

Diplomarbeit:

Lastenroboter

Höhere Technische Bundeslehranstalt Graz Gösting
Schuljahr 2024/25



Diplomanden:

Daniel Schauer	5AHEL	Prof. DI. Gernot Mörtl
Simon Spari	5AHEL	
Felix Hochegger	5AHEL	

Betreuer:

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

1 Einleitung	7
1.1 Kurzzusammenfassung	7
1.2 Abstract	8
2 Projektmanagement	9
2.1 Projektteam	9
2.2 Projektstrukturplan	10
2.3 Meilensteine	11
2.4 Kostenaufstellung	11
3 Antrieb	12
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 mit einem Motor	18
3.3.1 Komponenten	18
3.3.2 Schaltungsaufbau und Verbindungen	18
3.4 Schaltungsaufbau mit vier Motor	20
3.4.1 Komponenten	20
3.4.2 Schaltungserweiterung für vier Motoren:	21
3.5 Code	21
3.5.1 Initialisierung	21
3.5.2 Setup-Code	23
3.5.3 Handling der Steuerbefehle	24
3.5.4 Ansteuerung der Treiberplatinen:	25
4 Webserver	29
4.1 Grundlegende Ziele	29
4.2 Webserver	30
4.2.1 Webserver Setup	30
4.2.2 SPIFFS Setup	31
4.3 WebSocket Kommunikation	33
4.3.1 Kommunikation Setup	33

4.3.2	Message Handling	34
4.4	Kamera	38
4.4.1	Kamera Setup	38
4.5	Videoübertragung	40
4.6	Website	41
4.6.1	Implementierung der Steuerung	41
4.6.2	Echtzeit-Videoanzeige	52
4.6.3	Anzeige von Sensordaten und Verbindungsstatus	55
4.7	Herausforderungen und Optimierungen	61
4.7.1	Probleme bei der WebSocket Kommunikation	61
4.7.2	Latenz- und Performance Optimierungen	61
4.8	(Fazit und Ausblick)	61
4.8.1	(Mögliche Erweiterungen und Verbesserungen)	61
5	Gehäuse	62
5.1	Planung und Design	62
5.2	Realisierung	68
5.3	Materialliste	68
6	Platine	69
6.1	Grundschaltung	69
6.2	Circuit Board	69
6.3	Fertiger Prototyp	69
7	Kamera	70
7.1	Kamera im Überblick	70
7.2	Videoübertragung	70
7.3	Kameraschwenkung	70
7.3.1	Gehäuse	70
7.3.2	Servomotor	70
7.4	Code	70
8	Sensoren	71
8.1	Abstandssensor	71
8.1.1	Grundprinzip	71
8.1.2	Schaltungsaufbau	73
8.1.3	Nutzung	74
8.1.4	Code	75
8.2	Gewichtsmessung	76

8.2.1	Grundprinzip	76
8.2.2	Schaltungsaufbau	79
8.2.3	Nutzung	80
8.2.4	Code	81
9	Entwicklungstools	84
9.1	Autodesk Fusion	84
9.2	Eagle	84
9.3	VS-Code	84
9.3.1	Setup	84
9.3.2	Bibliotheken	85
9.3.3	verwendete Bibliotheken	85
9.4	LaTex	86
9.5	GitHub	86
10	Abbildungsverzeichnis	87
11	Literaturverzeichnis	91

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 zur visuellen Überwachung des Transportbereichs dient, 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 miteinander verbindet.

Die Steuerung des Roboters erfolgt über eine Website, welche die Bedienung sowie die Anzeige relevanter Daten wie Akkustand und Gewicht ermöglicht. Ein besonderer Fokus liegt 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-Mikrocontroller 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 is about the development of 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.

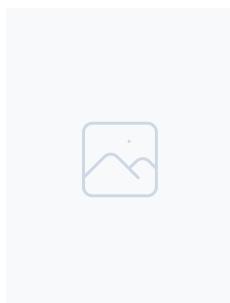
The robot is controlled via a website, which allows the user to operate the robot and access important data such as battery level and weight. A particular focus is also placed on transferring the camera feed to the web interface and integrating a swivel camera to ensure flexible viewing of the transport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



Daniel Schauer: OnBoard-Software

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

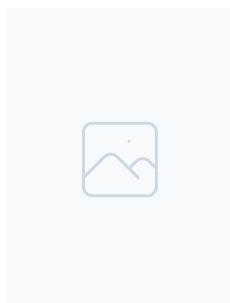
Abbildung 1:
Porträt
Daniel Schauer



Simon Spari: Software-App

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

Abbildung 2:
Porträt
Simon Spari



Felix Hocegger: Hardware-Design und Mechanik

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

Abbildung 3:
Porträt Felix
Hocegger

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 Website, 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 Hochegger: 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:

Um unser Projekt besser zu strukturieren, haben wir am Anfang einen Grobplan für den Lastenroboter erstellt. Er hilft uns, die einzelnen Aufgaben und deren Verknüpfungen besser zu definieren. So können wir sicherstellen, dass Mechanik, Hardware und Software ohne Probleme zusammenarbeiten. Durch diese Planung behalten wir den Überblick über unser Projekt.



Abbildung 4: Lastenroboter Projekt-Grobplan

Quelle: eigene Abbildung erstellt mit Obsidian

2.3 Meilensteine

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

Die folgenden Meilensteine wurden bei der Projektplanung definiert:

Meilenstein	Datum
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 GPIO Expander-Board	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 SG90	6.04 €	1	6.04 €
IRM-30-15ST AC/DC-Leistungsmodul	17.04 €	1	17.04 €
SOLSUM 0808 Solarladeregler	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 €
Summe			466.61 €

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, da er eine hohe Leistung bietet.



Abbildung 5: Motoren

Quelle: eigene Abbildung

3.1.2 Funktionsweise

Die Funktionsweise eines bürstenlosen Gleichstrommotors basiert auf der 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 dass die Spulen ein rotierendes Magnetfeld erzeugen, dreht sich der Rotor synchron mit diesem Magnetfeld mit.



Abbildung 6: BLDC-Motor

Quelle: www.elektrikrehberiniz.com

Damit die Phasen des Motors korrekt geschaltet werden, muss die genaue Position des Rotors im Inneren des Motors erfasst werden. Die genaue Position des Rotors wird durch Hallsensoren erfasst. Die drei 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 Pulsweitenmodulations-Signal (PWM-Signal) verwendet. Durch die PWM wird die Stromzufuhr zu den Phasen des Motors gesteuert. Wenn das Pulsweitenmodulations-Signal (PWM) erhöht wird, bedeutet das, dass die Phasen in kürzeren Abständen angesteuert werden. Dadurch erhält der Motor häufiger Energieimpulse, was zu einer höheren Drehzahl des Rotors führt. Das PWM-Signal regelt die Versorgungsspannung, indem es sie in schneller Folge ein- und ausschaltet. Je höher das Tastverhältnis des PWM-Signals, desto länger bleibt die Spannung während eines Zyklus eingeschaltet, wodurch der Motor mehr Energie erhält und sich schneller dreht.

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



Abbildung 7: Phasensteuerung

Quelle: davincii.de/arduino-projekte/brushless-motor-ansteuerung

3.1.3 Technische Daten

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

3.2 Motortreiber

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. Dies ermöglichen 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 einer Brems-Funktion.

3.2.2 Aufbau und Funktionen

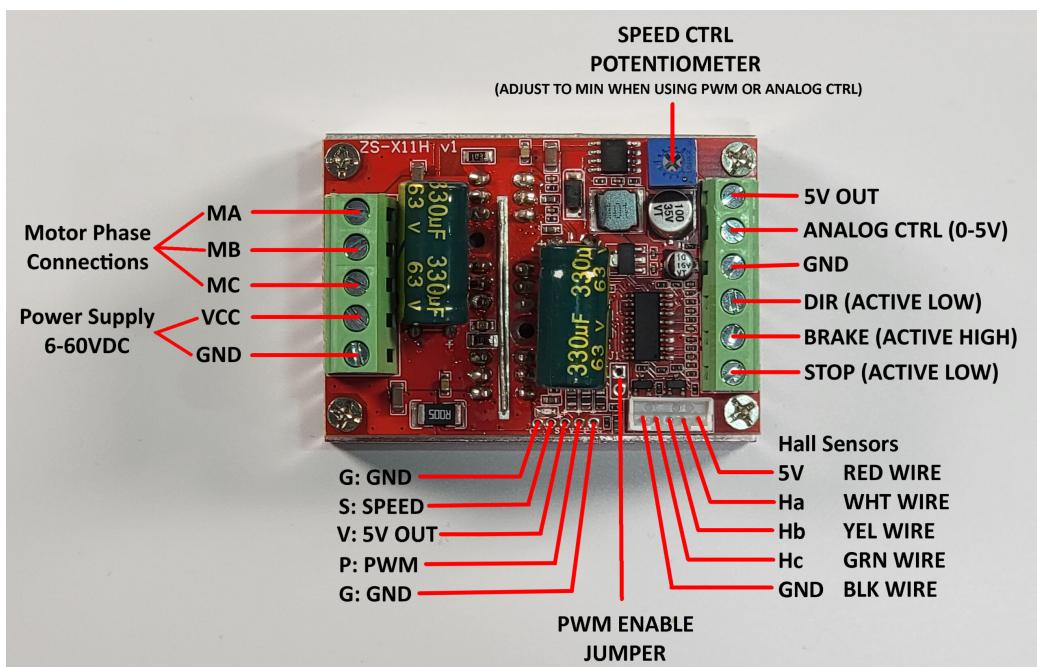


Abbildung 8: Motortreiber-ZS-X11H

Quelle: 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 9: Motorphasen Verbindung

Quelle: mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g

PWM-Steuerung

- Die Pulsweitenmodulation (PWM) steuert die Geschwindigkeit des Motors, indem sie die Einschaltzeit der Spannung variiert, wodurch die durchschnittliche Leistung, die den Motorwicklungen zugeführt wird, angepasst 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.
- Wenn die Brücke auf der Platine verbunden wird, kommt die externe PWM zum Einsatz.

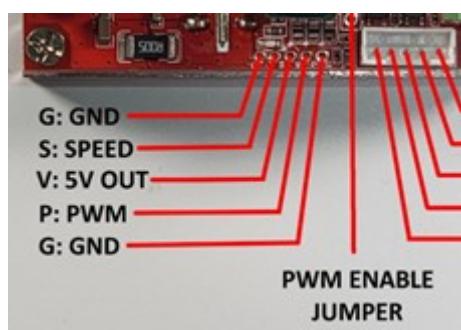


Abbildung 10: Pins für die PWM

Quelle: mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g

Richtungswechsel und Bremse

- Die Platine verfügt über zwei Steuereingänge ein 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 11: Pin für die Bremse und den Richtungswechsel
Quelle: 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 12: Versorgungs-Pins
Quelle: 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 13: Hall-sensoren Eingänge

Quelle: mad-ee.com/easy-inexpensive-hoverboard-motor-controller/g

Sicherheit und Schutzfunktionen

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

3.3 Schaltungsaufbau mit einem Motor

3.3.1 Komponenten

- BLDC-Motor
- Motortreiber (ZS-X11H)
- Mikrocontroller (ESP32)
- Expander Modul (MCP23017)

3.3.2 Schaltungsaufbau und Verbindungen



Abbildung 14: Schaltungsaufbau mit einem Motor

Quelle: eigene Abbildung

Phasenanschlüsse des Motors (blau):

Die Phasen des Motors werden mit der Treiberplatine verbunden. Diese werden abwechselnd mit Spannung versorgt, um eine Drehbewegung zu erzeugen. Die weiße Phase des Motors wird mit MA verbunden. Die blaue Phase des Motors wird mit MB verbunden. Die orange Phase des Motors wird mit MC verbunden.

5	B	AWG22	BU
6	C	AWG22	OR
7	A	AWG22	WH

1	H1	AWG26	YE
2	H3	AWG26	BN
3	H2	AWG26	GN

Abbildung 15: Hall-Sensoren Eingänge

Quelle:

easy-inexpensive-hoverboard-motor-controller

Abbildung 16: Hall-Sensoren-Motor

Quelle: eigene Abbildung

Hall-Sensoren Verbindungen (orange)

Die Hall-Sensoren des Motors werden mit der Treiberplatine verbunden und erfassen die aktuelle Rotorposition. Diese Informationen werden an die Steuerelektronik übermittelt, die daraufhin die Motorphasen entsprechend schaltet, um die gewünschte Drehbewegung zu erzeugen. Dabei wird der gelbe Hall-Sensor mit HA, der grüne mit HB und der braune mit HC auf dem Treiber verbunden.



Abbildung 17: Motor Phasen

Quelle: eigene Abbildung



Abbildung 18: Treiber Phasen

Quelle: eigene Abbildung

Versorgung (rot und schwarz)

- Der Treiber wird mit 12 Volt von der Batterie versorgt.
- Mit diesen 12 Volt steuert der Treiber den Motorstrom an den Phasen.

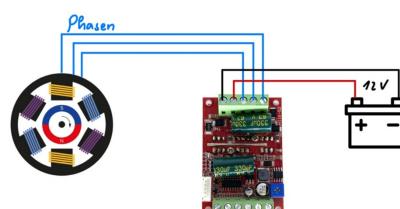


Abbildung 19: Versorgung der Motoren und Treiberplatine

Quelle: eigene Abbildung

Steuerverbindung (grün und grau)

Der ESP32 ist mit dem Motortreiber verbunden, um Geschwindigkeit zu regeln. Das funktioniert über ein PWM-Signal.

Das Expander Modul steuert die Richtung und die Bremsfunktion. Der Richtungswechsel erfolgt, indem das Expander Modul den entsprechenden Eingang auf LOW oder HIGH schaltet. Die Bremse reagiert auf eine HIGH Signal. Das heißt, wenn das Expander Modul den Eingang auf der Steuerplatine HIGH schaltet, wird der Motor gebremst.



Abbildung 20: Ansteuerung der Treiberplatine
Quelle: eigene Abbildung

3.4 Schaltungsaufbau mit vier Motor

3.4.1 Komponenten

- 4 BLDC-Motor
- 4 Motortreiber (ZS-X11H)
- Mikrocontroller (ESP32)
- Expander Modul (MCP23017)

3.4.2 Schaltungserweiterung für vier Motoren:

Der Lastenroboter wird von vier BLDC-Motoren angetrieben, die jeweils über einen eigenen Motortreiber gesteuert werden. Jeder Motortreiber wird über einen PWM-Pin des Esp32 verbunden, um die Geschwindigkeit zu regulieren. Zusätzlich wird jeder Motortreiber über das Expander-Modul angesteuert, um die Richtung zu ändern und die Bremse zu aktivieren. Alle Treiber werden mit 12 Volt von der Batterie versorgt.

```
int leftfrontwheel_pwm = 32;
const int leftfrontwheel_brake = 1;
const int leftfrontwheel = 2;

int leftrearwheel_pwm = 18;
const int leftrearwheel_brake = 3;
const int leftrearwheel = 4;

int rightfrontwheel_pwm = 27;
const int rightfrontwheel_brake = 5;
int rightfrontwheel = 6;

int rightrearwheel_pwm = 25;
const int rightrearwheel_brake = 7;
const int rightrearwheel = 8;
```

Abbildung 21: Motoren-GPIO Konfiguration

Quelle: eigene Abbildung

3.5 Code

3.5.1 Initialisierung

Um die Motoren beziehungsweise die Treiberplatinen anzusteuern, gibt es pro Motor drei Pins. Einer dieser Pins wird mit einem PWM-fähigen Pin am ESP32 verbunden. Die anderen beiden werden mit dem Expander Modul verbunden. Da das Expander Modul nur Digitale Ein-/Ausgänge besitzt, ist es nicht möglich die PWM-Signale auch über dieses Modul zu realisieren. Der Hauptgrund für diese Aufteilung auf den Expander ist, der das wir Analoge Pins am ESP32 einsparen müssen, um diese für andere Sensoren oder Aktoren einzusetzen.

```
int leftfrontwheel_pwm = 32;
const int leftfrontwheel_brake = 1;
const int leftfrontwheel = 2;

int leftrearwheel_pwm = 18;
const int leftrearwheel_brake = 3;
const int leftrearwheel = 4;

int rightfrontwheel_pwm = 27;
const int rightfrontwheel_brake = 5;
int rightfrontwheel = 6;

int rightrearwheel_pwm = 25;
const int rightrearwheel_brake = 7;
const int rightrearwheel = 8;
```

Abbildung 22: Motoren-GPIO Konfiguration

Quelle: eigene Abbildung

Um das Expander-Board zu nutzen, wird die Bibliothek Adafruit_MCP23X17 verwendet. Diese Bibliothek ermöglicht eine einfachere Initialisierung und Ansteuerung der MCP-Pins, sodass die GPIOs ähnlich wie die des ESP32 genutzt werden können. Außerdem wird ein Objekt der Klasse Adafruit MCP23X17 erstellt, über das man später auf den Expander zugreifen kann.

```
#include <Adafruit_MCP23X17.h>
Adafruit_MCP23X17 mcp;
```

Abbildung 23: Adafruit_MCP23X17.h-Bibliothek

Quelle: eigene Abbildung

3.5.2 Setup-Code

Am Anfang des Setup-Codes wird überprüft, ob ein Expander Modul über die I2C Schnittstelle verbunden ist und dieses auch funktionstüchtig ist. Falls das nicht der Fall ist, wird ein Error über die Serielle Schnittstelle ausgegeben und das Setup wird unterbrochen.

```
if (!mcp.begin_I2C())
{
    Serial.println("MCP Error.");
    while (1);
}
```

Abbildung 24: MCP-Setup

Quelle: eigene Abbildung

Danach werden die Pins des ESP32 sowie die des Expander-Moduls als Ausgänge initialisiert. Die Pins des Expander-Ports müssen mit dem Befehl mcp. vor pinMode initialisiert werden. Zudem wird die Motorbremse direkt über den MCP mit digitalWrite aktiviert, sodass sich beim Starten kein Motor ungewollt dreht.

```
mcp.pinMode(leftfrontwheel, OUTPUT);
pinMode(leftfrontwheel_pwm, OUTPUT);
mcp.pinMode(leftfrontwheel_brake, OUTPUT);
mcp.digitalWrite(leftfrontwheel_brake, HIGH);

mcp.pinMode(rightfrontwheel, OUTPUT);
pinMode(rightfrontwheel_pwm, OUTPUT);
mcp.pinMode(rightfrontwheel_brake, OUTPUT);
mcp.digitalWrite(rightfrontwheel_brake, HIGH);

mcp.pinMode(leftrearwheel, OUTPUT);
pinMode(leftrearwheel_pwm, OUTPUT);
mcp.pinMode(leftrearwheel_brake, OUTPUT);
mcp.digitalWrite(leftrearwheel_brake, HIGH);

mcp.pinMode(rightrearwheel, OUTPUT);
pinMode(rightrearwheel_pwm, OUTPUT);
mcp.pinMode(rightrearwheel_brake, OUTPUT);
mcp.digitalWrite(rightrearwheel_brake, HIGH);
```

Abbildung 25: Initialisierung Motoren

Quelle: eigene Abbildung

3.5.3 Handling der Steuerbefehle

Um den Roboter zu steuern, werden die verschiedenen Steuerbefehle von der Webseite durch spezifische Funktionen bearbeitet. Die Richtungsbefehle zur Steuerung des Roboters werden mit einer Art State-Maschine beziehungsweise mit einer switch-case Struktur realisiert. Dafür wird zunächst ein enum (Enumeration) erstellt, dass die möglichen Bewegungsrichtungen des Roboters definiert und die Direction wird sofort auf „HALT“ gesetzt. Außerdem wird die Variable speed_cb für die Geschwindigkeit initialisiert.

```
enum Direction
{
    HALT,
    UP,
    DOWN,
    LEFT,
    RIGHT,
    CAM_RIGHT,
    CAM_LEFT
};
int speed_cb = 0;
Direction dir = HALT;
```

Abbildung 26: Enumeration von Bewegungsrichtungen
Quelle: eigene Abbildung

Befehle zum Steuern:

- HALT: Der Roboter bremst.
- UP: Bewegt den Roboter vorwärts.
- DOWN: Bewegt den Roboter rückwärts.
- LEFT: Lässt den Roboter nach links drehen.
- RIGHT: Lässt den Roboter nach rechts drehen.

Um die Befehle von der Webseite für die Steuerung zu benutzen, werden zunächst die erhaltenen Strings, mit einer Funktion in die vorhin definierte Richtungsvariable umgewandelt.

```
Direction stringToDirection(const String &str)
{
    if (str == "up")
    {
        return UP;
    }
    else if (str == "down")
    {
        return DOWN;
    }
    else if (str == "left")
    {
        return LEFT;
    }
    else if (str == "right")
    {
        return RIGHT;
    }
    else
    {
        return HALT;
    }
}
```

Abbildung 27: Strings zu Enum Variable umwandeln

Quelle: eigene Abbildung

3.5.4 Ansteuerung der Treiberplatinen:

Die Geschwindigkeit und die Richtung des Roboters werden schlussendlich in der Funktion „void navigate()“ bearbeitet. Diese Funktion wird in der „void loop()“ die ganze Zeit ausgeführt.

```
void loop()
{
    websocket.loop();
    navigate();
```

Abbildung 28: navigate() in loop()

Quelle: eigene Abbildung

Zu Beginn werden zwei statische Variablen definiert, die dazu dienen, die zuletzt empfangene Fahrtrichtung (lastDir) und Geschwindigkeit (lastSpeed) zu speichern. Anschließend wird überprüft, ob sich entweder die Richtung oder die Geschwindigkeit im Vergleich zum vorherigen Wert geändert hat. Falls sich die Geschwindigkeit im Vergleich zum vorherigen Wert geändert hat, wird der empfangene String von der Webseite in einen Integer umgewandelt. Der Wert von „speed“ liegt im Bereich von 0–100 % und wird in einen Wert von 0–255 skaliert, der später für die PWM-Signale verwendet wird.

```
void navigate()
{
    static Direction lastDir = HALT;
    static int lastSpeed = -1;

    if (dir != lastDir || speed_cb != lastSpeed)
    {
        speed_cb = speed.toInt() * 2.55;
        lastSpeed = speed_cb;
```

Abbildung 29: Überprüfung auf Änderung und Speed
in Integer umwandeln in navigate()-Funktion
Quelle: eigene Abbildung

Außerdem wird die Motorbremse deaktiviert, sodass sich der Roboter überhaupt bewegt und sich steuern lässt.

```
mcp.digitalWrite(leftfrontwheel_brake, LOW);
mcp.digitalWrite(rightfrontwheel_brake, LOW);
mcp.digitalWrite(leftrearwheel_brake, LOW);
mcp.digitalWrite(rightrearwheel_brake, LOW);
```

Abbildung 30: Motorbremse deaktivieren in navigate()-Funktion
Quelle: eigene Abbildung

Die Steuerung der Richtung des Roboters erfolgt durch eine switch-case-Struktur, die anhand der aktuellen Fahrtrichtung (dir) die entsprechende Bewegungsfunktion aufruft. Diese Steuerlogik basiert auf dem zuvor definierten enum Direction, der die verschiedenen Zustände wie UP, DOWN, LEFT, RIGHT und HALT enthält. Diese Struktur funktioniert so ähnlich wie eine Gangschaltung, sie legt nämlich nur fest in welche Richtung sich der Motor drehen soll.

```

switch (dir)
{
case UP:
    moveForward();
    break;

case DOWN:
    moveBackward();
    break;

case LEFT:
    turnLeft();
    break;

case RIGHT:
    turnRight();
    break;

case HALT:
    stop();
    break;
default:
    stop();
    break;
}

void moveForward()
{
    mcp.digitalWrite(leftfrontwheel, HIGH);
    mcp.digitalWrite(rightfrontwheel, LOW);
    mcp.digitalWrite(leftrearwheel, HIGH);
    mcp.digitalWrite(rightrearwheel, LOW);
    Serial.println("Vorwärts");
}

void moveBackward()
{
    mcp.digitalWrite(leftfrontwheel, LOW);
    mcp.digitalWrite(rightfrontwheel, HIGH);
    mcp.digitalWrite(leftrearwheel, LOW);
    mcp.digitalWrite(rightrearwheel, HIGH);
    Serial.println("Rückwärts");
}

void turnLeft()
{
    mcp.digitalWrite(leftfrontwheel, LOW);
    mcp.digitalWrite(rightfrontwheel, LOW);
    mcp.digitalWrite(leftrearwheel, LOW);
    mcp.digitalWrite(rightrearwheel, LOW);
    Serial.println("Links");
}

void turnRight()
{
    mcp.digitalWrite(leftfrontwheel, HIGH);
    mcp.digitalWrite(rightfrontwheel, HIGH);
    mcp.digitalWrite(leftrearwheel, HIGH);
    mcp.digitalWrite(rightrearwheel, HIGH);
    Serial.println("Rechts");
}

void stop()
{
    mcp.digitalWrite(leftfrontwheel_brake, HIGH);
    mcp.digitalWrite(rightfrontwheel_brake, HIGH);
    mcp.digitalWrite(leftrearwheel_brake, HIGH);
    mcp.digitalWrite(rightrearwheel_brake, HIGH);
    Serial.println("Stop");
}

```

Abbildung 31: switch-case Struktur in navigate()-Funktion

Quelle: eigene Abbildung

Die Steuerung der Geschwindigkeit wird mithilfe von PWM-Signalen umgesetzt. Zunächst wird eine statische Variable „lastPWM“ definiert, die später den zuletzt verwendeten PWM-Wert speichert. Bevor die Treiber-Platinen mit einem PWM-Signal angesteuert werden, wird überprüft, ob sich die Geschwindigkeit tatsächlich geändert hat. Falls sich „speed_cb“ vom vorherigen Wert abweicht, werden die neuen PWM-Werte mit analogWrite() an die Motorsteuerungsplatinen gesendet, um die Geschwindigkeit der Motoren entsprechend anzupassen.

Danach wird der Wert von „lastPWM“ aktualisiert, um sicherzustellen, dass die gleiche Geschwindigkeit nicht unnötig erneut gesetzt wird. Zusätzlich wird die Variable „lastDir“ aktualisiert, um die letzte Richtung zu speichern.

```
static int lastPWM = -1;
if (lastPWM != speed_cb)
{
    analogWrite(leftfrontwheel_pwm, speed_cb);
    analogWrite(rightfrontwheel_pwm, speed_cb);
    analogWrite(leftrearwheel_pwm, speed_cb);
    analogWrite(rightrearwheel_pwm, speed_cb);
    lastPWM = speed_cb;
}

lastDir = dir;
}
```

Abbildung 32: PWM mit analogWrite übertragen und
letzte Werte speichern in navigate()-Funktion

Quelle: eigene Abbildung

4 Webserver

4.1 Grundlegende Ziele

In diesem Kapitel wird die geplante Website thematisiert, die die Steuerung des Lastenroboters ermöglichen soll. Zunächst werden die grundlegenden Funktionen definiert, die die Website erfüllen muss. Sobald diese Anforderungen umgesetzt sind, liegt der Fokus auf einer möglichst einfachen und benutzerfreundlichen Gestaltung des User Interfaces. Ein weiterer zentraler Aspekt ist die Darstellung spezifischer Messwerte und Daten, sodass diese über den Webserver schnell und unkompliziert zugänglich sind.

Für die Entwicklung der Website kommen HTML, CSS und JavaScript zum Einsatz, um sowohl die funktionalen als auch die optischen Anforderungen zu erfüllen.

Videoübertragung

Die Website ermöglicht die Echtzeit-Videoübertragung des Kamerabilds der ESP32-CAM. Die Bildfrequenz und die Qualität der Videoübertragung 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 Webserver

4.2.1 Webserver Setup

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

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

Abbildung 33: ESPAsyncWebServer.h-Bibliothek

Quelle: eigene Abbildung

Netzwerkkonfiguration

Die ESP32-CAM fungiert als Access Point mit folgenden Konfigurationen:

- SSID (Service Set Identifier) : “Carybot” – dient zur Identifikation des drahtlosen Netwerkes
- Passwort: “123456789“ – dient zum Schutz des Netwerkes
- 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
- Subnetzmase: 255.255.255.0 - ermöglicht die Kommunikation zwischen Geräten im Bereich 192.168.4.x
- Port: 80 - 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 34: WebServer Netzwerkkonfiguration

Quelle: eigene Abbildung

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

In der Setup Funktion wird danach überprüft, ob der Access Point erfolgreich konfiguriert worden ist und ob er 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 35: Webserver Setup

Quelle: eigene Abbildung

4.2.2 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 Dateien aus SPIFFS an Clients gesendet werden.

“/“ ist die Default-Route 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 36: 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 37: `readfile()`-Funktion

Quelle: eigene Abbildung

4.3 WebSocket Kommunikation

4.3.1 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 38: Bibliotheken für die Kommunikation

Quelle: eigene Abbildung

Für die Netzwerkkonfiguration als Client werden die gleichen Parameter wie in der Access Point Konfiguration gewählt (siehe 34), mit Ausnahmen der IP-Adresse und dem WebSocket Port. Als Lokale IP-Adresse wurde 192.168.4.3 und als Port 8080 gewählt.

In der Setup Funktion des Programmes wird 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 gestartet. Es wird definiert, dass die Funktion onWebSocketEvent aufgerufen wird, wenn Events über den WebSocket registriert werden. In der loop Funktion wird 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 39: Setup ESP32

Quelle: eigene Abbildung

4.3.2 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-Buffers. In der Funktion werden die Events mit einem switch-case verarbeitet

Übersicht der WebSocket-Ereignisse:

Ereignistyp	Beschreibung	Verarbeitung
WType_CONNECTED	Ein neuer Client verbindet sich.	Ausgabe der Client-ID & IP-Adresse in der Konsole
WType_TEXT	Eine Textnachricht wird empfangen.	Übergabe an handleWebSocketMessage()
WType_BIN	Binärdaten werden empfangen.	Nachricht in der Konsole (wird nicht verarbeitet)
WType_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, WType_t type, uint8_t *payload, size_t length)
{
    switch (type)
    {
        case WType_TEXT:
            handleWebSocketMessage(num, payload, length);
            break;

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

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

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

Abbildung 40: onWebSocketEvent()-Funktion

Quelle: eigene Abbildung

Die Funktion handleWebSocketMessage() verarbeitet eingehende WebSocket-Nachrichten, die als JSON-Objekte gesendet wird. Sie nimmt drei Parameter entgegen:

1. num - Client-ID der Websocket Verbindung
2. payload - die empfangene Nachricht als Byte-Array
3. length - die Länge der empfangenen Nachricht

Zuerst wird das payload-Byte-Array in einen String konvertiert und anschließend zu Debugging-Zwecken auf der seriellen Konsole ausgegeben. Danach wird ein StaticJsonDocument mit einer maximalen Größe von 200 Bytes erstellt, um die empfangenen JSON-Daten zu speichern. Anschließend wird die Nachricht mit deserializeJson() geparsst. Falls das Parsen fehlschlägt, wird die Verarbeitung abgebrochen. Je nachdem, welche Schlüssel in der JSON-Nachricht enthalten sind, werden unterschiedliche Funktionen ausgeführt.

Falls die JSON-Nachricht den Schlüssel `robot_direction` enthält, wird der zugehörige Wert ausgelesen und mithilfe der Funktion `stringToDirection()` in eine interne Richtungsvariable (`dir`) umgewandelt. Zusätzlich wird der Wert für "speed" aus der Nachricht extrahiert und als String gespeichert.

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

Abbildung 41: JSON-Beispiel für Steuerung

Quelle: eigene Abbildung

Falls die JSON-Nachricht den Schlüssel `camera_position` enthält, wird der Wert als Ganzzahl in die Variable `camera_pos` gespeichert. Anschließend wird die Funktion `cam_turn()` aufgerufen, um die Kamera entsprechend zu bewegen.

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

Abbildung 42: JSON-Beispiel für Kamerasteuerung

Quelle: eigene Abbildung

Falls die JSON-Nachricht den Schlüssel `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` True ist, wird die Funktion `lights_on()` aufgerufen, ansonsten wird die Funktion `lights_off()` aufgerufen.

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

Abbildung 43: 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 44: handleWebSocketMessage() -Funktion

Quelle: eigene Abbildung

4.4 Kamera

4.4.1 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 45: 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 festlegt werden können.

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

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.

```
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 46: Kamera-Initialisierung

Quelle: eigene Abbildung

4.5 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 eine 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 `window.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 ``-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, [])(AsyncWebRequest *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>
    )rawliteral";
    request->send(100, "text/html", html);
};

server.begin();
```

Abbildung 47: Videoübertragung über WebSocket

Quelle: eigene Abbildung

4.6 Website

4.6.1 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 48: 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 49: 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 man die Form eines Steuerkreuzes erhält. 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 50: 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 51: 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 52: EventListener für ungewünschte Aktionen

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 53: EventListener für Eingabe

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 54: `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 überschrieben. 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 55: 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 56: 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 57: 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 beim Hochschieben des Sliders die aktuelle Position erhöht und beim Runterschieben vermindert wird.

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

}

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

Abbildung 58: 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 Steuerkreuz 55)

```
var speed = 50;

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

Abbildung 59: 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 60: 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 61: 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 Slidernopf 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 62: 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 63: 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 blass-grünen Hintergrund.



Abbildung 64: Fernlicht-Icon auf der Website

Quelle: eigene Abbildung

Im HTML-Code wird das Icon als HTML -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 65: 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 sanfe Animation über 0,3 Sekunden angezeigt.

Die grayscale Klasse wandelt das Icon in die ausgegraute Darstellung um, um anzuseigen, dass das Fernlicht deaktiviert ist. Die active-light Klasse ändert die Hintergrundfarbe zu einem transparenten grünen Ton, um anzuseigen, 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 66: 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 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 67: togglelight()-Funktion

Quelle: eigene Abbildung

4.6.2 Echtzeit-Videoanzeige

Die Videoübertragung dient dazu, den Roboter über größere Entferungen 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 68: Videoübertragung auf der Website

Quelle: eigene Abbildung

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

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

Abbildung 69: 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 absolut gesetzt.

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

Abbildung 70: 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 (da Webserver auf ESP32-CAM gehostet wird) und dem Port 81 geöffnet. Wenn nun eine Nachricht auf der Websocket-Verbindung ankommt (siehe Videoübertragung 4.5) wird diese extrahiert. Das mitgesendete BLOB wird in einem neuen 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 ``-Tag erstellt. Die Quelle 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 71: Videoübertragung im JavaScript-Code

Quelle: eigene Abbildung

4.6.3 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 72: Menü-Icon auf der Website

Quelle: eigene Abbildung



Abbildung 73: Sidebar mit Sensordaten

Quelle: eigene Abbildung

Im HTML-Code wird im Body-Bereich wird das Icon als HTML -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 mir 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 74: 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 Höhe 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 75: 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 76: 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 77: 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 78: Statusbox auf der Website

Quelle: eigene Abbildung

Connected

Disconnected

Abbildung 79: Statusbox:
Verbunden

Abbildung 80: Statusbox:
Verbindung getrennt

Im HTML-Code wird die Status-Box als HTML ``-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 81: 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 82: CSS-Klassen für die Statusbox

Quelle: eigene Abbildung

Im JavaScript-Code werden die `onopen()` und `onclose()` Events der WebSocket Verbindung gehandelt. Bei einem Verbindungsauftakt (`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 Verbindungsauftakt 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 83: JavaScript Funktionen für den Verbindungsstatus

Quelle: eigene Abbildung

4.7 Herausforderungen und Optimierungen

4.7.1 Probleme bei der WebSocket Kommunikation

4.7.2 Latenz- und Performance Optimierungen

Kameraoptimierungen

Als Bildformat wird JPEG verwendet, um Speicherplatz zu sparen. Alternativ wäre RGB565 oder YUV422 möglich, jedoch benötigen diese aufgrund fehlender Kompression mehr Speicher und verlängern die Übertragungsdauer.

Für die Bildauflösung wird QVGA (320x240 Pixel) definiert. Alternativ wären noch VGA (640x480), SVGA (800x600) oder UXGA (1600x1200) möglich, jedoch benötigen diese mehr Speicher und mehr Bandbreite und verlängern dadurch die Übertragungsdauer.

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. (siehe Kamera-Initialisierung 46)

4.8 (Fazit und Ausblick)

4.8.1 (Mögliche Erweiterungen und Verbesserungen)

5 Gehäuse

5.1 Planung und Design

Das Gehäuse des Lastenroboters wurde entwickelt, um die Sicherheit und Stabilität beim Transport von schweren Lasten zu gewährleisten. Dabei lag der Fokus auf einer stabilen Bauweise, damit der Lastenroboter ein Gewicht von zirka 25 Kilo aushaltet.

Wichtigsten Aspekte bei der Planung:

1. Stabilität: Das Gehäuse muss das Eigengewicht des Roboters aber auch das Gewicht der Last aushalten.
2. Größe: Bei der Größe ist es wichtig darauf zu achten dass alle Komponenten sich sehr gut im Gehäuse aussehen.
3. Fahrverhalten: Durch die Verwendung von vier Rädern, die jeweils über einen Motor angetrieben werden, soll eine gute Manövrierfähigkeit erreicht werden.
4. Integration der Elektronik: Alle Komponenten müssen sicher im Gehäuse untergebracht werden.
5. Gewichtsmessung: Die Wägezelle muss in das Gehäuse integriert werden, um eine präzise Messung zu ermöglichen.
6. Wartungsfreundlichkeit: Bauteile sollten leicht zugänglich sein, um Reparaturen oder Anpassungen durchführen zu können.

Mechanische Konstruktion

Nach der Planung der wichtigsten Aspekte wurden zwei 3D-Modelle des Lastenroboters in Fusion 360 entworfen.

Erstes Modell

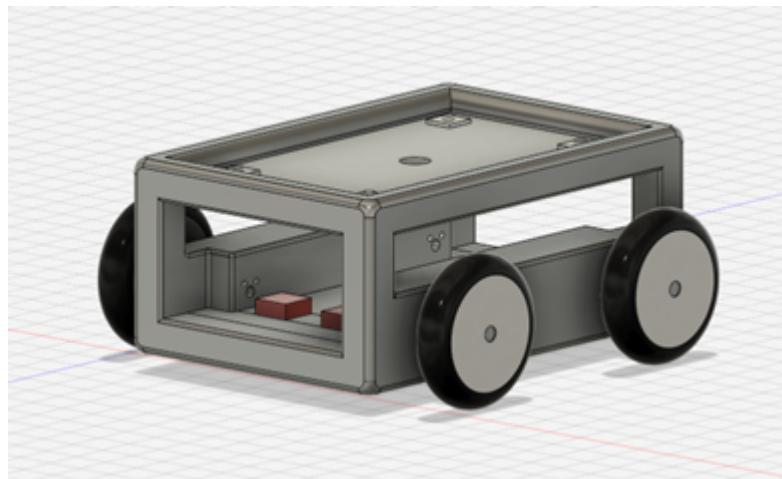


Abbildung 84: Erstes Modell

Quelle: eigene Abbildung

Das erste Modell des Lastenroboters diente hauptsächlich als Prototyp. Es wurden keine genauen Maße beachtet, sondern dient als grobe Vorstellung wie der Lastenroboter aussehen könnte. Wichtig dabei war es die grundlegenden Komponenten wie Motoren, Räder und Wägezelle zu skizzieren, um die allgemeine Funktionalität sicherzustellen.

In diesem frühen Entwicklungsstadium lag der Fokus auf der Mechanik und der Stabilität des Designs. Das erste Modell war sehr wichtig, um Schwachstellen und Optimierungsmöglichkeiten zu identifizieren und half dabei, das zweite Modell präzise und durchdacht zu planen.

Zweites Modell

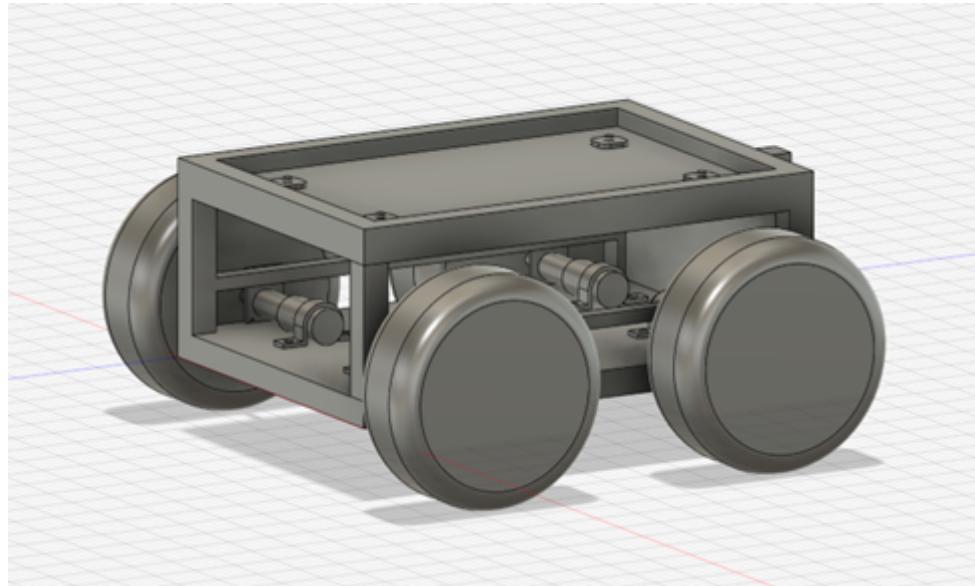


Abbildung 85: Zweites Modell

Quelle: eigene Abbildung

Anhand der Erkenntnisse aus dem ersten Modell wurde das zweite Modell entwickelt, das als fertiges Modell für den Bau des Lastenroboters dient. In dieser Version wurden exakte Maße festgelegt und die genaue Position der Komponenten wie Räder, Motoren, Kamera und Wägezelle festgelegt.

Das Wichtigste beim Entwickeln ist die Größe des Gehäuses. Das Gehäuse basiert auf einem rechteckigen Stahlrahmen. Dieser Rahmen ist wichtig, weil er die Grundstruktur und Stabilität gewährleistet. Der rechteckige Rahmen des Lastenroboters hat eine Höhe von 220mm, eine breite von 400mm und eine Länge von 560mm. Die Vierkantstahlrohre haben eine Größe von 30mm mal 40mm.

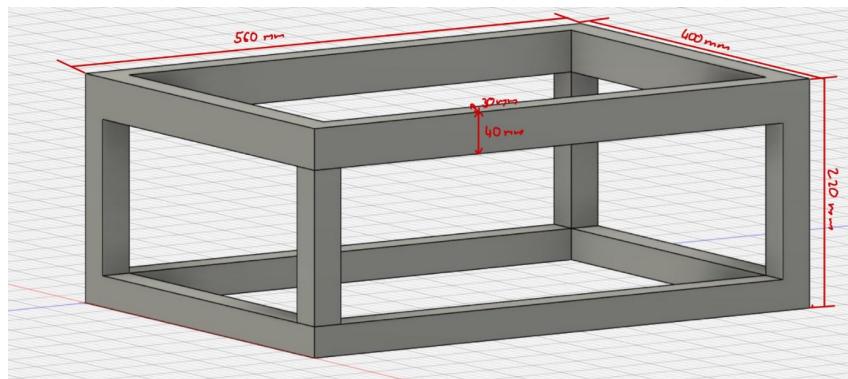


Abbildung 86: Rahmen

Quelle: eigene Abbildung

Im nächsten Schritt wurde Halterung für die Elektronik Komponenten entwickelt. Dabei handelt es sich um eine Platte, die auf der Unterseite mit zwei Querstreben verstärkt wird. Die Größe der Platte beträgt 34mm mal 50mm und hat eine Materialstärke von 5mm. Die Querstreben haben eine Länge von 50mm, sind 25mm hoch und 25mm breit.

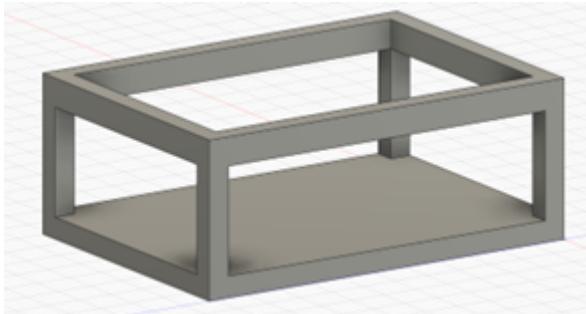


Abbildung 87: Halterung

1

Quelle: eigene Abbildung

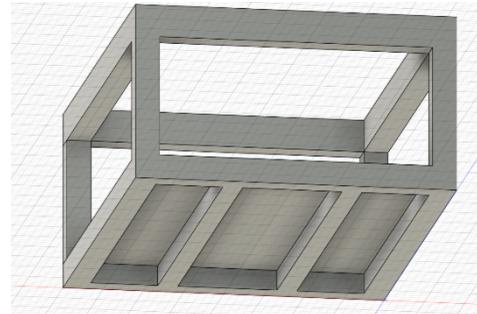


Abbildung 88: Halterung

2

Quelle: eigene Abbildung

Danach wurde die Halterung für die Motoren entwickelt. Dabei ist zu beachten, dass sich die Motoren während des Betriebs weder verdrehen noch verrutschen. Dies ist entscheidend, da die Motoren ein hohes Drehmoment erzeugen, das ohne eine stabile Befestigung zu Instabilitäten führen könnte. Die Motoren sind auf einer Querstange befestigt damit sie sich nicht mehr verdrehen können. Die Querstangen haben eine Länge von 50mm und sind 15mm hoch und 20mm breit. Zusätzlich sind die Motoren auch mit einer Schelle am Boden fixiert, um ungewolltes verrutschen zu vermeiden.

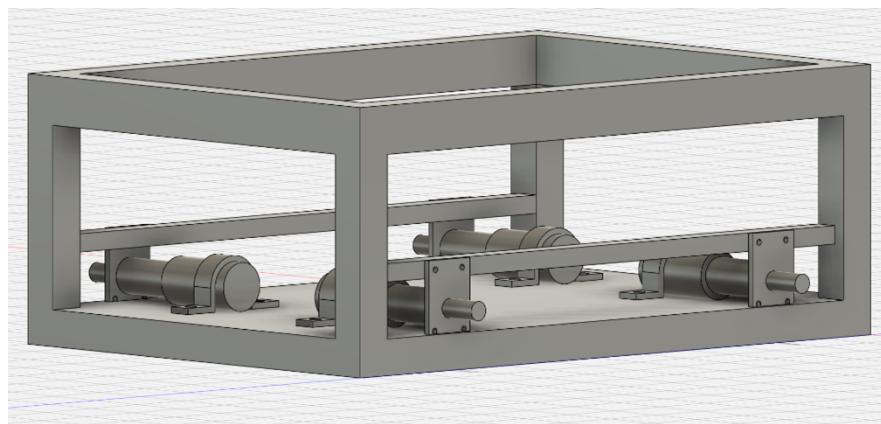


Abbildung 89: Motorenhalterung

Quelle: eigene Abbildung

Nach dem die Halterung für die Motoren entwickelt wurde, erfolgt die Auswahl und Befestigung der Reifen. Die Reifen spielen eine entscheidende Rolle für das Fahrverhalten.

Die Räder bestehen aus Gummi für eine gute Bodenhaftung. Jedes der vier Reifen ist direkt an der Motorachse befestigt. Die Motorachse hat einen Durchmesser von 16mm. Die Motorachse verfügt über ein M8-Gewinde, das für die Befestigung der Räder verwendet wird. Die Räder werden mit einer M8-Schraube in die Motorachse geschraubt.

Wichtig bei der Entwicklung war die Radgröße damit das Gehäuse genug Bodenabstand hat. Es wurden Reifen gewählt mit einem Durchmesser von 260mm und einer Breite von 85mm.

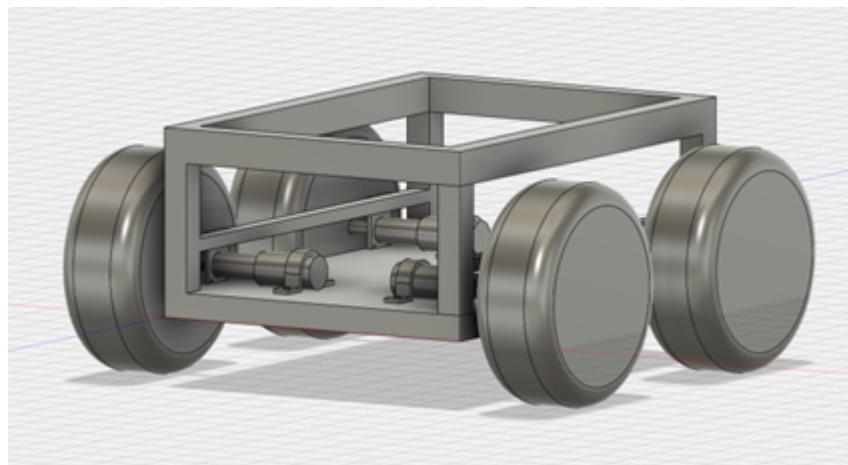


Abbildung 90: Räder

Quelle: eigene Abbildung

Ein weiterer großer Punkt bei der Planung war die Gewichtsmessung. Die Gewichtsmessung erfolgt über vier Wägezellen die in einem Rechteck auf einer Platte angeordnet sind. Auf die Wägezellen wird eine Platte montiert, auf die die Last gestellt wird.

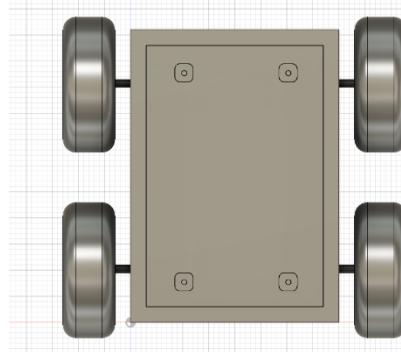


Abbildung 91: Wägezellen

Quelle: eigene Abbildung

Damit die Wägezellen sicher fixiert sind und nicht verrutschen, werden sie in einer 3D gedruckten Halterung befestigt. Die Halterung hat eine Einkerbung, die genau für die Wägezelle abgestimmt ist, damit die Wägezelle nicht mehr verrutschen kann. Es gibt zusätzlich eine Einkerbung für die Leitungen, damit diese nicht beschädigt werden. Am Rand der Halterung sind zwei Löcher damit die Halterung auf der Platte angeschraubt werden kann.

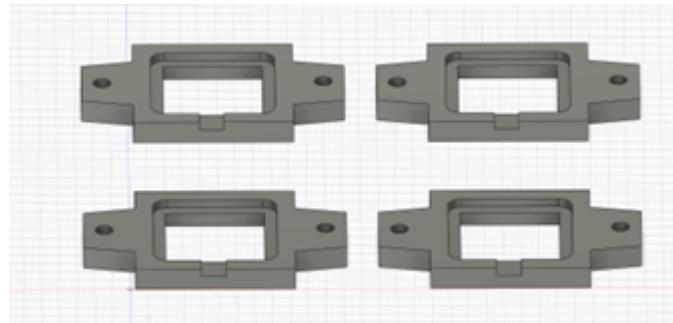


Abbildung 92: Wägezellenhalterung

Quelle: eigene Abbildung

Im letzten Schritt des Designs wurden zwei Platten an der Vorderseite und Rückseite des Lastenroboters angebracht. Die hintere Platte dient für die Befestigung des An/Aus Schalters und des Ladeanschlusses für den Lastenroboter. Auf der vorderen Platte soll die Kamera montiert werden.

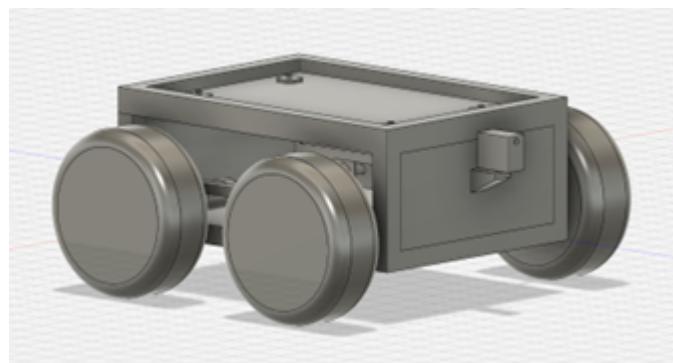


Abbildung 93: modell-fertig

Quelle: eigene Abbildung

Materialwahl für das Gehäuse

Für das Gehäuse wurde Stahl als Hauptmaterial gewählt, weil es sehr robust und widerstandsfähig gegenüber äußeren Einflüssen ist. Ein weiter Grund, warum Stahl als Hauptmaterial verwendet wurde, ist die einfache Verarbeitung. Stahl kann man gut Schweißen aber auch verschrauben.

5.2 Realisierung

5.3 Materialliste

6 Platine

6.1 Grundschaltung

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 Abstandssensor

8.1.1 Grundprinzip

Der HC-SR04 ist ein Ultraschallsensor, der mithilfe von Schallwellen Entfernungen messen kann. Sein Funktionsprinzip basiert auf der Laufzeitmessung von Schallwellen. Dabei sendet der Sensor hochfrequente Ultraschallwellen aus, die von Objekten in der Umgebung reflektiert und anschließend wieder vom Sensor empfangen werden. Durch die Messung der Zeit, die der Schall für den Hin- und Rückweg benötigt, kann die Entfernung zum Objekt bestimmt werden.

1. Senden eines Ultraschallimpulses

Der Ultraschallsensor verfügt über zwei Hauptkomponenten, der Trigger und das Echo. Zu Beginn des Messvorgangs sendet der Trigger einen kurzen Ultraschallimpuls aus. Dieser Impuls besteht aus acht Schallwellen mit einer Frequenz von 40 kHz, die sich mit Schallgeschwindigkeit in der Luft ausbreiten.

2. Reflexion des Signals

Die ausgesendeten Ultraschallwellen bewegen sich durch die Luft, bis sie auf ein Hindernis treffen. Sobald dies geschieht, werden die Wellen reflektiert und in Richtung des Sensors zurückgesendet. Je nach Entfernung des Hindernisses dauert diese Reflexion unterschiedlich lange.

3. Empfangen des Echos

Der Echo-Sensor registriert das reflektierte Signal und misst die Zeitspanne zwischen dem Aussenden und dem Empfang des Ultraschallimpulses. Diese Laufzeit des Schalls ist direkt proportional zur Entfernung des Hindernisses. Je weiter das Objekt entfernt ist, desto länger dauert es, bis das Echo zurückkehrt.

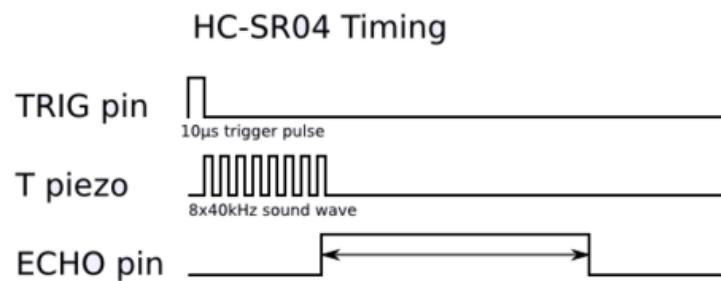


Abbildung 94: HC-SR04 Timing Diagram

Quelle:

https://wiki.hshl.de/wiki/index.php/Ultraschallsensor_HC-SR04

4. Berechnung der Entfernung

Die Entfernung zum Hindernis kann mit der folgenden Formel berechnet werden:

$$Distanz = \frac{Schallgeschwindigkeit \times Laufzeit}{2}$$

Da sich der Ultraschall mit einer Geschwindigkeit von ca. 343 m/s (bei 20°C) in der Luft ausbreitet und das Signal hin und zurück läuft, wird die Laufzeit durch zwei geteilt. Diese Berechnung ermöglicht eine präzise Abstandsmessung mit einem Ultraschallsensor.

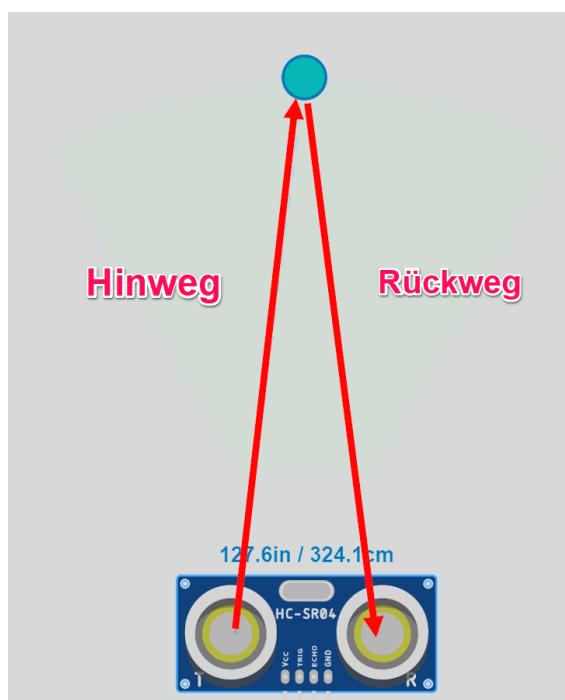


Abbildung 95: Beispiel für Messung

Quelle: <https://www.tinkercad.com/>

8.1.2 Schaltungsaufbau

Der Aufbau der Schaltung ist relativ einfach, da der Sensor nur vier Pins hat.

Der Sensor verfügt über folgende Anschlüsse:

- **VCC** → Versorgungsspannung (5V)
- **GND** → Masse
- **Trigger (TRIG)** → Eingangssignal zum Starten der Messung
- **Echo (ECHO)** → Ausgangssignal für die Messzeit

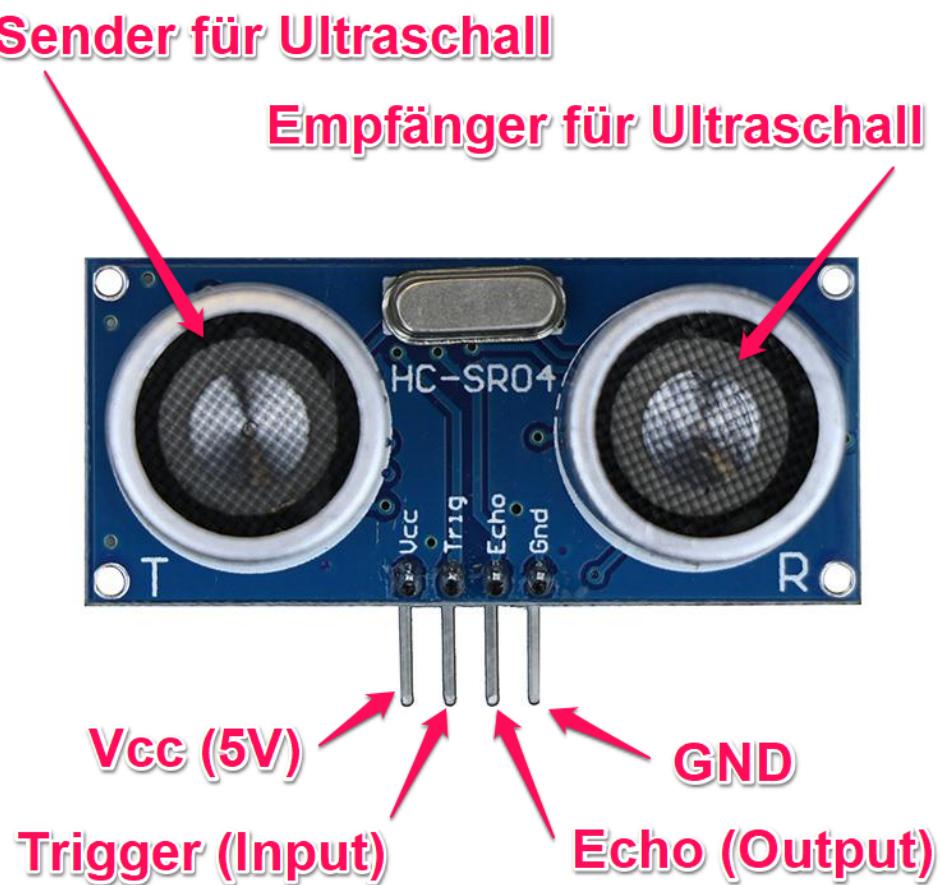


Abbildung 96: HC-SR04 Pinbelegung

Quelle: <https://at.rs-online.com/web/p/bbc-micro-bit-add-ons/2153181>

Der HC-SR04 Ultraschallsensor kann an beliebige digitale GPIO-Pins des ESP32 angeschlossen werden, solange man diese als Eingang (für Echo) und Ausgang (für Trigger) benutzen kann.

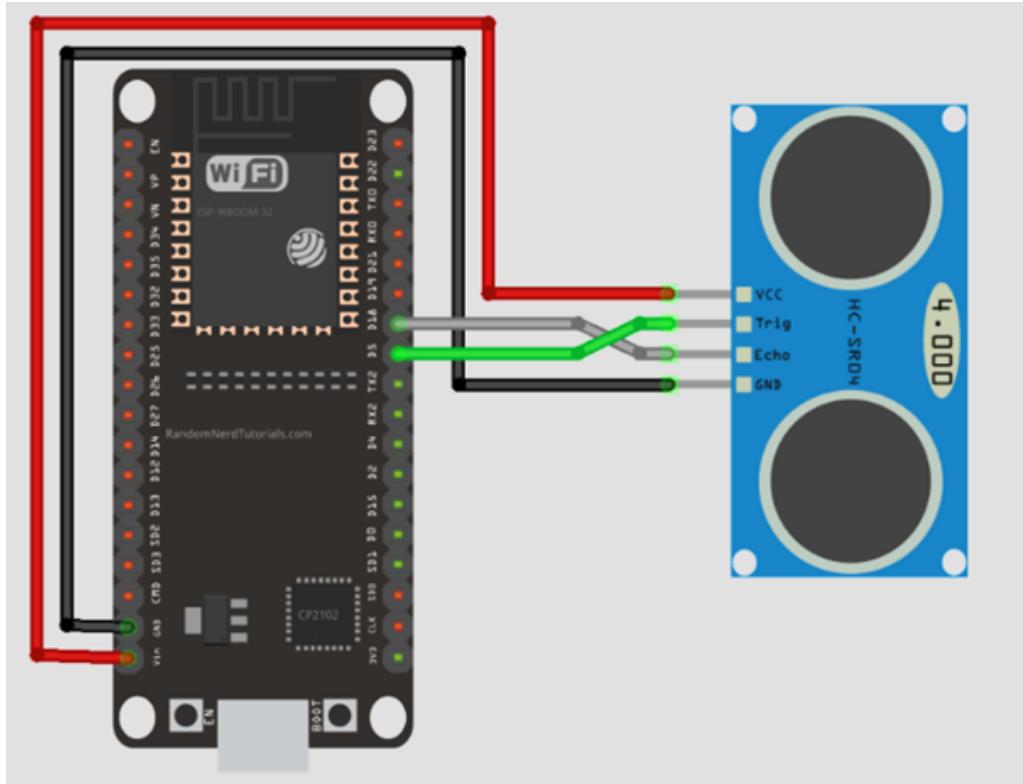


Abbildung 97: HC-SR04 Schaltungs Beispiel

Quelle: <https://randomnerdtutorials.com/esp32-hc-sr04-ultrasonic-arduino/>

8.1.3 Nutzung

Unser Lastenroboter hat zwei HC-SR04 Ultraschallsensoren verbaut, diese messen den Abstand vorne und hinten am Roboter. Diese Messungen sind wichtig für eine Sicherheitsfunktion, die verhindert, dass der Roboter unkontrolliert gegen ein Hindernis fährt.

Bei zu hoher Geschwindigkeit bremst der Roboter automatisch, um eine Kollision zu vermeiden. Fährt er jedoch langsam auf ein Objekt zu, bleibt die Bremse deaktiviert.

Dieses Verhalten ist ähnlich wie bei einem Auto, dass präzise Heranfahren an ein Hindernis ist bei langsamer Geschwindigkeit möglich, beispielsweise beim Einparken oder Positionieren. Bei hoher Geschwindigkeit kann es aber passieren, dass der Roboter jedoch nicht rechtzeitig stoppt, weshalb die automatische Bremse greift. Dadurch wird verhindert, dass der Roboter mit voller Wucht gegen ein Hindernis, wie etwa eine Mauer, fährt.

8.1.4 Code

Um die Ultraschallsensoren leicht zu verwenden, wird die HCSR04-Bibliothek eingebunden. Diese Bibliothek vereinfacht die Steuerung der HC-SR04-Sensoren, indem sie die Entfernungsmessung automatisch berechnet.

```
#include "HCSR04.h"
```

Abbildung 98: HC-SR04.h Bibliothek

Quelle: eigene Abbildung

Nach dem Einbinden der Bibliothek, werden Objekte der Klasse UltraSonicDistanceSensor erstellt, mit dem später auf die Ultraschallsensoren zugegriffen und Entfernungen gemessen werden können.

```
distanceSensor_rear(23, 19); ist der Sensor für die hintere Messung  
distanceSensor_front(13, 12); ist der Sensor für die vordere Messung.
```

```
UltraSonicDistanceSensor distanceSensor_rear(23, 19);  
UltraSonicDistanceSensor distanceSensor_front(13,12);
```

Abbildung 99: Erstellung von Objekten der Klasse UltraSonicDistanceSensor

Quelle: eigene Abbildung

Danach wird die ganze Zeit in der `loop()` die aktuelle Entfernung für den vorderen, als auch für den hinteren Ultraschallsensor ermittelt. Dazu wird die Methode „`measureDistanceCm()`“ der jeweiligen Sensorobjekte aufgerufen, um die Entfernung in cm zu messen. Die Messergebnisse werden in die dementsprechende float-Variable gespeichert.

```
distance_front = distanceSensor_front.measureDistanceCm();  
distance_rear = distanceSensor_rear.measureDistanceCm();
```

Abbildung 100: Messung der Entfernung in der `loop()`-Funktion

Quelle: eigene Abbildung

Mit einer if-Abfrage prüft der Code, ob der Roboter zu nah an ein Hindernis kommt und dabei zu schnell ist. Wenn die gemessene Entfernung vorne oder hinten zwischen fünf und zwanzig Zentimetern liegt und die Geschwindigkeit mindestens fünfzig beträgt, wird die Funktion `stop()` ausgeführt. Dadurch stoppt der Roboter automatisch, um eine Kollision zu vermeiden.

Der Bereich von fünf bis zwanzig cm wurde gewählt, weil der Sensor bei sehr großen Entferungen, ungenaue oder falsche Signale liefern kann. Dadurch das nur Messwerte innerhalb dieses Bereichs berücksichtigt werden, wird sichergestellt, dass der Roboter nur dann stoppt, wenn tatsächlich ein Hindernis erkannt wird.

```
if(((distance_front >= 5 && distance_front <= 20) && speed_cb >= 50) ||  
((distance_rear >= 5 && distance_rear <= 20) && speed_cb >= 50)){  
    stop();  
}
```

Abbildung 101: if-Abfrage für Mindestabstand und Geschwindigkeit

Quelle: eigene Abbildung

8.2 Gewichtsmessung

8.2.1 Grundprinzip

Um das Gewicht, das sich auf dem Roboter befinden zu wiegen, werden Wägezellen benutzt. Wägezellen sind Widerstände, die sich unter Krafteinfluss verändern. Diese Veränderung des Widerstandes kann man dann benutzen, um das darauf drückende Gewicht zu messen. Wägezellen bestehen meistens aus einem Federkörper und einem Dehnmessstreifen. Der Federkörper ist ein biegsames Metallstück, das sich verbiegt, wenn Gewicht darauf einwirkt. Diese Dehnung wird dann mit dem Dehnmessstreifen gemessen.

Wägezellen gibt es in unterschiedlichen Größen, von kleinen Modellen für geringe Lasten bis hin zu größeren Varianten für hohe Gewichte. Beim Lastenroboter kommen „Halbbrücken-Wägezellen“ zum Einsatz.



Abbildung 102: Halbbrücken-Wägezelle

Quelle: [https://bienenwaage.shop/auswahl-der-](https://bienenwaage.shop/auswahl-der-waegezelle)

waegezelle

Zum Auslesen der Wägezellen wird ein HX711 verwendet. Der HX711 ist ein präziser 24-Bit-Analog-Digital-Wandler, der speziell für Wägezellen entwickelt wurde. Er verstärkt und digitalisiert die schwachen Signale der Sensoren, sodass sie von Mikrocontrollern verarbeitet werden können. Die Daten des HX711 werden anschließend über eine serielle Datenverbindung an den ESP32 übertragen. Der Mikrocontroller empfängt die digitalen Signale und verarbeitet sie weiter, um das Gewicht präzise zu berechnen.

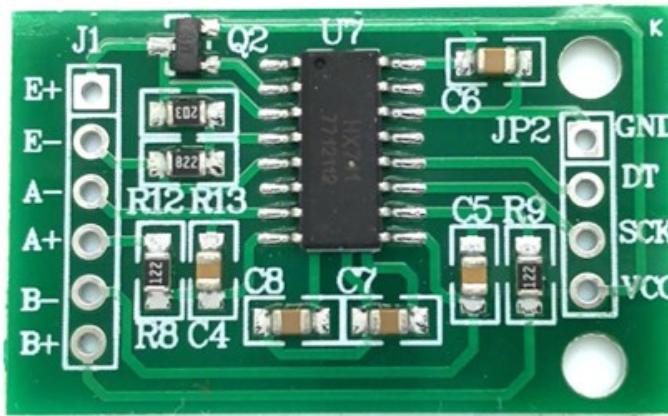


Abbildung 103: HX711 24-bit A/D-Wandler

Quelle: <https://www.berrybase.at/hx711-24-bit-a-d-gewichtssensor>

1. Kraftaufnahme durch die Wägezellen

Wird eine Last auf das System aufgebracht, verformen sich die Halbbrücken-Wägezellen minimal. Diese Verformung führt zu einer Änderung des elektrischen Widerstands der Dehnungsmessstreifen in der Wägezelle.

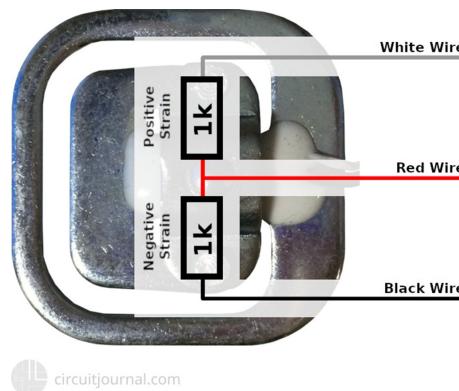


Abbildung 104: Prinzip einer
Wägezelle

Quelle:

<https://circuitjournal.com/50kg-load-cells-with-HX711>

2. Widerstandsänderung und Signalbildung

Die Halbbrücken-Wägezellen sind Teil einer Wheatstone-Brückenschaltung. Durch die Widerstandsänderung entsteht eine kleine Spannungsdifferenz, die proportional zur aufgebrachten Last ist.

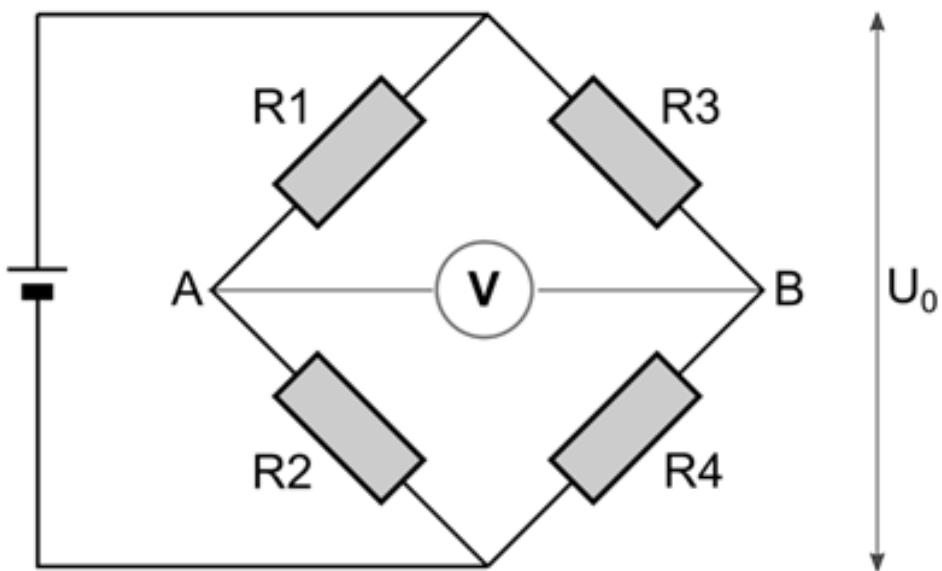


Abbildung 105: Wheatstone-Brückenschaltung

Quelle: <https://wolles-elektronikkiste.de/dehnungsmessstreifen>

3. Signalverstärkung durch den HX711

Der HX711 nimmt diese schwache Differenzspannung auf und verstärkt sie mit einem internen Verstärker. Die Verstärkung kann auf 32x, 64x oder 128x eingestellt werden, um auch sehr kleine Laständerungen messbar zu machen.

4. Analog-Digital-Wandlung durch den HX711

Die verstärkte Spannung wird im HX711 in ein digitales 24-Bit-Signal umgewandelt. Dadurch kann das Gewicht mit hoher Genauigkeit erfasst werden.

5. Analog-Digital-Wandlung durch den HX711

Der ESP32 empfängt die digitalen Werte über eine serielle Datenverbindung. Dann werden die Rohdaten interpretiert und in eine Gewichtseinheit (z. B. Kilogramm) umgerechnet. Eine vorherige Kalibrierung stellt sicher, dass die Messung korrekt erfolgt.

8.2.2 Schaltungsaufbau

Die vier Wägezellen sind in einer Wheatstone-Vollbrücke verbunden, dadurch kann die Gewichtsmessung präziser erfolgen und die Belastung auf die Plattform gleichmäßig erfasst werden, unabhängig von der Position der Last. Die vier Wägezellen sind so verbunden, dass ihre Signale zusammengeführt und an die differenziellen Eingänge des HX711 weitergegeben werden.

Für den Anschluss werden die folgenden Pins des HX711 genutzt:

- **E+ (Excitation +)** – Versorgungsspannung für die Wägezellen
- **E- (Excitation -)** – Masse der Wägezellen
- **A+ (Signal +)** – Positives Ausgangssignal der Brückenschaltung
- **A- (Signal -)** – Negatives Ausgangssignal der Brückenschaltung

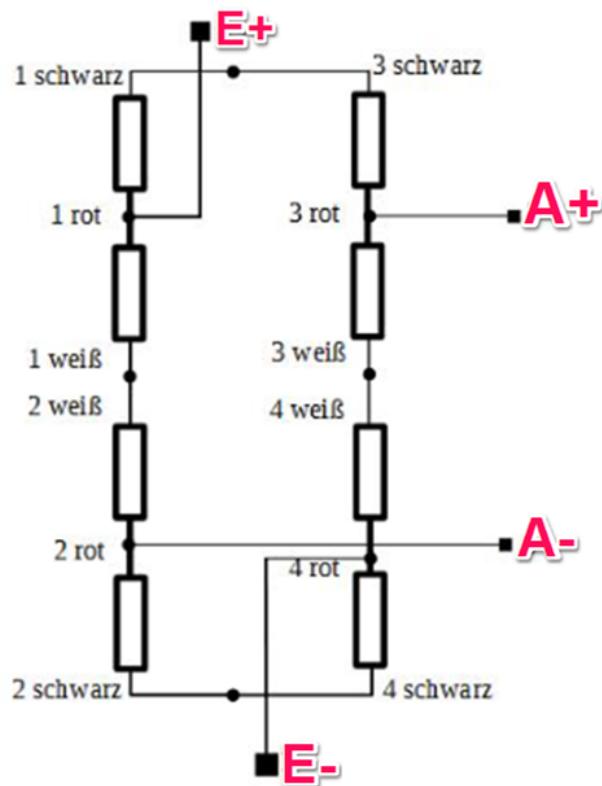


Abbildung 106: Beschaltung der Wägezellen
Quelle:

https://beelogger.de/stockwaage/aufbauanleitung_mit_halbbrueckensensoren

Der HX711 überträgt die digitalisierten Messwerte an den ESP32 über eine serielle Datenverbindung.:

- **DT (Data)** – Dieser Pin überträgt die digitalen Messwerte vom HX711 und wird mit dem I2C_SDA-Pin (GPIO 21) des ESP32 verbunden.
- **SCK (Clock)** – Dieser Pin steuert die Taktung der Datenübertragung und wird mit dem I2C_SCL-Pin (GPIO 22) des ESP32 verbunden.

Obwohl der HX711 kein echtes I2C-Protokoll verwendet, wird er oft ähnlich angeschlossen und mit einer einfachen seriellen Bit-Kommunikation ausgelesen. Der ESP32 sendet dabei regelmäßig ein Signal über den SCK-Pin, um die Datenübertragung zu starten, und liest anschließend über den DT-Pin die Gewichtswerte aus.

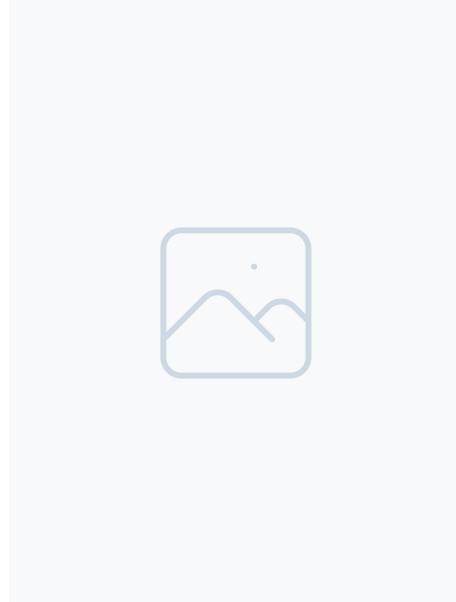


Abbildung 107: Verkabelung der
Wägezellen mit HX711
Quelle: eigene Abbildung

8.2.3 Nutzung

Die Wägezellen dienen dazu, das Gewicht auf dem Lastenroboter präzise zu messen. Der Mikrocontroller verarbeitet die vom HX711 empfangenen Daten und sendet sie in Echtzeit an die Website, auf der das aktuelle Gewicht des Roboters in einem Untermenü angezeigt wird.

Dadurch kann jederzeit überprüft werden, wie viel Last sich auf dem Roboter befindet. Diese Funktion trägt zur Überwachung des Gewichts bei und hilft Überlastungen bei den Transportprozessen zu vermeiden.

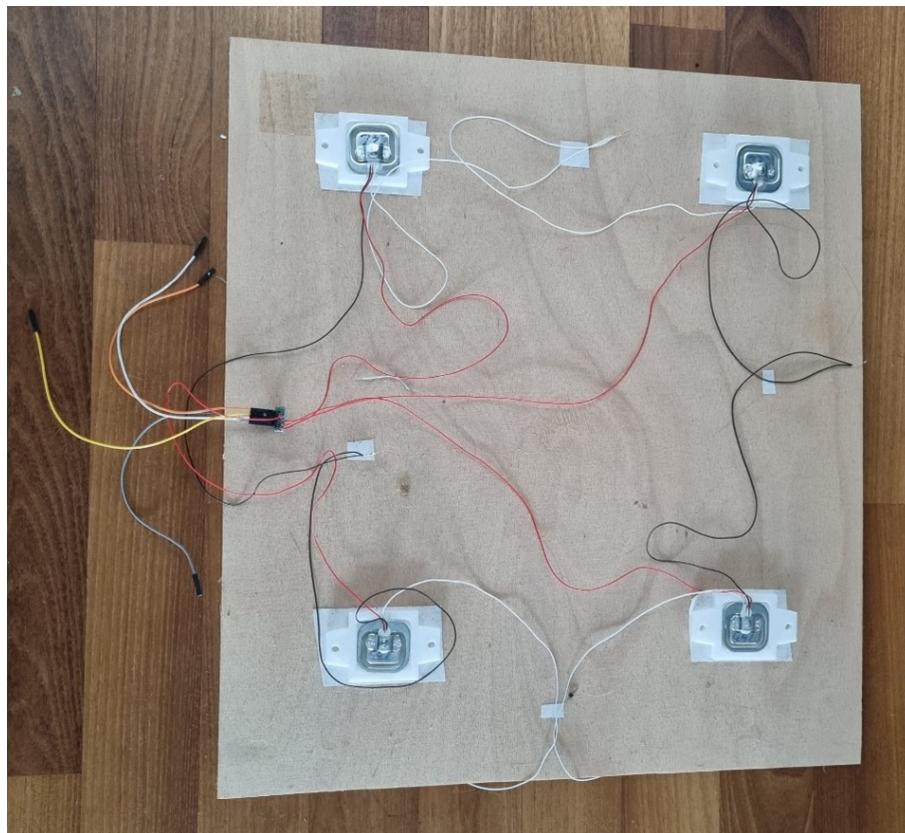


Abbildung 108: Testaufbau der Wägezellen

Quelle: eigene Abbildung

8.2.4 Code

Um die Wägezellen mit dem HX711 einfach auszulesen, wird die HX711_ADC- Bibliothek verwendet. Diese Bibliothek vereinfacht durch ihre vielen fertigen Funktionen das Auslesen und das Umrechnen der Daten. Nach Einbinden der Bibliothek, werden die GPIOs für das Auslesen festgelegt. Mit diesen wird dann ein Objekt der Klasse HX711_ADC erstellt, um auf den HX711 zuzugreifen. Außerdem wird eine Float-Variable für das Gewicht deklariert.

```
#include <HX711_ADC.h>
const int HX711_dout = 21;
const int HX711_sck = 22;
HX711_ADC LoadCell(HX711_dout, HX711_sck);
float weight = 0.0;
```

Abbildung 109: Einbinden der HX711_ADC-Bibliothek

Quelle: eigene Abbildung

Dannach wird mit `LoadCell.begin()`; der HX711-Sensor in der `Setup()`-Funktion gestartet. Anschließend wird die Variable `calibrationValue` als float definiert und auf 696.0 gesetzt. Dieser Wert dient zur Kalibrierung der Messung und muss an die spezifische Wägezelle angepasst werden.

Die Variable `stabilizingtime` vom Typ `unsigned long` wird auf 2000 Millisekunden gesetzt. Diese gibt an, wie lange der Sensor Zeit bekommt, um sich nach dem Start zu stabilisieren.

Die Variable `_tare` ist ein Boolean und wird auf `true` gesetzt. Das bedeutet, dass der Sensor beim Start automatisch „tarieren“ soll, also das aktuelle Gewicht als Nullpunkt speichert.

Schließlich wird mit `LoadCell.start(stabilizingtime, _tare)`; der Messvorgang gestartet, wobei die angegebene Stabilisierungszeit und die automatische Tara-Funktion berücksichtigt werden.

```
LoadCell.begin();
float calibrationValue;
calibrationValue = 696.0;
unsigned long stabilizingtime = 2000;
boolean _tare = true;
LoadCell.start(stabilizingtime, _tare);
```

Abbildung 110: `Setup()` - Code für den HX711

Quelle: eigene Abbildung

Dann wird noch überprüft, ob die Tara-Funktion des HX711 erfolgreich abgeschlossen wurde oder ob ein Timeout-Fehler aufgetreten ist.

Zuerst wird geprüft, ob die Tara-Funktion zu lange gedauert hat und fehlgeschlagen ist. Falls dies der Fall ist, wird eine Fehlermeldung über die serielle Schnittstelle ausgegeben. Anschließend wird eine Endlosschleife gestartet, die das Programm anhält.

Falls kein Timeout auftritt, wird die Kalibrierung durchgeführt. Danach wird über die serielle Schnittstelle eine Bestätigung ausgegeben, dass die Initialisierung erfolgreich abgeschlossen wurde.

```
if (LoadCell.getTareTimeoutFlag())
{
    Serial.println("Timeout, check MCU>HX711 wiring and pin designations");
    while (1)
        ;
}
else
{
    LoadCell.setCalFactor(calibrationValue);
    Serial.println("Startup is complete");
}
```

Abbildung 111: Timeout Abfrage in der Setup() ; -Funktion

Quelle: eigene Abbildung

In der loop() -Funktion wird kontinuierlich das Gewicht gemessen. Die getData() -Funktion ruft dabei den aktuellen Messwert vom HX711 A/D-Wandler ab und speichert ihn in der Variablen „weight“.

Dieser Wert stellt das aktuell gemessene Gewicht dar, das mithilfe der Kalibrierung in eine gewichtsspezifische Einheit umgerechnet wird. Dieser Wert kann dann auf die Website übertragen werden.

```
weight = LoadCell.getData();
```

Abbildung 112: Gewichtsmessung in der loop() ; -Funktion

Quelle: eigene Abbildung

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 Extentions mit vielen nützlichen Features und Sprachen immer wieder erweitert werden.²

9.3.1 Setup

Um ein Projekt in VS-Code erstellen zu können, müssen einige Schritte befolgt werden. Zuallererst 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 Speicherungsort des Projektes. Bei unserem Projekt wählten wir das Board “Espressif ESP32 Dev Module“ und als Framework “Arduino“, da wir einen EPS32 zum Programmieren verwendeten.

²<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 mehrere 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.³

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.⁴

HCSR04

Die HCSR04.h-Bibliothek ermöglicht es mit einem ESP32, das HC-SR04 Ultraschallmodul zur Entfernungsmessung zu nutzen. Sie erlaubt die Verwendung mehrerer Sensoren gleichzeitig, um die aktuelle Distanz in cm zu erfassen.⁵

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

⁴<https://arduinojson.org/>

⁵<https://docs.arduino.cc/libraries/hcsr04-ultrasonic-sensor/>

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.⁶

ESp32Servo

Die ESP32Servo-Bibliothek ermöglicht es, Servomotoren mit einem ESP32 einfach zu steuern. Sie unterstützt viele Servotypen und funktioniert ähnlich wie die Arduino Servo-Bibliothek⁷. Zusätzlich bietet sie zwei Funktionen zur Anpassung der PWM-Timerbreite des ESP32 für genauere Steuerung.⁸

Adafruit_MCP23x17

Die Adafruit MCP23017 Arduino Library ermöglicht die Nutzung des MCP23017 I/O-Expanders mit Arduino und ESP32. Dadurch lassen sich 16 zusätzliche digitale Ein- und Ausgänge über I2C oder SPI steuern. Die Bibliothek bietet einfache Funktionen für Input, Output, Pull-ups und Interrupts. So können Projekte mit mehr Ein- und Ausgängen erweitert werden, ohne zusätzliche Pins am Mikrocontroller zu belegen.⁹

HX711_ADC

Die HX711_ADC-Bibliothek ermöglicht die Nutzung des HX711 ADC-Moduls mit Arduino und ESP32 zur präzisen Messung von Gewichtssensoren (Wägezellen). Sie bietet eine verbesserte Signalverarbeitung, Tare-Funktion und Kalibrierungsfunktionen. Die Bibliothek erleichtert das Auslesen der Sensordaten und sorgt für eine stabile Gewichtsmessung, ohne komplexe Signalverarbeitung manuell durchführen zu müssen.¹⁰

9.4 LaTex

9.5 GitHub

⁶<https://github.com/Links2004/arduinoWebSockets>

⁷<https://docs.arduino.cc/libraries/servo/>

⁸<https://github.com/madhephaestus/ESP32Servos>

⁹<https://github.com/adafruit/Adafruit-MCP23017-Arduino-Library>

¹⁰https://github.com/olkal/HX711_ADC

10 Abbildungsverzeichnis

Abbildungsverzeichnis

1	Porträt Daniel Schauer	9
2	Porträt Simon Spari	9
3	Porträt Felix Hocegger	9
4	Lastenroboter Projekt-Grobplan	10
5	Motoren	12
6	BLDC-Motor	13
7	Phasensteuerung	13
8	Motorteiber-ZS-X11H	14
9	Motorphasen Verbindung	15
10	Pins für die PWM	15
11	Pin für die Bremse und den Richtungswechsel	16
12	Versorgungs-Pins	16
13	Hall-sensoren Eingänge	17
14	Schaltungsaufbau mit einem Motor	18
15	Hall-Sensoren Eingänge	19
16	Hall-Sensoren-Motor	19
17	Motor Phasen	19
18	Treiber Phasen	19
19	Versorgung der Motoren und Treiberplatine	19
20	Ansteuerung der Treiberplatine	20
21	Motoren-GPIO Konfiguration	21
22	Motoren-GPIO Konfiguration	22
23	Adafruit_MCP23X17.h-Bibliothek	22
24	MCP-Setup	23
25	Initialisierung Motoren	23
26	Enumeration von Bewegungsrichtungen	24
27	Strings zu Enum Variable umwandeln	25
28	navigate() in loop()	25
29	Überprüfung auf Änderung und Speed in Integer umwandeln in navigate()-Funktion	26
30	Motorbremse deaktivieren in navigate()-Funktion	26
31	switch-case Struktur in navigate()-Funktion	27

32	PWM mit analogWrite übertragen und letzte Werte speichern in <code>navigate()</code> -Funktion	28
33	<code>ESPAsyncWebServer.h</code> -Bibliothek	30
34	WebServer Netzwerkkonfiguration	30
35	Webserver Setup	31
36	SPIFFS Initialisierung	32
37	<code>readfile()</code> -Funktion	33
38	Bibliotheken für die Kommunikation	33
39	Setup ESP32	34
40	<code>onWebSocketEvent()</code> -Funktion	35
41	JSON-Beispiel für Steuerung	36
42	JSON-Beispiel für Kamerasteuerung	36
43	JSON-Beispiel für Lichtsteuerung	36
44	<code>handleWebSocketMessage()</code> -Funktion	37
45	Kamera-GPIO Konfiguration	38
46	Kamera-Initialisierung	39
47	Videoübertragung über WebSocket	40
48	Steuerkreuz auf der Website	41
49	Steuerkreuz HTML-Code	41
50	CSS-Klassen <code>.dpad-container</code> und <code>.button</code>	42
51	CSS-Richtungsklassen	43
52	EventListener für ungewünschte Aktionen	44
53	EventListener für Eingabe	44
54	<code>send()</code> -Funktion	45
55	<code>stop()</code> -Funktion	46
56	Geschwindigkeitssteuerung auf der Website	46
57	Geschwindigkeitssteuerung HTML-Code	47
58	CSS-Klassen für die Geschwindigkeitssteuerung	47
59	<code>speed-change()</code> -Funktion	48
60	Kamerasteuerung auf der Website	48
61	Kamerasteuerung HTML-Code	49
62	CSS-Klassen für die Kamerasteuerung	49
63	<code>camera-change()</code> -Funktion	50
64	Fernlicht-Icon auf der Website	50
65	HTML-Code für Fernlicht	51
66	CSS-Klassen für das Fernlicht	51
67	<code>togglelight()</code> -Funktion	52

68	Videoübertragung auf der Website	53
69	Videoübertragung HTML-Code	53
70	CSS-Formatierung für dynamic image	53
71	Videoübertragung im JavaScript-Code	54
72	Menü-Icon auf der Website	55
73	Sidebar mit Sensordaten	55
74	Sensordaten HTML-Code	56
75	CSS-Klassen für das Menü	57
76	CSS-Klasse .daten	58
77	JavaScript-Verarbeitung der Sensordaten	58
78	Statusbox auf der Website	59
79	Statusbox: Verbunden	59
80	Statusbox: Verbindung getrennt	59
81	Statusbox HTML-Code	59
82	CSS-Klassen für die Statusbox	60
83	JavaScript Funktionen für den Verbindungsstatus	60
84	Erstes Modell	63
85	Zweites Modell	64
86	Rahmen	64
87	Halterung 1	65
88	Halterung 2	65
89	Motorenhalterung	65
90	Räder	66
91	Wägezellen	66
92	Wägezellenhalterung	67
93	modell-fertig	67
94	HC-SR04 Timing Diagram	72
95	Beispiel für Messung	72
96	HC-SR04 Pinbelegung	73
97	HC-SR04 Schaltungs Beispiel	74
98	HC-SR04.h Bibliothek	75
99	Erstellung von Objekten der Klasse UltraSonicDistanceSensor	75
100	Messung der Entfernung in der loop()-Funktion	75
101	if-Abfrage für Mindestabstand und Geschwindigkeit	76
102	Halbbrücken-Wägezelle	76
103	HX711 24-bit A/D-Wandler	77
104	Prinzip einer Wägezelle	77

105	Wheatstone-Brückenschaltung	78
106	Beschaltung der Wägezellen	79
107	Verkabelung der Wägezellen mit HX711	80
108	Testaufbau der Wägezellen	81
109	Einbinden der HX711_ADC-Bibliothek	81
110	Setup(); - Code für den HX711	82
111	Timeout Abfrage in der Setup();-Funktion	83
112	Gewichtsmessung in der loop();-Funktion	83

11 Literaturverzeichnis

Literaturverzeichnis

1	https://github.com/lacamera/ESPAsyncWebServer	30
2	https://code.visualstudio.com/	84
3	https://github.com/lacamera/ESPAsyncWebServer	85
4	https://arduinojson.org/	85
5	https://docs.arduino.cc/libraries/hcsr04-ultrasonic-sensor/	85
6	https://github.com/Links2004/arduinoWebSockets	86
7	https://docs.arduino.cc/libraries/servo/	86
8	https://github.com/madhephaestus/ESP32Servos	86
9	https://github.com/adafruit/Adafruit-MCP23017-Arduino-Library	86
10	https://github.com/olkal/HX711_ADC	86