

# Diplomarbeit: Lastenroboter

Höhere Technische Bundeslehranstalt Graz Gösting  
Schuljahr 2024/25



**Diplomanden:**

Daniel Schauer 5AHEL  
Simon Spari 5AHEL  
Felix Hochegger 5AHEL

**Betreuer:**

Prof. DI. Gernot Mörtl

---

# Eidesstattliche Erklärung

Wir erklären an Eides statt, dass wir die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht haben.

---

Ort, am TT.MM.JJJJ

---

Daniel Schauer

---

Simon Spari

---

Felix Hochegger

---

# Danksagung

An dieser Stelle möchten wir unseren aufrichtigen Dank aussprechen.

Ein besonderer Dank gilt Herrn Prof. DI Gernot Mörtl für seine wertvolle Unterstützung, seine fachliche Begleitung und seine konstruktiven Anregungen während der gesamten Arbeit. Seine Expertise und sein Engagement haben maßgeblich zum Gelingen dieser Diplomarbeit beigetragen.

Ebenso danken wir unseren Freunden, insbesondere Michael Johannes Anderhuber, für seine Unterstützung beim Schweißen des Gehäuses. Sein handwerkliches Geschick und seine Hilfe waren für die Umsetzung unseres Projekts von großem Wert.

Unser großer Dank gilt zudem unserem großzügigen Sponsor, "Vogl Baumarkt Rosental", für das Sponsoring des Metalls für das Gehäuse. Durch diese Unterstützung konnten wir unser Projekt in dieser Form verwirklichen.

---

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>7</b>
1.1	Kurzzusammenfassung . . . . .	7
1.2	Abstract . . . . .	8
<b>2</b>	<b>Projektmanagement</b>	<b>9</b>
2.1	Projektteam . . . . .	9
2.2	Projektstrukturplan . . . . .	10
2.3	Meilensteine . . . . .	10
2.4	Kostenaufstellung . . . . .	11
<b>3</b>	<b>Antrieb</b>	<b>12</b>
3.1	Motoren . . . . .	12
3.1.1	Übersicht . . . . .	12
3.1.2	Funktionsweise . . . . .	12
3.1.3	Technische Daten . . . . .	14
3.2	Motorentreiber . . . . .	14
3.2.1	Überblick . . . . .	14
3.2.2	Aufbau und Funktionen . . . . .	14
3.3	Schaltungsaufbau . . . . .	17
3.4	Code . . . . .	17
<b>4</b>	<b>Webserver</b>	<b>18</b>
4.1	Grundlegende Ziele . . . . .	18
4.2	Ideen und Entwürfe . . . . .	19
4.3	Webserver . . . . .	19
4.3.1	Webserver Setup . . . . .	19
4.3.2	SPIFFS Setup . . . . .	20
4.4	WebSocket Kommunikation . . . . .	22
4.4.1	Kommunikation Setup . . . . .	22
4.4.2	Message Handling . . . . .	24
4.5	Kamera . . . . .	28
4.6	Kamera . . . . .	28
4.6.1	Kamera Setup . . . . .	28
4.7	Videoübertragung . . . . .	29
4.8	Website . . . . .	31

---

4.8.1	Implementierung der Steuerung . . . . .	31
4.8.2	Echtzeit-Videoanzeige . . . . .	44
4.8.3	Anzeige von Sensordaten und Verbindungsstatus . . . . .	47
4.9	Herausforderungen und Optimierungen . . . . .	53
4.9.1	Probleme bei der WebSocket Kommunikation . . . . .	53
4.9.2	Latenz- und Performance Optimierungen . . . . .	53
4.9.3	(Speicher- und Rechenleistungseinschränkungen des ESP32) . . .	54
4.10	(Fazit und Ausblick) . . . . .	54
4.10.1	(Mögliche Erweiterungen und Verbesserungen) . . . . .	54
<b>5</b>	<b>Gehäuse</b>	<b>55</b>
5.1	Planung und Design . . . . .	55
5.2	Realisierung . . . . .	55
5.3	Materialliste . . . . .	55
<b>6</b>	<b>Platine</b>	<b>56</b>
6.1	Grundschialtung . . . . .	56
6.2	Circuit Board . . . . .	56
6.3	Fertiger Prototyp . . . . .	56
<b>7</b>	<b>Kamera</b>	<b>57</b>
7.1	Kamera im Überblick . . . . .	57
7.2	Videoübertragung . . . . .	57
7.3	Kameraschwenkung . . . . .	57
7.3.1	Gehäuse . . . . .	57
7.3.2	Servomotor . . . . .	57
7.4	Code . . . . .	57
<b>8</b>	<b>Sensoren</b>	<b>58</b>
8.1	Abstandsensor . . . . .	58
8.2	Gewichtsmessung . . . . .	58
8.2.1	Grundprinzip . . . . .	58
8.2.2	Schaltungsaufbau . . . . .	58
8.2.3	Code . . . . .	58
<b>9</b>	<b>Entwicklungstools</b>	<b>59</b>
9.1	Autodesk Fushion . . . . .	59
9.2	Eagle . . . . .	59
9.3	VS-Code . . . . .	59

---

9.3.1	Setup . . . . .	59
9.3.2	Bibliotheken . . . . .	60
9.3.3	verwendete Bibliotheken . . . . .	60
9.4	LaTeX . . . . .	61
9.5	GitHub . . . . .	61
<b>10</b>	<b>Abbildungsverzeichnis</b>	<b>62</b>
<b>11</b>	<b>Literaturverzeichnis</b>	<b>64</b>

---

# 1 Einleitung

## 1.1 Kurzzusammenfassung

In dieser Diplomarbeit wird ein Lastenroboter entwickelt, der bis zu 25 Kilogramm transportieren kann. Der Roboter wird über eine Website gesteuert, die als Steuerungsplattform dient. Zusätzlich ist eine Kamera eingebaut, die den Transportbereich zeigt, sowie eine Waage, die das Gewicht der transportierten Last misst.

Ein Schwerpunkt der Arbeit liegt auf der mechanischen Konstruktion des Roboters, bei der ein stabiles Gehäuse aus Stahl gebaut wird, um Sicherheit und Stabilität zu gewährleisten. Außerdem wird eine eigene Platine entwickelt, die die verschiedenen Hardware-Komponenten, wie die Sensoren und die Motoren, steuert und miteinander verbindet.

Die Steuerung des Roboters erfolgt über eine Website, die es dem Benutzer ermöglicht, den Roboter zu bedienen und wichtige Daten wie Akkustand und Gewicht abzurufen. Ein besonderer Fokus liegt auch dabei auf der Übertragung des Kamerabildes auf die Web-Oberfläche sowie der Integration einer schwenkbaren Kamera, um eine flexible Sicht auf den Transportbereich zu gewährleisten. Der ESP32 sorgt dafür, dass die Befehle des Benutzers an den Roboter übermittelt werden.

Zusätzlich wird eine OnBoard-Software entwickelt, die es ermöglicht, die Sensoren auszulesen und die Motoren als auch die Kamera anzusteuern.

---

## 1.2 Abstract

This thesis develops a load robot that can carry up to 25 kilograms. The robot is controlled via a website, which serves as the control platform. Additionally, a camera is integrated to display the transport area, as well as a scale to measure the weight of the carried load.

A key focus of the work is on the mechanical design of the robot, where a sturdy steel housing is built to ensure safety and stability. Furthermore, a custom circuit board is developed to control and connect the various hardware components, such as sensors and motors.

The robot is controlled via a website, which allows the user to operate the robot and access important data such as battery level and weight. A particular focus is also placed on transferring the camera feed to the web interface and integrating a swivel camera to ensure flexible viewing of the transport area. The ESP32 ensures that the user's commands are transmitted to the robot.

Additionally, onboard software is developed to read the sensors and control the motors and camera.



---

# 2 Projektmanagement

## 2.1 Projektteam

**Betreuer: Prof. DI. Gernot Mörtl**

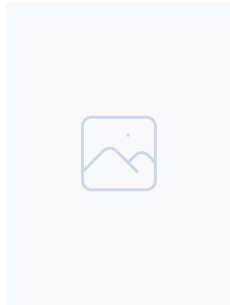


Abbildung 1:  
Porträt  
Daniel Schauer

### **Daniel Schauer: Software-OnBoard**

- Projektleiter
- Verbindung von Software und Hardware (ESP32 zu Sensoren)
- Kamera-Übertragung zur Web-Oberfläche
- Umsetzung einer schwenkbaren Kamera
- Dokumentation

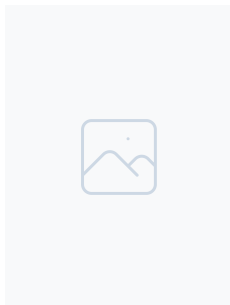


Abbildung 2:  
Porträt  
Simon Spari

### **Simon Spari: Software-App**

- Benutzeroberfläche (Web-Oberfläche) für Steuerung
- Übertragung der Steuerung/Befehle von Web-Oberfläche zu ESP32
- Kamera-Übertragung zur Web-Oberfläche
- Dokumentation

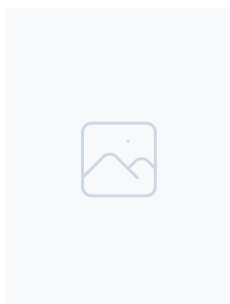


Abbildung 3:  
Porträt Felix  
Hohegger

### **Felix Hohegger: Hardware-Design und Mechanik**

- Bau des Roboter Gehäuse
- Ansteuerung und Verbindung von Hardware (Ansteuerung und Berechnung der Motoren)
- Dokumentation

---

## 2.2 Projektstrukturplan

Unsere Diplomarbeit beschäftigt sich mit der Entwicklung eines Lastenroboters. Der Lastenroboter soll etwa ein Gewicht von 25 Kilogramm tragen können. Die Steuerung erfolgt über eine Handy-App, und zusätzlich soll eine Kamera eingebaut werden. Im Lastenroboter wird auch eine Waage eingebaut, damit man sehen kann, wie schwer die transportierte Last ist.

Das Ziel dieser Diplomarbeit ist also, ein betriebsbereiter Prototyp eines Lastenroboters mit Kamerasystem und Waage zu entwickeln und zu realisieren.

### **Geplantes Ergebnis der individuellen Themenstellungen:**

**Felix Hoegger:** Bau des Roboter-Gehäuses und Integration der Komponenten, Verbindung der Hardware im besonderen Ansteuerung der Motoren

**Simon Spari:** Entwicklung einer Benutzeroberfläche (Web-Oberfläche) zur Steuerung des Roboters, Übertragung der Steuerbefehle in Echtzeit an die Hardware

**Daniel Schauer:** Verbindung von Software und Hardware zur Umsetzung der Steuerbefehle, Kamera-Übertragung in Echtzeit zur GUI, Umsetzung schwenkbare Kamera

**Projektstrukturplan:** Bild einfügen vom Grundplan.

## 2.3 Meilensteine

Um unseren Fortschritt und unsere Zeiteinteilung besser im Überblick zu behalten, haben wir uns bestimmte Meilensteine für unser Projekt gesetzt. Diese sind sehr hilfreich, um das Projekt strukturiert umzusetzen, angefangen von der Projektplanung bis hin zum fertigen Prototyp.

Die folgenden Meilensteine haben wir uns bei der Projektplanung gesetzt:

<b>Meilenstein</b>	<b>Datum</b>
Grundlegendes Gehäuse	07.11.2024
Funktionsfähige Website	19.12.2024
Funktionsfähige steuerbare Motoren	16.01.2025
Funktionsfähiger Prototyp	06.03.2025

---

## 2.4 Kostenaufstellung

Artikel	Einzelpreis	Stückzahl	Gesamt
Aufblasbares Rad 10" 260x85	16.70 €	4	66.80 €
MCP23017	4.54 €	2	9.08 €
PICAA LED Arbeitsscheinwerfer	6.55 €	2	13.10 €
2 Stück PWM Motor Steuerung Treiber Platinen	35.78 €	2	71.56 €
Micro Servo Motor	6.04 €	1	6.04 €
IRM-30-15ST	17.04 €	1	17.04 €
SOLSUM 0808	25.17 €	1	25.17 €
Platinen	37.14 €	1	37.14 €
ESP32 CAM	13.70 €	1	13.70 €
12 Stück Halbbrücken Wägezelle	11.09 €	1	11.09 €
ESP32	11.09 €	1	11.09 €
Dunkermotoren	65.00 €	2	130.00 €
AGM 12V Batterie	24.80 €	1	24.80 €
Diverse Kleinteile	30.00 €	1	30.00 €
<b>Summe</b>			<b>466.61 €</b>

---

## 3 Antrieb

### 3.1 Motoren

#### 3.1.1 Übersicht

Für den Antrieb des Lastenroboters werden BLDC-Motoren eingesetzt. Verwendet werden die BLDC-Motoren von Dunkermotoren (Typ BG 40X25). Ein BLDC-Motor auch bürstenloser Gleichstrommotor genannt wird über drei Phasen betrieben. Der BLDC-Motor wird oft in der Industrie eingesetzt weil er eine hohe Leistung bietet.



Abbildung 4: Motoren

Quelle: eigene Abbildung

#### 3.1.2 Funktionsweise

Die Funktionsweise eines bürstenlosen Gleichstrommotors basiert auf die Wechselwirkung zwischen dem Magnetfeld des Stators und dem Magnetfeld des Rotors.

**Stator:** Der Stator besteht aus mehreren Spulen, die üblicherweise drei Phasen umfasst. Die drei Phasen werden mit Wechselstrom angesteuert die dann ein rotierendes Magnetfeld erzeugen.

**Rotor:** Der Rotor besteht aus einem Permanentmagneten. Dadurch das die Spulen ein rotierendes Magnetfeld erzeugen dreht sich der Rotor synchron mit diesem Magnetfeld mit.

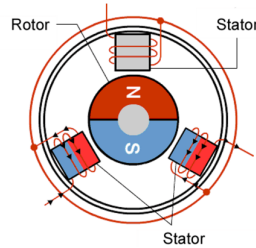


Abbildung 5: BLDC-Motor  
Quelle: [www.elektrikrehberiniz.com](http://www.elektrikrehberiniz.com)

Damit die Phasen richtig geschaltet werden muss man wissen, wo sich der Rotor befindet. Die genaue Position des Rotors wird durch Hallsensoren erfasst. Die Hallsensoren geben diese Informationen an die Steuerelektronik weiter. Mit diesen Daten kann die optimale Beschaltung der Phasen berechnet werden.

Zur Steuerung der Drehzahl wird ein PWM-Signal verwendet. Durch die PWM wird die Stromzufuhr zu den Phasen des Motors gesteuert. Wenn man die PWM erhöht, werden die Phasen schneller angesteuert und der Rotor dreht sich schneller. Das heißt durch das PWM-Signal wird die Versorgungsspannung an den Phasen schnell ein- und ausgeschaltet.

Auf diese Weise kann der Motor effizient mit variabler Geschwindigkeit betrieben werden.

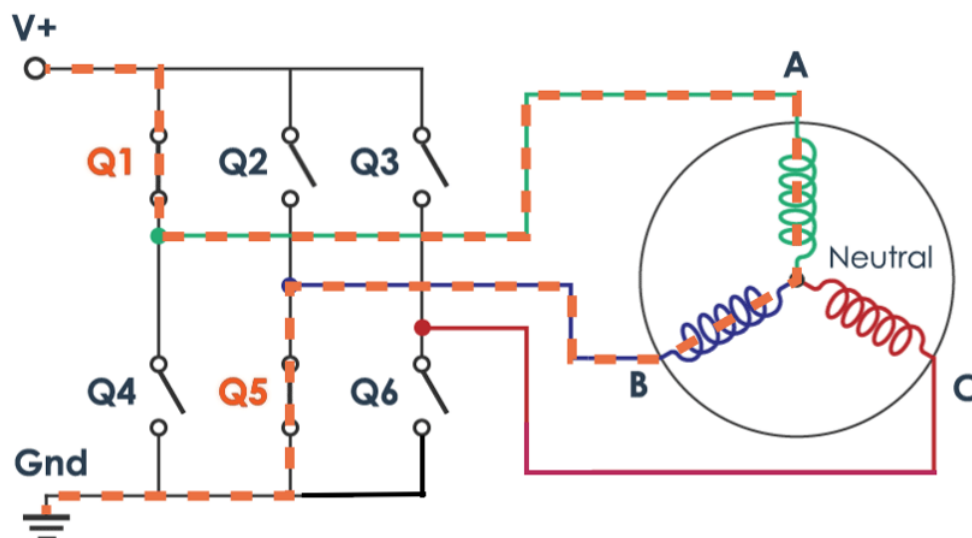


Abbildung 6: Phasensteuerung  
Quelle: [davincii.de/arduino-projekte/brushless-motor-ansteuerung](http://davincii.de/arduino-projekte/brushless-motor-ansteuerung)

### 3.1.3 Technische Daten

Eigenschaft	Wert
Spannung	15V
Rotationsgeschwindigkeit der Antriebswelle	3260 rpm
Max. Strom	2A

## 3.2 Motorentreiber

### 3.2.1 Überblick

Für die Ansteuerung der BLDC-Motoren wird der Motortreiber ZX-X11H verwendet. Dieser Motortreiber übernimmt die elektronische Kommutierung, das heißt dieser Treiber schaltet die Phasen des Motors damit sich der Rotor dreht. Die Rotorposition wird über Hallsensoren erfasst. Mit diesen Daten werden die Phasen richtig angesteuert damit sich der Rotor dreht. Dies ermöglichten einen gleichmäßigen Betrieb sowie eine exakte Regelung von Drehzahl und Drehmoment. Durch die integrierte PWM-Steuerung kann die Geschwindigkeit präzise eingestellt werden. Zusätzlich verfügt der ZS-X11H über eine Richtungssteuerung und Bremse.

### 3.2.2 Aufbau und Funktionen

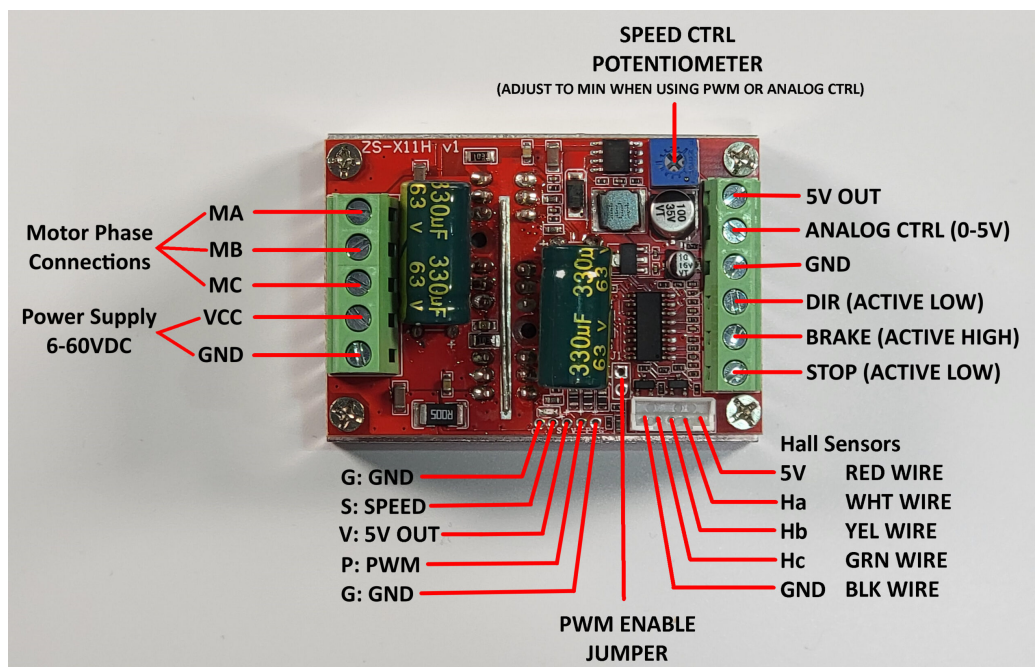


Abbildung 7: Motortreiber-ZS-X11H

Quelle: [mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g](http://mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g)

---

### Steuerung der Phasenströme:

- Der Treiber steuert die drei Motorphasen (MA, MB, MC) mithilfe eines integrierten MOSFET-Brückenschaltkreises.
- Mit den Daten der Hallsensoren schaltet der Treiber die Phasen damit sich der Rotor mit der gewünschten Geschwindigkeit dreht.



Abbildung 8: Motorphasen Verbindung

Quelle: [mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g](http://mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g)

### PWM-Steuerung

- Die PWM steuert die Geschwindigkeit, indem sie die Leistung moduliert, die den Motorwicklungen zugeführt wird..
- Die Amplitude des PWM-Signals muss zwischen 2,5-5 V liegen.
- Die PWM-Frequenz muss zwischen 50-20 kHz liegen.
- Die Platine ist ausgestattet mit einer externen und internen PWM-Steuerung.

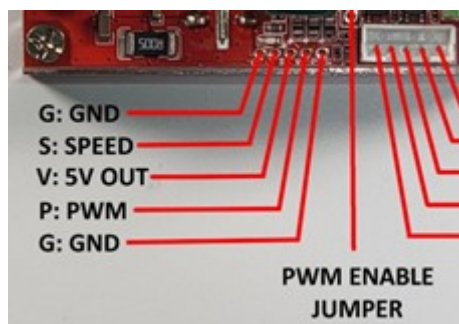


Abbildung 9: Pins für die PWM

Quelle: [mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g](http://mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g)

---

## Richtungswechsel und Bremse

- Die Platine verfügt über zwei Steuereingänge für Richtungswechsel (DIR) und eine Bremsfunktion (BRAKE).
- Der Richtungswechsel erfolgt, indem das Signal bei dem Eingang von LOW auf HIGH oder umgekehrt wechselt. Das Umschalten der Richtung wechselt die Reihenfolge der Phasenströme, wodurch sich dann der Motor in die andere Richtung dreht.
- Die Bremse reagiert auf ein HIGH-Signal. Das heißt wenn am Eingang ein HIGH-Signal angelegt wird, stoppt der Motor, indem die Wicklungen kurzgeschlossen werden.

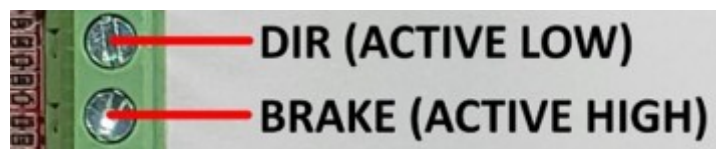


Abbildung 10: Pin für die Bremse und den Richtungswechsel  
Quelle: [mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g](http://mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g)

## Spannungs- und Stromversorgung

- Der Treiber kann mit einer Versorgungsspannung von 12V-60V betrieben werden, wodurch er für eine Vielzahl von BLDC-Motoren geeignet ist.
- Die maximale Stromaufnahme des Treibers liegt bei zirka 15A.
- Der Treiber ist für Motoren bis zu 500 Watt geeignet und kann damit leistungsstarke Motoren betreiben

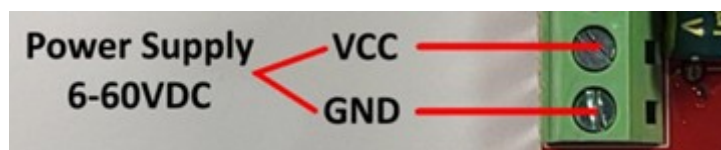


Abbildung 11: Versorgungs-Pins  
Quelle: [mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g](http://mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g)



---

## Hall-Sensoren

- Der Treiber verwendet drei Hall-Sensor-Eingänge. (Ha, Hb, Hc)
- Mit diesen Hall-Sensoren wird die Rotorposition ermittelt damit die Phasen des Motors richtig geschaltet werden.
- Die Sensoren ermöglichen eine effiziente und stabile Steuerung.



Abbildung 12: Hall-sensoren Eingänge

Quelle: [mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g](http://mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g)

## Sicherheit und Schutzfunktionen

- Der Treiber verfügt über einen Überstromschutz und Überspannungsschutz damit der Motor und die Elektronik Geschützt ist.
- Ein thermischer Schutz verhindert Schäden durch Überhitzung.

## 3.3 Schaltungsaufbau

## 3.4 Code

---

# 4 Webserver

## 4.1 Grundlegende Ziele

In diesem Kapitel befassen wir uns mit der geplanten Website, die den Benutzern und Benutzerinnen die Steuerung des Lastenroboters ermöglichen soll. Zuerst definieren wir die grundlegenden Funktionen, die die Website erfüllen soll. Sobald diese erfüllt sind, versuchen wir, das User Interface so einfach und benutzerfreundlich wie möglich zu gestalten. Ein weiterer Punkt ist die Darstellung der spezifischen Messwerte und Daten, damit diese am Webserver schnell und leicht zugänglich sind.

Für die Entwicklung der Website verwenden wir HTML, CSS und JavaScript, um alle funktionalen und optischen Anforderungen zu erfüllen.

### **Videübertragung**

Auf der Website soll eine Echtzeit Videübertragung der ESP32-CAM angezeigt werden. Die Bildfrequenz und die Qualität der Videübertragung sollte ausgeglichen sein, so dass in der Übertragung alle Objekte und ggf. Hindernisse frühzeitig erkennbar sind und noch Reaktionszeit zum Manövrieren besteht.

### **Steuerung**

Auf der Website soll eine grafische Steuereinheit implementiert werden, mit der der Roboter gesteuert und navigiert werden kann. Diese soll dann die entsprechenden Steuerbefehle bzw. Richtungen an den ESP32 senden, wo sie dann in Steuerungsbefehle für die Motoren übersetzt werden.

### **Anzeigen von Daten**

Auf der Website sollen bestimmte Messwerte und Daten, wie zum Beispiel Akkustand oder zurzeit aufliegende Last, die vom ESP32 durch Sensoren oder Messungen ausgewertet werden, übersichtlich und leicht zugänglich angezeigt werden.

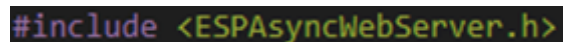
---

## 4.2 Ideen und Entwürfe

### 4.3 Webserver

#### 4.3.1 Webserver Setup

Der Webserver wird mithilfe der Bibliothek ESPAsyncWebServer auf einem ESP32-CAM Mikrocontroller eingerichtet. Diese Bibliothek ermöglicht einen nicht-blockierenden Betrieb, wodurch parallele Anfragen effizient verarbeitet werden können.<sup>1</sup>



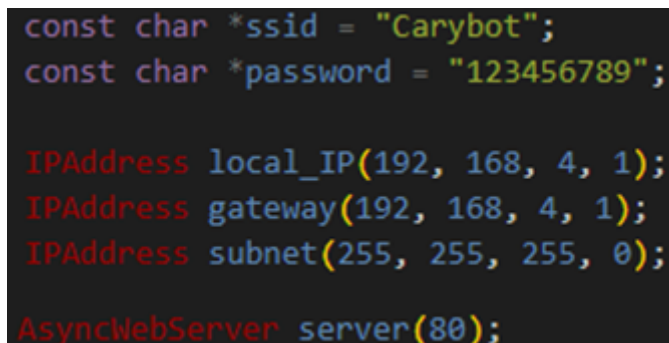
```
#include <ESPAsyncWebServer.h>
```

Abbildung 13: ESPAsyncWebServer.h-Bibliothek  
Quelle: eigene Abbildung

Für die Netzwerkkonfiguration wurde ein eigener Access Point mit folgenden Parametern definiert:

- SSID (Service Set Identifiert) : “Carybot” – dient zur Identifikation des drahtlosen Netzwerkes
- Passwort: “123456789“ – dient zum Schutz des Netzwerkes
- Lokale IP-Adresse: 192.168.4.1 – dient zum Zugriff auf die Webserver-Oberfläche
- Gateway-Adresse: 192.168.4.1 - ESP32-CAM fungiert als Access Point
- Subnetzmaske: 255.255.255.0 - ermöglicht die Kommunikation zwischen Geräten im Bereich 192.168.4.x

Der Webserver wird auf Port 80 erstellt. Port 80 ist der Standardport für HTTP-Dienste.



```
const char *ssid = "Carybot";  
const char *password = "123456789";  
  
IPAddress local_IP(192, 168, 4, 1);  
IPAddress gateway(192, 168, 4, 1);  
IPAddress subnet(255, 255, 255, 0);  
  
AsyncWebServer server(80);
```

Abbildung 14: WebServer Netzwerkkonfiguration  
Quelle: eigene Abbildung

---

<sup>1</sup><https://github.com/lacamera/ESPAsyncWebServer>

---

In der Setup Funktion wird danach überprüft, ob der Access Point erfolgreich konfiguriert worden ist und ob der Access Point erfolgreich gestartet werden kann. Falls ein Fehler auftreten sollte, wird der Setup unterbrochen und die jeweilige Fehlermeldung in der seriellen Konsole ausgegeben. Wenn alles erfolgreich konfiguriert ist und starten kann, wird im Seriellen Monitor die IP-Adresse in der seriellen Konsole ausgegeben. Anschließend wird der Webserver gestartet.

```
if (!WiFi.softAPConfig(local_IP, gateway, subnet)) {  
    Serial.println("AP-Konfiguration fehlgeschlagen.");  
    return;  
}  
  
if (!WiFi.softAP(ssid, password)) {  
    Serial.println("AP konnte nicht gestartet werden.");  
    return;  
}  
  
Serial.println("Access Point gestartet!");  
Serial.print("IP-Adresse: ");  
Serial.println(WiFi.softAPIP());  
server.begin();
```

Abbildung 15: Webserver Setup  
Quelle: eigene Abbildung

#### 4.3.2 SPIFFS Setup

SPIFFS (SPI Flash File System) ist ein leichtgewichtiges Dateisystem für Mikrocontroller mit SPI-Flash-Speicher. Es ermöglicht das Speichern und Verwalten von Dateien direkt im Flash-Speicher des Mikrocontrollers. SPIFFS wird in unserem Projekt benötigt, um statische Dateien für unseren Webserver (HTML-, CSS-, JavaScript Anwendungen) bereitzustellen.

In unserem Code wird zuallererst einmal überprüft, ob SPIFFS beim Start der ESP32-CAM richtig initialisiert werden kann. Falls es fehlschlägt, wird eine Fehlermeldung in der seriellen Konsole ausgegeben und das Programm gestoppt. Ansonsten werden die Webserver-Endpunkte über HTTP-GET-Routen definiert, über die unsere statischen Da-

---

teien aus SPIFFS an Clients gesendet werden.

“/“ ist die Standardroute des Servers. Somit wird dpad.html als Startseite angezeigt, wenn ein Client sich verbindet.

“menu-icon.svg“ und “Fernlicht.svg“ werden als SVG-Bilder (Scalable Vector Graphics) an den Browser gesendet. Image/svg+xml sorgt dafür, dass der Browser die Dateien als SVG-Bilder erkennt.

“mystyles.css“ wird mit text/css als CSS-Datei gesendet und dient zur Formatierung der Website.

“carybot.js“ wird mit application/javascript als Javascript-Datei gesendet und verarbeitet die Eingaben von Clients auf der Website.

Wenn alle Dateien erfolgreich geladen sind, wird eine Nachricht in der seriellen Konsole ausgegeben.

```
if (!SPIFFS.begin(true)) {  
    Serial.println("Fehler beim Mounten von SPIFFS");  
    return;  
}  
  
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {  
    String dpad = readFile(SPIFFS, "/dpad.html");  
    request->send(200, "text/html", dpad);  
});  
  
server.on("/menu-icon.svg", HTTP_GET, [](AsyncWebServerRequest *request) {  
    String icon = readFile(SPIFFS, "/menu-icon.svg");  
    request->send(200, "image/svg+xml", icon);  
});  
  
server.on("/Fernlicht.svg", HTTP_GET, [](AsyncWebServerRequest *request) {  
    String fernlicht = readFile(SPIFFS, "/Fernlicht.svg");  
    request->send(200, "image/svg+xml", fernlicht);  
});  
  
server.on("/mystyles.css", HTTP_GET, [](AsyncWebServerRequest *request) {  
    String css = readFile(SPIFFS, "/mystyles.css");  
    request->send(200, "text/css", css);  
});  
  
server.on("/carybot.js", HTTP_GET, [](AsyncWebServerRequest *request) {  
    String js = readFile(SPIFFS, "/carybot.js");  
    request->send(200, "application/javascript", js);  
});  
  
Serial.println("SPIFFS-Dateien erfolgreich geladen");
```

Abbildung 16: SPIFFS Initialisierung

Quelle: eigene Abbildung

---

Die `readFile()` Funktion wird benötigt, um die jeweiligen Dateien aus SPIFFS lesen zu können. Die Funktion liest eine Datei aus dem SPIFFS-Speicher und gibt den Inhalt als String zurück.

```
String readFile(fs::FS &fs, const char *path) {
    File file = fs.open(path, "r");
    if (!file || file.isDirectory()) {
        Serial.println("- failed to open file for reading");
        return String();
    }

    String fileContent;
    while (file.available()) {
        fileContent += String((char)file.read());
    }
    return fileContent;
}
```

Abbildung 17: `readFile()`-Funktion

Quelle: eigene Abbildung

## 4.4 WebSocket Kommunikation

### 4.4.1 Kommunikation Setup

Für die Kommunikation zwischen dem Webserver und dem ESP32 wird die `ArduinoJson` und die `ArduinoWebSockets` Bibliothek benötigt. Die `ArduinoJson` Bibliothek wird für die Umwandlung der JSON-Steuerbefehle benötigt. Die `ArduinoWebSockets` Bibliothek wird für die Kommunikation über das WebSocket Protokoll benötigt.

```
#include "ArduinoJson.h"
#include "WebSocketsServer.h"
```

Abbildung 18: Bibliotheken für die Kommunikation

Quelle: eigene Abbildung

---

Für die Netzwerkkonfiguration als Client werden folgenden Parameter definiert:

- SSID: "Carybot" – gleiche SSID wie ESP32-CAM
- Passwort: "123456789" – gleiches Passwort wie ESP32-CAM
- Lokale IP-Adresse: 192.168.4.3
- Gateway-Adresse: 192.168.4.1 – Adresse des Access Points (ESP32-CAM)
- Subnetzmaske: 255.255.255.0 – ermöglicht Kommunikation zwischen Geräten im Bereich 192.168.4.x

Für die WebSocket-Kommunikation wurde der Port 8080 gewählt.

```
const char *ssid = "Carybot";  
const char *password = "123456789";  
  
WebSocketsServer websocket(8080);  
  
IPAddress local_IP(192, 168, 4, 3);  
IPAddress gateway(192, 168, 4, 1);  
IPAddress subnet(255, 255, 255, 0);
```

Abbildung 19: Netzwerkkonfiguration Client

Quelle: eigene Abbildung

In der Setup Funktion des Programmes wird dann überprüft, ob die IP-Konfiguration erfolgreich abgeschlossen wurde. Ansonsten kommt es zu einer Fehlermeldung und das Setup wird abgebrochen. Danach wird versucht, sich mit dem WLAN-Netzwerk zu verbinden. Wenn sich der ESP32 erfolgreich mit dem WLAN verbunden hat, wird eine Nachricht und die IP-Adresse des ESP32 in der seriellen Konsole ausgegeben. Danach wird die WebSocket Konfiguration noch gestartet. Es wird definiert, dass die Funktion onWebSocketEvent aufgerufen wird, wenn Events über den WebSocket registriert werden. In der loop Funktion wird dann noch ständig überprüft, ob neue Events am WebSocket registriert werden.

```

if (!WiFi.config(local_IP, gateway, subnet))
{
    Serial.println("Fehler bei der IP-Konfiguration");
    return;
}

WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.println(".");
}
Serial.println("WLAN verbunden");
Serial.print("IP-Adresse: ");
Serial.println(WiFi.localIP());

websocket.begin();
websocket.onEvent(onWebSocketEvent);

void loop()
{
    websocket.loop();
}

```

Abbildung 20: Setup ESP32

Quelle: eigene Abbildung

#### 4.4.2 Message Handling

In der Funktion `onWebSocketEvent()` werden die WebSocket-Ereignisse verarbeitet. Sie wird aufgerufen, wenn sich ein WebSocket-Client verbindet, eine Nachricht sendet oder die Verbindung sich trennt. Der Parameter `num` steht für die ID des Clients, der das Event ausgelöst hat. Der Parameter `type` gibt die Art des WebSocket-Event an. Der Parameter `payload` sind die empfangenen Daten. Der Parameter `length` steht für die Größe des payload-Arrays. In der Funktion werden die Events mit einem switch-case verarbeitet

Übersicht der WebSocket-Ereignisse:

Ereignistyp	Beschreibung	Verarbeitung
WStype_CONNECTED	Ein neuer Client verbindet sich.	Ausgabe der Client-ID & IP-Adresse in der Konsole
WStype_TEXT	Eine Textnachricht wird empfangen.	Übergabe an <code>handleWebSocketMessage()</code>
WStype_BIN	Binärdaten werden empfangen.	Nachricht in der Konsole (wird nicht verarbeitet)
WStype_DISCONNECTED	Ein Client trennt die Verbindung.	Meldung mit der Client-ID in der Konsole.

Tabelle 3: Übersicht der WebSocket-Ereignisse



```

void onWebSocketEvent(uint8_t num, WStype_t type, uint8_t *payload, size_t length)
{
    switch (type)
    {
        case WStype_TEXT:
            handleWebSocketMessage(num, payload, length);
            break;

        case WStype_BIN:
            Serial.println("Binärdaten empfangen (nicht unterstützt)");
            break;

        case WStype_DISCONNECTED:
            Serial.printf("[WS] Client %u disconnected.\n", num);
            break;

        case WStype_CONNECTED:
            IPAddress ip = websocket.remoteIP(num);
            Serial.printf("[WS] Client %u connected from %s.\n", num, ip.toString().c_str());
            break;
    }
}

```

Abbildung 21: onWebSocketEvent()-Funktion

Quelle: eigene Abbildung

Die Funktion handleWebSocketMessage() verarbeitet die WebSocket-Nachricht, die als JSON-Objekte gesendet werden. Die Parameter sind wieder die Client-ID, die empfangene Nachricht und die Länge der empfangenen Nachricht.

Zuerst wird die empfangene Nachricht (payload) in einen String konvertiert. Diese wird dann in der seriellen Konsole ausgegeben. Danach wird ein JSON-Dokument mit max 200 Bytes erstellt. Die empfangene Nachricht wird dann mit deserializeJson() geparkt. Falls das Parsen erfolgreich war, wird die JSON-Nachricht verarbeitet.

Wenn die JSON-Nachricht den Namen robot\_direction enthält, wird die Richtung mit der Funktion stringToDirection() in eine eigene Variable umgewandelt. Auch die mitgesendete Variable speed wird ebenfalls in eine eigene Variable gespeichert.

```

{
  "robot_direction": "up",
  "speed": 100
}

```

Abbildung 22: JSON-Beispiel für Steuerung

Quelle: eigene Abbildung

---

Wenn die JSON-Nachricht den Namen `camera_position` enthält, wird der Wert der Nachricht in die Variable `camera_pos` gespeichert und die Funktion `cam_turn()` aufgerufen.

```
{  
  "camera_position": 45  
}
```

Abbildung 23: JSON-Beispiel für  
Kamerasteuerung

Quelle: eigene Abbildung

Wenn die JSON-Nachricht den Namen `light_status` enthält, wird der Wert der Nachricht in die boolesche Variable `light_status` gespeichert. Dieser Variable wird dann in die numerische Variable `light_st` umgewandelt (1 = an, 0 = aus). Wenn `light_st` eine 1 ist, wird die Funktion `lights_on()` aufgerufen, ansonsten wird die Funktion `lights_off()` aufgerufen.

```
{  
  "light_status": true  
}
```

Abbildung 24: JSON-Beispiel  
für Lichtsteuerung

Quelle: eigene Abbildung

```

void handleWebSocketMessage(uint8_t num, uint8_t *payload, size_t length)
{
    String message = String((char *)payload).substring(0, length);
    Serial.println("WebSocket-Nachricht empfangen: " + message);

    StaticJsonDocument<200> jsonDoc;
    DeserializationError error = deserializeJson(jsonDoc, message);

    if (!error)
    {
        if (jsonDoc.containsKey("robot_direction"))
        {
            const char *robot_direction = jsonDoc["robot_direction"];
            if (robot_direction)
            {
                dir = stringToDirection(robot_direction);
            }
            speed = jsonDoc["speed"].as<String>();
        }
        else if (jsonDoc.containsKey("camera_position"))
        {
            const char *camera_position = jsonDoc["camera_position"];
            if (camera_position)
            {
                camera_pos = atoi(camera_position);
                cam_turn();
            }
        }
        else if (jsonDoc.containsKey("light_status"))
        {
            bool light_status = jsonDoc["light_status"];
            light_st = light_status ? 1 : 0;

            if (light_st == 1)
            {
                lights_on();
            }
            else
            {
                lights_off();
            }
        }
    }
}

```

Abbildung 25: handleWebSocketMessage()-Funktion

Quelle: eigene Abbildung

---

## 4.5 Kamera

## 4.6 Kamera

### 4.6.1 Kamera Setup

Um die Kamera programmieren zu können, muss zunächst das richtige Board AI Thinker ESP32-CAM ausgewählt werden. Danach müssen die ESP32-Kamera-Treiber mit der Bibliothek `esp_camera.h` inkludiert werden. Dann muss das passende ESP32-CAM-Modell festgelegt werden. Da wir uns für das Ai-Thinker Modell entschieden haben, muss diese nun definiert werden. Falls ein anderes Modell genutzt wird, muss es entsprechend angepasst werden.

Als nächstes werden die GPIO-Pins der ESP32-CAM für die Kamera OV2640 konfiguriert. Diese Zuordnung ist spezifisch für das Ai-Thinker-Modell und muss für jedes Modell individuell angepasst werden.

```
#include "esp_camera.h"

#define CAMERA_MODEL_AI_THINKER

#if defined(CAMERA_MODEL_AI_THINKER)
#define PWDN_GPIO_NUM 32 // Kamera-Power-Down
#define RESET_GPIO_NUM -1 // Reset-Pin
#define XCLK_GPIO_NUM 0 // Externer Taktgeber
#define SIOD_GPIO_NUM 26 // I2C-Daten (SDA)
#define SIOC_GPIO_NUM 27 // I2C-Takt (SCL)

#define Y9_GPIO_NUM 35 // Kamera-Datenbit 9
#define Y8_GPIO_NUM 34 // Kamera-Datenbit 8
#define Y7_GPIO_NUM 39 // Kamera-Datenbit 7
#define Y6_GPIO_NUM 36 // Kamera-Datenbit 6
#define Y5_GPIO_NUM 21 // Kamera-Datenbit 5
#define Y4_GPIO_NUM 19 // Kamera-Datenbit 4
#define Y3_GPIO_NUM 18 // Kamera-Datenbit 3
#define Y2_GPIO_NUM 5 // Kamera-Datenbit 2
#define VSYNC_GPIO_NUM 25 // Vertikale Synchronisation
#define HREF_GPIO_NUM 23 // Horizontale Synchronisation
#define PCLK_GPIO_NUM 22 // Pixeltakt
#endif
```

Abbildung 26: Kamera-GPIO Konfiguration

Quelle: eigene Abbildung

Als nächstes muss die ESP32-CAM mit der ESP-IDF `esp_camera` Bibliothek konfiguriert und initialisiert werden. Als erstes muss mit `camera_config_t` eine Struktur definiert werden, mit der verschiedene Parameter und Eigenschaften wie GPIO-Pins, Bildgröße und Qualität festgelegt werden können.

LEDC-Kanal und Timer werden für das Taktsignal benötigt, um die Kamera zu betreiben.

```
camera_config_t config;
config.ledc_channel = LEDC_CHANNEL_0;
config.ledc_timer = LEDC_TIMER_0;
config.pin_d0 = Y2_GPIO_NUM;      // Kamera-Datenbit 2
config.pin_d1 = Y3_GPIO_NUM;      // Kamera-Datenbit 3
config.pin_d2 = Y4_GPIO_NUM;      // Kamera-Datenbit 4
config.pin_d3 = Y5_GPIO_NUM;      // Kamera-Datenbit 5
config.pin_d4 = Y6_GPIO_NUM;      // Kamera-Datenbit 6
config.pin_d5 = Y7_GPIO_NUM;      // Kamera-Datenbit 7
config.pin_d6 = Y8_GPIO_NUM;      // Kamera-Datenbit 8
config.pin_d7 = Y9_GPIO_NUM;      // Kamera-Datenbit 9
config.pin_xclk = XCLK_GPIO_NUM;  // Externer Taktgeber (24 MHz)
config.pin_pclk = PCLK_GPIO_NUM;  // Pixeltakt
config.pin_vsync = VSYNC_GPIO_NUM; // Vertikale Synchronisation
config.pin_href = HREF_GPIO_NUM;  // Horizontale Synchronisation
config.pin_sccb_sda = SIOD_GPIO_NUM; // I2C-Datenleitung (SDA)
config.pin_sccb_scl = SIOC_GPIO_NUM; // I2C-Taktleitung (SCL)
config.pin_pwdn = PWDN_GPIO_NUM;  // Power-Down
config.pin_reset = RESET_GPIO_NUM; // Reset-Pin
config.xclk_freq_hz = 24000000;    // Taktfrequenz
config.pixel_format = PIXFORMAT_JPEG;
config.frame_size = FRAMESIZE_QVGA;
config.jpeg_quality = 12;
config.fb_count = 3;

esp_err_t err = esp_camera_init(&config);
if (err != ESP_OK) {
    Serial.printf("Kamera-Init fehlgeschlagen mit Fehler 0x%x", err);
    return;
}
```

Abbildung 27: Kamera-Initialisierung

Quelle: eigene Abbildung

## 4.7 Videoübertragung

In unserem Projekt werden die Live-Bilder per WebSocket an den Webserver gesendet. Um dies umzusetzen, wird zuerst eine Webserver-Route benötigt. Dazu wird ei-

---

ne http-GET-Anfrage für die Hauptseite (“/“) definiert. Im Code wird ein JavaScript Skript benutzt, um eine Websocket-Verbindung herzustellen. Die IP-Adresse wird automatisch durch `windows.location.hostname` erkannt. Port 81 wird für das WebSocket-Streaming verwendet. Wenn die ESP32-CAM ein neues Bild als WebSocket-Nachricht versendet, wird es als JPEG-Blob gespeichert. Danach wird ein temporär URL-Link erstellt. Das Bild wird schlussendlich in einem `<img>`-Tag mit der id `stream` angezeigt.

Die WebSocket Verbindung wird die ganze Zeit überwacht. In der Konsole wird ausgegeben, wenn die Websocket-Verbindung aktiv ist. Falls die Verbindung abbrechen sollte, wird nach 5 Sekunden automatisch ein erneuter Verbindungsversuch gestartet. Zum Schluss wird der HTML-Code mit HTTP-Status 200 (OK) an den Browser gesendet.

```

server.on("/", HTTP_GET, [(AsyncWebServerRequest *request) {
  String html = R"rawliteral(
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>ESP32-CAM WebSocket Stream</title>
      <script>
        let websocket;
        function connectWebSocket() {
          websocket = new WebSocket('ws://' + window.location.hostname + ':81');

          websocket.onmessage = function(event) {
            const blob = new Blob([event.data], { type: 'image/jpeg' });
            const url = URL.createObjectURL(blob);
            const img = document.getElementById('stream');
            img.src = url;
          };

          websocket.onopen = function() {
            console.log('WebSocket verbunden');
          };

          websocket.onclose = function() {
            console.log('WebSocket getrennt. Erneuter Versuch in 5 Sekunden...');
            setTimeout(connectWebSocket, 5000);
          };
        }
        connectWebSocket();
      </script>
    </head>
    <body>
      <h1>ESP32-CAM WebSocket Stream</h1>
      <img id="stream" alt="Live Stream" style="width: 100%; max-width: 640px;" />
    </body>
  </html>
)rawliteral";
  request->send(100, "text/html", html);
});

server.begin();

```

Abbildung 28: Videoübertragung über WebSocket

Quelle: eigene Abbildung

## 4.8 Website

### 4.8.1 Implementierung der Steuerung

#### Steuerkreuz

Um den Roboter überhaupt steuern zu können, wird ein Steuerkreuz implementiert. Das Steuerkreuz befindet sich mittig am rechten Bildschirmrand. Das Steuerkreuz besteht aus fünf Tasten, nämlich: Vorwärts (UP), (DOWN ), links (LEFT), rechts (RIGHT) und in der Mitte Stop (HALT).



Abbildung 29: Steuerkreuz auf der Website

Quelle: eigene Abbildung

Im HTML-Code wird das Steuerkreuz als HTML `<div>`-Tag im body Bereich definiert. Für die Formatierung wird die CSS-Klasse `dpad-container` verwendet. Die einzelnen Richtungstasten des Steuerkreuze werden auch als HTML `<div>`-Tags definiert. Jede Taste wird mit der CSS-Klasse `button` und der jeweiligen Zusatzklasse `up/down/left/right/center` formatiert. Beim jeweiligen Tastendruck wird außerdem immer das Attribut `data-direction` mit der jeweiligen Richtung belegt.

```
<div class="dpad-container">
  <div class="button up" data-direction="up">↑</div>
  <div class="button left" data-direction="left">←</div>
  <div class="button center" data-direction="halt"></div>
  <div class="button right" data-direction="right">→</div>
  <div class="button down" data-direction="down">↓</div>
</div>
```

Abbildung 30: Steuerkreuz HTML-Code

Quelle: eigene Abbildung

Die CSS-Klasse `dpad-container` ist das übergeordnete Element, welches die Steuerkreuztasten enthalten. Es wird als CSS Grid mit einer 3x3 Struktur definiert. Es gibt Lücken (.) an den Ecken, damit das Layout wie ein Steuerkreuz aussieht. Der Abstand



---

zwischen den Tasten wird mit 10px festgelegt. Der Container hat eine feste Breite von 120px und ist vertikal so wie horizontal zentriert.

Die CSS-Klasse `button` sorgt für eine einheitliche Gestaltung der Tasten des Steuerkreuzes. Jede Taste hat eine Größe von 60x60px. Eine Flexbox wird verwendet, um die Tasten jeweils mittig in den einzelnen Positionen des Grids zu positionieren. Jede Taste hat einen dunkelgrauen Hintergrund, eine weiße Textfarbe, einen 2px dicken dunklen Rahmen und abgerundete Ecken. Die Optionen `-webkit-tap-highlight-color: transparent` und `user-select: none` sorgen dafür, dass auf mobilen Geräten der Text in den Tasten nicht blau markiert werden kann, damit die Steuerung nicht blockiert wird.

```
.dpad-container {
  display: grid;
  grid-template-areas:
    ". up ."
    "left center right"
    ". down .";
  gap: 10px;
  width: 120px;
  margin: 50px auto;
  position: absolute;
  right: 100px;
  top: 45%;
  transform: translateY(-50%);
}

.button {
  width: 60px;
  height: 60px;
  display: flex;
  align-items: center;
  justify-content: center;
  background-color: #333;
  color: white;
  font-size: 18px;
  cursor: pointer;
  border: 2px solid #444;
  border-radius: 5px;

  -webkit-tap-highlight-color: transparent;
  user-select: none;
}
```

Abbildung 31: CSS-Klassen `.dpad-container` und `.button`

Quelle: eigene Abbildung

---

Die fünf Richtungsklassen sorgen dafür, dass die Tasten des Steuerkreuzes in den zuvor definierten Grid-Bereichen zugewiesen und richtig platziert werden. Die mittlere Taste bekommt außerdem eine leicht hellere Farbe zugewiesen, um ihn optisch von den anderen abzuheben.

Außerdem bekommen alle Tasten einen Hover-Effekt, damit sie etwas heller werden, wenn eine Taste gedrückt wird.

```
.up {
|   grid-area: up;
| }

.down {
|   grid-area: down;
| }

.left {
|   grid-area: left;
| }

.right {
|   grid-area: right;
| }

.center {
|   grid-area: center;
|   background-color: #555;
| }

.button:hover {
|   background-color: #666;
| }
```

Abbildung 32: CSS-Richtungsklassen

Quelle: eigene Abbildung

Im JavaScript-Code sorgen zwei EventListener dafür, dass die Elemente auf der Seite nicht ziehbar und verschiebbar sind und dass das Rechtsklick-Menü nicht geöffnet werden kann. Diese Maßnahmen verhindern unerwünschte Benutzerinteraktionen und sorgen für eine reibungslose Bedienung.

```

document.addEventListener('dragstart', event => {
  |   event.preventDefault();
});

document.addEventListener('contextmenu', event => {
  |   event.preventDefault();
});

```

Abbildung 33: <hier gute Bildbeschreibung einfügen>

Quelle: eigene Abbildung

Vier weitere EventListener kümmern sich um die Eingabe des Steuerkreuzes. Es wird zuerst jeder mousedown (Linksklick) bzw. ein touchstart (klick auf mobile Geräte) abgefangen. Danach wird überprüft, ob das geklickte Element innerhalb eines .button Elements (Steuerkreuztaste) liegt. Falls ja, wird die Richtung aus dem data-direction-Attribut gelesen und an die send() Funktion übergeben. Wenn ein mouseup (Maus loslassen) oder ein touchend (Finger loslassen) abgefangen wird, wird die Funktion stop() aufgerufen.

```

document.addEventListener('mousedown', (event) => {
  const target = event.target.closest('.button');
  if (target) {
    const direction = target.dataset.direction;
    if (direction) {
      send(target.dataset.direction);
    }
  }
});

document.addEventListener('mouseup', (event) => {
  const target = event.target.closest('.button');
  if (target) {
    stop();
  }
});

document.addEventListener('touchstart', (event) => {
  const target = event.target.closest('.button');
  if (target) {
    const direction = target.dataset.direction;
    if (direction) {
      send(direction);
    }
  }
});

document.addEventListener('touchend', (event) => {
  const target = event.target.closest('.button');
  if (target) {
    stop();
  }
});

```

Abbildung 34: <hier gute Bildbeschreibung einfügen>

Quelle: eigene Abbildung

---

Die JavaScript Funktion `send()` kümmert sich um das Senden der Steuerungsbefehle. Zuerst wird überprüft, ob die aktuelle Richtung nicht bereits die gewünschte Richtung ist. Falls ja, wird ein JSON-Objekt erstellt, welche die Richtung sowie die Geschwindigkeit enthält. Zu debug-Zwecken wird die Richtung sowie die Geschwindigkeit in der Webkonsole ausgegeben. Danach wird überprüft, ob die WebSocket-Verbindung zur ESP32-CAM offen ist. Wenn ja, wird die Nachricht auch für debug-Zwecken an die ESP32-CAM gesendet, ansonsten wird eine Fehlermeldung ausgegeben. Dasselbe geschieht für die WebSocket-Verbindung zum ESP32. Dort werden jedoch die Steuerbefehle weiterverarbeitet.

```
let currentdir_robot = Directions.HALT;

function send(direction) {
  if (currentdir_robot !== direction) {
    currentdir_robot = direction;

    const message = JSON.stringify({
      robot_direction: currentdir_robot,
      speed: speed
    });

    console.log("Direction: " + direction + " Speed: " + speed);

    if (websocket_cam.readyState === WebSocket.OPEN) {
      websocket_cam.send(message);
      console.log("An CAM gesendet");
    } else {
      console.error("Senden fehlgeschlagen. WebSocket Cam is not open");
    }

    if (websocket_carybot.readyState === WebSocket.OPEN) {
      websocket_carybot.send(message);
      console.log("An Carybot gesendet");
    } else {
      console.error("Senden fehlgeschlagen. WebSocket Carybot is not open");
    }
  }
}
```

Abbildung 35: `send()`-Funktion

Quelle: eigene Abbildung

Die JavaScript Funktion `stop()` sorgt dafür, dass der Roboter anhält, wenn eine Taste des Steuerkreuzes losgelassen wird. Zuerst wird überprüft, ob die aktuelle Richtung nicht bereits `HALT` ist, ansonsten wird sie daraufgesetzt. Danach wird ein Stopp-Befehl als JSON-Nachricht vorbereitet. Zu debug-Zwecken wird wieder ein `halt` in der Webkonsole ausgegeben. Danach wird der Stopp-Befehl wieder an die WebSocket-Verbindung zur ESP32-CAM für debug-Zwecke gesendet. Danach wird sie auch an die WebSocket-Verbindung zum ESP32 gesendet, wo dieser weiterverarbeitet wird.

```
function stop() {
  if (currentdir_robot !== Directions.HALT) {
    currentdir_robot = Directions.HALT;

    const message = JSON.stringify({
      robot_direction: Directions.HALT,
      speed: speed
    });

    console.log('Message sent: halt');

    if (websocket_cam.readyState === WebSocket.OPEN) {
      websocket_cam.send(message);
    } else {
      console.error("Senden fehlgeschlagen. WebSocket Cam is not open");
    }

    if (websocket_carybot.readyState === WebSocket.OPEN) {
      websocket_carybot.send(message);
    } else {
      console.error("Senden fehlgeschlagen. WebSocket Carybot is not open");
    }
  }
}
```

Abbildung 36: stop()-Funktion

Quelle: eigene Abbildung

## Geschwindigkeitseinstellung

Um die Fortbewegungsgeschwindigkeit des Roboters kontrollieren zu können, gibt es auf der linken Seite neben dem Steuerkreuz einen vertikalen Geschwindigkeitsslider, mit der die Geschwindigkeit der Motoren gesteuert werden kann. Wenn der Slider nach oben gezogen wird, beschleunigt der Roboter, wenn der Slider nach unten gezogen wird, wird die Geschwindigkeit verlangsamt.



Abbildung 37: Geschwindigkeitssteuerung auf der Website

Quelle: eigene Abbildung

---

Im HTML-Code wird mit der CSS-Klasse `slider-container` ein Bereich im Hauptbereich definiert. Der Slider wird als HTML `<input>`-Tag mit dem type `range` von 0 bis 100 definiert. Für das Design wird die CSS-Klasse `slider` verwendet und beim Verschieben des Sliders wird die JavaScript Funktion `speed_change()` mit der aktuellen Position des Sliders aufgerufen.

```
<div class="slider-container">
  <input type="range" min="0" max="100" value="50" class="slider" id="rangeSlider"
    oninput="speed_change(this.value)">
</div>
```

Abbildung 38: Geschwindigkeitssteuerung HTML-Code

Quelle: eigene Abbildung

In der CSS-Klasse `slider-container` wird ein Bereich mit der Größe 50x220 Pixel definiert, in dem der Slider angezeigt werden soll. In `slider` wird definiert, dass der Slider vertikal und nicht horizontal angezeigt wird. Außerdem wird die Richtung auf `Right to Left` gesetzt, damit die aktuelle Position richtig erhöht bzw. vermindert bei Auf- und Niederschieben des Sliders wird.

```
.slider-container {
  display: grid;
  width: 50px;
  position: absolute;
  right: 220px;
  top: 35%;
}

.slider {
  width: 80%;
  height: 220px;
  margin-top: 10px;
  writing-mode: vertical-lr;
  direction: rtl;
  vertical-align: middle;
}
```

Abbildung 39: CSS-Klassen für die Geschwindigkeitssteuerung

Quelle: eigene Abbildung

Im JavaScript-Code wird die aktuelle Position des Sliders in die eigene Variable `speed`

gespeichert. Die Geschwindigkeit wird immer bei einem Steuerbefehl des Steuerkreuzes mitübertragen. (siehe Steuerung)

```
var speed = 50;

function speed_change(input_speed) {
    speed = input_speed;
}
```

Abbildung 40: speed-change()-Funktion

Quelle: eigene Abbildung

## Kamerasteuerung

Um mit der Videoübertragung besser Objekte und Hindernisse zu erkennen, ist die Kamera leicht schwenkbar. Um nun die Kamera auf unserer Website bewegen zu können, gibt es auf der linken Seite einen horizontalen Slider, mit der die Kamera ein Stück links und rechts geschwenkt werden kann.



Abbildung 41: Kamerasteuerung auf der Website

Quelle: eigene Abbildung

Im HTML-Code wird ein mit der CSS-Klasse camera-container ein Bereich im Hauptbereich definiert. Der Slider wird als HTML <input>-Tag mit dem type range von 0 bis 180 definiert. Für das Design wird die CSS-Klasse cam\_slider verwendet und beim Verschieben des Sliders wird die JavaScript Funktion camera\_change() mit der aktuellen Position des Sliders aufgerufen.

```
<div class="camera-container">
  <input type="range" min="0" max="180" value="90" class="cam_slider" id="cameraSlider"
  |   oninput="camera_change(this.value)">
</div>
```

Abbildung 42: Kamerasteuerung HTML-Code

Quelle: eigene Abbildung

In der CSS-Klasse camera-container wird ein Bereich erstellt, in dem der Slider angezeigt werden soll. In cam\_slider wird die Slidespur mit einer Breite von 150 Pixel und einer Höhe von 10 Pixel definiert. Der Hintergrund ist hellgrau und die Ecken werden leicht abgerundet. In webkit-slider-thumb wird der Sliderknopf mit 20x20 Pixel definiert. Der Hintergrund ist dunkelgrau mit einem 2px breiten, dunkleren Rand.

```
.camera-container {
  display: grid;
  position: absolute;
  top: 50%;
  left: 50px;
}

.cam_slider {
  -webkit-appearance: none;
  appearance: none;
  width: 150px;
  height: 10px;
  background: #ddd;
  outline: none;
  border-radius: 5px;
}

.cam_slider::-webkit-slider-thumb {
  -webkit-appearance: none;
  appearance: none;
  width: 20px;
  height: 20px;
  background: #333;
  cursor: pointer;
  border: 2px solid #444;
}
```

Abbildung 43: CSS-Klassen für die Kamerasteuerung

Quelle: eigene Abbildung

Im JavaScript-Code wird die aktuelle Position des Sliders verarbeitet. Dazu wird die Position in eine JSON-Nachricht gespeichert. Zu debug-Zwecken wird die Position noch in



---

der Website Konsole ausgegeben. Wenn der WebSocket Verbindung zum ESP32 offen ist, wird die aktuelle Position übermittelt.

```
function camera_change(input_position) {  
  const message = JSON.stringify({  
    camera_position: input_position  
  });  
  
  console.log("Camera Position: " + input_position);  
  
  if(websocket_carybot.readyState === WebSocket.OPEN){  
    websocket_carybot.send(message);  
  }  
}
```

Abbildung 44: camera-change()-Funktion

Quelle: eigene Abbildung

## Fernlicht

Die Funktion des Fernlichts dient bei unserem Roboter zur Beleuchtung bei schlechter Sicht bzw. Dunkelheit. Wenn eingeschalten, leuchtet auf der Vorder- sowie auf der Hinterseite des Roboters ein Scheinwerfer die Umgebung aus.

Um das Fernlicht auf unserer Website ein- bzw. auszuschalten gibt es ein eigenes Fernlicht-Icon rechts über dem Steuerkreuz. Im ausgeschalteten Zustand ist das Icon ausgegraut. Im aktiven Zustand ist das Scheinwerfer Symbol im Icon blau und das Icon hat einen blassgrünen Hintergrund.



Abbildung 45: Fernlicht-Icon auf der Website

Quelle: eigene Abbildung

Im HTML-Code wird das Icon als HTML `<img>`-Tag eingefügt. Als Quelle wird die im SPIFFS hochgeladene `Fernlicht.svg` Grafik verwendet. Für das Design wird die CSS-Klasse `fernlicht-icon` verwendet und beim `onclick` Event wird die JavaScript Funktion `togglelight()` aufgerufen.

```

```

Abbildung 46: HTML-Code für Fernlicht

Quelle: eigene Abbildung

In der CSS-Klasse `fernlicht-icon` wird das Icon 20% vom oberen Rand und 2% von rechten Rand mit einer Größe von 50x50 Pixel positioniert. Es wird auf der 2 z-Ebene angezeigt und der nicht sichtbare Hintergrund wird transparent angezeigt. Beim Ändern der Farbe wird eine sanfte Animation über 0,3 Sekunden angezeigt.

Die `grayscale` Klasse wandelt das Icon in die ausgegraute Darstellung um, um anzuzeigen, dass das Fernlicht deaktiviert ist. Die `active-light` Klasse ändert die Hintergrundfarbe zu einem transparenten grünen Ton, um anzuzeigen, dass das Fernlicht aktiv ist.

```
.fernlicht-icon {
  position: fixed;
  top: 20%;
  right: 2%;
  width: 50px;
  height: 50px;
  cursor: pointer;
  z-index: 2;
  background-color: transparent;
  transition: filter 0.3s ease;
}

.grayscale {
  filter: grayscale(100%)
}

.active-light {
  background-color: rgba(0, 255, 0, 0.3);
  border-radius: 10px;
}
```

Abbildung 47: CSS-Klassen für das Fernlicht

Quelle: eigene Abbildung

Im JavaScript Code wird bei jedem onclick Event die Variable light getoggled. Wenn die Variable light true ist, wird die CSS-Klasse active-light angewendet, ansonsten die CSS-Klasse grayscale. Anschließend wird eine JSON-Nachricht mit dem aktuellen Zustand der light Variable erstellt. Wenn die WebSocket Verbindung zum ESP32 offen ist, wird die JSON-Nachricht versendet, ansonsten wird eine Fehlermeldung ausgegeben.

```

var light = false;

function togglelight() {
  light = !light;
  const fernlichtIcon = document.getElementById("Fernlicht");

  if(light) {
    fernlichtIcon.classList.remove("grayscale");
    fernlichtIcon.classList.add("active-light")
  } else {
    fernlichtIcon.classList.add("grayscale");
    fernlichtIcon.classList.remove("active-light");
  }

  const message = JSON.stringify({
    light_status: light
  });

  if(websocket_carybot.readyState === WebSocket.OPEN) {
    websocket_carybot.send(message);
  } else {
    console.error("Senden fehlgeschlagen. WebSocket Carybot is not open");
  }
}

```

Abbildung 48: togglelight()-Funktion

Quelle: eigene Abbildung

#### 4.8.2 Echtzeit-Videoanzeige

Die Videoübertragung dient dazu, den Roboter über größere Entfernungen autonom steuern zu können. Mit der Videoübertragung werden Objekte und Hindernisse erkennbar und kann diese somit ausweichen.

Die Videoübertragung wird über der ganzen Website im Hintergrund angezeigt.



Abbildung 49: Videoübertragung auf der Website

Quelle: eigene Abbildung

Im HTML-Code wird die Videoübertragung als HTML `<img>`-Tag mit der id `dynamicimage` im Hauptbereich definiert.

```
<img id="dynamicimage" alt="Video Stream" />
```

Abbildung 50: Videoübertragung HTML-Code

Quelle: eigene Abbildung

Im CSS-Code wird nun das Element mit der id `dynamicimage` formatiert. Es wird definiert, dass die Größe 100% der Breite sowie 100% der Höhe einnimmt. Die Position wird dabei auf `absolute` gesetzt.

```
#dynamicimage {  
    width: 100%;  
    height: 100%;  
    position: absolute;  
}
```

Abbildung 51: CSS-Formatierung für  
`dynamicimage`

Quelle: eigene Abbildung

---

Im JavaScript-Code wird die WebSocket-Verbindung zur ESP32-CAM gehandelt. In der Funktion `connectWebSocket_cam()` wird zuerst ein `WebSocket` mit der eigenen IP-Adresse erstellt (da Webserver auf ESP32-CAM gehostet wird) und dem Port 81 erstellt. Wenn nun eine Nachricht auf der WebSoccke-Verbindung ankommt (siehe Videoübertragung ESP32-CAM) wird diese extrahiert. Das mitgesendete BLOB wird in einem neuem BLOB als jpeg-Bild gespeichert. Danach wird eine URL (Adresse) erstellt, die zu diesem BLOB führt. Dann wird eine Variable für die Videoübertragung im HTML `<img>`-Tag erstellt. Die Source dieses Bildes wird dann mit dem neuen, erhaltenen Bild ersetzt. Da die ESP32-CAM die ganze Zeit neue Blobs sendet, wird das Bild die ganze Zeit ersetzt, was dazu führt, dass es aussieht, als wäre es ein Video.

Wenn sich die ESP32-CAM verbindet, wird zu Debug-Zwecken eine Nachricht in der Webkonsole ausgegeben.

Wenn die Verbindung zur ESP32-CAM verloren geht, wird wieder zu Debug-Zwecke eine Nachricht in der Webkonsole ausgegeben und ein neuer Verbindungsversuch wird nach 5 Sekunden gestartet.

```
let websocket_cam;

function connectWebSocket_cam() {
    websocket_cam = new WebSocket('ws://' + window.location.hostname + ':81'); // WebSocket-Verbindung zur ESP32 CAM

    websocket_cam.onmessage = function (event) {
        const blob = new Blob([event.data], { type: 'image/jpeg' });
        const url = URL.createObjectURL(blob);
        const img = document.getElementById('dynamicimage');
        img.src = url;
    };

    websocket_cam.onopen = function () {
        console.log('WebSocket cam verbunden');
    };

    websocket_cam.onclose = function () {
        console.log('WebSocket Cam getrennt. Erneuter Versuch in 5 Sekunden...');
        setTimeout(connectWebSocket_cam, 5000); // Versuchen, die Verbindung erneut herzustellen
    };
}

connectWebSocket_cam();
```

Abbildung 52: Videoübertragung im JavaScript-Code

Quelle: eigene Abbildung

---

### 4.8.3 Anzeige von Sensordaten und Verbindungsstatus

#### Sensordaten

In der linken oberen Ecke der Website befindet sich das Menü-icon. Ein Klick darauf ruft die Sidebar für die Daten auf. Die Sidebar klappt sich am linken Bildschirmrand auf. Darin kann ist der aktuelle Akkustand sowie das aktuelle aufliegende Gewicht auslesbar. Um das Menü wieder schließen zu können, ist ein erneuter Klick auf das Icon erforderlich und die Sidebar wird wieder zugeklappt.



Abbildung 53: Menü-Icon auf der Website

Quelle: eigene Abbildung



Abbildung 54: Sidebar mit Sensordaten

Quelle: eigene Abbildung

---

Im HTML-Code wird im Body-Bereich das Icon als HTML `<img>`-Tag eingefügt. Als Quelle wird die im SPIFFS hochgeladene Grafik `menu-icon.svg` Grafik verwendet. Für das Design wird die CSS-Klasse `menu-icon` verwendet und beim onclick Event wird die JavaScript Funktion `togglemenu()` aufgerufen.

Im HTML `<div>`-Tag werden die einzelnen Daten festgelegt und mit der CSS-Klasse `mymenu` formatiert. Das Gewicht und der Akkustand werden als HTML `<p>`-Tag angezeigt und mit der CSS-Klasse `daten` formatiert.

```
<div id="menu" class="mymenu">
  <p id="Gewicht" class="daten">Gewicht: - kg</p>
  <p id="Akkustand" class="daten">Akkustand: - %</p>
</div>

```

Abbildung 55: Sensordaten HTML-Code

Quelle: eigene Abbildung

In der CSS-Klasse `mymenu` wird die sidebar formatiert. Das Menü ist fixiert und nimmt 100% der Höhe ein. Die Anfängliche ist 0, da es versteckt bzw. eingeklappt ist. Die Hintergrundfarbe ist schwarz und es befindet sich auf der z-Ebene 1 um den Hauptinhalt überdecken zu können. Beim Ein- und Ausblenden dauert 0.5 Sekunden, um einen sanften Übergang zu ermöglichen.

Der Hauptinhalt `#main` hat eine Animation, damit er beim Öffnen des Menüs nach rechts verschoben werden kann.

Falls der Bildschirm kleiner als 450px Höhe hat, werden die Abstände der angezeigten Daten verkleinert. Das dient dazu, um besonders auf Handys die Ansicht zu optimieren.

Die CSS-Klasse `menu-icon` positioniert das Icon in der linken oberen Ecke mit einer 30x30px Format. `Cursor: pointer` sorgt dafür, dass das Icon anklickbar ist.



```
.mymenu {
  height: 100%;
  width: 0;
  position: fixed;
  z-index: 1;
  top: 0;
  left: 0;
  background-color: #111;
  overflow-x: hidden;
  padding-top: 60px;
  transition: 0.5s;
}

#main {
  transition: margin-left .5s;
  padding: 0;
  height: 100%;
}

@media screen and (max-height: 450px) {
  .mymenu {
    padding-top: 15px;
  }
}

.menu-icon {
  position: fixed;
  top: 10px;
  left: 10px;
  width: 30px;
  height: 30px;
  cursor: pointer;
  z-index: 2;
  background-color: transparent
}
```

Abbildung 56: CSS-Klassen für das Menü

Quelle: eigene Abbildung

Die CSS-Klasse `daten` formatiert die einzelnen Daten im Menü. Sie werden in der Schriftart Arial, Helvetica oder sans-serif angezeigt und in der Farbe Weiß, mit der Schriftgröße 1.2em (20% größer) mittig angezeigt.

```
.daten {
  font-family: Arial, Helvetica, sans-serif;
  text-align: center;
  color: ■ white;
  font-size: 1.2em;
}
```

Abbildung 57: CSS-Klasse .daten

Quelle: eigene Abbildung

In der JavaScript Funktion `connectWebSocket_carybot()` wird im `onmessage()`-event die übertragenen Daten verarbeitet. Für debug-Zwecken werden die angekommen Daten in der WebKonsole ausgegeben. Danach werden die JSON-Nachrichten extrahiert. Wenn die Nachricht den Akkustand beinhaltet, wird dieser im HTML `<p>`-Tag mit der id Akkustand im Menü angezeigt. Wenn die Nachricht das Gewicht beinhaltet, wird dieses im HTML `<p>`-Tag mit der id Gewicht im Menü angezeigt. Falls beim Umwandeln ein Fehler auftreten sollte, wird dieser in der Webkonsole ausgegeben.

```
websocket_carybot.onmessage = (event) => {
  console.log("Received:", event.data);

  try {
    const data = JSON.parse(event.data);
    if (data.battery) {
      document.getElementById('Akkustand').innerText = "Akkustand: " + data.battery + "%";
    }
    if (data.weight) {
      document.getElementById('Gewicht').innerText = "Gewicht: " + data.weight + "kg";
    }
  } catch (e) {
    console.error("Error parsing JSON:", e);
  }
};
```

Abbildung 58: JavaScript-Verarbeitung der Sensordaten

Quelle: eigene Abbildung

## Verbindungsstatus

In der WebSocket Status-Box wird der aktuelle Verbindungszustand zum ESP32 angezeigt. Wenn keine Verbindung vorhanden ist, ist die Box rot. Wenn die Verbindung erfolgt, wird die Box grün.



Abbildung 59: Statusbox auf der Website

Quelle: eigene Abbildung



Abbildung 60: Statusbox: Verbunden

Quelle: eigene Abbildung



Abbildung 61: Statusbox: Verbindung getrennt

Quelle: eigene Abbildung

Im HTML-Code wird die Status-Box als HTML `<span>`-Tag definiert. Für das Design wird die CSS-Klasse `status-box` kombiniert je nach Verbindungsstatus mit der Klasse `connected` oder `disconnected`.

```
<span id="connectionStatus" class="status-box disconnected">Disconnected</span>
```

Abbildung 62: Statusbox HTML-Code

Quelle: eigene Abbildung

In der CSS-Klasse `status-box` wird die grundlegende Box am linken oberen Bildschirmrand positioniert. Die Box liegt auf der zweiten z-Ebene. Die Schrift wird weiß definiert und wird fett dargestellt. Wenn eine Verbindung hergestellt wurde, wird der Hintergrund auf grün festgelegt (`status-box.connected`), ansonsten ist der Hintergrund rot

---

(status-box.disconnected).

```
.status-box {  
  top: 10px;  
  left: 50px;  
  z-index: 2;  
  position: fixed;  
  padding: 5px 10px;  
  margin-left: 10px;  
  border-radius: 5px;  
  font-weight: bold;  
  color: ■ white;  
}  
  
.status-box.connected {  
  background-color: ■ green;  
}  
  
.status-box.disconnected {  
  background-color: ■ red;  
}
```

Abbildung 63: CSS-Klassen für die Statusbox

Quelle: eigene Abbildung

Im JavaScript-Code werden die `onopen()` und `onclose()` Events der WebSocket Verbindung gehandelt. Bei einem Verbindungsaufbau (`onopen()`) wird die Status-Box grün und der Text wechselt zu `Connected`. Außerdem wird eine Nachricht für Debug-Zwecke ausgegeben. Bei einem Verbindungsabbruch wird die Status-Box wieder rot und der Text wechselt zu `Disconnected`. Es wird automatisch alle 5 Sekunden ein neuer Versuch zur Verbindungsaufbau gestartet.

```

websocket_carybot.onopen = function () {
    console.log("WebSocket Carybot verbunden");
    document.getElementById('connectionStatus').innerText = 'Connected';
    document.getElementById('connectionStatus').className = 'status-box connected';
}

websocket_carybot.onclose = function () {
    document.getElementById('connectionStatus').innerText = 'Disconnected';
    document.getElementById('connectionStatus').className = 'status-box disconnected';

    console.log('WebSocket Carybot getrennt. Erneuter Versuch in 5 Sekunden...');
    setTimeout(connectWebSocket_carybot, 5000); // Versuchen, die Verbindung erneut herzustellen
}

```

Abbildung 64: JavaScript Funktionen für den Verbindungsstatus

Quelle: eigene Abbildung

## 4.9 Herausforderungen und Optimierungen

### 4.9.1 Probleme bei der WebSocket Kommunikation

### 4.9.2 Latenz- und Performance Optimierungen

#### Kameraoptimierungen

Als Bildformat wird JPEG verwendet, um Speicherplatz zu sparen. Alternativ wäre RGB565 oder YUV422 möglich, aber diese benötigen mehr Speicher da sie nicht komprimiert sind und sind somit langsamer zum Übertragen.

Für die Bildauflösung wird QVGA (320x240 Pixel) definiert. Alternativ wären noch VGA (640x480), SVGA (800x600) oder UXGA (1600x1200) möglich, jedoch brauchen diese mehr Speicher und mehr Bandbreite und sind somit langsamer zum Übertragen.

Den Wert für die Bildqualität kann von 0 (= beste Qualität) bis 63 (= schlechteste Qualität) definiert werden. Wir definierten die Bildqualität mit 12, was ein guter Kompromiss zwischen Qualität und Speicherverbrauch ist.

Wir wählten für die Anzahl der Framebuffer 3. Die Anzahl sagt aus, wie viele Bilder gleichzeitig gespeichert werden können. Mehr Framebuffer erhöhen die Bildrate, benötigen jedoch mehr RAM.

Zum Schluss wird die Kamera mit den konfigurierten Einstellungen initialisiert. Falls bei der Initialisierung ein Fehler auftreten soll, wird dieser in der seriellen Konsole ausgegeben und das Setup abgebrochen.

---

#### **4.9.3 (Speicher- und Rechenleistungseinschränkungen des ESP32)**

#### **4.10 (Fazit und Ausblick)**

##### **4.10.1 (Mögliche Erweiterungen und Verbesserungen)**

---

# **5 Gehäuse**

## **5.1 Planung und Design**

## **5.2 Realisierung**

## **5.3 Materialliste**

---

# **6 Platine**

## **6.1 Grundschtaltung**

## **6.2 Circuit Board**

## **6.3 Fertiger Prototyp**



---

# **7 Kamera**

## **7.1 Kamera im Überblick**

## **7.2 Videoübertragung**

## **7.3 Kameraschwenkung**

### **7.3.1 Gehäuse**

### **7.3.2 Servomotor**

## **7.4 Code**

---

# **8 Sensoren**

## **8.1 Abstandsensor**

## **8.2 Gewichtsmessung**

### **8.2.1 Grundprinzip**

### **8.2.2 Schaltungsaufbau**

### **8.2.3 Code**

---

## 9 Entwicklungstools

### 9.1 Autodesk Fushion

### 9.2 Eagle

### 9.3 VS-Code

Visual Studio Code (VS-Code) ist eine kostenlose IDE (integrated development environment) entwickelt von Microsoft. VS-Code funktioniert auch auf anderen Betriebssystemen wie zum Beispiel Windows, Linux oder macOS. VS-Code unterstützt einen Großteil der Programmiersprachen und kann durch Extensions mit vielen nützlichen Features und Sprachen immer wieder erweitert werden.<sup>2</sup>

#### 9.3.1 Setup

Um ein Projekt in VS-Code erstellen zu können, müssen einige Schritte befolgt werden. Zuerst muss die IDE von <https://code.visualstudio.com/> für das jeweilig passende Betriebssystem heruntergeladen und installiert werden. Um Mikrocontroller wie ESPs oder Arduinos in VS-Code programmieren zu können, wird die PlatformIO IDE Extension benötigt. Um die PlatformIO IDE Extension in VS-Code zu installieren, drückt man einfach auf das Extensions Symbol oder drückt die Tastenkombination Ctrl+Shift+X um das Extensions Menü zu öffnen. Danach gibt man in der Suchleiste "PlatformIO IDE" ein und wählt die Extension mit der Ameise als Icon. Dann drückt man auf "Install" und wartet, bis die Extension fertig heruntergeladen ist. Nach der Installation sollte das PlatformIO Icon (Ameisenkopf) auf der linken Seite unter dem Extension Menü erscheinen.

Um nun ein neues Projekt zu erstellen, klickt man auf das PlatformIO Icon und wählt "+ New Project".

Im Project Wizard wählt man nun den gewünschten Namen, das Board, das Framework als auch den Speicherort des Projektes. Bei unserem Projekt wählen wir das Board "Espressif ESP32 Dev Module" und als Framework "Arduino", da wir einen ESP32 zum Programmieren verwendeten.

---

<sup>2</sup><https://code.visualstudio.com/>

---

### 9.3.2 Bibliotheken

Bibliotheken sind ein weiterer wichtiger Bestandteil für das Programmieren. Bibliotheken beinhalten bereits eine Sammlung von vorgefertigtem Code, der für bestimmte Aufgaben, wie zum Beispiel zum Auswerten eines Sensors, verwendet werden kann. Bibliotheken werden verwendet, um den Code zu minimieren und dadurch die Lesbarkeit sowie die Effizienz zu steigern. Um eine Bibliothek für PlatformIO in VS-Code zu installieren, muss das PIO Home Menü geöffnet werden. Darin befindet sich der Reiter „Libraries“. Wenn dieses geöffnet wird, erscheint eine Suchleiste, mit der die gewünschten Bibliotheken zum Projekt hinzugefügt werden können.

### 9.3.3 verwendete Bibliotheken

#### ESPAsyncWebServer

Die ESPAsyncWebServer Bibliothek ermöglicht es, Webanwendungen effizient und mit hoher Performance auf ESP8266- und ESP32 Mikrocontroller zu hosten. Der Asynchrone Betrieb verhindert Blockierungen und sorgt für eine flüssige Verarbeitung mehrerer Anfragen gleichzeitig. Außerdem unterstützt die Bibliothek WebSockets, welche für die Echtzeitkommunikation zwischen Client und Server benötigt wurden. Die Bibliothek ist eine leistungsfähigere Alternative zur klassischen WebServer-Bibliothek, da sie ressourcenschonender und nicht blockieren arbeitet.<sup>3</sup>

#### ArduinoJSON

Die ArduinoJson Bibliothek ermöglicht die Verarbeitung von JSON-String in Objekte und umgekehrt. Sie ist speziell für Geräte mit begrenztem Speicher und Rechenleistung optimiert. Die Bibliothek wird benötigt, um die erhaltenen Steuerbefehle am ESP32 zu konvertieren und um sie anschließend weiterzuverarbeiten.<sup>4</sup>

#### HCSR04

#### arduinoWebSockets

Die WebSockets Bibliothek ermöglicht eine Kommunikation über das WebSocket Protokoll für Arduino-Boards, ESP8266 und ESP32. Die Bibliothek wird für die Echtzeit-Kommunikation zwischen dem Webserver und den ESP32 benötigt. Außerdem werden die Bilder der ESP32-CAM über einen WebSocket an den Webserver gesendet. Die Bibliothek ist eine großartige Ergänzung zur ESPAsyncWebServer Bibliothek.<sup>5</sup>

---

<sup>3</sup><https://github.com/lacamera/ESPAsyncWebServer>

<sup>4</sup><https://arduinojson.org/>

<sup>5</sup><https://github.com/Links2004/arduinoWebSockets>

---

**ESp32Servo**

**Adafruit\_MCP23x17**

**HX711\_ADC**

**9.4 LaTeX**

**9.5 GitHub**

---

# 10 Abbildungsverzeichnis

## Abbildungsverzeichnis

1	Porträt Daniel Schauer . . . . .	9
2	Porträt Simon Spari . . . . .	9
3	Porträt Felix Hochegger . . . . .	9
4	Motoren . . . . .	12
5	BLDC-Motor . . . . .	13
6	Phasensteuerung . . . . .	13
7	Motorteiber-ZS-X11H . . . . .	14
8	Motorphasen Verbindung . . . . .	15
9	Pins für die PWM . . . . .	15
10	Pin für die Bremse und den Richtungswechsel . . . . .	16
11	Versorgungs-Pins . . . . .	16
12	Hall-sensoren Eingänge . . . . .	17
13	ESPAsyncWebServer.h-Bibliothek . . . . .	19
14	WebServer Netzwerkkonfiguration . . . . .	19
15	Webserver Setup . . . . .	20
16	SPIFFS Initialisierung . . . . .	21
17	readfile()-Funktion . . . . .	22
18	Bibliotheken für die Kommunikation . . . . .	22
19	Netzwerkkonfiguration Client . . . . .	23
20	Setup ESP32 . . . . .	24
21	onWebSocketEvent()-Funktion . . . . .	25
22	JSON-Beispiel für Steuerung . . . . .	25
23	JSON-Beispiel für Kamerasteuerung . . . . .	26
24	JSON-Beispiel für Lichtsteuerung . . . . .	26
25	handleWebSocketMessage()-Funktion . . . . .	27
26	Kamera-GPIO Konfiguration . . . . .	28
27	Kamera-Initialisierung . . . . .	29
28	Videoübertragung über WebSocket . . . . .	31
29	Steuerkreuz auf der Website . . . . .	32
30	Steuerkreuz HTML-Code . . . . .	32
31	CSS-Klassen .dpad-container und .button . . . . .	33
32	CSS-Richtungsklassen . . . . .	34

---

33	<hier gute Bildbeschreibung einfügen> . . . . .	35
34	<hier gute Bildbeschreibung einfügen> . . . . .	35
35	send()-Funktion . . . . .	36
36	stop()-Funktion . . . . .	37
37	Geschwindigkeitssteuerung auf der Website . . . . .	37
38	Geschwindigkeitssteuerung HTML-Code . . . . .	38
39	CSS-Klassen für die Geschwindigkeitssteuerung . . . . .	38
40	speed-change()-Funktion . . . . .	39
41	Kamerasteuerung auf der Website . . . . .	39
42	Kamerasteuerung HTML-Code . . . . .	40
43	CSS-Klassen für die Kamerasteuerung . . . . .	40
44	camera-change()-Funktion . . . . .	41
45	Fernlicht-Icon auf der Website . . . . .	42
46	HTML-Code für Fernlicht . . . . .	42
47	CSS-Klassen für das Fernlicht . . . . .	43
48	togglelight()-Funktion . . . . .	44
49	Videoübertragung auf der Website . . . . .	45
50	Videoübertragung HTML-Code . . . . .	45
51	CSS-Formatierung für dynamicimage . . . . .	45
52	Videoübertragung im JavaScript-Code . . . . .	46
53	Menü-Icon auf der Website . . . . .	47
54	Sidebar mit Sensordaten . . . . .	47
55	Sensordaten HTML-Code . . . . .	48
56	CSS-Klassen für das Menü . . . . .	49
57	CSS-Klasse .daten . . . . .	50
58	JavaScript-Verarbeitung der Sensordaten . . . . .	50
59	Statusbox auf der Website . . . . .	51
60	Statusbox: Verbunden . . . . .	51
61	Statusbox: Verbindung getrennt . . . . .	51
62	Statusbox HTML-Code . . . . .	51
63	CSS-Klassen für die Statusbox . . . . .	52
64	JavaScript Funktionen für den Verbindungsstatus . . . . .	53

---

# **11 Literaturverzeichnis**