a culmination of sensory data interpretation and algorithmic decision-making.

7. Fine-Tuning for Parallel Orientation:

Inside the parking slot, the car fine-tunes its position by executing further rotations until it is parallel to the wall that was initially in front of it. This meticulous adjustment ensures the optimal utilization of space and enhances the overall aesthetics of the parked vehicle.

Implementation:

MeasureDistanceLeft Function:

```
void measureDistanceLeft() {
  digitalWrite(trigPin1, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin1, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin1, LOW);

  duration1 = pulseIn(echoPin1, HIGH);
  distance1 = duration1 / 58.2;
  Serial.print("Left distance: ");
  Serial.println(distance1);

}
```

- ### Ultrasonic Sensor Setup:

  The code seems to assume that there are three pins, trigPin1 (for triggering the ultrasonic sensor), echoPin1 (for receiving the echo signal), and duration1 and distance_1 are likely variables used to store the measurement information.

- ### Triggering the Ultrasonic Sensor:

  digitalWrite(trigPin1, LOW);: Initially, the trigger pin is set to LOW.
  delayMicroseconds(2);: A small delay of 2 microseconds is introduced.
  digitalWrite(trigPin1, HIGH);: The trigger pin is set to HIGH to trigger the ultrasonic pulse.
  delayMicroseconds(10);: A 10-microsecond delay is introduced while the trigger pin is HIGH.
  digitalWrite(trigPin1, LOW);: The trigger pin is set back to LOW to stop the pulse.

- ### Measuring Echo Duration:

  Duration_1 = pulseIn(echoPin1, HIGH);: The pulseIn function is used to measure the duration of the echo pulse. It waits for the pin echoPin1 to go HIGH and then measures the duration the pin stays HIGH. This duration is stored in the variable duration1.

- ### Calculating Distance:

  distance_1 = duration_1 / 58.2;: The duration is converted into distance using a typical conversion factor for ultrasonic sensors. The division by 58.2 is a common conversion factor for converting microseconds to centimeters (for sensors operating at a 16 MHz clock frequency).

MeasureDistanceFront Function:

```
void measureDistancFront() {
  //Move the servo back to the front if needed
  if (currentServoPosition != frontAngle) {
    servo.write(frontAngle);
    delay(400); // Allow the servo to reach the positio
    currentServoPosition = frontAngle;
  }

  digitalWrite(trigPin2, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin2, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin2, LOW);

  duration2 = pulseIn(echoPin2, HIGH);
  distance2 = duration2 / 58.2;
  Serial.print("the front distance: ");
  Serial.println(distance2);

}
```

- **Servo Motor Adjustment:**
  The If Condition checks if the current position of a servo motor (currentServoPosition) is different from the specified front angle (frontAngle).
  If they are not equal, it moves the servo to the front angle position using servo.write(frontAngle).
  The delay(400) is introduced to allow time for the servo to physically reach the specified position.
  After moving the servo, it updates the currentServoPosition variable to the front angle.

- **Ultrasonic Sensor Triggering:**
  It triggers the ultrasonic sensor by sending a pulse: sets trigPin2 to LOW, introduces a small delay, sets it to HIGH for 10 microseconds, and then sets it back to LOW.

- **Measuring Echo Duration:**
  It uses pulseIn to measure the duration of the echo pulse on echoPin2.

MeasureDistanceLeft Servo:

```
void measureDistanceLeftServo() {
  // Move the servo back to the front if needed
  if (currentServoPosition != leftAngle) {
    servo.write(leftAngle);
    delay(400); // Allow the servo to reach the pos
    currentServoPosition = leftAngle;
  }

  digitalWrite(trigPin2, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin2, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin2, LOW);

  duration3 = pulseIn(echoPin2, HIGH);
  distance3 = duration3 / 58.2;
  Serial.print("left servo distance: ");
  Serial.println(distance3);
}
```

- **Servo Motor Adjustment:**
  The If Condition checks if the current position of a servo motor (currentServoPosition) is different from the specified left angle (leftAngle).
  If they are not equal, it moves the servo to the left angle position using servo.write(leftAngle).
  The delay(400) is introduced to allow time for the servo to physically reach the specified position.
  After moving the servo, it updates the currentServoPosition variable to the left angle

- **Ultrasonic Sensor Triggering:**
  It triggers the ultrasonic sensor by sending a pulse: sets trigPin2 to LOW, introduces a small delay, sets it to HIGH for 10 microseconds, and then sets it back to LOW.

- **Measuring Echo Duration:**
  It uses pulseIn to measure the duration of the echo pulse on echoPin2.

Rotate Parallel to Barrier:

This function measures distances on the left and left servo sides, compares the results, and takes actions accordingly. If the system is approximately parallel to a barrier (based on the distance measurements), it Sets rotated to true; otherwise, it initiates a right rotation.The absolute difference between the two distance measurements is also printed to the serial monitor for monitoring purposes.

```
void rotateParallelToBarrier(){
  measureDistanceLeft();
  //sum1 += distance1;
  delay(10);
  measureDistanceLeftServo();
  //sum2 += distance3;
  delay(10);
  if(abs(distance1 - distance2 ) <= 2){
    rotatted = true;
  }
  else{
    rotateRight();
  }
  Serial.print("distance diff: ");
  Serial.println(abs(distance1 - distance3 ));
}
```

MeasureDistanceLeft Function:

This function call is used to measure the distance on the left side using the `measureDistanceLeft()` function. This likely involves adjusting a servo to the front angle and triggering an ultrasonic sensor to measure the distance.

MeasureDistanceLeft Servo:

This function call is used to measure the distance on the left side using the `measureDistanceLeftServo()` function. This function likely adjusts a servo to the left angle and triggers an ultrasonic sensor to measure the distance.

Comparison and Condition:

This If condition compares the absolute difference between distance_1 and distance_2. If the absolute difference is less than or equal to 2, it sets the variable rotated to true. Otherwise, it calls the rotateRight() function.The purpose of this block is likely to check if the distances measured on the left and left servo sides are approximately equal, indicating that the system is parallel to a barrier. If they are not, it initiates a right rotation.

```
void rotateRight() {
  // Serial.println("Rotating right...");
  // startTime = millis();
  analogWrite(EnableA, speed1);
  analogWrite(EnableB, speed2);
  digitalWrite(In1, LOW);
  digitalWrite(In2, HIGH);
  digitalWrite(In3, HIGH);
  digitalWrite(In4, LOW);
  // printElapsedTime();
}
```

Find Available Slot :

This function measures distances on the left and left servo sides, checks if there is enough space (both distances are greater than or equal to 20), and takes actions accordingly. If there is sufficient space, it sets the slot variable to true. Otherwise, it initiates forward movement.

```
void findAvalibleSlot(){
  measureDistanceLeft();
  delay(10);
  measureDistanceLeftServo();
  delay(10);

  if(distance1 >= 20 && distance3 >= 20 ){
      slot = true;
  }
  else{
    moveForward();
  }
}
```

.

Conditions And Actions:

This if condition checks if both distance1 (from measureDistanceLeft()) and distance3 (from measureDistanceLeftServo()) are greater than or equal to 20.If this condition is true, it sets the variable slot to true. Otherwise, it calls the moveForward() function.The purpose of this block is likely to determine if there is an available parking slot. If both distances are greater than or equal to 20 (indicating enough space), it marks the slot as available. Otherwise, it initiates forward movement

```
void moveForward() {
  // Serial.println("Moving forward...");
  //startTime = millis();
  analogWrite(EnableA, speed1);
  analogWrite(EnableB, speed2);
  digitalWrite(In1, HIGH);
  digitalWrite(In2, LOW);
  digitalWrite(In3, HIGH);
  digitalWrite(In4, LOW);
  //printElapsedTime();
}
```

Adjust To Slot:

This function measures distances on the left and left servo sides, accumulates the distance values, checks if the system is aligned with the parking slot, and takes actions accordingly. If the system is sufficiently aligned, it sets the adjust variable to true. Otherwise, it initiates a left rotation. The absolute difference between the two distance measurements are also printed to the serial monitor for monitoring purposes.

```
void adjustToSlot(){
  measureDistanceLeft();
  sum1 += distance1;
  delay(10);
  measureDistanceLeftServo();
  sum2 += distance3;
  delay(10);

  if(abs(distance1 - distance3 ) <= 2){
    adjusted = true;
  }
  else{
    rotateLeft();
  }
  Serial.print("distance diff: ");
  Serial.println(abs(distance1 - distance3 ));
}
```

This section checks if the absolute difference between `distance1` (from `measureDistanceLeft()`) and `distance3` (from `measureDistanceLeftServo()`) is less than or equal to 2. If this condition is true, it sets the variable `adjusted` to true. Otherwise, it calls the `rotateLeft()` function.

```
void rotateLeft() {
  // Serial.println("Rotating left...");
  // startTime = millis();
  analogWrite(EnableA, speed1 );
  analogWrite(EnableB, speed2);
  digitalWrite(In1, HIGH);
  digitalWrite(In2, LOW);
  digitalWrite(In3, LOW);
  digitalWrite(In4, HIGH);
  // printElapsedTime();
}
```

Rotate left:

This code sets the direction of the motors. The exact configuration depends on the specific motor driver or H-bridge setup used in the project.
This configuration suggests that it's setting the left motor (EnableA) to rotate in one direction (probably forward), and the right motor (EnableB) to rotate in the opposite direction (probably backward).

```
void loop() {
  if(!reachedBarrier){
    if(!reachedBarrier){
      if(!phase4){
        ForwadrdTillReachingBarrier(12);
      }
      else{
        ForwadrdTillReachingBarrier(11);
      }
    }
    if(reachedBarrier){
      if(phase4){
        rotatted = false;
        phase2 = false;
      }
        phase1 = true;
        rotateRight();
        delay(1000);
        stopMotors();
        delay(250);
        sum1 = 0;
    }
  }
```

This code is a nested conditional block that seems to handle the behavior of a robot in a specific phase.

- The first inner condition (if (!reachedBarrier)) is redundant since it's already checked in the outermost condition.
- If the robot has not reached a barrier, it checks if it is not in phase 4 If not in phase 4, it is called Forward_TillReachingBarrier(12).
- In phase 4, it is called Forward_TillReachingBarrier(11). After that, it checks if the robot has reached a barrier.
- If so, it checks if it is in phase 4. If yes, it sets rotated and phase2 to false. Then, it sets phase1 to true, rotates right, introduces a delay, stops motors, introduces another delay, and resets sum1 to 0 .

This code segment controls the behavior of the robot in `phase1` when it has not rotated (`rotated` is false). It initiates a rotation and, based on the outcome, sets the LED state, transitions to the next phase (`phase2`), and stops the motors. The LED state is set based on whether the robot is in `phase4` or not.

```
if(!rotatted && phase1 ){
  if(!rotatted){
    rotateParallelToBarrier();
  }
  if(rotatted){
    if(phase4){
    digitalWrite(ledpin, HIGH);
    }
    else{
    digitalWrite(ledpin, LOW);
    }
    phase2 = true;
    stopMotors();
  }
}
```

The first inner condition (if (!rotated)) checks if rotated is false. If true, it calls the rotateParallelToBarrier() function.
The second inner condition (if (rotated)) checks if rotated is true.
If true, it further checks if phase4 is true. If yes, it sets the LED pin (led pin) high; otherwise, it sets it low.
It then sets phase2 to true, indicating a transition to the next phase , and stops the motors.

This code segment controls the behavior of the robot in phase2 when there is no available slot (slot is false). It initiates a process to find an available slot, and based on the outcome, it performs a sequence of movements and transitions to the next phase (phase3).

- The first inner condition (if (!slot)) checks if the slot is false. If true, it calls the findAvalibleSlot() function.
- The second inner condition (if (slot)) checks if slot is true. If true, it executes a series of movements:
- Move forward for 100 milliseconds.
  Rotate left for 1400 milliseconds.
- Move forward again for 70 milliseconds.
  Stop the motors.

After executing these movements, it sets phase3 to true, indicating a transition to the next phase

```
if(!slot && phase2){
  if(!slot){
    findAvalibleSlot();
  }
  if(slot){
    moveForward();
    delay(100);
    rotateLeft();
    delay(1400);
    moveForward();
    delay(70);
    stopMotors();
    phase3 = true;
  }
}
```

This code segment controls the behavior of the robot in `phase3` when it has not been adjusted to the slot (`adjusted` is false). It initiates the adjustment process, and if adjustment is successful, it updates phase variables, resets some flags, and stops the motors. This sets the stage for the robot to enter `phase4` and possibly start a new cycle of its operation.

- The first inner condition (if (!adjusted)) checks if adjusted is false.
- If true, it calls the adjustToSlot() function.
- The second inner condition (if (adjusted)) checks if adjusted is true. If true, it performs the following actions:
- Sets phase4 to true.
- Sets phase1 to false.
- Sets reachedBarrier to false.
- Stop the motors.

```
if(!adjusted && phase3){
  if(!adjusted){
    adjustToSlot();
  }
  if(adjusted){
    phase4 = true;
    phase1 = false;
    reachedBarrier = false;
    stopMotors();
  }
}
```

Application:

This system architecture leverages the combination of HTML, ESP in AP mode, and socket programming to enable remote control of a robot. It showcases the potential for web-based interfaces to simplify user interaction and control in robotics applications.

- **ESP in AP Mode:**
  The ESP is configured to act as an Access Point (AP), creating its own Wi-Fi network.

  Devices can connect to this network, and the ESP serves as the access point for communication.

- **Web Server on ESP:**
  The ESP device is set up to act as a web server.

  When users connect to the ESP's Wi-Fi network, they can access the hosted web page from their devices.

- **HTML Interface:**
  The HTML page served by the ESP contains buttons for four directions:
  reverse, forward, left, and right.

  These buttons serve as user input elements that users can interact with on their devices.

- **Socket Communication:**
  The HTML page uses JavaScript to establish a socket connection with the ESP.

  This socket communication facilitates real-time bidirectional data exchange between the HTML page and the ESP.

- **Button Actions:**
  Each button on the HTML page is associated with a specific action or command.

  When a user clicks a button, JavaScript sends the corresponding command through the socket to the ESP.

- **ESP Command Handling:**

  The ESP, upon receiving commands through the socket, interprets these commands and triggers appropriate actions.

  For example, if the "forward" button is pressed, the ESP may activate motors to make the robot move forward.

- **Robot Movement:**
  The commands received by the ESP trigger movements or actions in the robot.

  The actual implementation of motor control, robot movement, or any other actions depends on the specific code running on the ESP.

- **Real-time Interaction:**
  The socket communication ensures real-time interaction between the HTML interface and the robot.

  Users experience minimal delay, providing a responsive and interactive control interface.

- **Potential Enhancements:**
  Depending on the complexity of the robot and desired features, additional functionalities can be incorporated into the HTML interface, such as feedback on the robot's status, sensor data display, or more advanced control options.

This code is a part of the WebSocket server's event handling logic, allowing the server to respond to various WebSocket events such as client connection, disconnection, data reception, Pong frames, and errors. The processCarMovement function seems to be involved in handling the movement of a car, possibly based on the received WebSocket data. The actual implementation of processCarMovement and related functions is not provided in this snippet.

```
switch (type)
{
  case WS_EVT_CONNECT:
    Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
    //client->text(getRelayPinsStatusJson(ALL_RELAY_PINS_INDEX));
    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:
    break;
}
```

- **WS_EVT_CONNECT:**
  This case is executed when a WebSocket client connects to the server.

  It prints information about the connected client, including the client's ID and IP address.

- **WS_EVT_DISCONNECT:**
  This case is executed when a WebSocket client disconnects from the server.

  It prints information about the disconnected client, including the client's ID.

  It then calls processCarMovement("0"), possibly to handle a specific action related to the disconnection.

- **WS_EVT_DATA:**
  This case is executed when the server receives WebSocket data from a client.

  It checks various conditions (e.g., final frame, frame index, frame length, and opcode) to ensure it's a valid WebSocket text message.

  If the conditions are met, it converts the received data into a C++ string (myData) and calls processCarMovement with the received data.

- **WS_EVT_PONG and WS_EVT_ERROR:**
  These cases are provided for handling WebSocket Pong frames and errors, respectively.

  In this code, no specific actions are taken for Pong frames or errors.

- **Default Case:**

  The default case is included but does nothing. It serves as a placeholder for any unhandled WebSocket event types.

It looks like the provided code snippet is an HTML markup for a web page with a user interface designed for Wi-Fi control, presumably for a device or system related to "Hash Include Electronics.

```
<body class="noselect" align="center" style="background-color:white">

<h1 style="color: teal;text-align:center;">Hash Include Electronics</h1>
<h2 style="color: teal;text-align:center;">Wi-Fi &#128663; Control</h2>

<table id="mainTable" style="width:400px;margin:auto;table-layout:fixed" CELLSPACING=10>
  <tr>
    <td ontouchstart='onTouchStartAndEnd("5")' ontouchend='onTouchStartAndEnd("0")'><span class="arrows" >&#11017;</span></td>
    <td ontouchstart='onTouchStartAndEnd("1")' ontouchend='onTouchStartAndEnd("0")'><span class="arrows" >&#8679;</span></td>
    <td ontouchstart='onTouchStartAndEnd("6")' ontouchend='onTouchStartAndEnd("0")'><span class="arrows" >&#11016;</span></td>
  </tr>

  <tr>
    <td ontouchstart='onTouchStartAndEnd("3")' ontouchend='onTouchStartAndEnd("0")'><span class="arrows" >&#8678;</span></td>
    <td></td>
    <td ontouchstart='onTouchStartAndEnd("4")' ontouchend='onTouchStartAndEnd("0")'><span class="arrows" >&#8680;</span></td>
  </tr>

  <tr>
    <td ontouchstart='onTouchStartAndEnd("7")' ontouchend='onTouchStartAndEnd("0")'><span class="arrows" >&#11019;</span></td>
    <td ontouchstart='onTouchStartAndEnd("2")' ontouchend='onTouchStartAndEnd("0")'><span class="arrows" >&#8681;</span></td>
    <td ontouchstart='onTouchStartAndEnd("8")' ontouchend='onTouchStartAndEnd("0")'><span class="arrows" >&#11018;</span></td>
  </tr>

  <tr>
    <td ontouchstart='onTouchStartAndEnd("9")' ontouchend='onTouchStartAndEnd("0")'><span class="circularArrows" >&#8634;</span></td>
    <td style="background-color:white;box-shadow:none"></td>
    <td ontouchstart='onTouchStartAndEnd("10")' ontouchend='onTouchStartAndEnd("0")'><span class="circularArrows" >&#8635;</span></td>
  </tr>
</table>
```

HandleRoot:

This function is likely designed to handle requests to the root ("/") endpoint of your web server. When a client sends a request to the root endpoint, this function sends a response with a status code of 200 (OK), content type "text/html," and the content of the htmlHomePage variable.

HandleNotFound:

This function is a handler for requests that are not found (404 Not Found).
If a client sends a request for a resource that is not available on the server, this function responds with a 404 status code and a plain text message saying "File Not Found."

```
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)
{
```

OnWebSocketEvent:

This function is likely a callback that gets executed when there is an event on the WebSocket.

It receives parameters related to the WebSocket event, including the server instance, the client instance, the type of event (type), additional arguments (arg), a pointer to the received data (data), and the length of the data (len).

The actual logic inside this function would depend on the specific WebSocket events you want to handle (e.g., connection, disconnection, data reception, etc.).