Wi-Fi Controlled Mobile Robot with Camera

Authors: Antoni Bajor, Filip Izworski

6.02.2025

Contents

1 Introduction

This document describes the development of a Wi-Fi-controlled mobile robot equipped with a camera for real-time video streaming. The project encompasses both the physical construction of the robot and the development of software to ensure its functionality. The microcontroller used in this project is ESP32-CAM equipped with a camera and a Wi-Fi module. This documentation provides an overview of the project assumptions, key functionalities, hardware assembly, software design and a testing results.

2 Project assumptions

This section outlines the key assumptions made during the planning of the project. These assumptions define the scope of the project, technical constraints and the contribution of each team member.

1) The robot uses the ESP32-CAM module for real-time video streaming and Wi-Fi communications
2) The robot moves on four wheels, of which the front two are steerable
3) The robot is battery powered
4) The video stream is accessible via a web browser
5) The robot is controlled via a web browser using a keyboard
6) The target platform for robot control software is a PC
7) The project includes hardware assembly and software development
8) The software is developed using C++ language and ESP-IDF framework
9) The project is divided into two parts, each part for a different person:
   - **Antoni Bajor**: responsible for hardware assembly and software development of the robot motors control **(sections: 3.1, 4, 5.1 - 5.2, 6.2)**
   - **Filip Izworski**: responsible for software development of the robot remote control and camera video streaming **(sections: 3.2 – 3.4, 5.3 - 5.6, 6.1)**

3 Functionality

This section outlines the key functionalities of the mobile robot, which were designed to meet the requirements of the project.

3.1 Robot movement

The microcontroller is receiving commands through Wi-Fi network as described below. Each key can trigger different action, when it is pressed, or released. The robot uses respective functions from SteeringLibrary on every key action, to ensure fluid motion. It can work in two ways. First is “tank” like motion, when the wheels do not turn sideways. Second one is like typical car, when two front wheels can turn freely. Thanks to two mounted servos, the robot can use the second option.

3.2 Remote control

The robot operates over a Wi-Fi connection, enabling remote control through a web application. The robot operates in station mode, connecting to an existing Wi-Fi network. To establish the connection, the user must provide the SSID and password. This allows the robot to be seen by other devices within the same network. The developed Wi-Fi module supports both static IPv4 address configuration and DHCP. The first option is more useful here, because to see the address given by DHCP user must connect the microcontroller to PC and use terminal.

To enable remote control the robot hosts an HTTP server that manages the user interface. The HTTP server module allows to configure the port number, to avoid conflicts with other services. It also provides registration and handling of specific URIs to process commands sent from the user.

3.3 Camera video streaming

The camera module provides real-time video streaming to, allowing user to observe the robot’s environment. The video stream runs on its own dedicated HTTP server independently of the control system to ensure uninterrupted streaming. To ensure more smooth streaming the camera uses JPEG compression to encode video frames. This module’s HTTP server uses the specific URI responsible for managing video stream.

3.4 User interface

The central element of the UI is the live view from the robot's camera. The view is embedded directly into the interface, eliminating the need to switch between multiple pages. The interface includes a set of buttons for controlling the robot's movements and additionally controlling the onboard LED. The UI provides visual representation of the currently pressed button. When a button is pressed it changes color to black, otherwise it stays white. Four arrow keys are used for movement and the L key for the speed control. Figure 1 shows the layout of the user interface.

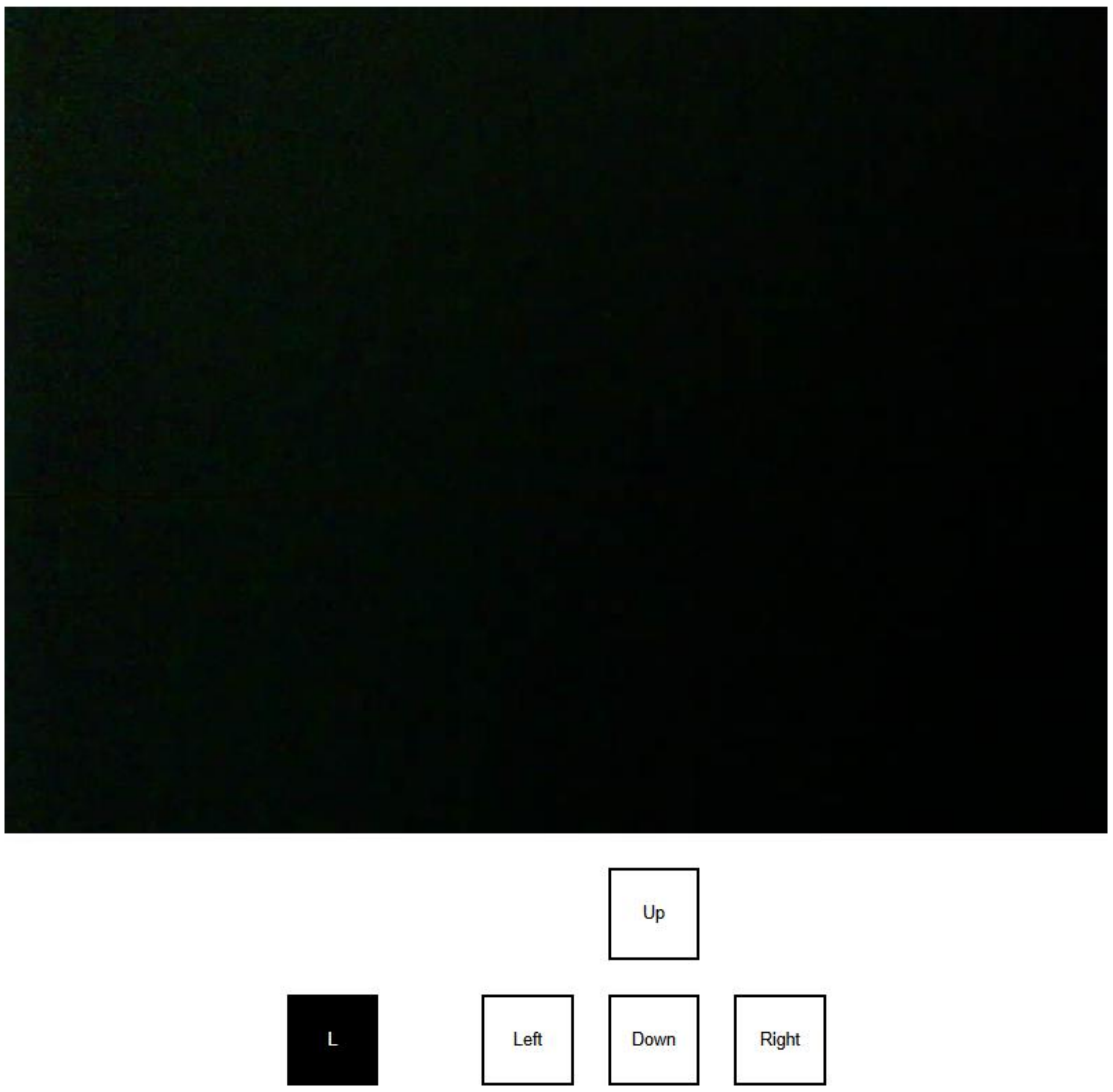


Figure 1. User interface layout (with the L key pressed)

4 Hardware

The base of the robot is Waveshare Robot Chassis NS, with modified way of assembly. On the underside in the front, there are two SG90 servos, with mounted TT motor and wheel on each. The motors are secured by double-sided tape, paperclips, rubberbands and screws, so they don't slip on the servos. The second, smaller plate is mounted on the underside in the back, with motors screwed to it, so the wheels are in line and the robot is not tilted. On the top, there is mounted all of its logic, that is: ESP32 WROVER-E CAM on modified breadboard, L298N driver board, ON/OFF switch and 2x18650 battery basked. All of these elements are glued to the chassis with double-sided tape. The logic pins are connected by standard breadboard cables, and the motors by universal thin cables. Below are schematics and pinouts used in project.

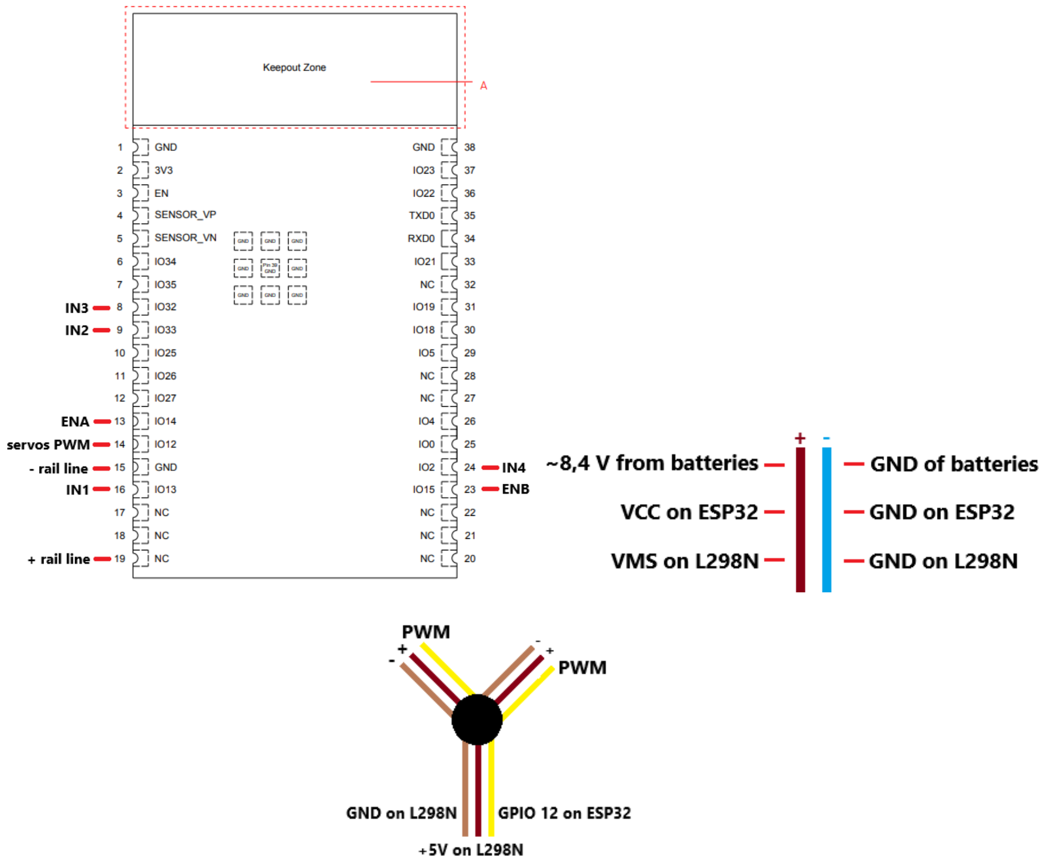


Figure 2. Hardware schematics

Y splitter is used to control two steering servos, with only one GPIO pin. The main switch enabling the robot, is connected in series, between the battery basket and + power rail. The camera is connected to the ESP32, by its own dedicated connector – SCCB.

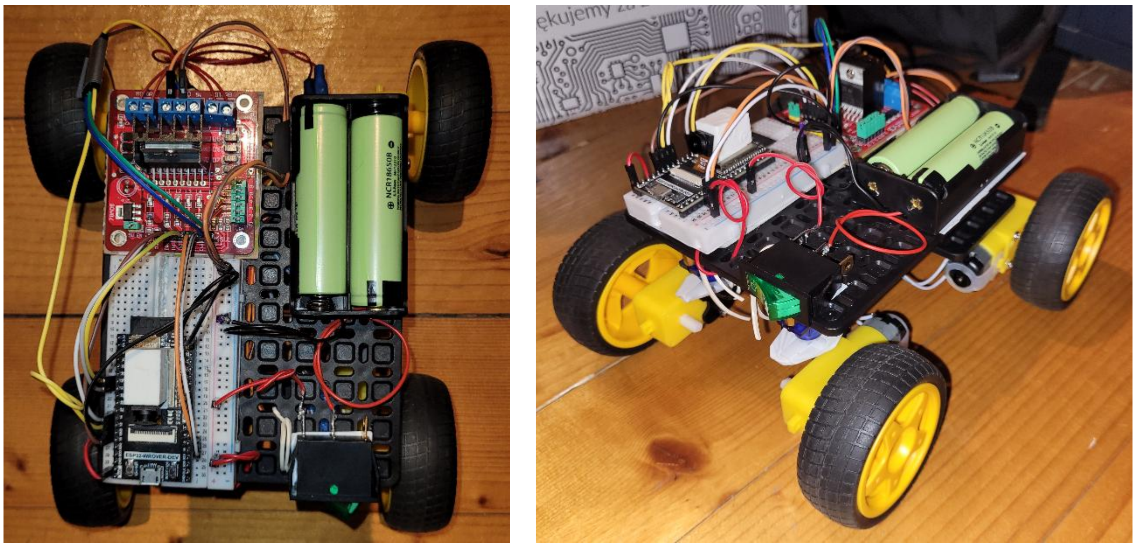


Figure 3. Photos of the robot

5 Software

This section provides a description of the software developed for the project. It outlines the software architecture and tools used during development. The software consists of several classes, each responsible for specific aspects of the robot's operation. Below is a class diagram illustrating the relationships and dependencies between these classes. At the very bottom is the main function that combines all the functionality.

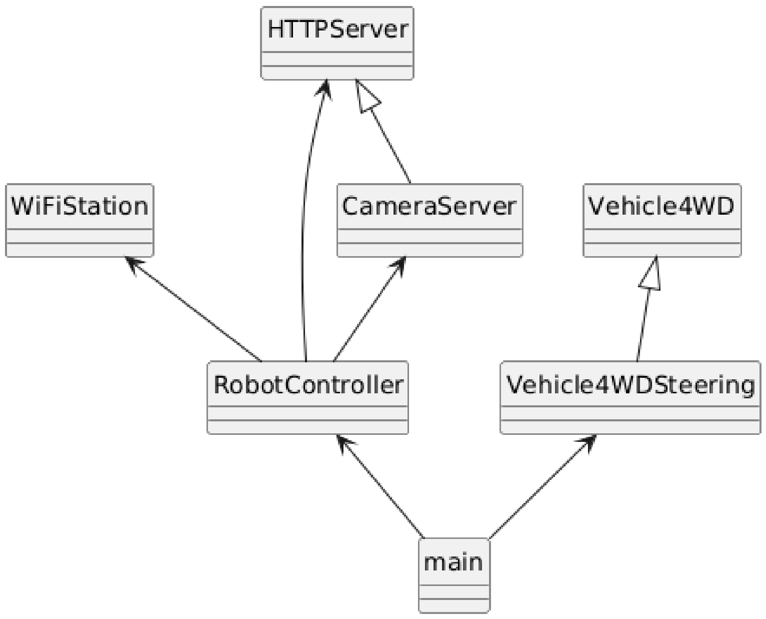


Figure 4. Class diagram of the software developed for the mobile robot

5.1 Vehicle4WD

The Vehicle4WD class is the base class of SteeringLibrary. It sets up the GPIO pins to output mode, and make two PWM signals one for setting speed of the vehicle, and the other one for controlling servos. In this class, all wheels' angles are fixed in one position, and robot can turn by setting both sides to rotate in opposite directions.

Public methods:

- **Vehicle4WD()** – constructor, initializes the GPIO pins and sets up PWM signals with starting values
- **driveForward()** – drive forward for the given value in milliseconds, or drive infinitely if 0 is passed
- **driveBackwards()** – drive backwards for given amount of time in milliseconds, or drive infinitely if 0 is passed
- **stop()** – stop the vehicle.
- **turnLeft()** – turn left, with left wheels rotating backwards, and right wheels rotating forward for given amount of time in milliseconds, or infinitely if 0 is passed.
- **turnRight()** – Same as turnLeft() but the other way. Both methods are virtual
- **setSpeed()** – sets speed of the robot in 0-100 range, which changes the duty of PWM signal enabling motors
- **getSpeed()** – get current speed of the vehicle
- **getDirection()** – get current direction in which the vehicle is moving
- **makeSquare()** – example function not used in our project

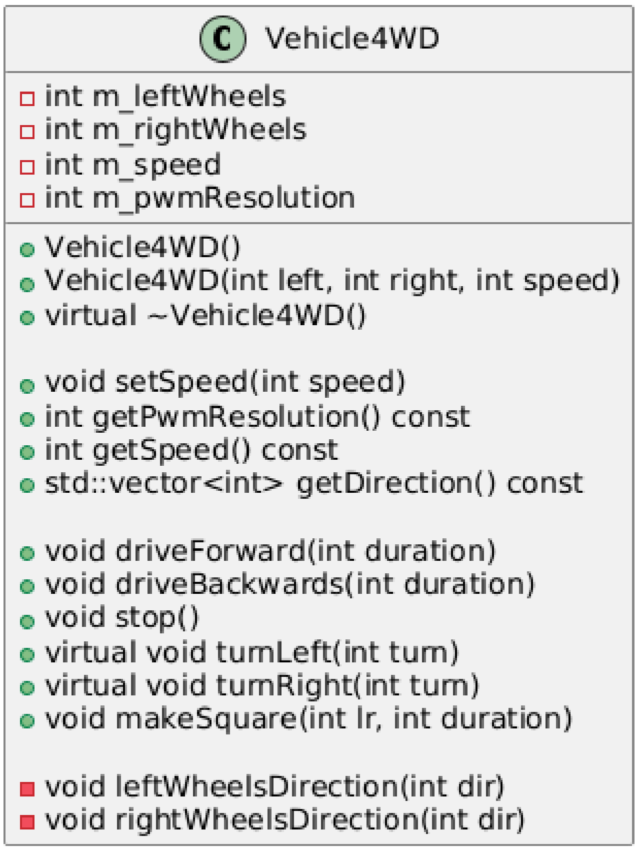

C
Vehicle4WD
int m_leftWheels
int m_rightWheels
int m_speed
int m_pwmResolution
Vehicle4WD()
Vehicle4WD(int left, int right, int speed)
virtual ~Vehicle4WD()
void setSpeed(int speed)
int getPwmResolution() const
int getSpeed() const
std::vector<int> getDirection() const
void driveForward(int duration)
void driveBackwards(int duration)
void stop()
virtual void turnLeft(int turn)
virtual void turnRight(int turn)
void makeSquare(int lr, int duration)
void leftWheelsDirection(int dir)
void rightWheelsDirection(int dir)

Figure 5. Vehicle4WD class diagram

5.2 Vehicle4WDSteering

The Vehicle4WDSteering class is derived from Vehicle4WD, and adds functionality of changing angle of the two front wheels, by overriding the turning functions.

Public methods:

- **Vehicle4WDSteering()** – constructor, uses constructor of base class to initialize necessary things, and sets default angle of the wheels, which normally is 0 degrees
- **turnLeft()** – overrides function from base class, instead of turning wheels in opposite directions, it changes the angles of servos with attached motors. The range that can be passed is 0-40 degrees
- **turnRight()** – The same as turnLeft() function but to the right side. The overall range of turn is -40 – 40 degrees
- **getTurnAngle()** – get current angle of the front wheels in degrees
- **makeCircle()** – example function not used in our project

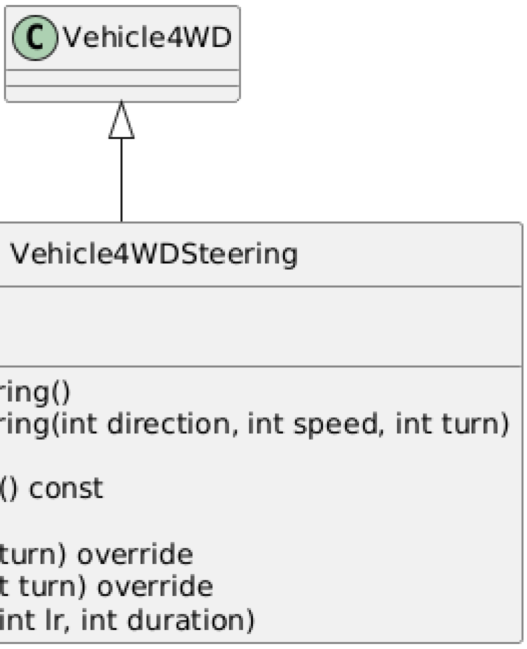

Vehicle4WD
Vehicle4WDSteering
int m_turnAngle
int m_duty
Vehicle4WDSteering()
Vehicle4WDSteering(int direction, int speed, int turn)
int getTurnAngle() const
void turnLeft(int turn) override
void turnRight(int turn) override
void makeCircle(int lr, int duration)

Figure 6. Vehicle4WDSteering class diagram

5.3 WiFiStation

The **WiFiStation** class is responsible for managing the configuration and operation of the Wi-Fi module in the station mode on ESP32 board. It provides functionality for setting credentials, configuring static IPv4 address, initializing the Wi-Fi module and handling connection events. It defines an enumeration **State** to track the current status of the station. This class was created based on the Wi-Fi station sample project provided by the ESP-IDF framework. Below is a description of the implemented methods. The whole **WiFiStation** class is shown on figure 6.

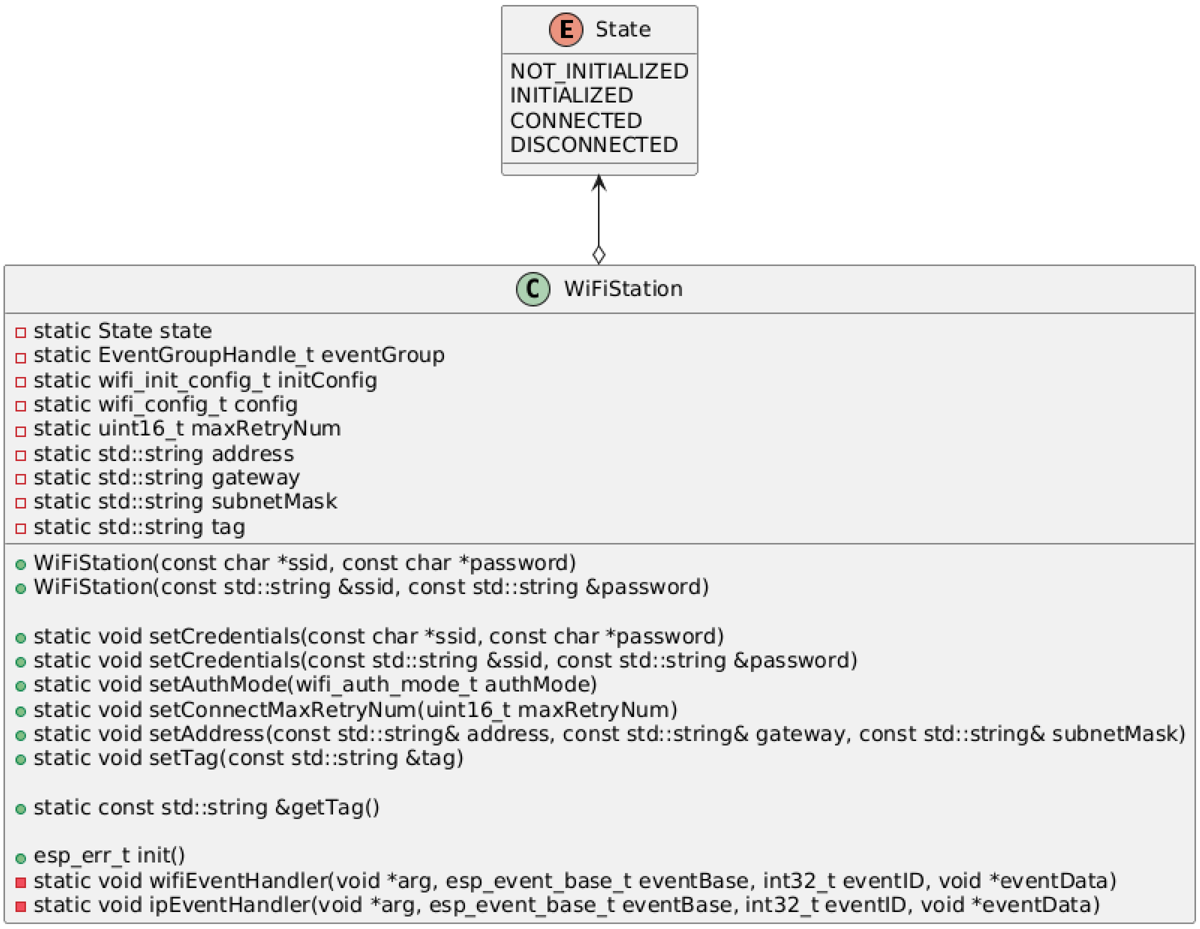
Public methods:

- **WiFiStation()** – constructor, initializes the Wi-Fi station using either C-style strings or C++ std::string credentials
- **setCredentials()** – sets the SSID and password using either C-style strings or C++ std::string
- **setAuthMode()** – configures Wi-Fi authentication mode (WPA2-PSK is default)
- **setConnectMaxRetryNum()** – sets the maximum number of retry attempts for connecting to a network
- **setAddress()** – assigns a static IPv4 address, gateway and subnet mask to the station

- **setTag()** – sets a custom tag for log messages used during debugging
- **getTag()** – retrieves the current log tag identifier
- **init()** – initializes the Wi-Fi station and applies the configured settings, it also registers event handlers for Wi-Fi and IP events

Private methods:

- **wifiEventHandler()** – handles Wi-Fi events such as connection and disconnection
- **ipEventHandler()** – handles IP events including successful acquisition of an IP address



```
State
NOT_INITIALIZED
INITIALIZED
CONNECTED
DISCONNECTED
```

```
WiFiStation
static State state
static EventGroupHandle_t eventGroup
static wifi_init_config_t initConfig
static wifi_config_t config
static uint16_t maxRetryNum
static std::string address
static std::string gateway
static std::string subnetMask
static std::string tag

WiFiStation(const char *ssid, const char *password)
WiFiStation(const std::string &ssid, const std::string &password)

static void setCredentials(const char *ssid, const char *password)
static void setCredentials(const std::string &ssid, const std::string &password)
static void setAuthMode(wifi_auth_mode_t authMode)
static void setConnectMaxRetryNum(uint16_t maxRetryNum)
static void setAddress(const std::string& address, const std::string& gateway, const std::string& subnetMask)
static void setTag(const std::string &tag)

static const std::string &getTag()

esp_err_t init()
static void wifiEventHandler(void *arg, esp_event_base_t eventBase, int32_t eventID, void *eventData)
static void ipEventHandler(void *arg, esp_event_base_t eventBase, int32_t eventID, void *eventData)
```

Figure 7. WiFiStation class diagram

5.4 HTTPServer

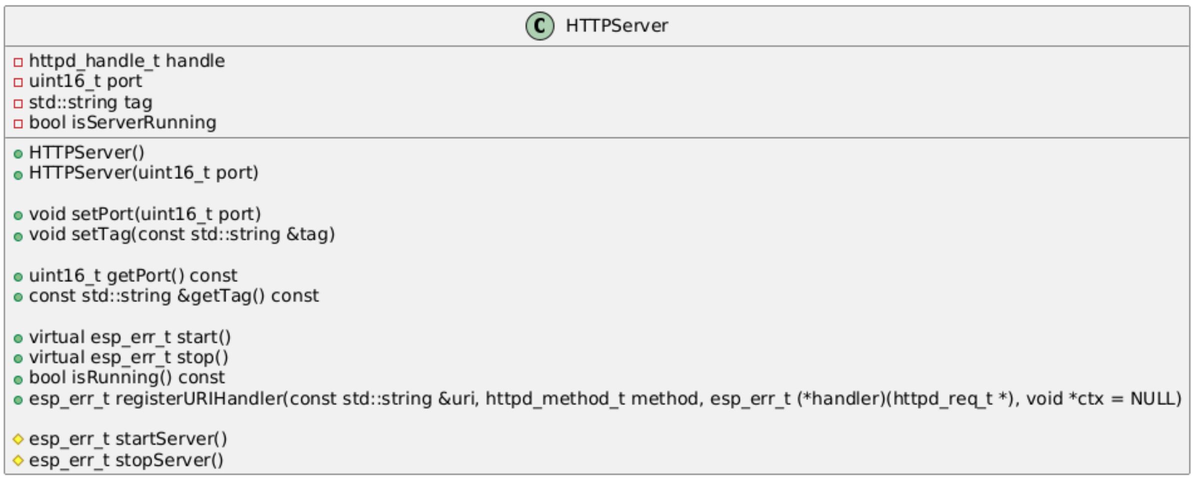
The **HTTPServer** class manages the setup and operation of an HTTP server on the ESP32 microcontroller. It provides functionality for configuring the server, starting and stopping it and registering URI handlers for processing incoming HTTP requests. The server operates on a configurable port, with default value of 80.

Public methods:

- **HTTPServer()** – constructor, initializes the server with either a user-specified port or a default port 80
- **setPort()** – sets the port number for the server
- **setTag()** – sets a custom tag for log messages used during debugging
- **getPort()** – retrieves the currently configured port number
- **getTag()** - retrieves the current log tag identifier
- **start()** – starts the HTTP server by calling internally the **startServer()** method, it is virtual and can be overridden in derived classes
- **stop()** – stops the HTTP server by calling internally the **stopServer()** method, it is virtual and can be overridden in derived classes
- **isRunning()** – returns true if the server is currently running, otherwise false
- **registerURIHandler()** – registers a handler for the specified URI and HTTP method, the handler function processes incoming requests

Protected methods:

- **startServer()** – initializes and starts the HTTP server
- **stopServer()** – stops the HTTP server


```
HTTPServer

httpd_handle_t handle
uint16_t port
std::string tag
bool isServerRunning

HTTPServer()
HTTPServer(uint16_t port)

void setPort(uint16_t port)
void setTag(const std::string &tag)

uint16_t getPort() const
const std::string &getTag() const

virtual esp_err_t start()
virtual esp_err_t stop()
bool isRunning() const
esp_err_t registerURIHandler(const std::string &uri, httpd_method_t method, esp_err_t (*handler)(httpd_req_t *), void *ctx = NULL)

esp_err_t startServer()
esp_err_t stopServer()
```

Figure 8. HTTPServer class diagram

5.5 CameraServer

The **CameraServer** class extends the functionality of the **HTTPServer** class to support real-time video streaming from a camera module connected to the ESP32. It heavily relies on the external esp_camera library to manage the camera module's hardware. This library provides all the necessary functionality for configuring and capturing images, including support for JPEG compression. This class acts like a wrapper around the example provided in the esp_camera library repository, simplifying the setup and integration of camera streaming functionality.

Public methods:

- **CameraServer()** – constructor, initializes the server with either a user-specified port or a default port 80
- **start()** – starts the HTTP server and initializes the camera module, it also registers the JPEG stream URI (/stream) handler for real-time video streaming

Private methods:

- **cameraInit()** – configures and initializes the camera module using the esp_camera library
- **jpgStreamHandler()** – handles incoming HTTP requests for the /stream URI by sending a video stream to the client

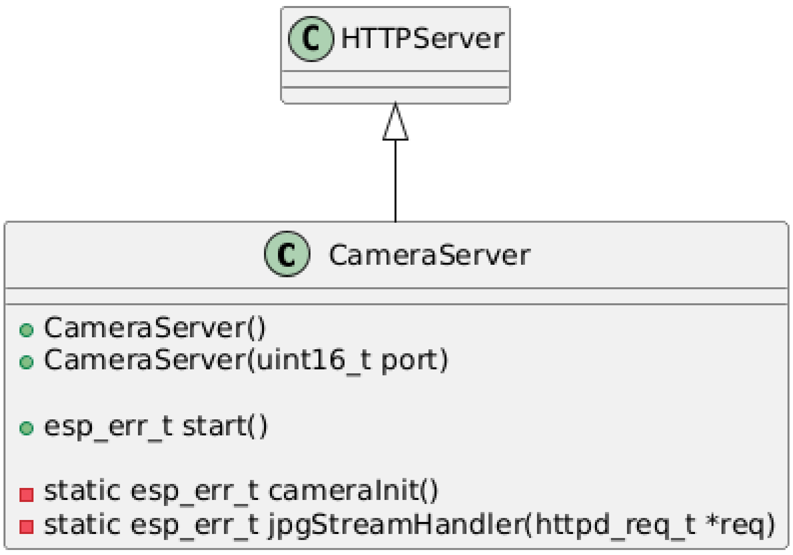

C HTTPServer
C CameraServer
CameraServer()
CameraServer(uint16_t port)
esp_err_t start()
static esp_err_t cameraInit()
static esp_err_t jpgStreamHandler(httpd_req_t *req)

Figure 9. CameraServer class diagram

5.6 RobotController

The **RobotController** class controls the operation of a Wi-Fi connected mobile robot, managing its network connection, user interaction via a web browser and integration with the camera server. It uses the **WiFiStation** class to establish and maintain the robot's connection to a Wi-Fi network while managing two HTTP servers: one for handling control commands such as movement and another for real-time video streaming. The class provides key event handling, supporting predefined keys like ARROW_UP, ARROW_DOWN and others, with the ability to register custom callback functions for specific key press or release events. Additionally, the **RobotController** contains the HTML code for the user interface, including embedded JavaScript for detecting and sending key events.

Public methods:

- **RobotController()** – constructor, initializes the communication by setting up the Wi-Fi connection and starting two HTTP servers, configures the robot to operate at the fixed IP address (192.168.0.100) with specific ports (80 for control, 81 for video streaming)
- **registerKeyEventHandler()** – allows the registration of callback functions to handle specific key events

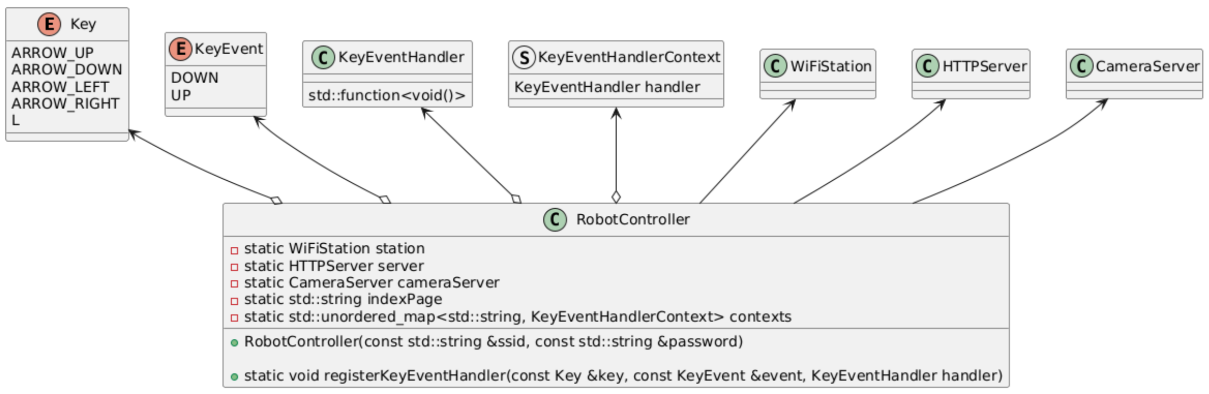

E Key
ARROW_UP
ARROW_DOWN
ARROW_LEFT
ARROW_RIGHT
L
E KeyEvent
DOWN
UP
C KeyEventHandler
std::function<void()>
S KeyEventHandlerContext
KeyEventHandler handler
C WiFiStation
C HTTPServer
C CameraServer
C RobotController
static WiFiStation station
static HTTPServer server
static CameraServer cameraServer
static std::string indexPage
static std::unordered_map<std::string, KeyEventHandlerContext> contexts
RobotController(const std::string &ssid, const std::string &password)
static void registerKeyEventHandler(const Key &key, const KeyEvent &event, KeyEventHandler handler)

Figure 10. RobotController class diagram

5.7 Tools

- Visual Studio Code

  Using this IDE enables the integration of the ESP-IDF extension, which significantly simplifies software development for ESP32 boards. The IDE also offers a built-in terminal for debugging and IntelliSense, providing syntax highlighting and code suggestions.

- C++ language

  C++ was chosen for its compatibility with the ESP32 platform, allowing modular software design.

- ESP-IDF framework

  The ESP-IDF (Espressif IoT Development Framework) is the official framework provided by Espressif for developing applications on ESP32 devices. It includes libraries for key functionalities such as Wi-Fi, HTTP server, and camera modules. The framework provides a variety of example projects, which demonstrate the usage of its libraries and serve as a foundation for custom implementations. ESP-IDF also automates the generation of CMake files required for project building.

- GitHub

  GitHub is used to manage the source code and track progress. It also facilitates collaboration.

6 Testing

6.1 Class-specific testing

Each class in the project was tested using dedicated test project created specifically to verify its functionality. These test projects focused entirely on the behavior of the corresponding class, ensuring that each component worked correctly in isolation. For classes such as **HTTPServer** and **CameraServer**, testing required the additional use of the **WiFiStation** class to establish a Wi-Fi connection.

Testing involved running each test project on the ESP32 microcontroller and observing the output generated by it. To facilitate debugging and validation, log messages were used throughout the code. The terminal was used to monitor and analyze these logs.

All tests on the individual classes were completed successfully, confirming that the functionality of each component met the design requirements and operated as expected.

6.2 Final testing

After creating object of **Vehicle4WDSteering** and **RobotController**, with SSID, password, and static IP of the network to which it would be connected, the ESP32 was flashed. After few seconds, steering interface and camera image, could be accessed through mentioned IP, in web browser. The robot was behaving as intended, driving and turning, the camera was providing live image as well. Everything was working well, with small interruptions when Wi-Fi signal was getting weak.

Sources

[1] https://esp32tutorials.com/
[2] https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html
[3] https://github.com/espressif/esp32-camera