THM TECHNISCHE HOCHSCHULE MITTELHESSEN
CAMPUS FRIEDBERG
IEM Informationstechnik-Elektrotechnik-Mechatronik

CCCE
Case Study Project
Summer Semester 2024

Design and Development for the Control of a Mecanum Wheel Car

Anil Manubhai Chhotala
Matriculation Number: ...

Axit Kantibhai Kakadiya
Matriculation Number: ...

Devraj Ajaykumar Solanki
Matriculation Number: ...

Edmund Jochim
Matriculation Number: 5729098

Supervisor: Prof. Dr.-Ing. Fabian Mink

Submitted on: xx.09.2024

Abstract

Table of Contents

1 Introduction **1**

2 Theoretical Background **2**

2.1 Mecanum Wheel Technology | EJ 2

2.2 Motor Control using PWM 5

2.3 Wireless Communication Protocols 5

2.3.1 WiFi and UDP 5

2.3.2 Bluetooth 5

2.4 Sensor Integration 5

3 Methodology **6**

3.1 Hardware Selection 6

3.1.1 Microcontroller (ESP32) | EJ 6

3.1.2 Power System | EJ 8

3.1.3 Motor Driver 9

3.1.4 Additional Components 9

3.2 Software Development 9

3.2.1 Command Protocol Design | EJ 9

3.2.2 Python Module | EJ 10

3.2.3 Firmware for ESP32 | EJ 10

3.2.4 Motor Control Algorithm 12

3.2.5 Sensor Integration 12

3.2.6 Android App | EJ 12

3.3 Communication Interface 14

3.3.1 WiFi/UDP Control System 14

3.3.2 Bluetooth Integration with PS4 Controller 14

4 Results and Discussion **15**

4.1 Hardware Assembly 15

4.2 Software Performance 15

4.3 Sensor Data and System Feedback 15

4.4 Challenges and Limitations 15

5 Conclusion **16**

Declaration of Authorship

1. Introduction

2. Theoretical Background

2.1 Mecanum Wheel Technology | EJ

Mecanum wheels, a distinctive omnidirectional wheel design, enable vehicles to maneuver freely in any direction, including forward, backward, laterally, and rotationally. This unique capability is achieved through the arrangement of rollers on the wheel, which are mounted at an angle to the wheel's axis. The side view of such an wheel can be seen in fig. 2.1. Unlike conventional wheeled systems, which are limited to two degrees of freedom (longitudinal and steering), Mecanum wheels provide full omnidirectional movement with three Degrees of Freedom (DoF): longitudinal, lateral, and rotational (yaw) [1].

The rollers on a Mecanum wheel are positioned such that their axes are skewed relative to the

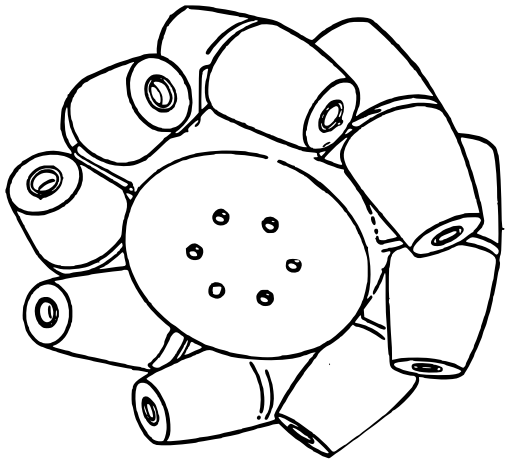

Figure 2.1: Side view of a Mecanum wheel [1]

central wheel axle. This geometry allows each wheel to produce a vector of force that, when combined with the forces generated by the other wheels, results in omnidirectional motion. This configuration avoids the singularities common in traditional wheel systems, which often require significant propulsion adjustments to perform small or intricate maneuvers. Vehicles equipped with Mecanum wheels are therefore able to perform complex maneuvers in confined spaces with greater efficiency, reducing both the time and area required for movement [1].

The kinematics of Mecanum wheels also play a crucial role in achieving precise control over a vehicle's movement. As each wheel operates independently, the control system must convert the desired motion into specific commands for each wheel, accounting for the interaction between longitudinal, lateral, and rotational forces. While this may seem computationally

complex, efficient algorithms have been developed to simplify the process. These algorithms, often incorporating compensation for wheel slip, ensure that the vehicle can execute smooth, omnidirectional movement without requiring significant computational overhead [1]. In this case a simple control algorithm will be developed that does not use a compensation for wheel slip. As it is a fairly simple and lightweight remote controlled car for studying purposes, the required precision is not so high, that it would justify the increased development effort.
One notable disadvantage of the Mecanum wheel design is the inefficient transfer of kinetic energy from the motors to the ground. As the exterior rollers rotate, only a portion of the force generated by the wheels is effectively applied to the ground. This is due to the angled orientation of the rollers, which causes the total force exerted by each wheel to be split into components. Consequently, only a fraction of the force directly contributes to the vehicle's motion, leading to reduced overall efficiency in propulsion compared to conventional wheels [2].
Therefore, Mecanum wheels are not commonly used in applications where efficient energy use is a priority but are favored in scenarios where the efficient use of time and space is essential. Many researchers focus on utilizing these wheels in the design of autonomous vehicles [3] and robots, particularly for environments where maneuverability is critical. Omni-directional platforms, like those equipped with Mecanum wheels, offer the ability to move instantly in any direction from any position. This provides significant advantages in congested environments filled with static and dynamic obstacles, such as factory workshops, warehouses, hospitals, and elderly care facilities, where conventional wheeled designs would struggle to navigate narrow aisles and tight spaces efficiently [4].

In this design, four Mecanum wheels are arranged at the four corners of a rectangular vehicle platform, with each wheel oriented to face forward along the X-axis, as can be seen in fig. 2.2. The arrangement allows the vehicle to achieve omnidirectional movement, with the X-axis rep-

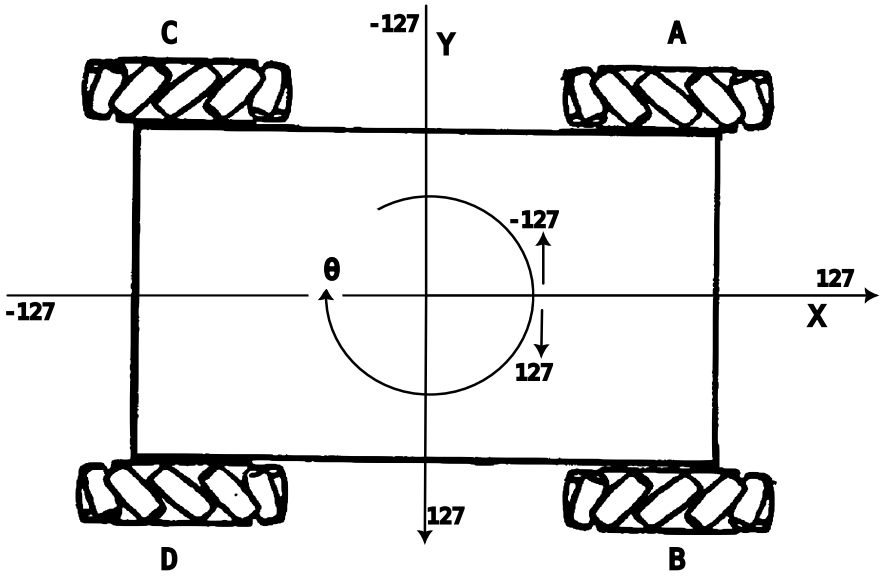

Figure 2.2: Wheel configuration and coordinates of Mecanum wheel car [1]

resenting forward and backward motion, the Y-axis representing lateral (side-to-side) motion,

and the θ corresponds to rotational (yaw) movement.
The control system uses 8-bit signed integers to define motion in each direction, with values ranging from -127 to 127 to ensure symmetry in both positive and negative directions. While 128 would be the theoretical maximum for positive values, it is capped at 127 to maintain balance with the maximum negative value of -127, thus allowing precise and symmetrical control over the vehicle's motion in all three axes.
In order to achieve the desired translational movement of the vehicle along the X and Y axes (see fig. 2.2), each of the four Mecanum wheels must rotate in a specific direction. The table 2.1 provides a detailed breakdown of the rotational direction of each wheel corresponding to various translational movements. The wheels, designated as A, B, C, and D in the configuration, can rotate either positively, negatively, or remain idle (0) depending on the intended movement. A positive direction (+) indicates forward rotation, while a negative direction (-) refers to backward rotation.

Table 2.1: Rotation Direction for Translational Movement [3]

Wheel	**A**	**B**	**C**	**D**
Forward	+	+	+	+
Back	-	-	-	-
Left	-	+	+	-
Right	+	-	-	+
Forward-Right	+	0	0	+
Forward-Left	0	+	+	0
Back-Right	-	0	0	-
Back-Left	0	-	-	0
CW Turn	+	-	+	-
CCW Turn	-	+	-	+

The precise coordination of wheel rotations enables the vehicle to move efficiently in any direction, leveraging the omni-directional capability of the Mecanum wheel design. To translate user input into individual wheel velocities for the Mecanum wheel vehicle, a control scheme is employed that requires three signed integer values. These values correspond to the movement along the X-axis, Y-axis, and the rotational motion (θ). The user provides these three inputs to control the vehicle's motion, which are represented as a 1x3 vector. This vector is then multiplied by a pre-defined translational matrix to convert the input into wheel-specific velocity values for each of the four wheels—designated A, B, C, and D. The resulting output is a 1x4 vector that determines the movement of each wheel individually. The transformation is achieved through the matrix equation 2.1.

$$\begin{bmatrix} x_{val} & y_{val} & \theta_{val} \end{bmatrix} \cdot \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & -1 & +1 \\ +1 & -1 & +1 & -1 \end{bmatrix} = \begin{bmatrix} A & B & C & D \end{bmatrix} \tag{2.1}$$

The 3x4 matrix represents the relationship between the vehicle's translational and rotational movements and the corresponding velocities of each wheel. The output vector provides the computed speed and direction for each of the four wheels.

2.2 Motor Control using PWM

2.3 Wireless Communication Protocols

2.3.1 WiFi and UDP

2.3.2 Bluetooth

2.4 Sensor Integration

3. Methodology

3.1 Hardware Selection

3.1.1 Microcontroller (ESP32) | EJ

The selection of an appropriate microcontroller was a critical aspect of the hardware design for this project. Given the project's constraints and the need to utilize available components, it was necessary to choose a microcontroller based on readily accessible development boards. After evaluating several options, the ESP32 was identified as the most suitable choice due to its versatile features and broad compatibility with the project's requirements.
The ESP32 microcontroller is a highly versatile and capable device that offers a broad range of features, making it ideal for this project. At its core, the ESP32 incorporates a dual-core, 32-bit Tensilica LX6 microcontroller, based on the Harvard architecture. This architecture separates data and instruction memory, enhancing processing efficiency. The CPU speed is adjustable, ranging from $160\ \mathrm{MHz}$ to $240\ \mathrm{MHz}$, allowing the system to optimize for either performance or energy consumption depending on the operational requirements [5]. It offers several advantages over other similarly priced microprocessor. Its very low power consumption is particularly beneficial in battery-powered systems like this remote controlled vehicle, extending operational time while minimizing energy drain. Additionally, the ESP32 boasts a high level of integration, including a built-in antenna and step-down converter, simplifying the overall hardware design and reducing the need for additional components [6].
The ESP32 Dev Kit C is a widely used development board, designed for easy prototyping with the ESP32 microcontroller, it can be seen in fig. 3.1. The key components of the board are marked as follows:

1. **Micro USB connector**: Used for powering the board and programming the ESP32 via a USB connection.
2. **AMS1117 Voltage regulator**: Ensures the board operates at the required $3.3\ \mathrm{V}$ by stepping down input voltage from the Vin pin or the USB connection.
3. **CP2102 USB to UART bridge**: Facilitates communication between the USB port and the ESP32, enabling data transfer and debugging.
4. **ESP32-WROOM-32 module**: The heart of the board, containing the ESP32 microcontroller.

5. **Antenna**: Built-in antenna for wireless communication.

Other notable connections include:

- **GND**: Ground connection used as the negative voltage reference.
- **3V3**: Provides $3.3\ \mathrm{V}$ output or requires exactly $3.3\ \mathrm{V}$ of input supply voltage for stable operation.
- **Vin**: Can accept a wide input voltage range from $4.4\ \mathrm{V}$ to $12\ \mathrm{V}$ [7], regulated down to $3.3\ \mathrm{V}$ internally.

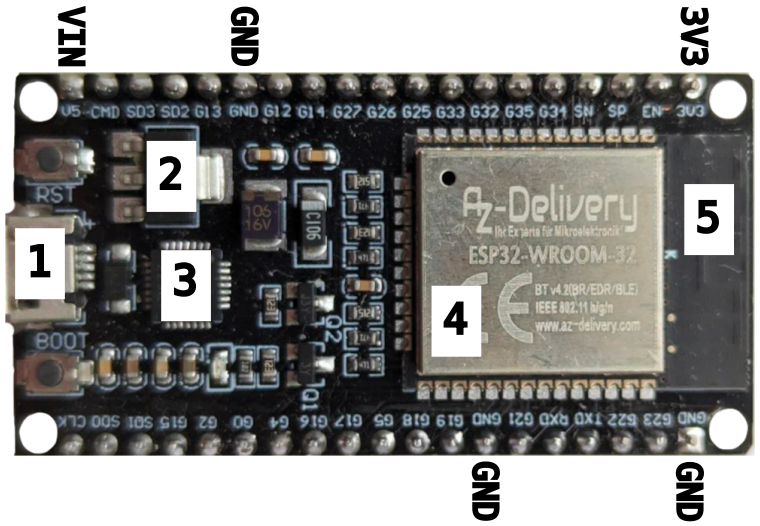

Figure 3.1: ESP32 Dev Kit C Board

The ESP32 also boasts an extensive range of input and output (I/O) options. It has 36 general-purpose input/output (GPIO) pins, 18 of which can be used as analog-to-digital converter (ADC) inputs, allowing the microcontroller to read sensor data and other analog signals. Furthermore, the ESP32 is equipped with two dedicated digital-to-analog converter (DAC) pins, which can generate true analog outputs, a feature that is particularly useful in applications where precise control over analog signals is required. Almost all output-capable pins can also produce pulse-width modulation (PWM) signals, which are integral to this project for controlling motor speed and direction [5].
In terms of memory, the ESP32 features both on-chip (internal) and onboard (external) memory. The internal memory is composed of 448 KB of ROM, which is responsible for booting and core functions, and 520 KB of SRAM used for data storage and instructions during runtime. Additionally, the ESP32 supports up to 16 MB of external ISP flash RAM, which can be used to store larger programs or data sets that exceed the capacity of the internal memory [5]. One of the ESP32's most significant features is its built-in communication capabilities. For wireless communication, it supports WiFi with full 802.11 b/g/n/e/i standards. This allows the ESP32 to function as both a client, connecting to a router, and as an access point, creating its own wireless network. This versatility is beneficial for remotely controlling the vehicle via User Datagram Protocol (UDP) packets over WiFi. The Bluetooth module supports v4.2 BR/EDR and Bluetooth Low-Energy (BLW) standards, with data transfer speeds of up to 4.0 Mbps, making

it possible to control the vehicle with Bluetooth devices, such as a Sony Playstation 4 (PS4) controller [5].
In addition to its wireless communication, the ESP32 supports several wired communication protocols, including UART, I2C, ISP, CAN, and I2S. These protocols allow the microcontroller to communicate with a variety of peripheral devices, such as sensors, motor drivers, and other controllers, providing flexibility in system design and integration [5].
The ESP32 can be programmed using two main frameworks. The first, ESP-IDF, is the official development framework provided by Espressif Systems. It offers low-level access to the hardware and allows developers to take full advantage of the ESP32's dual-core architecture. However, due to its complexity, it is more suited for advanced projects requiring intricate customization. The second framework is the Arduino IDE, which simplifies development by abstracting some of the low-level details [5]. Although it offers less flexibility in core management compared to ESP-IDF, the Arduino framework provides sufficient functionality for this project and is widely supported with extensive libraries and an open-source community. Given the project's requirements and the ease of development, the Arduino framework was chosen for this application.

3.1.2 Power System | EJ

The power system for the Mecanum wheeled car is designed around two lithium polymer (LiPo) batteries connected in series. Each LiPo cell has a nominal voltage of $3.7\ \mathrm{V}$, resulting in a total voltage of $7.4\ \mathrm{V}$ when combined. This configuration provides sufficient voltage to drive the motors and other key components of the car, offering a compact and efficient energy solution for the system. Initially, the plan was to use a dedicated $5\ \mathrm{V}$ output step-down converter to supply power to the ESP32 microcontroller. However, this approach was abandoned due to issues arising from the converter's inability to handle the low power requirements of the ESP32 in isolation. The step-down converter was designed for heavier loads, and with only the microcontroller connected, it was operating under a "too light" load condition, causing instability in the power supply.
To resolve this, the power system was simplified. The $7.4\ \mathrm{V}$ output from the LiPo batteries became the primary supply voltage for both the motor driver and the ESP32 microcontroller. This configuration is advantageous for two reasons. The motor driver is capable of handling the $7.4\ \mathrm{V}$, which is adequate to power the motors at optimal performance. The ESP32 development board has its own built-in voltage regulator capable of stepping down input voltages as high as $12\ \mathrm{V}$ to its required $3.3\ \mathrm{V}$ operating voltage. Therefore, the same $7.4\ \mathrm{V}$ battery supply could be used to power the ESP32 directly, eliminating the need for an external converter and simplifying the system's overall design. The ESP32 development board includes a dedicated pin that outputs a regulated $3.3\ \mathrm{V}$, enabling the powering of additional components, such as sensors, that lack their own voltage regulation. This simplifies integration by providing a stable power source for peripherals directly from the microcontroller. This revised power strategy enhances the robustness of the power system while reducing component count and complexity, ensuring reliable operation of both the microcontroller and motors.

3.1.3 Motor Driver

3.1.4 Additional Components

3.2 Software Development

3.2.1 Command Protocol Design | EJ

The development of a custom command protocol for controlling the Mecanum wheeled car was essential to ensure smooth and efficient communication between the user and the vehicle. The design is inspired by the Software Development Kit (SDK) of the Ryze Tello Drone, which provided a useful framework for structuring commands in a clear and scalable manner [8]. In this system, the Mecanum car operates as a WiFi access point, with a fixed IP address of 192.168.10.1. Devices connecting to the car, such as smartphones or laptops, can communicate through UDP protocol on specific ports. Commands are sent to UDP port 4444, and status reports are received on UDP port 4445.
The command protocol allows for three core commands to be sent to the Mecanum car. These commands, described in the command list, include:

- **rc x y z**: The primary control command that applies speed values to the car's motion. x, y and z are 8-bit signed integer values that correspond to the user input that is translated to the wheels.
- **keepalive**: This command maintains the connection between the client device and the car without issuing any new control instructions. It prevents the system from timing out and maintains the current state of motion until new instructions are given.
- **stop**: This command halts all movement, immediately setting the speeds on all axes to zero, ensuring the car stops regardless of its previous state.

Each of these commands is transmitted as a simple string over UDP to ensure minimal latency in communication, and responses such as "ok" or "error" are returned depending on the validity and execution of the command.
In addition to receiving commands, the Mecanum car provides real-time feedback via status reports. These reports are sent every second through UDP port 4445 to the last client device that issued a command. The report is transmitted as a string containing various key data points about the car's current state. This includes:

- **Battery Voltage** (batV): The voltage level of the car's battery, reported to two decimal places.
- **Battery Percentage (batP)**: The remaining charge of the battery, expressed as a percentage.
- **Motor Duty Cycles** (mA, mB, mC, mD): The current PWM duty cycles for each of the four motors (A, B, C, D), expressed in percentage values.

This status feedback is critical for maintaining control, as it allows the user to monitor battery levels and motor activity in real time, ensuring effective command execution and enabling adaptive decision-making during vehicle operation.
To implement the command protocol, the Mecanum car requires a UDP client to be set up on the controlling device. The client sends commands to the car and listens for responses on the designated ports. The fixed IP address simplifies the connection process, while the compact command set ensures a minimal overhead for data transmission, making the system both lightweight and responsive. With the foundation of this protocol, the car can be efficiently controlled in real-time, whether for basic movement or more complex, synchronized operations. The whole documentation of Mecanum Car Commands in the style of the Ryze Tello Drone SDK can be accessed in the appendix.

3.2.2 Python Module | EJ

A python module is developed to provide access to control the car using its SDK with short and easy commands in Python. The user only has to connect a client, where the Python module is installed, to the car's Wi-Fi Access Point (AP). Each Python command is translated to a string that is sent to the car's IP address and a predefined port. This string is then interpreted by the firmware running on the microcontroller that controls the car. The commands used for steering the car are interpreted as a vector, which is used to control the direction.

3.2.3 Firmware for ESP32 | EJ

The firmware developed for the ESP32 is the core of the Mecanum wheel car's control system, responsible for managing communication protocols, motor control, and system monitoring. The structure of the firmware has been designed to handle different modes of operation, including control via WiFi and Bluetooth, while providing real-time feedback on the car's status, including battery level monitoring.
The firmware begins with the inclusion of essential libraries such as MotorPWM, UDPcontrol, PS4Control, BatteryStatus, and StatusLED. These libraries are integral to the car's operation, providing control over motor speed via PWM, handling the UDP or PS4 control modes, monitoring battery levels, and updating LED indicators for status feedback.
The user can choose between two control modes by defining either `UDP_CONTROL_MODE` or `PS4_CONTROL_MODE`. Only one of these modes should be active at any time, controlling whether the car is operated via WiFi (UDP) or through a PS4 controller using Bluetooth. The `SERIAL_DEBUG` define can be uncommented to enable serial debugging, which prints relevant system information such as motor speeds and battery levels to the serial monitor. This is useful for development and testing.

The `setup()` function is responsible for initializing all key components, including the status LED, playing the startup animation indicating successful booting, and to start the communication

processes. At the end of the startup process the battery value is checked once to store it in memory. Additionally memory space is reserved for the motor control arrays where the control input from the user is and the PWM values for motors are stored. The whole process can be seen in fig. 3.2.

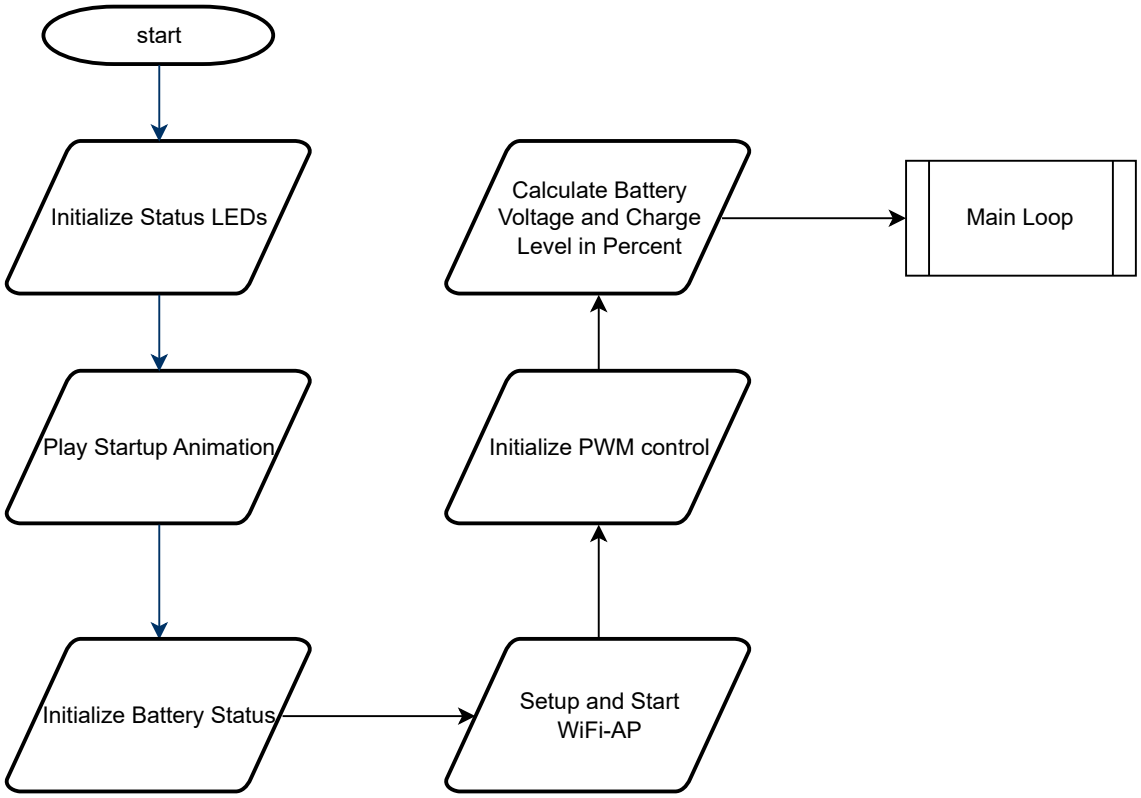

Figure 3.2: Setup Sequence of Microprocessor

Afterwards the main `loop()` function is entered, which loops infinitely. This function continuously checks for control input by the user and updates the motor outputs.

This function is the main execution loop that continuously checks for control input (either from UDP packets or PS4 controller), calculates motor speeds, and updates the motor outputs. It also checks the battery status every second and sends status messages back to the controller in UDP mode. At the same time this function checks the battery status periodically and sends the status message back to the user after a specified time interval passes. This process can be seen in fig. 3.3.

The firmware for the ESP32 microcontroller plays a crucial role in the operation of the Mecanum wheel car. It efficiently manages the control modes, motor control, and system monitoring while providing flexibility in terms of communication protocols (WiFi/UDP or Bluetooth). The modularity of the firmware allows for future extensions and adjustments as needed for enhanced functionality or new features. For example new sensors can be added which are checked at the same time as the battery level. The status message can also be expanded to accustom for more sensor data that needs to be transmitted.

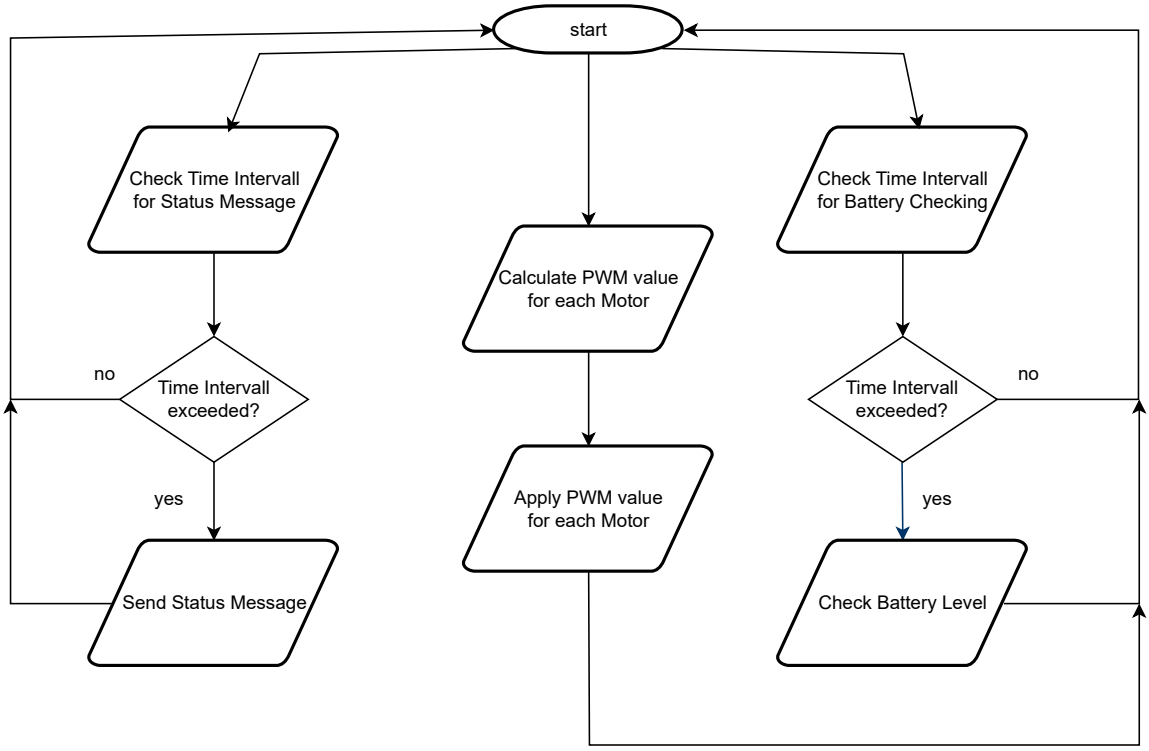

Figure 3.3: Main Loop Sequence of Microprocessor

3.2.4 Motor Control Algorithm

3.2.5 Sensor Integration

3.2.6 Android App | EJ

The Android application used to control the Mecanum wheel car was developed in Kotlin and built using Android Studio. The primary motivation behind creating the app was to offer a smartphone-based control system for the car via WiFi, using UDP packets for communication. This system offers a simple interface for directional movement, speed control, and vehicle status feedback, all integrated into an intuitive layout designed for ease of use, which can be seen in fig. 3.4.

The app allows users to control the car by sending directional commands to move the car in one of eight base directions—up, down, left, right, and the diagonals—using on-screen buttons. Additionally, users can rotate the car clockwise or counterclockwise using designated buttons. The interface ensures that each directional button initiates movement when pressed and sends a "stop" command as soon as the button is released. If a button is pressed for longer than 8 seconds, the app sends a "keepalive" message, ensuring the car does not time out due to inactivity. At the bottom of the screen, a slider allows the user to adjust the speed of the car dynamically, ranging from slow to fast movement. The speed value is converted into control commands that are transmitted via UDP to the car.

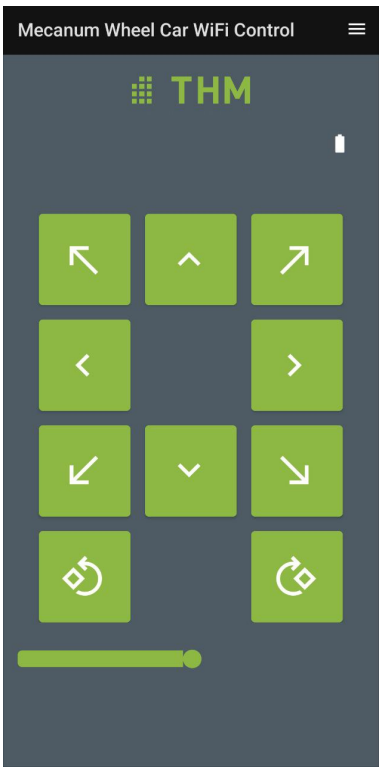

Figure 3.4: Screenshot from the Android App

The layout of the app consists of three key areas:

- **Directional Buttons**: These are located centrally on the screen and allow users to control the car's movement. Each button corresponds to one of the eight base directions or rotation commands (clockwise and counterclockwise).
- **Speed Control Slider**: Positioned at the bottom, this slider allows users to adjust the car's speed. The value is continuously read and used to form the movement command strings sent over UDP.
- **Status Area**: The top of the app displays a battery symbol, which is updated with real-time information about the car's battery status and motor activity, based on UDP status messages sent by the car.

To control the car, the app transmits UDP packets to the Mecanum car, which is set to the IP address "192.168.10.1" on port 4444. A mapping function inside the app converts button presses into the respective command strings in the format "rc x y z". Here, x and y represent the speed in the respective direction, and z indicates rotation speed. The app ensures that each command reflects the user's intended movement, and every release of a button triggers a "stop" command to halt the car. The app makes use of the keepalive-mechanism, which is automatically triggered if a button is held down for more than 8 seconds. The app sends this message to prevent the car from entering a timeout state during continuous use. The keepalive packet is sent periodically to maintain the connection and avoid interruptions in operation.
In addition to sending movement commands, the app listens for status updates from the car on UDP port 4445. These updates contain vital information about the car's battery voltage

and motor performance, which are displayed in the status area at the top of the screen. The data is presented clearly for the user, with battery voltage and percentage shown next to a battery icon, while individual motor performance is displayed in terms of percentage. The app's interface also includes a battery status checker, which updates in real time based on incoming messages from the car. This ensures that the user can monitor the car's battery condition and make informed decisions about operational time.
Though the current version of the app provides a reliable interface for controlling the Mecanum car, future iterations could integrate additional features such as customizable control schemes or enhanced feedback, such as warnings when the battery is low. This Kotlin-based application forms a robust foundation for further development, ensuring that users can control the car seamlessly over WiFi, with real-time status monitoring and intuitive controls.

3.3 Communication Interface

3.3.1 WiFi/UDP Control System

3.3.2 Bluetooth Integration with PS4 Controller

4. Results and Discussion

4.1 Hardware Assembly

4.2 Software Performance

4.3 Sensor Data and System Feedback

4.4 Challenges and Limitations

5. Conclusion

List of Acronyms

ADC analog-to-digital converter
AP Access Point
BLW Bluetooth Low-Energy
CW clockwise
CCW counterclockwise
DAC digital-to-analog converter
DoF Degrees of Freedom
GPIO general-purpose input/output
LiPo lithium polymer
PS4 Sony Playstation 4
PWM pulse-width modulation
SDK Software Development Kit
UDP User Datagram Protocol
EJ Edmund Jochim

Appendix

Mecanum Car Commands

Use Wi-Fi to establish a connection between the Mecanum Car and a client device.

Send Command & Receive Response

Mecanum Car IP: **192.168.10.1** UDP Command Port: **4444** UDP Status Port: **4445**

Step 1: Set up a UDP client on the client device to send and receive messages from the Mecanum Car via the same port.

Step 2: Send any available command to the Mecanum Car to control it.

Control Commands

Command	Description	Possible Response
rc x y z	Apply speed to x y or z direction. x (backward - forward) = -127 to 127 y (left - right) = -127 to 127 z (rotation ccw - cw) = -127 to 127	-/error
keepalive	Keeps connection to Mecanum Car alive. Prevents timeout without changing last command.	ok/error
stop	Set all directions to 0.	ok/error

Status Message | Data Type: String

Data string received every second while controlling via WiFi

"batV:%.2f,batP:%d,mA:%d,mB:%d,mC:%d,mD:%d,:\n"

Description

"batV" = Measured voltage of battery in Volt

"batP" = Charge of battery in Percent

"mX" = Duty cycle of motor X in Percent

References

[1] S. L. Dickerson and B. D. Lapin, “Control of an omni-directional robotic vehicle with Mecanum wheels,” in *National Telesystems Conference proceedings*. New York: Inst. of Electrical and Electronics Engineers, 1991, pp. 323–328.

[2] F. Adascalitei and I. Doroftei, *Practical applications for mobile robots based on Mecanum wheels - a systematic survey*, 2011. [Online]. Available: https://www.researchgate.net/profile/ioan-doroftei/publication/233867057_practical_applications_for_mobile_robots_based_on_mecanum_wheels_-_a_systematic_survey

[3] N. Tlale and M. de Villiers, “Kinematics and Dynamics Modelling of a Mecanum Wheeled Mobile Platform,” in *2008 15th International Conference on Mechatronics and Machine Vision in Practice*, T. Moir, Ed. Piscataway, NJ: IEEE, 2008, pp. 657–662.

[4] O. Diegel, A. Badve, G. Bright, J. Potgieter, and S. Tlale, “Improved Mecanum Wheel Design for Omni-directional Robots,” 2002. [Online]. Available: https://forums.parallax.com/uploads/attachments/49124/55525.pdf

[5] H. Kareem and D. Dunaev, “The Working Principles of ESP32 and Analytical Comparison of using Low-Cost Microcontroller Modules in Embedded Systems Design,” in *2021 4th International Conference on Circuits, Systems and Simulation (ICCSS 2021)*. Piscataway, NJ: IEEE, 2021, pp. 130–135.

[6] Espressif Systems, “ESP32 Wi-Fi & Bluetooth SoC | Espressif Systems,” 2024. [Online]. Available: https://www.espressif.com/en/products/socs/esp32

[7] Advanced Monolithic Systems Inc, “Datasheet AMS1117: 1A Low Dropout Voltage Regulator.” [Online]. Available: http://www.advanced-monolithic.com/pdf/ds1117.pdf

[8] Ryze Tech, “Tello SDK 2.0 User Guide,” 2018. [Online]. Available: https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf