



OSTBAYERISCHE  
TECHNISCHE HOCHSCHULE  
REGENSBURG

DATENVERARBEITUNG IN DER TECHNIK

SOMMERSEMESTER 2025

# Projektbericht

## Teammitglieder:

Fabian Becker

---

Jendrik Jürgens

---

Nicolas Koch

---

Michael Specht

---

Jonathan Wohlrab

---

## Betreuung:

Dr. Alexander Metzner, Matthias Altmann

## Abgabedatum:

15.07.2025

3. Juli 2025

# Inhaltsverzeichnis

<b>1</b>	<b>Stromversorgung</b>	<b>3</b>
1.1	Analyse des Aufbaus und der Komponenten des vorherigen Projekts (Koch) .	3
1.2	Aufbau der eigenen Stromversorgung (Koch) . . . . .	3
<b>2</b>	<b>CAD-Konstruktion</b>	<b>5</b>
2.1	Setup und Einarbeitung (Becker, Specht) . . . . .	5
2.2	Geschützarm Version 1 (Specht) . . . . .	5
2.3	Geschützarm Version 2 (Specht) . . . . .	6
2.4	Montagehalterung Motortreiber (Specht) . . . . .	7
2.5	Geschützplattform (Becker) . . . . .	8
2.5.1	Unterbau . . . . .	8
2.5.2	Abdeckung . . . . .	8
2.6	Magazin und Verbindungsstück (Becker) . . . . .	9
2.7	Magazingewicht (Becker) . . . . .	9
2.8	Geschütztrigger (Becker) . . . . .	10
2.9	Halterungen Stormversorgung (Becker) . . . . .	10
2.10	Halterungen Flywheel Motortreiber (Becker) . . . . .	10
2.11	Mikrocontroller-Case (Becker) . . . . .	11
2.12	Halterung Lautsprecher (Becker) . . . . .	12
<b>3</b>	<b>ESP32 Programmierung</b>	<b>13</b>
3.1	Einführung (Becker, Specht) . . . . .	13
3.2	Dualshock4 Treiber (Becker) . . . . .	13
3.2.1	Übertragung . . . . .	14
3.2.2	Version 1: eigens entwickelter Treiber . . . . .	14
3.2.3	Version 2: Treiber basierend auf der Bluepad32 Bibliothek . . . . .	15
3.2.4	Testing . . . . .	15
3.3	Plattformsteuerung (Becker) . . . . .	16
3.3.1	PWM Board Treiber (Becker) . . . . .	16
3.3.2	Ansteuerung der Servo-Motoren (Becker) . . . . .	17
3.3.3	Ansteuerung der Flywheel Motoren (Becker, Koch, Wohlrab) . . . . .	19
3.4	Motor-Treiber (Specht) . . . . .	19
3.4.1	Low-Level Treiber . . . . .	20
3.4.2	Testing . . . . .	21
3.4.3	Differentialantrieb . . . . .	22

3.4.4	Testing . . . . .	23
3.5	MQTT-Anbindung (Specht) . . . . .	24
3.5.1	WiFi-Stack . . . . .	25
3.5.2	MQTT-Stack . . . . .	25
3.5.3	Testing . . . . .	26
3.6	Integration/Fahrzeugsteuerung (Becker, Specht) . . . . .	27
3.6.1	Dualshock4 Treiber (Becker) . . . . .	27
3.6.2	Plattformsteuerung (Becker) . . . . .	27
3.6.3	Differentialantrieb (Specht) . . . . .	27
3.6.4	MQTT-Anbindung (Specht) . . . . .	28
<b>4</b>	<b>Raspberry Pi Programmierung</b>	<b>29</b>
4.1	Lautsprecher (Becker) . . . . .	29
4.2	Gyrosensor Programmierung (Koch) . . . . .	29
4.2.1	Kalman Filter Implementierung (Koch) . . . . .	30
<b>5</b>	<b>Webserver (Koch)</b>	<b>32</b>
5.1	Funktion und Anforderung des Webservers . . . . .	32
5.2	Lösungsansatz und Umsetzung . . . . .	32
	<b>Abbildungsverzeichnis</b>	<b>36</b>
	<b>Tabellenverzeichnis</b>	<b>37</b>
	<b>Literaturverzeichnis</b>	<b>38</b>

# **1. Stromversorgung**

## **1.1. Analyse des Aufbaus und der Komponenten des vorherigen Projekts (Koch)**

Zu Beginn wurde die bestehende Stromversorgung und die dafür genutzten Komponenten eines früheren Semesterprojekts analysiert, um anhand dessen bestimmen zu können, welche Teile wiederverwendet werden können, sowie ob das gegebene Layout in etwa für das eigene Projekt genutzt werden kann.

Essentiell bestand die Stromversorgung aus zwei Step-Down-Wandlern, die aus einer Eingangsspannung eine 8V und eine 5V Ausgangsspannung erzeugten, was ebenfalls für unser eigenes Projekt benötigt wird. Außerdem wurden zwei Verteiler genutzt, um die Spannungen auf die verschiedenen Sensoren und Aktoren zu verteilen. Das vorhandene Layout auf dem Lochrastergerüst war für uns jedoch nicht geeignet, da wir einen übersichtlicheren Aufbau und ein sinnvolles Color-Coding der Kabel für die verschiedenen Anschlüsse und für einen besseren Überblick anstrebten.

## **1.2. Aufbau der eigenen Stromversorgung (Koch)**

Nachdem die vorhandenen Teile analysiert wurden, wurde die Entscheidung getroffen nur die Step-Down-Wandler, da der Rest nicht relevant für unser Projekt war. Lediglich die Verteiler brauchten wir auch, mussten allerdings ersetzt werden, da die Schraubverbindungen kaputt waren. Die Step-Down-Wandler waren so aufgebaut, dass ein Modul die Eingangsspannung erhielt und am Ausgang ein selbstangefertigtes Y-Kabel hatte, welches dann jeweils in einen Verteiler und in den anderen Step-Down-Wandler ging. Diese Kombination sollte auch so für unser Projekt übernommen werden, allerdings mussten dafür die Kabel erneuert werden, da die alten Kabel nicht dem geplanten Color-Coding entsprachen und zu kurz waren. Dabei stellte sich heraus, dass der entstandene Durchmesser, durch die Kombination aus zwei Kabeln zu einem Y-Kabel, zu groß war, um in die Steckverbindung zu passen. Aus diesem Grund entstand das alternative Konzept die ausgehenden Kabel des ersten Step-Down-Wandlers mit dem ersten Verteiler zu verbinden. Das war vor allem dadurch leicht realisierbar, da jeder Verteiler 12 Ports besitzt und 8V lediglich für die Motoren zum fahren benötigt werden. Somit konnte eine Verbindung vom 8V-Verteiler zum zweiten Step-Down-Wandler hergestellt werden ohne dabei die Steckverbindungen zu beschädigen.

Das Color-Coding der Kabel wurde wie folgt eingeführt:

- **Rot:** Versorgungsspannung
- **Schwarz:** Masse
- **Gelb:** PWM-Verbindung für Motoren
- **Weiß:** Direction Pin für Motoren

Des Weiteren wurde darauf geachtet, dass die Kabel so kurz wie möglich gehalten werden und wenn möglich unter der Platte verlegt werden, um eine bessere Übersicht zu gewährleisten.

Als Eingangsspannung wurde zu Beginn ein 6V-Batterieverbund genutzt, der im Laufe des Projekts durch einen 12V-Batterieverbund ausgetauscht wurde, da beim Testing der Motortreiber festgestellt wurde, dass die Motoren eine höhere Spannung benötigen, um korrekt zu funktionieren. Außerdem wurde versucht den Raspberry Pi 5 über den 5V-Verteiler zu versorgen, was jedoch nicht funktionierte, da die Stromstärke zu niedrig war, wenn der Pi aufwendigere Aufgaben erledigen musste. Aus diesem Grund wurde eine Powerbank genutzt, die den Pi mit Strom versorgt und somit die 5V-Verteilung entlastet.

Der Gesamtaufbau der Stromversorgung sieht dabei wie folgt aus:

- **12V-Batterieverbund:**
  - Step-Down-Wandler (8V) → 8V-Verteiler
    - \* 2 PWM Boards für Motoren
    - \* Step-Down-Wandler (5V) → 5V-Verteiler
      - ESP32
      - 2 PWM Boards für die Flywheel Motoren
      - Servo-Motor für die Geschützplattform
      - Servo-Motor für den Geschützarm
- **Powerbank:**
  - Raspberry Pi 5
    - \* Pi-Camera
    - \* MPU6050 Gyrosensor
    - \* SRF02 Ultraschallsensor

## 2. CAD-Konstruktion

### 2.1. Setup und Einarbeitung (Becker, Specht)

Zu Beginn des Projekts wurde in Abstimmung mit Fabian Becker sowie im Austausch mit Andreas Wittmann (Studienkollege) entschieden, FreeCAD als CAD-Software zu verwenden. Der Grund hierfür war die einfache Kollaboration innerhalb der Projektgruppe sowie der unkomplizierte Erfahrungsaustausch mit der Arbeitsgruppe um A. Wittmann. Andere Softwarelösungen wie OnShape wurden diskutiert, aufgrund der Komplexität und der damit verbundenen Einarbeitungszeit im Hinblick auf die Projektlaufzeit jedoch verworfen. FreeCAD ist zudem eine Open-Source-Software, die neben Fedora auch auf Debian-Systemen lauffähig ist. So konnte die Software problemlos auf den Arbeitsrechnern der Teammitglieder installiert werden.

Grundsätzlich stützt sich die Konstruktion auf vorhandene STL-Vorlagen. Ein Beispielprojekt aus dem Internet diente als Grundlage für die Arbeit.

### 2.2. Geschützarm Version 1 (Specht)

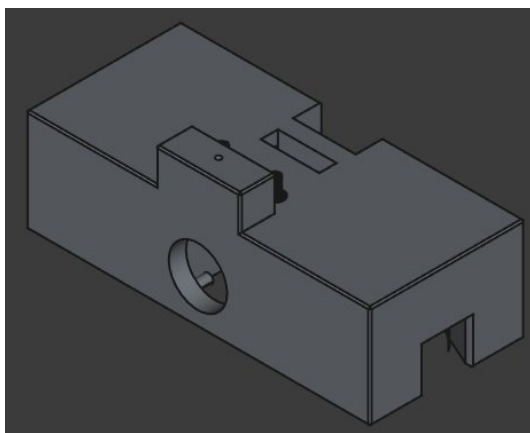
Bevor mit der Konstruktion begonnen wurde, wurde im Team besprochen, welche Komponenten nötig sind, um die Position des Flugobjekts eindeutig zu bestimmen. Die Wahl fiel auf folgende Komponenten, die aus vorherigen Studienprojekten übernommen werden konnten:

- GY-521 MPU-6050 3-Achsen-Gyroskop und Beschleunigungssensor
- SRF02 Ultraschall Entfernungssensor
- Raspberry Pi 5 Kamera Modul
- 2x 28BYJ-48 Schrittmotor

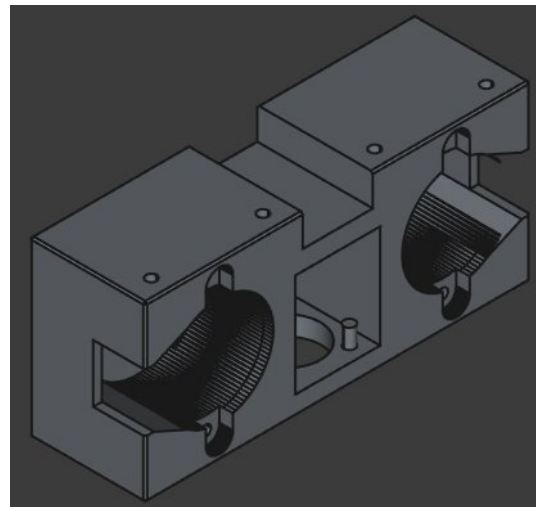
Ziel des ersten Entwurfs war es, diese kompakt auf dem Arm zu integrieren. Die angedachten Schrittmotoren wichen jedoch von der Vorlage aus dem Beispielprojekt ab, weshalb der Geschützarm von Grund auf neu konstruiert werden musste.

Alle Module sollten zentral über der Abschusseinrichtung platziert werden, um eine korrekte Berechnung der Flugbahn zu ermöglichen. Der Ultraschall-Sensor sollte dabei hochkant nach vorne gerichtet sein, um die Entfernung zum Ziel zu messen. Die Kamera sollte schräg nach oben gerichtet sein, um den Himmel zu überwachen. Der Beschleunigungssensor sollte liegend

auf dem Arm montiert werden, um die Beschleunigung des Arms zu messen. Die Schrittmotoren mussten in einem geeigneten Abstand zueinander montiert werden, sodass die Flywheel-Konstruktion des Arms funktioniert. Letzteres konnte durch das Vermessen der Vorlage aus dem Beispielprojekt realisiert werden. Für die restlichen Anforderungen waren die korrekten Maße nötig. Für die Montage der Kamera konnte eine bereits 3D-gedruckte Halterung aus einer anderen Gruppe benutzt werden. Die Haltevorrichtung für den Beschleunigungssensor wurde aus einer STL-Vorlage übernommen und angepasst. Auch für die Schrittmotoren konnte auf ein Modell aus dem Internet zurückgegriffen werden, weshalb es nicht nötig war, die komplexe Geometrie eigenständig zu vermessen. Einzig die Maße für den Ultraschall-Sensor wurden recherchiert und durch Nachmessen validiert.



(a) Geschützarm Version 1 - Frontansicht



(b) Geschützarm Version 1 - Rückansicht

Abbildung 2.1: Geschützarm Version 1

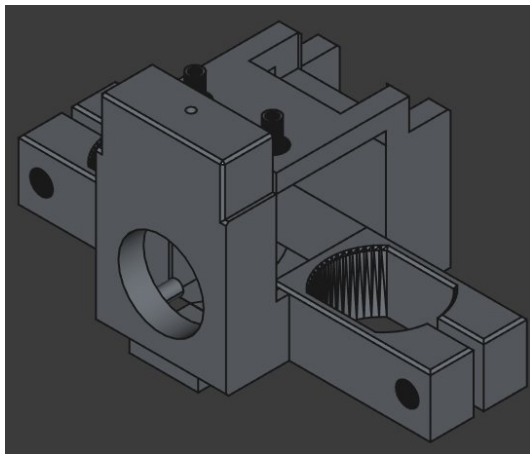
Neben den eigentlichen Maßen der Komponenten war das Kabelmanagement ein wichtiges Thema. Alle Kabel sollten nach hinten am Magazin entlang geführt werden, um eine saubere Optik zu gewährleisten. Für Beschleunigungssensor und Kamera stellte dies kein Problem dar, da diese ganz oben angebracht sein sollten. Der Ultraschall-Sensor und die Schrittmotoren hingegen waren in das neukonstruierte Gehäuse integriert, sodass Aussparungen, wie in Abbildung 2.1 zu sehen, angebracht werden mussten, um die Kabel aus dem Gehäuse herauszuführen.

Außerdem musste sichergestellt werden, dass der Geschützarm an das Magazin montiert werden kann. Hierzu wurden vom Kollegen Fabian Becker Montagepunkte am Magazin konstruiert, die mit dem Geschützarm verschraubt werden können.

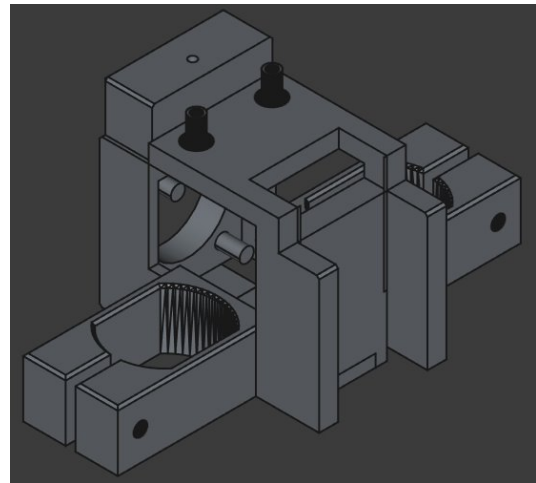
### 2.3. Geschützarm Version 2 (Specht)

Im Laufe des Projektes wurde in Abstimmung mit Fabian Becker klar, dass die Schrittmotoren aufgrund zu geringer Leistung nicht für die Flywheel-Konstruktion geeignet sind. Daraufhin wurde sich für die originalen Motoren aus dem Beispielprojekt entschieden. Das hatte zur Folge, dass der Geschützarm neu konstruiert werden musste, da die Maße der neuen Motoren

von den Alten abweichen. Aus diesem Grund wurde der Geschützarm der Vorlage als Basis genommen und die Grundidee der Version 1 beibehalten.



(a) Geschützarm Version 2 - Frontansicht



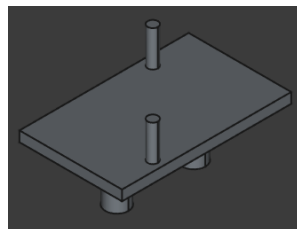
(b) Geschützarm Version 2 - Rückansicht

Abbildung 2.2: Geschützarm Version 2

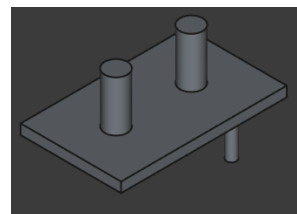
Im Gegensatz zur Version 1 wird der Ultraschallsensor nun seitlich eingeführt anstatt von unten, siehe 2.2. Problematisch waren dabei die beschränkten Platzverhältnisse, da die Schrittmotoren näher am Kanonenrohr angebracht wurden als im vorherigen Entwurf. Außerdem wurden die zuvor angedachten Montagepunkte am Magazin wieder entfernt. Die Zusammenführung des Geschützarms mit dem Magazin erfolgte deshalb mittels Modellbaukleber.

### 2.4. Montagehalterung Motortreiber (Specht)

Für die ersten Funktionstests wurden die Pololu-Motoren provisorisch auf Kork-Schnipseln montiert. Dieses Vorgehen ermöglichte eine zügige Inbetriebnahme, jedoch erwies es sich hinsichtlich Stabilität und Sicherheit als unzureichend. Im Rahmen der Tests kam es zum Abrutschen eines Motors von der Korkunterlage, was zu einer kurzfristigen Wärmeentwicklung und Geruchsbildung führte. Glücklicherweise wurde eine Beschädigung der Hardware vermieden.



(a) Polulu - Draufsicht



(b) Polulu - Bodensicht

Abbildung 2.3: Montagehalterung für Pololu-Motortreiber

Für die finale Abnahme wurde daher ein dauerhaftes und sicheres Montagesystem umgesetzt, das ein sauberes und zuverlässiges Setup gewährleistet. Wie in Abbildung 2.3 zu sehen, kann die Halterung direkt auf der Montageplatte des Fahrzeugs geklippt werden.



### 2.5. Geschützplattform (Becker)

Als Geschützplattform wird der Unterbau des Geschützes bezeichnet, welcher den Geschützarm mit der Lochplatte des Fahrzeugs verbindet. Diese Plattform besteht aus zwei, 3D-gedruckten Komponenten.

#### 2.5.1. Unterbau

Der erste Teil des Objekts ist der Unterbau, welcher eine zylindrische Form aufweist und mit einer Bodenplatte versehen ist. Diese ist mit Schraublöchern ausgestattet, welche dazu dienen, den Aufbau mit dem Fahrzeug zu verbinden. Das vorliegende Bauteil wurde aus dem vorherigen Projekt übernommen, da die Konstruktion bereits auf dem Fahrzeug verbaut war und eine Eigenkonstruktion sehr ähnlich aufgebaut wäre.

Der Unterbau ist so konstruiert, dass er Platz für folgende Komponenten bietet:

- Eine **PCA9685 PWM-Treiberplatine** zur Ansteuerung der Servomotoren auf dem Geschütz. Die erforderlichen Befestigungsbohrungen für die Platine waren bereits im Design integriert.
- Einen **MG996R Servomotor**, der für die Rotation des darüberliegenden Aufbaus verantwortlich ist.

#### 2.5.2. Abdeckung

Die zweite Komponente ist die Abdeckung des Unterbaus. Diese verfügt über Bohrungen zur Verbindung mit dem im Unterbau positionierten Servomotor sowie über Montagepunkte für den Geschützarm.

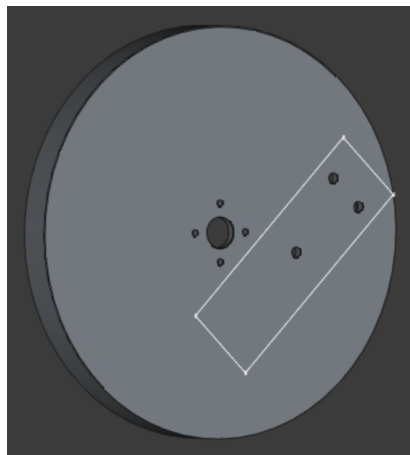


Abbildung 2.4: Abdeckung Plattform mit Position des Verbindungsstücks

Obwohl auch diese Abdeckung bereits vorhanden war, wurde sie exakt vermessen und in FreeCAD rekonstruiert. Ziel dieser Rekonstruktion war es, wie in Abbildung 2.4 ersichtlich, die Bohrlöcher für die Verbindung mit dem Geschützarm so zu positionieren, dass das Verbindungsstück und das Magazin ohne weitere Anpassungen direkt montiert werden konnten.

## 2.6. Magazin und Verbindungsstück (Becker)

Das Magazin, sowie das Verbindungsstück wurden auf Basis einer bestehenden Konstruktionsvorlage [1] 3D-gedruckt. Das Verbindungsstück ist so ausgelegt, dass es einen weiteren MG996R Servomotor aufnimmt, welcher die vertikale Neigung des Geschützes steuert.

Die Struktur des Magazins umfasste einen linken und einen rechten Teil, die in der Vorlage zusammengeklebt wurden. Wie bereits im Abschnitt 2.2 dargelegt, erfolgte für die Verbindung mit dem Geschützarm der ersten Version die Konstruktion von Verbindungsstücken an beiden Enden des Magazins, um mittels Schrauben eine Verbindung zwischen beiden Teilen zu gewährleisten. Darüber hinaus wurden Schraublöcher in beide Teile des Magazins integriert, um eine Verbindung beider Teile mittels Schrauben zu ermöglichen.

In der zweiten Version des Geschützarmes wurden die Verbindungsstücke entfernt, da der Geschützarm nun direkt mit dem Magazin verbunden wird. Die vorgenommene Änderung resultierte aus der Tatsache, dass es aufgrund der signifikant geringeren Dimension des Geschützarmes unmöglich war, Verbindungsstücke mit Schraublöchern zu versehen.

Zuletzt wurde auch die Länge des Laufs vergrößert, um eine bessere Stützung des Geschützarms zu gewährleisten.

## 2.7. Magazingewicht (Becker)

Das Magazin des Geschützes verfügt über eine Kapazität für sechs Nerf-Darts sowie ein Magazingewicht. Letzteres dient dazu, einerseits bei hoher Vibration des Fahrzeugs das Ausfallen der Darts zu verhindern und andererseits sicherzustellen, dass nach einem Schuss das nächste Geschoss nachrutscht.

Zunächst wurde die Vorlage [1] für das Magazingewicht angepasst, indem der Projektname eingraviert wurde.

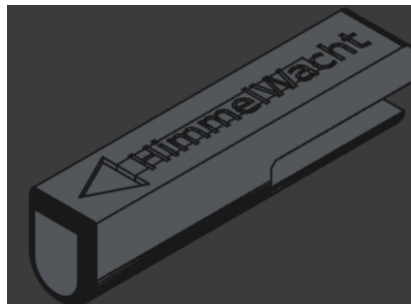


Abbildung 2.5: Magazingewicht

Das Gewicht besitzt außerdem eine Aussparung, die dafür sorgt, dass das Bauteil die Bewegung des Servo Motors nicht einschränkt, welcher die Darts bei der Schussabgabe in die Flywheel Motoren befördert. Ohne entsprechende Aussparung würde der Servo versuchen, das Gewicht in den Lauf zu schieben, was zu Materialschäden, entweder am Motor oder am Geschütz führen würde.

Die vorliegende Aussparung führte allerdings zu Problemen. Wird die Plattform sehr weit nach hinten geneigt, so kam es vor, dass der hintere Teil des Gewichts nicht schwer genug war, um

die Nerfs-Darts in den Lauf zu drücken. Die Lösung für dieses Problem bestand im Einsatz selektiver Infill-Technik. Die übrigen Teile wurden standardmäßig mit einem Infill-Gehalt von 15% gedruckt. Im hinteren Teil des Gewichts wurde jedoch eine 100% Infill eingesetzt. Dieses Vorgehen resultierte in der Behebung des Problems.

### 2.8. Geschütztrigger (Becker)

Der Auslösemechanismus initiiert den Schussvorgang. Durch die Aktivierung eines MG92B Servomotors wird ein Dart in die laufenden Flywheel-Motoren geschoben, welche ihn beschleunigen und abfeuern.

Der Mechanismus ist eine zweiteilige Konstruktion, die auf einer bestehenden Vorlage [1] basiert, deren Schraublöcher jedoch für den spezifischen Anwendungsfall angepasst wurden. Am unteren Teil wurde eine Öffnung für eine M4-Schraube konstruiert, welche in das Magazin hineinreicht und Kraft auf den Dart ausübt. Bei der Verbindung der beiden Teile wurde darauf geachtet, dass das Schraubloch des ersten Teils etwas größer ist, so dass die Schraube hier nicht greift. Diese wird lediglich im zweiten Teil festgeschraubt, wodurch eine Art Gelenk entsteht. Abschließend erfolgt noch die Verbindung des ersten Teils mit dem Servomotor.

### 2.9. Halterungen Stormversorgung (Becker)

Da einige Komponenten der Stromversorgung ebenfalls in ein früheres Projekt integriert wurden, wurde auch untersucht, wie die Vorgängergruppe diese Teile montiert hat. Zu diesem Zweck hat die Gruppe Komponenten mit Stelzen gefertigt, die in die Lochplattform des Fahrzeugs eingesetzt werden konnten.

Dieses System wurde unter anderem für die Stromverteiler verwendet. Für dieses Projekt wurde ein weiteres Teil nach gleichem Prinzip für die Step-Down-Module erstellt, um auch diese in gleicher Weise befestigen zu können.

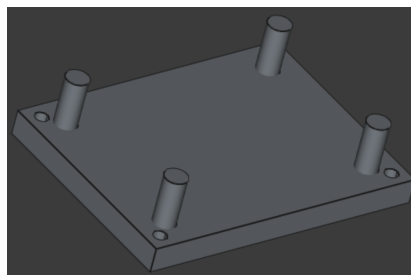


Abbildung 2.6: Halterung DC Step-Down mit Stecksystem

### 2.10. Halterungen Flywheel Motortreiber (Becker)

Des Weiteren wurde die Halterung für die Motortreiber der Flywheel-Motoren mit den gleichen Stempeln ausgestattet. Die Halterung stellt eine modifizierte Version einer Vorlage [2] dar. Die ursprüngliche Konstruktion dieser Vorlage beinhaltete seitliche Schraublöcher, welche jedoch im Zuge der Implementierung des Stecksystems entfernt wurden.

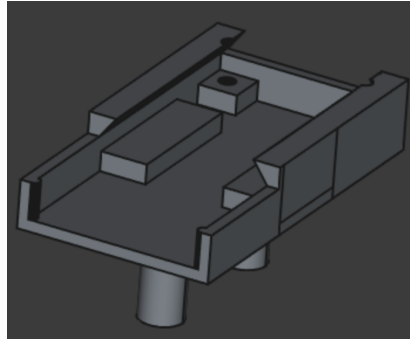


Abbildung 2.7: Halterung für Flywheel Motortreiber

### 2.11. Mikrocontroller-Case (Becker)

Die Mikrocontroller-Case bietet Platz für einen ESP-32 auf einem Freenove Breakout Board und für einen Raspberry Pi 5 mit aktivem Luftkühler. Beide Controller sind in diesem Fall übereinander angeordnet, da auf der Lochplattform ansonsten nicht genügend Platz zur Verfügung stehen würde, um alle weiteren erforderlichen Komponenten, wie beispielsweise jene für die Stromversorgung, unterzubringen. Aufgrund der Verwendung fester Kabel, wie beispielsweise des Kamerakabels für den Raspberry PI, und der allgemeinen Kabellänge müssen beide Controller in unmittelbarer Nähe zur Geschützplattform platziert werden.

Der ESP-32 ist im unteren Teil des Gehäuses untergebracht. Im ersten Entwurf wurde lediglich eine Vorlage [3] gedruckt. Es stellte sich jedoch heraus, dass es von dem Freenve-Steckbrett mehrere Varianten in unterschiedlichen Größen gibt. Die gewählte Vorlage erwies sich als zu klein, um unser konkretes Modell darin zu platzieren. Daher wurde auf Basis der Vorlage eine Version mit passenden Dimensionen erstellt. Darüber hinaus wurden einige Änderungen vorgenommen. So wurde ein zusätzliches Schraubloch hinzugefügt, um eine externe Antenne anzuschließen und somit die Bluetooth- und WLAN-Abdeckung zu optimieren. Im nächsten Schritt wurden im Deckel Kühllöcher in Form des Textes HimmelWacht integriert, um den Mikrocontroller mit zusätzlicher Luft zu versorgen.

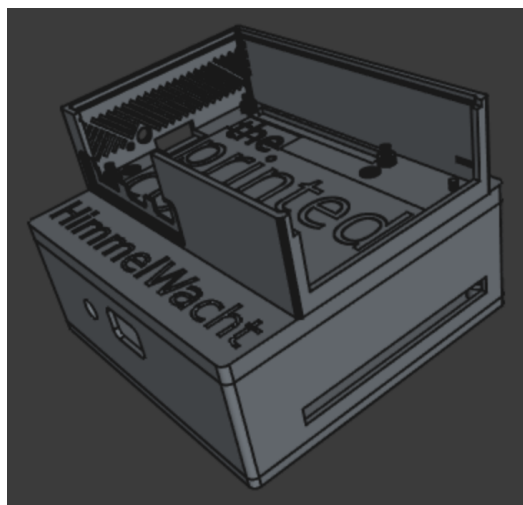


Abbildung 2.8: Mikrocontroller Case

Zuletzt wurde die Unterseite der Case noch mit dem modularen Stecksystem ausgestattet, welches auch für Teile der Stromversorgung verwendet wird. Die Montage kann somit ohne den Einsatz von Klebstoff oder Schrauben durchgeführt werden.

### 2.12. Halterung Lautsprecher (Becker)

Auch für den ursprünglich vorgesehenen Lautsprecher wurde eine Befestigung konstruiert. Die vorliegende Halterung wurde konzipiert, um sowohl den Lautsprecher als auch das zugehörige Verstärkerboard unterzubringen, womit kurze Kabellängen und eine damit einhergehend einfachere Verkabelung gewährleistet werden. Diese Halterung ist ebenfalls mit dem modularen Stecksystem ausgestattet.

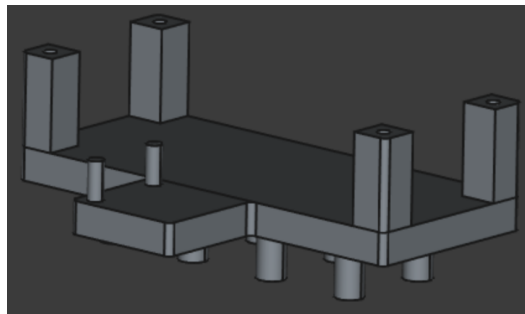


Abbildung 2.9: Lautsprecherhalterung

## 3. ESP32 Programmierung

### 3.1. Einführung (Becker, Specht)

Die Programmierung der zeitkritischen Funktionalitäten erfolgt unter Verwendung eines ESP-32, da bestimmte Funktionen strengere zeitliche Anforderungen aufweisen, als dies mit einem General-Purpose-Betriebssystem realisierbar wäre.

Nach Abwägung aller relevanten Faktoren wurde sich dazu entschieden, den klassischen ESP-32 Chip zu wählen, da dieser den Bluetooth Classic Standard unterstützt. Dieses Protokoll wird für die Ansteuerung des Dualshock4-Controllers benötigt. Darüber hinaus trägt das weite Verbreitungsgebiet dieser Serie dazu bei, dass eine Vielzahl an Material zur Verfügung steht, was wiederum die Fehlersuche erheblich vereinfacht.

Es wurde jedoch ausdrücklich darauf verzichtet, den Controller unter Verwendung des Arduino Frameworks zu programmieren. Die Ursache hierfür war insbesondere die Größe der entstehenden Kompilate. Es sei darauf hingewiesen, dass das System über zahlreiche Funktionalitäten verfügt, was eine begrenzte Speicherverfügbarkeit zur Folge hat. Daher erfolgte die Programmierung des Controllers unter Verwendung des *Espressif IoT Development Frameworks* (ESP-IDF) [4]. Dieses Framework stellt ebenfalls eine Vielzahl von Treibern für unterschiedliche Funktionen bereit, ist jedoch gezielt für Espressif-Controller konzipiert und zeichnet sich dadurch aus, dass es einen geringen Speicherplatzbedarf aufweist.

Die Programmierung erfolgte in der Programmiersprache C und nutzt das Betriebssystem FreeRTOS, welches von Espressif auf Multiprozessoren portiert wurde.

### 3.2. Dualshock4 Treiber (Becker)

Gemäß der Anforderung ist die Steuerung des Fahrzeugs sowie der Geschützplattform durch einen DualShock4-Controller vorgesehen. Einerseits dient dies dem Testen der Plattformsteuerung für den späteren Betrieb der KI. Des Weiteren fungiert sie als Gamification-Element. Das Ziel ist die Steigerung des Nutzens und der Freude an dem Projekt, indem eine bekannte Steuerungsmethode verwendet wird, welche die Benutzerfreundlichkeit erhöht.

Für die Realisierung dieser Funktionalität wurde ein spezieller Treiber entwickelt, der den Zustand des Controllers in regelmäßigen Abständen an den ESP-32 übermittelt. Darüber hinaus umfasste mein Wunsch eine Funktion, die Vibrationen und Farbwechsel am Gamepad auslöst, und zwar vom Mikrocontroller aus.

### 3.2.1. Übertragung

Vor jeglicher Datenübertragung muss zunächst eine Kopplung per Bluetooth hergestellt werden. Der Sony Dualshock 4 Controller verwendet in diesem Fall den Bluetooth Classic 4.0 Standard. Für die Verbindung des Controllers mit den Geräten ist im Flash-Speicher des Gamepads eine MAC-Adresse gespeichert. Beim Anschalten des Controllers wird versucht, Kontakt zu dieser Adresse aufzubauen. Unter normalen Betriebsbedingungen wäre dies die Adresse der PlayStation4-Konsole. Für unser Projekt wurde jedoch die gespeicherte MAC-Adresse mit der des ESP-32 überschrieben. Nach dem Einschalten unternimmt das Gamepad sofort den Versuch, sich mit dem Mikrocontroller zu verbinden.

Die Übertragung der Daten zwischen Gamepad und Mikrocontroller erfolgt mittels sogenannter HID-Reports. HID-Reports sind binäre Datenstrukturen, welche die Reihenfolge der Informationen festlegen. Die Struktur der Reports wurde von Sony nicht öffentlich zur Verfügung gestellt. Allerdings war es einigen Personen möglich, durch Reverse-Engineering wichtige Felder zu ermitteln [5].

Das Human Interface Device (HID)-Protokoll ist ein standardisiertes Kommunikationsprotokoll zur Übertragung von Eingabe- und Steuerdaten zwischen Peripheriegeräten (z.B. Tastaturen, Mäusen, Gamepads) und einem Host-System. Es wurde ursprünglich für USB spezifiziert [6] und später für Bluetooth Classic übernommen.

Data Format								
byte index	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
[0]	0x0a				0x00		0x01	
[1]	0x01							
	The following structure is a supposition.							
[2]	Left Stick X (0 = left)							
[3]	Left Stick Y (0 = up)							
[4]	Right Stick X							
[5]	Right Stick Y							
[6]	TRI	CIR	X	SQR	D-PAD (hat format, 0x08 is released, 0=N, 1=NE, 2=E, 3=SE, 4=S, 5=SW, 6=W, 7=NW)			
[7]	R3	L3	OPT	SHARE	R2	L2	R1	L1
[8]	Counter (counts up by 1 per report)						T-PAD	PS
[9]	Left Trigger (0 = released, 0xFF = fully pressed)							
[10]	Right Trigger							

Abbildung 3.1: Beispielhafter Aufbau eines HID-Reports aus [5]

### 3.2.2. Version 1: eigens entwickelter Treiber

Eine Websuche nach existierenden DualShock-4-Treibern für das ESP-IDF brachte zunächst nur veraltete Implementierungen zutage, die auf nicht mehr unterstützten APIs basierten. Zahlreiche andere Ansätze zielten ausschließlich auf das Arduino-Framework ab und ließen sich daher nur eingeschränkt oder gar nicht in ein ESP-IDF-basiertes Projekt integrieren.

Da die Struktur der vom Gamepad gesendeten HID-Reports dank Reverse Engineering bekannt war und das ESP-IDF über eine Bluetooth-HID-Host-API verfügte [7], wurde im ersten Ansatz ein eigener Treiber entwickelt.

Der Verbindungsaufbau verlief problemlos, und zu Beginn wurden die Input-Reports korrekt empfangen. Nach kurzer Zeit fror die Datenübertragung jedoch ein, obwohl alle anderen Tasks auf dem System weiterhin fehlerfrei ausgeführt wurden. Recherchen ergaben, dass dieses Verhalten auch von anderen Nutzern beobachtet wurde [8]. Als Ursache wird die vergleichsweise hohe Sendefrequenz des Controllers vermutet, die laut Nutzertests bei über 700 Hz liegt [5]. Um den Overhead in der Callback-Funktion für empfangene Reports zu reduzieren, wurde ein Zeitfilter implementiert. Ein Timer prüft beim Eintreffen eines Reports, ob seit dem letzten verarbeiteten Report mindestens  $16.\bar{6}$  ms (entsprechend 60 Hz) vergangen sind. Ist dies der Fall, wird der aktuelle Report in eine durch FreeRTOS bereitgestellte Queue eingereiht. Diese wird von einem separaten Task abgearbeitet, der die Reports auf der Konsole protokolliert. Auf diese Weise konnte das Logging, das potenziell blockierend wirkt, vom zeitkritischen Pfad entkoppelt werden.

Obwohl diese Maßnahme die Zeitspanne, in der die Daten stabil übertragen wurden, verlängerte, wurde das zugrundeliegende Problem damit nicht vollständig gelöst. Nach einigen Minuten unterbrach der Controller die Übertragung der Reports erneut, obwohl die Status-LED weiterhin eine aktive Bluetooth-Verbindung anzeigte.

Da der ESP-32 in dieser Anwendung zusätzlich weitere zeitkritische Komponenten wie den Motortreiber verwalten muss, ist die Stabilität und Effizienz des Treibers von zentraler Bedeutung. Aus diesem Grund wurde entschieden, die Entwicklung eines eigenen Treibers nicht weiter zu verfolgen und stattdessen nach einer robusteren und getesteten Alternative zu suchen.

#### 3.2.3. Version 2: Treiber basierend auf der Bluepad32 Bibliothek

#### 3.2.4. Testing

Um den funktionierenden Treiber zu testen wurden zwei Tasks auf dem ESP-32 erstellt:

- **Input-Test Task:** Dieser Task wartet vor jedem Schleifendurchlauf auf verbinden, das bedeutet er ist so lange blockiert, bis der Dualshock-Controller die Verbindung hergestellt hat. Dann wartet er an der Input Queue bis der Controller einen Zustandsreport schickt. Dieser wird dann aus der Queue entnommen und auf der Konsole ausgegeben. Zuletzt wird  $16.\bar{6}$  ms (60 Hz) gewartet.
- **Output-Test Task:** Hier wird ebenfalls blockiert bis das Gamepad verbunden ist. Nachdem die Verbindung aufgebaut ist wird je ein Vibrations- und Farbwechselevent in die Queue gelegt. Da das Vibrationsevent eine Dauer von 100 ms bekommen hat wird hier 100 ms gewartet.

Mit diesem simplen Testsetup konnten alle Funktionen des Controller getestet werden. Auch das Fortfahren nach einem Verbindungsabbruch konnten durch Aus- und Einschalten des Gamepads getestet werden.

Bei Testen fiel weiterhin auf, dass die Batteriestandswarnung dauerhaft ausgelöst war, ein Blick auf die Input Werte zeigte, dass der Controller eine komplette leere Batterie anzeigte.



Auch nach dem Aufladen fielen die Werte schnell wieder auf 0, obwohl am PC ganz andere Ladestände angezeigt wurden. (WIP!)

## 3.3. Plattformsteuerung (Becker)

Die sogenannte Plattformsteuerung bezeichnet alle technischen Komponenten, die erforderlich sind, um die Plattform in ihrer Rotation, vertikalen Neigung sowie für die Abgabe eines Schusses zu steuern.

Für den genannten Zweck werden folgende Komponenten benötigt:

- **Drehung und Neigung**
  - zwei MG996R Servo Motoren
- **Schussabgabe**
  - zwei DC-Motoren (Flywheels)
  - MG92B Servo Motor

Da die maximale Ausgangsstärke eines GPIO-PINs des ESP-32 mit 40mA [9, S. 53] für die benötigten Motoren nicht ausreicht [10–12], wurden entsprechende Treiberboards verwendet. Konkret handelt es sich hierbei um ein PCA9685 PWM-Treiberboard für die Servomotoren und um per PWM steuerbare MOSFET-Module für die Flywheel-Motoren.

In der nachfolgenden Sektion wird der Entwurf des Codes erörtert, der erforderlich ist, um die genannten Teile anzusteuern.

### 3.3.1. PWM Board Treiber (Becker)

Das PCA9685 PWM-Treiberboard gestattet die gleichzeitige Anbindung von bis zu 16 Servomotoren. Für die Stromversorgung steht ein Eingang mit einer Spannung von 5 Volt zur Verfügung.

Die Steuerung des Boards erfolgt durch das Schreiben verschiedener Werte in Konfigurationsregister, wobei das I<sup>2</sup>C-Protokoll zum Einsatz kommt. Der vorliegende Treiber wurde aus der Portierung eines bereits bestehenden Treibers [13] entwickelt, welcher in der Programmiersprache C++ implementiert war. Es wurde bewusst nur die Funktionalität portiert, die für den Umfang des Projekts von Relevanz war. Der Treiber umfasst demzufolge lediglich drei Funktionen:

- **pca9685\_init**: Die Funktion erhält die gewünschte Konfiguration für das Board (beispielsweise die Bus-Adresse, die SDA- und SCL-Ports für den I<sup>2</sup>C-Bus) und initialisiert den I<sup>2</sup>C-Bus. Im Anschluss registriert sie das Treiberboard und konfiguriert schließlich das Board mit der gewünschten PWM-Frequenz.
- **pca9685\_set\_pwm\_on\_off**: Mithilfe dieser Funktion besteht nun die Möglichkeit, einen Motor auf einem der 16 Kanäle zu steuern. Der Parameter *ON* ist eine 12-Bit Zahl beschreibt hierbei den Zeitpunkt in der Phase, an welchem der Ausgang auf 5 Volt

geschaltet wird. *OFF*, ebenfalls eine 12-Bit Zahl bezeichnet den Zeitpunkt, zu welchem der Ausgang wieder auf 0 Volt geregelt wird. Eine grafische Veranschaulichung ist in Abbildung 3.2 ersichtlich. Da für die Ansteuerung der Servo Motoren keine symmetrischen PWM-Signale benötigt werden, wird der Parameter *ON* im Folgenden immer den Wert 0 annehmen.

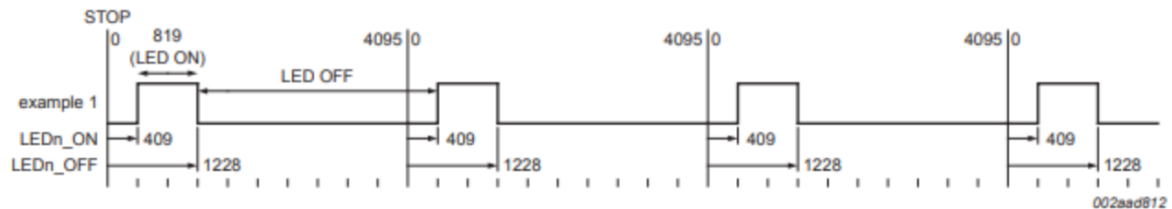


Abbildung 3.2: Erklärung ON/OFF Parameter für PCA9685 aus [14, S. 17]

- **pca9685\_set\_off**: Mittels dieser Funktion kann das PWM-Signal auf einem bestimmten Kanal deaktiviert werden.

Auf Basis dieses Treiber wurden im nächsten Schritt zwei Interfaces programmiert: Einerseits das Interface zur Plattform-Kontrolle und andererseits das Interface zur Schusskontrolle, welches zusätzlich der Logik zur Ansteuerung der Flywheel-Motoren enthält.

### 3.3.2. Ansteuerung der Servo-Motoren (Becker)

Um nun die Plattform sowohl manuell über den DualShock4-Controller als auch semi-automatisch per künstlicher Intelligenz über MQTT präzise steuern zu können, wurde ein Interface entwickelt, das die Ansteuerung eines Motors an eine bestimmte Position erlaubt.

In diesem Kontext wird mit der Einheit Grad gearbeitet. Ein Blick in die Referenz [15] der Servo-Motoren zeigt, dass durch die Einstellung der Duty-Cycle-Länge des PWM-Signals eine Drehung auf eine bestimmte Grad-Position erreicht wird. Im ersten Schritt wurden die *OFF* Werte für die Punkte  $-90^\circ$  (max. Drehung nach links),  $0^\circ$  und  $90^\circ$  (max. Drehung nach rechts) manuell ermittelt. Die absoluten Werte für die Drehungen werden im Folgenden als *value<sub>-90°</sub>*, *value<sub>0°</sub>* und *value<sub>90°</sub>* bezeichnet.

Für die Berechnung des Zielwerts *value<sub>off</sub>* für den *OFF* Wert des PWM-Board-Treibers aus einem gegebenen Winkel wird Formel 3.1 verwendet.

Sei  $\theta$  der gewünschte Winkel,

$|\theta|$  = Betrag von  $\theta$

$$n_3 = \left\lfloor \frac{|\theta|}{3} \right\rfloor$$

$$n_2 = |\theta| - n_3$$

(3.1)

$$s = 2 \cdot n_2 + 3 \cdot n_3$$

$$\tilde{s} = \begin{cases} s, & \theta > 0 \\ -s, & \theta \leq 0 \end{cases}$$

$$value_{off} = value_{zero} + \tilde{s}$$

Die Idee der Formel beruht auf der Tatsache, dass die Differenz aus  $value_{90^\circ}$  bzw.  $value_{-90^\circ}$  und  $value_{0^\circ}$  gebildet und anschließend gleichmäßig auf den Bereich von  $]1, 90[$ , also auf 90 Werte, aufgeteilt wird.

Für jedes zusätzliche Grad Rotation um den Nullpunkt müssten demnach entweder  $2\bar{3}$  addiert oder subtrahiert werden. Da dies in Anbetracht der Verwendung von Festkommazahlen nicht realisierbar wäre, erscheint die naheliegende Idee, den Wert zu runden. Diese Praktik würde jedoch im Zeitverlauf zu einer zunehmenden Abweichung und folglich zu einem Verlust an Präzision führen. Um das genannte Problem zu vermeiden, wird der *OFF* Wert für jedes dritte Grad Drehung um den Faktor 3 verändert und für jedes andere Grad um Faktor 2. Die Formel 3.1 berechnet dabei in  $n_3$  die Anzahl der Faktor-3-Drehungen und in  $n_2$  die Anzahl der Faktor-2-Drehungen. Ausgehend davon wird  $value_{0^\circ}$  verändert.

Das Endergebnis stellt eine Funktion dar, mittels derer eine präzise gradweise Ansteuerung beider Plattformachsen möglich ist.

Darüber hinaus wurde im Interface ein Clipping der Werte als Sicherheitmechanismus integriert. Im Zuge der Initialisierung des Plattforminterfaces muss für jede Achse ein Startwinkel sowie ein linker und rechter Stopwinkel angegeben werden. Sollte es bei der Steuerung durch den Dualshock4-Controller in dessen Regelschleife oder in der KI-Berechnung zu einem fehlerhaften Gradwert kommen, wird dieser Wert automatisch an den nächstgelegenen Stopwinkel geclippt. Durch diese Maßnahme werden potenzielle Materialschäden, die durch fehlerhafte Drehungen verursacht werden könnten, verhindert.

Die Realisierung der Ansteuerung des kleinen Servos, dessen Funktion darin besteht, die Nerf-Darts in die Flywheel-Motoren zu schieben, würde durch die Verwendung dieser Art der Ansteuerung zu einer unnötigen Komplexität führen. Daher wurde für diesen Motor lediglich eine Startposition (Geschütztrigger ganz hinten; Dart kann in das Magazin fallen) und eine Endposition (Geschütztrigger ganz vorne; Dart wurde in die Flywheels geschoben) durch manuelles Einsetzen von *OFF* Werten festgelegt. Wird nun ein Schuss ausgelöst, so fährt der Servo-Motor zunächst in seine Endposition und von dort aus wieder zurück in seine Startposition.

#### 3.3.3. Ansteuerung der Flywheel Motoren (Becker, Koch, Wohlrab)

Für die Vervollständigung des Fire-Control Interfaces wird nun noch die Ansteuerung der Flywheel-Motoren benötigt. In dem vorliegenden Projekt erfolgt die Steuerung der beiden Gleichstrommotoren jeweils durch ein Power-MOSFET-Modul.

Die Funktionsweise des Modul lässt sich folgendermaßen beschreiben:

- Auf einer Seite wird der Eingangstrom, in unserem Fall 5V, vom Stromverteiler eingeführt.
- Andererseits wird der Ausgang jeweils mit einem Motor verbunden.
- Die am Ausgang ausgegebene Spannung kann über einen PWM-Pin geregelt werden [10].

Die Wahl fiel auf dieses relativ simple Modul, da lediglich die Anforderung bestand, dass sich die Motoren bei der Schussabgabe möglichst schnell drehen sollen. Im Rahmen dieses Projekts waren keine Änderungen der Richtung oder präzisere Steuerung erforderlich.

Im Rahmen des manuellen Tests wurde festgestellt, dass zur Erzielung eines optimalen Schussbildes eine Versorgung der Motoren mit 5 Volt erforderlich ist. Infolgedessen beträgt der Duty-Cycle entweder 0 oder 100 Prozent. Daher wurde der ursprüngliche PWM-Code letztlich auf das reine Ein- und Ausschalten eines GPIO-Pins reduziert.

#### 3.4. Motor-Treiber (Specht)

Wesentlich für den Einstieg in die Motor-Treiber-Programmierung waren die verwendeten Hardware-Komponenten. Folgende Teile wurden verwendet:

- 2x Pololu G2 High-Power Motor Driver 24v13 (MD31C)
- 2x MFA/Como Drills 919D501

Im ersten Ansatz wurde der Versuch unternommen, den Low-Level-Treiber (LL-Treiber) direkt mit der Differenzialantriebs-Logik (Differentialantrieb) zu koppeln. Nach intensiver Recherche und ersten Programmieransätzen wurde jedoch deutlich, dass eine klare Trennung beider Ebenen unter modularen Gesichtspunkten vorzuziehen ist. Diese Trennung resultiert in einer signifikanten Steigerung sowohl der Wiederverwendbarkeit als auch der Wartbarkeit des Codes. Abhilfe schafften hierbei vor allem die Verwendung von ESP-IDF-Komponenten.

Um die Ansteuerung der Motoren zu realisieren, war es von zentraler Bedeutung, die Spezifikationen der verwendeten Hardware zu berücksichtigen. Das Datenblatt der Pololu-Motor-Treiber wurde in Form einer Webseite gefunden. Die darin enthaltenen Informationen waren ausreichend, um die Logik zu implementieren. Für die verwendeten Motoren lieferte das Datenblatt insbesondere elektrische Kenngrößen, die für die Absicherung der Hardware von entscheidender Bedeutung waren. Dazu zählten maximale und nominale Ströme. Zu Beginn des Projektes konnte nur auf einen 6V-Akku zurückgegriffen werden, obwohl die Motoren bis zu

12V-Betriebsspannung zulassen. Daraufhin wurden entsprechende Widerstände auf den Treiberboards angebracht, um den maximalen Strom für 6V zu begrenzen. Des Weiteren konnte aus dem Datenblatt des Pololu-Motor-Treibers die Notwendigkeit überdimensionierter Kondensatoren abgeleitet werden, um eine gute und stabile Performance sicherzustellen.

Der Fokus der Implementierung lag vorrangig auf Modularität, Erweiterbarkeit, Clean-Code und Best-Practices. Dafür wurde wenn möglich auf globale Variablen verzichtet und stattdessen auf die Verwendung von Strukturen und Funktionen in Kombination mit Pointern gesetzt.

#### 3.4.1. Low-Level Treiber

Die Aufgabe des LL-Treibers bestand darin, genau einen Motor anzusprechen und zu steuern. Das verwendete Framework ESP-IDF bietet eine Vielzahl an API-Funktionen, die eine abstrahierte und einfache Ansteuerung der Hardware ermöglichen. Für das Erzeugen von PWM-Signalen sind vor allem zwei API's von zentraler Bedeutung:

- LED Control (LEDC)
- Motor Control Pulse Width Modulator (MCPWM)

Wie den Namen zu entnehmen ist, ist LEDC für die einfache Ansteuerung von LEDs gedacht, während MCPWM speziell für Motoren entwickelt wurde. Der MCPWM-Generator besteht aus einer Reihe von Submodulen, wie bspw. einem Fault-Module und einem Brake-Operator. Die Pololu-Boards bieten ebenfalls einen Fault-Pin, weshalb im weiteren Projektverlauf der MCPWM-Generator verwendet wurde, um diese Funktionalität nutzen zu können.

Die MCPWM-API umfasst mehrere Funktionen und Strukturen. Deswegen wurde der erste Entwurf auf Basis einer Kombination aus KI-generiertem Code und Beispielcode von Github erstellt. Die grundlegende Funktionalität konnte dadurch unkompliziert und schnell erfasst werden, wodurch Zeit gespart wurde. Nichtsdestotrotz war es notwendig, entsprechende Literatur zur API zu lesen und zu verstehen. Im Folgenden wurde der Code Stück für Stück angepasst, modularisiert und erweitert.

Ein Key-Konzept entstand aus dem Gedanken, was passieren würde, wenn ein Duty-Cycle von beispielsweise 100 % (Volllast) gesetzt wird und die Drehrichtung des Motors umgekehrt wird. Die Annahme war, dass der Motortreiber + und - umpolen würde und der Motor währenddessen solange als Generator arbeitet, bis die Richtung letztendlich umgekehrt wird. Der dabei möglicherweise auftretende Rückstrom könnte eventuell die Hardware beschädigen. Aus diesem Grund wurde sich auf eine Ramping-Strategie geeinigt, die eine sichere und kontrollierte Änderung der Drehrichtung ermöglicht. Dabei soll sichergestellt werden, dass der Duty-Cycle des PWM-Signals bzw. die Richtung für den Motor nicht abrupt geändert wird. Stattdessen wird der Duty-Cycle in konfigurierbaren Schritten dem Nullbereich angenähert. In einem sicheren Hysteresebereich wird dann die Richtung geändert und der Duty-Cycle der neuen

gewünschten Geschwindigkeit angepasst.

Ein weiterer wichtiger Baustein sollte das automatisierte Erkennen von Fehlern der Motoren sein. In solch einem Fall sollte der Motor sofort gestoppt werden und eine Signalleuchte angehen. Ein einfacher Testaufbau bestehend aus Pull-up Widerständen und einer LED sollte die grundlegende Funktionalität des Fault-Pins sicherstellen.

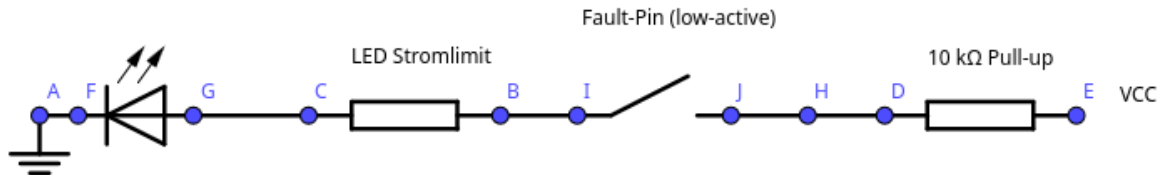


Abbildung 3.3: Testaufbau des Fault-Pins des Pololu-Motor-Treibers

Der Fault-Pin ist laut Datenblatt ein Open-Drain-Ausgang, der bei einem Fehler auf Low gezogen wird. Im Testaufbau wurde ein Pull-Up-Widerstand von 10 kΩ verwendet, um den Pin im Normalfall auf High zu halten. Folgendes Verhalten wurde erwartet: Bei einem Fehler wird der Pin auf Low gezogen und die LED leuchtet auf. Dies war jedoch nicht der Fall. Stattdessen leuchtete die LED dauerhaft, obwohl der Motor einwandfrei funktionierte. Nach Rücksprache mit dem Betreuer stellte sich heraus, dass dieser Pin in der Vergangenheit nie benutzt wurde. Um den Projektfortschritt nicht unnötig zu gefährden, wurde beschlossen, den Fault-Pin nicht weiter zu verwenden und mit der Implementierung des Differentialantrieb fortzufahren.

#### 3.4.2. Testing

Das Testing unterteilte sich in mehrere Schritte. Für die Erstellung der Testfälle wurde dabei zum Großteil KI-generierter Code verwendet. Dadurch konnte der damit verbundene Zeitaufwand signifikant minimiert und zugleich eine ausreichende Testabdeckung garantiert werden.

Zunächst sollte die grundlegende Funktionalität des LL-Treibers getestet werden. Dazu wurden LEDs auf einem Breadboard angebracht, welche die Motoren simulieren sollten. Die Helligkeit der LEDs sollte dabei die Geschwindigkeit des Motors repräsentieren.

Nach erfolgreicher Validierung der Ergebnisse wurde der Testaufbau an die echte Hardware angepasst. Zudem sollte ein Wrapper erstellt werden, der beide Motoren über einzelne Treiber-Objekte anspricht. Dies stellte somit eine Vorstufe zum Differentialantrieb dar. Ziel war es, die beiden Motoren unabhängig voneinander ansteuern zu können. Dafür wurde das Fahrzeug auf einer Holzkonstruktion bestehend aus zwei Holzlatten platziert. Obwohl die Tests positiv verliefen, sollte sich der Test ohne Bodenkontakt im weiteren Projektverlauf als suboptimal herausstellen.

#### 3.4.3. Differentialantrieb

Die Grundidee des Differentialantriebs ist aus der Robotik abgeleitet und beschreibt eine Antriebsart, bei der die Bewegung eines Fahrzeugs durch die unterschiedliche Drehgeschwindigkeit der Räder auf beiden Seiten gesteuert wird (cite paper). In diesem Fall sind an jeder Seite des Fahrzeugs zwei Räder angebracht. Die beiden Räder jeder Seite werden von einem Motor angetrieben, der über einen Keilrippenriemen mit den Rädern verbunden ist. Dies hat den Vorteil, dass die Balance des Fahrzeugs verbessert wird und eine höhere Stabilität erreicht wird. Die Konstruktion konnte aus einem Projekt des Vorsemesters übernommen werden.

Um ein reibungsloses Fahrerlebnis zu gewährleisten, ist es notwendig, die Geschwindigkeit und Richtung der Motoren individuell steuern zu können. Dies wird durch die Verwendung des zuvor beschriebenen LL-Treibers erreicht. Pro Motor wird ein Handle des LL-Treibers erstellt.

Der Fahralgorithmus wurde so konzipiert, dass er die Controllerwerte des PS4-Controllers in Geschwindigkeiten und Richtungen umwandelt. Der Wertebereich für die horizontale Achse (Lenkung) reicht von -512 bis +512, wobei -512 die maximale linke Lenkung und +512 die maximale rechte Lenkung darstellt. Der Wertebereich für die vertikale Achse (Fahren) reicht von +512 bis -512. Da diese Skalierung unintuitiv ist, wurde das Vorzeichen gedreht. Somit steht -512 für die maximale Rückwärtsfahrt und +512 für die maximale Vorwärtsfahrt. Ein Wert von 0 bedeutet, dass der Stick in der Mitte ist und somit keine Bewegung stattfindet. Dieses Szenario ist aufgrund des Stick-Drifts des linken Sticks jedoch nicht gegeben. Eine zusätzliche Prüfung im Code auf eine sog. Deadzone ist mit überschaubarem Aufwand zu erzielen, wurde aufgrund eines Workaround, der zudem auch die Anzahl der Befehle in der Queue reduziert, jedoch nicht implementiert. Weitere Informationen dazu sind im Abschnitt 3.6.3 zu finden.

Die Logik des Fahralgorithmus stützt sich dabei auf folgende Spezifikationen:

- Synchrones Fahren: Beide Motoren mit gleicher Geschwindigkeit.
- Lenkung Fahren: Unterscheidung zwischen sanfter und harter Lenkung.
  - Sanft: Inneren Motor stoppen.
  - Hart: Inneren Motor entgegengesetzt.
- Stop: Beide Motoren stoppen.
- Drehung Stand: Je nach Richtung Motoren entgegengesetzt.

Für das synchrone Fahren und die Drehung im Stand ist es wichtig, dass kleinere Änderungen wie beispielsweise Stick-Drift oder nicht perfekte Haltung des Sticks nicht unnötig zu einer abrupten Änderung führen. Deshalb wurde für das synchrone Fahren ein Hysteresebereich in X-Richtung von  $|75|$  festgelegt. Befindet sich die vertikale Achse innerhalb dieses Bereichs, so bestimmt die horizontale Achse die Geschwindigkeit. Selbiges gilt für die Drehung im Stand,

wobei hier der Hysteresebereich in Y-Richtung von  $|75|$  festgelegt wurde und die horizontale Achse die Drehrichtung bestimmt. Um dies in Kombination mit der Nullposition und den anderen Parametern zu verdeutlichen, wurde eine Abbildung erstellt, die die Logik des Differentialantriebs veranschaulicht.

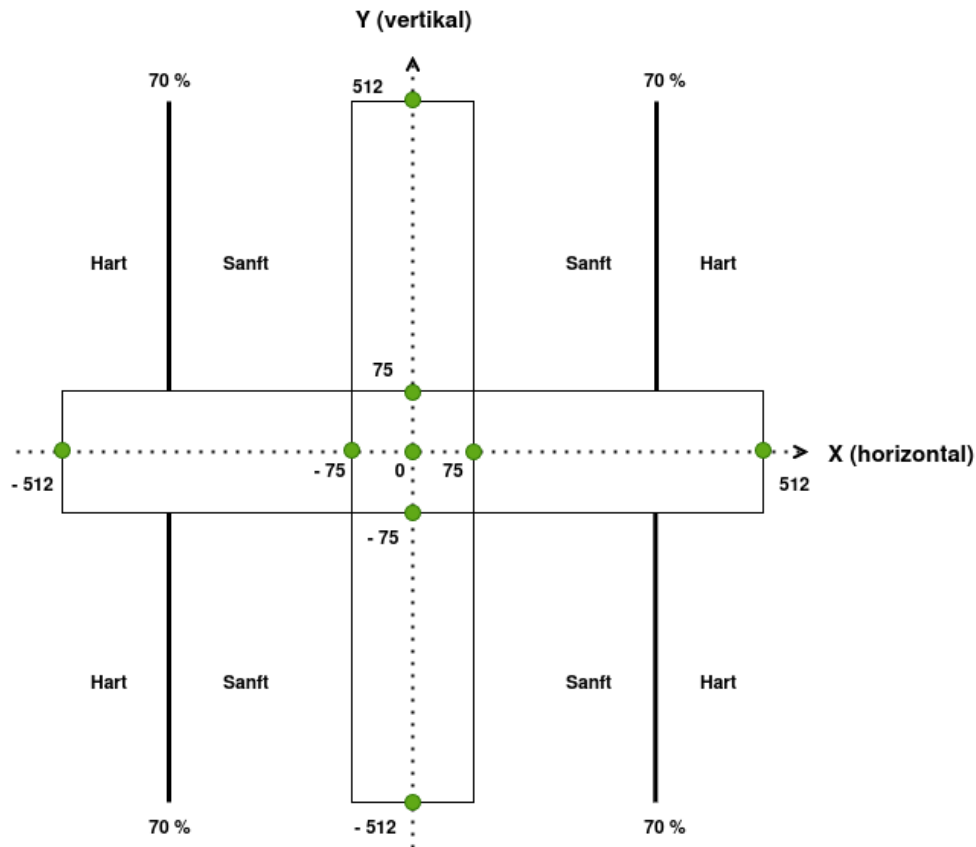


Abbildung 3.4: Differentialantrieb Logik

Im Zentrum der Abbildung 3.4 befindet sich die Nullposition. Der Grund dafür wird im Abschnitt 3.6.3 erläutert. Die beiden Kanäle im Bereich  $|75|$  in X- bzw. Y-Richtung sind die Hysteresebereiche für die Geschwindigkeit bzw. die Drehung im Stand. Die äußeren Bereiche sind für das Fahren mit Lenkung relevant. Die 70 % Markierungen symbolisieren dabei die Abgrenzung zwischen sanfter und harter Lenkung.

#### 3.4.4. Testing

Wie beim Testing des Low-Level Treibers, wurde der Code auch hier größtenteils KI-generiert. In der Summe gab es zwei Testszenarien:

- Logik-Test auf Holzkonstruktion
- Logik-Test auf Laborboden

Im ersten Fall fielen gleich zu Beginn kleinere Fehler, wie beispielsweise das fehlerhafte Freigeben von Ressourcen, auf. Diese konnten umgehend behoben werden. Trotz der Anpassungen



war ein größeres Problem ersichtlich: Die rechte Antriebsseite funktionierte nicht. Der linke Antriebsstrang lief zwar mit der korrekten Geschwindigkeit, dafür passten die Richtungswechsel nicht.

Für Debugging-Zwecke bietet das verwendete Framework ESP-IDF einen Loggingmechanismus an, der es ermöglicht, den Status von Variablen in Form von Konsolenausgaben nachzuvollziehen. Dafür ist es notwendig, dass der ESP32 über die USB-Schnittstelle mit dem Rechner verbunden ist.

Folgendes Verhalten wurde festgestellt: Die Werte wurden korrekt berechnet, die Queue jedoch nicht richtig abgearbeitet. Das Problem lag bei der Initialisierung der Queue, welche Steuerbefehle entgegennimmt. Die API erlaubt es, Pointer auf Structs oder die komplette Größe des Structs als Queuewerte festzulegen. Im Code wurde beim Laden der Werte aus der Queue das komplette Struct angegeben, beim Laden in die Queue hingegen nur ein Pointer. Aufgrund der Pointergröße konnten nur die ersten 8-Byte richtige Werte enthalten, der Rest enthielt Zufallswerte. Durch Padding/Aligning des Compilers wurde die erste Variable (Speed links, float, 4 Byte) offensichtlich auf 8 Byte skaliert. Dadurch kam es zu dem undefinierten Verhalten, dass nur die linke Seite eine Drehzahl hatte. Die Richtung wurde aus einem Standardwert übernommen. Bei der rechten Seite traf man die Annahme, dass ein Zufallswert aus dem Speicher genommen wurde. Mit großer Wahrscheinlichkeit war dieser außerhalb des Bereichs, weshalb 0 (keine Drehzahl) angenommen wurde. Nach Anpassung des Übergabeparameters liefen beide Motoren mit der gewünschten Geschwindigkeit und die Richtungen entsprechend korrekt gewechselt.

Im zweiten Fall wurde das Fahrzeug auf den Boden des Labors gestellt. Nach einem Timeout von 10 Sekunden sollte das Fahrzeug eine spezifizierte Routine durchlaufen, bei der das Drehverhalten im Stand, Beschleunigen bzw. Abbremsen und gegensätzliches Lenken getestet wurde. Die Beschleunigungstests verliefen durchwegs positiv. Tests, welche hingegen mit einer Drehung des Fahrzeugs bzw. Lenkung in Verbindung standen, wurden nicht erfolgreich abgeschlossen. Erste Versuche, die Ursache zu finden, ließen darauf schließen, dass die Motoren nicht ausreichend Leistung haben, um das Fahrzeug zu drehen. Nach eingehenden Untersuchungen - auch in Verbindung mit dem Betreuer - im Labor wurde festgestellt, dass die zuvor aufgelöteten Widerstände, die den Strom bei 6V Betriebsspannung begrenzen sollten, zu hoch dimensioniert waren. Selbst beim Anlegen einer höheren Spannung von 12V war es nicht möglich, das Fahrzeug zu drehen. Nach Entfernen der Widerstände und Anlegen von 12V Betriebsspannung konnte das Fahrzeug wie gewünscht drehen.

### 3.5. MQTT-Anbindung (Specht)

Für den Semi-automatischen Modus ist es von zentraler Bedeutung, dass das Fahrzeug Befehle für die Geschützsteuerung empfangen kann. In Abstimmung mit dem Kollegen Jürgens wurde beschlossen, die Kommunikation zwischen KI-Komponente und Fahrzeug über

MQTT umzusetzen. MQTT ist ein leichtgewichtiges Publish-Subscribe-Protokoll, das auch in Industrielösungen zum Einsatz kommt. Im ISO/OSI Referenzmodell entspricht es der Anwendungsschicht. In diesem Fall übernimmt der ESP32 die Rolle des Clients, der über einen zuvor festgelegten Topic Nachrichten empfängt. Die Vehicle-Control wurde deshalb so erweitert, dass sie auf den Topic lauscht. Demnach empfängt und verarbeitet die Vehicle-Control Nachrichten, die vom Zentralrechner, auf dem die KI-Komponente und der MQTT-Broker laufen, gesendet werden.

#### 3.5.1. WiFi-Stack

Der ESP32 bietet zwei Möglichkeiten, um eine Verbindung zu einem WLAN-Netzwerk herzustellen. Methode 1 nutzt die Hilfsfunktion *example\_connect*. Diese arbeitet beim Verbindungsaufbau nach dem Busy-Waiting-Prinzip, denn sie blockt so lange, bis eine Verbindung zum Netzwerk besteht und eine IP-Adresse zugewiesen wurde. Um die Verwendung der Funktion einfach zu gestalten, werden mögliche Fehlerfälle wie beispielsweise ein Timeout nicht ordentlich behandelt. Die Funktion ist somit nur begrenzt einsetzbar und für den produktiven Einsatz nicht geeignet.

Ein weiterer Nachteil dieser Variante ist, dass das Setzen der Credentials (SSID und Passwort) normalerweise über die Konsole mittels *idf.py menuconfig* erfolgt. Beim Testing fiel auf, dass dies auf dem im Projekt zum Einsatz kommenden ESP32 nicht funktioniert.

Die zweite Methode nutzt die WiFi-API des ESP-IDF. Trotz erhöhter Komplexität bietet dieser Weg eine bessere Fehlerbehandlung und im Allgemeinen ein flexibleres Interface. Aufgrund dessen wurde sich für die zweite Methode entschieden.

Aus dem Developer Portal von Espressif (cite) konnte eine Beispielimplementierung entnommen werden, die die grundlegende Funktionalität des Verbindungsaufbaus demonstriert. Diese wurde als Grundlage für die Implementierung der WiFi-Komponente verwendet. Trotz diverser Möglichkeiten, den WiFi-Stack noch zu optimieren, war die Idee diesen dennoch einfach zu halten. Hauptgrund für diese Entscheidung war, dass der WiFi-Stack und der damit verbundene MQTT-Stack zusätzliche Speicherressourcen benötigen, obwohl der Speicher aufgrund der Vielzahl an implementierten Komponenten bereits begrenzt ist.

#### 3.5.2. MQTT-Stack

Bevor mit der Programmierung begonnen werden konnte war es notwendig, sich mit der API des MQTT-Stacks auseinanderzusetzen. Abhilfe schaffte hierbei die Dokumentation des ESP-IDF (cite). Das Framework bietet neben TCP auch eine Reihe verschlüsselter Übertragungswege, wie beispielsweise SSL. Wegen der damit verbundenen erhöhten Komplexität und Einarbeitungszeit wurde sich aber in Absprache mit dem Kollegen Jürgens auf eine MQTT-Kommunikation mittels TCP geeinigt. Neben den Netzwerkeigenschaften und den API-Details

enthält die Dokumentation auch Referenzen zu Beispielen in deren Github-Repository. So konnte die grundlegende Funktionalität anhand des Beispielscodes zügig erstellt werden. Das Programm stellt jedoch nur eine einfache Implementierung dar, die zur Orientierung dienen soll und für den Einsatz in größeren Projekten nicht geeignet ist. Deshalb waren weitere Anpassungen notwendig. Dabei wurde auf eine Kombination aus KI-generiertem Code, Beispielcode und eigener Implementierung zurückgegriffen. Die eigene Implementierung stützt sich dabei auf die Erkenntnisse aus der Analyse der Dokumentation. Darüber hinaus wurde großer Wert auf Konformität mit den bereits implementierten Komponenten gelegt. Dies betrifft insbesondere das Abfragen der Geschütz- und Feuersteuerung, das ebenfalls über Queues realisiert wird.

Für die Verarbeitung der erhaltenen Nachrichten ist das Format der Daten von zentraler Bedeutung. Die Nachrichten werden im JSON-Format gesendet. Dies bietet einen einfachen Weg, um strukturierte Daten mit einem geringen Overhead zu übertragen. Die Nachrichten weisen dabei folgende Struktur auf:

```
{
  "platform_x_angle": -45,
  "platform_y_angle": 80,
  "fire_command": true
}
```

Beide Winkel sind in der Einheit Grad angegeben. Der X-Winkel beschreibt die horizontale Ausrichtung des Geschützes im Bereich -90 bis +90 Grad, wobei -90 Grad nach links und +90 Grad nach rechts zeigt. Die Grad-Werte für den Y-Winkel bzw. die vertikale Orientierung liegen im Bereich 0 bis 80 Grad, wobei 48 Grad der Nullposition entspricht. Die Werte sind der Platform-Control entnommen und wurden vom Kollegen Becker entsprechend getestet.

Das Kommando *fire\_command* ist ein boolescher Wert, der angibt, ob das Geschütz feuern soll oder nicht. Prinzipiell ist dieser Wert im semi-automatischen Modus nicht relevant und somit immer *false*, da der Feuerbefehl nur manuell ausgelöst werden kann. Um zukünftig eine automatische Feuersteuerung zu ermöglichen, wurde dieser Wert dennoch in die Nachrichtenstruktur aufgenommen.

Die Nachrichten werden über den Topic *vehicle/control* gesendet. Da im Projekt MQTT über TCP implementiert wurde, ist es potentiell möglich, dass von anderen Netzwerkteilnehmern Nachrichten an den Topic gesendet werden. Dies wurde jedoch nicht weiter berücksichtigt, da das Fahrzeug in einem dedizierten Netzwerk betrieben wird.

#### 3.5.3. Testing

Erste Tests der MQTT-Komponente konnten ohne Fahrzeug durchgeführt werden. Dazu wurde *mosquitto* als MQTT-Broker auf dem Rechner installiert. Mit Hilfe des Tools *mosquitto\_pub* können Nachrichten an den Topic gesendet werden. Mittels der bereits unter Ab-

schnitt 3.4.4 beschriebenen Logging-Funktionalität des ESP-IDF konnte die korrekte Verarbeitung der Nachrichten mit Hilfe von Konsolenausgaben überprüft werden. Die Nachrichten wurden dabei korrekt empfangen.

## 3.6. Integration/Fahrzeugsteuerung (Becker, Specht)

Die Fahrzeugsteuerung ist eine Abstraktionsebene, die als zentrale Anlaufstelle für Befehle vom Controller dient. Darin ist der Haupttask implementiert, der die Steuerbefehle entgegennimmt und an die entsprechenden Queues weiterleitet. Diese Queues werden dann von den Tasks der jeweiligen Komponenten abgearbeitet. Neben der Geschütz- und der Feuersteuerung ist auch die Ansteuerung der Differentialantrieb Logik eine der zentralen Komponenten. Darüber hinaus ist die Fahrzeugsteuerung auch das Modul, das letztendlich in der Main-Methode mithilfe diverser komponenten-spezifischer Strukturen initialisiert wird.

### 3.6.1. Dualshock4 Treiber (Becker)

### 3.6.2. Plattformsteuerung (Becker)

### 3.6.3. Differentialantrieb (Specht)

Die Integration in die bestehende Code-Basis war relativ unkompliziert, da das Abgreifen der Steuerbefehle über den PS4-Treiber einfach handzuhaben ist. Ein Struct enthält alle notwendigen Stick- bzw. Button-Werte, die am Controller gedrückt werden. Problematisch war jedoch der Stick-Drift des linken Sticks, denn dadurch wurde der PS4-Treiber ständig mit Werten versorgt, was zu einer permanenten Anpassung der Geschwindigkeit führte. Die ersten Befehle konnten teilweise noch übermittelt werden, jedoch lief nach kurzer Zeit die Queue über und es kam zu einem Absturz des Programms.

Die Lösung des Problems bestand darin, den Stick-Drift einzudämmen. Daraufhin wurde durch Ausprobieren am Fahrzeug ein Wert ermittelt, ab dem ein neuer Befehl interpretiert werden sollte. Dieser Wert wurde in der Fahrzeugsteuerung als Deadzone definiert. Wenn der linke Stick des Controllers innerhalb dieser Deadzone bewegt wird, wird die Geschwindigkeit des Fahrzeugs nicht angepasst. Dadurch wird verhindert, dass die Queue überläuft und das Programm abstürzt.

Allerdings wurde infolgedessen festgestellt, dass die Queue im Stand überläuft, wenn der linke Stick nicht bewegt wird. Das Problem war, dass die bis dato verwendete Logik die Nullposition als normalen Fahrbefehl interpretierte und somit ständig Werte sendete.

Deshalb wurde, wie im Zentrum der Abbildung 3.4 zu sehen, eine Nullposition definiert. Diese wird beim Auftreten, sprich Nicht-Bewegen des linken Sticks, detektiert. Daraufhin wird ein Stop eingeleitet. Sobald dieser abgearbeitet wurde wird ein Flag gesetzt, das anzeigt, dass die Nullposition bereits abgearbeitet wurde. Somit wird verhindert, dass das Stoppen unnötig

wiederholt wird und die Queue überläuft. Sobald der linke Stick wieder bewegt wird, wird das Flag zurückgesetzt und die Geschwindigkeit des Fahrzeugs angepasst.

#### **3.6.4. MQTT-Anbindung (Specht)**

## 4. Raspberry Pi Programmierung

### 4.1. Lautsprecher (Becker)

### 4.2. Gyrosensor Programmierung (Koch)

Zu Beginn des Projekts war geplant den Gyrosensor MPU6050 zu nutzen, um die Position der Geschützplattform zu bestimmen, da eine kontinuierliche Bewegung um die eigene Achse aufgrund der Kabel nicht möglich ist. Dabei sollte der Sensor über den I2C-Bus mit dem Raspberry Pi 5 verbunden werden, um die Daten auszulesen und zu verarbeiten. Der MPU6050 ist dabei ein 3-Achsen-Gyroskop und 3-Achsen-Beschleunigungssensor, welcher entsprechende Drehbewegungen und Beschleunigungen entlang der Raumachsen messen kann, welche für eine genaue Winkelbestimmung notwendig sind.

Nach kurzem Einlesen in die Dokumentation waren erste Rohdaten leicht auszulesen. Diese Rohdaten liegen in Form von 16 Bit in zwei Registern bereit und haben die Einheit  $LSB/g$  für die Beschleunigungswerte und  $LSB/^\circ/s$  für die Gyroskop-Werte. Dies gilt es in tatsächliche physikalische Größen umzuwandeln, was bei unserem Projekt letztlich einem Winkel entspricht. Um die Beschleunigungswerte nutzen zu können, muss dafür mittels des Skalierungsfaktors die Fallbeschleunigung  $g$  errechnet werden, indem man den erhaltenen Wert  $x/16384$  rechnet. Die 16384 ergeben sich aus der Dokumentation und entsprechen den LSB bei einem Messbereich von  $\pm 2g$ , welches der Standardauflösung entspricht und auch die höchste Auflösung des MPU6050 für ist. Ähnlich wird nun auch Winkelgeschwindigkeit ( $^\circ/s$ ) errechnet. Hierbei beträgt der Teiler standardmäßig 131. [16, S. 12-13]

Nach der Umrechnung der Rohdaten in physikalische Größen können nun die Neigungswinkel (Roll- und Pitch-Winkel) des Sensors berechnet werden. Diese ergeben sich aus der Richtung der Erdbeschleunigung relativ zum Sensor.

Dazu wird die Erweiterung des Arkustangens genutzt, genauer gesagt die Funktion `atan2`, da sie im Gegensatz zum gewöhnlichen Arctangens auch die Orientierung in allen vier Quadranten berücksichtigt und somit stabile Winkelwerte über den gesamten Bereich von  $-180^\circ$  bis  $+180^\circ$  liefert. [17]

Die Winkelberechnung erfolgt nach der Formel 4.1:

$$\begin{aligned}\varphi_{\text{pitch}} &= \arctan 2 \left( a_x, \sqrt{a_y^2 + a_z^2} \right) \cdot \frac{180^\circ}{\pi} \\ \varphi_{\text{roll}} &= \arctan 2 (a_y, a_z) \cdot \frac{180^\circ}{\pi}\end{aligned}\tag{4.1}$$

Hierbei sind  $a_x$ ,  $a_y$  und  $a_z$  die normierten Beschleunigungswerte in  $g$ -Einheiten (berechnet aus den Rohwerten durch Division durch 16384).

Die `atan2`-Funktion liefert den Winkel zwischen der positiven  $x$ -Achse und dem Punkt  $(x, y)$  in der Ebene, wodurch Sprünge oder Mehrdeutigkeiten bei  $90^\circ$  vermieden werden. [17]

Nachdem der Term im Code implementiert wurde, konnte ein starkes Rauschen beobachtet werden, was zunächst auf natürliche Schwankungen des Sensors zurückgeführt wurde, weshalb sich dazu entschieden wurde zuerst einen Komplementärfilter zu implementieren, welcher allerdings das Problem nur bedingt beheben konnte. Deshalb wurde auch noch der Kalman-Filter ausgetestet, wodurch auch eine starke Rauschunterdrückung festgestellt werden konnte, doch auch hier zeichnete sich ein überdurchschnittliches Rauschverhalten ab, weshalb der Fehler nicht mehr auf ein natürliches Rauschen zurückzuführen war. Daraufhin wurde auch ein zweiter MPU6050 getestet, welcher ebenfalls dieses Verhalten aufwies, wodurch klar wurde, dass es sich hierbei um einen Programmierfehler handeln muss. Dieser konnte nach einiger Zeit auch herausgefunden werden und lag an der Interpretation der Rohdaten, welche fälschlicherweise als `int16_t` statt `uint16_t` interpretiert wurden.

Aufgrund der aufgewendeten Zeit für die Implementierung des Kalman-Filters sollte dieser aber trotzdem Anwendung im Projekt finden und wird in 4.2.1 behandelt.

Wie Eingangs erwähnt, sollte der Gyrosensor MPU6050 genutzt werden, um die Position der Geschützplattform zu bestimmen, wofür der Winkel der Drehung um die eigene Achse benötigt wird. Dabei stellte sich heraus das der MPU6050 für diesen Wert zu einem starken Drift neigt, weshalb empfohlen wird diesen mit einem Magnetometer zu kombinieren [16, S. 26]. Zur gleichen Zeit stellte sich heraus, dass diese Funktionalität nicht benötigt wird, da über die Servo-Motoren bereits ein Nullpunkt definiert werden konnte, weshalb der Gyrosensor letztlich nur noch für die Neigung der Geschützplattform genutzt wird, um diese als Debug-Information auf dem Webserver 5 anzuzeigen.

### 4.2.1. Kalman Filter Implementierung (Koch)

Der Kalman-Filter ist ein Algorithmus zur Schätzung des Zustands eines dynamischen Systems. Er nutzt Messwerte mit einem mathematischen Modell, um aus verrauschten Daten optimale Schätzungen zu erzeugen und zu filtern, was insbesondere bei Sensoren mit Rauschen, wie dem MPU6050, von Bedeutung ist [18]. In der Abbildung 4.1 ist sowohl für den Roll, als auch den Pitch ein signifikanter Unterschied zwischen den Rohdaten und den gefilterten Daten zu erkennen. Die ersten Schwankungen der gefilterten Werte sind vermutlich darauf zurückzuführen, dass der Kalmanfilter mit wenig Referenzwerten nur ungenaue Schätzungen liefern kann, weshalb die Schätzungen auch erst mit steigender Anzahl an Werten genauer werden. Der Pitch konnte mithilfe des Kalmanfilters eine Rauschreduktion von 59.4% erzielen und für den Roll 37.4% bei jeweils 100 Testwerten.

Aufgrund der Art wie der Gyrosensor auf dem Geschützarm montiert ist, ist für die Neigung allerdings nur der Roll-Wert von Bedeutung.



Abbildung 4.1: Vergleich der Rohdaten und Kalman-Filter geschätzten Werte für Pitch und Roll



## 5. Webserver (Koch)

### 5.1. Funktion und Anforderung des Webservers

Der Webserver des Projekts sollte die zentrale Schnittstelle zwischen verschiedenen Sensoren werden und sowohl Videomaterial mit bereits verarbeiteten KI-Informationen darstellen können und wissenswerte Daten wie die Entfernung des Fliegers oder die Geschützneigung anzeigen. Als zentrales Problem stand dabei das Videomaterial mit entsprechenden Bounding Boxes anzuzeigen im Vordergrund, da bereits über das ganze Projekt hinweg die KI die leistungsintensivste Aufgabe war und entsprechende Latenzprobleme bereits auf das Minimum verringert wurden. Der Grundlegende Aufbau der KI-Verarbeitung mitsamt der Videoausgabe ist in ?? erklärt.

### 5.2. Lösungsansatz und Umsetzung

Ein Lösungsansatz diesbezüglich war deshalb, dass die Sensordaten vom Raspberry Pi per MQTT an den PC geschickt werden, welcher die KI-Verarbeitung übernimmt, um die Daten direkt im Videostream abzubilden. Das Video mit den Bounding Boxes und den Sensordaten sollte dann zurück auf den MediaMTX-Server geschickt werden, worüber man dann das Video sehen kann. Dieser Ansatz war allerdings keine geeignete Lösung, da die Verarbeitung zu lange dauerte und somit eine zu geringe Framerate erreicht wurde, wodurch dies keine annehmbare Lösung darstellte.

Deshalb wurde der alternative Ansatz gewählt, dass der Raspberry Pi selbst den Webserver bereitstellt, da dieser bereits die Sensordaten und den MediaMTX-Server bereitstellt. Das Problem hierbei war jedoch, dass somit zwar der Videostream und die Sensordaten dargestellt werden konnten, allerdings noch keine Bounding Boxes, welche eine harte Anforderung an den Webserver waren. Es musste also eine Lösung gefunden werden, den verarbeiteten Videostream mit den Bounding Boxes vom PC zum Raspberry Pi zu übertragen, doch das senden des Videostreams ist nicht ohne Verluste möglich, wie bereits erwähnt wurde.

Die Umsetzung erfolgte über einen zentralen Knotenpunkt auf dem Raspberry Pi, welcher sowohl die Bereitstellung der Benutzeroberfläche als auch die Echtzeitkommunikation mit den Sensoren und der externen KI-Komponente zur Objekterkennung übernimmt. Die Kommunikation erfolgt über WebSockets, wobei sowohl Sensordaten als auch Bounding-Box-Koordinaten empfangen, verarbeitet und an das Frontend weitergeleitet werden. Die Darstellung im Browser kombiniert dabei Livevideo, Sensordaten und erkannte Objekte in einem einheitlichen Interface. Dabei wird die Entfernung nur in Kombination mit der Bounding Box ausgegeben wie in Abbildung 5.1 zu sehen. Während der Entwicklung zeigte sich, dass die zunächst gewählte

Architektur zur Sensoranbindung zu erheblichen Verzögerungen führte. Die ursprüngliche Implementierung versuchte, beide Sensoren (Gyrosensor und Ultraschallsensor) über asynchrone Generatoren auszulesen, jedoch geschah dies nur sequentiell. Vermutlich bremste dabei die Verarbeitung des Ultraschallsensors, der mit geringerer Frequenz Daten liefert, die Verarbeitung des Gyrosensors aus. In der Folge wurden die Gyroskopdaten gepuffert und verzögert ausgegeben. Um dieses Problem zu beheben, wurde die Architektur angepasst und die Sensorverarbeitung parallelisiert. Statt beide Generatoren sequentiell abzufragen, wurden zwei Tasks erstellt, die jeweils eigenständig mit ihrem Sensor kommunizieren. Dadurch können beide Datenströme unabhängig voneinander verarbeitet und weitergeleitet werden, was die Reaktionszeit des Systems verbesserte und die gleichzeitige Darstellung der Sensorinformationen im Frontend ermöglichte.

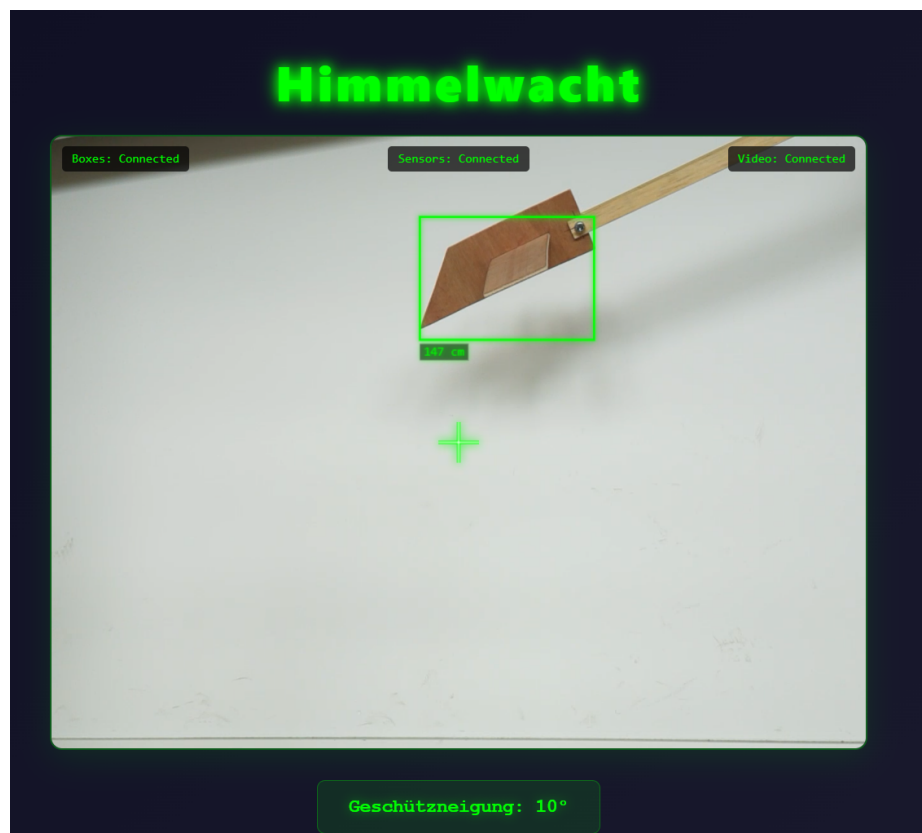


Abbildung 5.1: Webinterface zur Darstellung des Videostreams mit Bounding Boxes und Sensordaten

# Stundenliste

**Name:** Fabian Becker  
**Gruppe:** 1 HimmelWacht

Pos.	Bezeichnung	Beschreibung	Stunden in h
1	Vorbereitung	Gruppenbildung, Projektfindung, erste Recherche	4
2	Projektorganisation	Erstellung Projektbeschreibung, Terminplan & Plakat	8
3	CAD Design	Einarbeitung FreeCAD	5
4	CAD Design	Entwurf Plattform, Anpassung Magazin & Schussarm	11
5	Ps4 Controller Treiber	Einarbeitung ESP-IDF, erster Entwurf mit Bluetooth HID	3
6	Ps4 Controller Treiber	Debugging PS4 Treiber aufgrund ESP Freeze	10
7	Ps4 Controller Treiber	Erstellung ESP32 Projekt mit Bluepad32 Dependencies	2
8	Ps4 Controller Treiber	Erstellung Custom Bluepad32 Plattform	4
9	Ps4 Controller Treiber	Implementierung Datenabgriff, Vibration, Farbwechsel	8
10	Ps4 Controller Treiber	Treiber auf Multiprozessor portiert	6
11	Ps4 Controller Treiber	Low Battery Warning durch rote blinkende LED	3
12	CAD Design	Design + Druck für Stromversorgung und Motortreiber	6
13	PWM Treiber	Portierung PCA9685 für Servosterung	4
14	Plattformsteuerung	Gradweise Ansteuerung für Servomotoren implementiert	6
15	CAD Design	Redesign + Druck Schussarm und Magazin	5
16	Lautsprecher	Treiberboard gelötet und Test Lautsprecher	5
17	Flywheel Motoren	Test Flywheelmotoren + Implementierung Treibercode	5
18	Geschütz	Zusammenbau, Kalibrierung Servos	10
19	Geschütz	Integration PS4 Treiber	15
20	CAD Design	Druck & Design Mikrocontroller Cases	10
21	Stromversorgung	Finale Verkabelung ESP32	6
22	Gesamttests	Integriertes Fahrzeug getestet (Controller & Plattform)	4
23	Plattformsteuerung	Implementierung Moduswechsel (manuel <-> KI)	2
24	Ps4 Controller Treiber	Tiefpassfilter zur Unterdrückung von Jitter implementiert	3
25	Abschlusspräsentation	Präsentationsfolien erstellt & Bilder aufgenommen	3
26	Objekttracking	Implementierung & Testen des Objekttrackings	3
27	Dokumentation	Erstellung der wissenschaftlichen Dokumentation	20
<b>Gesamt:</b>			<b>171</b>

# Stundenliste

Name: Michael Specht  
 Gruppe: 1 Himmelwacht

Pos.	Bezeichnung	Beschreibung	Stunden in h
1	Einführung	Gruppe, Thema, Herangehensweise	4
2	Einführung	GANTT, Lastenheft, Einteilung, Teileliste	8
3	Flieger	Fliegermodell inkl. Befestigungsarm bauen	2
4	CAD	Einführung in 3D-Druck, Software, Aufteilung	5
5	CAD	Geschützarm v1	15
6	Recherche	Motor-Control-PWM (MCPWM), Motor + Treiber	5
7	HW Setup	Widerstände + Kondensatoren löten	2
8	ESP-IDF	Programmierung Motortreiber	15
9	ESP-IDF	Programmierung Differential Drive	10
10	Test	Test am Fahrzeug + Debugging (Motortreiber)	5
11	Test	Test am Fahrzeug + Debugging (Differential Drive)	10
12	Integration	Differential Drive, Steuerung mittels Controller	5
13	Integrationstest	Deadzones + Algorithmus anpassen	10
14	HW Setup	Debugging, Widerstände entfernen, neuer Akku	2
15	CAD	Geschützarm v2	10
16	CAD	Pololu Halterung	1
17	HW Setup	Diverse Arbeiten für Zusammenbau	3
18	HW Setup	Riemen-Trieb zerlegt, gereinigt, zusammengebaut	2
19	Test	Leistung neue Flywheelmotoren	2
20	ESP-IDF	Logik-Umbau auf Festkommaarithmetik	3
21	ESP-IDF	Integration WiFi- + MQTT-Stack	15
22	Integration	Test am Fahrzeug + Debugging (MQTT)	5
23	3D-Druck	Diverse Drucks begleitet (Slicer, Infill entfernen)	5
24	Präsentation	PowerPoint anpassen	3
25	Objekttracking	Implementierung + Testen des Objekttrackings	6
26	Dokumentation	Wissenschaftliche Arbeit verfassen	20
Gesamt:			173

## Abbildungsverzeichnis

2.1	Geschützarm Version 1 . . . . .	6
2.2	Geschützarm Version 2 . . . . .	7
2.3	Montagehalterung für Pololu-Motortreiber . . . . .	7
2.4	Abdeckung Plattform mit Position des Verbindungsstücks . . . . .	8
2.5	Magazingewicht . . . . .	9
2.6	Halterung DC Step-Down mit Stecksystem . . . . .	10
2.7	Halterung für Flywheel Motortreiber . . . . .	11
2.8	Mikrocontroller Case . . . . .	11
2.9	Lautsprecherhalterung . . . . .	12
3.1	Beispielhafter Aufbau eines HID-Reports aus [5] . . . . .	14
3.2	Erklärung ON/OFF Parameter für PCA9685 aus [14, S. 17] . . . . .	17
3.3	Testaufbau des Fault-Pins des Pololu-Motor-Treibers . . . . .	21
3.4	Differentialantrieb Logik . . . . .	23
4.1	Vergleich der Rohdaten und Kalman-Filter geschätzten Werte für Pitch und Roll	31
5.1	Webinterface zur Darstellung des Videostreams mit Bounding Boxes und Sensordaten . . . . .	33

## Tabellenverzeichnis

## Literaturverzeichnis

- [1] Elephant333. „Nerf Turret - Modular Arduino Tank (M.A.T).“ (26. Mai 2022), Adresse: <https://www.thingiverse.com/thing:4870102/files> (besucht am 01.07.2025).
- [2] Higany. „Holder for XY-MOS D4184 Power MOSFET breakout module.“ (1. Jan. 2023), Adresse: <https://www.printables.com/model/355368-holder-for-xy-mos-d4184-power-mosfet-breakout-modu> (besucht am 01.07.2025).
- [3] A. Whizzbizz. „Case and camera clip for Freenove Breakout Board for ESP32 / ESP32-S3.“ (15. Jan. 2024), Adresse: <https://www.printables.com/model/723594-case-and-camera-clip-for-freenove-breakout-board-f> (besucht am 01.07.2025).
- [4] „Getting Started with ESP-IDF.“ (), Adresse: <https://idf.espressif.com/> (besucht am 01.07.2025).
- [5] Matlo. „DS4-BT.“ (15. Jan. 2015), Adresse: [http://eleccelerator.com/wiki/index.php?title=DualShock\\_4](http://eleccelerator.com/wiki/index.php?title=DualShock_4) (besucht am 02.07.2025).
- [6] USB Implementers' Forum. „Device Class Definition for Human Interface Devices (HID).“ (25. Juli 2001), Adresse: [https://usb.org/sites/default/files/hid1\\_11.pdf](https://usb.org/sites/default/files/hid1_11.pdf) (besucht am 02.07.2025).
- [7] „Bluetooth® HID Host API.“ (2025), Adresse: [https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/bluetooth/esp\\_hidh.html](https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/bluetooth/esp_hidh.html) (besucht am 02.07.2025).
- [8] „ESP HID Host API Issues.“ (17. Feb. 2025), Adresse: <https://github.com/espressif/esp-idf/issues/9767> (besucht am 02.07.2025).
- [9] Espressif. „ESP32 Series Datasheet.“ (), Adresse: [https://www.espressif.com/sites/default/files/documentation/esp32\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf) (besucht am 01.07.2025).
- [10] „DC Toy / Hobby Motor - 130 Size.“ (), Adresse: <https://www.adafruit.com/product/711> (besucht am 01.07.2025).
- [11] „TowerPro MG92B Servo.“ (), Adresse: <https://servodatabase.com/servo/towerpro/mg92b> (besucht am 01.07.2025).
- [12] „TowerPro MG996R Servo.“ (), Adresse: <https://servodatabase.com/servo/towerpro/mg996r> (besucht am 01.07.2025).
- [13] J. Supcik. „ESP/IDF PCA9685 I2C PWM Driver.“ (19. Okt. 2024), Adresse: <https://github.com/supcik/idf-pca9685> (besucht am 01.07.2025).
- [14] NXP Semiconductors. „PCA9685 Product data sheet.“ (16. Apr. 2015), Adresse: <https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf> (besucht am 01.07.2025).

- [15] „SG90 - Servomotor.“ (), Adresse: [https://www.elektronik-kompodium.de/sites/praxis/bauteil\\_sg90.htm](https://www.elektronik-kompodium.de/sites/praxis/bauteil_sg90.htm) (besucht am 02.07.2025).
- [16] I. InvenSense. „MPU-6000 and MPU-6050 Product Specification.“ (Aug. 2013), Adresse: <https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf>.
- [17] MathWorks. „atan2.“ (), Adresse: <https://www.mathworks.com/help/matlab/ref/double.atan2.html>.
- [18] R. Aachen. „Kalmanfilter.“ (), Adresse: [https://www.irt.rwth-aachen.de/global/show\\_document.asp?id=aaaaaaaaacdvoafc](https://www.irt.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaacdvoafc).