



**University of  
Nottingham**

UK | CHINA | MALAYSIA

Title: Wireless Vehicle Control, Sensory Feedback and  
Command-Driven Maze Navigation

**Applied Electrical and Electronic  
Engineering: Construction Project (EEEE1002  
UNUK) (FYR1 24- 25): Technical Report 2**

Date of Submission: 29/04/25

## Abstract

In certain unique circumstances the full autonomy of a vehicle is either unnecessary or not the desired control method for the situation. An industry example of this are teleoperated robots. Teleoperated robots are robotic systems that are remotely controlled by a human operator. An example of such robotic systems would be robotic arms which are widely used in manufacturing plants for welding, assembly and material handling. Granting full autonomy to such systems for complex tasks such as welding or assembly may increase the risk of faulty products as robots are designed for repetitive tasks. Therefore, having a human operator remotely control these robotic arms would allow for the robotic systems to have the precision of a human being and maintain the benefits of using robotic systems over human labour. The showcase this in action two tests will be done; the first one will include the wireless control of a vehicle that also sends sensory feedback to the wireless controller depending on the situation, the second test will be a command driven maze in which the vehicle will traverse a maze whilst being controlled remotely. The control device for the first test will be a joystick driven remote control, as for the second test the vehicle will be given commands via a number pad and each value of the number pad corresponds to a specific action the vehicle will perform.

## Table of Contents

Title: Wireless Vehicle Control, Sensory Feedback and Command-Driven Maze Navigation .....	1
Abstract .....	2
Table of Contents .....	2
Introduction .....	3
1. Design & Implementation of the Wireless Remote.....	3
1.1 Wireless Remote Construction.....	3
1.2 Wireless Sensory Feedback .....	12
2. Command Driven Maze Navigation.....	16
2.1 Principle of Operation .....	16
2.2 Hardware Solutions .....	17
2.3 Software Solutions .....	19
Conclusion .....	23
References.....	23
Appendix.....	24

## Introduction

The basis of these challenges is the wireless communication between two ESP32 micro-controllers. For the wireless control of the vehicle as mentioned prior a joystick driven remote control will be used to determine the direction in which the vehicle will move. In addition to that a LED and buzzer will be used for sensory feedback if a specific event has occurred. As for the sensory feedback an ultrasound sensor will be used to calculate distance and provide feedback if the vehicle has reached a certain distance from an object. As for the maze navigation challenge, the control method will be a keypad which dictates the direction the bot goes and extra buttons can be configured for other functions such as stopping the motors or turning on the sensory feedback systems such as a LED or a buzzer.

## 1. Design & Implementation of the Wireless Remote

This section delves into the design and construction of the wireless remote used to operate the vehicle. As explained in the introduction, the main control method of this remote will be a joystick that can move 360 degrees. In addition, the sensory feedback aspect will also be detailed in this section alongside all hardware and software solutions required to make this work.

### 1.1 Wireless Remote Construction

#### *Principle of Operation*

The main control method of the remote is a simple two axis joystick akin to those used in old PlayStation controllers. The way this joystick works is via the use of potentiometers and two Gimbals. A gimbal is a device that allows an object to incline freely in any direction or suspends it so that it remains level when the support is tipped. The joystick is connected to a thin rod that is also connected to one end of each gimbal which in this case is a rotatable slotted shaft one for the x direction and another for the y direction. Each direction has a potentiometer connected to it, as such by rotating the joystick in either direction it causes the wiper of the potentiometer to also rotate thereby causing the resistive track of the potentiometer to move thus varying the resistance of the potentiometer[1]. As a result, moving the joystick will cause an analogue voltage output that can be measured and translated into a digital output that's sent to the main vehicle to determine its direction of movement.

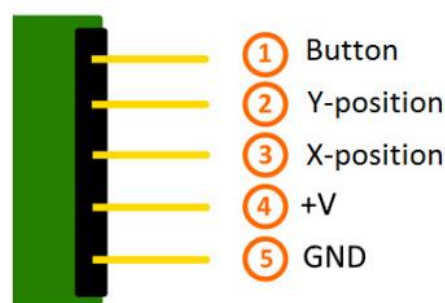


Figure 1: Joystick Pinout

The decision to use a joystick with a 5-pin interface like this one is to allow for easier integration with the Arduino IDE software environment as developing software for the function of this joystick would be made easier. As stated, prior, the x and y position of the joystick are analogue outputs thus should be connected to pins of the microprocessor (which in this case is the ESP32) that allow for analogue outputs. As for the button on the joystick, a pullup resistor must be used. A pullup resistor ensures that a digital output has a well-defined logic level under all conditions. For instance, when the button isn't pressed the logic level should be set to LOW and the reverse should be true when the button is pressed. The lack of a pull-up resistor may lead to the microcontroller unexpectedly interpret the input values as either a logic high or logic low[2].

#### 1.1.1 Joystick and ESP32 Integration

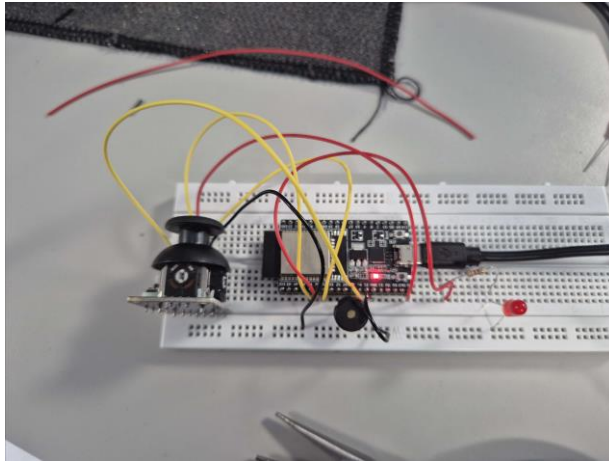


Figure 2: Joystick Breadboard Layout

Figure 2 shows the joystick pinout alongside the ESP32. The use of the LED and buzzer will be useful for sensory feedback however in this case the LED and buzzer will show feedback whenever the joystick moves in any direction to make sure the joystick works.



Figure 3 displays the basic pinout diagram for the joystick to ESP32 integration. As the joystick is a 5V device there's no need for any voltage correction measures. However, for the LED a pull-up resistor is needed to decrease the voltage to meet the specifications needed for the LED to light up.

```

// Define pin connections

#define JOYSTICK_X_PIN 32

#define JOYSTICK_Y_PIN 34

#define JOYSTICK_BTN_PIN 27

#define BUZZER_PIN 26

#define LED_PIN 35 // Optional LED

// Threshold for joystick movement detection

#define JOYSTICK_THRESHOLD 2000

void setup() {

    pinMode(JOYSTICK_BTN_PIN, INPUT_PULLUP);

    pinMode(BUZZER_PIN, OUTPUT);

    pinMode(LED_PIN, OUTPUT);

    Serial.begin(115200);

}

void loop() {

    int xValue = analogRead(JOYSTICK_X_PIN);

    int yValue = analogRead(JOYSTICK_Y_PIN);

    int buttonState = digitalRead(JOYSTICK_BTN_PIN);

    Serial.print("X: "); Serial.print(xValue);

    Serial.print(" | Y: "); Serial.print(yValue);

    Serial.print(" | Button: "); Serial.println(buttonState);

    // Check if the joystick is moved beyond threshold

    if (abs(xValue - 2048) > JOYSTICK_THRESHOLD || abs(yValue - 2048)
> JOYSTICK_THRESHOLD) {

        tone(BUZZER_PIN, 1000); // Play sound at 1kHz

        digitalWrite(LED_PIN, HIGH); // Turn on LED

    } else {

        noTone(BUZZER_PIN); // Stop buzzer

        digitalWrite(LED_PIN, LOW); // Turn off LED

    }

    delay(100);

}

```

To test whether the joystick was working this small piece of code was created in order to add visual and auditory feedback whenever the joystick moved. The function of this code is to read the analogue output of the joysticks x and y direction and compare it to the joystick threshold which in this case is 2000. The reason why the absolute value of the x and y directions are subtracted by 2048 is because 2048 is the default value of the joystick when it is stationary. In addition, any small movements in the joystick isn't taken into consideration as it acts as a filter for any accidental movements of the joystick.

Figure 4: Joystick-ESP32 Software Test

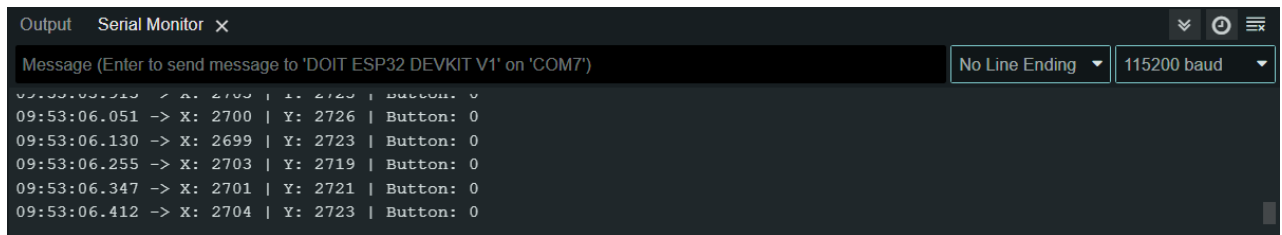


Figure 5: Idle Joystick Values

For instance, in this case the LED and buzzer will not turn on as the joystick is technically idle despite the values in the x and y direction being greater than 2048. The joystick may be moving slightly but it's not moving enough to warrant considering it as an input.

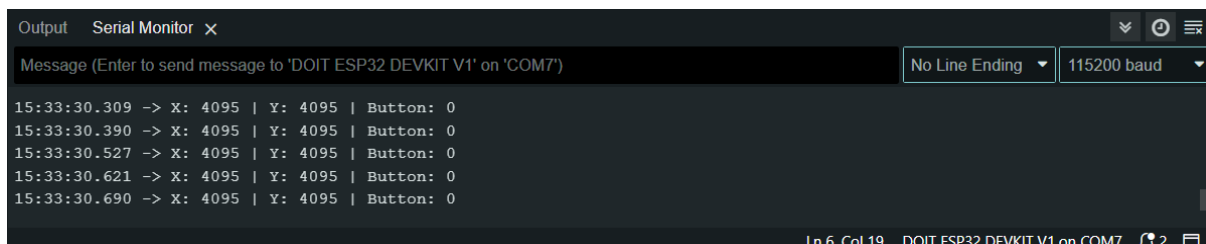


Figure 6: Moving Joystick Values

However, the values shown in figure 6 clearly displays the joystick as being moved deliberately as such in the context of wireless control if the joystick is in this state then this will be considered a conscious input and be used to dictate the vehicle's direction.

### 1.1.2 ESP-NOW Communication

The wireless connectivity between the remote and the main vehicle relies upon the ESPNOW communication protocol. ESPNOW is a peer-to-peer direct communication method between two ESP family devices. ESPNOW can either be configured with one way or two-way communication depending on the situation[3]. A peer to peer connection is one in which both devices acts as a server and the client thus removing the need for a centralised server. As a result, (in the context of the ESPNOW communication method) causes faster data delivery speeds due to the lack of an overhead that would've been caused by a centralised server[4] ESPNOW specifically also transmit low data quantities due to utilisation of action frames for their communication. Action frames are specialised data frames that contain data for a specific action. For example, there could be a data frame for MAC address of the client, or the main body of data being transmitted. Each action frame has a limited quantity of bytes available to be used which allow for ESPNOW devices to efficiently exchange data[3]. In order to use ESPNOW between devices the mac address of the client must be extracted. For two-way communication the mac address of both devices involved is needed as in order to send data back and forth both devices will act as a client and server.

```

#include <WiFi.h>

#include <esp_wifi.h>

void readMacAddress(){

  uint8_t baseMac[6];

  esp_err_t ret = esp_wifi_get_mac(WIFI_IF_STA, baseMac);

  if (ret == ESP_OK) {

    Serial.printf("%02x:%02x:%02x:%02x:%02x:%02x\n",

      baseMac[0], baseMac[1], baseMac[2],

      baseMac[3], baseMac[4], baseMac[5]);

  } else {

    Serial.println("Failed to read MAC address");

  }

}

void setup(){

  Serial.begin(115200);

  WiFi.mode(WIFI_STA);

  WiFi.STA.begin();

  Serial.print("[DEFAULT] ESP32 Board MAC Address: ");

  readMacAddress();

}

void loop(){

}

```

Figure 7:ESPNOV Mac address Extraction

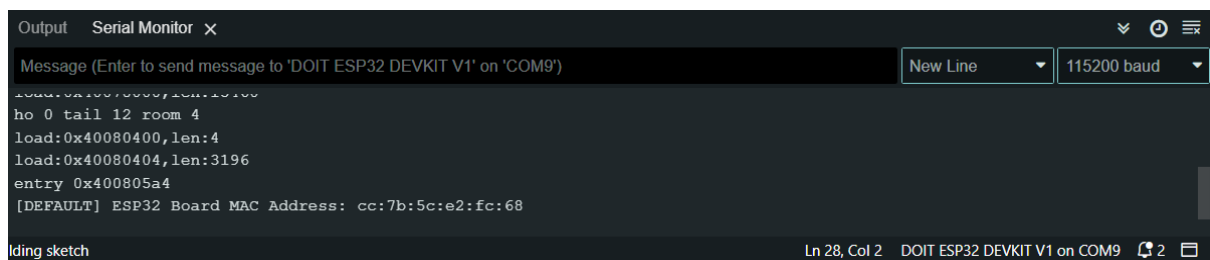


Figure 7:Successful Mac Address Print



As the mac address of the client has been printed it's added to the master code to initialise the ESPNOW wireless connection and start to transmit the joystick data to the main vehicle.

### 1.1.3 Hardware and Software Designs

#### 1.1.3.1 Hardware Designs

This stripboard design is unique in the sense that there was a space limitation due to the fact that both control methods detailed in the introduction had to fit on one stripboard. As such, one half of the stripboard had to be used to fit all 3 components needed for joystick control. Such a constraint needed a design in which there was enough space for every component, the components were not spaced in a way that could cause short circuits and that there was space left for the second control method.

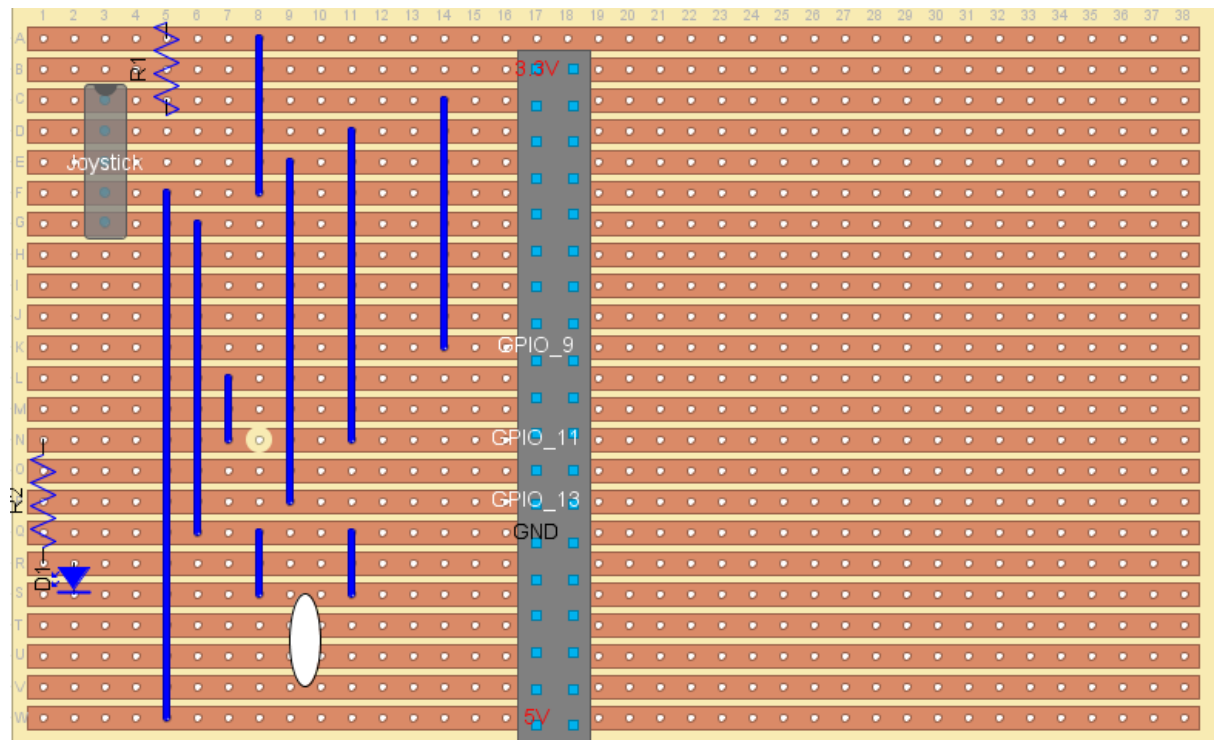


Figure 8:Stripboard Design

This stripboard design separates all 4 major components whilst leaving space for the keypad to be added. As a result, this causes no issues in regard to both future proofing the design for future additions and also any short circuits which could arise. The resistor value used for the pull up resistor is 4.7k $\Omega$ . The reason for this resistor value is due to the input impedance of the ESP32 pins. The value of a pull up resistor should be a lot smaller than the input impedance of device receiving the signal. Since the ESP32 GPIO pins have a very high impedance a value of 4.7k $\Omega$  should be small enough to ensure a strong pull-up effect.

#### 1.1.3.2 Software Designs

The software design uses a master-slave dynamic using the ESPNOW communication protocol detailed in 1.1.2 . The master will be the wireless remote and act as the server and the slave will be the main vehicle thus receiving information as a client.

### 1.1.3.2.1 Master Code

The premise behind the master code is to first determine the direction the joystick is moving and then assign a value to variable 'D' to then send to the slave

ESP32. To determine the direction the joystick is moving the software uses a similar premise found in 1.1.1 in which the basic software was to determine whether the joystick has moved at all. In this case the default value of 2048 is added or subtracted from the joystick threshold of 2000 to determine whether the joystick has moved in any of the four main directions. For instance, if the y value is greater than the default value 2048 added to the joystick threshold being 2000 then the software perceives the joystick to be moving north, and the reverse is true if the y value is smaller than 2048-2000. Since the true direction of the joystick can't be accurately determined with software, an accurate enough estimate is needed in order to send directional data to the slave ESP32.

```
if (yValue > 2048 + JOYSTICK_THRESHOLD) {
    D = 1;
} else if (yValue < 2048 - JOYSTICK_THRESHOLD) {
    D = 2;
} else if (xValue > 2048 + JOYSTICK_THRESHOLD) {
    D = 3;
} else if (xValue < 2048 - JOYSTICK_THRESHOLD) {
    D = 4;
} else if (buttonState == 0) {
    D = 5;
    tone(BUZZER_PIN, 1000);
    digitalWrite(LED_PIN, HIGH);
}
```

Figure 9: Master-Code Snippet 1

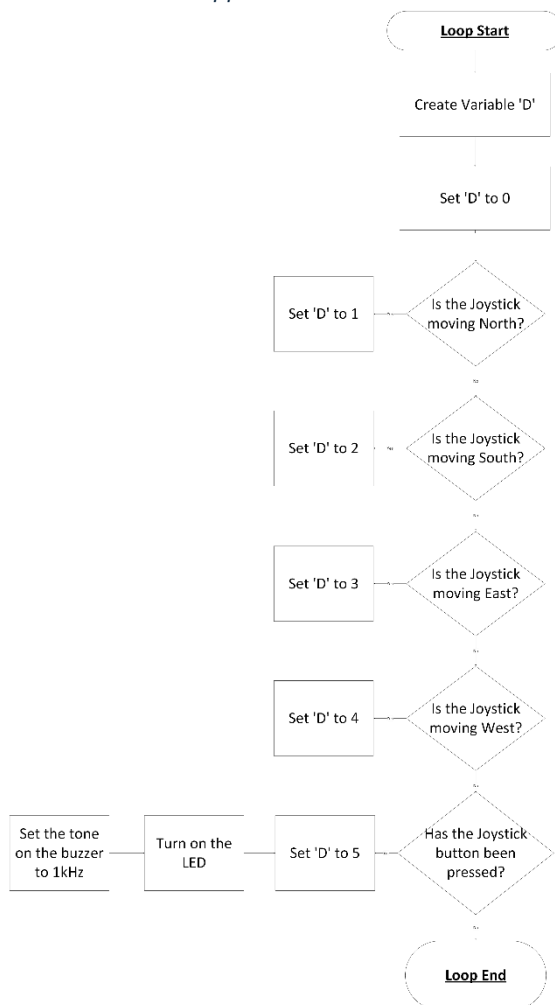


Figure 10: Master-Code Flowchart

Figure 10s flowchart displays a simplified design for the master software. The value of the variable 'D' is determined by the actions of the joystick may it be its direction or the logic state of the button. This variable 'D' is then sent to the slave ESP32 via the ESPNOW communication method detailed in 1.1.2 . The button on the joystick can be configured for any use and setting the variable 'D' to respond to the button press would allow the slave ESP32 to decide on what to do once the button is pressed.

### 1.1.3.2.2 Slave Code

The design of the slave code is to receive the data from the master ESP32 which in this case is the value of the variable 'D' and assign said variable to a direction in which the vehicle will move in.

As the variable 'd' changes the command of the vehicle also changes thus corresponding to the estimated movement of the joystick. Functions such as Centralise() are used to prevent situations in which previous commands can impact newer commands. An example of this is with regards to moving left and right. If the vehicle moves left first without centralising the servos, then when asked to move right the servos will rotate from the previous angle treating it as the new origin point. As a result, the vehicle may not steer right and even if it does the movement may not be accurate

```
switch (d) {
  case 1:
    Serial.println(d);
    Centralise();
    //delay(20);
    goForwards();
    break;
  case 2:
    Serial.println(d);
    Centralise();
    //delay(20);
    goBackwards();
    break;
  case 3:
    Serial.println(d);
    Centralise();
    //delay(20);
    moveSteering();
    //delay(20);
    goClockwise();
    break;
  case 4:
    Serial.println(d);
    Centralise();
    //delay(20);
    moveSteeringL();
    //delay(20);
    goAntiClockwise();
    break;
  case 5:
    Serial.println(d);
    stopMotors();
    break;
}
```

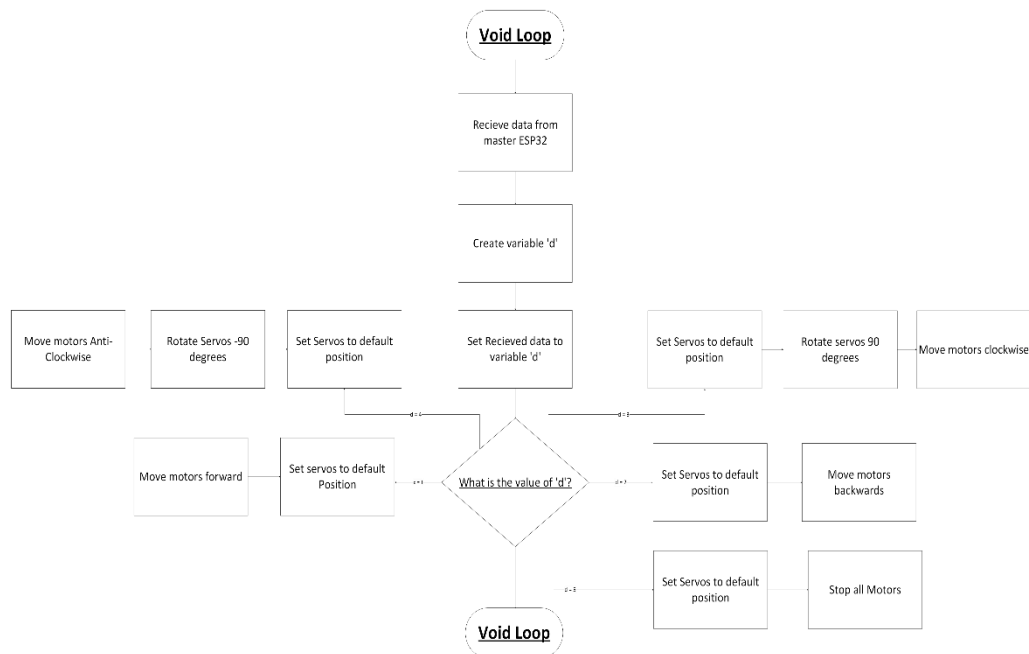


Figure 12:Slave-Code FC

Similarly to the master code, figure 12 displays a simple graphical explanation of the slave code software. As previously stated, the button on the joystick can be assigned to any action. In this specific case when the logic state of the button is set to HIGH (when the button is pressed) all the motors stop thus functioning as a emergency stop button. A function like this could be useful to immediately stop any unnecessary actions that may cause damage to the vehicles.

Figure 11:Slave-Code Snippet 1

## 1.2 Wireless Sensory Feedback

This additional section revolves around the addition of sensory feedback in which a specific event leads to a reaction from the remote itself. As such, this provides the user information with regards to the environment of the vehicle. The event in question is whether or not an object is within 10cm from the vehicle. The LED and buzzer seen in the designs in 1.1.3.1 will be used as feedback. When the vehicle is 10cm away from an object the LED will light up and the buzzer will produce a sound.

### 1.2.1 Principle of Operation

In order for the vehicle to know whether an object is within 10cm of it a sensor is required in order to calculate distances. A sensor that is perfect for this use case is an ultrasound sensor. An ultrasound sensor works by emitting an ultrasound wave and calculating the time it takes for the emitted sound wave to get bounced back and detected by the ultrasound sensor[5]. The distance between the sensor and an object can be calculated using the rearrangement of the speed equation.

$$Speed = \frac{Distance}{Time}$$

By rearranging the speed equation to make distance the subject, distance can be calculated by the equation

$$Distance = Speed \times Time$$

Since the speed of sound in air is constant at 343 m/s, the only value needed to calculate distance is the time it takes between emission and detection. The reasoning behind the use of an ultrasound sensor for object detection rather than a basic camera is due to differences in response time and the accuracy. An ultrasound sensor has a much higher response time than a camera due to the lack of image processing required. For a camera to be used in object detection it first has to process any images before sending sensory feedback whereas an ultrasound sensor just needs to measure time and use basic calculations to figure out distance. The sensor that will be used for this purpose is the HR-SR04. The HR-SR04 has a simple four wire interface which can be easily connected to a microcontroller such as an ESP32. As such, this makes the HR-SR04 very compatible with the Arduino IDE software environment allowing for enhanced control over the sensor.

### 1.2.2 Hardware Solutions

The bulk of the hardware solutions required for the sensory feedback to work is to connect the HR-SR04 to the bottom ESP32. In previous use cases the vehicle was split between the main ESP32 which controlled the motor driver circuit and servos and the upstairs section which controlled the different sensors. Since the upstairs ESP32 is being used for the wireless remote, the HR-SR04 must be connected to the bottom ESP32 in order to send sensory feedback. The first aspect to consider is the voltage correction required in order for the HR-SR04 to function properly. Since the

ESP32 is a 3.3V device and the HR-SR04 is a 5V device the voltage needs to be stepped down in order to prevent any harm to the ESP32.

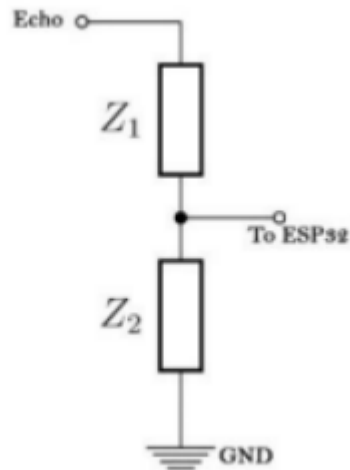


Figure 13: Voltage Divider Circuit

This basic voltage divider circuit can be used to split the voltage between the two resistors thus reducing the voltage going to the ESP32. The equation needed to calculate the value of resistors required to bring down the voltage from 5V to 3.3V is:

$$\frac{V_{out}}{V_{in}} = \frac{R_1}{R_1 + R_2}$$

Since  $V_{out}$  is equal to 3.3V and  $V_{in}$  is 5V the equation can be rearranged to be:

$$R_1 = \frac{33}{17} R_2$$

As  $33/17$  can be approximated to be equal to 2, we can say that  $R_1$  is approximately double the value of  $R_2$  therefore if  $R_2$  is  $1k\Omega$  then  $R_1$  would be  $2k\Omega$ .

The second aspect of the hardware solution is to connect the HR-SR04 to the bottom ESP32. The way to do this is to use the molex connections in order to connect the sensors echo and trigger pins to the TX and RX pins on the ESP32. The TX and RX pins are transmit and receiver pins which either receive data or transmit data to and from another device. Connect the trigger pin of the HR-SR04 to the TX pin on the ESP32 and the echo pin to the RX pin on the ESP32.

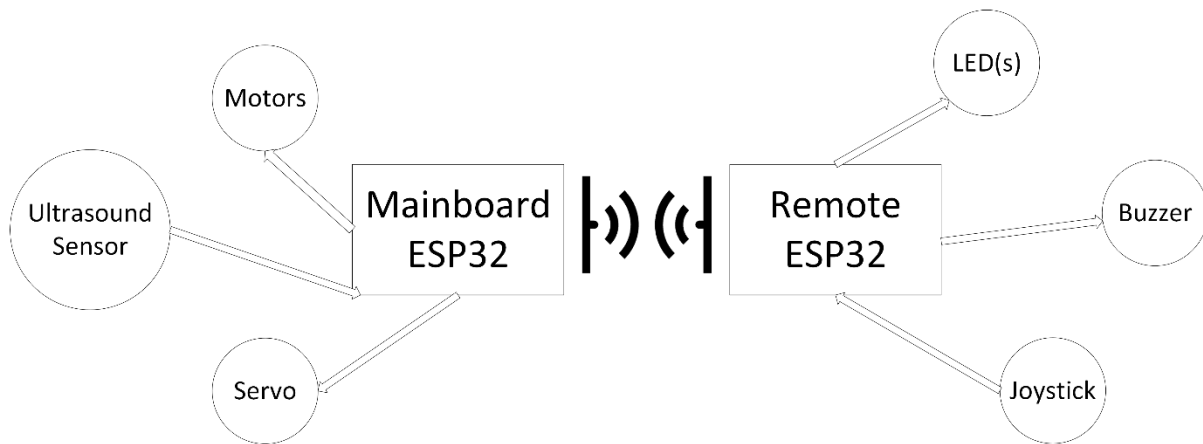


Figure 14: Sensory Feedback System Overview

Figure 14 shows the system overview for the sensory feedback. As stated earlier, the output of the ultrasound sensor dictates whether or not the LED or buzzer turns on. However, the way the main ESP32 delivers the sensory information is via the same ESPNOW communication method explained in 1.1.2. As a result, both the main ESP32 and the remote ESP32 will have to receive and transmit data to one another both acting as a client and server simultaneously.

### 1.2.3 Software Solutions

The primary difference between the software being used in this section and the one used in 1.1.3.2 is that both the master and slave code will be receiving and transmitting data. As a result, the mac address of the remote ESP32 will also be needed in order to make this work. To edit the existing master code to allow for remote ESP32 to receive data a singular function could be added in order to make the code more efficient.

```

void OnDataRecv(const esp_now_recv_info_t
*recv_info, const uint8_t *incomingData, int len) {

    memcpy(&incomingValues, incomingData,
sizeof(incomingValues));

    Serial.print("Bytes received: ");

    d = incomingValues.data1; // Correctly update d

    Serial.println(d);

}
  
```

Figure 15: Data Receive Function

The function `OnDataRecv()` copies the raw byte data from 'incomingdata' into a defined structured variable 'incomingValues'. The variable 'incomingValues' comes from a struct of integers made to hold information to either place received information into or to hold data meant to be transmitted. The function then updates the variable `d` to one of the incoming values specifically the first integer within the structure. Since the master code is only receiving one piece of data the other data points within the structure remain untouched. This variable is then printed out as a way to debug the code as if nothing is printed out or the printed value is an unexpected one then the debugging process can start.

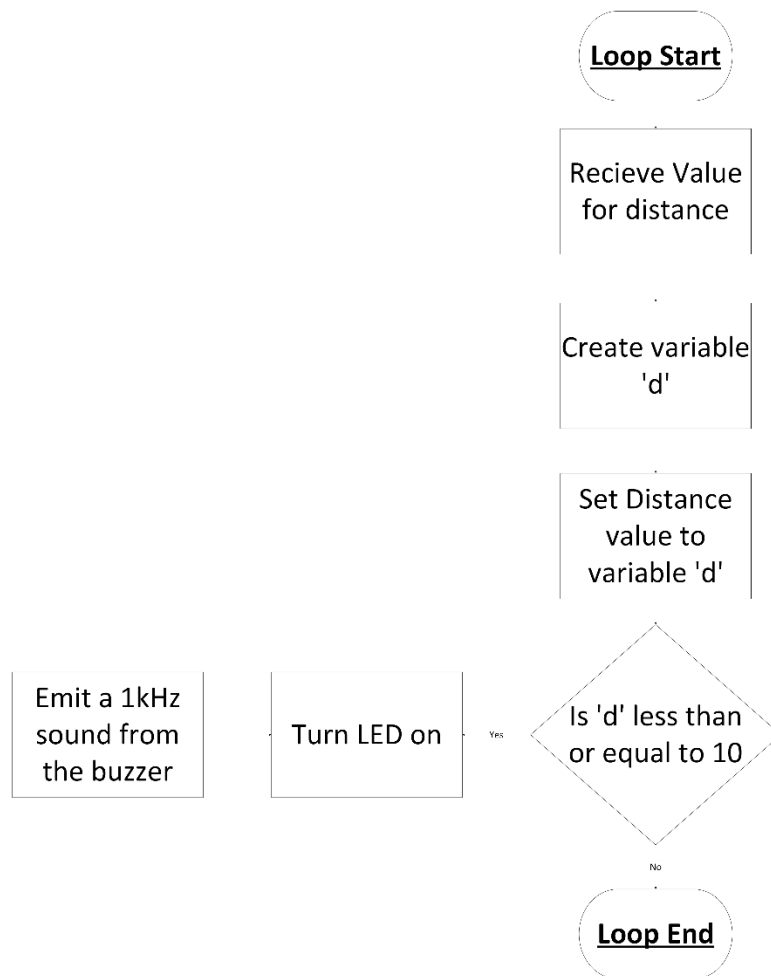


Figure 16:SF Master Code

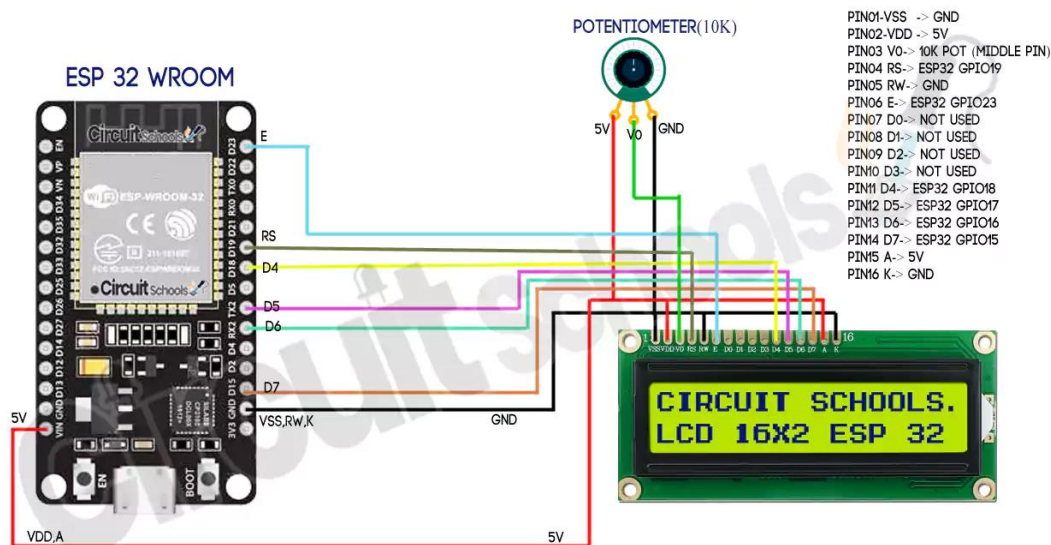
Figure 16s flow chart shows the added section to the master code which dictates how the sensory feedback is done. If the distance between the object and the vehicle is less than or equal to 10cm then the LED and buzzer should turn on. As for the slave code, the only additions are concerned with transmitting the output from the ultrasound sensor via ESPNOW to the remote ESP32 thus creating a constant feedback loop in which the user always knows whether the vehicle is too close to an object. This works perfectly in conjunction with the emergency stop button described in 1.1.3.2.2 as if the user can visibly see on the remote whether the vehicle is too close to an object and choose to stop it remotely. Even if the user may not see the LED turn on maybe due to the brightness of the LED or the brightness of the environment, the sound of the buzzer would also let the user know the distances between the vehicle and the object thus preventing most collisions that could cause damage to the vehicle.

## 2. Command Driven Maze Navigation

This section delves into the second control method mentioned in the introduction. The construction of a number pad driven control method in order to control the vehicle throughout a maze. The rest of this section details the hardware and software solutions in order to get the vehicle through this maze

### 2.1 Principle of Operation

The secondary control method of this remote is the use of a configurable number pad and an LCD screen to act as visual feedback. The number pad is a 4x3 matrix containing numbers from 0 – 9 and also includes two special characters which are '\*' and '#'. Each button on the number pad will be configured to control vehicle in numerous ways. The numerical buttons will be used to dictate the direction the vehicle and its angle of rotation. For instance, pressing 1 may make the vehicle rotate  $45^\circ$  while pressing 2 will make the vehicle rotate  $90^\circ$ . However, the different buttons may dictate the distance the vehicle moves. For example, pressing 3 may make the vehicle move 10cm while pressing 4 may make the vehicle move 30cm. The number pad is used in conjunction with an LCD screen as a form of sensory feedback. The purpose of the LCD screen is to not only display the character pressed by the number pad but to also display the corresponding command. A Liquid Crystal Display (LCD) consists of a layer of liquid crystals sandwiches between two transparent electrodes. When an electric current is applied, the crystals align in a way to control the amount of light that goes through thus creating the images shown on the screen[6].



Circuit Diagram a: Circuit Schools Pinout

Figure 17: LCD Pinout to ESP32

The decision to use this LCD screen over other sources of visual feedback is due to the simplicity of understanding. Different visual feedback that's suitable for this context could be an array of LEDs that correspond to whatever button on the number pad is pressed. However, such a solution despite being easier to setup may be interpreted differently by distinct users especially if it wasn't done properly. An LCD screen output is very simple and can be configured to be understood by myriads of users.



## 2.2 Hardware Solutions

### 2.2.1 Testing and Simulations

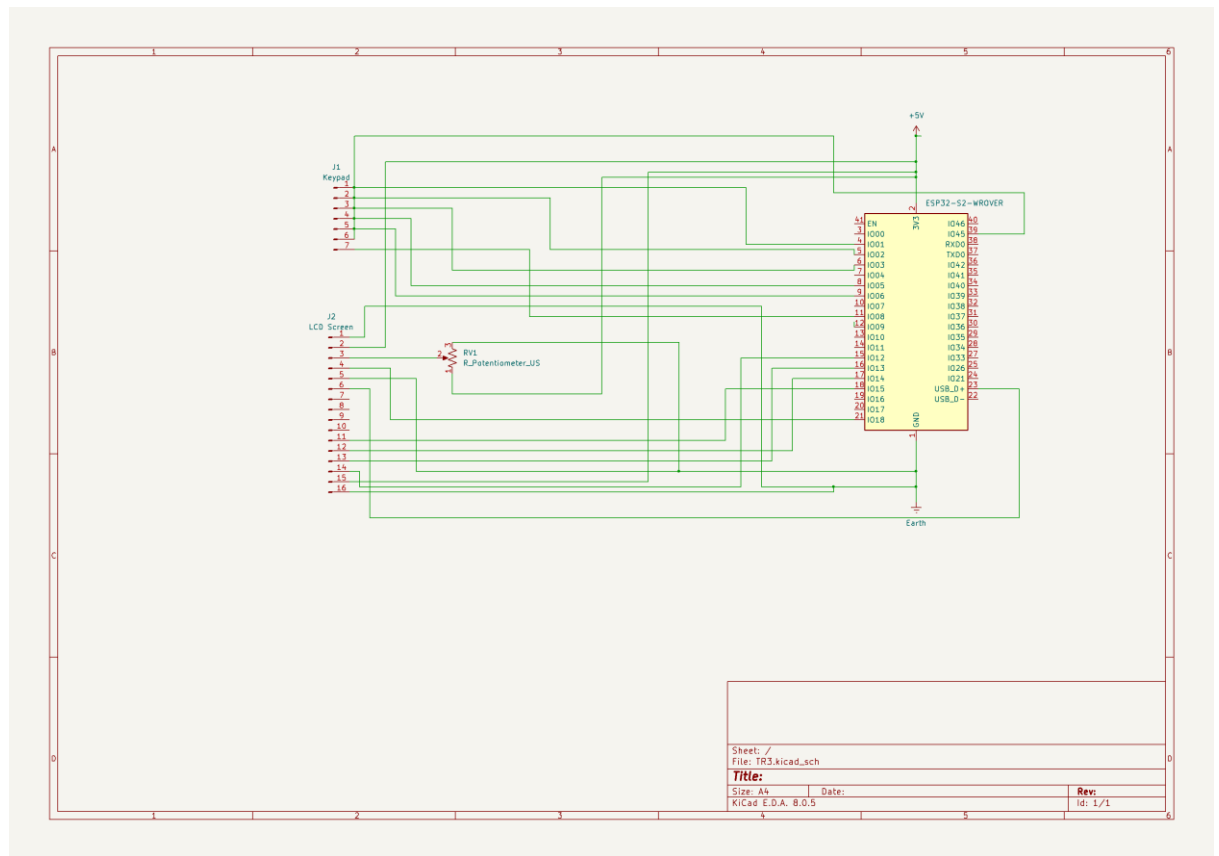


Figure 18: Basic Circuit Schematic

Figure 18 shows a rough schematic of the LCD and keypad circuit needed for both components to work as expected. The LCD Screen uses a potentiometer rated at 10kΩ in order to control its contrast. A potentiometer is a type of variable resistor that transfers a specific amount of input voltage to different terminals. The amount of input voltage at each terminal is determined by the position of the wiper on a resistive track[7]. By varying the voltage at the 2<sup>nd</sup> terminal which connects to the Vo pin (3<sup>rd</sup> pin) on the LCD screen it's contrast can be controlled by changing the position of the wiper. The contrast of an LCD screen is the ratio of bright and dark areas, thus by changing the contrast the output of the LCD screen can become more visible[8].

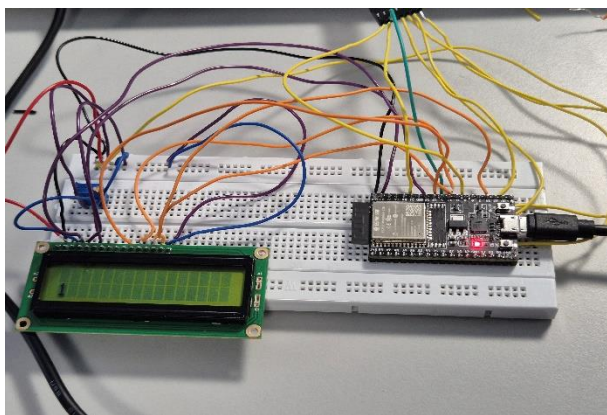


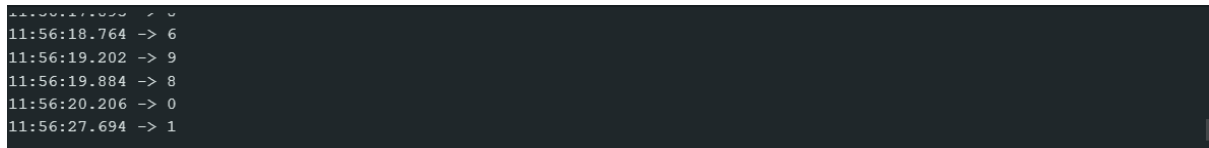
Figure 19: Breadboard Layout

Figure 19 exhibits the breadboard layout of the schematic shown in Figure 18. Through uploading some basic example code, the LCD Screen displays the output of the keypad.

```
void loop() {
    char key = keypad.getKey(); // Get the pressed
    key
    if (key) { // Only process if a key is actually
    pressed
        Serial.println(key); // Print to Serial Monitor
        lcd.setCursor(0, 1);
        lcd.print(key);
    }
}
```

The way the LCD screen displays the output of the keypad is through the test code in Figure 20. The code gets the pressed key and checks whether or not the key is pressed. Then the LCD screen displays the pressed key in the bottom row as the LCD screen is capable of displaying characters in two rows. The bottom row is the '0' row while the top row is the '1' row and each row can be set with the function `lcd.setCursor()`.

Figure 20:Test Code



```
11:56:18.764 -> 6
11:56:19.202 -> 9
11:56:19.884 -> 8
11:56:20.206 -> 0
11:56:27.694 -> 1
```

Figure 21:Serial Monitor Output

### 2.2.2 Stripboard Design

The main aim for the stripboard design is to make it compact as these components will be soldered onto the same stripboard design seen in Figure 8. As such components will have to be placed in a way that does not interfere with the components that are already present thus causing short circuits which may lead to the damage to some components. The biggest challenge is the LCD screen due to the number of pins that need to be connected to the ESP32 or other components. Designing the LCD screen pin out incorrectly may lead to a myriad of issues such as short circuits or voltages going to different pins thus risking damage.

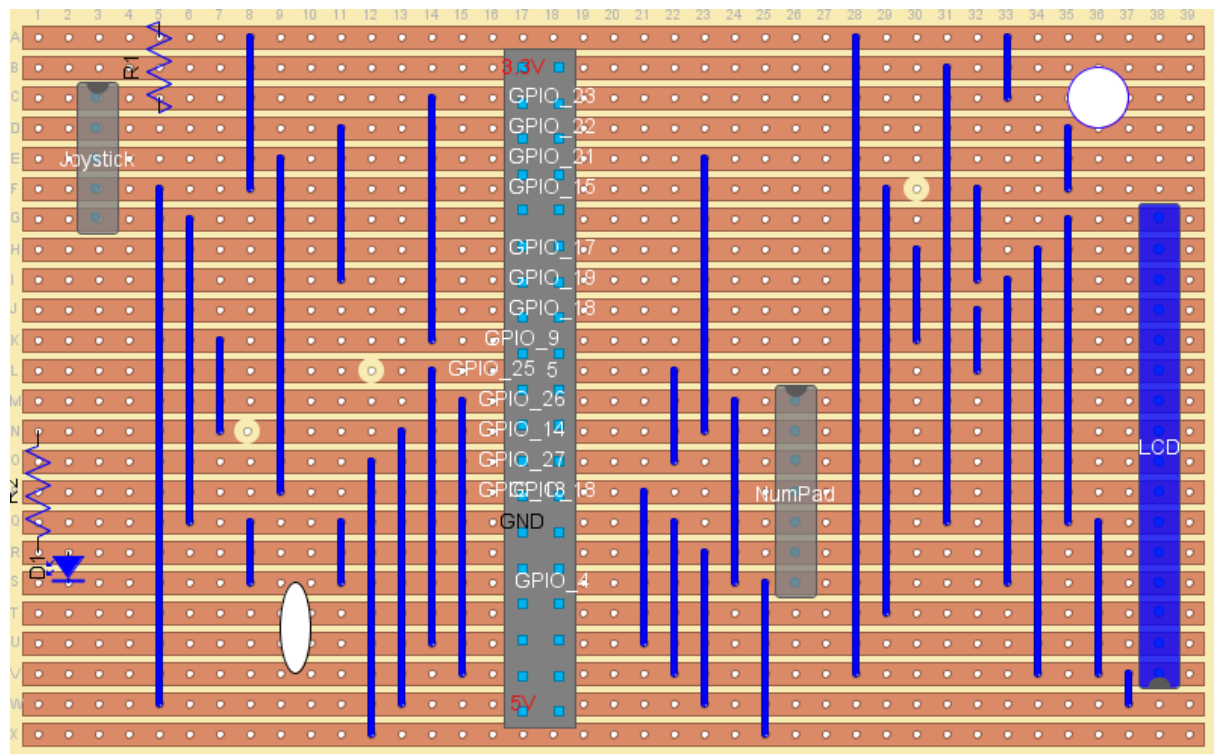


Figure 22: Keypad-LCD Stripboard Design

This design aims to keep the components from interfering with the previous components and also prevent short circuits. The decision to put the LCD at the furthestmost point is to provide space for all the wiring required to make the LCD work. However, this design isn't perfect as due to the sheer number of wires required mistakes can easily be made thus leading to short circuits.

## 2.3 Software Solutions

Similarly to the software solutions for the construction of the wireless remote this will be split between the master and slave code. The master code being the remote ESP32 sending data to the mainboard ESP32.

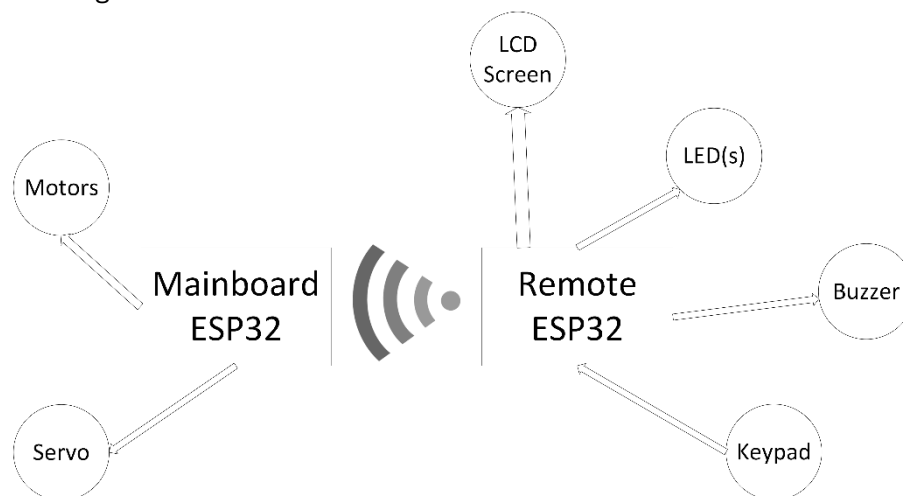


Figure 23: Keypad-LCD System Overview

The system overview diagram displayed above is very similar to Figure 14 with a couple differences. First off, the wireless communication is one way between the remote and the mainboard ESP32 where the remote acts a server and the mainboard ESP32 acts as the client.

### 2.3.1 Master Code

The design behind the master code is similar to the one seen in 1.1.3.2.1 where the output of the keypad is sent wirelessly to the mainboard ESP32.

```
char key = keypad.getKey(); // Get the pressed
key

if (key) { // Only process if a key is actually
pressed

  Serial.println(key);

  valuesToSend.data1 = key;

  valuesToSend.data2 = 0;

  lcd.setCursor(0, 1);

  lcd.print("State:");

  lcd.setCursor(1, 1);

  lcd.print(key)

}
```

This code snippet checks a key has been pressed before proceeding to do anything. If a key has been pressed the character is set to the variable 'valuesToSend.data1' which is the message struct that is sent to the mainboard ESP32 via the ESPNOW communication method. This same key is then printed on both the lcd screen, and the serial monitor for debugging reasons. If the character appearing on the lcd screen is different from the one shown in the serial monitor then its clear that an issue has occurred which could then be fixed.

Figure 24: Master-Code Snippet

### 2.3.2 Slave Code

```
switch (State) {
  case 1:
    Serial.println(State);
    Centralise();
    goForwards();
    delay(Min_DISTANCE_TIME);
    stopMotors();
    break;

  case 2:
    Serial.println(State);
    Centralise();
    goForwards();
    delay(Mid_DISTANCE_TIME);
    stopMotors();
    break;

  case 3:
    Serial.println(State);
    Centralise();
    goForwards();
    delay(Max_DISTANCE_TIME);
    stopMotors();
    break;
}
```

This snippet from the switch case for the slave code shows the different commands that could be done through the keypad. The distance timers seen in the first 3 cases are the time it takes for the vehicle to move 10cm, 20cm and 30cm respectively. The time for the bot to move in these directions was calculated using the equation:

$$t = \frac{255 \times d}{V_{max} \times PWM}$$

This equation is derived from the simple distance equation in which time is equal to the distance of an object divided by the speed of an object. Since the speed of the vehicle is dictated by the PWM value, the speed is equal to the maximum possible speed of the vehicle (which in this case it is assumed to be 1 m/s) multiplied by PWM/255 value. As a result, the time it takes for the vehicle to move any distance can be accurately estimated by plugging in both the PWM value and the distance the vehicle needs to travel. For example, in case one the vehicle will move 10cm if '1' is pressed on the keypad. By plugging 10cm into the equation (first convert 10cm to metres) the time it takes the vehicle to travel 10cm would be around 0.1275 seconds. Since the 'delay()' function works in milliseconds the 'Min\_DISTANCE\_TIME' would be equal to 127.5ms or 128ms if rounded up. This methodology is used in all other cases where the vehicle needs to travel specific distances.

Figure 25: Slave Code Snippet 1-3

```
case 4:

    Serial.println(State);

    min_angle = 0;

    max_angle = 90;

    moveSteering();

    break;
```

```
case 5:

    Serial.println(State);

    min_angle = 0;

    max_angle = 180;

    moveSteering();

    break;
```

```
case 6:

    Serial.println(State);

    min_angle = 0;

    max_angle = 90;

    moveSteeringL();

    break;
```

```
case 7:

    Serial.println(State);

    min_angle = 0;

    max_angle = 180;

    moveSteeringL();

    break;
```

The options 4-7 dictate the rotation of the vehicle and also in which direction. The direction of the vehicles rotation is determined by the type of moveSteering() function used. The default moveSteering() function will cause the vehicle to move right while the moveSteeringL() function will cause the vehicle to move to the left. The two options in regard to the magnitude of rotation is 90° and 180°. A 90° rotation is more useful for traversing the maze especially when facing up against any dead ends. The 180° is used as a fail safe in situations where the vehicle ends up in a scenario it can't move on from so a full half turn is required to get it back to it's last position.

Figure 26: Slave-Code Snippet 4-7

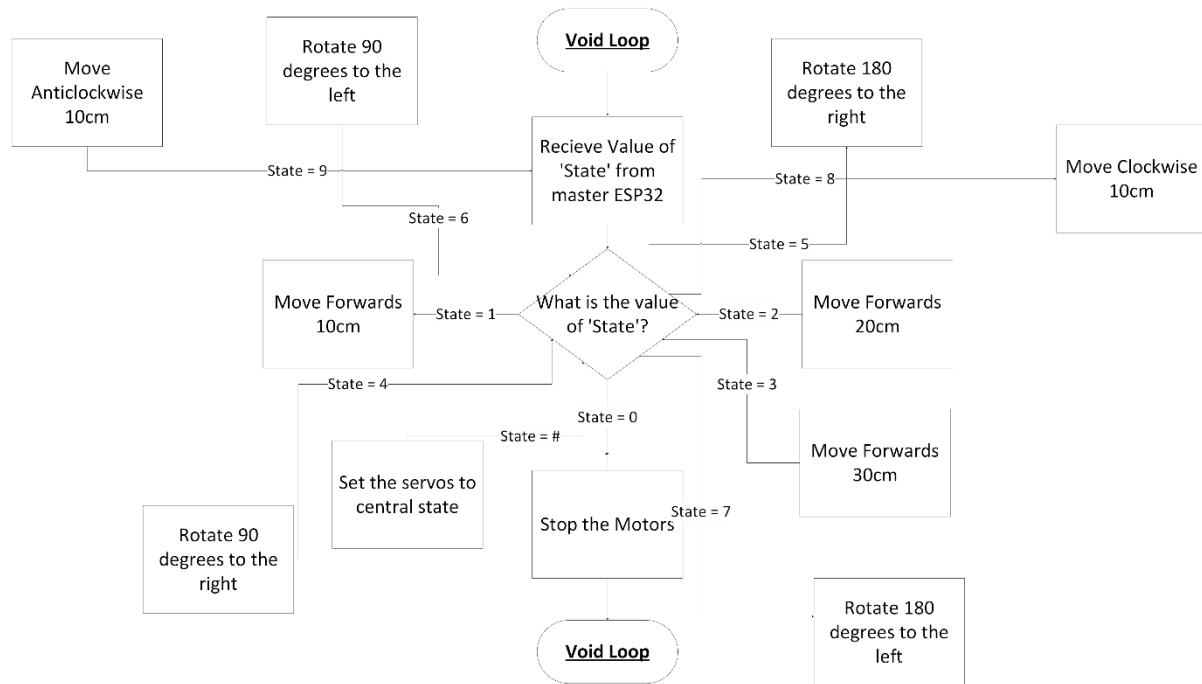


Figure 27: Keypad-Vehicle Options

This flowchart details the other options and also includes the previous options displayed in a more coherent way. The reasoning behind limiting the clockwise and anticlockwise movement to only 10cm is because further distances are unnecessary as the distance required for a left or right turn will rarely ever exceed 10cm. In addition, the decision to use one of the special characters to centralise servos was to fix any situation where the servos may be rotated at the wrong angle or stuck at an angle due to previous use. With the sheer number of options available to the user traversing through the maze using wireless commands should in theory be possible.

## Conclusion

As for my own results, regarding the wireless remote I was able to successfully control the vehicle with the wireless remote I constructed using the hardware and software designs detailed in this report. However, the accuracy of the movement was disrupted by issues with my motors. Due to the unexpected overheating of the motor driver circuit in the mainboard, the left motor stopped functioning leaving the vehicle to move extremely slowly and thus presented difficulties in moving left and right. As a result of this issue, I wasn't able to successfully implement the ultrasound sensory feedback as the majority of time was spent trying to fix the motor issue. When it comes to the keypad control, the flaws I stated earlier with regards to the compact soldering eventually caused issues as I misplaced a couple wires leading to a lot of short circuits which took some time to fix.

## References

- [1] 'In-Depth: How 2-Axis Joystick Works? Interface with Arduino & Processing', Last Minute Engineers. Accessed: Apr. 03, 2025. [Online]. Available: <https://lastminuteengineers.com/joystick-interfacing-arduino-processing/>

- [2] 'Pull-up and Pull-down Resistors | Resistor Applications | Resistor Guide'. Accessed: Apr. 03, 2025. [Online]. Available: <https://eepower.com/resistor-guide/resistor-applications/pull-up-resistor-pull-down-resistor/>
- [3] 'Device to Device Communication with ESP-NOW | Arduino Documentation'. Accessed: Apr. 03, 2025. [Online]. Available: <https://docs.arduino.cc/tutorials/nano-esp32/esp-now/>
- [4] 'Peer-To-Peer Networks: Features, Pros, and Cons - Spiceworks', Spiceworks Inc. Accessed: Apr. 03, 2025. [Online]. Available: <https://www.spiceworks.com/tech/networking/articles/what-is-peer-to-peer/>
- [5] 'What is an ultrasonic sensor? | Sensor Basics: Principle-based Guide to Factory Sensors | KEYENCE'. Accessed: Apr. 06, 2025. [Online]. Available: <https://www.keyence.co.uk/ss/products/sensor/sensorbasics/ultrasonic/info/>
- [6] 'What is a Liquid Crystal Display (LCD)? Advantages & Disadvantages | Lenovo UK'. Accessed: Apr. 07, 2025. [Online]. Available: <https://www.lenovo.com/gb/en/glossary/what-is-lcd/>
- [7] 'Potentiometers - A Complete Guide'. Accessed: Apr. 07, 2025. [Online]. Available: <https://uk.rs-online.com/web/content/discovery/ideas-and-advice/potentiometers-guide>
- [8] 'Adjusting the Contrast of an LCD Module - Focus LCDs'. Accessed: Apr. 07, 2025. [Online]. Available: <https://focuslcs.com/journals/application-notes/adjusting-the-contrast-of-an-lcd-module/>

## Appendix

- [1] LiquidCrystal.h Library - <https://docs.arduino.cc/libraries/liquidcrystal/#Usage/Examples>
- [2] Keypad.h Library - <https://docs.arduino.cc/libraries/keypad/#Compatibility>
- [3] LCD Screen to ESP32 Pinout - <https://www.circuitschools.com/interfacing-16x2-lcd-module-with-esp32-with-and-without-i2c/>