



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

Inhaltsverzeichnis

1 Stromversorgung	4
1.1 Analyse des Aufbaus und der Komponenten des vorherigen Projekts (Koch)	4
1.2 Aufbau der eigenen Stromversorgung (Koch)	4
2 CAD-Konstruktion	6
2.1 Setup und Einarbeitung (Becker, Specht)	6
2.2 Geschützarm Version 1 (Specht)	6
2.3 Geschützarm Version 2 (Specht)	7
2.4 Montagehalterung Motortreiber (Specht)	8
2.5 Geschützplattform (Becker)	9
2.5.1 Unterbau	9
2.5.2 Abdeckung	9
2.6 Magazin und Verbindungsstück (Becker)	10
2.7 Magazingewicht (Becker)	10
2.8 Geschütztrigger (Becker)	11
2.9 Halterungen Stormversorgung (Becker)	11
2.10 Halterungen Flywheel Motortreiber (Becker)	11
2.11 Mikrocontroller-Case (Becker)	12
2.12 Halterung Lautsprecher (Becker)	13
3 ESP32 Programmierung	14
3.1 Einführung (Becker, Specht)	14
3.2 Dualshock4 Treiber (Becker)	14
3.2.1 Übertragung	15
3.2.2 Version 1: eigens entwickelter Treiber	15
3.2.3 Version 2: Treiber basierend auf der Bluepad32 Bibliothek	16
3.2.4 Testing	18
3.3 Plattformsteuerung (Becker)	18
3.3.1 PWM Board Treiber (Becker)	19
3.3.2 Ansteuerung der Servo-Motoren (Becker)	19
3.3.3 Ansteuerung der Flywheel Motoren (Becker, Koch, Wohlrab)	21
3.4 Motor-Treiber (Specht)	21
3.4.1 Low-Level Treiber	22
3.4.2 Testing	23
3.4.3 Differentialantrieb	24

INHALTSVERZEICHNIS

3.4.4	Testing	25
3.5	MQTT-Anbindung (Specht)	27
3.5.1	WiFi-Stack	27
3.5.2	MQTT-Stack	28
3.5.3	Testing	29
3.6	Integration/Fahrzeugsteuerung (Becker, Specht)	29
3.6.1	Regelschleife (Becker)	29
3.6.2	Plattformsteuerung & Schussabgabe (Becker)	30
3.6.3	Differentialantrieb (Specht)	31
3.6.4	MQTT-Anbindung (Specht)	32
4	Raspberry Pi Programmierung	35
4.1	Einführung (Becker)	35
4.2	Lautsprecher (Becker)	35
4.3	Gyrosensor Programmierung (Koch)	36
4.3.1	Kalman Filter Implementierung (Koch)	38
5	Erstellung & Ausführung eines KI-Modells zur Objektdetektion	41
5.1	Anforderungen an das KI-Modell (Jürgens)	41
5.2	Modellauswahl (Jürgens)	41
5.2.1	Das ONNX-Format (Jürgens)	42
5.3	Trainingsdaten & Annotation (Jürgens)	42
5.4	Training des Modells (Jürgens)	44
5.5	Ausführung des Modells im ONNX-Format (Jürgens)	45
5.6	Inferenz des PyTorch-Modells (Jürgens)	46
5.6.1	Installation der KI-Abhängigkeiten (Jürgens)	46
5.6.2	Erstellung & Evaluierung der GStreamer-Pipeline (Jürgens)	47
5.6.3	Inferenz des Modells auf dem Raspberry Pi 5 (Jürgens)	47
5.6.4	Inferenz des Modells auf dem Laborrechner (Jürgens)	48
6	Einrichtung einer Ultra-Low-Latency Videoübertragung	51
6.1	Anforderungen an die Videoübertragung (Jürgens)	51
6.2	Grundlegende Informationen zu WebRTC (Jürgens)	51
6.3	Herstellung der WebRTC-Verbindung (Jürgens)	51
7	Objekttracking	53
7.1	Ziel des Objekttrackings (Jürgens, Specht)	53
7.2	Einrichtung des Kommunikationskanals (Jürgens)	53
7.3	Implementierung des Objekttrackings (Jürgens, Specht)	54
8	Webserver	57
8.1	Funktion und Anforderung des Webservers (Koch)	57
8.2	Lösungsansatz und Umsetzung (Koch)	57

INHALTSVERZEICHNIS

Abbildungsverzeichnis	64
Literaturverzeichnis	65

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 selbstangeschafftes 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 Motortrieber 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. Mehrere Beispielprojekte aus dem Internet dienten als Grundlage für die Arbeit [1–3].

2.2. Geschützarm Version 1 (Specht)

Bevor mit der Konstruktion begonne 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 [1] 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 [1] 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 [4] übernommen und angepasst. Auch für die Schrittmotoren konnte auf ein Modell aus dem Internet [5] zurückgegriffen werden, weshalb es nicht nötig war, die komplexe Geometrie eigenständig zu vermessen. Einzig die Maße für den Ultraschall-Sensor wurden recherchiert [6] und durch Nachmessen validiert.

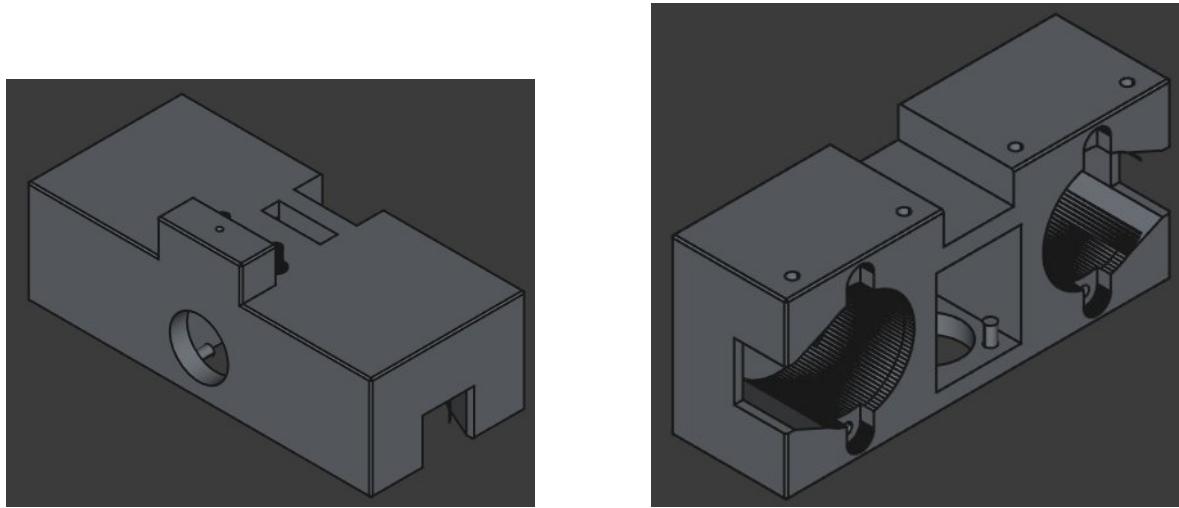


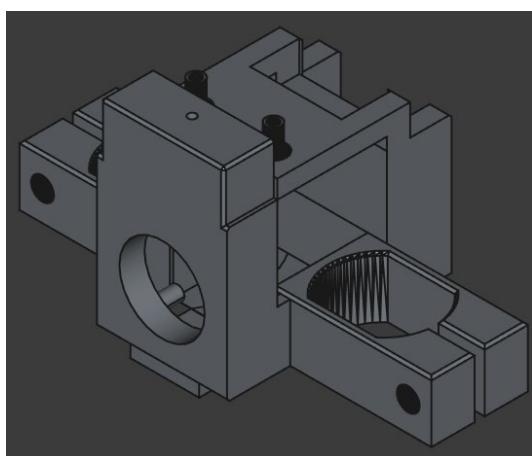
Abbildung 2.1: Geschützarm Version 1

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 neu konstruierte Gehäuse integriert, sodass Aussparungen, wie in Abbildung 2.1 zu sehen, angebracht werden mussten, um die Kabel aus dem Gehäuse herauszuführen.

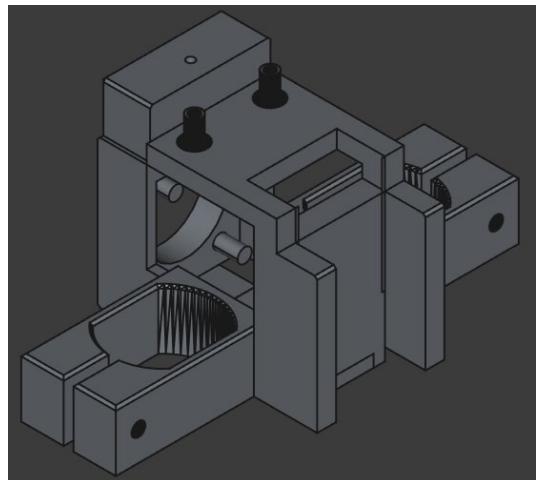
Außerdem musste sichergestellt werden, dass der Geschützarm an das Magazin montiert werden kann. Hierzu wurden vom Kollegen Fabian Becker Montagepunkte am Magazin konstruiert, die mit dem Geschützarm verschraubt werden können.

2.3. Geschützarm Version 2 (Specht)

Im Laufe des Projektes wurde in Abstimmung mit Fabian Becker klar, dass die initial angedachten Schrittmotoren aufgrund zu geringer Leistung nicht für die Flywheel-Konstruktion geeignet sind. Daraufhin wurde sich für die originalen Motoren aus dem Beispielprojekt [1] entschieden. Das hatte zur Folge, dass der Geschützarm neu konstruiert werden musste, da die Maße der neuen Motoren von den Alten abwichen. Aus diesem Grund wurde der Geschützarm der Vorlage [1] als Basis genommen und die Grundidee der Version 1 beibehalten.



(a) Geschützarm Version 2 - Frontansicht



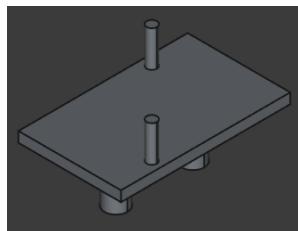
(b) Geschützarm Version 2 - Rückansicht

Abbildung 2.2: Geschützarm Version 2

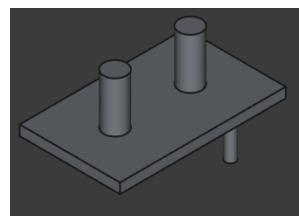
Im Gegensatz zur Version 1 wird der Ultraschallsensor nun seitlich eingeführt anstatt von unten, siehe 2.2. Problematisch waren dabei die beschränkten Platzverhältnisse, da die Schrittmotoren näher am Kanonenrohr angebracht wurden als im vorherigen Entwurf. Außerdem wurden die zuvor angedachten Montagepunkte 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) Pololu - Draufsicht



(b) Pololu - 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. Die Maße bzw. Position der Löcher wurden dabei manuell mit Geodreieck vermessen. 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 worden 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. Sie verfügt über Bohrungen zur Verbindung mit dem im Unterbau positionierten Servomotor, sowie über Montagepunkte für den Geschützarm.

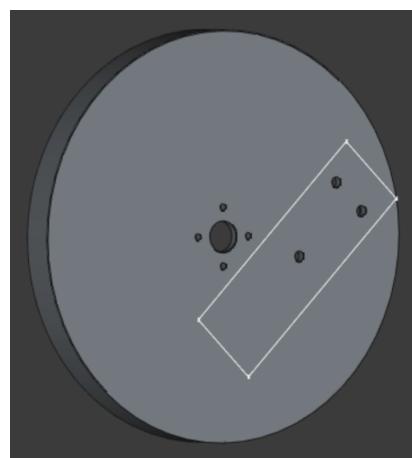


Abbildung 2.4: Abdeckung Plattform mit Position des Verbindungsstücks

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 konstruieren.

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 von sechs Nerf-Darts sowie dem Magazingewicht. Letzteres dient dazu, einerseits bei hoher Vibration des Fahrzeugs das Herausfallen der Darts zu verhindern und um andererseits sicherzustellen, dass nach einem Schuss das nächste Geschoss nachrutscht.

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

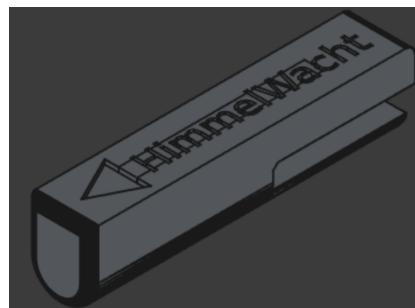


Abbildung 2.5: Magazingewicht

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 deutlich schmäleren Lauf zu schieben, was zu Materialschäden, entweder am Motor oder am Geschütz führen würde.

Die vorliegende Aussparung führte allerdings zu Problemen. Wird die Platform 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 von selektivem Infill, der Infill bezeichnet hierbei den prozentualen Füllgrad des Innenvolumens eines 3D-gedruckten Bauteils. Die übrigen Teile wurden standardmäßig mit einem Infill-Gehalt von 15% gedruckt, im hinteren Teil des Gewichts wurde jedoch 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 Stromversorgung (Becker)

Da einige Komponenten der Stromversorgung ebenfalls aus einem früheren Projekt herangezogen 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 unser Projekt wurde ein weiteres Teil nach gleichem Prinzip für die Step-Down-Module erstellt, um sie in gleicher Weise befestigen zu können.

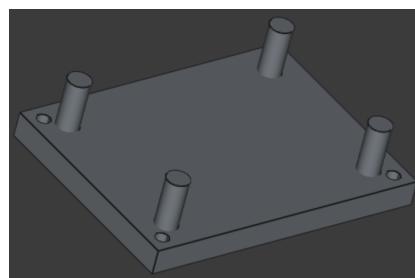


Abbildung 2.6: Halterung DC Step-Down mit Stecksystem

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. In

der ursprünglichen Konstruktion besaß die Vorlage seitliche Schraublöcher, welche jedoch im Zuge der Implementierung des Stecksystems entfernt wurden.

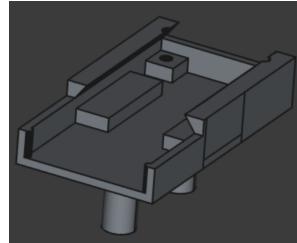


Abbildung 2.7: Halterung für Flywheel Motortreiber

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. Zuerst 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.

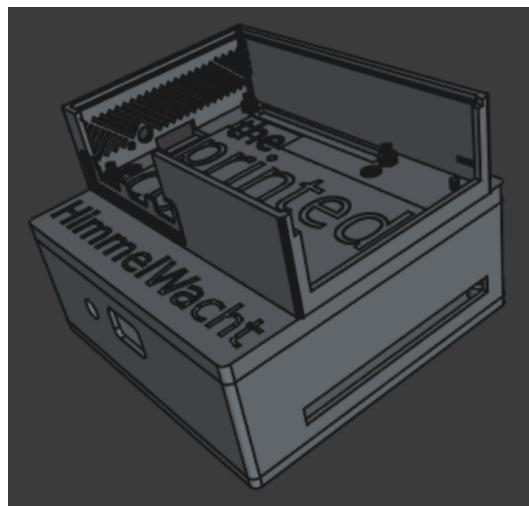


Abbildung 2.8: Mikrocontroller Case

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 einfachere Verkabelung gewährleistet werden. Diese Halterung ist ebenfalls mit dem modularen Stecksystem ausgestattet.

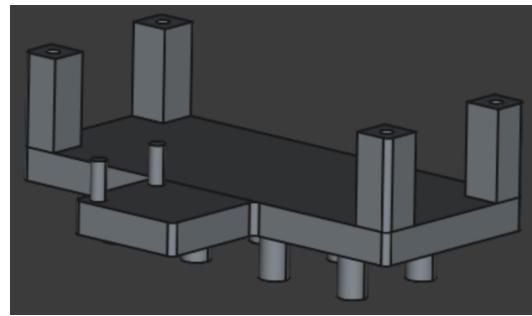


Abbildung 2.9: Lautsprecherhalterung

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) [7]. 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, zusätzlich fungiert sie auch 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 die Implementierung von Funktionen, die Vibrationen und Farbwechsel am Gamepad auslösen können 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 dann 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 [8].

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 [9] 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 [8]

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 [10], wurde im ersten Ansatz ein eigener Treiber entwickelt.

Der Verbindungsaufbau verlief problemlos, und zu Beginn wurden die Input-Reports korrekt empfangen. Nach kurzer Zeit brach 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 [11]. Als Ursache wird die vergleichsweise hohe Sendefrequenz des Controllers vermutet, die laut Nutzertests bei über 700 Hz liegt [8]. 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 brach der Controller die Übertragung der Reports erneut ab, obwohl die Status-LED weiterhin eine aktive Bluetooth-Verbindung anzeigen sollte.

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

Als Alternative wurde die Bibliothek Bluepad32 evaluiert. Diese ermöglicht das Auslesen von Gamepad-Daten auf verschiedenen Mikrocontrollern, darunter auch dem ESP32.

Ein wesentlicher Vorteil dieser Bibliothek liegt darin, dass bereits eine Vielzahl von Controllern unterstützt wird. Darüber hinaus erlaubt sie auch die Steuerung von Zusatzfunktionen wie der Vibrationsmotoren und der LED-Farbleiste des Gamepads. Im Gegensatz zum im ESP-IDF integrierten Bluedroid Bluetooth-Stack verwendet Bluepad32 den sogenannten BT-Stack. Dieser wurde ursprünglich für 8- und 16-Bit-Mikrocontroller entwickelt und ist dadurch deutlich ressourcenschonender. [12]

Ein entscheidender Nachteil ist jedoch die vergleichsweise geringe Dokumentation zur Integration in das IoT Development Framework (ESP-IDF). Im Gegensatz zum Arduino-Framework ist die Unterstützung hier deutlich schwächer ausgeprägt, daher wurde Bluepad32 im vorherigen Schritt zunächst verworfen. Die Möglichkeit, im Code-Editor direkt zu den verwendeten Funktionsdefinitionen springen zu können, erwies sich in diesem Fall als sehr hilfreich. [13]

Letztlich konnte ein funktionsfähiger Treiber implementiert werden, der neben dem Auslesen der Controllereingaben auch die Steuerung der Vibration und Farbwechsel unterstützt. Der schematische Aufbau des Treibers ist in Abbildung 3.2 dargestellt:

- Der **bluepad32_task** führt sowohl den BTStack als auch die Bluepad32 Bibliothek aus. Über die dort eingebundene Plattform (eine Menge von Callback-Funktionen) erfolgt die Kommunikation mit dem Gamepad. Beispielsweise wird beim Eintreffen eines

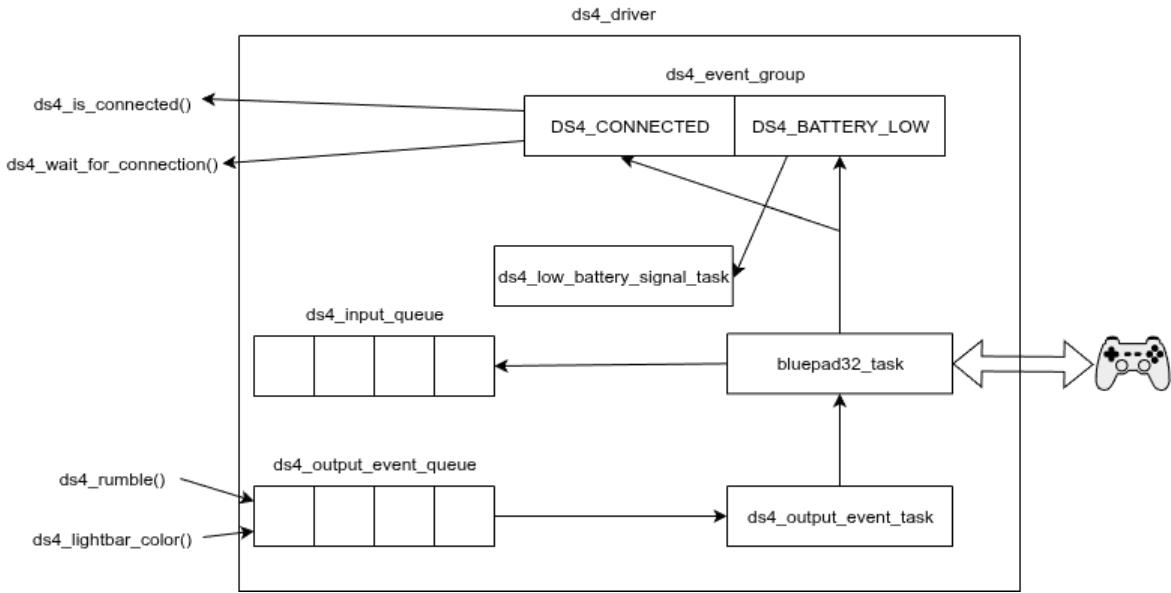


Abbildung 3.2: Aufbau des Dualshock4 Treibers

Controller-Datenpakets (Callback *on_controller_data*) die **ds4_input_queue** mit aktuellen Informationen zu Tasten, Sticks und Batteriestand gefüllt. Dies erfolgt mit einer Frequenz von 60 Hz. Zusätzlich wird überprüft, ob der Batteriestand unter eine kritische Grenze fällt, in diesem Fall wird das Event-Bit *DS4_BATTERY_LOW* gesetzt bzw. wieder entfernt. Die Callbacks *on_device_ready* und *on_device_disconnected* aktualisieren den Verbindungsstatus, welcher im Event-Bit *DS4_CONNECTED* gespeichert ist.

- Die **ds4_input_queue** ist eine von FreeRTOS bereitgestellte Queue, die thread- bzw. multiprozessorsicher ist. Hier kann der aktuelle Zustand des Controllers am Treiberinterface abgegriffen werden.
- Der Verbindungsstatus kann über die Funktionen **ds4_is_connected()** (nicht blockierend), sowie **ds4_wait_for_connection()** (blockierend) abgefragt werden. Letztere blockiert den aufrufenden Task, bis eine Verbindung besteht. Diese Funktion wird z.B. von der Regelschleife der Fahrzeugsteuerung verwendet (vgl. Kapitel 3.6) und basiert auf den FreeRTOS EventGroups.
- Farbänderungen und Vibrationsbefehle können in Bluepad32 ausschließlich aus dem Haupttask (**bluepad32_task**) heraus gesendet werden. Um auch außerhalb der Plattform-Callbacks Effekte steuern zu können, werden entsprechende Befehle über die Funktion **ds4_rumble()** bzw. **ds4_lightbar_color()** in die **ds4_output_event_queue** eingetragen. Diese Vorgehensweise sorgt dafür, dass die Funktion auch gleichzeitig von beiden Prozessor-Kernen aufgerufen werden kann. Die Verarbeitung der Queue erfolgt dann sequenziell durch den **ds4_output_event_task**, der die Events beim **bluepad32_task** registriert.
- Schließlich kann ein niedriger Batteriestand durch ein rotes Blinken der Lichtleiste am

Gamepad signalisiert werden. Dazu wird der `ds4_low_battery_signal_task` aktiviert, sobald das Event-Bit `DS4_BATTERY_LOW` gesetzt ist. Währenddessen werden andere Farbbefehle verworfen.

3.2.4. Testing

Zum Testen des implementierten Treibers wurden zwei separate Tasks auf dem ESP32 erstellt:

- **Input-Test Task:** Dieser Task wartet zu Beginn jeder Schleife blockierend auf eine bestehende Verbindung mit dem DualShock-Controller. Sobald die Verbindung hergestellt ist, blockiert der Task an der Input-Queue, bis ein Zustandsreport vom Controller eintrifft. Der empfangene Report wird aus der Queue entnommen und auf der Konsole ausgegeben. Anschließend erfolgt eine Wartezeit von 16.6 ms, entsprechend einer Frequenz von 60 Hz.
- **Output-Test Task:** Auch dieser Task wartet zunächst auf eine erfolgreiche Verbindung mit dem Gamepad. Nach dem Verbindungsauflauf wird jeweils ein Vibrations- und ein Farbwechselereignis in die Output-Queue eingetragen. Da dem Vibrationsbefehl eine Dauer von 100 ms zugewiesen wurde, pausiert der Task entsprechend lange.

Mit diesem einfachen Testaufbau konnten alle Kernfunktionen des Treibers erfolgreich validiert werden. Auch das Wiederherstellen der Funktionalität nach einem Verbindungsabbruch konnte durch Aus- und Einschalten des Gamepads demonstriert werden.

Das visuelle Feedback bei niedrigem Batteriestand wurde simuliert, indem der Schwellenwert zur Auslösung des zugehörigen Events künstlich angehoben wurde.

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 [14, S. 53] für die benötigten Motoren nicht ausreicht [15–17], 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 I²C-Protokoll zum Einsatz kommt. Der vorliegende Treiber wurde aus der Portierung eines bereits bestehenden Treibers [18] 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 I²C-Bus) und initialisiert den I²C-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 geschalten 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 3.3 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.

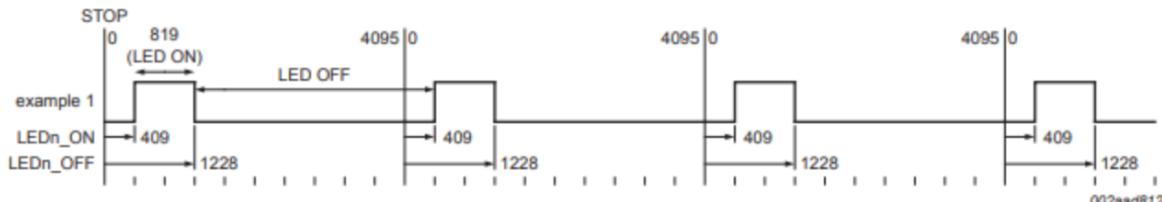


Abbildung 3.3: Erklärung ON/OFF Parameter für PCA9685 aus [19, S. 17]

- **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 [20] 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° (max. Drehung nach links), 0° und 90° (max. Drehung nach rechts) manuell ermittelt. Die absoluten Werte für die Drehungen werden im Folgenden als $value_{-90^\circ}$, $value_{0^\circ}$ und $value_{90^\circ}$ bezeichnet.

Für die Berechnung des Zielwerts $value_{off}$ für den *OFF* Wert des PWM-Board-Treibers aus einem gegebenen Winkel wird Formel 3.1 verwendet.

Sei θ der gewünschte Winkel,

$$|\theta| = \text{Betrag von } \theta$$

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

$$n_2 = |\theta| - n_3 \quad (3.1)$$

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

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

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

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 Sicherheitsmechanismus 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 geclipt. 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 nur unnötig komplex werden. 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 [15].

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.

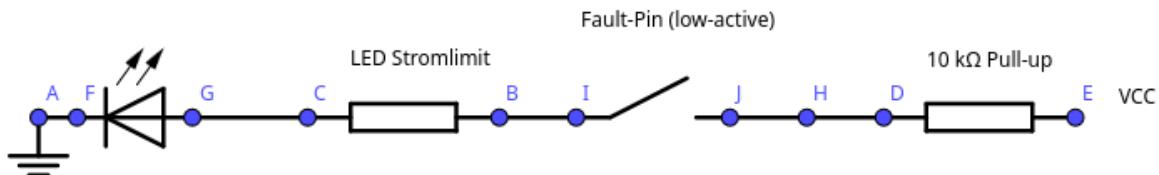


Abbildung 3.4: Testaufbau des Fault-Pins des Pololu-Motor-Treibers

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 $10\text{ k}\Omega$ 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 [21]. 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.

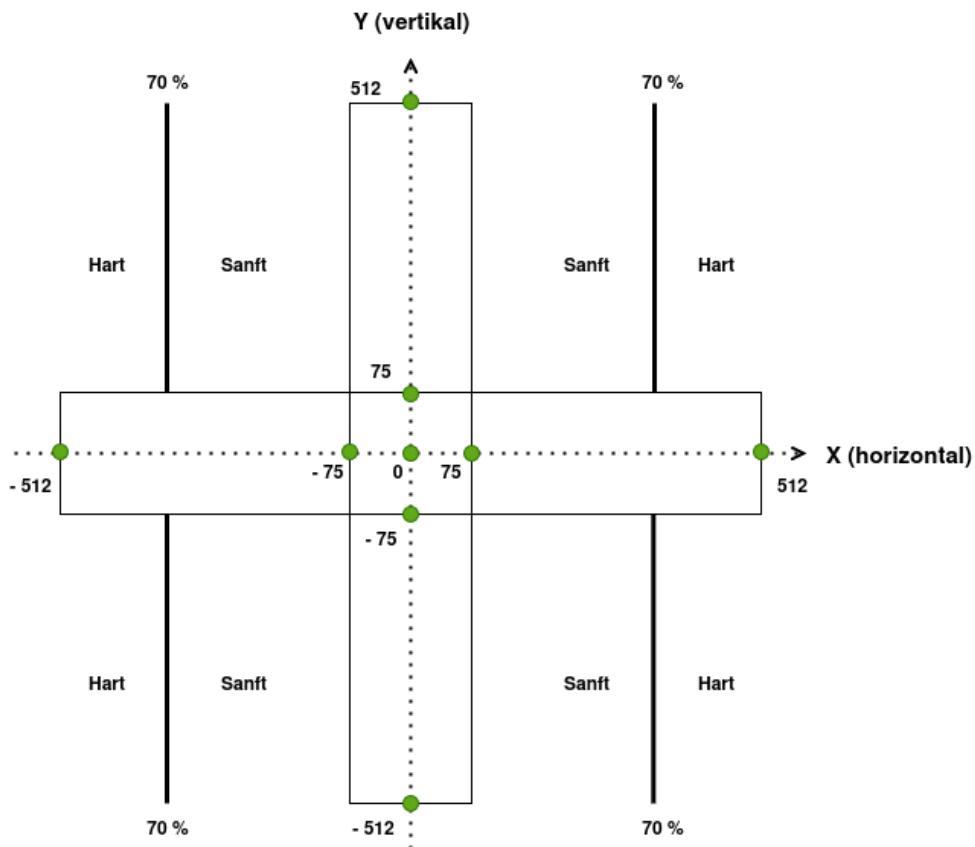


Abbildung 3.5: Differentialantrieb Logik

Im Zentrum der Abbildung 3.5 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 Queue-Werte 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 aufgelösten 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 [22].

Methode 1 nutzt die Hilfefunktion *example_connect*. Diese arbeitet beim Verbindungsauftbau 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. [23]

Ein weiterer Nachteil dieser Variante ist, dass das Setzen der Credentials (SSID und Passwort) normalerweise über die Konsole mittels *idf.py menuconfig* erfolgt [22]. 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 [22] konnte eine Beispielimplementierung entnommen werden, die die grundlegende Funktionalität des Verbindungsauftbaus 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 [24]. Das Framework bietet neben TCP auch eine Reihe verschlüsselter Übertragungswege, wie beispielsweise TSL. 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 [25]. 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 Abschnitt 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. Regelschleife (Becker)

Die Grundlage der Fahrzeugsteuerung bildet eine Regelschleife, welche synchron zur Update-Frequenz des DualShock4-Treibers mit 60 Hz ausgeführt wird. Zunächst wird in dieser Schleife gewartet, bis eine Verbindung zum Controller hergestellt wurde. Im Anschluss daran wird der aktuelle Status des Controllers aus der entsprechenden Queue abgerufen (vgl. Abbildung 3.2). Abhängig vom aktuellen Modus, also dem Zustand, in dem sich das Fahrzeug befindet, werden mit diesen Daten verschiedene Berechnungen durchgeführt. Das Fahrzeug verfügt über zwei Modi:

- Im **manuellen Modus** erfolgt die Steuerung des Fahrzeugs, der Plattform sowie die Schussabgabe manuell über den DualShock4-Controller.
- Im **semi-automatischer Modus** wird die Steuerung der Plattform durch ein KI-Modell übernommen.

Der aktuelle Modus ist anhand der Lichtleiste des Controllers erkennbar: Im manuellen Modus leuchtet die Lichtleiste in einem hellen Grün, während sie im semi-automatischen Modus in einem orangen Farnton leuchtet. Der Modus kann durch gleichzeitiges Halten der oberen Pfeiltaste und der Kreuztaste für 1.5 Sekunden gewechselt werden, beim Moduswechsel vibriert der Controller für einen kurzen Moment.

3.6.2. Plattformsteuerung & Schussabgabe (Becker)

Die Plattform wird im manuellen Modus anhand der Eingabe des Joysticks gesteuert, wobei zuerst die digitalisierten Werte des analogen Controller-Sticks in Winkel umgerechnet werden müssen. Dafür werden folgende Schritte ausgeführt:

1. Deadzone-Filter

Um kleine Bewegungen des Joysticks, die durch Stick-Drift entstehen, zu ignorieren, wird ein Deadzone-Filter angewendet. Wenn der Wert des Controller-Sticks kleiner als ein bestimmter Schwellenwert ist, wird der Wert auf null gesetzt, um ungewollte Eingaben zu vermeiden.

$$\text{if } (|\text{stick}| < \text{deadzone}) \text{ then stick} = 0$$

2. Normalisierung

Die vorliegenden Digitalwerte des Sticks werden nun vom Intervall $]-512, 512[$ auf das Intervall von $]-1, 1[$ normalisiert.

$$\text{normalized_stick} = \frac{\text{stick}}{512}$$

3. Low-Pass-Filter zur Glättung der Eingabe

Ein Low-Pass-Filter wird verwendet, um schnelle Schwankungen zu glätten und sanfte Bewegungen zu erzeugen. Dieser wurde hinzugefügt, da es bei schneller Drehung durch heftiges Bewegen des Sticks häufig zu zitterigen Drehbewegungen kam. Der Filter berechnet einen geglätteten Wert basierend auf der aktuellen Eingabe sowie dem vorherigen Wert, wodurch schnelle Änderungen abgeflacht werden.

$$\text{filtered_stick} = \alpha \cdot \text{normalized_stick} + (1 - \alpha) \cdot \text{filtered_stick_previous}$$

Da $\alpha = 0.2$ beträgt, fließt die aktuelle Eingabe lediglich zu 20 Prozent in die Berechnung ein, während der vorherige Wert zu 80 Prozent berücksichtigt wird. Dieser Wert wurde manuell gewählt, da hier zitterige Drehungen am besten verhindert werden.

4. Berechnung der Geschwindigkeit

Die Geschwindigkeit der Plattform wird durch die gefilterte Eingabe sowie den maximalen Drehwinkel pro Sekunde berechnet. So wird sichergestellt, dass die Plattform lediglich mit einer Geschwindigkeit rotiert, die von den Servos verarbeitet werden kann. Der Wert wurde anhand des Datenblatts [17] festgelegt und anschließend an beiden Plattformachsen reduziert, um eine übermäßige Empfindlichkeit gegenüber Controller-Eingaben zu vermeiden.

$$\text{speed} = \text{filtered_stick} \cdot \text{max_deg_per_sec}$$

5. Berechnung des neuen Winkels

Schließlich wird der Winkel der Plattform auf Basis der berechneten Geschwindigkeit

und der Zeitdifferenz (dt , der Abstand zwischen den Regelzyklen) angepasst.

$$\text{platform_angle} = \text{platform_angle} \pm (\text{speed} \cdot dt)$$

Ob Addition oder Subtraktion verwendet wird, basiert hierbei auf der Verbaurichtung des Servomotors.

Durch diese Berechnungen lässt sich die Plattform nun über die vertikale Achse des rechten Sticks des DualShock4-Controllers drehen und über die horizontale Achse neigen. Bedauerlicherweise führen die Vibrationen des Geschützaufbaus, insbesondere bei längerer und schnellerer Drehung, weiterhin zu ruckartigen Bewegungen der Motoren. Es konnte beobachtet werden, dass sich das Verhalten durch den Einsatz eines Low-Pass-Filters marginal verbessert hat. Aufgrund von Zeitmangel konnten keine weiteren Lösungsansätze mehr in Betracht gezogen werden. Ein möglicher Ansatz wäre beispielsweise der Wechsel der Servomotoren zu digitalen Motoren, die ein genaueres Feedback liefern, oder der Einbau einer PID-Regelung. Zusätzlich wird, sobald der Nutzer die Enden der Drehwinkel erreicht hat, haptisches Feedback in Form von Vibration des Controllers an den Nutzer weitergegeben. Durch das Halten der Kreistaste ist es im manuellen Modus ebenfalls möglich, sich zurück auf die Nullposition beider Achsen zu bewegen.

Die Schussabgabe erfolgt über den rechten unteren Trigger des Controllers, analog zur Steuerung in bekannten Shootern. Da der Treiber die Position des Triggers als 10-Bit-Wert liefert, wurde ein Threshold-Wert von 800 gewählt, um wirklich nur gewollten Druck zu registrieren und nicht versehentlich Schüsse abzufeuern.

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 mit jedem Abgreifen der Werte eine Änderung im Task detektiert. 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 3.5 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)

Um die MQTT-Komponente in der Fahrzeugsteuerung nutzen zu können, waren Code-seitig nur wenige Anpassungen notwendig. Die Regelschleife, vgl. Abschnitt 3.6.1, wurde so erweitert, dass eine Unterscheidung des Modus, wie bereits durch den Kollegen Becker beschrieben, vorgenommen wird. Zu Beginn des Projekts wurde der manuelle Modus priorisiert, weshalb in diesem Integrationsschritt Änderungen im Code vorgenommen mussten, um auch den semi-automatischen Modus vollumfänglich nutzen zu können.

Beim Kompilieren des Codes kam daraufhin eine Meldung, dass die Größe des erzeugten Binarys (1,254,496 bytes) den verfügbaren Speicher (1,048,576 bytes - 1MB) überschreitet. Der erste Eindruck war, dass dies vorrangig auf die Vielzahl an Komponenten zurückzuführen, die im Projekt implementiert wurden, zurückzuführen ist.

```

Error: app partition is too small for binary HimmelWachtEsp32.bin size 0x132460:
- Part 'factory' 0/0 @ 0x10000 size 0x100000 (overflow 0x32460)
ninja: build stopped: subcommand failed.
Running ninja in directory /home/xgsngxguy/workspace/dt_g1_himmel_wacht_sose2025/esp
Executing "ninja size"...
[0/1] cd /home/xgsngxguy/workspace/dt_g1_himmel_wacht_sose2025/esp/HimmelWachtEsp32/
;-m;esp_idf_size" -D MAP_FILE=/home/xgsngxguy/workspace/dt_g1_himmel_wacht_sose2025/
cht_sose2025/esp/esp-idf-v5.4.1/tools/cmake/run_size_tool.cmake
Memory Type Usage Summary

```

Memory Type/Section	Used [bytes]	Used [%]	Remain [bytes]	Total [bytes]
Flash Code	905600	27.1	2436704	3342304
.text	905600	27.1		
Flash Data	201872	4.81	3992400	4194272
.rodata	201616	4.81		
.appdesc	256	0.01		
IRAM	123159	93.96	7913	131072
.text	122131	93.18		
.vectors	1027	0.78		
DRAM	82168	65.96	42412	124580
.bss	58448	46.92		
.data	23720	19.04		
RTC FAST	28	0.34	8164	8192
.force_fast	28	0.34		
RTC SLOW	24	0.29	8168	8192
rtc_slow_reserved	24	0.29		
Total image size: 1254378 bytes (.bin may be padded larger)				

Abbildung 3.6: Speicherauslastung vor der Optimierung

Die Ausgabe der Speicherauslastung, wie in Abbildung 3.6 zu sehen, erfolgt über die Anweisung `idf.py size`. Sie zeigt zwar zum einen den Fehler an sich, jedoch auch, dass noch etwas Speicher verfügbar ist. Nach eingehender Analyse konnte damit das Problem isoliert werden. Grund war nicht, dass prinzipiell zu wenig Speicher zur Verfügung stand, sondern dass die Konfiguration der Partitionierung des Flash-Speichers im ESP32 nicht optimal war. Im ESP ist standardmäßig eine Partitionierung mit 1MB Flash-Speicher für das Binary vorgesehen. Diese Partitionierung ist für die meisten Anwendungsfälle ausreichend, jedoch nicht für dieses Projekt. Die Partitionierung kann jedoch über den Befehl `idf.py menuconfig` und dem sich öffnenden Konfigurationsassistenten angepasst werden.

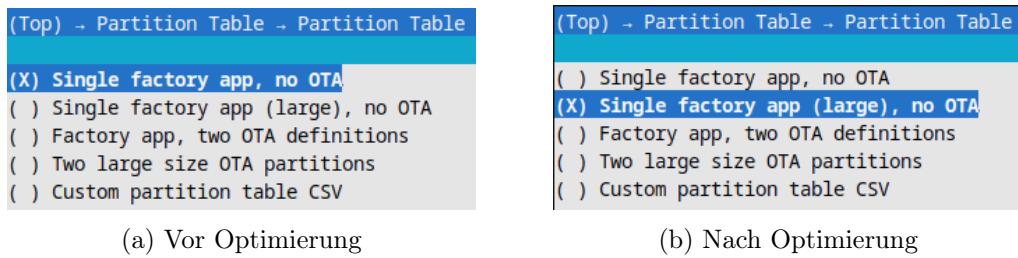


Abbildung 3.7: Partitionierung des Flash-Speichers vor und nach der Optimierung

Abbildung 3.7 zeigt die Partitionierung des Flash-Speichers vor und nach der Optimierung. OTA steht dabei für Over-the-Air Updates. Diese wurden in diesem Projekt nicht berücksichtigt, weshalb kein Speicher dafür reserviert werden musste.

Das Problem konnte letztendlich dadurch behoben werden, dass anstatt der normalen App-Größe nun die *Large*-Variante verwendet wurde. In dieser Konfiguration werden ca. 1,5 MB für die App zur Verfügung gestellt. Die Auslastung des Speichers sah danach identisch wie in Abbildung 3.6 aus, jedoch wurde der Fehler nicht mehr angezeigt. Insgesamt benötigt die Binary nun 82 % des verfügbaren Speichers, was für den ESP32 eine akzeptable Auslastung in Anbetracht der implementierten Komponenten darstellt.

```
HimmelWachtEsp32.bin binary size 0x132460 bytes. Smallest app partition is 0x177000 bytes. 0x44ba0 bytes (18%) free.
```

Abbildung 3.8: Freier Speicher nach der Optimierung

Die Initialisierung des WiFi-Stacks und des MQTT-Stacks sind in der main-Funktion des Programms implementiert. Dabei wird der WiFi-Stack zuerst initialisiert, um eine Verbindung zum WLAN-Netzwerk herzustellen. Anschließend wird der MQTT-Stack mit allen notwendigen Parametern, wie beispielsweise der IP-Adresse des MQTT-Brokers, geladen. Auch dieser Integrationsschritt wurde durch paralleles Logging in der Konssole begleitet. Die Verbindungsversuche zum Netzwerk verliefen durchwegs positiv. Ausgenommen von Spitzen der Latenz, die durch die Übertragung der Daten bei mehreren Teilnehmern über das WLAN-Netzwerk verursacht wurden. Sollte es zu einem Verbindungsabbruch kommen, soll in erster Linie versucht werden, die Verbindung automatisch wiederherzustellen. Dies gelang meistens auch, jedoch kam es in seltenen Fällen zu einem Timeout, der nicht automatisch behoben werden

konnte. In diesem Fall wurde der ESP32 neu gestartet.

Erste Armbewegungen konnten ohne Fahrzeug durchgeführt werden. Dabei wurde abermals auf *mosquitto* als MQTT-Broker auf dem Arbeitsrechner zurückgegriffen. Zu diesem Zeitpunkt war die Implementierung des Objekttrackings noch nicht abgeschlossen, weshalb die Nachrichten noch manuell gesendet werden mussten. Feste Winkelpositionen konnten nach manueller Anpassung erfolgreich an den Topic *vehicle/control* gesendet werden. Die Fahrzeugsteuerung konnte die Nachrichten korrekt empfangen und die Plattform entsprechend bewegen. Diese Integration ermöglichte nun für den weiteren Verlauf des Projekts, dass zukünftig die KI-Komponente die Plattform steuert. Die konkrete Umsetzung des Objekttrackings wird im Abschnitt 7.3 beschrieben.

4. Raspberry Pi Programmierung

4.1. Einführung (Becker)

Der verbaute Raspberry Pi 5 mit 8 GB RAM war ursprünglich dafür vorgesehen, das KI-Modell zur Verfolgung des Holzfliegers lokal auszuführen. Um eine Drosselung der CPU-Taktfrequenz infolge von Überhitzung (Thermal Throttling) zu vermeiden, wurde ein aktiver Luftkühler installiert.

Im weiteren Projektverlauf zeigte sich jedoch, dass die Rechenleistung des Raspberry Pi nicht ausreichte, um das KI-Modell mit der erforderlichen Verarbeitungsgeschwindigkeit auszuführen (siehe Kapitel 5.6.3). Aus diesem Grund wirkt die derzeit auf dem Mikrocontroller implementierte Funktionalität im Vergleich zur ursprünglich geplanten Systemarchitektur für diese Hardware unterdimensioniert.

4.2. Lautsprecher (Becker)

Ziel war es, mittels Lautsprechern akustisches Feedback bei bestimmten Ereignissen wie dem Systemstart, der Schussabgabe sowie dem erfolgreichen Erkennen des Holzfliegers zu geben, um den Gamification-Aspekt des Projekts zu stärken. Hierzu wurden 3-Watt Miniurlautsprecher verwendet, die über MAX98357A-Verstärkerboards angesteuert werden sollten.

Zunächst wurde die grundlegende Funktionalität mit einem einzelnen Lautsprecher getestet. Gemäß eines online verfügbaren Leitfadens zur Ansteuerung über den Raspberry Pi [26] wurde das I²S-Protokoll (Inter-IC Sound) aktiviert. I²S ist ein digitales Audioprotokoll, welches ähnlich wie I²C ursprünglich von Philips entwickelt wurde und der seriellen Übertragung von Audiodaten dient. Nach erfolgreicher Verdrahtung von Mikrocontroller, Verstärkerboard und Lautsprecher auf einem Breadboard, sowie der Auswahl des richtigen Audioausgangs am Raspberry Pi konnte die Tonausgabe erfolgreich realisiert werden. Alle drei bestellten Verstärkerboards und zwei bereits vorhandene Lautsprecher wurden daraufhin auf ihre Funktionsfähigkeit geprüft. Dabei fiel auf, dass eines der Verstärkerboards keinen Ton ausgab und sich ungewöhnlich stark erhitzte, es wurde daraufhin aus dem weiteren Projekt ausgeschlossen. In einem weiteren Schritt wurde basierend auf demselben Leitfaden die Stereo-Konfiguration getestet. Das MAX98357A-Verstärkerboard verfügt hierfür über einen sogenannten Shutdown-Pin (*SD*), über diesen lässt sich der gewünschte Audiokanal auswählen, indem eine entsprechende Spannung an den Pin gelegt wird. Standardmäßig ergibt sich durch die interne Be- schaltung eines 1 MΩ Widerstandes zwischen *SD* und *V_{in}* und eines 100 kΩ Widerstandes zwischen *SD* und *GND* eine gemischte Ausgabe beider Kanäle (jeweils 50% des linken und

rechten Audiokanals). Möchte man stattdessen nur einen Kanal ausgeben, müssen die Widerstandswerte angepasst werden.

Da in diesem Projekt nicht die im Leitfaden verwendeten originalen Adafruit-Boards, sondern günstigere Alternativen zum Einsatz kamen, wurde zunächst geprüft, ob die interne Beschaltung dieser Module identisch ist. Obwohl kein offizielles Datenblatt vorlag, konnte eine schematische Darstellung in den Artikelbildern gefunden werden, die denselben Aufbau bestätigte. Zusätzlich waren dort die Zielwiderstände für beide Kanäle angegeben [27]:

- Um den linken Kanal anzusprechen, musste SD direkt mit V_{in} verbunden werden.
- Für den rechten Kanal war es laut Schaltplan erforderlich, den ursprünglichen $1 \text{ M}\Omega$ Widerstand durch einen effektiven Widerstand von $370 \text{ k}\Omega$ zu ersetzen. Da die internen Widerstände nicht entfernbar waren, wurde eine Parallelschaltung mit einem externen Widerstand nach folgender Berechnung vorgenommen:

$$\frac{1}{370\text{k}\Omega} = \frac{1}{1\text{M}\Omega} + \frac{1}{R_2} \Rightarrow R_2 \approx 587\text{k}\Omega$$

Durch Kombination mehrerer Widerstände in Serie ist ein Gesamtwiderstand von etwa $590 \text{ k}\Omega$ entstanden, welcher zwischen SD und V_{in} geschaltet wurde. Der tatsächliche Widerstandswert ist mit einem Multimeter validiert worden.

Trotz korrektem Aufbau wurde in dieser Konfiguration jedoch kein Ton ausgegeben. Da zu diesem Zeitpunkt der mechanische Zusammenbau der Plattform Vorrang hatte, wurde die Fehlersuche zurückgestellt.

Im weiteren Projektverlauf stellte sich heraus, dass der ESP32-Chip nicht zum vorhandenen Breakoutboard passte. Herr Altmann stellte daraufhin ein alternatives, allerdings deutlich größeres Board zur Verfügung. Aufgrund des begrenzten Bauraums ließ sich jetzt nur noch ein einzelner Lautsprecher montieren, der zudem teilweise über den Rand der Plattform hinausragte. Für diesen Anwendungsfall ist eine passende Halterung entworfen worden (vgl. Kapitel 2.12).

Beim erneuten Test des ursprünglich funktionierenden Mono-Setups (SD in Standardkonfiguration) wurde nun jedoch kein Ton mehr ausgegeben. Beide verbliebenen, funktionstüchtigen Verstärkerboards sind getestet worden, jedoch ohne Erfolg.

Da das Projektteam den Robotor ohne Lautsprecher auf der Plattform optisch ansprechender fand und das Fehlen des Lautsprechers keine grundlegenden Funktionen einschränkt, wurde sich darauf geeinigt, auf die Integration des Lautsprechersystems zu verzichten.

4.3. 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 $LSB/\text{°}/\text{s}$ 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 ($\text{°}/\text{s}$) errechnet. Hierbei beträgt der Teiler standardmäßig 131. [28, 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° bis $+180^\circ$ liefert. [29]

Die Winkelberechnung erfolgt nach der Formel 4.1:

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

Hierbei sind a_x , a_y und a_z die normierten Beschleunigungswerte in g (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° vermieden werden. [29]

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 4.3.1 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 [28, 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 8 anzuzeigen.

4.3.1. 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 [30]. Die Berechnung des Kalman-Filters erfolgt nun mittels der Werte des Gyroskops, einer Zeitdifferenz und der zuvor berechneten Roll-und Pitchwinkel. Die Neigungswinkel liefern eine absolute Orientierung relativ zur Erdgravitation, sind jedoch anfällig gegenüber Rauschen und dynamischen Bewegungen, da sie auf Momentanwerten basieren, weshalb man hier auf die Gyroskopwerte zurückgreift. Diese liefern die Winkelgeschwindigkeit, also die Änderungsrate der Orientierung. Diese Werte werden über die Zeit integriert, um eine relative Winkelschätzung zu erhalten. Sie zeichnen sich durch hohe Kurzzeitstabilität und geringe Reaktionsverzögerung aus, unterliegen jedoch einem Driftverhalten aufgrund von Messabweichungen und systematischen Fehlern. Der Kalmanfilter fusioniert nun diese beiden Quellen und dessen Vorteile in Relation zum letzten bekannten Zustand.

Dabei besteht der Kalman-Filter aus zwei Hauptphasen: dem Vorhersageschritt (Prediction) und dem Aktualisierungsschritt (Update). Im Vorhersageschritt wird der neue Winkel $\hat{\theta}_{\text{new}}$ basierend auf dem vorherigen Winkel $\hat{\theta}_{\text{prev}}$ und der aktuellen korrigierten Gyroskoprate ω geschätzt:

$$\omega = \text{newGyroRate} - \text{bias}, \quad \hat{\theta}_{\text{new}} = \hat{\theta}_{\text{prev}} + dt \cdot \omega$$

Zusätzlich wird die Unsicherheit dieser Vorhersage, dargestellt durch die Kovarianzmatrix P , angepasst. Dabei werden sowohl das Zeitintervall dt als auch Modellunsicherheiten, wie der Drift des Gyroskops, berücksichtigt.

Im Aktualisierungsschritt wird der berechnete Winkel θ_{meas} , welcher mithilfe der Beschleunigungswerte berechnet wurde, mit der Vorhersage verglichen. Die Differenz

$$S = \theta_{\text{meas}} - \hat{\theta}_{\text{prior}}$$

wird als *Innovation* bezeichnet und wird benötigt um den Kalman-Gain K zu bestimmen. Der Kalman-Gain bestimmt, wie stark diese neue Information zur Korrektur verwendet wird. Er wird aus dem Verhältnis von Unsicherheits-Kovarianzmatrix P und der Innovation S berechnet:

$$K_0 = \frac{P_{00}}{S}, \quad K_1 = \frac{P_{10}}{S}$$

Hierbei steht P_{00} für die Unsicherheit in der Winkelschätzung, und P_{10} beschreibt die Kovarianz zwischen Winkel und Bias des Gyroskops.

Ein höherer Wert von K bedeutet, dass der Filter der neuen Messung mehr vertraut und ein niedriger Wert zeigt, dass die eigene Vorhersage als zuverlässiger eingeschätzt wird.

Abschließend wird die Kovarianzmatrix P angepasst, um die reduzierte Unsicherheit nach der Messung zu reflektieren. Dadurch wird der Filter präziser in seiner nächsten Schätzung [31]. Insgesamt erlaubt dieser Algorithmus eine robuste und gleitende Fusion der Sensordaten, wobei kurzfristige Genauigkeit des Gyroskops und langfristige Stabilität des Beschleunigungssensors optimal kombiniert werden.

In der Abbildung 4.1 und 4.2 ist sowohl für den Pitch, als auch den Roll ein signifikanter Unterschied zwischen den Rohdaten und den gefilterten Daten zu erkennen. Die ersten Schwankungen der gefilterten Werte sind vermutlich genau darauf zurückzuführen, dass der Kalmanfilter mit wenigen Schätzungen noch keinen stabilen Werte für die Kovarianzmatrix und den Bias berechnen konnte, 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.

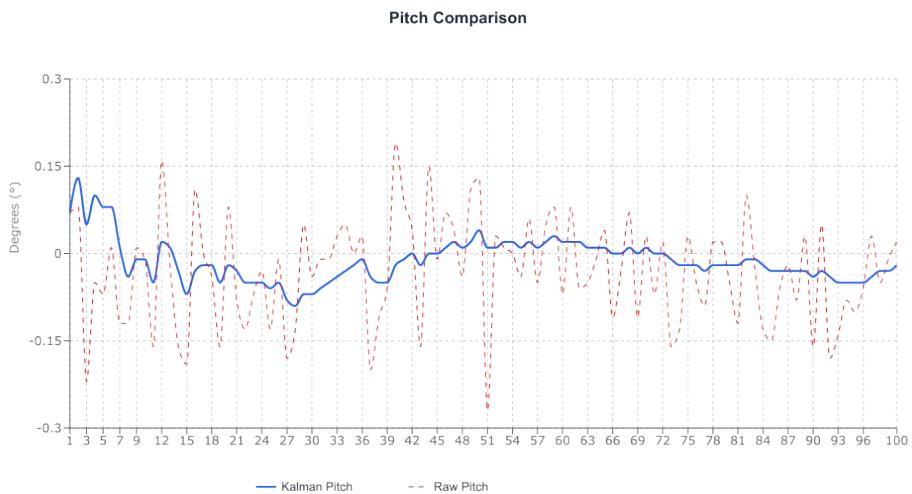


Abbildung 4.1: Pitch: Vergleich Rohdaten und Kalman-Filter

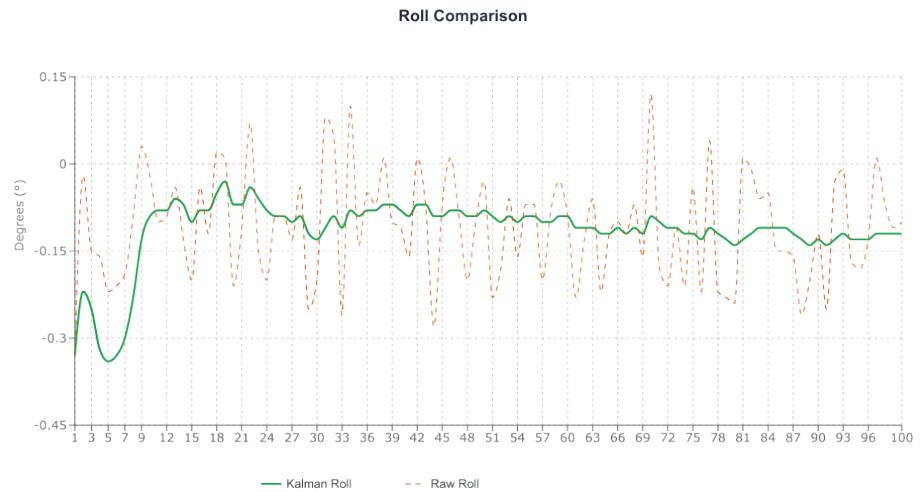


Abbildung 4.2: Roll: Vergleich Rohdaten und Kalman-Filter

5. Erstellung & Ausführung eines KI-Modells zur Objektdetektion

5.1. Anforderungen an das KI-Modell (Jürgens)

Ziel dieser Teilaufgabe war die Entwicklung eines KI-Modells zur automatischen Erkennung eines Holzfliegers als Zielobjekt. Eine zentrale Anforderung bestand darin, das Modell lokal auf einem Raspberry Pi 5 zu implementieren und auch dort auszuführen. Aufgrund der fehlenden dedizierten Grafikkarte des Systems war ein leichtgewichtiges, CPU-optimiertes Modell erforderlich. Der verwendete Raspberry Pi 5 verfügt über einen 64-Bit Quad-Core ARM Cortex-A76 Prozessor und 8 GB Arbeitsspeicher. [32] Aufgrund dieser Hardwarebeschränkung und persönlicher Erfahrungen wurde entschieden, dass ein YOLO Modell des Unternehmens Ultralytics verwendet wird. Dieses hat sich auf state-of-the-art KI-Modellen der YOLO-Familie im Bereich der Objekterkennung und Bildsegmentierung spezialisiert. [33] Die Modelle von Ultralytics zeichnen sich durch eine hohe Präzision und schnelle Inferenzzeiten aus. Diese Eigenschaften sind für die lokale Ausführung auf dem Raspberry Pi 5 von großer Bedeutung, da hier die Inferenz auf der CPU durchgeführt werden muss. Besonders interessant sind die Open-Source Implementierungen der YOLOv8 und YOLOv11 Modelle. Diese sind bereits vortrainiert und haben bestehende Gewichte und bieten somit die Möglichkeit direkt verwendet oder auf einem eigenen Datensatz weitertrainiert zu werden. Ein weiterer Vorteil der YOLO Modelle und Ultralytics liegt in der sehr guten Dokumentation und einer geringen Einstieghürde. Mit der Ultralytics Python Bibliothek können Modelle einfach erstellt, trainiert und evaluiert werden. Dies bildet die Grundlage für die Entwicklung und Verwendung des Modells in diesem Projekt.

5.2. Modellauswahl (Jürgens)

Die YOLOv8 und YOLOv11 Modelle von Ultralytics werden in verschiedenen Größen angeboten. Mit steigender Größe des Modells steigt zwar die Genauigkeit des Modells, aber auch die Inferenzzeit und der Speicherbedarf. Die Modelle werden in folgenden Größen angeboten:

- YOLOv8/11n - Nano
- YOLOv8/11s - Small
- YOLOv8/11m - Medium
- YOLOv8/11l - Large

- YOLOv8/11x - Extra Large

Die Modellauswahl ist durch die Hardware des Raspberry Pi 5 begrenzt. So sind insbesondere die mittleren und großen Modelle aufgrund der hohen Inferenzzeiten nicht für die Ausführung auf der CPU des Raspberry Pi 5 geeignet. Bereits das YOLOv8m Modell hat eine Inferenzzeit von 234,7 ms. [34] Anhand dessen wurde entschieden, dass eine Beschränkung auf die kleineren Modelle YOLOv8n, YOLOv8s, YOLO11n und YOLO11s sinnvoll ist. Diese Modelle haben eine geringere Inferenzzeit und könnten somit besser für die Ausführung auf dem Raspberry Pi 5 geeignet sein. Die folgenden Leistungsmetriken stammen aus der offiziellen Online-Dokumentation von Ultralytics [[34–36]]:

- YOLOv8n \approx 80.4 ms
- YOLOv8s \approx 128.4 ms
- YOLO11n \approx 56.1 ms
- YOLO11s \approx 90.0 ms

Anhand dieser Werte und persönlicher Erfahrungen wurde zunächst entschieden, dass das YOLOv8n Modell im ONNX-Format verwendet wird. Dieses Modell hat mit einer Inferenzzeit von 80,4 ms eine vertretbare Geschwindigkeit.

5.2.1. Das ONNX-Format (Jürgens)

ONNX steht für Open Neural Network Exchange und stellt ein Open-Source-Format für KI-Modelle bereit. Es ermöglicht KI-Modelle über verschiedene Frameworks und Plattformen hinweg zu verwenden. Modelle können beispielweise in einem Framework mit PyTorch trainiert und anschließend in das ONNX-Format exportiert werden. Oftmals ermöglicht die Verwendung des ONNX-Formats eine bessere Performance von KI-Modellen auf verschiedenen Hardwareplattformen durch optimierte Inferenzmaschinen. [37]

Der Ansatz für dieses Projekt war es, ein Modell auf einer leistungsstarken Maschine mit dicker Grafikkarte zu trainieren und dieses anschließend in das ONNX-Format zu exportieren. Die Python Bibliothek von Ultralytics bietet unter Anderem die Möglichkeit ein Modell zu trainieren und anschließend in das ONNX-Format zu exportieren. Nach dem Training hat das Modell zunächst das PyTorch-Format mit der Dateiendung .pt. Nach dem Export in das ONNX-Format hat das Modell die Dateiendung .onnx. Dieses Format kann beispielsweise mit Tools wie NVIDIA TensorRT oder Intels OpenVINO optimiert werden. Laufzeitumgebungen wie ONNX Runtime sind dabei für Hochleistungsinferenz über verschiedene Hardware optimiert. [37]

5.3. Trainingsdaten & Annotation (Jürgens)

Die Qualität und Performance eines KI-Modells hängt maßgeblich von der Qualität der Trainingsdaten ab. Der erste Schritt ist es, einen qualitativ hochwertigen Datensatz zum Trainieren des Modells zu erstellen. Das Ziel ist es endgültig einen Holzflieger auf einem Bild zu

erkennen. Die Trainingsdaten wurden direkt mit dem Raspberry Camera Module 3 aufgenommen, welches mit einer Auflösung von 11,9 Megapixeln und einer Bittiefe von 24 Bit bei einer Bildgröße von 4608x2592 Pixeln arbeitet. [38] Die Kamera wird neben der Erstellung der Trainingsdaten auch für die Inferenz des Modells im späteren Verlauf verwendet. Um eine robuste Objekterkennung unabhängig von der räumlichen Orientierung des Zielobjekts zu gewährleisten, wurde ein Datensatz von 1.201 Bildern erstellt. Diese umfassen verschiedene Umgebungsbedingungen, Beleuchtungssituationen und Perspektiven, wobei das Zielobjekt in nahezu allen möglichen Neigungen erfasst wurde. Für das überwachte Lernverfahren sind neben den Bilddaten entsprechende Annotationen erforderlich. Der Annotationsprozess erfordert die manuelle Markierung der zu erkennenden Objekte in einem geeigneten Tool sowie deren Konvertierung in ein kompatibles Format. Dieser arbeitsintensive Prozess ist von entscheidender Bedeutung, da die Qualität der Annotationen direkt die Modellperformanz beeinflusst. Vor Beginn des Annotationsprozesses muss zunächst die Art der Annotierung definiert werden. Für diese Arbeit gab es zwei Arten der Annotation, die in Betracht gezogen wurden:

- Bounding Box: Ein Rechteck wird um das zu erkennende Objekt gezeichnet. Diese Methode ist einfach und schnell, aber weniger präzise.
- Polygon: Eine komplexere Form wird um das Objekt gezeichnet. Diese Methode ist genauer, aber auch zeitaufwändiger.

Die Entscheidung fiel auf die zeitaufwändige Polygon-Annotation. Diese ermöglicht eine genauere Erkennung des Holzfliegers weil nicht nur die äußeren Kanten sondern auch die komplexere Form des Objekts berücksichtigt wird. Neben der speziellen Form des Holzfliegers war diese Entscheidung auch durch die relativ geringe Anzahl vorhandener Trainingsbilder begründet. Zur Annotation wurde das Open-Source Tool LabelStudio verwendet. Dieses bietet eine benutzerfreundliche Oberfläche und lässt sich sehr genau personalisieren. Durch die integrierte Projektverwaltung und verschiedener unterstützter Exportformate bietet das Tool eine gute Lösung für die Annotierung. Aufgrund fehlender Implementation der Exportfunktion für größere Datensätze (Export von Bildern und korrespondierenden Annotationen) war eine Nachbereitung des Datensatzes notwendig. Nach Abschluss der Annotation wurden die Labels im YOLO-Format exportiert und eine Ordnerstruktur erstellt, die zum Training des YOLO-Modells verwendet werden kann.

Folgende Ordnerstruktur wurde erstellt:

```

train/
|   images/
|   labels/
|
val/
|   images/
|   labels/
|
test/
|   images/
|   labels/

```

Die Bilder und Labels im Ordner `train` werden für das Training des Modells verwendet, die Bilder und Labels im Ordner `val` für die Validierung des Modells während des Trainings und die Bilder und Labels im Ordner `test` für die abschließende Evaluation des Modells. Die Bilder wurden in den Ordner `images` und die Annotationen in den Ordner `labels` gespeichert. Die Annotationen werden in Textdateien mit der Endung `.txt` gespeichert, wobei jede Textdatei die gleiche Bezeichnung wie das zugehörige Bild hat. Beispielsweise wird das Bild `image1.jpg` im Ordner `images` durch die Textdatei `image1.txt` im Ordner `labels` annotiert.

5.4. Training des Modells (Jürgens)

Ultralytics bietet mit der Python-Bibliothek `ultralytics` eine einfache Möglichkeit, ein YOLO-Modell zu trainieren, zu validieren und zu evaluieren. Vor dem Training wird eine Konfigurationsdatei benötigt, welche die Pfade des Trainingsordners, des Validierungsordners und des Testordner enthält. Neben den Pfaden beinhaltet Sie noch die Anzahl der Klassen und den dazugehörigen Namen. In diesem Fall gibt es nur den Holzflieger als Klasse, welcher zukünftig mit der Bezeichnung 'Plane' gelabelt wird.

Das Training erfolgt auf einer Mittelklasse NVIDIA Grafikkarte (RTX 3070) mit 8 GB Grafikkartenspeicher und wurden der offiziellen NVIDIA Produktseite entnommen.[39] Während des Trainings werden die Trainingsbilder augmentiert, um die Robustheit des Modells zu erhöhen. So werden verschiedene Transformationen auf die Bilder angewendet, wie z.B. Rotation, Skalierung oder Änderung der Helligkeit. Hiermit wird die Varianz der Trainingsbilder künstlich erhöht, was zu einer besseren Performance des Modells führt. [40] Durch das Verwenden von vortrainierten Gewichten wird die Trainingszeit verkürzt und die Performance des Modells verbessert. Diese vortrainierten Gewichte werden vor dem Trainings als `.pt` Datei heruntergeladen und bereitgestellt und bieten den Ausgangspunkt für das Training. Durch Transferlearning, also dem weitertrainieren eines vortrainierten Modells auf einen neuen Datensatz, kann die Performance des Modells weiter verbessert werden. Zudem wird die benötigte Trainingszeit reduziert. [41]

Nach Abschluss des Trainings werden automatisch die besten Gewichte des Modells unter `runs/detect/train/weights/best.pt` gespeichert. Neben den reinen Gewichten erstellt das Trainingsskript unter Anderem die Datei `results.png` in welcher verschiedene Metriken des Trainings, wie zum Beispiel die `mAP50` und `mAP50-95`, visualisiert werden. Die Metrik `mean Average Precision` wie zum Beispiel `mAP50` und `mAP50-95` geben die Genauigkeit des Modells an, wobei `mAP50` die Genauigkeit bei einer IoU-Grenze von 0,5 und `mAP50-95` bei IoU-Werten zwischen 0,5 und 0,95 angibt [MAPGlossary]. IoU steht für Intersection over Union und gibt die prozentuale Schnittmenge zwischen der vorhergesagten Bounding Box und der tatsächlichen Bounding Box an.[42] So kann man sagen, dass ein hoher `mAP50`-Wert und `mAP50-95`-Wert gute Indikatoren sind wie gut das Modell die Objekte erkennt.

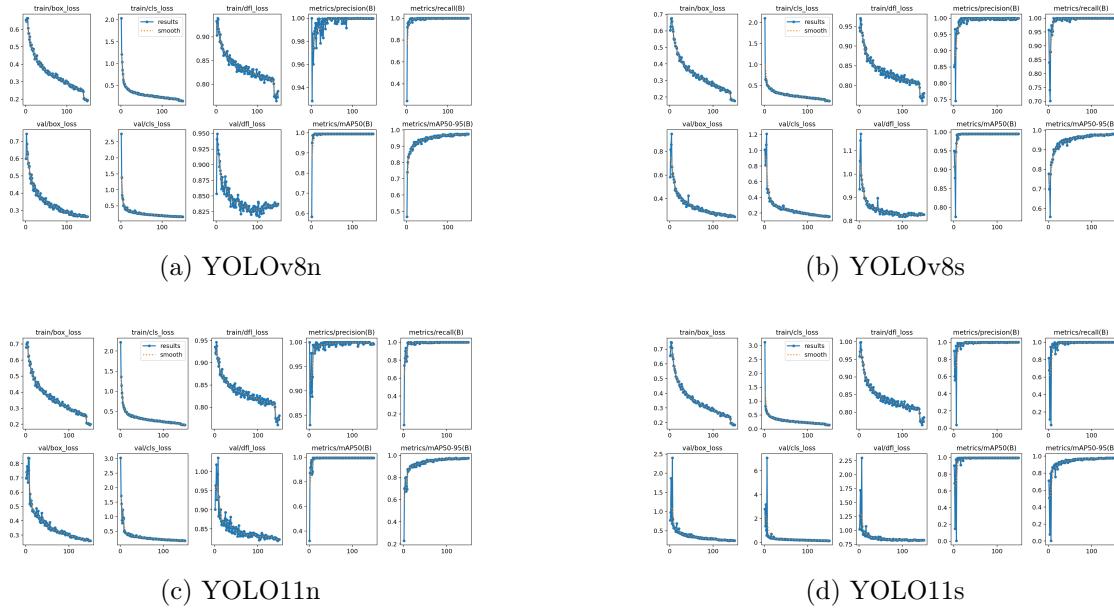


Abbildung 5.1: Vergleich Trainingsresultate YOLOv8n, YOLOv8s, YOLO11n und YOLO11s

Wie in 5.1 zu sehen ist, schneiden alle Modelle nach der Trainingsphase von 150 Epochen ähnlich gut ab. Eine Epoche stellt dabei ein vollständiger Durchlauf durch den gesamten Trainingsdatensatz dar.[43] Anhand dieser Resultate wurde zunächst das YOLOv8n Modell ausgewählt, da es entsprechend der Trainingsmetriken trotz der geringeren Größe im Vergleich zu den s-Modellen eine ähnlich gute Performance aufweist dabei aber eine geringe Inferenzzeit aufweist. Dazu kommen positive persönliche Erfahrungen mit dem YOLOv8n Modell.

Um das Modell schließlich auch als ONNX-Modell zu betreiben muss dieses zunächst in das ONNX-Format exportiert werden. Der Export erfolgt mit der Ultralytics Python-Bibliothek.

5.5. Ausführung des Modells im ONNX-Format (Jürgens)

Um generell ein KI-Modell im ONNX-Format auszuführen, sind einige Voraussetzung an das System zu erfüllen. Es werden verschiedene Programme und Bibliotheken benötigt, um das Modell auszuführen. Um schnelle Ergebnisse und erstes Feedback zu erhalten wie die Inferenz des Modells im ONNX-Format abläuft wurde zunächst eine Laufzeitumgebung auf dem persönlichen Computer eingerichtet. Folgende Bibliotheken und Programme waren auf jeden Fall notwendig, um das Modell auszuführen:

- ONNX Runtime v1.21.0
- OpenCV (mit GStreamer Unterstützung)
- CUDA 12.6 (optional, für GPU-Beschleunigung)
- cuDNN 9.9 (NVIDIA Deep Neural Network library, NVIDIA Developer Account erforderlich)

Zwischen diesen Komponenten gibt es Abhängigkeiten speziell in der Versionierung. So muss beispielsweise die CUDA-Version zur eingesetzten Grafikkarte und die cuDNN-Version zur CUDA-Version passen.

Um den Anforderungen einer minimalen Laufzeit pro Inferenz gerecht zu werden, war die Zielprogrammiersprache C++ und nicht Python. Aufgrund fehlender Erfahrungen mit KI-Modellen im ONNX-Format wurde die Erstellung eines ersten Prototypen mit Hilfe von ChatGPT und der ONNX-Dokumentation gelöst. Es stellte sich heraus, dass die Ergebnisse der Inferenz im ONNX-Format mangelhaft sind, wobei man eine fehlerhafte Implementierung im C/C++ Code nicht ausschließen kann. Im Vergleich zu den Ergebnissen der Inferenz mit der von Ultralytics bereitgestellten Python-Bibliothek kann man die Ergebnisse der Inferenz des C/C++ Prototypen nicht verwenden. Um ein vergleichbares Ergebnis zu produzieren, werden beide Inferenzen mit dem selben Video getestet. Das Video wurde direkt mit dem Raspberry Pi 5 und dem Camera Module 3 aufgenommen.

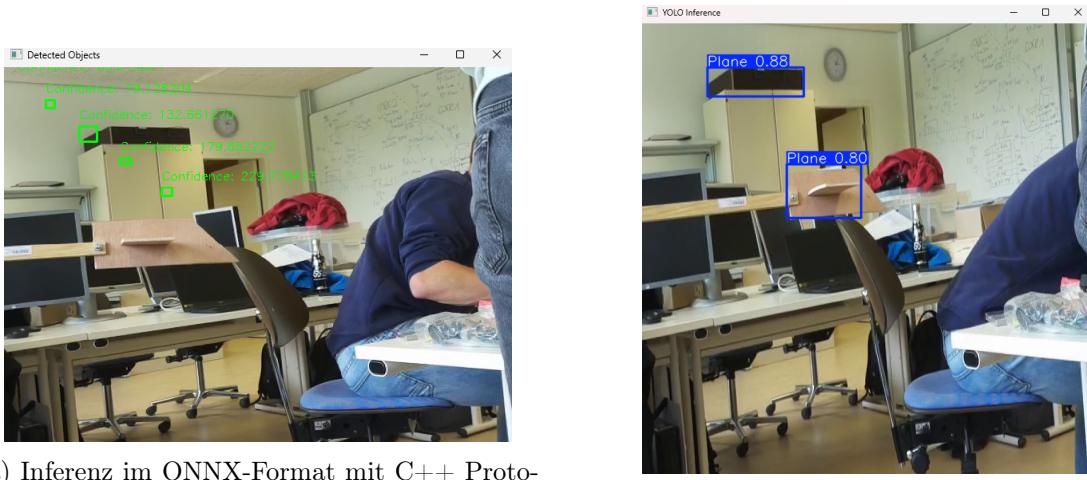


Abbildung 5.2: Vergleich Inferenz ONNX C++ und Python

Wie in 5.2 zu sehen ist, sind die Ergebnisse der Inferenz in Python mit der Ultralytics-Bibliothek deutlich besser. Basierend auf diesen Resultaten wurde entschieden, dass eine weitere Implementierung in C/C++ nicht sinnvoll ist. So fällt besonders die hohe Komplexität, die fehlerhaften Erkennungen, die hohe Einarbeitungszeit hinsichtlich der Projektdauer, aber auch der hohe Aufwand für die Einrichtung einer funktionierenden Laufzeitumgebung ins Gewicht. Mit der Verwendung der Ultralytics Bibliothek fällt ein Großteil dieser Komplexität weg.

5.6. Inferenz des PyTorch-Modells (Jürgens)

5.6.1. Installation der KI-Abhängigkeiten (Jürgens)

Neben der reinen Ultralytics-Bibliothek welche mit einer Reihe weiterer Abhängigkeiten installiert wird, ist besonders die Python-Version von OpenCV von Bedeutung. OpenCV ist größtenteils in C++ geschrieben, bietet aber auch eine Python-Schnittstelle an. Diese soll

dafür verwendet werden, um auf den Kamera-Stream des Raspberry Pi 5 zuzugreifen und bildet die Grundlage für die Inferenz mit dem YOLO-Modell. In der Regel kann ein Stream in OpenCV mit der Funktion `cv2.VideoCapture()` geöffnet werden. Der Stream kann dabei eine angebundene Kamera, eine Videodatei oder auch ein RTSP-Stream sein. Aufgrund fehlender Unterstützung des Kamera-Softwarestacks des Raspberry Pi 5 kann die Kamera nicht direkt mit der Funktion `cv2.VideoCapture()` geöffnet werden **??**. Der Test mit einer USB-Kamera dagegen funktionierte problemlos. Da OpenCV eine essenzielle Abhängigkeit in diesem Projekt ist, wurde ein Workaround gefunden in dem die Kamera des Raspberry Pi 5 mittels GStreamer an OpenCV übergeben wird. Eine reguläre OpenCV-Installation mit dem Python Paketmanager PIP enthält standardmäßig keine GStreamer-Unterstützung. Diese muss explizit aktiviert werden, indem OpenCV aus dem Quellcode mit GStreamer-Unterstützung kompiliert wird. Hierbei ist darauf zu achten, dass nicht nur die GStreamer-Bibliothek korrekt installiert und angegeben wird, sondern auch der Python-Interpreter welcher auf dem Raspberry Pi 5 verwendet wird. Die Kompilierzeit kann dabei mehrere Stunden in Anspruch nehmen und ist abhängig von der Leistung des Systems. So hat die Kompilierung und Installation von OpenCV mit GStreamer-Unterstützung etwa 2 Stunden gedauert.

5.6.2. Erstellung & Evaluierung der GStreamer-Pipeline (Jürgens)

Die Video-Pipeline wird insgesamt in zwei Kommunikationspartner aufgeteilt. Der erste Kommunikationspartner bildet eine libcamera-vid-Instanz, welche per Kommandozeile gestartet wird. Diese Instanz öffnet die Kamera des Raspberry Pi 5 und sendet diesen Stream an eine angegebene IP-Adresse.

Der zweite Kommunikationspartner ist der Empfänger des Kamera-Streams. In diesem Fall stellt das Python-Skript mit der OpenCV Installation inklusive GStreamer-Unterstützung den Empfänger dar. Hier wird das trainierte KI-Modell ausgeführt und auf den Stream angewandt. Bei der Erstellung der Pipelines wird H264 als Video-Codec und UDP als Transportprotokoll gewählt. Mit dieser Kombination soll eine hohe Performance und eine geringe Latenz erreicht werden.

Diese Pipeline überträgt den Kamera-Stream an OpenCV mit einer Latenz von etwa 1 Sekunde. Diese Latenz ist für diese Anwendung inakzeptabel, da die Inferenz des Modells aber auch der Kamerastream so nahe wie möglich an der Echtzeit sein soll.

5.6.3. Inferenz des Modells auf dem Raspberry Pi 5 (Jürgens)

Die Inferenz des Modells erfolgt entsprechend der Dokumentation von Ultralytics. Diese wird in einer Endlosschleife durchgeführt und verarbeitet den Kamera-Stream Bild für Bild. Trotz der relativ guten Hardware des Raspberry Pi 5 (im Vergleich zu älteren Raspberry Pi Systemen) ist die Inferenzzeit und damit Nutzbarkeit dieses Ansatz sehr eingeschränkt. So konnte sehr markantes Ruckeln bei der Anzeige des Kamera-Streams festgestellt werden. Die Ausführung wirkt sich dabei besonders auf die CPU-Last aus. Wie in 5.3a dargestellt wurde die CPU vom Raspberry Pi 5 teilweise über 90% ausgelastet. Dieses Verhalten spiegelt sich bei der Performance der KI-Auswertung wieder. So hatte das System trotz der Nutzung des kleinstmöglichen Modells einen sehr geringen Durchsatz (siehe 5.3b, grüne Linie). Die Berechnung

durch das KI-Modell wurde nach Abstimmung mit Herrn Altmann und dem Team auf einen Laborrechner mit dedizierter Grafikkarte ausgelagert.

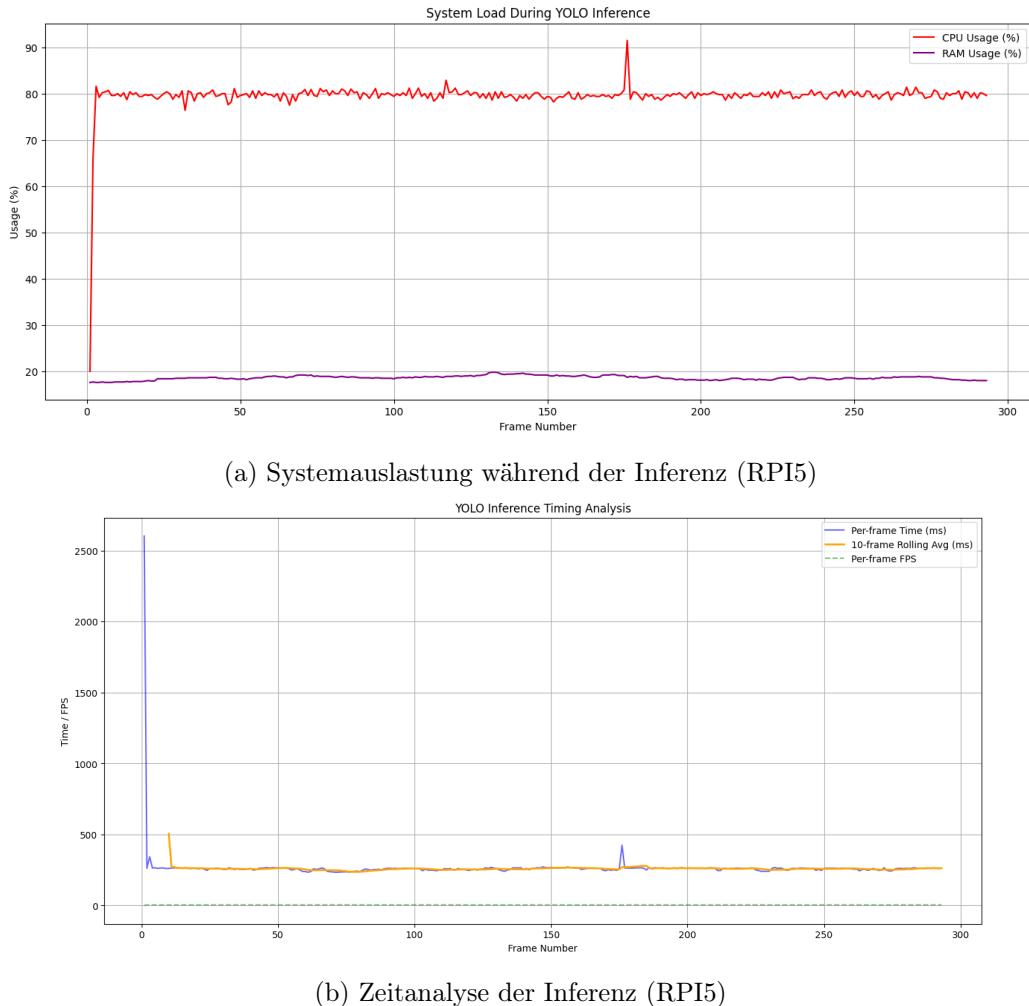


Abbildung 5.3: System- und Zeitanalyse während der Inferenz (RPI5)

5.6.4. Inferenz des Modells auf dem Laborrechner (Jürgens)

Mit dem Plattformwechsel wurde neben der Ausführungsumgebung ebenfalls das verwendete Modell geändert. So wurde das YOLOv8n Modell durch das YOLOv11 Modell ersetzt, welche hinsichtlich der Performance und Genauigkeit eine kleine Verbesserung bietet. Die Inferenz des Modells auf dem Laborrechner erfolgt ebenfalls mit der Ultralytics Python-Bibliothek. Anders als auf dem RPI5 läuft die Berechnung des Modells auf der dedizierten Grafikkarte des Laborrechners. Hierbei handelt es sich um eine NVIDIA T1000 Grafikkarte welche die Nutzung von CUDA ermöglicht. Um dieses auch aktiv zu verwenden muss der **NVIDIA Cuda Compiler Driver** und PyTorch mit CUDA-Unterstützung installiert sein. Die Inferenzzeit des Modells auf dem Laborrechner fällt mit etwa 8 ms deutlich geringer aus und bietet eine flüssigere Anzeige des annotierten Kamera-Streams (siehe 5.4).

```

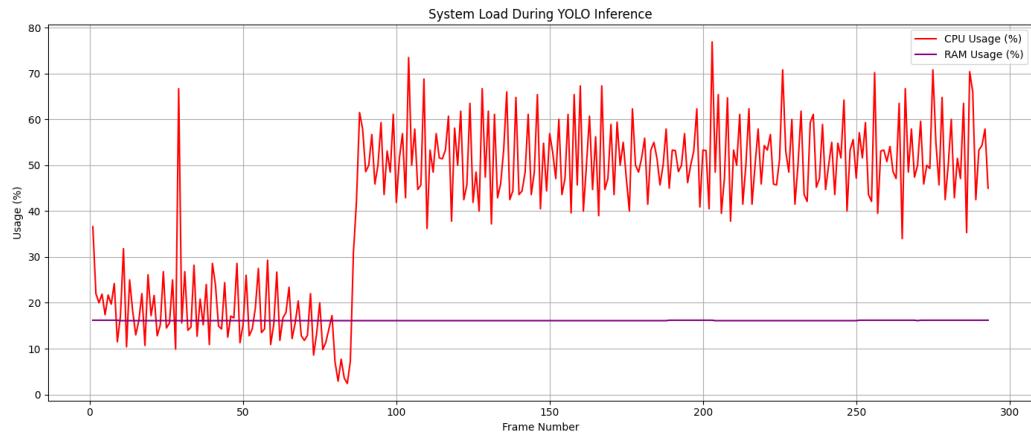
==== Timing Summary - CUDA ====
Processed 293 frames
Total elapsed time (real-time): 8.22s
Total inference time: 5.27s
Overhead time (display, I/O, etc.): 2.95s
Average FPS (inference only): 55.61
Average FPS (real-time): 35.65

==== Timing Summary - RPI5 ====
Processed 293 frames
Total elapsed time (real-time): 77.55s
Total inference time: 76.87s
Overhead time (display, I/O, etc.): 0.68s
Average FPS (inference only): 3.81
Average FPS (real-time): 3.78

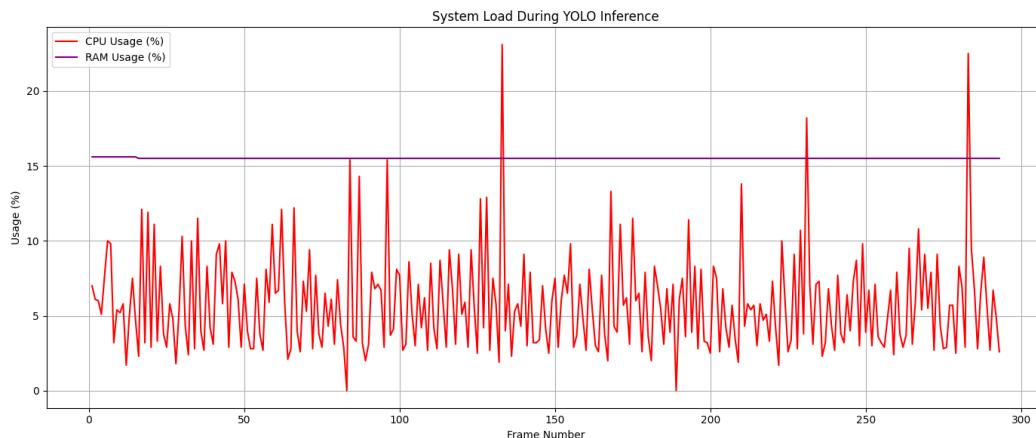
```

Abbildung 5.4: Timing Zusammenfassung CUDA vs RPI5

Durch die Verwendung der dedizierten Grafikkarte wird nun anstelle der CPU die GPU für die Inferenz verwendet. Dies führt zu einer deutlich geringeren Belastung der CPU des Systems (siehe 5.5)



(a) Systemauslastung Laborrechner ohne CUDA



(b) Systemauslastung Laborrechner mit CUDA

Abbildung 5.5: Vergleich Inferenz ONNX C++ und Python

Der Rückgabewert der Inferenz des Modells beinhaltet unter anderem die Koordinaten der Bounding Box, die Klasse und den Konfidenzwert des erkannten Objekts. Die Konfidenz gibt an, zu wie viel Prozent sich das Modell bei dem erkannten Objekt sicher ist. [44]

Diese Informationen können nun direkt für weitere Aktionen verwendet werden. Die detektierte Bounding Box wird dem Frame des Kamera-Streams hinzugefügt und auf dem Bildschirm angezeigt.

6. Einrichtung einer Ultra-Low-Latency Videoübertragung

6.1. Anforderungen an die Videoübertragung (Jürgens)

Die angebundene Kamera des RPi5 soll zukünftig neben dem Stream für die KI-Auswertung auch das 'Auge' des Roboters darstellen. So soll die Sicht des Roboters in nahezu Echtzeit abrufbar sein. Daher muss neben einer robusten Verbindung besonders auch die Latenz der Videoübertragung so gering wie möglich gehalten werden. Die bisherige Implementation mittels einer libcamera-Pipeline als Sender und einer GStreamer-Pipeline als Empfänger ist mit einer Latenz von etwa 1 Sekunde nicht für die Echtzeitübertragung geeignet. Nach einer Recherche zu verschiedenen Streaming-Protokollen und -Techniken wurde die Entscheidung getroffen, dass zukünftig eine Lösung mittels WebRTC verwendet werden soll. WebRTC bietet eine Lösung mit einer Latenz von unter 500 ms an, welches für dieses Projekt mehr als ausreichend ist [45].

6.2. Grundlegende Informationen zu WebRTC (Jürgens)

WebRTC ist ein freies, offenes Softwareprojekt welches eine Lösung für die Echtzeitkommunikation von Browern und mobilen Anwendungen bietet. Mittels einer Peer-To-Peer Verbindung können unter Anderem Audio- und Videodaten in Echtzeit zwischen den Peers übertragen werden. Dieses Vorgehen ermöglicht eine direkte Kommunikation zwischen den Peers ohne einen zentralen Server, was zu einer geringeren Latenz führt. Bei dem Verbindungsaufbau wirken verschiedene Komponenten und Protokolle zusammen, um eine stabile und sichere Verbindung zwischen den Peers herzustellen. Ein wichtiger Bestandteil von WebRTC ist der Signalausstausch. Dieser ist für den Austausch von Metadaten zwischen den Peers verantwortlich. [46]

6.3. Herstellung der WebRTC-Verbindung (Jürgens)

Insgesamt wurden zwei Ansätze bei der Herstellung der WebRTC-Verbindung zwischen dem RPi5 und dem Laborrechner verfolgt. Der erste Ansatz war der direkte Aufbau einer Peer-To-Peer Verbindung. Mit der Python-Bibliothek `aiortc` konnte eine WebRTC-Verbindung hergestellt werden. Mittels GStreamer und OpenCV wird diese Verbindung mit dem Kamerabild angereichert und an den Empfänger gesendet. Durch das Hinzufügen eines Keep-Alives-Kanal wird sichergestellt, dass die Verbindung zwischen den Peers beibehalten wird und nicht nach einer gewissen Zeit abbricht. Mit diesem Ansatz konnte eine stabile Verbindung mit geringer Latenz zwischen dem RPi5 und dem Laborrechner hergestellt werden.

Durch weitere Absprachen mit dem Team kam der Wunsch auf, dass die Videoübertragung nicht nur einzige und allein für die KI-Auswertung genutzt werden soll, sondern auch aus dem LAN über eine Website aufrufbar sein soll. Mit dieser Anforderung wurde der zweite Ansatz verfolgt, welcher den Einsatz eines MediaMTX-Servers beinhaltet. MediaMTX ist ein Open-Source Media-Server welcher eine Vielzahl von Protokollen unterstützt, darunter auch WebRTC. Der MediaMTX-Server wird dabei auf dem RPI5 installiert und ausgeführt. Die Konfiguration des Servers erfolgt über eine YML-Datei. In dieser können beispielsweise Ports und Protokolle festgelegt werden. Neben der grundlegenden Konfiguration kann hier auch das Startverhalten des Servers festgelegt werden. So wird beim Start automatisch ein Libcamera-Stream an den Server mittels RTSP übergeben. Über den Standardport 8889 kann der Stream schlussendlich über WebRTC mittels Browser abgerufen werden. Um den WebRTC-Stream auch außerhalb einer Browser-Umgebung abrufen zu können, bietet MediaMTX eine WHEP-Schnittstelle an. Über diese WHEP-Schnittstelle kann der Stream beispielsweise in anderen Anwendungen abgerufen werden. Diese Schnittstelle ermöglicht es, den Stream in unserem Python-Script zu verwenden und die KI-Auswertung auf dem Stream durchzuführen.

Mit dieser Implementation ist es möglich die KI-Auswertung auf einem Ultra-Low-Latency Stream anzuwenden und so in nahezu Echtzeit die Objekterkennung durchzuführen während der Kamerastream von anderen Geräten aus dem LAN per Browser angeschaut werden kann. In der Abbildung 6.1 ist die Architektur des WebRTC-Streams in Zusammenarbeit mit der KI-Auswertung noch einmal dargestellt.

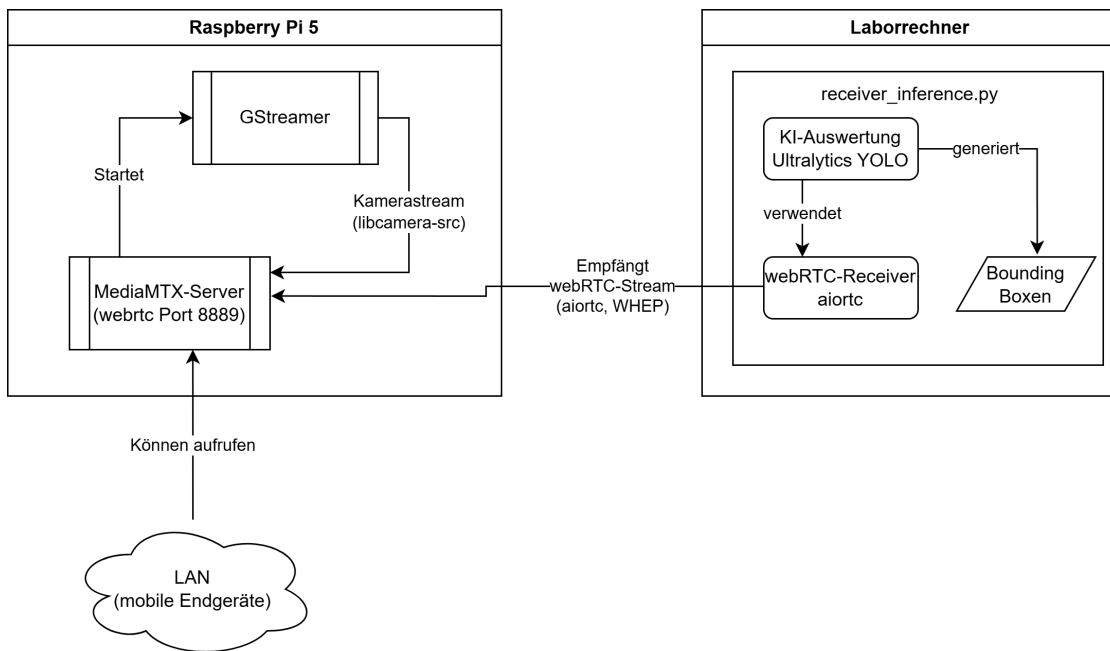


Abbildung 6.1: Architekturübersicht: Zusammenspiel von KI-Modell und WebRTC-Streaming

7. Objekttracking

7.1. Ziel des Objekttrackings (Jürgens, Specht)

Eine optionale Anforderung die in diesem Projekt am Ende noch aktiv verfolgt und umgesetzt wurde, war die Implementierung eines Objekttrackings. So sollte es möglich sein, die Koordinaten der Bounding Box zur Steuerung des Geschützarms zu verwenden. Das Ziel war also, den Geschützarm immer so auszurichten, dass die Bounding Box des Objekts immer in der Mitte des Kamerabildes ist.

7.2. Einrichtung des Kommunikationskanals (Jürgens)

Um tatsächlich Steuersignale an den Roboter zu senden, muss zunächst entschieden werden, wie dies geschehen soll. Dabei gab es zwei Ansätze die in Betracht gezogen wurden:

- Kommunikation über WebSockets
- Kommunikation über MQTT

Nach Absprachen mit dem Team wurde entschieden, dass die Kommunikation über MQTT erfolgen soll. Dies hat den Vorteil, dass mittels eines MQTT-Brokers eine zentrale Stelle geschaffen wird, an dem alle Nachricht gesammelt werden können. Dies ermöglicht neben dem grundlegend einfachen Austausch von Nachrichten auch eine gute Möglichkeit um das Projekt zu erweitern. So können weitere Sensorwerte oder auch die Stellung der Motoren über den MQTT-Broker ausgetauscht werden und optional für weitere Aktionen verwendet werden. Es wurde sich auf die Lösung mit dem Open-Source MQTT-Broker **Mosquitto** geeinigt. Dieser MQTT-Broker stellt eine leichtgewichtige Lösung bereit, welche nach dem Publish-Subscribe-Prinzip arbeitet [47]. So kann jeder Subscriber (Abonnent) ein oder mehrere Topics (Themen) abonnieren und erhält über diese dann Nachrichten. Dieser Ansatz soll den Nachrichtenverkehr für den Microcontroller minimieren, da hier zuvor bereits selektiert wird, welche Topics abonniert werden.

Die Mosquitto-Instanz wird auf dem Laborrechner installiert und ausgeführt. Die Konfiguration erfolgt über eine zentrale Konfigurationsdatei. Trotz korrekter Konfiguration konnte zunächst keine Verbindung zwischen dem MQTT-Broker und dem ESP32 hergestellt werden. Dieser Umstand wurde durch Ergänzung von Firewall-Regeln auf dem Laborrechner behoben. Es wurde auf dem Windows Rechner explizit eine eingehende und ausgehende Regel für Mosquitto erstellt. So konnte erfolgreich eine Verbindung zwischen dem ESP32 und dem MQTT-Broker hergestellt werden.

7.3. Implementierung des Objekttrackings (Jürgens, Specht)

Mit bestehender und funktionierender MQTT-Verbindung konnte nun das Objekttracking implementiert werden. Dabei musste besonders auf die Art und Weise geachtet werden, wie die Motorsteuerung implementiert wurde. Die Motoren werden mit Statusnachrichten versorgt, welche die absolute Position der Motoren angeben. So würde das Senden mehrere Nachrichten mit dem selben Inhalt zu einer einzigen Bewegung führen. Somit sind die Nachrichten idempotent und können mehrfach gesendet werden, ohne dass sich die Position der Motoren ändert. Wichtig war außerdem, dass die Berechnung der Drehwinkel auf dem Laborrechner erfolgt. Hiermit soll der ESP32 entlastet werden, da dieser nur als Abonnent des MQTT-Tops fungiert und die Motoren steuert.

Mit diesem Wissen wurde zunächst der Ansatz eines Mappings verfolgt. Dabei wurde der Geschützarm in die Null-Stellung gebracht. Durch Trial-and-Error wurden dabei die maximal benötigten Drehwinkel der Motoren ermittelt. Diese Werte wurden dann in ein Mapping überführt. So sollte sich das errechnete Zentrum der Bounding Box in einen benötigten Drehwinkel übersetzen lassen.

Dieses Vorgehen hat sich jedoch als nicht praktikabel herausgestellt. So wurden beispielsweise für das Erreichen des rechten Bildrandes eine andere Motorstellung benötigt als für das Erreichen des linken Bildrandes. So wurde beispielsweise für das Erreichen des rechten Bildrandes der horizontale Winkel auf -15 gesetzt, für das Erreichen des linken Bildrandes jedoch +18. Aufgrund dieser Abweichungen und einem inkonsistenten Verhalten der Motoren wurde der Ansatz verworfen und nicht weiter verfolgt.

Der zweite Ansatz wurde gemeinsam mit Michael Specht erarbeitet und verfolgt die Idee, die Kamermerkmale direkt mit in die Berechnung des Drehwinkels einzubeziehen. Dieser Ansatz erwies sich als deutlich erfolgreicher hatte dabei aber auch eine höhere Komplexität. Das Projekt war zu diesem Zeitpunkt bereits sehr weit fortgeschritten, weshalb sich das Ziel gesetzt wurde, einen einfachen Entwurf zu erstellen, der die grundlegende Funktionsweise des Objekttrackings demonstriert und die Erweiterbarkeit des Projekts zeigt.

Im ersten Schritt war es wichtig, die Kamermerkmale zu ermitteln. Die Recherche ergab, dass die Kamera ein diagonales Sichtfeld von 75° und eine Auflösung von 1280×1080 Pixeln hat. Weitere Merkmale wie beispielsweise der vertikale und horizontale Sichtwinkel (field of view, fov) konnten für das Raspberry Camera Module 3 nicht gefunden werden. Mittels Claude.ai wurde folgender Ansatz entworfen, um die Parameter programmatisch zu ermitteln:

Gegeben ist ein diagonales Sichtfeld von 75° sowie eine Bildauflösung von 1280×1080 Pixeln. Daraus ergibt sich ein Seitenverhältnis von

$$r = \frac{1280}{1080} \approx 1,185.$$

Der diagonale FOV wird zunächst in Bogenmaß umgerechnet:

$$\varphi_d = \text{rad}(75^\circ).$$

7. Objekttracking

Die horizontalen und vertikalen FOVs berechnen sich über trigonometrische Umformungen für rechteckige Sensoren zu:

$$\varphi_h = 2 \cdot \tan^{-1} \left(\frac{r \cdot \tan(\varphi_d/2)}{\sqrt{1+r^2}} \right), \quad \varphi_v = 2 \cdot \tan^{-1} \left(\frac{\tan(\varphi_d/2)}{\sqrt{1+r^2}} \right).$$

Einsetzen der Werte ergibt:

$$\varphi_h \approx 60,8^\circ, \quad \varphi_v \approx 52,7^\circ.$$

Diese Werten konnten für die weitere Berechnung der Drehwinkel verwendet werden. Auch dieser Teil wurde mit Hilfe von Claude.ai erarbeitet. Dabei war die Grundidee, eine Klasse namens *Servo Tracker* zu erstellen, die die Kamermerkmale und die absoluten Winkelpositionen der Motoren enthält. Die Funktionalität der Klasse ist im Wesentlichen in zwei Funktionen unterteilt:

1. *calculate_camera_relative_angles*: Diese Funktion berechnet die relativen Drehwinkel in horizontaler und vertikaler Richtung zwischen dem Zentrum des Kamerabildes und dem erkannten Punkt bzw. dem Mittelpunkt der Bounding Box. Dabei wird der horizontale und vertikale Öffnungswinkel (FOV) der Kamera auf die Anzahl der Bildpunkte (Pixel) umgerechnet, um so einen Winkel pro Pixel zu erhalten. Die Winkelabweichung ergibt sich anschließend durch die Differenz zwischen dem Zielpunkt und dem Bildzentrum.
2. *update_servo_position*: Diese Funktion nimmt die zuvor berechneten relativen Winkel und überträgt sie auf die aktuellen absoluten Positionen der Servomotoren. Wichtig ist dabei, dass die zuvor berechneten relativen Winkel auf den derzeitigen Absolutwinkel addiert werden. Da diese auch negativ sein können, werden sie in diesem Fall subtrahiert. Darüber hinaus werden Grenzen für minimale und maximale Positionen berücksichtigt, um mechanische Einschränkungen zu vermeiden. Anschließend werden die neuen Winkel gespeichert und zurückgegeben.

Des weiteren wurden die Hilfsfunktionen *get_current_position* zum Abgreifen der Werte der Motoren und *reset_to_zero* zum Zurücksetzen der Motorwerte implementiert.

Zusätzlich wurden zwei Hilfsfunktionen implementiert: *get_current_position* gibt die aktuellen absoluten Positionen der Servos zurück, während *reset_to_zero* die Motoren auf eine definierte Ausgangsposition zurücksetzt bzw. die Kamera geradeaus ausrichtet.

Die Klasse ermöglicht somit eine einfache und direkte Umsetzung einer objektzentrierten Verfolgung durch Umrechnung von Bildkoordinaten in mechanische Steuerbefehle.

Nun war es wichtig, Randbedingungen festzulegen, sodass beispielsweise genügend Zeit bleibt, um die Motoren zu bewegen und ein neues Bild zu erhalten. Bei der Definition unterstützte auch der Kollege Fabian Becker. Diese Randbedingungen waren für das Objekttracking relevant:

- *Reset nach 5 Sekunden*: Die Kamera liefert 30 Frames pro Sekunde, was bedeutet, dass 150 Frames in 5 Sekunden verarbeitet werden. Diese Zeit wird mit einem einfachen Counter überwacht. Über den Modulooperator wird geprüft, ob der Counter den Wert

7. Objekttracking

150 erreicht hat. In diesem Fall wird die Kamera auf die Null-Position zurückgesetzt und der Counter auf 0 gesetzt. Außerdem werden die Motoren auf die Null-Position zurückgesetzt. Dies ist wichtig, dass der Zustand der Motoren nur über die Klasse *ServoTracker* verwaltet wird und nicht über die MQTT-Kommunikation. Damit ist gemeint, dass keine Motorwerte über MQTT empfangen werden.

- *Minimale Bewegung ignorieren:* Um zu verhindern, dass kleine Bewegungen der Kamera bereits zu einer Neupositionierung des Geschützarms bzw. der Plattform führen, wurde eine Deadzone von 7° definiert. Dies bedeutet, dass die neue relative Abweichung von mindestens einer Achse (vertikal oder horizontal) größer-gleich 7 Grad sein muss. Andernfalls wird die Ausrichtung nicht verändert.
- *Bilder für Berechnung begrenzen:* Aufgrund von Latenzen zwischen der Interferenz, der Übertragung der Daten und der Verarbeitung der Bilder, wurde sich darauf festgelegt, nur jedes 7. Bild für die Berechnung der neuen Position zu verwenden. Dies wurde durch Testing im Labor ermittelt und hat sich als praktikabel erwiesen.

Abschließend wurden sowohl die Funktionen bzw. auch die Hilfsfunktionen der Klasse in Kombination mit den Randbedingungen und der MQTT-Kommunikation in die Funktion *run_track* integriert. Diese Funktion läuft in einer Endlosschleife aufgerufen und ... JENDRIK BITTE KURZ ERKLÄREN

8. Webserver

8.1. Funktion und Anforderung des Webservers (Koch)

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 Kapitel ?? erklärt.

8.2. Lösungsansatz und Umsetzung (Koch)

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 8.1 zu sehen.

Während der Entwicklung zeigte sich, dass die zunächst gewählte Architektur zur Sensoranbindung zu erheblichen Verzögerungen bei der Ausgabe der Gyrosensordaten führte. Die ersten Problemvermutungen bezogen sich dabei auf einen Konflikt mit der gleichzeitigen Nutzung desselben I²C Busses des Raspberry Pi's, was allerdings schnell durch unabhängige Tests ausgeschlossen werden konnte, indem man die Binaries des Ultraschallsensors und des Gyrosensors in zwei verschiedenen Terminals ausführten lies und es dort zu keiner sichtbaren Verzögerung kam. Die zweite Vermutung beruhte auf dem Verdacht, dass sich die schreibenden Befehle gegenseitig blockierten oder zumindest in diesem Fall der Ultraschallsensor den Vorrang erhält, jedoch konnte auch dies unter Verwendung eines Mutex für die Ausgabe geschlossen werden. Als tatsächliche Ursache stellte sich heraus, dass die ursprüngliche Implementierung versuchte, mittels einer Python-Funktion asynchron auf neue Sensornachrichten zu warten. In der Praxis wurden die Nachrichten jedoch nicht parallel, sondern nacheinander verarbeitet: Eine neue Nachricht konnte erst dann bearbeitet werden, wenn die vorherige vollständig abgearbeitet war. Da der Gyrosensor in den ersten Sekunden nach dem Start noch seine Offsets berechnet und dadurch verzögert Daten liefert, beginnt der Ultraschallsensor frühzeitig mit der Datenübertragung. Aufgrund seiner niedrigeren Abtastrate blockierte dessen Verarbeitung jedoch den Zugriff auf die Gyroskopdaten, welche erst nach Abschluss des Ultraschall-Zyklus gelesen werden konnten. So wurde der Gyrosensor dauerhaft ausgebremst, was zu verzögerten und gepufferten Messwerten führte [48]. 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.

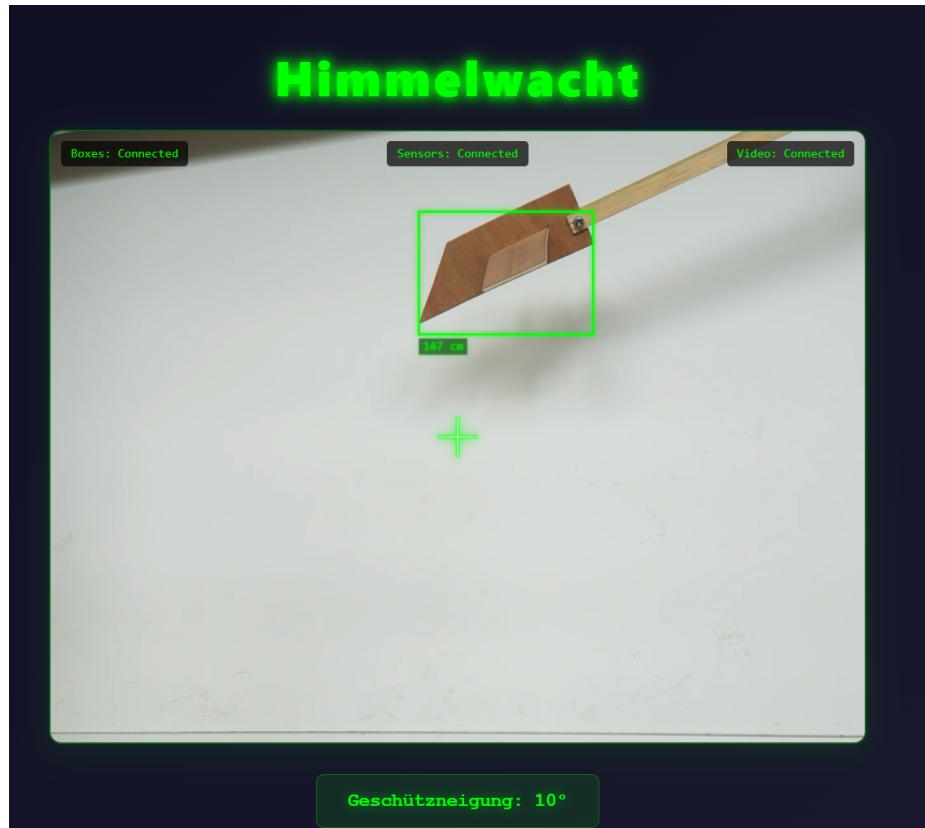


Abbildung 8.1: Webinterface zur Darstellung des Videostreams mit Bounding Boxes und Sensorsordaten

Stundenliste

Name: Fabian Becker
 Gruppe: 1 HimmelWacht

Pos.	Bezeichung	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 Platform, 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 Servostering	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 & Platform)	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
Gesamt:			171

Stundenliste

Name: Nicolas Koch
 Gruppe: 1 Himmelwacht

Pos.	Bezeichung	Beschreibung	Stunden in h
1	Vorbereitung	Gruppenbildung, Projektfindung, erste Recherche	4
2	Projektorganisation	Erstellung Projektbeschreibung, Terminplan & Plakat	8
3	Raspberry Pi	Mehrfaches Neuaufsetzen, Installation OpenCV	6
4	Stromversorgung	Altes Setup analysieren, Teile einplanen	5
5	KI	Packages installieren, gstreamer funktionalität prüfen	4
6	Stromversorgung	Aufbau planen, Kabel zurechtschneiden	4
7	Gyrosensor	Gyrodokumentation lesen, Programmierung beginnen	6
8	KI	Unterstützung KI tooling	3
9	Stromversorgung	Aufsetzen von neuem Plan + neue Bestelliste	2
10	Gyrosensor	Rohdaten umwandlung in Winkel	6
11	Gyrosensor	Implementierung Komplementärfilter	4
12	Gyrosensor	Fehleranalyse, Kalman-Filter implementierung	10
13	Lautsprecher	Unterstützung programmierung Mono-Ausgabe	4
14	Flywheel-Motoren	Löten Kondensator + Überarbeitung Code + Tests	4
15	Gyrosensor	Fehlerbehebung	3
16	Stromversorgung	Finale Verkabelung Grob, Anfang Verkabelung ESP32	6
17	Gesamttests	Gesamttest Fahrzeug	2
18	Webserver	Konzeptüberlegung, Aufbau HTML-Gerüst	5
19	Gyrosensor	Vergleich Rauschen mit/ohne Kalman, Diagramm	5
20	Webserver	Gyrowerte integrieren, versuch Koordinaten zu erhalten	5
21	Webserver	Bounding Boxen Koordinaten empfangen	5
22	Webserver	Daemon zum starten des Webservers und Sockets erstellen	5
23	Präsentation	Powerpointfolien erstellen	3
24	Webserver	Verschönerung Frontend, Integration Ultraschallsensor	4
25	Webserver	Fehleranalyse Output Gyro/Ultraschall	7
26	Dokumentation	Verfassen einer wissenschaftlichen Arbeit	20
Gesamt:			140

Stundenliste

Name: Jendrik Jürgens
 Gruppe: 1 - Himmelwacht

Pos.	Bezeichung	Beschreibung	Stunden in h
1	Vorbereitung	Gruppenbildung, Projektfindung, erste Recherche	4
2	Projektorganisation	Erstellung Projektbeschreibung, Terminplan & Plakat	8
3	Recherche	Recherche Yolo-Modelle, CUDA	5
4	Brainstorming	Erste Zusammenbauten, Aufgabenplanung,	3
5	KI-Vorbereitung	Bilder aufnehmen, Installation Labelstudio	2
6	Labeling	Datenannotierung Rectangles	5
7	Labeling	Datenannotierung Polygone	12
8	KI-Training	Training YOLOv8n/s, YOLO11n/s	3
9	Recherche	RTSP-Server, Mediatrix, Pi Camera Module<-->Python Tests	4,5
10	Setup	Kompilieren CMAKE, Einrichtung Raspberry PI 5,	6,5
11	Analyse	Evaluierung Performance Modelle	2
12	Setup	Kompilieren libcamera & ONNXRuntime	3,5
13	Implementation	Erstellung: C/C++ Quellcode Prototyp, Anbindung libcamera	3
14	Recherche & Testing	Gstreamer UDP Src <-> Windows Receiver	3
15	Konfiguration	Einrichtung MediaMTX Server	2
16	Setup	Einrichtung ONNX Runtime, CUDA/ cuDNN	2
17	Implementation	Codierung ONNX-Network, C++	7
18	Setup	Installation / Compiling OpenCV mit Gstreamer Python	2,5
19	Recherche	WebRTC & MediaMTX, aiortc, WHEP/WHIP, SDP	12
20	Codierung	Peer-to-Peer WebRTC Verbindung in Python	2,5
21	Implementation	Senden des annotierten Streams zum Mediatrix-Server	3,5
22	Troubleshooting	Glitches bei der Rücksendung des annotierten Servers	3,5
23	Implementation	Implementation: WebRTC-Stream Receiver Client <-> Mediatrix	3,5
24	Setup	Einrichtung Mosquitto Broker	3,5
25	Recherche	Windows Firewall Regeln für Mosquitto Broker	1,5
26	Implementation	Erfolgreiches Versenden von MQTT-Nachrichten	1
27	Brainstorming	Einbindung: Sensorwerte in die Objektdetektion, Teambesprechungen	3,5
28	Implementation	Steuerung X- und Y- Motoren des Geschützarmes per MQTT	4
29	Recherche & Testing	Impl. Kalman-Filter, Testen Ultralytics built-in Tracking-Funktionen	3,5
30	Implementation	Ansatz: Max. Drehwinkel auf Kamerabild mappen	4,5
31	Implementation	Implementation Objekttracking	7
32	Dokumentation	Erstellung Dokumentation	30
Gesamt:			161,5

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	7
2.2	Geschützarm Version 2	8
2.3	Montagehalterung für Pololu-Motortreiber	8
2.4	Abdeckung Plattform mit Position des Verbindungsstücks	9
2.5	Magazingewicht	10
2.6	Halterung DC Step-Down mit Stecksystem	11
2.7	Halterung für Flywheel Motortreiber	12
2.8	Mikrocontroller Case	12
2.9	Lautsprecherhalterung	13
3.1	Beispielhafter Aufbau eines HID-Reports aus [8]	15
3.2	Aufbau des Dualshock4 Treibers	17
3.3	Erklärung ON/OFF Parameter für PCA9685 aus [19, S. 17]	19
3.4	Testaufbau des Fault-Pins des Pololu-Motor-Treibers	23
3.5	Differentialantrieb Logik	25
3.6	Speicherauslastung vor der Optimierung	32
3.7	Partitionierung des Flash-Speichers vor und nach der Optimierung	33
3.8	Freier Speicher nach der Optimierung	33
4.1	Pitch: Vergleich Rohdaten und Kalman-Filter	39
4.2	Roll: Vergleich Rohdaten und Kalman-Filter	40
5.1	Vergleich Trainingsresultate YOLOv8n, YOLOv8s, YOLO11n und YOLO11s	45
5.2	Vergleich Inferenz ONNX C++ und Python	46
5.3	System- und Zeitanalyse während der Inferenz (RPI5)	48
5.4	Timing Zusammenfassung CUDA vs RPI5	49
5.5	Vergleich Inferenz ONNX C++ und Python	49
6.1	Architekturübersicht: Zusammenspiel von KI-Modell und WebRTC-Streaming	52
8.1	Webinterface zur Darstellung des Videostreams mit Bounding Boxes und Sensordaten	59

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] discoded_2331131. „MPU 6050 (GY-521) input shaper mount (remix).“ (10. Aug. 2024), Adresse: <https://www.printables.com/model/969760-mpu-6050-gy-521-input-shaper-mount-remix/files> (besucht am 01.07.2025).
- [5] laurb9_2937543. „28BYJ-48 Stepper Model.“ (16. Feb. 2024), Adresse: <https://www.printables.com/model/1193469-28byj-48-stepper-model> (besucht am 01.07.2025).
- [6] R. AG. „SRF02 - Low Cost, High Performance Ultraschall Entfernungssensor.“, Adresse: <https://de.reichel-versand.de/DEV-SRF02.shtml> (besucht am 01.07.2025).
- [7] „Getting Started with ESP-IDF.“, Adresse: <https://idf.espressif.com/> (besucht am 01.07.2025).
- [8] Matlo. „DS4-BT.“ (15. Jan. 2015), Adresse: http://eleccelerator.com/wiki/index.php?title=DualShock_4 (besucht am 02.07.2025).
- [9] USB Implementers' Forum. „Device Class Definition for Human Interface Devices (HID).“ (25. Juli 2001), Adresse: https://usb.org/sites/default/files/hid1_11.pdf (besucht am 02.07.2025).
- [10] „Bluetooth® HID Host API.“ (2025), Adresse: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/bluetooth/esp_hidh.html (besucht am 02.07.2025).
- [11] „ESP HID Host API Issues.“ (17. Feb. 2025), Adresse: <https://github.com/espressif/esp-idf/issues/9767> (besucht am 02.07.2025).
- [12] BlueKitchen GmbH. „BTstack.“ (9. Juni 2025), Adresse: <https://github.com/bluekitchen/btstack> (besucht am 04.07.2025).
- [13] R. Quesada. „Bluepad32 documentation.“ (2024), Adresse: <https://bluepad32.readthedocs.io/> (besucht am 04.07.2025).

- [14] Espressif. „ESP32 Series Datasheet.“, Adresse: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (besucht am 01.07.2025).
- [15] „DC Toy / Hobby Motor - 130 Size.“, Adresse: <https://www.adafruit.com/product/711> (besucht am 01.07.2025).
- [16] „TowerPro MG92B Servo.“, Adresse: <https://servodatabase.com/servo/towerpro/mg92b> (besucht am 01.07.2025).
- [17] „TowerPro MG996R Servo.“, Adresse: <https://servodatabase.com/servo/towerpro/mg996r> (besucht am 01.07.2025).
- [18] J. Supcik. „ESP-IDF PCA9685 I2C PWM Driver.“ (19. Okt. 2024), Adresse: <https://github.com/supcik/idf-pca9685> (besucht am 01.07.2025).
- [19] NXP Semiconductors. „PCA9685 Product data sheet.“ (16. Apr. 2015), Adresse: <https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf> (besucht am 01.07.2025).
- [20] „SG90 - Servomotor.“, Adresse: https://www.elektronik-kompendium.de/sites/praxis/bauteil_sg90.htm (besucht am 02.07.2025).
- [21] W. Z. B. D. Hirpo. „Design and Control for Differential Drive Mobile Robot.“ (Okt. 2017), Adresse: <https://www.ijert.org/research/design-and-control-for-differential-drive-mobile-robot-IJERTV6IS100138.pdf> (besucht am 01.07.2025).
- [22] „Getting Started with Wi-Fi on ESP-IDF,“ Developer Portal. (31. Juli 2024), Adresse: <https://developer.espressif.com/blog/getting-started-with-wifi-on-esp-idf/> (besucht am 01.07.2025).
- [23] „Esp-idf/examples/protocols at master · espressif/esp-idf,“ GitHub., Adresse: <https://github.com/espressif/esp-idf/tree/master/examples/protocols> (besucht am 07.07.2025).
- [24] „ESP-MQTT - ESP32 - — ESP-IDF Programming Guide v5.4.2 Documentation,“ Adresse: <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/protocols/mqtt.html> (besucht am 07.07.2025).
- [25] „Esp-idf/examples/protocols/mqtt/tcp/main/app_main.c at v5.4.1 · espressif/esp-idf,“ GitHub., Adresse: https://github.com/espressif/esp-idf/blob/v5.4.1/examples/protocols/mqtt/tcp/main/app_main.c (besucht am 07.07.2025).
- [26] B. Albrecht. „Max98357 - What is I2S?“ (4. Apr. 2022), Adresse: <https://www.az-delivery.de/en/blogs/azdelivery-blog-fur-arduino-und-raspberry-pi/max98357-was-ist-eigentlich-i2s> (besucht am 06.07.2025).
- [27] „Max98357 I2S 3W Class D Amplifier Breakout,“ Adresse: https://www.amazon.com/Teyleton-Robot-Amplifier-Interface-Filterless/dp/B0B4GK5R1R/ref=sr_1_2?sr=8-2 (besucht am 06.07.2025).
- [28] InvenSense, Inc. „MPU-6000 and MPU-6050 Product Specification.“ (Aug. 2013), Adresse: <https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf> (besucht am 01.07.2025).

- [29] MathWorks. „atan2.“, Adresse: <https://www.mathworks.com/help/matlab/ref/double.atan2.html> (besucht am 02.07.2025).
- [30] RWTH Aachen. „Kalmanfilter.“, Adresse: https://www.irt.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaacdvoafc (besucht am 02.07.2025).
- [31] Lauszus. „A Practical Approach to Kalman Filter and How to Implement It.“ (2012-09-10), Adresse: <https://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/> (besucht am 06.07.2025).
- [32] Raspberry Pi Ltd. „Raspberry Pi 5.“ (2025), Adresse: <https://www.raspberrypi.com/products/raspberry-pi-5/> (besucht am 28.06.2025).
- [33] Ultralytics Inc. „Ultralytics YOLO Frequently Asked Questions (FAQ).“ (Nov. 2023), Adresse: <https://docs.ultralytics.com/faq/> (besucht am 28.06.2025).
- [34] Ultralytics Inc. „Performance Metric.“ Letzte Änderung am 01. April 2025. (Nov. 2023), Adresse: <https://docs.ultralytics.com/de/models/yolov8/#performance-metrics> (besucht am 29.06.2025).
- [35] G. Jocher, A. Chaurasia und J. Qiu, *Ultralytics YOLOv8*, Version 8.0.0, 2023. Adresse: <https://github.com/ultralytics/ultralytics> (besucht am 29.06.2025).
- [36] G. Jocher und J. Qiu, *Ultralytics YOLO11*, Version 11.0.0, 2024. Adresse: <https://github.com/ultralytics/ultralytics> (besucht am 29.06.2025).
- [37] Ultralytics Inc. „ONNX (Open Neural Network Exchange).“, Adresse: <https://docs.ultralytics.com/de/glossary/onnx/> (besucht am 30.06.2025).
- [38] Raspberry Pi Ltd. „Raspberry Pi Camera Module 3.“ (Juni 2024), Adresse: <https://www.raspberrypi.com/products/camera-module-3/> (besucht am 28.06.2025).
- [39] NVIDIA Corporation. „GEFORCE RTX 3070-Familie.“, Adresse: <https://www.nvidia.com/de-de/geforce/graphics-cards/30-series/rtx-3070-3070ti/> (besucht am 30.06.2025).
- [40] Ultralytics Inc. „Data Augmentation.“ Letzte Änderung am 22. Juni 2025. (März 2025), Adresse: <https://docs.ultralytics.com/de/guides/yolo-data-augmentation/#why-data-augmentation-matters> (besucht am 30.06.2025).
- [41] Ultralytics Inc. „Transfer Learning.“, Adresse: <https://www.ultralytics.com/glossary/transfer-learning> (besucht am 30.06.2025).
- [42] Ultralytics Inc. „Kreuzung über Union (IoU).“, Adresse: <https://www.ultralytics.com/de/glossary/intersection-over-union-iou> (besucht am 30.06.2025).
- [43] Ultralytics Inc. „Epoch.“, Adresse: <https://www.ultralytics.com/de/glossary/epoch> (besucht am 30.06.2025).
- [44] Ultralytics Inc. „Modellvorhersage mit Ultralytics YOLO.“, Adresse: <https://docs.ultralytics.com/de/modes/predict/#working-with-results> (besucht am 30.06.2025).

- [45] A. Lorenzetti. „WebRTC Latency: Comparing Low-Latency Streaming Protocols (Update).“ (Feb. 2024), Adresse: <https://www.nanocosmos.de/blog/webrtc-latency/> (besucht am 30.06.2025).
- [46] flussonic. „WebRTC-Server: Der Schlüssel zur Erstellung moderner Videochats.“ (Apr. 2024), Adresse: <https://flussonic.com/de/blog/news/webrtc-videochat#was-ist-webrtc-ein-tiefer-einblick-in-die-technologie> (besucht am 30.06.2025).
- [47] Mosquitto. „Mosquitto - Open Source MQTT Broker.“, Adresse: <https://mosquitto.org/> (besucht am 03.07.2025).
- [48] RealPython. „anext().“ (26. März 2025), Adresse: <https://realpython.com/ref/builtin-functions/anext/> (besucht am 03.07.2025).