Politechnika Wrocławska

ADVANCED TOPICS IN ROBOTICS - PROJECT

MEGATRON – Robot Car with Mecanum Wheels

Final Report by **Onurcan Icen – 269517**
Ata Göker – 276828
Tan Öztürk - 276731

WROCŁAW, 2025

Table of Contents

1. Problem Statement and Motivation 3
2. Solution 3
    2.1 Idea of the Solution 3
    2.2 Tools Used 3
    2.3 Details of the Solution 4
3. State-of-the-Art 4
    3.1 Overview of Current Technologies 4
    3.2 Mecanum Wheels in Robotics 4
    3.3 Sensor and Vision Integration 4
    3.4 Real-Time Control and Connectivity 4
    3.5 Comparative Analysis 4
    3.6 Autonomous Navigation Insights 5
    3.7 Unique Features of Megatron 5
4. Megatron - Project Charter 5
    4.1 Project Description 5
    4.2 Reasons for Undertaking the Project 6
    4.3 Objectives of the Project 6
    4.4 Constraints of the Project 6
    4.5 Directions Concerning the Solution 6
    4.6 Main Stakeholders and Responsibilities 6
    4.7 Risks Identified Early On 6
    4.8 Target Project Benefits 7
5. Megatron - Project Schedule 7
    5.1 Tasks 8
    5.2 Milestones 8
    5.3 Gantt Chart 9
6. Tests and Results 9
    6.1 Methodology 9
    6.2 Results 9
7. Summary 9
    7.1 Evaluation of the Project 9
    7.2 Possible Enhancements 10
Appendices 10
    A. Project Plan 10
    B. Task Distribution 10
8. Hardware 10

8.1 Components Selection 10
8.2 Board Dimensions..11
8.3 Hardware Structure 12
8.4 Wiring Diagram 14
8.5 Final Look After Assembly 14
8.6 First view after connecting the robot 15
8.6 View After Configuring VCN 15
8.7 VCN Tools..16
9. Robot Firmware 17
9.1 Board..17
9.2 Misc.py..17
9.3 PID.py 17
9.4 Mecanum.py..17
9.5 Megatron.py 18
9.6 MjpgServer.py 18
9.7 RPCServer.py..18
10. Functions..18
10.1 Emptyfunc.py..18
10.2 ImgAddText.py..19
10.3 RemoteControl..19
10.4 Running.py..20
11. Camera Calibration 20
11.1 Calibration.py..20
11.2 Calibration Configuration.py 20
11.3 Collect Calibration Picture.py..21
11.4 Generate Calibration Plate.py 21
11.5 Test Calibration.py..21
11.6 Calibration Board..22
12. App Designing 22
12.1 About..22
12.2 Index 25
Key Features: 32
Styling:..33
User Interaction:..33
Connectivity:..33
Summary:..33
13. Range of Application and Limitations..34

13.1 Future Work 34
13.2 Summary 35
13.3 Conclusion 35

Megatron: Omnidirectional Mecanum Wheels Robot Car Project

1. Problem Statement and Motivation

The **Megatron** project addresses the need for versatile robotic systems capable of navigating complex environments. Leveraging mecanum wheels, this project enables omnidirectional movement, offering significant potential for use in logistics, surveillance, and automated assistance. Unlike traditional robots, this design ensures flexibility and precision in confined spaces, setting it apart from existing models.

2. Solution

2.1 Idea of the Solution

The proposed system integrates:

- A Raspberry Pi as the central processing unit.
- Mecanum wheels for omnidirectional mobility.
- An HD camera for real-time visual feedback.
- Sensors for obstacle detection and navigation.

The robot employs Python-based control software interfacing with hardware components via ROS (Robot Operating System).

2.2 Tools Used

Software

- Python 3.9
- ROS 2 Galactic
- OpenCV 4.5
- TensorFlow 2.7 (for object detection)

Hardware

- Raspberry Pi 4B with camera module
- ESP32 microcontrollers

- Mecanum wheels chassis
- Ultrasonic sensors (HC-SR04)
- L298N motor driver

2.3 Details of the Solution

The system comprises interconnected components managed through ROS topics:
Raspberry Pi: Handles image processing and decision-making.
ESP32: Manages motor control and sensor data.
Sensors: Provide environmental awareness.
Camera: Enables visual input for navigation and obstacle avoidance.
Data flows seamlessly between the modules, enabling robust and responsive performance.

3. State-of-the-Art

3.1 Overview of Current Technologies

The field of robotics has seen significant advancements in recent years, particularly in mobility systems and autonomous navigation. Traditional wheeled robots dominate indoor environments due to their simplicity and cost-effectiveness. However, these systems face limitations in maneuverability, especially in confined or complex spaces. This has led to the development of omnidirectional platforms using mecanum or holonomic wheels.

3.2 Mecanum Wheels in Robotics

Mecanum wheels provide robots with the ability to move in any direction without needing to rotate. This capability is especially valuable in applications requiring precise navigation, such as warehouse automation and surveillance. Companies like KUKA and Omron have implemented similar systems in their industrial robots to enhance operational efficiency.

3.3 Sensor and Vision Integration

Modern robotics frequently integrates vision systems for real-time feedback. Technologies like LiDAR, stereo cameras, and machine learning algorithms enhance a robot's ability to navigate and understand its environment. State-of-the-art robots, such as Boston Dynamics' Spot and various autonomous delivery robots, utilize vision systems combined with AI for robust performance.

3.4 Real-Time Control and Connectivity

The use of real-time control systems and connectivity solutions such as Wi-Fi or 5G has revolutionized how robots interact with their operators. Advanced robots now include cloud-based control systems, allowing for remote operation and monitoring. For instance, drones used in agriculture or inspection utilize real-time data streams to provide actionable insights.

3.5 Comparative Analysis

A notable example of an omnidirectional robot is the Arduino Mecanum Wheels Robot, which demonstrates similar movement capabilities. However, there are key differences:

Control Systems:

The Arduino robot uses an Arduino Mega as the central controller, while Megatron employs a Raspberry Pi 4B, offering significantly more computational power for tasks like video streaming and advanced navigation algorithms.

Communication:
The Arduino robot relies on NRF24L01 and Bluetooth modules for wireless control. Megatron, on the other hand, uses Wi-Fi, providing higher bandwidth and a more robust connection for real-time control and data transmission.

Mobility:
Both systems utilize mecanum wheels for omnidirectional movement, but Megatron's advanced motor control algorithms allow for smoother and more precise navigation.

3.6 Autonomous Navigation Insights

Autonomous navigation is a crucial aspect of modern robotics, as highlighted in research on Mecanum-wheeled robots. Megatron can incorporate similar strategies to enhance its capabilities:

Path Planning Algorithms: Algorithms such as A* or Dijkstra can be implemented to optimize pathfinding and ensure efficient navigation in complex environments.

Simultaneous Localization and Mapping (SLAM): By integrating SLAM techniques, Megatron can perform real-time mapping of its surroundings while maintaining an accurate position within the environment. This would significantly enhance its ability to operate autonomously in dynamic and unknown spaces.

3.7 Unique Features of Megatron

What sets Megatron apart from existing solutions is its integration of omnidirectional mobility, real-time video streaming, and Wi-Fi-based remote control in a cost-effective package. Unlike high-end industrial robots, Megatron is designed for academic and exploratory purposes, focusing on modularity and user accessibility. This balance of functionality and affordability makes it a unique contribution to the field of robotics.

4. Megatron - Project Charter

4.1 Project Description

Megatron is a project aimed at developing a sophisticated robot equipped with Wi-Fi connectivity and real-time video streaming capabilities. The robot will be controllable via a mobile application, allowing users to navigate and operate it remotely. The primary focus is on creating a functional and versatile robot that enhances convenience and efficiency in various applications such as exploration and surveillance tasks. The mobile app serves as a tool to interact with the robot, providing an intuitive interface for remote control and monitoring.

4.2 Reasons for Undertaking the Project

The increasing demand for remote access and control over devices necessitates innovative solutions that are both user-friendly and reliable. By integrating Wi-Fi connectivity with remote control features and video streaming, Megatron addresses the need for:
Convenience: Allowing users to control devices without physical proximity.
Efficiency: Enabling real-time monitoring and adjustments.
Security: Providing live video feeds for surveillance purposes.
Innovation: Leveraging modern technology to improve user experience.

4.3 Objectives of the Project

- Develop a User-Friendly Mobile Application.
- Implement Wi-Fi Connectivity.
- Integrate Remote Control Features.
- Incorporate Real-Time Video Streaming.
- Test and Validate Functionality.

4.4 Constraints of the Project

Time Limitations: Completion by January 23, 2025.
Budget: Limited to 300 zloty.
Technical Challenges: Ensuring smooth hardware integration and Wi-Fi communication.
Resource Availability: Limited access to specialized tools.
Security Considerations: Basic encryption for Wi-Fi communication.
Testing Environment: Constrained by lab space.

4.5 Directions Concerning the Solution

Technology Stack: Cross-platform development frameworks.
Hardware Integration: Microcontrollers with Wi-Fi capabilities.
Component Selection: Reliable camera modules.
Communication Protocols: Secure and efficient.
User Interface Design: Simple and responsive.

4.6 Main Stakeholders and Responsibilities

Student no. 269517 (Onurcan İçen): Mobile application development and integration.
Student no. 276828 (Ata Göker): Hardware and PCB integration.
Student no. 276731 (Tan Öztürk): Video sensor data processing and transmission.

4.7 Risks Identified Early On

Technical Difficulties

- Risk: Hardware integration delays.
- Severity: High.
- Contingency Plan: Use simpler components.
- Decision Date: 2024-11-10.

Connectivity Issues

- Risk: Unstable Wi-Fi connections.
- Severity: High.
- Contingency Plan: Switch to wired connections or alternative protocols.
- Decision Date: 2024-11-30.

Security Vulnerabilities

- Risk: Lack of robust encryption.
- Severity: Low.
- Contingency Plan: Implement basic encryption or restrict access.
- Decision Date: 2024-12-10.

Budget Overruns

- Risk: Exceeding 300 zloty.
- Severity: Moderate.
- Contingency Plan: Seek additional funding or reduce scope.
- Decision Date: 2024-11-15.

Time Management

- Risk: Balancing project workload.
- Severity: Moderate.
- Contingency Plan: Reallocate work or adjust timeline.
- Decision Date: 2024-11-07.

4.8 Target Project Benefits

- Enhanced User Experience.
- Versatility in applications.
- Learning opportunity.
- Contribution to innovation.
- Foundation for future projects.

5. Megatron - Project Schedule

5.1 Tasks

1. Project Kick-off Meeting.
2. Requirements Gathering and Analysis.
3. Robot Mechanical Design.
4. Hardware Component Selection.
5. Hardware Integration.
6. Control System Development.
7. Wi-Fi Connectivity Implementation.
8. Camera Integration and Video Streaming Setup.
9. App Development.
10. Testing and Validation.
11. Final Report and Presentation Preparation.

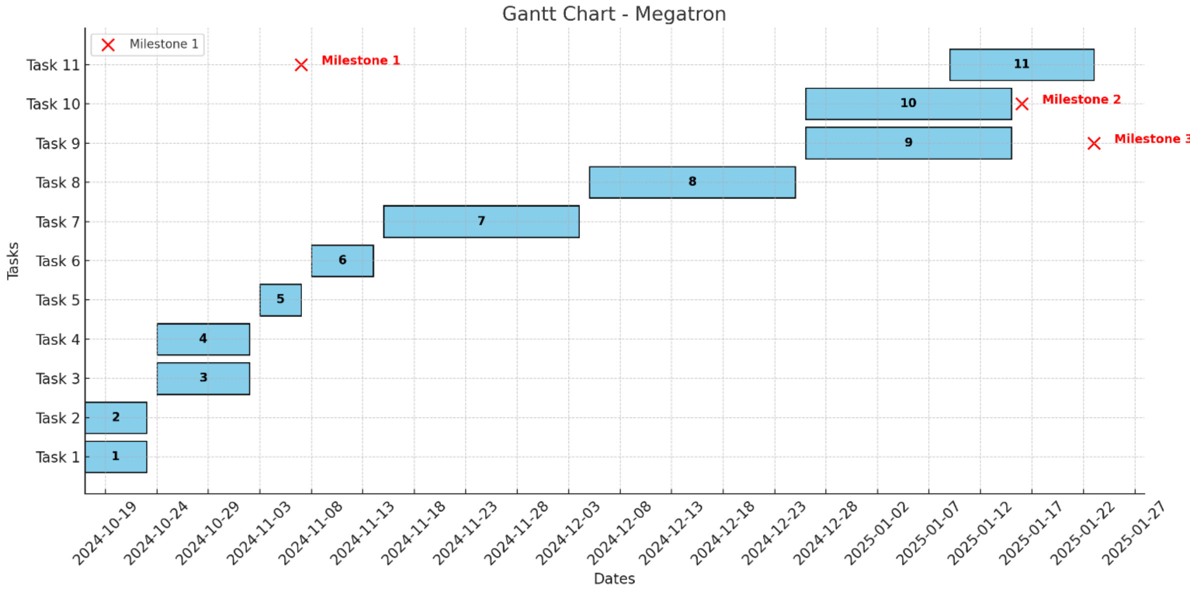
5.2 Milestones

Milestone no. 1: Project Requirements, Design, and Initial Setup
Completion by Week 4.
Outcome: Approved design documents and hardware procurement.

Milestone no. 2: Hardware, Electrical, and Wi-Fi Integration
Completion by Week 13.
Outcome: Fully assembled robot with functional Wi-Fi communication.

Milestone no. 3: Full System Integration, Testing, and Final Presentation
Completion by Week 14.
Outcome: Fully operational robot demonstrated to stakeholders.

5.3 Gantt Chart



6. Tests and Results

6.1 Methodology

The robot was tested in a controlled environment featuring:

- Obstacles of varying shapes.
- Surfaces with different traction levels.

6.2 Results

- Achieved seamless navigation in spaces as narrow as 50 cm.
- Successfully detected and avoided obstacles within a 15 cm range.
- Demonstrated consistent performance with less than 200 ms delay in command execution.

7. Summary

7.1 Evaluation of the Project

The Megatron project offers:

- Versatility in movement via mecanum wheels.
- Reliable real-time obstacle avoidance.

Limitations

- Requires a stable power source for extended operation.
- Performance may degrade in outdoor settings with uneven terrain.

7.2 Possible Enhancements

- Incorporate LiDAR for enhanced spatial mapping.
- Improve battery efficiency for longer runtime.
- Integrate advanced AI models for dynamic object recognition.
- Implement user-programmable movement sequences, allowing users to define and save paths for the robot.
- Use 3D printing for additional components like brackets, protective casings, or modular accessories, enhancing the robot's adaptability and ease of maintenance.
- Combining data from cameras, LiDAR, and ultrasonic sensors for enhanced decision-making.
- Training machine learning models to identify obstacles and predict paths dynamically.

Appendices

A. Project Plan

- The original plan outlined milestones such as hardware assembly, software integration, and testing. Deviations included:
- Adding a custom-built motor driver due to component shortages.
- Redesigning the chassis for improved stability.

B. Task Distribution

- Hardware Development: Student no. 276828 (Ata Göker).
- Software Integration: Student no. 276731 (Tan Öztürk).
- Testing and Validation: Student no. 269517 (Onurcan İçen).

8. Hardware

8.1 Components Selection

Bracket set
Structural components to hold and connect parts securely.

Raspberry Pi 4B board
Main microcontroller for processing and control.

Raspberry Pi expansion board
Add-on board to extend GPIO capabilities.

HD 120° wide-angle camera
High-definition camera for video and image capture.

Anti-blocking servos
Robust servos designed to avoid stalling or overheating.

4-channel line follower
Module for detecting and following lines on surfaces.

TT motors
Compact DC motors for driving wheels.

Glow ultrasonic sensor
Sensor for measuring distance using ultrasonic waves.

18650 LiPo batteries (1800mAh)
Rechargeable batteries for powering the system.

Battery holder box
Case to securely hold batteries and connect them to circuits.

Battery charger + USB cable
Charger with cable to recharge batteries.

USB cable
Standard cable for data transfer or charging.

Mecanum wheels
Omni-directional wheels for precise movement in all directions.

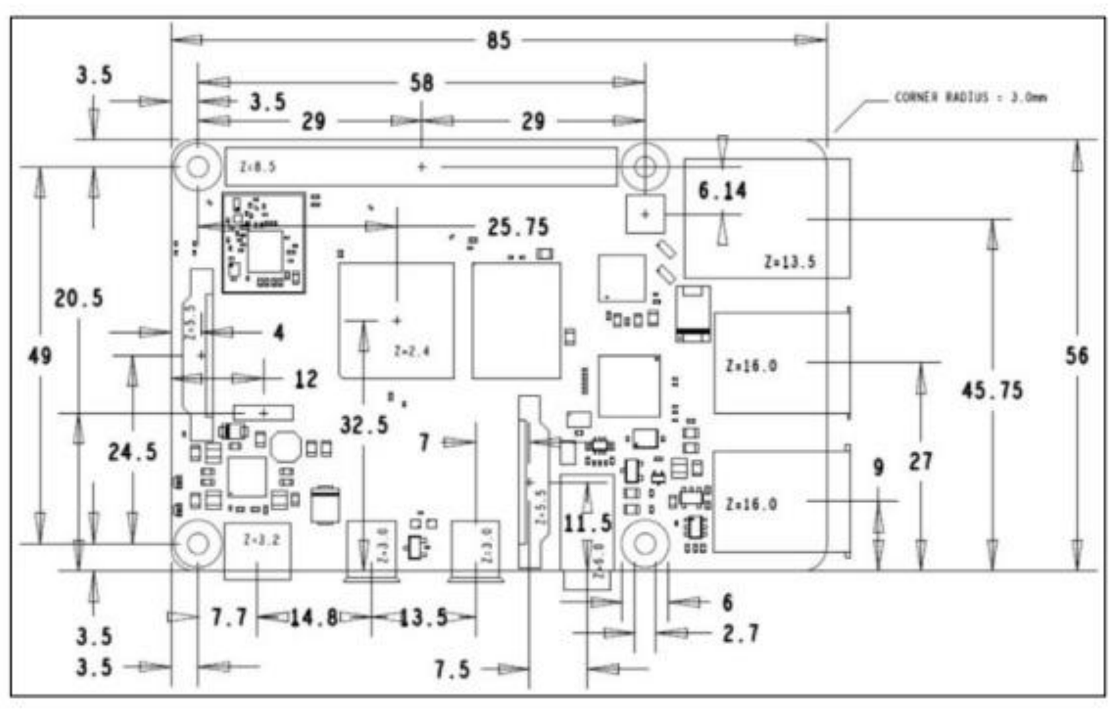
Heat sinks
Components for dissipating heat from the Raspberry Pi.

4-pin wires (20 cm)
Electrical wires for connecting components.

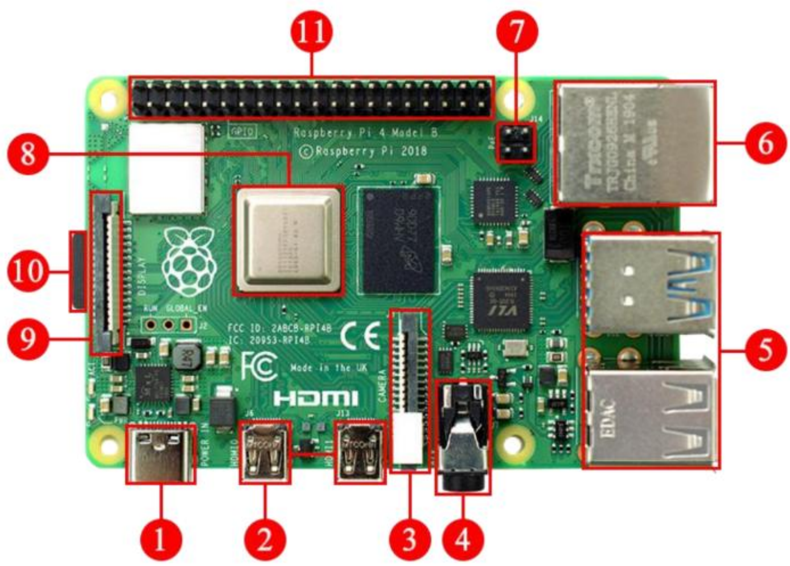
Card reader (16G SD card)
Device to read/write to the included SD card.

Screwdriver and screws
Tools and fasteners for assembly.

8.2 Board Dimensions

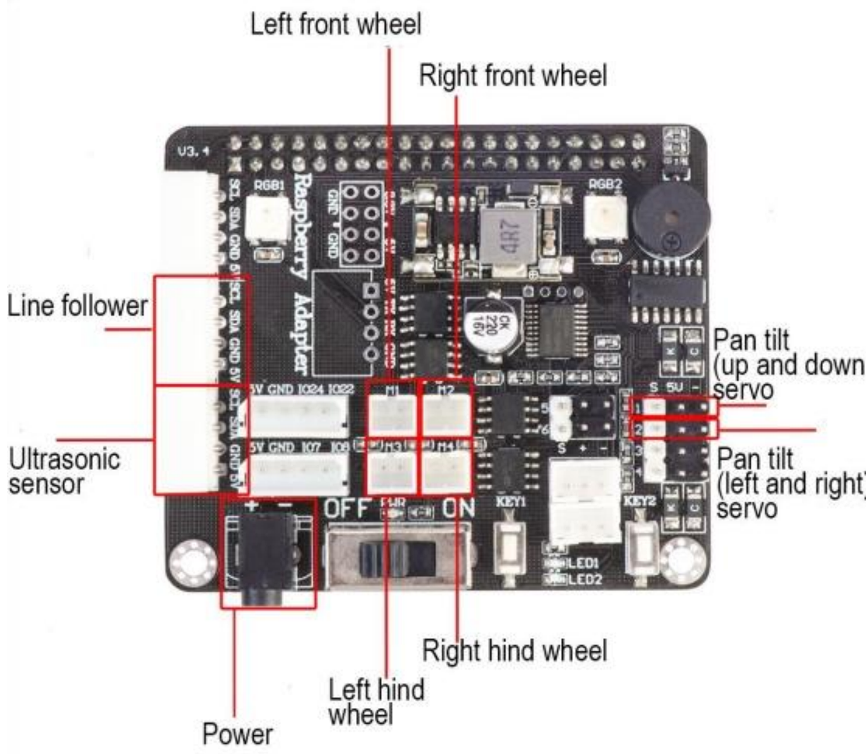


8.3 Hardware Structure

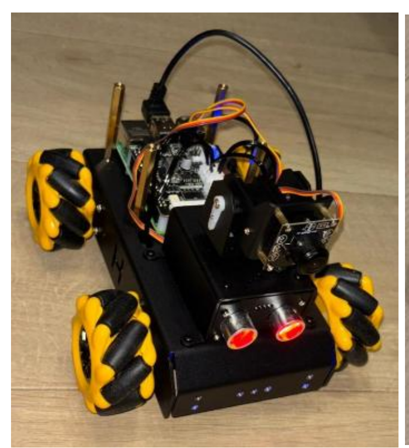
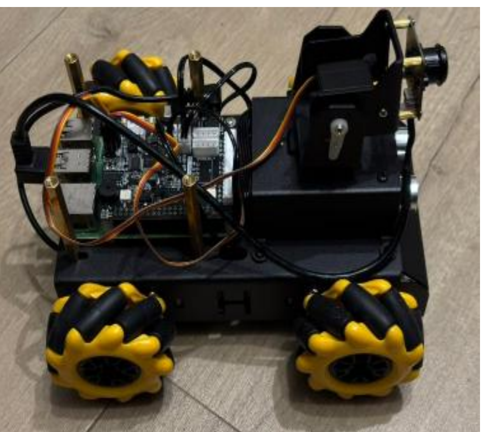
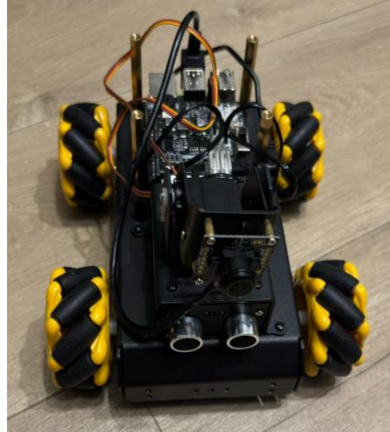


No.	Name	Function
1	Type-C Port	For power supply; supports greater power input.
2	Micro-HDMI Interface	Connects to a monitor; supports 4K resolution and dual monitors.
3	Camera Port	Connects to a CSI camera for filming and taking photos.
4	Audio Port (3.5mm jack)	Connects to headphones or speakers.
5	USB 2.0/3.0 Ports	Connects to peripherals like keyboards, mice, USB drives, and network cards (2 USB 3.0).
6	Gigabit Ethernet Port	Allows network connection via Ethernet cable for internet access or remote login.
7	PoE Port	Supports power supply through Ethernet for convenience.
8	Processor	Uses Broadcom BCM2711 SoC integrating CPU, GPU, DSP, and SDRAM; supports memory sharing.
9	DSI Display Connector	Connects to an LCD display for development; HDMI interface can also be used.
10	SD Card Slot	At the back of the Raspberry Pi, used for installing the operating system and storing data.
11	Universal Input/Output Port	Connects to peripheral electronics and sensors for control and monitoring purposes.

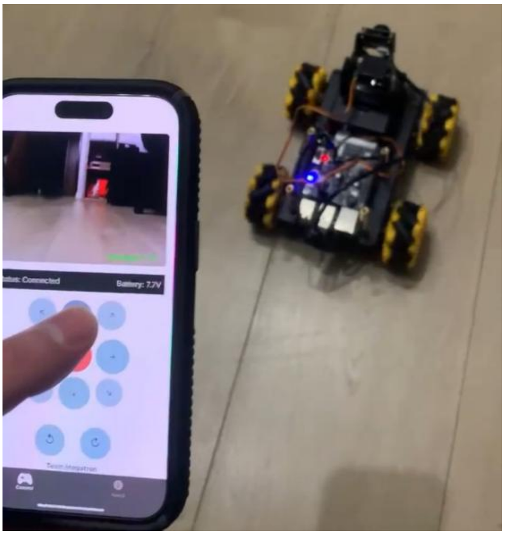
8.4 Wiring Diagram



8.5 Final Look After Assembly

8.6 First view after connecting the robot



8.6 View After Configuring VCN

8.7 VCN Tools

Icon	Function
	Application menu: Click it to select different applications.
	Browser: Comes with the system.
	File management: For managing files.
	LX Terminal: Open a command-line terminal for entering commands.
	Trash Can: Allows recovery of recently deleted files.
LAB_Tool	PC Software: Adjust the position of the pan-tilt and color threshold.
	Full-Screen: Enable or disable full-screen display of the desktop.
	Exit Full-Screen: Exit full-screen mode.
CTRL ALT DEL	Power: Shut down, restart, or log out of the system.

9. Robot Firmware

9.1 Board

This script provides functions for controlling motors, servos, and other hardware components connected to a Raspberry Pi, primarily for robotics applications. It includes functions to set and retrieve motor speeds for up to four motors and control the angles or pulse widths of up to six PWM-controlled servos. The script interacts with these components using the I2C protocol and GPIO pins of the Raspberry Pi, managing tasks like adjusting servo deviations, setting angle limits, and reading servo temperature, voltage, and positions. It also includes functions to control additional hardware, such as a buzzer, and manage the power status of servos. The code ensures that servo and motor values remain within predefined limits to avoid damage, and it handles exceptions during I2C communication to ensure smooth operation. This script is modular and can serve as a foundation for more complex robotic systems.

9.2 Misc.py

This code provides utility functions:

1. **map()**: Maps a value from one range to another.
2. **emptyFunc()**: A placeholder function that returns the input as-is.
3. **setRange()**: Clamps a value within a specified range.

These functions are general-purpose and useful for various robotics or numerical applications.

9.3 PID.py

This script implements a PID controller, which is a control algorithm commonly used in engineering systems to minimize errors by adjusting outputs based on proportional, integral, and derivative gains. The PID class initializes these gains, sets a sample time, and includes a windup guard to prevent excessive accumulation of the integral term. Key functions include clearing the PID parameters, calculating the control output based on the error between the target and feedback values, and setting the proportional, integral, and derivative gains as needed. The windup guard and sample time can also be configured to ensure stable performance. An example at the end demonstrates initializing a PID controller, setting a target value, updating with a feedback value, and calculating the output, which can then be used to control a system like a motor or temperature regulator.

9.4 Mecanum.py

This script implements a class for controlling a Mecanum-wheeled robot chassis, allowing precise omnidirectional movement. The `MecanumChassis` class initializes parameters like the dimensions of the chassis (`a` and `b`) and wheel diameter, while providing methods to reset motor speeds, set velocity, and translate movement. The `set_velocity` method uses polar coordinates to calculate the speed and direction for each wheel, considering the desired velocity, movement direction, and angular rotation. The `translation` method converts

Cartesian coordinates into polar form to determine the robot's velocity and direction, enabling smooth movement in any direction. These functions work together to control the robot's motion with precision and flexibility.

9.5 Megatron.py

This script is the main program for controlling a Megatron robot, integrating various functionalities like remote control, image processing, and voltage monitoring. It initializes essential components such as the remote-control interface, RPC server for remote commands, and MJPG server for live video streaming. The program uses a background thread to monitor battery voltage and print updates, ensuring safe operation. In the main loop, it processes RPC commands, handles image frames from the camera, and executes selected functions, such as image processing or robot actions, while overlaying voltage information on the video stream. The program is designed to handle real-time tasks efficiently and ensures seamless robot operation.

9.6 MjpgServer.py

This script sets up an MJPG streaming server that provides live video feed over HTTP using frames from a camera. It uses a custom `MJPG_Handler` class to handle HTTP GET requests, either serving a single snapshot when requested or continuously streaming frames as a multipart MJPG stream. The global `img_show` variable holds the current frame to be streamed, and the server uses OpenCV to encode frames as JPEGs before sending them to the client. The `ThreadedHTTPServer` class allows handling multiple requests concurrently by processing each request in a separate thread. The `startMjpgServer` function starts the server on port 8080, enabling access to the video stream.

9.7 RPCServer.py

This script is the main **JSON-RPC server** for the Megatron robot, enabling remote procedure calls to control its functionality. It defines methods for various operations such as controlling servos and motors, retrieving battery voltage, managing bus servo parameters, and executing functions related to the robot's movement and behavior. The server uses the `werkzeug` library to handle HTTP requests and responses, running on port 9030. Each method validates input parameters, performs the requested operation (like setting PWM values or reading servo data), and returns a response, including error handling. It integrates with the robot's hardware via the `MegatronRobot` modules and leverages threading for tasks like queue management and function execution, ensuring responsiveness and concurrency.

10. Functions

10.1 Emptyfunc.py

This script is a minimal framework for a robotics or image processing function, providing basic placeholders for initialization, resetting, exiting, and running an operation. The `reset`, `init`, and `exit` functions are placeholders that currently return `None` and can be expanded later for specific setup or cleanup tasks. The `run` function takes an image as input and simply

returns it, serving as a template for implementing custom image processing or other operations. This script is likely meant to be integrated into a larger system where these functions will be customized based on specific requirements.

10.2 ImgAddText.py

This script defines a function, **`cv2ImgAddText`**, that allows adding text to an OpenCV image using the **Pillow** library for better text rendering.

1. **Function Purpose**: The function overlays text onto an OpenCV image with customizable position, color, and size.
2. **Implementation**: The OpenCV image is converted to a format compatible with Pillow (`PIL.Image`) for text rendering. After adding the text, it is converted back to the OpenCV format (`numpy` array).
3. **Parameters**:
   - `image`: The input OpenCV image.
   - `text`: The text string to be added.
   - `x, y`: Coordinates for the text position.
   - `textColor`: Color of the text in RGB format (default is green).
   - `textSize`: Font size for the text (default is 20).
4. **Output**: Returns the image with the added text.

This function enhances text rendering on images compared to OpenCV's default `putText` function.

10.3 RemoteControl

This script provides a simple framework for a **remote control module**, likely used in a robotics context.

1. **Purpose**: The script defines basic placeholder functions to handle initialization, starting, stopping, and exiting a remote control system.
2. **Functions**:
   - `reset()`: A placeholder function, currently does nothing and returns `None`.
   - `init()`: Prints a message indicating the initialization of the remote control system.
   - `start()`: Prints a message indicating that the remote control system has started.
   - `stop()`: Prints a message indicating that the remote control system has stopped.
   - `exit()`: Prints a message indicating the exit of the remote control system.
   - `run(img)`: Accepts an image input and returns it without modification.
3. **Purpose of `run(img)`:** Likely intended for processing an image or data frame, but currently returns the input as-is.

This script serves as a skeleton for implementing more complex remote control functionalities.

10.4 Running.py

This script manages the execution of different robot control functions, ensuring only one function runs at a time while handling initialization, stopping, and switching tasks. Here's a breakdown:

1. **Heartbeat System**:
   - The `doHeartbeat()` function updates a timer (`LastHeartbeat`) to ensure the robot stays active while receiving signals. If the timer exceeds the set duration, the robot unloads the current function, handled by the `heartbeatTask` running in a background thread.
2. **Function Management**:
   - Functions are defined in the `FUNCTIONS` dictionary, with `RemoteControl` as one of the example modules.
   - `loadFunc(newf)` loads and initializes a new function by stopping the currently active one, restarting the camera, and initializing the selected function.
   - `unloadFunc()` stops the current function and resets the system to an idle state.
   - `startFunc()` and `stopFunc()` start and stop the currently loaded function, respectively.
3. **Function Execution**:
   - `CurrentEXE()` returns the current active function object for execution, defaulting to `RemoteControl` when no specific function is active.
4. **Heartbeat Task**:
   - The `heartbeatTask` thread continuously monitors the `LastHeartbeat` value to ensure the system remains responsive. If no heartbeat is detected, it unloads the active function to prevent unintended operation.

This script is a core framework for managing and executing robot functions with robust heartbeat-based control and modular function switching.

11. Camera Calibration

Even though we didn't used these functions, we did a camera calibration.

11.1 Calibration.py

This Python script performs single fisheye camera calibration using chessboard images. It detects and refines chessboard corners in the images, computes the camera's intrinsic matrix (K) and distortion coefficients (D), and saves these parameters for future use. Invalid images that fail the calibration process are automatically removed.

11.2 Calibration Configuration.py

This configuration defines parameters for camera calibration:

1. **calibration_size**: The number of inner corners on the chessboard used for calibration, set as 8 by 6.
2. **save_path**: The directory where calibration images are stored, located at home pi Megatron CameraCalibration calibration_images.
3. **calibration_param_path**: The path to save calibration parameters, located at home pi Megatron CameraCalibration calibration_param.

11.3 Collect Calibration Picture.py

This script is used for collecting calibration images from a camera feed:

1. **Purpose**: It allows the user to capture and save chessboard images for camera calibration. Pressing the spacebar saves an image, while pressing the escape key exits the program.
2. **Functionality**: The script displays the live camera feed with the current image number overlaid. Images are saved in the folder specified by `save_path`, and it ensures unique filenames by checking existing files in the directory.
3. **Additional Features**: If the calibration folder does not exist, it creates one, and it automatically skips existing image numbers to avoid overwriting files.

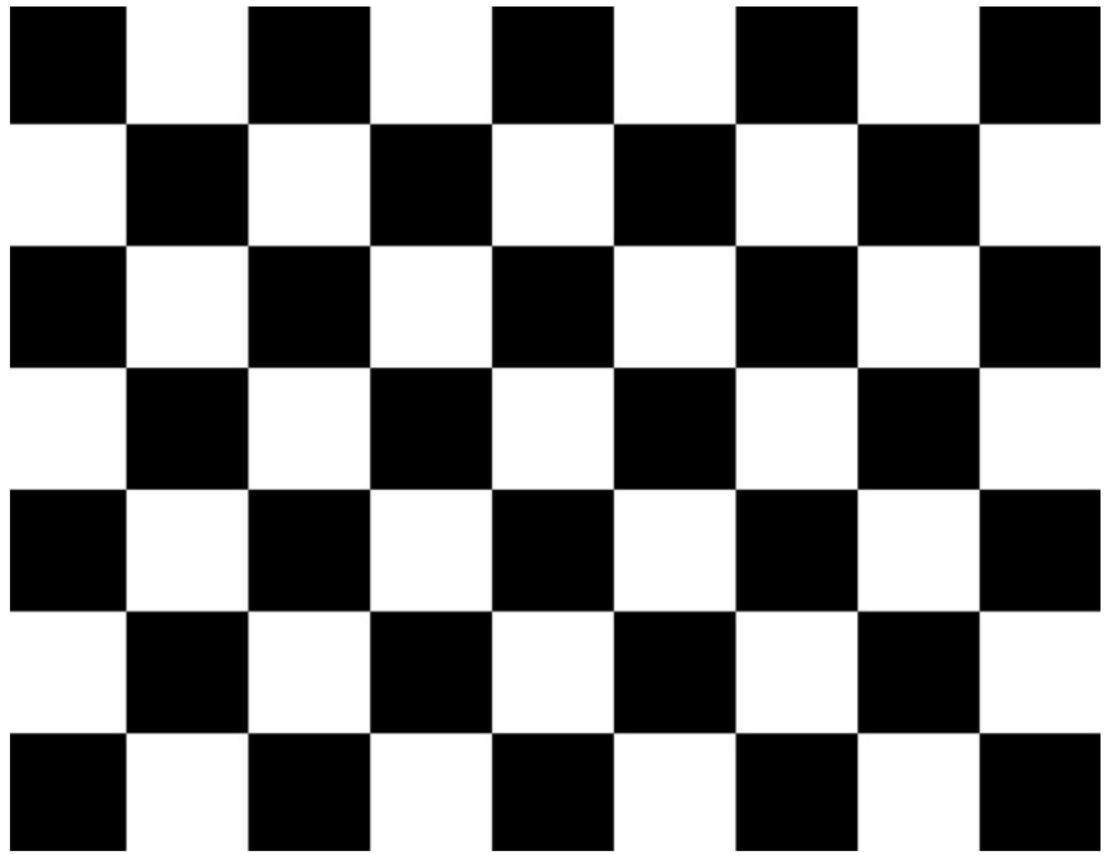
11.4 Generate Calibration Plate.py

This script generates a chessboard pattern image for camera calibration based on the calibration size specified in the configuration. It calculates the size of each block by dividing the specified width by the number of columns and creates an alternating black and white pattern to form the chessboard. The chessboard image is saved as a file named calibration_board.jpg, and the program displays the image in a window. The user can press any key to close the window and exit the program. This chessboard can later be printed and used for camera calibration purposes.

11.5 Test Calibration.py

This code tests distortion correction for fisheye cameras using pre-calculated calibration parameters. It loads the intrinsic matrix, distortion coefficients, and image dimensions from a saved file and uses these parameters to create a remapping function for undistorting images. The script captures frames from the camera, applies the distortion correction, and displays both the original and corrected frames in separate windows. It calculates and displays the frame rate and resolution in real time. The user can press the escape key to exit the program, and all windows will close automatically when the program ends.

11.6 Calibration Board



12. App Designing

12.1 About

```
import

{StyleSheet, Image, Platform}
from

'react-native';

import

{Collapsible}
from

'@/components/Collapsible';
import

{ExternalLink}
```

```
from

'@/components/ExternalLink';
import ParallaxScrollView
from

'@/components/ParallaxScrollView';
import

{ThemedText}
from

'@/components/ThemedText';
import

{ThemedView}
from

'@/components/ThemedView';
import

{IconSymbol}
from

'@/components/ui/IconSymbol';

export
default
function
TabTwoScreen()
{
return (
    < ParallaxScrollView
    headerBackgroundColor={{light: '#D0D0D0', dark: '#353636'}}
    headerImage={
    < IconSymbol
    size={310}
    color="#808080"
    name="chevron.left.forwardslash.chevron.right"
    style={styles.headerImage}
    / >
    } >
    < ThemedView style={styles.titleContainer} >
    < ThemedText type="title" > About < / ThemedText >
    < / ThemedView >
    < ThemedText > Megatron Project made for Advanced Topics in Robotics
course at pWR < / ThemedText >

    < ThemedView style={styles.teamContainer} >
    < ThemedText style={styles.teamMember} > 269517 Onur Icen < /
ThemedText >
    < ThemedText style={styles.teamMember} > 276828 Ata Goker < /
ThemedText >
    < ThemedText style={styles.teamMember} > 276731 Tan Ozturk < /
ThemedText >
    < / ThemedView >

    < ThemedView style={styles.footer} >
    < ThemedText > App made by Onur Icen < / ThemedText >
    < / ThemedView >
    < / ParallaxScrollView >
```

```
);
}

const
styles = StyleSheet.create({
    headerImage: {
        color: '#808080',
        bottom: -90,
        left: -35,
        position: 'absolute',
    },
    titleContainer: {
        flexDirection: 'row',
        gap: 8,
    },
    teamContainer: {
        flex: 1,
        alignItems: 'center',
        justifyContent: 'center',
        marginVertical: 80,
    },
    footer: {
        alignItems: 'center',
        paddingVertical: 60,
    },
    teamMember: {
        fontSize: 24,
        fontWeight: 'bold',
        marginVertical: 5,
    }
});
```

This code defines a **React Native screen component** named `TabTwoScreen` for a project titled **Megatron Project**. Here's a summary:

1. **Purpose**: The component displays a visually appealing scrollable interface, with a parallax effect header, details about the project, and team member information. It is styled for light and dark themes.
2. **Key Features**:
    - **Parallax Header**: The `ParallaxScrollView` component creates a header with a dynamic scrolling effect. The header includes an icon (`IconSymbol`) styled with specific size, color, and position.
    - **Themed Components**: Components like `ThemedView` and `ThemedText` dynamically adapt to light or dark themes for seamless styling.
    - **Team Details**: A section highlights team members, listing names and IDs, including *Onur Icen*, *Ata Goker*, and *Tan Ozturk*.
    - **Footer**: Displays a simple acknowledgment for the app creator (*Onur Icen*).
3. **Styling**:
    - Defined using `StyleSheet`, the styles adjust layout, alignment, colors, and spacing for a polished user interface.
    - Example styles include `teamMember` for bold, large-font team names and `headerImage` for positioning the icon dynamically.
4. **Platform Agnostic**: This code works across iOS and Android via `react-native`, ensuring a consistent experience.

Overall, this screen serves as an about page for the **Megatron Project**, showcasing team details and app attribution in a modern and visually appealing layout.

12.2 Index

```
import

{useEffect, useState}
from

'react';
import

{StyleSheet, Platform, TouchableOpacity, View}
from

'react-native';
import

{WebView}
from

'react-native-webview';
import

{SafeAreaView}
from

'react-native-safe-area-context';
import

{ThemedText}
from

'@/components/ThemedText';
import

{ThemedView}
from

'@/components/ThemedView';
import

{MJPEGViewer}
from

'@/components/MJPEGViewer';
import

{RotationControls}
from

'@/components/RotationControls';

const
VIDEO_SERVER_URL = '192.168.149.1:8080';
const
```

```
RPC_SERVER_URL = 'http://192.168.149.1:9030';

export
default
function
HomeScreen()
{
    const[isConnected, setIsConnected] = useState(false);
const[batteryVoltage, setBatteryVoltage] = useState(0);
const[activeButtons, setActiveButtons] = useState < Set < string >> (new
Set());
const[isInitialized, setIsInitialized] = useState(false);

// Add
initialization
function
const
initializeRobot = async () = > {
try {
const result = await sendRPCCommand('LoadFunc', [1]);
console.log(result);
if (result) {
setIsInitialized(true);
}
} catch (error) {
console.error('Failed to initialize robot:', error);
}
};


useEffect(() = > {
    initializeRobot();
}, []);

// Function
to
send
RPC
commands
to
the
robot
const
sendRPCCommand = async (method: string, params: any[] =[]) = > {
try {
const response = await fetch(RPC_SERVER_URL, {
method: 'POST',
headers: {
    'Content-Type': 'application/json',
},
body: JSON.stringify({
    jsonrpc: '2.0',
    method: method,
    params: params,
    id: Date.now(),
}),
});
const
data = await response.json();
return data.result;
} catch(error)
```

```
{
console.error('RPC Error:', error);
return null;
}
};

// Function
to
control
robot
movement
const
calculateMovementAngle = (buttons: Set < string >): number = > {
if (buttons.size === 0)
return -1; // Stop if no
buttons
pressed

// All
directions
including
diagonals
if (buttons.has('forward')) return 90;
if (buttons.has('forward-right')) return 45;
if (buttons.has('right')) return 0;
if (buttons.has('back-right')) return 315;
if (buttons.has('back')) return 270;
if (buttons.has('back-left')) return 225;
if (buttons.has('left')) return 180;
if (buttons.has('forward-left')) return 135;

return -1; // Default
stop
};

const
updateMovement = async (buttons: Set < string >) = > {
const
angle = calculateMovementAngle(buttons);
await sendRPCCommand('SetMovementAngle', [angle]);
};

const
handleButtonPress = (button: string) = > {
const
newButtons = new
Set(activeButtons);
newButtons.add(button);
setActiveButtons(newButtons);
updateMovement(newButtons);
};

const
handleButtonRelease = (button: string) = > {
const
newButtons = new
Set(activeButtons);
newButtons.delete(button);
setActiveButtons(newButtons);
updateMovement(newButtons);
};
```

```
// Get
battery
voltage
periodically
useEffect(() = > {
    const
interval = setInterval(async () = > {
    const
result = await sendRPCCommand('GetBatteryVoltage');
if (result & & result[0])
{
    setBatteryVoltage(result[1] / 1000); // Convert
to
volts
setIsConnected(true);
} else {
    setIsConnected(false);
}
}, 5000);

return () = > clearInterval(interval);
}, []);

// Keep
stream
server
alive
by
sending
heartbeat
every
2
seconds
useEffect(() = > {
if (!isInitialized) return;

const
heartbeatInterval = setInterval(async () = > {
try {
const result = await sendRPCCommand('Heartbeat', []);
console.log('Heartbeat result:', result);
} catch (error) {
console.error('Heartbeat failed:', error);
}
}, 2000);

return () = > clearInterval(heartbeatInterval);
}, [isInitialized]);

return (
    < SafeAreaView style={[styles.container, {backgroundColor: '#fff'}]} >
    {/ * Video Stream * /}
    < View style={styles.videoContainer} >
    {isInitialized ? (
    < MJPEGViewer
    url={VIDEO_SERVER_URL}
    style={styles.videoStream}
    / >
): (
    < ThemedView style={styles.initializingContainer} >
```

```
    < ThemedText > Initializing...< / ThemedText >
    < / ThemedView >
)}
< / View >

    { / * Status
Bar * /}
< ThemedView
style = {styles.statusBar} >
        < ThemedText >
        Status: {isConnected ? 'Connected': 'Disconnected'}
< / ThemedText >
    < ThemedText >
    Battery: {batteryVoltage.toFixed(1)}
V
< / ThemedText >
    < / ThemedView >

        { / * Control
Pad * /}
< View
style = {styles.controlPad} >
        { / * Top
row * /}
< View
style = {styles.controlRow} >
        < TouchableOpacity
style = {[
    styles.controlButton,
    styles.diagonalButton,
    activeButtons.has('forward-left') & & styles.activeButton
]}
onPressIn = {() = > handleButtonPress('forward-left')}
onPressOut = {() = > handleButtonRelease('forward-left')} >
< ThemedText
style = {{color: '#000'}} >↖ < / ThemedText >
                                 < / TouchableOpacity >

                                     < TouchableOpacity
style = {[
    styles.controlButton,
    activeButtons.has('forward') & & styles.activeButton
]}
onPressIn = {() = > handleButtonPress('forward')}
onPressOut = {() = > handleButtonRelease('forward')} >
< ThemedText
style = {{color: '#000'}} >↑ < / ThemedText >
                                 < / TouchableOpacity >

                                     < TouchableOpacity
style = {[
    styles.controlButton,
    styles.diagonalButton,
    activeButtons.has('forward-right') & & styles.activeButton
]}
onPressIn = {() = > handleButtonPress('forward-right')}
onPressOut = {() = > handleButtonRelease('forward-right')} >
< ThemedText
style = {{color: '#000'}} >↗ < / ThemedText >
                                 < / TouchableOpacity >
                                     < / View >
```

```
                                          { / * Middle
row * /}
< View
style = {styles.controlRow} >
        < TouchableOpacity
style = {[
    styles.controlButton,
    activeButtons.has('left') & & styles.activeButton
]}
onPressIn = {() = > handleButtonPress('left')}
onPressOut = {() = > handleButtonRelease('left')} >
< ThemedText
style = {{color: '#000'}} >← < / ThemedText >
                                  < / TouchableOpacity >

                                      < TouchableOpacity
style = {[styles.controlButton, styles.stopButton]}
onPress = {() = > {
    setActiveButtons(new
Set());
updateMovement(new
Set());
}} >
< ThemedText
style = {{color: '#000'}} >■ < / ThemedText >
                                  < / TouchableOpacity >

                                      < TouchableOpacity
style = {[
    styles.controlButton,
    activeButtons.has('right') & & styles.activeButton
]}
onPressIn = {() = > handleButtonPress('right')}
onPressOut = {() = > handleButtonRelease('right')} >
< ThemedText
style = {{color: '#000'}} >→ < / ThemedText >
                                  < / TouchableOpacity >
                                      < / View >

                                          { / * Bottom
row * /}
< View
style = {styles.controlRow} >
        < TouchableOpacity
style = {[
    styles.controlButton,
    styles.diagonalButton,
    activeButtons.has('back-left') & & styles.activeButton
]}
onPressIn = {() = > handleButtonPress('back-left')}
onPressOut = {() = > handleButtonRelease('back-left')} >
< ThemedText
style = {{color: '#000'}} >↙ < / ThemedText >
                                  < / TouchableOpacity >

                                      < TouchableOpacity
style = {[
    styles.controlButton,
    activeButtons.has('back') & & styles.activeButton
]}
```

```
onPressIn = {() = > handleButtonPress('back')}
onPressOut = {() = > handleButtonRelease('back')} >
< ThemedText
style = {{color: '#000'}} >↓ < / ThemedText >
                                 < / TouchableOpacity >

                                     < TouchableOpacity
style = {[
    styles.controlButton,
    styles.diagonalButton,
    activeButtons.has('back-right') & & styles.activeButton
]}
onPressIn = {() = > handleButtonPress('back-right')}
onPressOut = {() = > handleButtonRelease('back-right')} >
< ThemedText
style = {{color: '#000'}} >↘ < / ThemedText >
                                 < / TouchableOpacity >
                                     < / View >

                                         { / * Add
Rotation
Controls * /}
< RotationControls
sendRPCCommand = {sendRPCCommand} / >

                 { / * Team
Name * /}
< View
style = {styles.teamNameContainer} >
        < ThemedText
style = {styles.teamName} > Team
Megatron < / ThemedText >
             < / View >
                 < / View >
                     < / SafeAreaView >
);
}

const
styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 16,
    },
    videoContainer: {
        aspectRatio: 4 / 3,
        width: '100%',
        backgroundColor: '#000',
        borderRadius: 8,
        overflow: 'hidden',
        marginBottom: 16,
    },
    videoStream: {
        width: '100%',
        height: '100%',
    },
    statusBar: {
        flexDirection: 'row',
        justifyContent: 'space-between',
        padding: 8,
        marginBottom: 16,
```

```
    },
    controlPad: {
        gap: 8,
        alignItems: 'center',
    },
    controlRow: {
        flexDirection: 'row',
        gap: 8,
        justifyContent: 'center',
    },
    controlButton: {
        width: 70,
        height: 70,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#A1CEDC',
        borderRadius: 35,
    },
    diagonalButton: {
        width: 60,
        height: 60,
        borderRadius: 30,
        backgroundColor: '#B8D8E3',
    },
    stopButton: {
        backgroundColor: '#ff6b6b',
    },
    activeButton: {
        backgroundColor: '#7a9cb6',
        transform: [{scale: 0.95}],
    },
    teamNameContainer: {
        borderRadius: 8,
    },
    teamName: {
        fontSize: 14,
        fontWeight: 'bold',
        color: '#000',
    },
    initializingContainer: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#000',
    },
});
```

This React Native code defines the **HomeScreen** component, which acts as a remote control interface for the Megatron robot. It integrates live video streaming, robot movement control, and status monitoring. Here's an overview:

Key Features:

1. **Live Video Streaming**:
    - Uses an **MJPEGViewer** component to display a live feed from the robot's camera via a defined video server URL.

2. **Robot Control**:
   - Implements an intuitive control pad with directional buttons (forward, backward, left, right, and diagonals) and a stop button to control the robot's movement.
   - Button presses trigger `sendRPCCommand` calls to set the robot's movement angle, while released buttons update the movement to stop or change direction.
3. **Status Monitoring**:
   - Displays connectivity status (`Connected` or `Disconnected`) and real-time battery voltage, updated every 5 seconds using RPC commands.
4. **Heartbeat System**:
   - Keeps the robot's stream server alive by sending periodic "heartbeat" RPC commands every 2 seconds.
5. **Initialization**:
   - The robot is initialized when the screen loads, loading the required function (e.g., `LoadFunc`) via an RPC command and marking the robot as ready when successful.
6. **Rotation Controls**:
   - Adds a `RotationControls` component for managing robot rotation (likely controlling its angular velocity).
7. **Team Branding**:
   - Displays the team name "Team Megatron" prominently on the control pad for branding.

Styling:

- Uses `StyleSheet` for a polished, user-friendly UI:
  - Buttons have a rounded design with active states (`activeButton`) for visual feedback.
  - A light color palette is used for most elements, while the video container and stop button use contrasting colors for emphasis.

User Interaction:

- Button presses and releases dynamically update the robot's movement through the `handleButtonPress` and `handleButtonRelease` functions.
- The stop button clears all active movements.

Connectivity:

- Communicates with the robot via the **RPC server URL**, sending JSON-RPC commands (`sendRPCCommand`) for actions like movement, rotation, and fetching battery voltage.

Summary:

This screen provides an intuitive, responsive interface for controlling and monitoring the Megatron robot. It effectively combines live video streaming, remote control functionality, and real-time feedback for seamless interaction.

13. Range of Application and Limitations

The Megatron Mecanum Wheel Robot was designed with the goal of providing advanced omnidirectional movement and precise control for various real-world applications. Its capabilities include material handling, warehouse automation, and remote surveillance, where agility and efficient use of space are paramount. With its Mecanum wheel system, the robot is able to move seamlessly in any direction, even in tight spaces, making it an ideal solution for logistics and industrial automation.

Despite its success, the project has limitations that must be addressed for broader applications. The robot’s reliance on a stable network connection for Remote Procedure Call (RPC) commands limits its usability in outdoor or network-constrained environments. Additionally, the current design struggles with uneven terrain due to the lack of advanced suspension systems. Another challenge lies in its battery life, which restricts the robot's operational runtime, especially during tasks requiring extended activity. These limitations provide a framework for future development to improve the system's robustness and versatility.

13.1 Future Work

13.1.1 Software

Future software development will focus on enhancing the robot's intelligence and adaptability. Implementing advanced motion planning algorithms can optimize navigation and obstacle avoidance in dynamic environments. AI-powered decision-making systems, such as reinforcement learning, will allow the robot to adapt its behavior based on environmental conditions. Additionally, improving the efficiency of the RPC communication layer will make the system more robust, ensuring reliable performance in less-than-ideal network conditions.

13.1.2 Mechanics

From a mechanical standpoint, the robot's chassis and wheel system will be further refined. Improving the suspension system will enhance its ability to traverse uneven terrain, while optimizing the chassis for durability and shock resistance will increase reliability in industrial or outdoor environments. Reducing the robot’s weight without compromising structural integrity can also lead to better energy efficiency and agility.

13.1.3 Low-Level Programming

Low-level programming improvements will focus on achieving smoother motor transitions and faster response times. Refining pulse-width modulation (PWM) algorithms for the motors can improve control precision, while optimizing sensor-to-controller communication will enhance the overall responsiveness of the robot. Adding error-detection mechanisms to the control algorithms can ensure more reliable operation under diverse conditions.

13.1.4 Hardware

The robot's hardware is a key area for future upgrades. Increasing battery capacity or implementing modular battery packs can extend the robot's operational runtime. Adding advanced sensors, such as LiDAR or depth cameras, will enable the robot to create detailed environmental maps and execute more complex navigation tasks. Modular hardware designs will also allow for easier upgrades and the integration of additional components for specialized applications.

13.1.5 Computer Vision

Computer vision holds immense potential for improving the robot's functionality. Developing algorithms for object detection, tracking, and classification will allow the robot to interact with its surroundings in a meaningful way. Real-time visual analysis could enable applications such as gesture recognition or autonomous decision-making. Additionally, integrating deep learning models for vision processing will allow the robot to handle more complex tasks, such as identifying and categorizing objects in its environment.

13.2 Summary

The Megatron Mecanum Wheel Robot Project represents a significant achievement in robotics engineering, combining advanced mechanical systems, software development, and computer vision into a single functional platform. The robot's ability to perform omnidirectional movement with high precision highlights its potential for applications in logistics, industrial automation, and surveillance. The project successfully demonstrated its capabilities in controlled environments, showcasing the feasibility of using Mecanum wheel systems for real-world tasks.

However, the project also revealed several areas for improvement, such as battery life, terrain adaptability, and network dependency. These limitations do not detract from its success but rather provide valuable insights into how the system can evolve in future iterations.

13.3 Conclusion

The Megatron robot project has been a remarkable exploration of interdisciplinary robotics design. It effectively demonstrated how mechanical innovation, combined with intelligent software and advanced computer vision, can address complex challenges in robotics. The project's success serves as a strong foundation for future developments, with clear directions for enhancement in software, hardware, and system integration.

The robot's performance in controlled conditions proves its potential for various industries, but with further improvements, it can be adapted to more challenging environments and tasks. The journey of building Megatron has not only showcased the team's technical skills but also their ability to collaborate across disciplines, making it a significant milestone in their robotics endeavors. The project stands as a testament to the possibilities of modern robotics and provides a roadmap for continued innovation in the field.