TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät

Elektro- & Informationstechnik

Projektarbeit

Roboterformation

Fortführung

Abgabetermin: Nürnberg, den 23.06.2022

Projektteilnehmer:
Cara Bettendorf, Maximilian Dösch, Kevin Heise und Moin Sammari

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einführung in die Projektarbeit

Als ein ansprechendes Modell zum Vorführen bei Messen oder Tag-der-offenen-Tür der Hochschule und des Lehrstuhls wurde das Projekt Roboterformation in Auftrag gegeben. Von Professor Bernhard Wagner betreut, soll im Laufe mehrerer Projektgruppen eine Formation von bis zu zwanzig mobilen Robotern entwickelt werden, die einen gesteuerten Tanz aufführen.
Diese Projektgruppe ist nunmehr das vierte Team, das sich dieser Aufgabe widmet.

An die Ergebnisse unserer Vorgänger Bachelor- und Mastergruppen anknüpfend ist es unser Ziel den bestehenden Roboter zu verbessern und sowohl hardware- als auch softwaretechnisch so auszulegen, damit dieser für die Formation in massentauglicher Stückzahl produziert werden kann. Gleichzeitig ist es das Ziel, dass der Roboter genau einer vorgegebenen Trajektorie folgen kann. Hierbei liegt unser Fokus darin, die Grundlage dazu an einem bis zu maximal zwei Robotern zu schaffen. Der Aufbau von mehreren Robotern als Formation, die synchronisierte Bewegungsabläufe vollziehen, wird erst von nachfolgenden Gruppen bearbeitet werden.

Die Hardware ist durch die vorangegangene Gruppe bereitgestellt worden. Der Aufbau besteht aus zwei Rädern, die jeweils durch einen Motor angesteuert werden. Zusätzlich ist ein Stützrad angebracht, welches aus zwei Kugellagern besteht, sodass eine dreieckförmige Anordnung entsteht. Dadurch kann ein 360 Grad Fahren ermöglicht werden. Der hierzu benötigte Strom wird durch einen Akku mit 5V bereitgestellt.

Die Software wird zweigeteilt ausgeführt. Zum einen wird in einer ROS (Robot Operating System) Umgebung auf einem externen Computer die Trajektorienplanung des Roboters berechnet. Hierbei stellt ROS ein Softwarepaket dar, das viele Bibliotheken mit sich bringt, die das Programmieren eines Roboters deutlich vereinfachen und in verschiedene Nodes aufgeteilt wird. Der Code wird in der Programmiersprache C++ verfasst und sendet dann über sogenannte Publisher Daten an Topics. Der ROS-Server speichert diese Nachrichten und stellt sie anderen Nodes, die dieses Topic abonnieren, den sogenannten Subscribern, zur Verfügung. Zum Bestimmen der Trajektorie werden einzelne Wegpunkte, die der Roboter auf dem Weg zum gewünschten Endpunkt abfahren soll, mit den geeigneten mathematischen Splines ermittelt. Nach der Berechnung wird die Trajektorie als Array mit allen errechneten Punkten und korrespondierenden Zeitwerten an den Mikrocontroller übergeben. Auf diesem findet der zweite Teil der Software-Entwicklung statt.

Als Mikrocontroller verwendet unsere Projektgruppe den 2-Kern Prozessor ESP32. Dieser verfügt über

WLAN, Bluetooth, effizientes Powermanagement und verschiedene Peripherien als Funktionen. Hierüber werden die Motoren angesteuert. Um die Geschwindigkeit einzustellen, wird eine Drehzahlregelung verwendet. Dazu wird an der AuSSenseite des Motors gemessen, wie oft sich das Rad innerhalb einer Sekunde vollständig gedreht hat. Über die ermittelte Drehzahl kann mit der Differenzbildung zum Sollwert die entsprechend benötigte Geschwindigkeit eingestellt werden. Gleichzeitig läuft auf dem ESP32 übergeordnet eine Top Level State machine. Diese bestimmt, ob zu einem gewissen Zeitpunkt Positionen eingelesen oder Trajektorien berechnet wurden und gibt letztlich den Befehl, dass eine Trajektorie ausgeführt werden soll. Da der Roboter und die zukünftige Formation besonders zu Werbezwecken aufgeführt werden soll, wird auch auf eine besonders ansprechende Visualisierung geachtet. Dazu befinden sich auf der Platine über 40 LEDs, die je nach Wunsch in verschiedenen Farben erleuchten. Zudem wurde mittels des 3D-Druckers ein transparentes Gehäuse gedruckt, durch das das LED-Licht gestreut wird. Auch die Ansteuerung des Lichts wird über den Mikrocontroller abgewickelt.

Unsere Webseite stellt die Oberfläche für den Endnutzer zur Verfügung. In den Programmiersprachen CSS und react.js verfasst, bietet sie die Schnittstelle zwischen der Eingabe der gewünschten Zielpunkte, Lichteffekte, manuellen Steuerung des Roboters sowie einer Anzeige der aktuellen Positionen der jeweiligen Roboter. Grundsätzlich kann man sich den Prozessablauf wie folgt vorstellen: Auf einem zentralgesteuerten PC wird die Trajektorie im ROS-System berechnet und über das selbst entwickelte Verbindungsprotokoll an den Mikrocontroller geschickt. Dieser sitzt auf der Platine, die auf dem Roboter aufgebracht ist. Dann setzt sich der Roboter zum angegebenen Ort oder in der vorgegeben Formation in Bewegung. Dem geht zuvor die Eingabe des gewünschten Punktes auf unserer Webseite einher. Zusätzlich können dort Einstellungen zur Geschwindigkeit und Licht vorgenommen werden.

1.1 Einordnung dieser Projektarbeit in das Gesamtprojekt

Verwendung der durch die Mastergruppe vorgegebenen Regelungen und Trajektoriengenerierung → Integration/Implementierung auf echte Hardware

1.2 Zielsetzung der Projektarbeit

Kapitel 2

Platinenentwurf

Ein Ziel dieser Projektarbeit war die Entwicklung einer Platform, welche alle notwendigen Komponenten kompakt zusammenfasst. Hierfür müssen elektronische Bauteile und mechanische wie Motoren oder 3D-Druckteile gut aufeinander abgestimmt werden. Während auf der einen Seite die Funktionalität des Gesamtsystems wichtig ist, soll andererseits jeder Roboter auch visuell durch ein Array von RGB-Leds auffallen. Mithilfe eines doppel Platinenstack wurde erreicht, dass die Motoren direkt auf die untere Platine (Main_PCB) aufgeschraubt werden können und die Leds auf der oberen Platine (Led_PCB) den richtigen Abstand zum Led-Diffusor haben.

2.1 Blockschaltbild

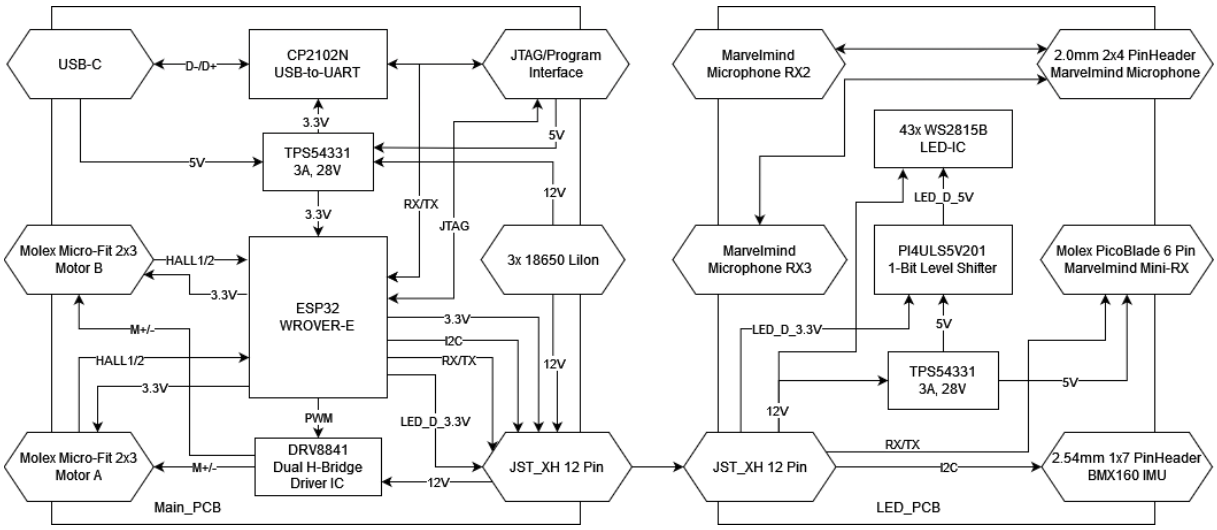

Abbildung 2.1: Blockdiagramm PCB

Das Blockdiagramm zeigt den Leistungs- und Signalfluss zwischen den Bauteilen auf beiden Platinen. Die primäre Stromversorgung übernehmen drei 18650 LiIon Zellen. Diese haben im geladenen Zustand eine Gesamtspannung von $3 * 4.2 = 12.6V$ und können über einen Schalter an- und ausgeschalten werde. Der Roboter ist gegenüber Verpolung aller Zellen geschützt. Für die Versorgung des Mikrocontrollers müssen die 12V mit einem Buck-Converter auf 3.3V gewandelt werden. AuSSerdem soll der Roboter für

den Test-/Programmierbetrieb auch über die 5V der USB-C oder der JTAG/Programmierschnittstelle stationär betrieben werden können. Hierfür müssen die Spannungen elektronisch getrennt werden, um einen Kurzschluss zwischen den Potentialen zu verhindern. Der ESP32 kann über die USB-C Schnittstelle und eine separate serielle Schnittstelle programmiert werden. Für den 2 Quadranten betrieb der Motoren ist eine doppel H-Brücke vorgesehen. Die Ist-Geschwindigkeit wird über Hallsensoren auf den Bürstenmotoren ermittelt. Auf die zweite Platine wird der Marvelmind Mini-RX Empfänger aufgesteckt. Dieser kommuniziert über UART mit dem Mikrocontroller. Die beiden externen Mikrofone können einfach auf die LED_PCB Platine gelötet werden. Da das Marvelmind Modul 5V benötigt werden diese über einen zweiten Buck-Converter erzeugt. Die 43 RGB Led-ICs werden über die 12V Spannung versorgt und mittels einer Datenleitung vom Mikrocontroller angesteuert. Da die Led Bausteine 5V Logikpegel benötigen, wird das 3.3V Signal vom ESP32 mittels 1-Bit Level Shifter verstärkt.

2.2 Bauteile

2.2.1 ESP32

Das Herzstück des Roboters bildet ein ESP32-WROVER-E in der 8MB Flash Variante. Dieses SOM (System On Module) von Espressife beinhaltet einen Xtensa LX6 Dualcore Mikrocontroller mit 520kB internen und 8MB externen SRAM, sowie 2MB internen und 8MB externen Flash Speicher. Die externen Speicher sind über SPI verbunden und werden über eine MMU (Memory Management Unit) in den Speicherbereich gemappt. Des Weiteren bietet der ESP32 verschiedene Peripherie wie Timer, GPIO, UART, I2C, PWM und Wifi. Espressife stellt mit dem ESP-IDF Framework unter anderem ein Hardwareabstraktions Layer bereit über welches Peripherie leicht initialisiert und gesteuert werden kann. Aus diesem Grund bleibt es dem Programmierer erspart sich mit der Hardware auf Registerebene zu beschäftigen. Dies macht den ESP32 zu einem beliebten IOT-Mikrocontroller im Hobbybereich. Diagramm 2.2 und Tabelle 2.3 beschreibt die Pinbelegung am ESP32.

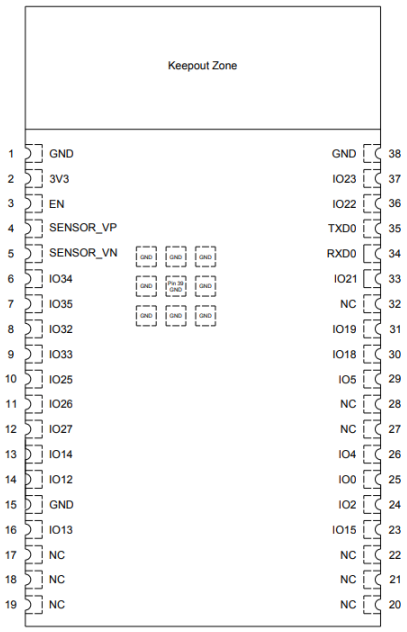

Abbildung 2.2: ESP32 Pin Layout

Name	Pin Nr.	Funktion
GND	1	Ground
3V3	2	Power
EN	3	Enable Signal
Sensor_VP	4	Hall Sensor 1 Motor B
Sensor_VN	5	Hall Sensor 2 Motor B
IO34	6	Hall Sensor 1 Motor A
IO34	7	Hall Sensor 2 Motor A
IO32	8	SCL I2C Bus
IO33	9	SDA I2C Bus
IO25	10	serielles LED Signal
IO26	11	Marvelmind UART TX
IO27	12	Marvelmind UART RX
IO14	13	JTAG_TMS
IO12	14	JTAG_TDI
GND	15	Ground
IO13	16	JTAG_TCK
IO15	23	JTAG_TDO
IO2	24	GPIO2 (sollte nicht verwendet werden, da Strapping Pin)
IO0	25	GPIO0 Boot Mode Selektor
IO4	26	NC
IO5	29	GPIO5 (sollte nicht verwendet werden, da Strapping Pin)
IO18	30	DRV8841 Motor B IN2
IO19	31	DRV8841 Motor B IN1
IO21	33	DRV8841 Motor A IN1
RXD0	34	UART für Programmierung RX
TXD0	35	UART für Programmierung TX
IO22	36	DRV8841 Motor A IN2
IO23	37	DRV8841 Enable
GND	38	Ground

Abbildung 2.3: ESP32 Pin Belegung

2.2.1.1 Power-Up

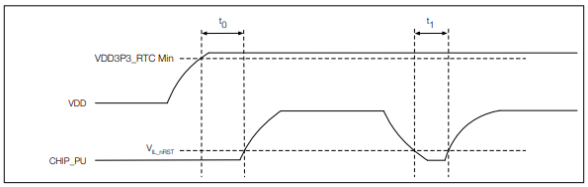

Abbildung 2.4: ESP32 Power-Up

Wie die Illustation 2.4 zeigt muss der Enable Pin (im Datenblatt CHIP_PU genannt) für eine bestimmte Zeit $t_0 = 50\mu s$ nach dem Power-Up auf Low gehalten werden. Das gleiche gilt für den Reset des Mikrocontrollers $t_1 = 50\mu s$. Die Power-Up/Reset Schaltung könnte zum Beispiel als Parallelschaltung eines RC Tiefpasses mit einem Taster umgesetzt werden.

2.2.1.2 Boot-Mode

Der ESP32 verfügt über zwei verschiedene Boot Modis. Beim SPI Boot wird die Firmware aus dem Flash in den Arbeitsspeicher geladen. Der Download Boot dient zum flashen einer neuen Firmware. Für den Wechsel zwischen den Modis wertet der Mikrocontroller beim Power-Up die Strapping Pins GPIO0 und GPIO2 aus.

Pin	Default	SPI Boot	Download Boot
GPIO0	Pull-Up	1	0
GPIO2	Pull-Down	Don't care	0

Tabelle 2.1: ESP32 Boot Mode

Da GPIO2 intern auf LOW gezogen wird, muss dieser Pin nicht verbunden werden. Die Boot Modus Wahl geschieht dann alleine über GPIO0.

2.2.1.3 UART0

Über die Pins UART0-RX/TX wird eine serielle Verbindung erstellt. Diese ermöglicht das Schreiben in den Flash-Speicher im Dowload-Boot und einen seriellen Monitor für z.B. printf Debugging im SPI-Boot.

2.2.1.4 JTAG

TMS, TDI, TCK, TDO können für JTAG-Debugging verwendet werden.

2.2.1.5 I2C

Auf beiden Platinen sind Anschlüsse an den I2C-Bus des ESP32 vorgesehen. Über diesen soll aber primär eine 9DOF-IMU (Intertial Measurement Unit) betrieben werden.

2.2.2 CP2102N

Der CP2102N wirkt als "Übersetzerßwischen der USB2.0 Fullspeed Schnittstelle und dem UART Interface. Somit kann durch einen Virtual COM Port Treiber am Host-PC über USB- auf die UART-Schnittstelle zugegriffen werden. Die UART Seite wird über die Pins RXD/TXD und das differenzielle USB Signal über die D+/D- Pins mit dem CP2102N verbunden. Dies ermöglicht das Programmieren des ESP32 einfach über USB. AuSSerdem kann auch auf den seriellen Monitor zugegriffen werden. Des Weiteren kann das RTS (Ready to Send; LOW aktiv) und DTR (Data Terminal Ready; LOW aktiv) Signal zur Auswahl des Boot Modus und Neustarten des Mikrocontrollers verwendet werden. Die folgende Tabelle 2.2 zeigt die notwendige logische Verknüfung der Signale.

DTR	RTS	EN	GPIO0
1	1	1	1
0	0	1	1
1	0	0	1
0	1	1	0

Tabelle 2.2: ESP32 DTR RTS

2.2.3 Pololu 20D 63:1 Getriebemotoren

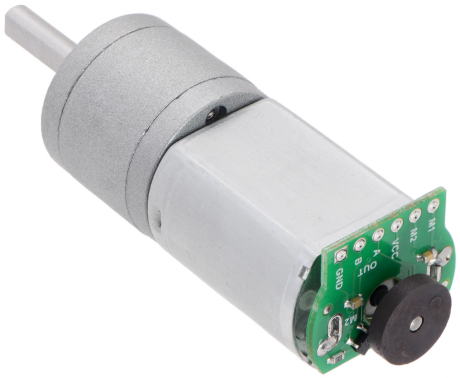

Abbildung 2.5: Pololu 20D Motor

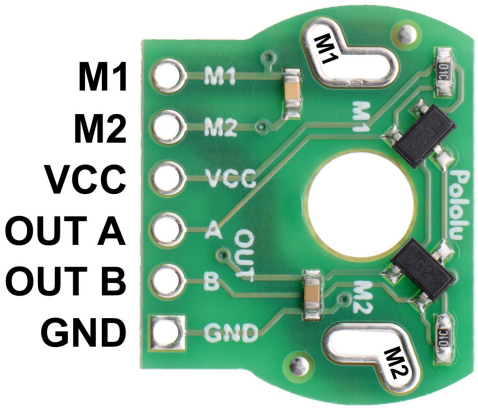

Abbildung 2.6: Pololu 20D Rückplatte

Die Pololu 20D Getriebemotoren sind für 12V Spannung ausgelegt. Dabei haben sie einen Leerlaufstrom von $80mA$ und im Stillstand $1.6A$. Die Leerlaufdrehzahl wird durch das eingebaute Getriebe um den Faktor 63 auf $220RPM$ reduziert. Für Drehzahlmessungen kann der Motor durch eine Platine und eine Magnetscheibe mit 10 Polen an der Motorwelle erweitert werden. Auf der Platine befinden sich jeweils zwei Hall-Magnetfeldsensoren. Diese geben je nach Magnetfeldrichtung einen HIGH oder LOW Spannungspegel aus. Durch Zählen der Pegeländerungen pro Zeiteinheit kann die Geschwindigkeit des Motors berechnet werden. Die Drehrichtung wird über das Vorzeichen der Phasendifferenz der Signale von den beiden Hallsensoren ermittelt.

2.2.4 DRV8841

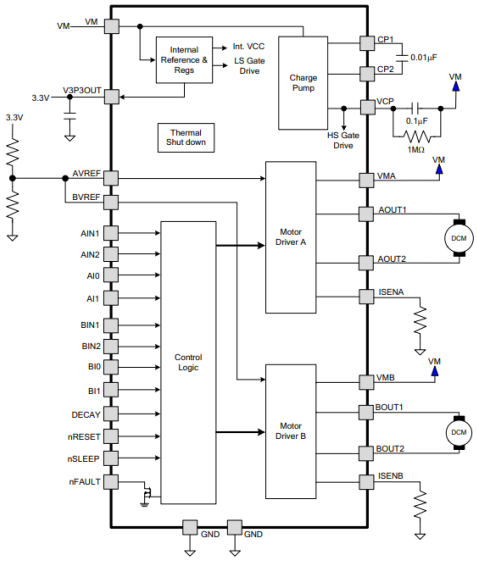

Abbildung 2.7: DRV8841 Pinout

xIN1	xIN2	xOUT1	xOUT2
0	0	L	L
0	1	L	H
1	0	H	L
1	1	H	H

Abbildung 2.8: DRV8841 H-Brücken Logik

Mit der doppel H-Brücke DRV8841 von Texas Instruments können zwei DC-Motoren im 2 Quadranten-Betrieb betrieben werden. Die beiden Motorausgänge (AOUT/BOUT) dürfen jeweils mit bis zu 2.5A belastet werden. Die Pololu 20D Motoren haben einen Stillstandsstrom von maximal 1.6A. Somit sollte der Treiber auch ohne die interne Strombegrenzung sicher betrieben werden können. Dies bedeutet die Pins AI0, AI1, BI0, BI1 müssen nicht verbunden werden, ISENA und ISENB können direkt mit GND verbunden werden und AVREF, BVREF werden auf 3.3V gezogen. Geschwindigkeit und Drehrichtung der Motoren wird über die Pegel und Pulseweite der (AIN1/AIN2; BIN1/BIN2) Eingangssignale gesteuert. 2.8 Um den Motortreiber zu aktivieren müssen nRESET und nSLEEP auf 3.3V gezogen werden. Des Weiteren ist auch noch ein Bootstrap-Kondensator zwischen den Pins CP1/CP2 als Ladepumpe notwendig.

2.2.5 Marvelmind Mini-RX

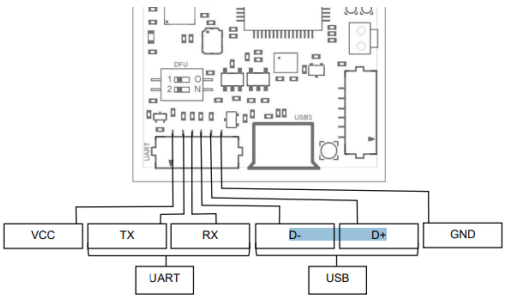

Abbildung 2.9: Marvelmind Molex Pinout

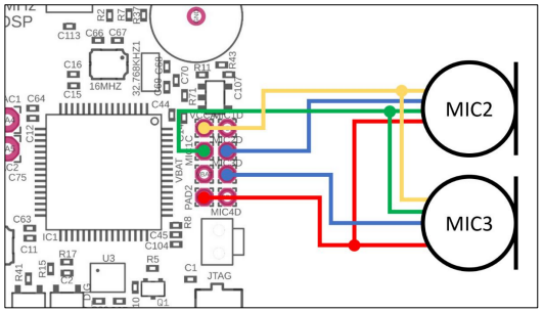

Abbildung 2.10: Marvelmind Mic Pinout

Der Marvelmind Mini-Rx Beacon empfängt die Ultraschallsignale der HW4.9 Beacons für die Berechnung der Pose des Roboters. Über die integrierte UART Schnittstelle kann dann die Pose an den Mikrocontroller gesendet werden. Da durch das Modul effektiv eine dritte Platine entsteht, wurde darauf geachtet diese möglichst platzschonend in den Platinenstack zu integrieren. Hierfür wird das Gehäuse und der eingebaute Akku entfernt. Die Stromversorgung wird über auf der LED Platine erzeugte 5V gewährleistet. Die 5V Spannung sowie die seriellen Signale RX/TX sind über die interne Molex PicoBlade Steckverbindung 2.9 mit dem Modul verbunden. Da die UART Schnittstelle mit 3.3V Logikpegeln arbeitet kann diese ohne Probleme direkt mit dem ESP32 verbunden werden. Des Weiteren soll das eingebaute Mikrofon durch zwei externe ersetzt werden. Diese sind über das 2.0mm 2x4 Pinout mit dem Mini-RX verbunden 2.10.

2.2.6 WS2815B

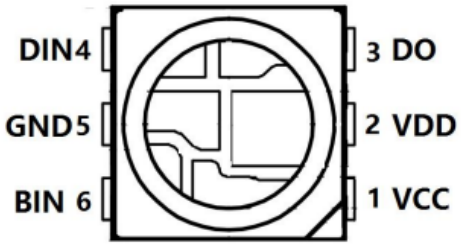

Abbildung 2.11: WS2815B Pinout

Der WS2815B ist eine RGB-Led mit eingebautem LED-Treiber. Dieser Baustein kann über die Pins DO, DIN in Reihe zu einem Led-Array verschalten werden. Der RGB Farbwert wird in einem 24Bit (8Bit grün, 8 Bit rot, 8 Bit blau) Feld codiert. Dieser Wert wird für alle LEDs in einem Datenstrom kaskadiert und an den Eingang DIN der erste LED gesendet. Diese schneidet sich den ersten Farbwert ab und gibt den restlichen Datenstrom über DO an die nächste LED weiter. Somit sind alle Bausteine einzeln adressierbar. Des Weiteren wird 0 und 1 nicht über den Wert der Spannungspegel codiert. Um ein Bit zu übertragen benötigt es zu Beginn eine positive Flanke, dann eine negative und zum Schluss wieder ein positive Flanke. Der Wert wird dann in die Dauer der High und Low-Phasen codiert. Die LED wird mit 12V gespeist. Die Logikpegel der Datensignale arbeiten allerdings mit 5V Spannung. An den VCC Pin kann ein Entkopplungskondensator angeschlossen werden. AuSSerdem wird der Eingang BIN mit DO der vorletzten LED verbunden und bringt somit Redundanz beim Aufall eines Treibers.

2.2.7 PI4ULS5V201

Da die LED-Treiber 5V Signalpegel benötigen, können diese nicht direkt mit dem ESP32 angesteuert werden. Der PI4ULS5V201 ist ein Level-Shifter IC und kann somit den 3.3V Pegel des ESP32 auf die benötigten 5V verstärken.

2.2.8 TPS54331

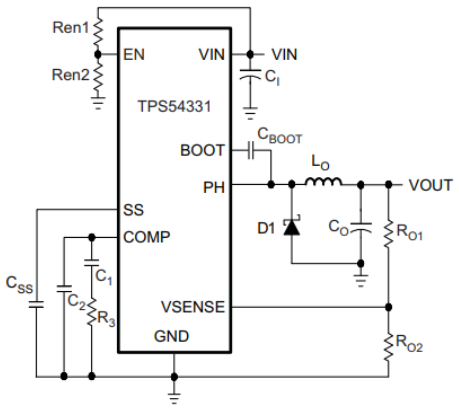

Abbildung 2.12: TPS54331 Pinout

Auf den beiden Platinen gibt es insgesamt drei Spannungen (12V, 5V, 3.3V). Mit dem Buck Converter TPS54331 werden jeweils 5V und 3.3V aus den 12V erzeugt. Da mit 5V und 3.3V keine Leistung gespeist wird, ist der Baustein mit maximal 3A Dauerstrom ausreichend dimensioniert. Abbildung 2.12 zeigt eine typische Verschaltung des Buck Converters. Die Ausgangsspannung kann über die beiden Widerstände R_{D1} und R_{D2} eingestellt werden. Das Datenblatt liefert eine Tabelle für die Dimensionierung aller passiven Bauelemente je nach Eingangs- und Ausgangsspannung. Die beiden Widerstände R_{en1} und R_{en2} am EN Eingang dienen zur Einstellung des erlaubten Spannungsbereiches für VIN und werden mit folgender Formel berechnet:

$$R_{en1} = \frac{V_{start} - V_{stop}}{3\mu A}$$

$$R_{en2} = \frac{1.25V}{\frac{V_{start}-1.25V}{R_{en1}} + 1\mu A}$$

2.3 Schaltplan

Die Schaltpläne für dieses Projekt wurden mit KiCad erstellt. KiCad ist ein freies ECAD Programm unter der GNU GPL Lizenz. Es integriert unter anderem Tools wie einen Schaltplan Editor Ëeschemaünd einen Layout Editor "PCBNew". Die beiden Schaltpläne MainPCB und LedPCB für dieses Projekt liegen als KiCad und PDF Dateien bei. Die Bauteile wurden nach den Vorgaben und Applikationsbeispielen in den Datenblättern verschaltet. Aus diesem Grund wird im Folgenden nur auf Besonderheiten eingegangen.

2.3.1 Hauptschalter und Verpolungsschutz

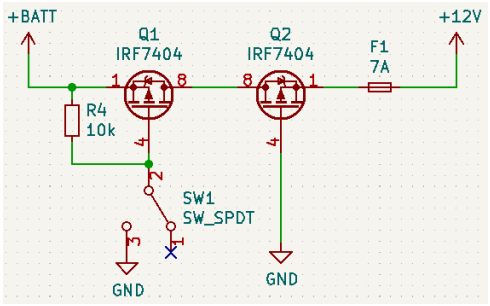

Abbildung 2.13: KiCad Hauptschalter und Verpolungsschutz

Dieser Teil ermöglicht das Anschalten über einen mechanischen Schalter. AuSSerdem wird die Platine vor Verpolung der LiIon Zellen und Überstrom/Kurzschluss geschützt. Der IRF7404 ist ein PMOS Transistor mit niedrigem Drain-Source Widerstand. Befindet sich der Schalter SW1 in der dargestellten Stellung ist $V_{GS} = 0V$ und der Mosfet sperrt. In der zweiten Schalterstellung wird $V_{GS} = 0V - V_{+BATT} \approx -12V$ und der Mosfet wird leitend. Q2 wirkt als Verpolungsschutz für die gesamte Schaltung. Bei richtiger Polung von V_{+BATT} ist die Body-Diode leitend und $V_{GS} \approx -12V$. Bei Verpolung hat das Transistor Gate das Potential 12V. $V_{GS} < 0V$ ist somit unmöglich und der Mosfet sperrt. Die Sicherung F1 schützt den Rest der Schaltung vor Überlast und Kurzschluss.

2.3.2 Eingangsquellenschutz

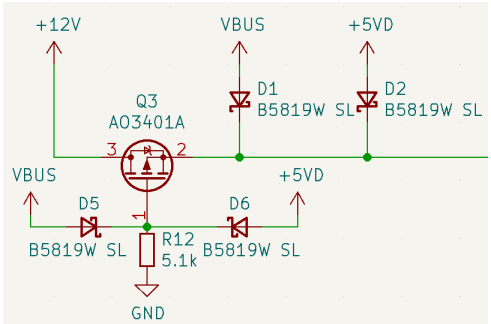

Abbildung 2.14: KiCad Eingangsquellenschutz

Die obere Schaltung befindet sich am Eingang des 3.3V Spannungswandlers. Da dieser den Mikrocontroller versorgt, muss der Buck Converter über die LiIon Akkus (12V), USB (5V) und JTAG-/Programmierinterface

(5V) betrieben werden können. Es reicht nicht aus die verschiedenen Spannungschienen direkt zu verbinden, da sonst die Gefahr eines Kurzschlusses zwischen den Potentialen besteht, wenn mehrere Spannungsquellen gleichzeitig aktiv sind. VBUS und 5VD werden durch Dioden vor Potentialausgleich geschützt. Diese Spannungen werden nur zum Debuggen verwendet. Es flieSSt nur wenig Strom und der Spannungsabfall über den Dioden ist somit hinnehmbar. Da die 12V den Roboter im Normalbetrieb mit teilweise groSSer Stromaufnahme versorgen, darf es keinen groSSen Spannungsabfall über der schützenden Komponente geben. Der PMOS Transistor AO3401 hat einen geringen Drain-Source Widerstand und somit wenig Leistungsverlust bei groSSem Strom. Sind 12V verbunden schaltet Q3 immer durch da $V_{GS} < 0V$. Falls nur VBUS oder 5VD aktiv ist, wird $V_{GS} = 0V$. Die 12V Schiene bleibt somit Spannungsfrei.

2.3.3 Reset- und Bootmodeschaltung

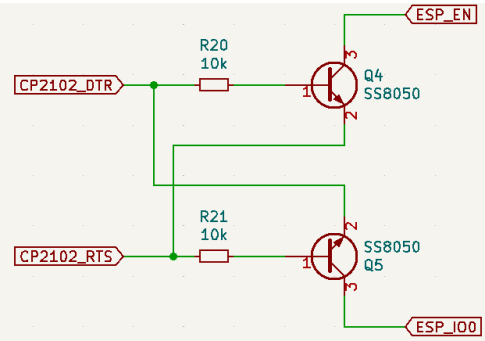

Abbildung 2.15: KiCad Reset- und Bootmodeschaltung

In 2.2.2 wurde gezeigt, dass es möglich ist den ESP32 über das DTR und RTS Signal neuzustarten und den Bootmodus zu wechseln. Die obere Schaltung setzt die hierfür notwendige logische Verknüfung zwischen DTR/RTS und EN/GPIO0 2.2 um. Die beiden SS8050 sind npn-Bipolartransistoren. Diese Schaltung ist parallel geschaltet zu der Möglichkeit ESP_EN und ESP_IO0 über einen Taster auf GND zu ziehen.

2.4 Layout

Das Layout wurde mit dem Tool PCBNew aus KiCad erstellt. Die MainPCB Platine 2.16 wurde als 4 Layer PCB umgesetzt (1. Schicht: Signal, 2.: 3.3V, 3.: 12V, 4.: GND) und LedPCB 2.17 als 2 Layer (1.: 12V/Signal, 2.: GND). Bei der Erstellung wurde versucht Angaben in den Datenblättern der Bauteile sowie "Good Practices"des Leiterplattenentwurf zu befolgen. Im Folgenden wird somit nicht weiter darauf eingegangen.

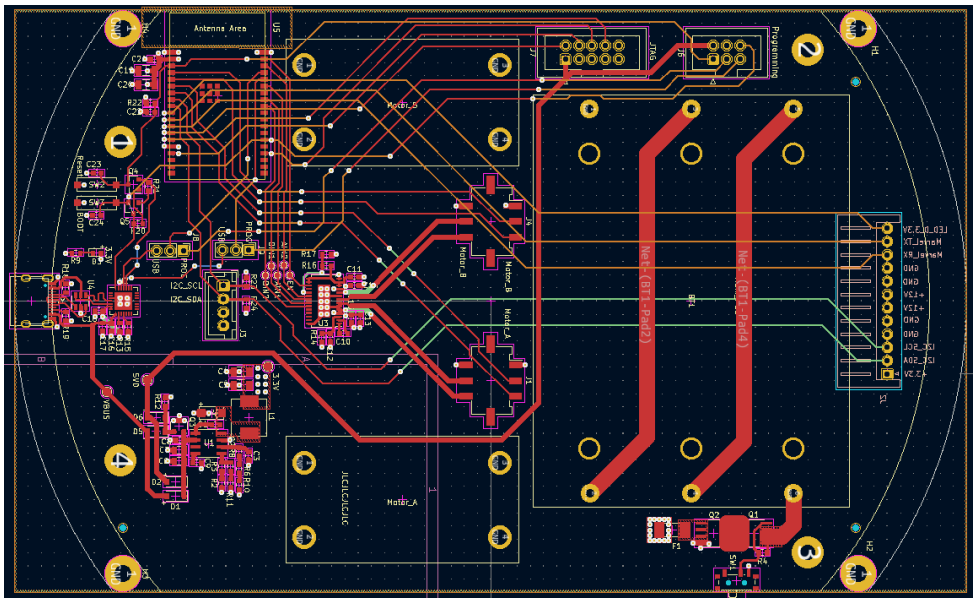

Abbildung 2.16: KiCad MainPCB Layout

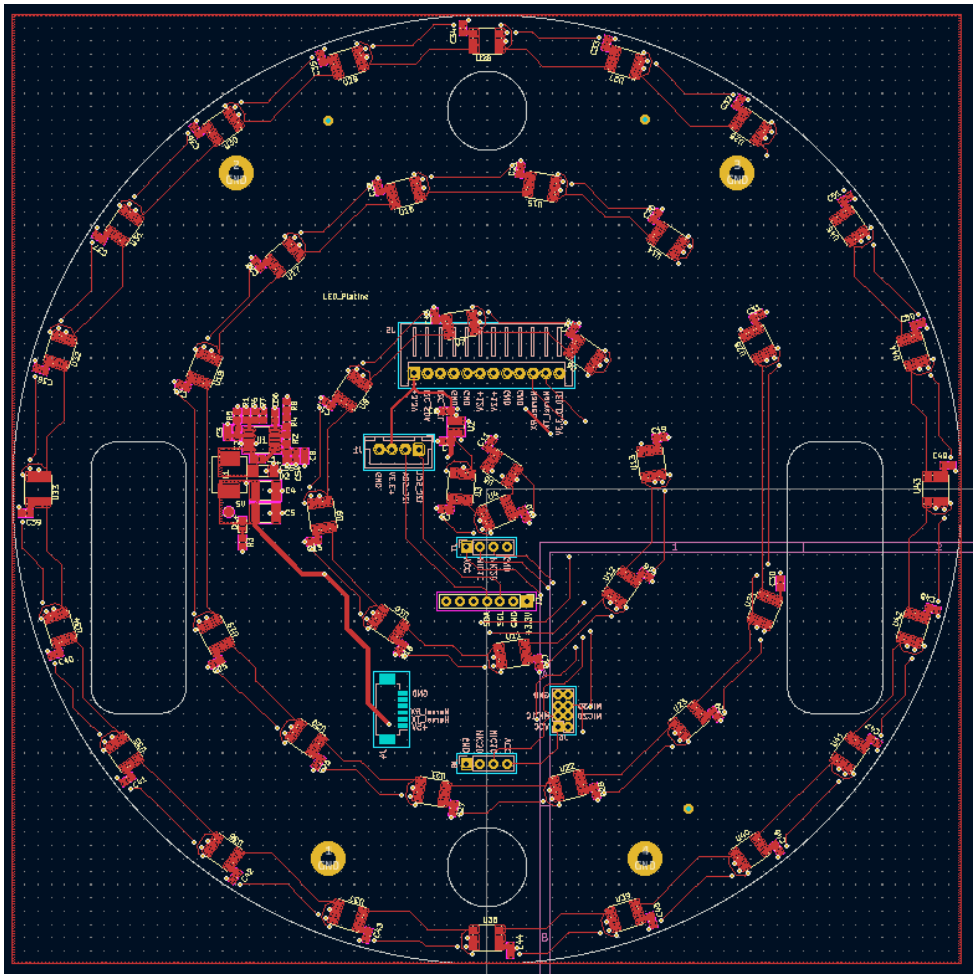

Abbildung 2.17: KiCad LedPCB Layout

2.5 Designfehler

Beim Schaltplan Design wurden leider 3 Fehler gemacht. Diese können allerdings ausgebessert werden oder beinträchtigen die Nutzung der Platine nur gering. Der Dokumentation liegen einmal das fehlerhafte KiCad Projekt und eines mit ausgebesserten Fehlern bei.

2.5.1 3.3V Spannungswandler

Fehlerbeschreibung: Der 3.3V Spannungwandler schaltet bei der Wandlung von 5V (USB, JTAG-/Programmierinterface) zu 3.3V ab. Die Wandlung von 12V zu 3.3V funktioniert.

Grund: Über den Eingang EN des TPS54331 wird ein Unterspannungsschutz (UVLO) realisiert. Fällt die Spannung an EN unter $V_{EN} = 1.25V$ schaltet der Spannungswandler ab. Wie im Kapitel 2.2.8 beschrieben lässt sich die minimal erlaubt Eingangsspannung über den Spannungsteiler R_{en1} und R_{en2} einstellen. Diese ist für 5V falsch gewählt worden.

Fehlerbehebung: Um den Unterspannungsschutz zu deaktivieren, können die Widerstände $R_{en1} = R1$ und $R_{en2} = R5 + R7$ entfernt werden.

2.5.2 CP2102N Stromversorgung

Fehlerbeschreibung: Der Mikrocontroller startet nicht richtig, wenn die Platine nur über 12V verorgt wird und USB nicht verbunden ist.

Grund: Da der CP2102N direkt über VBUS der USB Schnittstelle versorgt wird, ist der Chip nicht aktiv wenn nur die 12V Versorgung vorhanden ist. Die Ausgangssignale RTS und DTR sind somit "floatend/undefiniert und können beim Einschalten der 12V einen Start in den Download Boot-Modus triggern.

Fehlerbehebung: Damit die Ausgänge RTS und DTR immer auf definiertem Pegel sind, muss der CP2102N dauerhaft über die 3.3V versorgt sein. Dazu muss REGIN mit 3.3V verbunden werden. Da der interne 5V zu 3.3V Regulator nicht mehr genutzt wird, kann das Label "CP2102N_VDD"2.18 auch direkt mit 3.3V verbunden werden. Auf den bestehenden Platinen kann der Fehler behoben werden indem der CP210N Chip abgelötet wird und die Leiterspur zwischen VBUS und REGIN aufgetrennt wird. (Abbildung 2.19 grün) AnschlieSSend muss REGIN mit VDD verbunden werden (Lötbrücke) und mithilfe von z.B. einem Kupferdraht mit 3.3V versorgt werden. (Abbildung 2.19 blau)

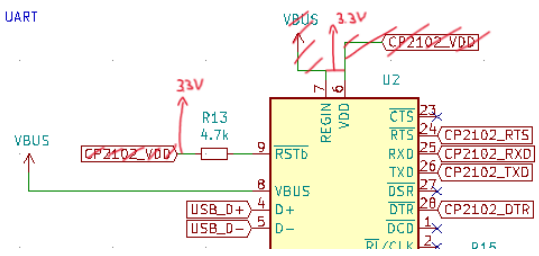

Abbildung 2.18: CP2102N Fehlerkorrektur

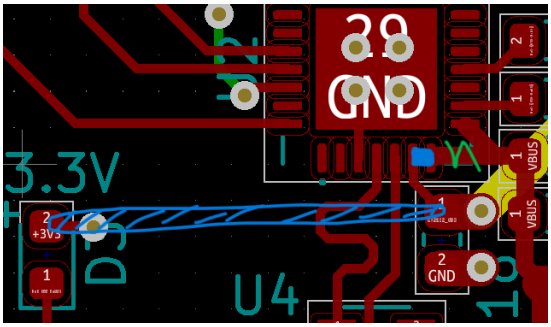

Abbildung 2.19: CP2102N Korrektur Layout

2.5.3 USB-C Anschluss

Fehlerbeschreibung: Die Signalübertragung über USB-C funktioniert nur in einer Steckrichtung.

Grund: Die Pins B6 und B7 wurden nicht mit D+/D- auf der Platine verbunden.

Fehlerbehebung: Da die Lötstellen an der USB Buchse sehr klein sind, ist es nur schwer möglich eine Brücke zu löten.

2.6 Verbesserungen

In zukünftigen Projektarbeiten wird die Platine wahrscheinlich weiter entwickelt. Im Folgenden sollen deshalb mögliche Verbesserungsvorschläge aufgelistet werden.

- Da das Ziel des Gesamtprojektes eine Roboterformation mit möglichst vielen einzelnen Robotern ist, muss zukünftig die Ladeinfrastruktur möglichst einfach gehalten werden. Aus diesem Grund ist es sinnvoll die Ladeelektronik für die LiIon Zellen direkt in die Platine zu integrieren. Über USB-C Power Delivery könnten die Roboter dann sehr einfach geladen werden.
- Momentan ist das Main_PCB als 4 Layer Platine ausgelegt. Um die kosten der einzelnen Platine zu senken, sollte das Layout auf 2 Layer reduziert werden.
- Die Pinbelegung 2.2 des ESP32 zeigt, dass es kaum mehr freie Pins am Mikrocontroller gibt. Die Regelung der Motoren belegt alleine neun dieser Pins. Diese könnte auf einen zweiten/sekundären Mikrocontroller ausgelagert werden. Somit würden wieder mehr Pins am ESP32 nutzbar sein. AuSSerdem wird die ESP32 Software-Architektur vereinfacht und Rechenleistung für zukünftige Software Erweiterungen frei.

Kapitel 3

3D-Roboterdesign

In this chapter, we're actually using some code!

```
x = 1
if x == 1:
    # indented four spaces
    print("x is 1.")
```

Listing 3.1: This is an example of inline listing

You can also include listings from a file directly:

Kapitel 4

Systemarchitektur

4.1 Überblick

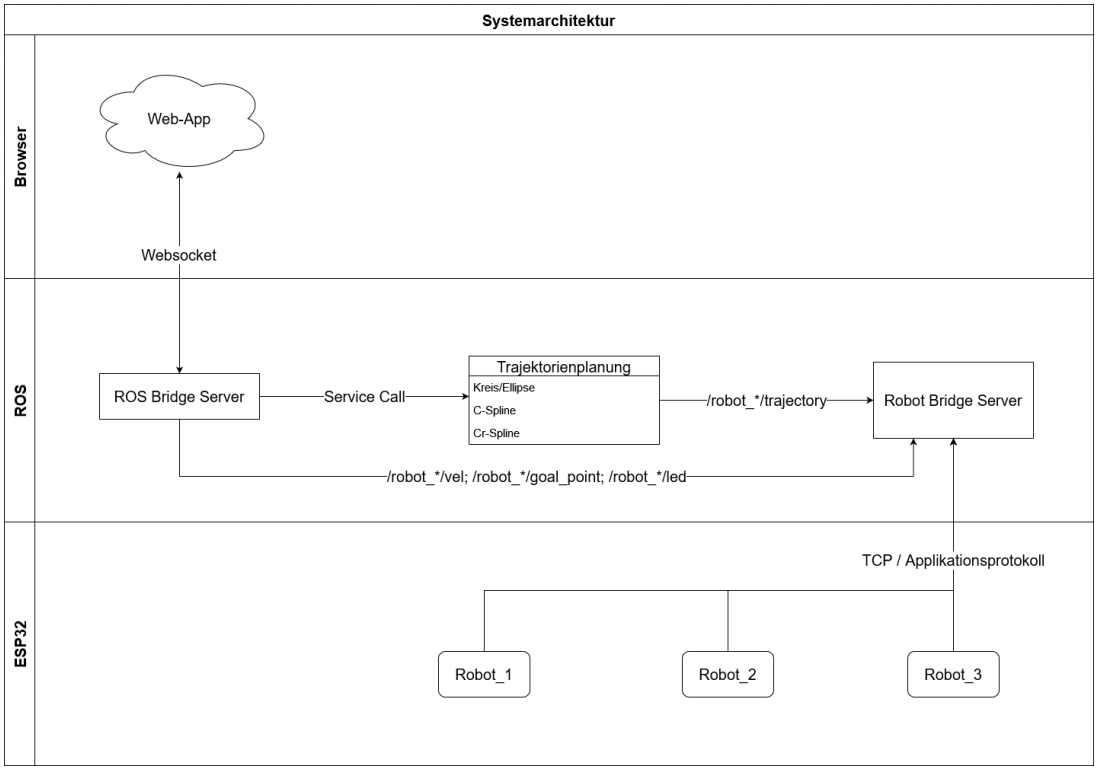

Abbildung 4.1: Systemarchitektur

Dieses Kapitel soll einen Überblick über den Aufbau und die Kommunikation zwischen den Software Modulen in diesem Projekt geben. Für genauere Informationen wie z.B. zur Implementierung wird auf das Kapitel 5 und Kapitel 6 verwiesen. Die Abbildung 4.1 zeigt, dass sich das Gesamtsystem in drei Teile aufteilen lässt. Diese unterscheiden sich durch den Ausführungsort der jeweiligen Software.

Jeder Roboter wird durch einen ESP32 Mikrocontroller gesteuert. Die Firmware auf den Robotern ist identisch bis auf eine Stringkonstante im Flash des Mikrocontrollers. Dieser Name des Roboters dient als eindeutiges Identifikationsmerkmal für die oberen Softwareschichten. Über die Wifi Schnittstelle verbindet

sich der Mikrocontroller mit einem WLAN Access Point und kann somit eine TCP Verbindung mit einem Rechner im Netzwerk aufbauen.

Dieser TCP Server ist eine ROS Node und stellt über ein eigenes Applikationsprotokoll ein Interface zwischen ROS Topics und ESP32 bereit. Mit der entsprechenden TCP Client Task auf dem ESP32 wirkt es so als wäre der Mikrocontroller direkt in das ROS Netzwerk eingebunden. Es werden somit Roboterposition "gepublishtünd Topics wie Robotergeschwindigkeit, Zielpunkt, LED-Farben/Muster und Trajektorien durch den Roboter ßubscribed". Der Hintergedanke zur Verwendung des ROS Frameworks ist die Auslagerung der rechenleistungsintensiven Trajektoriengenerierung von Mikrocontroller auf einen leistungsstarken Rechner. Der Roboterschwarm wird somit zentral durch ROS koordiniert. Während in diesem Projekt nur einfache Trajektorien unabhängig von den einzelnen Roboterpositionen generiert werden können, sollte dies zukünftig durch eine komplexe Multiroboterpfadplanung ersetzt werden. Die ROS Nodes tauschen sich über Topics aus. Damit zwischen den einzelnen Roboter unterschieden werden kann, besteht der Topicname aus einem Namespace und der Topicbezeichung:

/namespace/topicbezeichnung

Der Namespace ist die Identifikationskonstante im Flash des ESP32 und wurde meist auf *robot_* + Nummerierung gesetzt.

Damit das Gesamtsystem über eine benutzerfreundliche Schnittstelle einfach bedient werden kann, wurde eine Webb-App entwickelt. Die Webb-App läuft über den Browser auf verschiedenen Endgeräten und kommuniziert über das Websocket Protokoll mit einer weiteren ROS Node. Ursprünglich war geplant die Trajektoriengenerierung über ROS Service Calls aus der Webbapp zu steuern. Dies konnte allerdings nicht mehr umgesetzt werden. Das gleiche gilt für das Anzeigen der Roboterpose und setzten von Zielpunkten. Zum Ende der Projektarbeit ist es möglich die Robotergeschwindigkeit und LED Muster/Farben über das Web Interface zu steuern.

4.2 Applikationsprotokoll

Für die Kommunikation zwischen Mikrocontroller und Host Rechner wurde ein eigenes Applikationsprotokoll entwickelt. Dieses soll als virtuelles Interface zwischen ROS und dem ESP32 wirken damit diese mithilfe von ROS Topics miteinander kommunizieren können. Für den Entwickler soll sich somit das SSubscribenünd Ädvertisen"von Topics auf dem Mikrocontroller anfühlen wie in einer normalen ROS Applikation. Eine wichtige Hauptaufgabe des Protokolls im Projekt ist allerdings die Übertragung der Trajektorien über die Topic /robot_*/trajectory".

Eine wichtige Überlegung für den Aufbau des Protokolls war die Entscheidungs zwischen UDP oder TCP als Transportunterschicht. Das User Datagram Protokoll ist ein verbindungsloses Protokoll. Es garantiert somit nicht die sichere, verlustfreie Übertragung der Nutzdaten. Da es zu keinen Neuübertragungen kommt, hat es geringe Latenzzeiten und eignet sich somit super zur Übertragung von Echtezeitdaten wie die Position des Roboters. Die Hauptaufgabe des Protokolls in diesem Projekt ist die Übertragung von Kilobyte groSSen Trajektorien. Diese erfordert eine zuverlässige Verbindung ohne Datenverluste. Desweiteren ist eine Übertragung der Daten in Echtezeit gut, aber nicht unbedingt erforderlich. Das Transmission Control Protocol baut eine virtuelle Verbindung zwischen Server und Client auf und garantiert somit eine verlustfreie Übertragung der Daten. Aus diesem Grund wurde TCP als Transportunterschicht für das eigene Protokoll verwendet. In diesem Kapitel wird der Aufbau des Protokolls beschrieben. Für die genaue Implementierung der Server und Client Seite wird auf die Kapitel 5 und 6 verwiesen.

4.2.1 Datenpakete

Die Kommunikation mit dem Applikationsprotokoll ist Paketbasierend. Diese müssen von der Software im Server oder Client aus dem TCP Datenstrom gefiltert werden. Das Protokoll definiert insgesamt 5 verschiedene Pakete:

ID	Paketname
0x01	Initialisierungspaket
0x02	Advertise Paket
0x03	Subscribe Paket
0x04	Keep-Alive Paket
0x05	Publish Paket

Tabelle 4.1: Applikationsprotokoll Paketarten

Jedes Paket beginnt mit der jeweiligen Identifikationsnummer. Der restliche Aufbau ist Paketabhängig. Im Folgenden der Aufbau und der Nutzen der unterschiedlichen Pakete erklärt werden.

4.2.1.1 Initialisierungspaket

Das Initialisierungspaket ist das erste Paket, das nach Aufbau der TCP Verbindung vom Client an den Server geschickt wird. Dabei überträgt es den Robotернamen, der von der ROS Server Node als Topic Namespace genutzt wird. Das Initialisierungspaket wird nur vom Client an den Server geschickt werden.

0x01	Robotername	'\0'
1 Byte	x Bytes	1 Byte

4.2.1.2 Advertise Paket

Das Advertise Paket entspricht dem Aufruf der ROSCPP Methode advertise(). Dieses Paket kann nur vom Client an den Server geschickt werden. Dabei fordert dieser den Server dazu auf die übermittelte Topic mit dem übermittelten Nachrichtentyp in ROS zu ädvertisen". Erst nach dem Advertise Paket dürfen Nachrichten von der entsprechenden Topic mit dem Publish Paket verschickt werden.

0x02	Topic Name	'\0'	Nachrichtentyp	'\0'
1 Byte	max. 32 Bytes	1 Byte	max. 32 Bytes	1 Byte

4.2.1.3 Subscribe Paket

Ähnlich wie das Advertise Paket teilt das Subscribe Paket dem Server mit dass der Client eine Topic mit dem entsprechenden Nachrichtentyp ßubscriben"möchte. Auch das Subscribe Paket wird nur vom Client an den Server geschickt. Der Server darf erst nach dem er ein Subscribe Paket zu einer Topic empfangen hat, Publish Pakete an den Client weiterleiten.

0x03	Topic Name	'\0'	Nachrichtentyp	'\0'
1 Byte	max. 32 Bytes	1 Byte	max. 32 Bytes	1 Byte

4.2.1.4 Keep-Alive Paket

Das Keep-Alive Paket wird alle $500ms$ vom Client an den Server und vom Server an den Client geschickt. Empfängt eine der beiden Seiten kein Keep-Alive für $3000ms$ wird ein Verbindungsabbruch festgestellt. Dies ist vor allem notwendig wenn keine Nutzdaten übertragen werden und zum Beispiel der Client abstürzt. Der Server wird nach drei Sekunden feststellen, dass keine Keep Alive Pakete vom Client kommen und die TCP Verbindung schlieSSen. Zusätzlich wird ein Zeitstempel des jeweiligen Senders an den Empfänger übertragen. Dieser kann z.B. zur Zeitsynchronisation zwischen ROS und Roboter verwendet werden.

0x04	Zeitstempel in μs
1 Byte	8 Bytes (uint64_t)

4.2.1.5 Publish Paket

Das Publish Paket dient der eigentlichen Übertragung von Nutzdaten vom Client zum Server und vom Server zum Client. Durch das Feld "Topic Name"können die Daten einer durch das Subscribe- und Advertise-Paket initialisierten Topic zugeordnet werden. Der Datenteil des Pakets unterscheidet zwischen Array- und Strukturdaten.

Strukturdaten serialisieren den ROS Nachrichtentyp der jeweiligen Topic. Auf das serialisieren und deserialisieren der Nachrichten wird genauer in den Kapiteln 5 und 6 eingegangen.

0x05	Topic Name	Strukturdaten
1 Byte	max. 32 Bytes	x Bytes

Arraydaten werden in diesem Projekt für die Übertragung der Trajektorien und zur Übertragung des ROS Nachrichtentyp std_msgs::String verwendet. Nach dem Topic Namen folgt ein Feld für die jeweilige Arraylänge. Dieses beinhaltet die GröSSe des Arrays in Bytes.

0x05	Topic Name	Arraylänge in Bytes	Arraydaten
1 Byte	max. 32 Bytes	4 Bytes (int32_t)	x Bytes

4.2.2 Anwendungsbeispiel

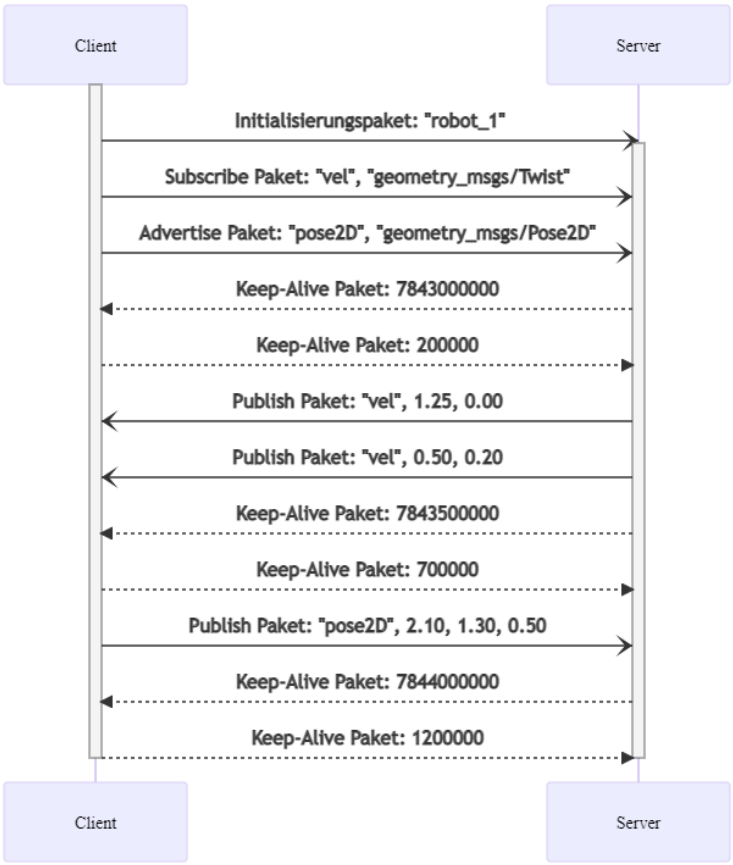

Abbildung 4.2: Applikationsprotokoll Anwendungsbeispiel

Die Abbildung 4.2 zeigt eine typische Kommunikation zwischen Client und Server. Der Client startet die Verbindung mit dem Initialisierungspaket und teilt dem Server mit welche Topics er ßubscribenünd ädvertisen"will. Alle $500ms$ tauschen beide Seiten ihr Keep-Alive Paket mit dem jeweiligen Zeitstempel aus. Daten zu den Topics werden mithilfe des Publish Pakets vom Client zum Server oder vom Server zum Client geschickt. Ein Verbindungsabbruch würde durch ein fehlendes Keep-Alive bemerkt werden. Das Verhalten beider Seiten in diesem Fall wird in den Kapiteln 5 und 6 beschrieben.

Kapitel 5

Softwarearchitektur auf dem ESP32

5.1 ESP-IDF

Das Espressife IOT Development Framework kurz ESP-IDF ist das offizielle Tool zur Einrichtung und Programmierung des ESP32. Das Open Source Framework beinhaltet unter anderem Bibliotheken zur Hardwareabstraktion, ein Konfigurationssystem, es integriert das Echtzeitbetriebssystem FreeRTOS und ermöglicht über Automatisierungsskripte Compilieren und Flashen von Binaries. Der ESP32 lässt sich mit C oder C++ programmieren. Compilieren und Linken der Software wird von dem Build System CMake übernommen. Die Installation der ESP-IDF wird im "Programming Guide"ESP von Espressife beschrieben. Das Projekt wurde mit der Version v4.4 und v5.0 getestet.

Damit die Peripherie des ESP32 nicht über Register angesteuert werden muss, bietet Espressife eine Hardwareabstraktionsschicht. Diese erleichtert die Initialisierung und Steuerung von Hardwaremodulen wie zum Beispiel GPIO, UART, I2C, etc. über Funktionsaufrufe.

Das Konfigurationssystem von Espressife basiert auf KConfig. Mit dem Befehl *idf.py menuconfig* wird ein Konfigurationsmenü gestartet. Hier können z.b. hardwarespezifische Einstellungen für den ESP32 gemacht werden. Es ist allerdings auch möglich eigene Einstellungspunkte über die KConfig Syntax zu erstellen. Im Projekt sind zum Beispiel Einstellungspunkte für Wifi SSID, Password und Server IP Addresse definiert. Die Konfigurationen werden in der Datei sdkconfig gespeichert. Beim Bauen wird diese in eine Header Datei umgewandelt. Durch Inkludieren der Datei sdkconfig.h werden die Einstellungen über "definesïn die C/C++ Programme eingebunden.

Die ESP-IDF ermöglicht die Unterteilung der Software in Komponenten. Dabei werden die Sourcen in einzelne Ordner unterteilt. Dies erhöht die Projektstruktur und Wiederverwendbarkeit des Codes. Jedes neu erstellte ESP-IDF Projekt enthält eine standard "main"Komponente diese beinhaltet die main() Funktion des Programms. Neue Komponenten können mit dem Befehl *idf.py -C components create-component <Komponentenname>* erstellt werden.

Mit Hilfe der von Espressife bereitgestellten Automatisierungsskripte können zum Beispiel Binaries gebaut und geflasht werden. Die Skripte werden mit Befehlen für die Kommandozeile gestartet. Die folgende Auflistung enthält die für dieses Projekt wichtigsten Befehle:

- *idf.py build* Compiliert und Linkt alle Binaries
- *idf.py flash* Flasht die Binaries auf den Mikrocontroller
- *idf.py monitor* Zeigt die Ausgabe der seriellen Schnittstelle

- *idf.py menuconfig* Öffnet das Konfigurationsmenü
- *idf.py -C components create-component <Komponentenname>* Erstellt neue Komponente

FreeRTOS ist ein Echtzeitbetriebssystem für Mikrocontroller. Ähnlich wie ein normales Betriebssystem arbeitet es mit Tasks, so dass mehrere Programmzweige parallel ablaufen können. Die ESP-IDF Software portiert FreeRTOS auf den ESP32. Neben den Tasks implementiert FreeRTOS auch andere Konzepte wie "Queues", "Message Buffer", SSemaphorsünd "Mutexes". Diese dienen unter anderem der Synchronisierung von Prozessen und verhindern "Race Conditions"beim Zugriff auf geteilte Ressourcen.

5.2 Überblick

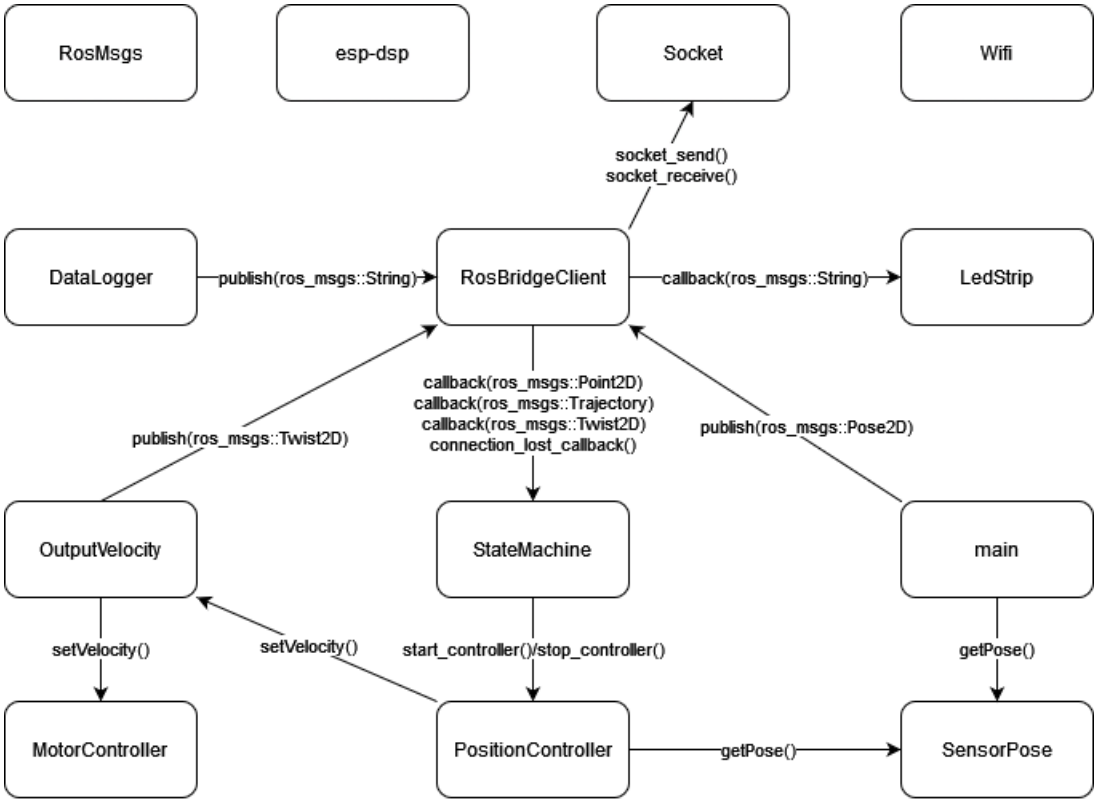

Abbildung 5.1: Überblick ESP32 Architektur

Abbildung 5.1 zeigt alle Systemkomponenten des Projekts. Diese Kommunizieren gröSStenteils über Funktionsaufrufe. Die Pfeilrichtung der dargestellten Aufrufe zeigt immer von der Aufrufenden zur aufgerufenen Komponente. Der Programmeinstieg befindet sich in der main Komponente. Hier werden alle anderen Komponenten initialisiert. In RosBridgeClient befindet sich die Ablaufsteuerung des Applikationsprotokolls zum Topicaustausch mit dem ROS-System. Die Daten hierfür werden über Socket empfangen und gesendet. Empfängt der RosBridgeClient Daten auf der Geschwindigkeits, Trajektorien oder Zielpunkt Topic, wird über einen Callback die Komponente StateMachine informiert. Die Zustandsmaschine versetzt den Roboter basierend auf dem aktuellen Zustand und dem jeweiligen Aufruf in einen neuen Zustand und aktiviert z.B. für einen empfangenen Zielpunkt oder einer Trajektorie den PositionController. Diese Komponente regelt die Position des Roboters und kann somit einer Trajektorie folgen oder einen Zielpunkt anfahren. Hierfür besorgt sie sich die Roboterpose aus der Komponente SensorPose und gibt

zyklisch einen neuen Sollgeschwindigkeitsvektor an OutputVelocity weiter. OutputVelocity transformiert den Geschwindigkeitsvektor in Solldrehzahlen für die beiden Motoren. Diese werden dann in MotorController ausgeregelt. Optional kann der Roboter auch in einen Simulationsmodus versetzt werden. Hierbei wird der Sollgeschwindigkeitsvektor an ROS gesendet. AuSSerdem erhält die Komponente SensorPose eine Pose von ROS und nicht der Sensorik auf dem Roboter. Die Komponente Wifi übernimmt den Verbindungsauf- und abbau mit dem verwendeten Wlan Access Point. LedStrip steurt das Led-Array auf dem Roboter. Mithilfe von DataLogger können Programmlogs erstellt und in das ROS-System gesendet werden. RosMsgs definiert die Datentypen der Topics für die Kommunikation mit ROS. Die Komponente esp-dsp ist eine von Espressife bereitgestellte Bibliothek zur digitalen Signalverarbeitung und stellt für dieses Projekt vor allem Funktionen zur Matrizenrechnung bereit.

5.3 Wifi

Die Komponente Wifi konfiguriert den ESP32 als WLAN Station und steuert Verbindungsaufbau sowie das Verhalten bei Verbindungsabbruch. Dies wird in den Methoden der gleichnamigen Singleton Klasse umgesetzt. Zunächst wird die Wifi und ESP-NETIF Bibliothek von Espressife initialisiert und anschlieSSend für bestimmte Wifi Events die Event Loop Library konfiguriert. Dabei handelt es sich um eine Zustandsmaschine, die bei Events wie z.B. "Verbindung hergestelltöder "Verbindung abgebrochenëine Callback-Funktion im User Code ausführt. Hier kann das entsprechende Verhalten für die Events vom Programmierer gesteuert werden. Nach dem Initialisieren und Starten der Wifi Klasse über `init()` und `begin()` interagiert die Komponente nicht mehr mit anderen Komponenten. Das steuern der Event Zustandsmaschine übernimmt ein interner Prozess der Espressife Bibliotheken. Die Komponente Wifi beinhaltet drei Einstellungspunkte im KConfig Menü:

- `Wifi SSID` SSID des Wlan Routers
- `WIFI Password` Passwort des Wlan Routers
- `WIFI_MAX_RETRY` Anzahl der Verbindungsversuche bei fehlgeschlagenem Verbindungsaufbau

5.4 Socket

Die Komponente Socket dient als Abstraktionsschicht der POSIX Socket API, die von Espressife zum Aufbau einer TCP Verbindung genutzt wird. Der ESP32 agiert dabei als Client und Verbindet sich mit einer ROS-Node die den Server darstellt. Eine Verbindung wird über ein Objekt der Klasse Socket gekapselt. Über die folgenden Methoden kann eine Verbindung auf- und abgebaut werden, sowie Daten empfangen und versendet werden:

- `Socket(int port, std::string ip_addr)` Dem Konstruktor müssen der Port sowie die IP-Adresse des Servers übergeben werden.
- Über `void connect_socket()` wird eine Verbindung mit dem Server aufgebaut. Die Methode blockiert solange bis erfolgreich eine Verbindung aufgebaut werden konnte. Ist dies nicht der Fall wird versucht durch Abbau und erneutem Aufbau eine Verbindungs herzustellen. Des Weiteren konfiguriert die Methode die POSIX Socket API als Ö_NONBLOCK". Dies verhindert ein Blockieren beim Aufruf der POSIX Funktionen `recv()` und `send()`, da das Verhalten beim Senden und Empfangen durch die eigene Socket Klasse geregelt wird.

- `void disconnect_socket()` schlieSSt die TCP-Verbindung.
- `int socket_receive(uint8_t* rx_buffer, int recv_bytes)` empfängt die mit `recv_bytes` übergebene Anzahl an Bytes aus dem internen TCP-Puffer. Die Methode blockiert solange bis alle Bytes vollständig empfangen wurden oder ein Fehler beim Emfangen auftritt. Es wird die Anzahl empfangener Bytes oder bei einem Fehler das Makro `SOCKET_FAIL` zurück gegeben.
- `int socket_receive_string(std::string& rx_string, int max_bytes)` dient zum Empfangen von mit `'\0'` beendeten Zeichenketten. Die Methode blockiert solange bis eine vollständige Zeichenkette empfangen wurde, `max_bytes` aus dem Puffer entnommen wurden oder ein Fehler beim Empfangen auftritt. Es wird die Anzahl empfangener Bytes oder bei einem Fehler das Makro `SOCKET_FAIL` zurück gegeben.
- `int socket_receive_nonblock(uint8_t* rx_buffer, int recv_bytes)` empfängt die mit `recv_bytes` übergebene Anzahl an Bytes aus dem internen TCP-Puffer. Falls nicht genügend Bytes vorhanden sind oder ein Fehler beim Empfangen auftritt, endet die Methode sofort. Es wird die Anzahl empfangener Bytes oder bei einem Fehler das Makro `SOCKET_FAIL` zurück gegeben.
- Mit `int socket_send(uint8_t const* tx_buffer, int buffer_len)` werden `buffer_len` Bytes an den Server gesendet. Paralleles senden durch mehrere Prozesse könnte zu Überlagerung der Daten führen. Aus diesem Grund verhindert ein Mutex paralleles senden. Rufen mehrere Prozesse gleichzeitig `socket_send()` auf, darf der Erste senden, für die Anderen endet die Methode. Darf ein Prozess senden, wird solange blockiert bis alle Bytes gesendet wurden oder ein Fehler auftritt. Es wird die Anzahl gesendeter Bytes oder bei einem Fehler das Makro `SOCKET_FAIL` zurück gegeben.

5.5 RosBridgeClient

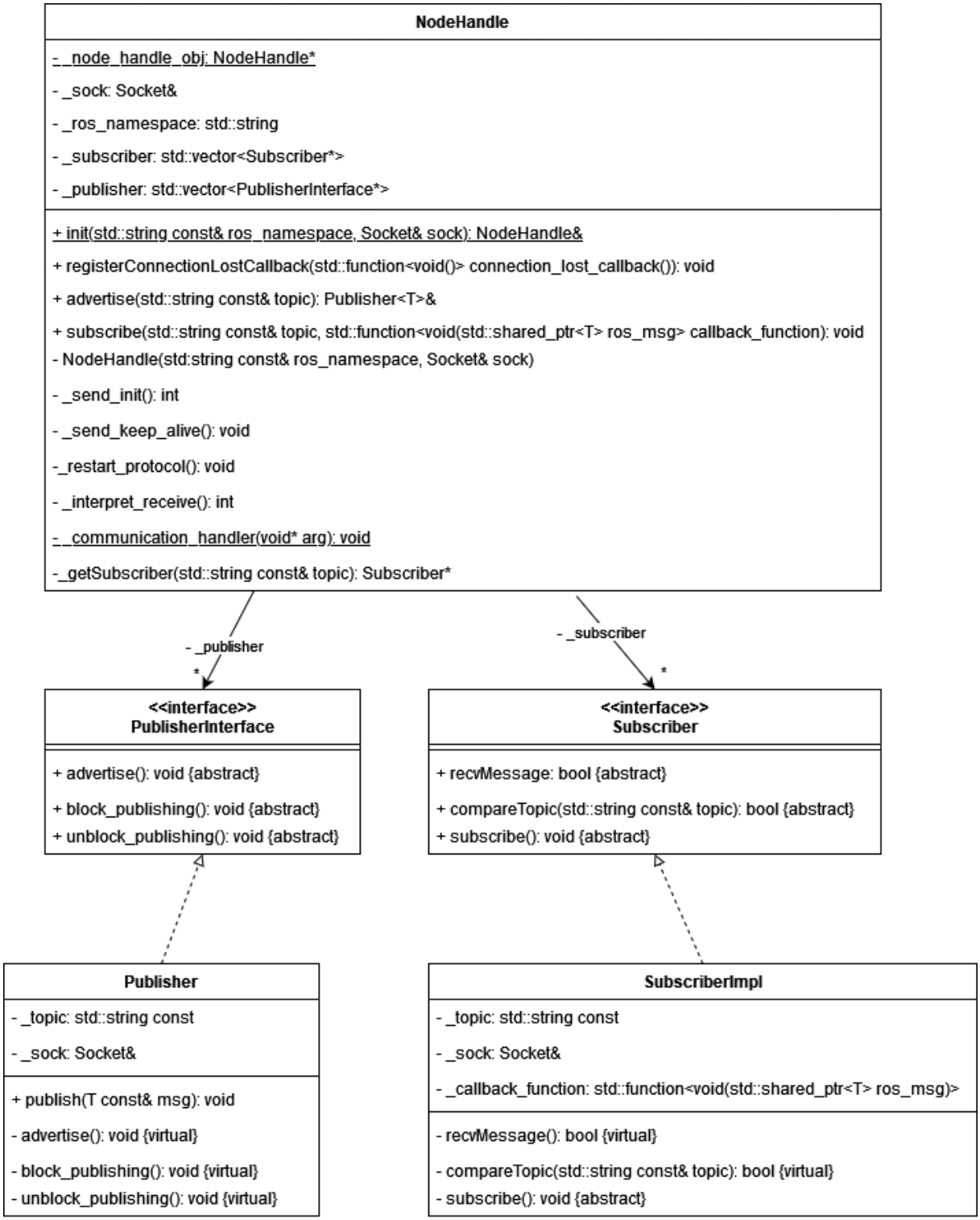

Abbildung 5.2: Klassendiagramm RosBridgeClient

Die Komponente RosBridgeClient steuert das Applikationsprotokolls zur Kommunikation mit ROS. Ein Ziel des Applikationsprotokolls ist es die TCP Kommunikation zu abstrahieren. Damit wirkt es als wäre der ESP32 wie eine ROS-Node in das System eingebunden. Aus diesem Grund sind Klassen- und Funktionsnamen, sowie die Funktionsweise der Programmierschnittstelle stark an ROS orientiert. Das Klassendiagramm in Abbildung 5.2 zeigt den Aufbau von RosBridgeClient.

5.5.1 Publisher und SubscriberImpl

Für jede im Protokoll verwendete Topic wird ein Objekt der Klasse `Publisher` oder `SubscriberImpl` erstellt. Über Objekte der Klasse `Publisher` kann der Programmierer Nachrichten auf der jeweiligen Topic verschicken. Objekte vom Typ `SubscriberImpl` werden nur intern von NodeHandle zum empfangen von Topics verwendet. Empfängt SubscribeImpl eine Nachricht wird das restliche Programm über eine Callback Funktion informiert. Beide Klassen sind parametrierbar mit dem Nachrichtentyp der jeweiligen Topic. Die Schnittstellen `PublisherInterface` und `Subscriber` dienen zum speichern der verschieden parametrierten Objekte in NodeHandle. `Publisher` oder `SubscriberImpl` haben beide einen privaten Konstruktor. Es ist somit nur möglich Objekte über die Methoden `advertise()` und `subscribe()` der Freundesklasse NodeHandle zu erstellen.

Im Folgenden wird auf die Methoden der Klasse `Publisher` genauer eingegangen:

- `Publisher(std::string const& topic, Socket& sock)` ist der Konstruktor der Klasse.
- `void advertise()` sendet das Advertise Paket über Socket an den ROS-Server. Die Methode wird von der namensgleichen Funktion aus NodeHandle oder beim Neustart des Protokolls aufgerufen.
- Mit `void publish(T const& msg)` kann der Programmierer Nachrichten auf der jeweiligen Topic an den ROS-Server versenden. Der Parameter `T` ist der Nachrichtentyp des Publish Pakets.
- Die Methoden `void block_publishing()` und `void unblock_publishing()` werden beim Neustart des Protokolls genutzt, um Publish Pakete aus anderen Teilen/Prozessen des Programms zu verhinden.

Im Folgenden wird auf die Methoden der Klasse `SubscriberImpl` eingegangen:

- `SubscriberImpl(std::string const& topic, Socket& sock, std::function<void(std::shared_ptr<T> ros` ist der Konstruktur der Klasse. Dem Konstruktor wird ein Zeiger auf eine Callback-Funktion übergeben. Diese wird nach dem Empfangen einer Nachricht auf der jeweiligen Topic ausgeführt.
- `void subscribe()` sendet das Subscribe Paket an den ROS-Server. Die Methode wird von der namensgleichen Methode aus NodeHandle oder beim Neustart des Protokolls ausgeführt.
- `void recvMessage()` interpretiert den Datenteil eines empfangenen Publish-Pakets. Die Methode unterscheidet, ob es sich um Arraydaten oder reine Strukturdaten handelt. AnschlieSSend werden die Daten deserialisiert und die enstprechende Callback-Funktion aufgerufen.
- `bool compareTopic(std::string const& topic)` vergleicht den übergebenen Topicnamen mit dem Internen. Die Methode wird von NodeHandle verwendet, um für ein empfangenes Publish-Paket den richtigen Subscriber zu finden.

5.5.2 NodeHandle

NodeHandle ist eine Singleton Klasse. Es existiert somit nur ein NodeHandle Objekt im Programm. Dieses wird beim Aufruf von init() einmalig erstellt. Parallel dazu wird das Initialisierungspaket an den ROS-Server über die Methode `_send_init()` versendet. AnschlieSSend startet der Hauptprozess `_communication_handler`. Dieser läuft parallel zum restlichen Programm. Die in der Abbildung 5.3 gezeigte Schleife wird vom Hauptprozess alle $50ms$ ausgeführt. Dabei wird zyklisch alle $500ms$ ein Keep-Alive Paket versendet, Daten aus dem Socket Eingangspuffer ausgewertet und überprüft ob das letzte empfangene Keep-Alive Paket älter als $3000ms$ ist. Falls das letzte Keep-Alive älter ist oder beim empfangen und interpretieren der Daten etwas schief gelaufen ist, wird das Protokoll neu gestartet.

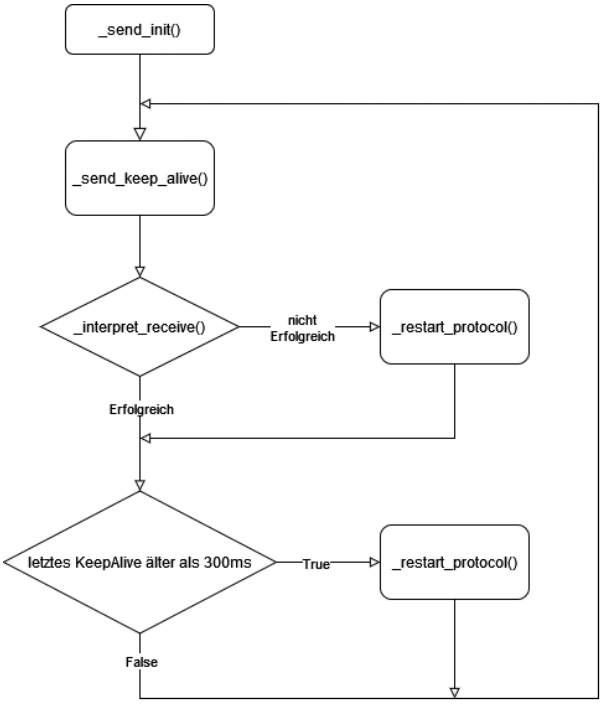

Abbildung 5.3: Ablaufdiagramm NodeHandle

Im Folgenden wird genauer auf die Methoden aus NodeHandle eingegangen:

- `static NodeHandle& init(std::string const& ros_namespace, Socket& sock)` erzeugt beim ersten Aufruf einmalig ein NodeHandle Objekt. Der String `ros_namespace` ist der Robotername, der über das Initialisierungspaket an den ROS-Server gesendet wird.
- Mit `Publisher<T>& advertise(std::string const& topic)` wird ein neues Publisher Objekt erzeugt. Dieses wird als Zeiger in NodeHandle abgespeichert. AuSSerdem wird ein Advertise Paket an den ROS Server gesendet.
- Mit `void subscribe(std::string const& topic, std::function<void(std::shared_ptr<T> ros_msg)> cal` wird ein neues SubscriberImpl Objekt erzeugt. Dieses wird als Zeiger in NodeHandle abgespeichert. AuSSerdem wird ein Subscribe Paket an den ROS Server gesendet.
- Der Methode `void registerConnectionLostCallback(std::function<void()> connection_lost_callback)` wird ein Funktionszeiger übergeben. Diese Funktion wird ausgeführt wenn es zu einem Fehler kommt

und das Protokoll neu gestartet wird. Dies kann z.B. dazu verwendet werden den Roboter bei Verbindungsverlust zu stoppen.

- Im Konstruktur `NodeHandle(std::string const& ros_namespace, Socket& sock)` wird über `_send_init()` das Initialisierungspaket versendet und es wird der Hauptprozess `_communication_handler()` erstellt.
- `static void _communication_handler(void* arg)` ist der Hauptprozess von NodeHandle.
- `int _send_init()` sendet das Initialisierungspaket an den ROS-Server.
- `void _send_keep_alive()` sendet das Keep-Alive Paket an den ROS-Server.
- `int _interpret_receive()` interpretiert die empfangenen Daten aus Socket. Hierfür wird zunächst basierend auf dem ersten Message ID Byte entschieden, ob ein KeepAlive oder ein Publish Paket empfangen wurde. Falls ein Keep-Alive Paket empfangen wurde, wird der übertragene Zeitstempel aus dem Socket Puffer entnommen und abgespeichert. Falls ein PublishPaket empfangen wurde, wird basierend auf dem übertragenen Topicnamen mithilfe der Methode `_getSubscriber()` das richtige Subscriber Objekt gesucht und mit `recvMessage()` der Datenteil interpretiert. Läuft beim empfangen oder interpretieren etwas schief gibt die Methode das Makro `SOCKET_FAIL` zurück.
- `Subscriber* _getSubscriber(std::string const& topic)` gibt das jeweilige Subscriber Objekt zu der Topic zurück.
- `void _restart_protocol()` wird dazu verwendet das Protokoll neu zu starten. Bei einem Neustart wird zunächst das Publishen für alle Publisherobjekte blockiert, alle registrierten "ConnectionLost"Callbackfunktionen ausgeführt, die TCP Verbindung über disconnect und connect neu aufgebaut, ein Initialisierungspaket versendet und für alle Publisher und Subscriber ein Advertise bzw. Subscribe Paket an den ROS Server versendet. Zuletzt wird die Publishblockierung für alle Publisher aufgehoben.

5.6 RosMsgs

Die Komponente RosMsgs definiert die Nachrichtentypen mit denen das Applikationsprotokoll kommuniziert. Diese Nachrichtentypen orientieren sich stark an den entsprechenden Typen aus ROS. Die Nachrichtenklassen werden dem C++ Namensraum `ros_msgs` untergeordnet. Ein zweiter Namensraum `ros_msgs_lw` beinhaltet namensgleiche Klassen. Wobei hier ausschlieSSlich mit float im Gegensatz zu double bei `ros_msgs` gearbeitet wird. Da die Floating Point Unit des ESP32 nur den float Datentyp unterstützt, dient `ros_msgs_lw` für Berechnungen und `ros_msgs` ausschlieSSlich zur Kommunikation mit dem ROS Server. Im Applikationsprotokoll gibt es Stuktur- und Arraynachrichtentypen. Der Aufbau dieser Nachrichten ist durch die Klassen in `ros_msgs` vorgegeben. Identische Klassen extistieren auch in den Sourcen des ROS-Servers. Um einem Publisher oder Subscriber aus der Komponente RosBridgeClient einen Nachrichtentyp zu zuweisen, werden diese mit den Klassen aus `ros_msgs` parametriet. Aus dem Grund müssen alle Nachrichtenklassen die folgenden Methoden implementieren:

- `size_t getSize()` gibt die GröSSe der Daten der entsprechenden Klasse in Bytes zurück. Für Arraynachrichtentypen gibt die Methode null zurück, wenn dem Array noch keine Daten zugewiesen wurden.

- Mit `void allocateMemory(int32_t msg_len)` wird Speicherplatz für die Daten bei Arraynachrichtentypen reserviert. Für Stukturdatentypen muss nur ein leerer Methodenrumpf implementiert sein.
- `std::string getMsgType()` gibt den Namen des entsprechenden ROS Nachrichtentyps zurück.
- `void serialize(uint8_t* buffer)` wird von der Klasse Publisher aus RosBridgeClient zur Serialisierung der Daten verwendet.
- `void deserialize(uint8_t* buffer)` wird von der Klasse Subscriber aus RosBridgeClient zur Deserialisierung der Daten verwendet.

5.7 MotorController

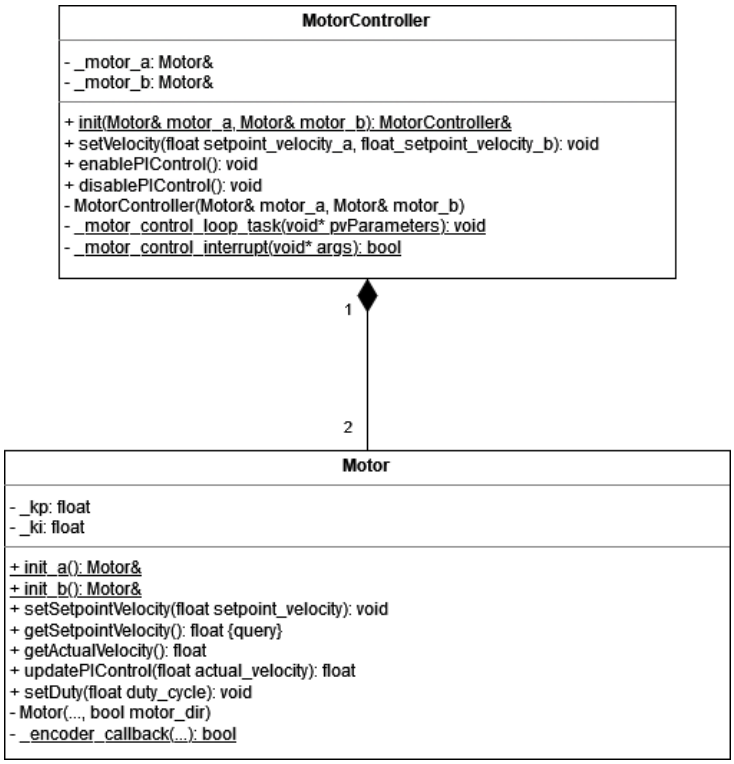

Abbildung 5.4: Klassendiagramm MotorController

Die Komponente MotorController steuert die beiden Motoren des Roboters. Die Drehzahl der Motoren wird über einen PI-Regler geregelt. Das Stellsginal ist hierbei die Pulsbreite der Ausgangssignale für den Motortreiber. In der Software haben diese einen Wertebereich von -100.0 bis 100.0 Prozent. Ein Überschreiten des Wertebereichs wird durch eine Stellbegrenzung mit Begrenzungsbeobachter verhindert. Die beiden Regelkreise laufen mit einer Widerholungsrate von $1000Hz$. Des Weiteren besitzt die Komponente einige Einstellungspunkte im KConfig Menü. Es können die Pinbelegungen für die Signale xIN1, xIN2, Enable des Motortreibers DRV8841, die Hallsensoreingängen und die PWM Frequenz des Stellsignals verändert werden. Die Standardeinstellung passt allerdings schon zur Pinbelegung der Platine. Die Komponente MotorController ist in die Klassen Motor und MotorController aufgeteilt.

5.7.1 Motor

Die Klasse Motor interagiert sehr viel mit der Hardware. Aus diesem Grund können nur zwei Objekte (Motor_a, Motor_b) für jeweils einen Motor erzeugt werden. Der ESP32 besitzt ein Hardwaremodul "MCPWM_UNITßur Steuerung von Motoren. Dieses Hardwaremodul hat mehrere Timer Kanäle über die ein PWM Signal zur Steuerung der IN1 oder IN2 Eingänge des Motortreibers erzeugt werden kann. Zur Erfassung der Motordrehzahl besitzt die "MCPWM_UNIT"mehrere "CaptureËingänge. Mit diesen werden die beiden Hallsensoren eines Motors verbunden. Eine positive Flanke erzeugt ein Interrupt, in dem die Zeitdifferenz zwischen zwei Hallsensorflanken berechnet wird. Basierend darauf kann dann die Motordrehzahl ermittelt werden. Die Klasse Motor übernimmt somit die Einrichtung der "MCPWM_UNIT"über die Hardwareabstraktionsschicht von Espressife und stellt die Interruptfunktion zur Verfügung. AuSSerdem befindet sich auch eine Methode zur Aktualisierung des PI-Reglers in der Motor Klasse. Diese Methode wird periodisch für beide Motoren von MotorController aufgerufen.

Im Folgenden wird auf die Methoden der Klasse Motor eingegangen:

- `void init_a()` und `void init_b()` erzeugt das Motor_a und Motor_b Objekt.
- `void setSetpointVelocity(float setpoint_velocity)` setzt die Solldrehzahl des Motors.
- `float getSetpointVelocity()` gibt die Solldrehzahl des Motors zurück.
- `float getActualVelocity()` gibt die aktuelle Motordrehzahl zurück. Diese wird aus der im Interrupt berechneten Zeitdifferenz ermittelt.
- `float updatePIControl(float actual_velocity)` aktualisiert den PI-Regler. Die Methode gibt die Pulsbreite des PWM Stellsignals zurück.
- `void setDuty(float duty_cycle)` setzt die Pulsbreite für den Motortreiber.
- `bool _encoder_callback(...)` wird von der "MCPWM_UNIT"bei einem Interrupt durch eine positive Flanke auf einem der Hallsensoreingängen ausgeführt. Es wird die Zeitdifferenz zum letzten Aufruf ermittelt und über die Art der letzten Flanke (positiv/negativ) des zweiten Hallsignals die Drehrichtung bestimmt.

5.7.2 MotorController

Die Klasse MotorController steuert den PI-Regler in Motor. Ein Timer Interrupt des ESP32 führt jede Millisekunde eine Interruptfunktion aus MotorController aus. Die `updatePIControl()` Methode aus Motor wird allerdings nicht aus dem Interrupt ausgeführt. Der Aufruf befindet sich in einem FreeRTOS Prozess `_motor_control_loop_task()`. Dieser blockiert solange bis er aus der Interruptfunktion aufgeweckt wird. Dann holt sich der Prozess die aktuelle Drehzahl der Motoren `getActualVelocity()`, aktualisiert die PI-Regler `updatePIControl()`, setzt die neue Pulsbreite `setDuty()` und wartet wieder darauf aus dem Interrupt aufgeweckt zu werden. Das Steuern eines Prozesses aus einem Interrupt wird "Deferred Interrupt Handling"genannt und hat den Vorteil, dass das Interrupt möglichst schnell durchläuft und niemals blockiert wird. AuSSerdem wird aus MotorController der Enable Pin für den Motortreiber aktiviert.

Im Folgenden wird auf die Methoden der Klasse MotorController eingegangen:

- `MotorController& init(Motor& motor_a, Motor& motor_b)` erstellt das MotorController Objekt.

- `void setVelocity(float setpoint_velocity_a, float setpoint_velocity_b)` setzt die Solldrehzahlen beider Motoren.
- `void enablePIcontrol()` aktiviert den PI-Regler
- `void disablePIcontrol()` deaktiviert den PI-Regler. Wenn der PI-Regler deaktiviert ist, werden die Solldrehzahlen zur Steuerung der Pulsbreiten des PWM-Signals genommen.
- `void _motor_control_loop_task(void* pvParameters)` ist der FreeRTOS Prozess zur Steuerung des PI-Regelkreises.
- `bool _motor_control_interrupt(void* args)` ist die Interruptfunktion, die den Prozess zyklisch aktiviert.

5.8 OutputVelocity

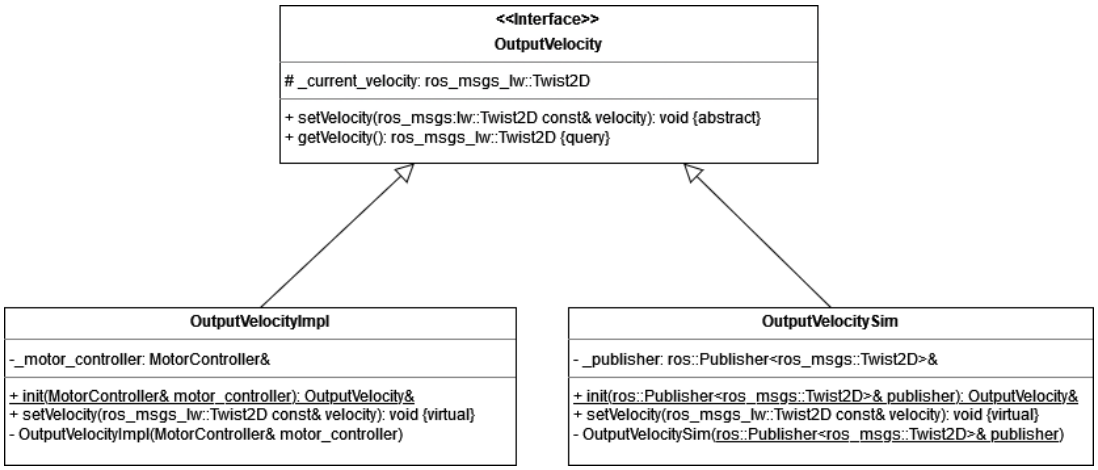

Abbildung 5.5: Klassendiagramm OutputVelocity

Die Komponente OutputVelocity besteht aus zwei Unterklassen die von einer Oberklasse OutputVelocity erben. Es kann nur ein Objekt von OutputVelocityImpl oder OutputVelocitySim existieren. Die Unterklasse OutputVelocityImpl ist für den echten Roboterbetrieb zuständig und OutputVelocitySim für den Simulationsbetrieb. Durch die abstrakte Oberklasse ist ein einfacher Wechsel zwischen Simulation und Echtbetrieb möglich. OuputVelocity ist nicht komplett abstrakt, da die Methode `ros_msgs_lw::Twist2D getVelocity()` implementiert wird. Über diese Methode kann der aktuelle Geschwindigkeitsvektor des Roboters abgefragt werden.

5.8.1 OuputVelocityImpl

Die Unterklasse OutputVelocityImpl transformiert den Sollgeschwindigkeitsvektor des Roboters in Drehzahlen (ω_L *und* ω_R) für die Motoren und gibt diese an den MotorController weiter. Der Geschwindigkeitsvektor eines Roboters mit Differentialantrieb besteht aus einer lineare Geschwindigkeit v und einer Rotationsgeschwindigkeit ω. Für die Transformation muss auSSerdem der Radabstand d und der Raddurchmesser r bekannt sein.

$$\begin{pmatrix} \omega_R \\ \omega_L \end{pmatrix} = \frac{1}{2\pi \cdot r} \begin{pmatrix} 1 & -0.5 \cdot d \\ -1 & -0.5 \cdot d \end{pmatrix} \cdot \begin{pmatrix} v \\ \omega \end{pmatrix}$$

Die maximale Drehzahl der im Projekt verwendeten Polulu 20D Motoren beträgt 220 Umdrehungen pro Minute. Ergibt die Transformation eine gröSSere Drehzahl werden beide Drehzahlen um den gleichen Faktor runterskaliert.

- `OutputVelocity& init(MotorController& motor_controller)` erzeugt das OuputVelocityImpl Objekt.
- `void setVelocity(ros_msgs_lw::Twist2D const& velocity)` setzt einen neuen Geschwindigkeitsvektor, transformiert diesen in Drehzahlen und gibt diese an MotorController weiter.

5.8.2 OutputVelocitySim

Die Unterklasse OutputVelocitySim ermöglicht den Simulationsbetrieb. Geschwindigkeitvektoren werden nicht in Drehzahlen umgewandelt, sondern über einen Publisher an ROS gesendet. In ROS können diese dann als Eingangssignal für z.B. den "TurtlesimSSimulator verwendet werden.

- `OutputVelocity& init(MotorController& motor_controller)` erzeugt das OuputVelocitySim Objekt.
- `void setVelocity(ros_msgs_lw::Twist2D const& velocity)` setzt einen neuen Geschwindigkeitsvektor und sendet diesen an ROS.

5.9 SensorPose

Die Komponente SensorPose dient zur Erfassung der Pose des Roboters. Durch die Schnittstelle SensorPose ist es möglich zwischen Unterschiedlichen SSensorenßu wechseln. Diese Sensoren sind als Unterklasse implementiert. Im Projekt gibt es die Möglichkeit zwischen Marvelmind, SensorPoseSim und KalmanFilter zu wechseln. Der Sensor Marvelmind liefert die Pose direkt aus den Messungen des Marvelmind Mini-RX Beacons. SensorPoseSim bekommt Positionsdaten aus ROS und ermöglicht dadurch die Simulation des Roboters in ROS. KalmanFilter besitzt eine Liste von sogenannten KalmanSensoren. Marvelmind und SensorPoseSim sind auch Unterklassen der Schnittstelle KalmanSensor. Die Positionsdaten der KalmanSensoren werden aufgwertet, so dass Positionsdaten mit einer höheren Aktualisierungsrate von $100Hz$ verfügbar sind. Ursprünglich war auch eine Sensorfusion mit z.B. Motordrehzahl und Messungen einer IMU geplant. Dies konnte allerdings nicht mehr aus Zeitgründen umgesetzt werden.

5.9.1 SensorPose

Die Klasse SensorPose definiert die gemeinsame Schnittstelle aller Sensoren und ermöglicht somit eine einfaches Wechseln zwischen unterschiedlichen Sensoren. SensorPose definiert zwei Möglichkeiten die Pose des Roboters zu lesen.

- `bool getPose(ros_msgs_lw::Pose2D& current_pose)` liest nur eine Position, falls seit dem letzten Lesen eine neue Messung verfügbar ist.
- `bool peekAtPose(ros_msgs_lw::Pose2D& current_pose)` liefert immer eine Position. Unabhängig von dem Alter der letzten Messung.

- Über `void reInit()` können interne Zustände des Sensors neu initialisiert werden.

5.9.2 KalmanFilter

Die Positionsmessungen der Klassen Marvelmind und SensorPoseSim haben eine relativ geringe Aktualisierungsrate. Marvelmind gibt z.B. nur $7Hz - 15Hz$ an. Da für die Positionsregelung allerdings eine Aktualisierungsrate von $100Hz$ vorgesehen ist, müssen über einen Kalman Filter öfter Positionsdaten bereitgestellt werden. Der Kalman Filter schätzt über die Systemgleichung des Roboters zyklisch eine neue Position. Wenn eine Messung verfügbar wird, wird die Plausibilität der Positionsmessung und der geschätzten Position anhand der jeweiligen Kovarianzmatrix bewertet. Basierend darauf werden beide Position dann fusioniert. Der Zustandsvektor z_t des Roboters ist:

$$z_t = \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} \tag{5.1}$$

Der Geschwindigkeitsvektor u_t dient als Steuersignal des Roboters:

$$u_t = \begin{pmatrix} v \\ \omega \end{pmatrix} \tag{5.2}$$

Der Differentialantrieb des Roboters wird mit dem Unicycle Modell HEILMANN UND KNOBLACH [2021] angenähert. Die Zustandsgleichung beträgt somit:

$$z_t = f(z_{t-1}, u_t) = \begin{pmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{pmatrix} + \Delta t \begin{pmatrix} \cos\theta_{t-1} & 0 \\ \sin\theta_{t-1} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v \\ \omega \end{pmatrix} \tag{5.3}$$

Die Messungen der Sensoren y_t werden über die folgende Beobachtungsgleichung in den Zustandsraum des Roboters z_t transformiert:

$$y_t = H \cdot z_t = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} \tag{5.4}$$

Der Kalman Filter geht von einem linearen Zustandsraum aus. Da die Zustandsgleichung des Roboters nicht linear ist, muss diese mithilfe der Jacobi-Matrix linearisiert werden. Im Allgemeinen gilt dies auch für die Beobachtungsgleichung, wenn diese nicht linear ist.

$$F_t = \left.\frac{\partial f}{\partial z}\right|_{z_{t-1}, u_t} = \begin{pmatrix} 1 & 0 & -v \cdot \sin(\omega_t) \\ 0 & 1 & v \cdot \cos(\omega_t) \\ 0 & 0 & 1 \end{pmatrix} \tag{5.5}$$

Bei dieser Erweiterung spricht man vom Extended Kalman Filter. Die Funktionsweise lässt sich in zwei Schritte unterteilen. Im ersten Vorhersage Schritt wird ein neuer Zustand $\hat{z}_t^-$ mithilfe der Zustandsgleichung 5.3 geschätzt. AuSSerdem wird die Unsicherheit P_t^- dieser Schätzung mithilfe des Prozessrauschens Q und mit der vorherigen Kovarianz P_{t-1}^+ berechnet.

$$\begin{aligned} \hat{z}_t^- &= f(\hat{z}_{t-1}^+, u_t) \\ P_t^- &= F_t P_{t-1}^+ F_t^T + Q \end{aligned} \tag{5.6}$$

Im zweiten Update Schritt wird die Schätzung mithilfe der Positionsmessung $\tilde{y}_t$ von z.B. dem Marvelmindsensor korrigiert. Da auch jede Messung etwas Messrauschen R besitzt, muss die Plausibilität der Messung und der Schätzung bestimmt werden. Hierfür wird der Kalman Gain K_t aus P_t^- und R berechnet. Der Kalman Gain bestimmt die Gewichtung der Messung in der abschlieSSenden Positionsschätzung $\hat{z}_t^+$. AuSSerdem wird noch die Kovarianz P_t^+ dieser Schätzung für den nächsten Durchlauf berechnet.

$$\begin{aligned} K_t &= P_t^- H^T (H P_t^- H^T + R)^{-1} \\ \hat{z}_t^+ &= \hat{z}_t^- + K_t(\tilde{y}_t - H\hat{z}_t^-)) \\ P_t^+ &= (I - K_t H) P_t^- \end{aligned} \tag{5.7}$$

Falls neue Messungen noch nicht verfügbar sind, kann der zweite Schritt auch übersprungen werden und für den nächsten Zyklus $\hat{z}_t^+ = \hat{z}_t^-$ und $P_t^+ = P_t^-$ gesetzt werden. Des Weiteren ist es durch Wiederholung des zweiten Schrittes für unterschiedliche Sensoren möglich, mehrere Messungen zu Fusionieren. Dies wird im Projekt über eine Liste von KalmanSensoren umgesetzt. Diese wird in jedem Zyklus durchlaufen und die Zustandsvariablen aktualisiert, falls neue Messungen verfügbar sind. Die Berechnung geschieht in einem FreeRTOS Prozess `_kalman_filter_loop_task()`, der über einen Softwaretimer auf $100Hz$ getaktet wird.

- `SensorPose& init(std::initializer_list<KalmanSensor const*> const& sensor_list, OutputVelocity c` erzeugt ein KalmanFilter Objekt. Dem KalmanFilter muss eine Liste von KalmanSensoren übergeben werden sowie das OuputVelocity Objekt, aus dem der Sollgeschwindigkeitsvektor bezogen wird.
- Über die Methode `void reInit()` wird der interne Zustandsvektor im KalmanFilter neu initialisiert. Hierfür stellt der erste Sensor aus der internen Liste Positionsdaten bereit. Dieser Mechanismus ist notwendig, wenn der Roboter z.B. händisch versetzt wird.
- Der Prozess `void _kalman_filter_loop_task(void* pvParameters)` berechnet beide KalmanFilter Schritte. Nach einem Zyklus blockiert der Prozess bis er durch den Software Timer entblockt wurde.
- `void _kalman_filter_loop_timer(TimerHandle_t timer)` ist die Callbackfunktion des Softwaretimers und entblockt den `_kalman_filter_loop_task()` Prozess.

5.9.3 KalmanSensor

Die Klasse KalmanSensor definiert eine Schnittstelle für alle Sensoren, die von KalmanFilter fusioniert werden sollen. Ein KalmanSensor muss die folgenden Methoden implementieren:

- `bool calculateKalman(ros_msgs_lw::Pose2D const& a_priori_estimate, dspm::Mat const& a_priori_cov` berechnet den Update Schritt für den jeweiligen KalmanSensor.
- `bool getAbsolutePose(ros_msgs_lw::Pose2D& initial_pose)` gibt die aktuelle Pose des Roboters zurück. Diese Methode muss nur für Kalman Sensoren, die die absolute Position erfassen implementiert werden.

- `void getMeasurementNoiseCov(dspm::Mat& measurement_cov)` gibt die Kovarianzmatrix des Messrauschens des jeweiligen Sensors zurück.
- `void calculateMeasurementNoiseCov()` ermittelt das Messrauschen des jeweiligen Sensors.

5.9.4 Marvelmind

Die Klasse Marvelmind kann als KalmanSensor und auch als alleinstehender Sensor über SensorPose genutzt werden. Marvelmind setzt die Hardwarekonfiguration für die UART Schnittstelle des ESP32 um und kommuniziert über UART mit dem Marvelmind Mini RX Beacon. In dem Marvelmind Dashboard lassen sich unterschiedliche Pakete aktivieren, die über die UART Schnittstelle versendet werden. Diese sind in dem Dokument "Hardware interfaces and protocols of data exchange with Marvelminddevices"Mar definiert. Abbildung 5.7 zeigt das in diesem Projekt genutzte Paket. Dieses übermittelt Position in mm und Orientierung in Dezigrad. Die Klasse Marvelmind liest die seriellen Pakete über einen FreeRTOS Prozess `_uart_read_data_task()`, deserialisiert die Daten und speichert sie in einer FreeRTOS "Queue". Die Pose kann dann über die Methoden der SensorsPose Schnittstelle ausgelesen werden oder als KalmanSensor von der Klasse KalmanFilter vewendet werden. Marvelmind definiert auch einige KConfig Menüeinstellungen für die UART Pins am ESP32, Geschwindigkeit der UART Kommunikation und die GröSSe eines Empfangspuffers.

5.9.5 SensorPoseSim

SensorPoseSim ist auch eine Unterklasse von SensorPose und KalmanSensor. Die Klasse dient zur Simulation des Roboters in ROS über z.B. den Turtlesim. Die Roboterpose wird auf der Topic /robotername/poseëmpfangen. Diese wird im Konstruktor des SensorPoseSim Objektes abonniert. Die Methode `void _setPose()` dient dabei als Callbackfunktion für den Subscriber.

5.10 PositionController

Die Komponente PositionController setzt den äuSSeren Regelkreise des Roboters um. Da der Roboter mit unterschiedlichen Regelaufgaben umgehen muss, ist kein Regler fest implementiert. Es wird dem Programmmierer erlaubt unterschiedliche Regler zu implementieren, die über eine gemeinsame Schnittstelle angesteuert werden. Im Projekt wurde der p2pController, approxLinController, statInOutLinController und dynInOutLinController Regler implementiert. Mit dem ersten können Punkte angefahren werden. Die letzen drei dienen zum Folgen von Trajektorien. Da Funktionsweise und mathematische Herleitung dieser Regler in der vorhergehenden Projektarbeit "Regelung und Simulation von Schwarmrobotern"Heilmann und Knoblach [2021] behandelt wurde, wird in diesem Kapitel nur auf Besonderheiten der Implementierung eingegangen.

5.10.1 ControllerMaster

Die Klasse ControllerMaster schlieSSt den Regelkreis der Positionsregelung. Es wird zunächst die aktuelle Position aus dem SensorPose Objekt bezogen, anschlieSSend wird ein neuer Geschwindigkeitsvektor als StellgröSSe über das PositionController Objekt berechnet und dieser Vektor an das OutputVelocity Objekt weitergereicht. Zuletzt überprüft der ControllerMaster, ob das Ziel der Regelung erreicht wurde und beendet den Regelungszyklus. Durch eine Callbackfunktion werden andere Komponenten über das Ende

der Regelung informiert. Der FreeRTOS Prozess `_control_loop_task()` beinhaltet den Regelkreislauf und ein Softwaretimer taktet diesen auf $100Hz$.

- `ControllerMaster& init(OutputVelocity& output_velocity, SensorPose& sensor_pose)` erzeugt das ControllerMaster Objekt. ControllerMaster ist eine Singleton Klasse. Es kann somit nur ein Objekt existieren.
- `void start_controller(PositionController* pos_controller, std::function<void()>destination_reach` startet die Regelung. Falls eine Regelung bereits im Gange ist wird diese beendet und mit der neuen ersetzt. Der Methode wird neben dem PositionController Objekt auch die Callbackfunktion übergeben.
- Mit `void stop_controller()` kann die Regelung von auSSerhalb der PositionController Komponente beendet werden.
- `void _control_loop_task(void* pvParameters)` ist ein FreeRTOS Prozess und tätigt die Berechnung des Regelzyklus.
- `void _control_loop_timer(TimerHandle_t timer)` ist die Callbackfunktion des Softwaretimers über den der Prozess getaktet wird.

5.10.2 PositionController

Die Klasse PositionController definiert die gemeinsame Schnittstelle über die die Regler vom ControllerMaster gesteuert werden. Die Regelung ist komplett innerhalb der Unterklassen dieser Schnittstelle gekapselt. ControllerMaster besitzt somit keine Information über die Art der Regelung.

- Mit `ros_msgs_lw::Twist2D update(ros_msgs_lw::Pose2D const& actual_pose)` wird für eine neue Istposition ein neuer Geschwindigkeitsvektor als StellgröSSe berechnet. Der ControllerMaster muss diese Methode alle $10ms$ aufrufen.
- Mit `bool destination_reached()` kann der ControllerMaster überpüfen, ob das Ziel der Regelung (Zielpunkt oder Trajektorienende) erreicht wurde.

5.10.3 p2pController

Über den Punkt zu Punkt Regler können einzelne Punkte angefahren werden. Dieser Zielpunkt wird dem Konstruktor des Reglers übergeben.

5.10.4 approxLin-, statInOutLin-, dynInOutLinController

Mit den Reglern approximierte Linearisierung, statische Ein- und Ausgangslinearisierung und dynamische Ein- und Ausgangslinearisierung können Trajektorien abgefahren werden. Eine Trajektorie besteht aus einem Array von Trajektorienpunkten. Jeder Trajektorienpunkte beinhaltet Positions-, Geschwindigkeits, Beschleunigungsvektoren und einen Zeitstempel. Über den Zeitstempel wird bei jedem Aufruf der `update()` Funktion der aktuellste Trajektorienpunkt als Sollwert für die Trajektorienregelung genommen. Da die Trajektorien als Nachrichten aus dem ROS-System entspringen, stimmt der Wert des Zeitstempels nicht mit der ESP32 internen Zeit überein. Diese Zeitdifferenz wird zum Beginn der Regelung berechnet und folgende Punkte korrigiert.

5.11 StateMachine

Die Komponente StateMachine steuert das Verhalten des Roboters auf der höchsten Ebene. Eine Zustandsmaschine kontrolliert das Verhalten bei bestimmten Aktionen. Die Zustandsmaschine ist nach dem SState PatternËntwurfsmuster implementiert. Für jede Methode des Objektes StateMachine definiert der Zustand eine Methode mit dem selben Namen. Ein Aufruf von StateMachine wird an die Methode des derzeitigen Zustandes weitergereicht. Somit ändert sich das Verhalten des StateMachine Objektes basierend auf dem internen Zustand. Abbildung 5.9 zeigt das Klassendiagramm dieses Entwurfmusters. Die Übergänge der Zustandsmaschine sind von der Implementierung der Unterklassen von State abhängig. Abbildung 5.10 zeigt die in diesem Projekt umgesetzte Zustandsmaschine. Die Methoden aus StateMachine werden als Callbackfunktion des RosBridgeClient Subscribers verwendet. Somit würde z.B. eine empfangene Geschwindigkeitsvektor Nachricht einen Zustandsübergang von Idle nach DriveWithVelocity veranlassen.

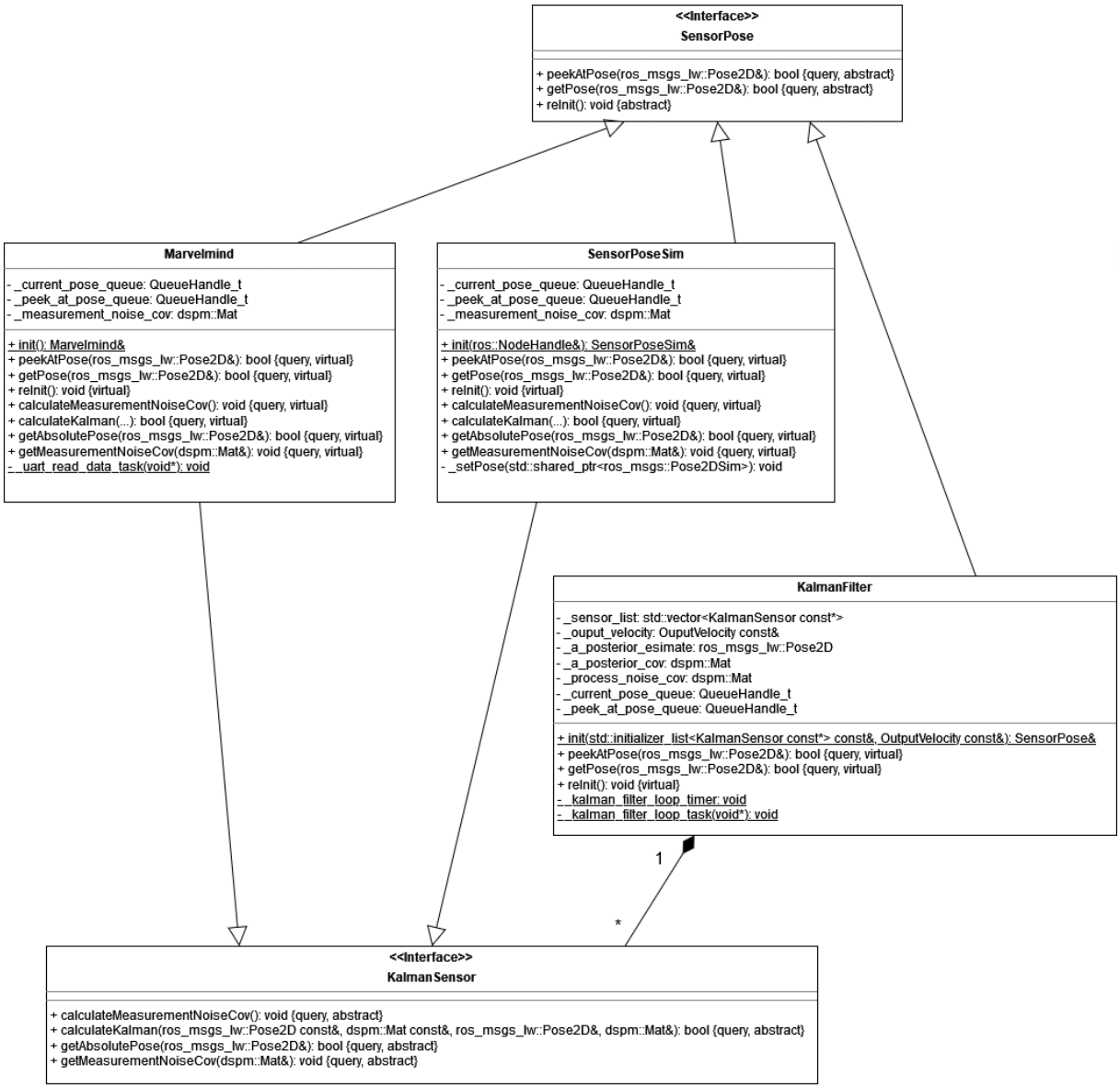

Abbildung 5.6: Klassendiagramm SensorPose

Offset	Size (bytes)	Type	Description	Value
0	1	uint8_t	Destination address	0xff
1	1	uint8_t	Type of packet	0x47
2	2	uint16_t	Code of data in packet	0x0011
4	1	uint8_t	Number of bytes of data transmitting	0x16
5	4	uint32_t	Timestamp – internal time of beacon ultrasound emission, in milliseconds from the moment of the latest wakeup event. See note.	
9	4	int32_t	Coordinate X of beacon, mm	
13	4	int32_t	Coordinate Y of beacon, mm	
17	4	int32_t	Coordinate Z, height of beacon, mm	
21	1	uint8_t	Byte of flags: Bit 0: 1 - coordinates unavailable. Data from fields X,Y,Z should not be used. Bit 1: timestamp units indicator (see note) Bit 2: 1 - user button is pushed (V5.23+) Bit 3: 1 - data are available for uploading to user device, see section 2 (V5.34+) Bit 4: 1 - want to download data from user device, see section 2 (V5.34+) Bit 5: 1 – second user button is pushed (V5.74+) Bit 6: 1 – data for another hedgehog (not same one that sending this packet) Bit 7: – 1 – out of geofencing zone	
22	1	uint8_t	Address of hedgehog	
23	2	uint16_t	Bit 0...11: orientation of hedgehogs pair in XY plane, decidegrees (0...3600) Bit 12: 1 – coordinates are given for center of beacons pair; 0 – coordinates for specified hedgehog Bit 13: 1 - orientation is not applicable Bit 14...15: reserved (0)	
25	2	uint16_t	Time passed from ultrasound emission to current time, milliseconds (V5.88+)	
27	2	uint16_t	CRC-16 (see appendix 1)	

Abbildung 5.7: Marvelmind Positions Paket

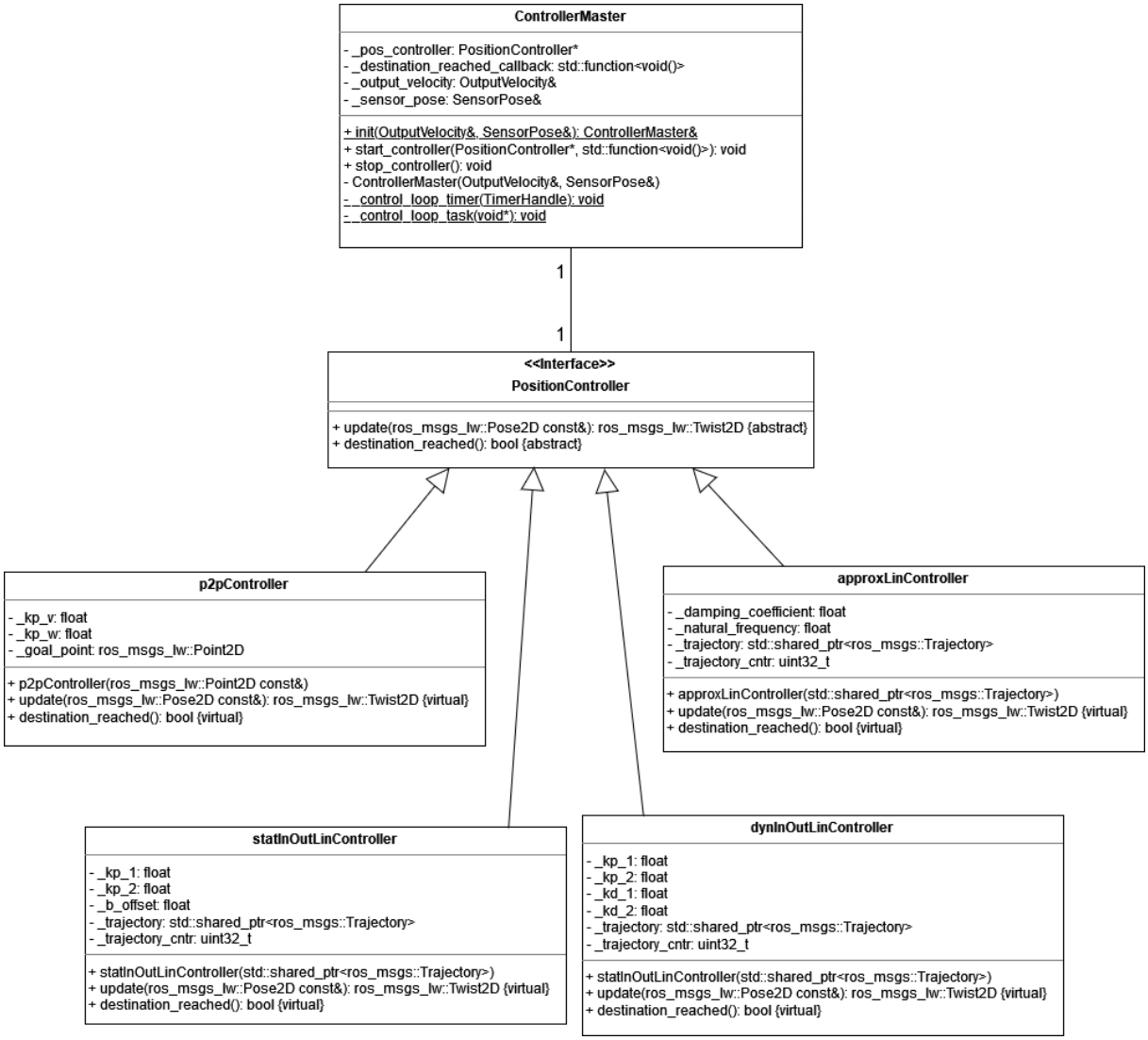

Abbildung 5.8: PositionController Klassendiagramm

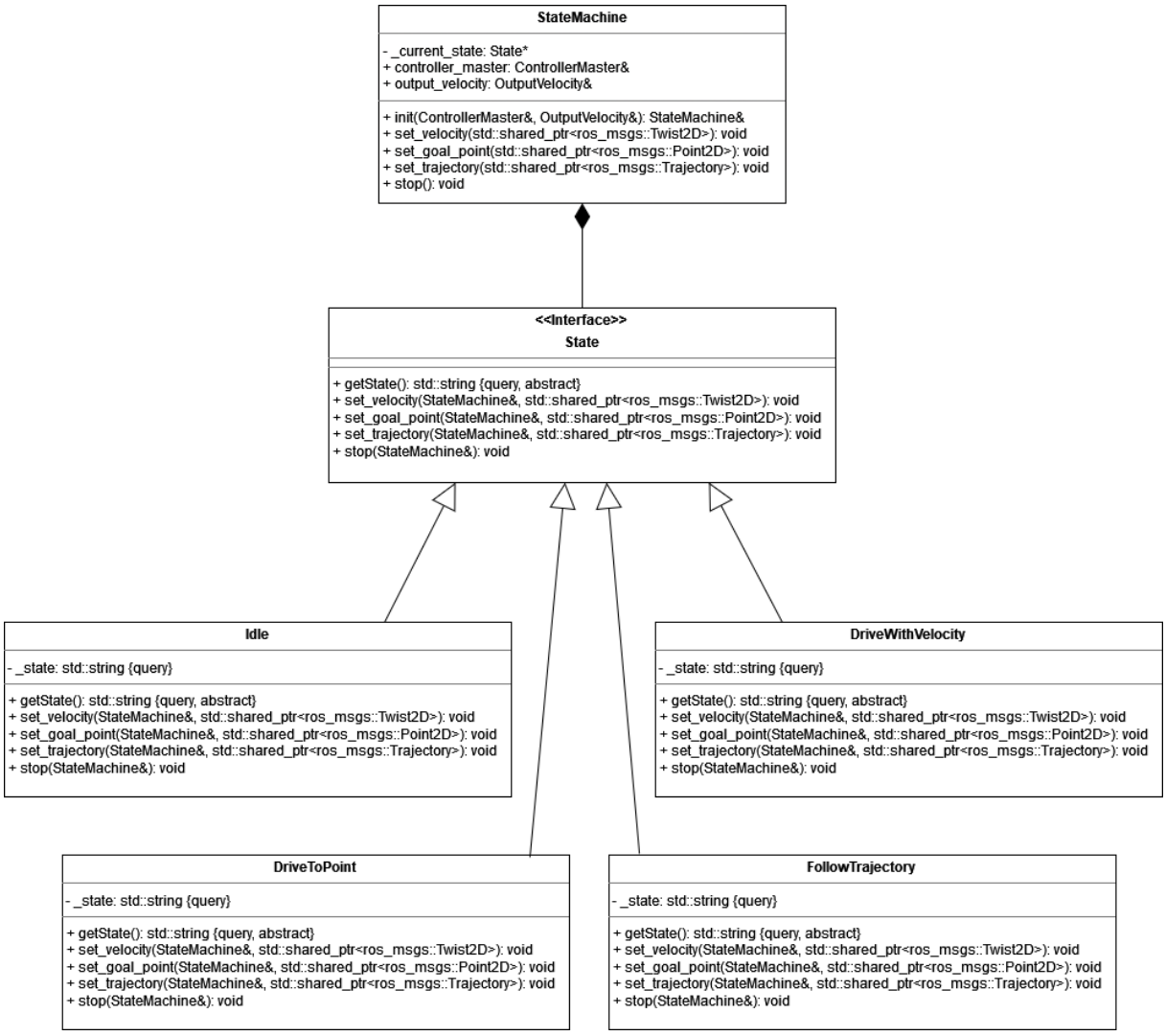

Abbildung 5.9: StateMachine Klassendiagramm

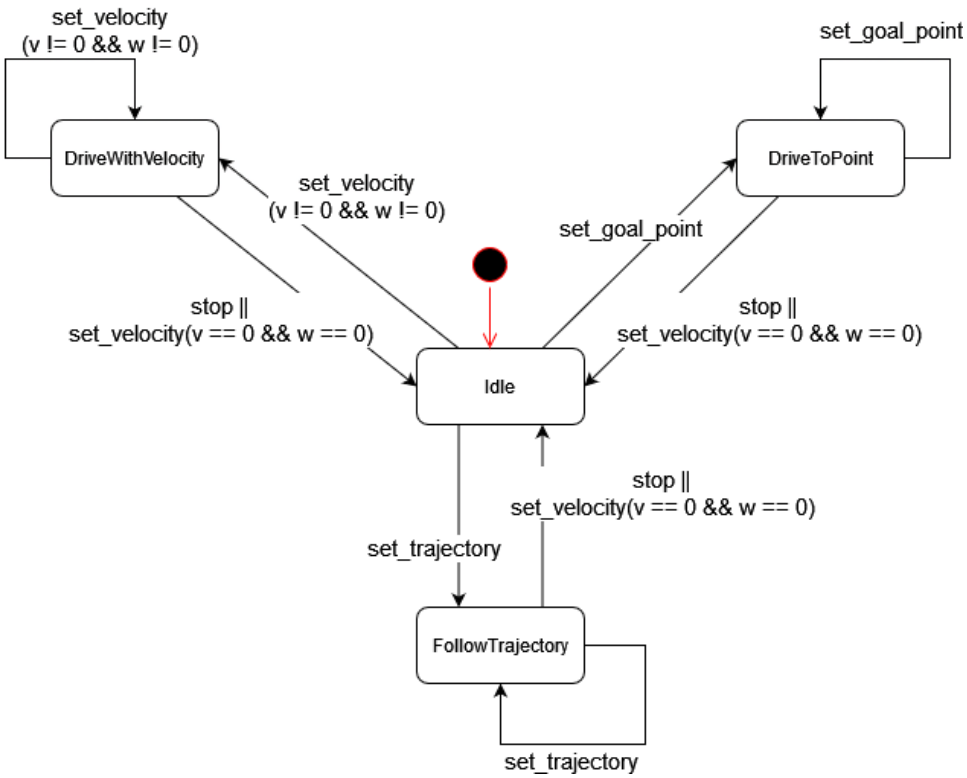

Abbildung 5.10: Zustandsdiamm Roboter

Kapitel 6

ROS

6.1 Softwarearchitektur in ROS

6.2 rosbridge-server

6.3 Trajektorienplanung

Kapitel 7

ROS-Trajektorie

7.1 Trajektorienplanung

Kapitel 8

Marvelmind

8.1 Marvelmind-Aufbau

8.2 Probleme in der Nutzung von Marvelmind

Kapitel 9

Docker

Zur Vereinfachung des Arbeitsflusses und Ordnung einzelner Softwarekomponenten wurde `Docker` und `Docker Compose` verwendet.
Im folgenden Kapitel wird nun die Funktion und Vorgehensweise während der Nutzung von `Docker` erklärt.

9.1 Docker

9.1.1 Einführung und Funktion von Docker

Docker ist eine offene Plattform für die Entwicklung, Bereitstellung und Ausführung von Anwendungen. Dabei verpackt Docker die Software in standardisierte Einheiten, sogenannte Container. Diese enthalten alles, was zum Ausführen der Software erforderlich ist, wie Bibliotheken, Systemtools, Code und Laufzeit.[1]
Docker führt also Container aus. Dabei fungiert ein Container als eine Art von Virtueller Maschine. Allerdings ist hier der Aufbau relativ leicht, da dieser nicht virtualisiert ist und auf dem selben Kernel läuft. Gleichzeitig ist der Container trotzdem abgeschottet, was dem Prinzip des Sandboxings [2] ähnelt.

Dabei erstellt man Container von einem Abbild. Diese kann man selbst erstellen und nennt man Container-Image. Der groSSe Vorteil ist, dass man Container letztlich wie Rezepte zum Beschreiben verwendet, die ein leichtes Installieren der Software garantieren.
Eine Erweiterung stellt Docker Compose dar. Dies erlaubt uns einen kompletten Stack zu beschreiben, also ganze Ansammlungen von Containern (und nicht nur einzelne Container) zusammen zu starten oder mit denen zu kommunizieren.

Bei einem Start des Projekts muss nur das repository gecloned werden und ein Befehl gestartet werden. Diese Art zu Arbeiten ermöglicht einen schnellen Workflow.

9.2 Installation von Docker-Desktop

1. Stellen Sie zunächst sicher, dass Windows auf dem neuesten Stand ist
   - Geben Sie in der Windows-Suche "Windows Updateëin und wählen Sie `Windows Update-Einstellung`.
   - Sie sollten ein grünes Häkchen sehen und SSie sind auf dem neuesten Stand". Falls nicht, klicken Sie auf "Nach Updates suchen". Sie müssen diesen Vorgang so lange wiederholen, bis Sie keine Updates mehr zu installieren haben.

2. Installation von WSL2
   - Geben Sie in der Windows-Suche "powershellëin, klicken Sie mit der rechten Maustaste auf `Windows PowerShell` und dann auf `Als Administrator ausführen`.
   - Klicken Sie auf "Ja", um PowerShell zu erlauben, Änderungen an Ihrem Gerät vorzunehmen.
   - Führen Sie im Windows PowerShell-Fenster den Befehl "wsl –install -d Ubuntuäus.

[1] weitere Infos unter `https://docs.docker.com/get-started/overview/`
[2] Sandboxing ist eine Softwareverwaltungsstrategie, die Anwendungen von wichtigen Systemressourcen und anderen Programmen isoliert Sie bietet eine zusätzliche Sicherheitsebene, die verhindert, dass sich Malware oder schädliche Anwendungen negativ auf Ihr System auswirken.
Nähere Infos unter: `https://techterms.com/definition/sandboxing`

- Aktivieren Sie anschlieSSend die Plattform für virtuelle Maschinen. Im Windows PowerShell ausführen. (kopieren und einfügen) "dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart".
- Laden Sie das WSL2 Linux-Kernel-Update-Paket für x64-Maschinen herunter und installieren Sie es.
- Windows neu starten.
- Geben Sie nochmal in der Windows-Suche "powershellëin, klicken Sie mit der rechten Maustaste auf `Windows PowerShell` und dann auf `Als Administrator ausführen`.
- Führen Sie im Windows PowerShell-Fenster den Befehl "wsl –set-default-version 2äus.
- Als nächstes installieren Sie eine Linux-Distribution aus dem Microsoft Store. Ich empfehle Ubuntu 20.04.4 LTS .(Das Herunterladen und Installieren wird einige Minuten dauern)
- Sie werden aufgefordert, einen Linux-Benutzer einzurichten. Am besten Verwenden Sie dafür denselben Benutzernamen, den Sie für Windows verwenden.
- Sie können nun Linux-Befehle im Ubuntu-Terminalfenster ausführen. Ich empfehle, das Ubuntu-Symbol an die Taskleiste zu heften.

3. Jetzt können Sie Docker Desktop für Windows installieren

- Führen Sie das Installationsprogramm aus und starten Sie anschlieSSend Windows neu.
- Melden Sie sich bei Windows an und starten Sie Docker-Desktop.Lassen sie Docker die Einrichtung abschlieSSen, dies kann je nach Rechner einige Minuten dauern.

9.3 Unsere Verwendung von Docker

9.3.1 Docker-Compose

Sämtliche Software, die in unserem Projekt ausgeführt wird, läuft in Docker-Containern. Im folgenden werden die `Docker Compose`-Befehle, welche wir in unserem *dokcer-compose.yml* Datei verwenden auführlich erklärt.

```
version: '3.9' #Compose file format wird definiert, hierfuer wird Docker Engine 19.03.0 und hoeher verwendet
networks: #Mit diesen Befehlen erzeugen wir ein Netzwerk names rosnet
  rosnet:

services: #Unter services werden die einzelnen Containern definiert

  master: #Hier wird ein Container mit dem Namen "master" angelegt
    image: ros:noetic-robot # Als image wird hier "ros:noetic-robot" aus Docker-Hub verwendet
    command:
      - roscore #Mit command wird der Befehl "roscore" in der Kommandozeile ausgefuehrt
    networks:
      - rosnet #networks definiert das Netzwerk des masters

#Da, die Einstellungen und Befehle von den letzten zwei Container sich aehneln, wird es nur fuer ein
     Container erklaert.
  rosbridge: #Hier wird ein Container mit dem Namen "rosbridge" angelegt
    build:
      context: ./rosbridge #Als image, bzw build wird hier zu einem Dockerfile navigiert, welche unter dem
          Verzeichnis "./rosbridge" befindet
    environment:
      - "ROS_HOSTNAME=rosbridge"
      - "ROS_MASTER_URI=http://master:11311" #Unter environment werden Environmentvariablen festgelegt, in
          dem Fall wie der HOSTNAME des Containers lautet und wo sich der master im Netzwerk befindet
    networks:
      - rosnet
    depends_on:
      - master #Der Container faehrt erst hoch, wenn der master bereits laeuft und schaltet sich vorm
          master aus
    ports:
      - 9090:9090 #Hier werden Ports in diesem Form (HOST-PORT:CONTAINER-PORT) nach aussen freigegeben

  rosrobotbridge:
    build:
      context: ./rosrobotbridge
    environment:
      - "ROS_HOSTNAME=rosrobotbridge"
      - "ROS_MASTER_URI=http://master:11311"
    networks:
      - rosnet
    depends_on:
      - master
    ports:
      - 2888:2888
```

Listing 9.1: docker-compose.yml

9.3.2 Dockerfile

Im Unterkapitel Docker-Compose werden in einem *dokcer-compose.yml* Datei mehrere build-Befehle aufgerufen, welche zu dem eigentlichen *Dockerfile* nagivieren und diesen als Hauptimage bauen. Im folgenden werden die Befehle aus dem *Dockerfile* und deren Funktionen abhand eines Beispiels aus unserem Projekt erklärt und geschildert.

```
FROM ros:noetic-robot #Hier wird ein bereits vorhandenes Image aus dem Dockerhub aufgerufen und als Basis
     verwendet
```

```
2 RUN apt-get -y update #Mit dem RUN-Befehl werden Befehle in der Kommandozeile ausgefuehrt
3 RUN apt-get -y upgrade # -y sorgt dafuer, dass wenn es nach dem ausfuehren von dem Befehl eine JA-NEIN-
      Frage erscheinen soll, diese automatisch mit JA bzw YES beantwortet wird
4 RUN apt-get -y install ros-noetic-rosbridge-server
5 COPY ./start.sh start.sh #COPY-Befehl kopiert die Datei start.h auf unserem Image. COPY SOURCE-Verzeichnis
      DESTINATTION-Verzeichnis
6 RUN chmod +x start.sh
7 CMD ["./start.sh"] #Die Datei start.h wird ausgefuehrt
```

Listing 9.2: Dockerfile (rosbridge)

Kapitel 10

Web-App

Die Web-App ist eine Browserapplikation, die manuelle Echtzeit Steuerung der Roboter und deren LED-Farben, unabhängig von deren Anzahl ermöglicht. Die Steuerung erfolgt über eine ROS-Schnittstelle "rosbridge"mit der sich die Web-App, über ein Javabibliothek "roslibjs", mit dem Rosserver verbinden kann, um Daten einzulesen oder zusenden.

10.1 React

React ist ein Javascript Bibliothek zum Entwickeln und Erstellen von Benutzeroberflächen. React ist ein Opensource Projekt, welche damals von Facebook jetzt Meta entwickelt wurde. Um in React Web-Applikationen zuerstellen, braucht man Grundkenntnisse in CSS, HTML und Javascript.
Die wichtigste Eigenschaft von React ist, dass die Zustände der Applikation und der Benutzeroberfläche synchronisiert agieren, das heiSSt, wenn Änderungen am Sourcecode vorgenommen werden , verändert sich auch die Benutzeroberfläche. React ist Komponenten basiert, ein Reactapp besteht daher aus vielen kleinen React-Komponenten, welche das Programmieren und die Wiederverwendbarkeit von Objekten erleichtern.

Kapitel 11

Fazit & Ausblick

11.1 Fazit

11.2 Ausblick

Literaturverzeichnis

ESP

ESP-IDF Programming Guide. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html, . – 18-04-2022

Mar

Hardware interfaces and protocols of data exchange with Marvelminddevices. https://marvelmind.com/pics/marvelmind_interfaces.pdf, . – 03-06-2022

Heilmann und Knoblach 2021

Heilmann, Tim ; Knoblach, Josef: *Regelung und Simulation von Schwarmrobotern*. 2021

Anhang

11.3 Eidesstattliche Erklärung

Wir, Cara Bettendorf, Maximilian Dösch, Kevin Heise und Moin Sammari, versichern hiermit, dass wir die **Projektarbeit** mit dem Thema

Roboterformation – Fortführung

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben, wobei wir alle wörtlichen und sinngemäSSen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Nürnberg, den 23.06.2022

Cara Bettendorf, Maximilian Dösch, Kevin Heise und Moin Sammari