

# **Belegarbeit**

Zum Modul Echtzeitsysteme und mobile Robotik

Sommersemester 2025

Hochschulbetreuer: M. Sc. Moritz Thümmler

## **Steuerung mobiler Roboter**

### **praktische Anwendungen mit ROS2**



Vorgelegt von:

Eric Lausch, Nils Wiora

Studiengang:

Elektro- und Informationstechnik

Seminargruppe:

24EIM-AT

Bearbeitungszeitraum:

11. November 2025

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>IV</b>
<b>Abbildungsverzeichnis</b>	<b>V</b>
<b>1 Einleitung</b>	<b>5</b>
<b>2 Programmieraufgaben</b>	<b>6</b>
2.1 Sockets . . . . .	6
2.1.1 Teilaufgabe A . . . . .	6
2.1.2 Teilaufgabe B . . . . .	6
2.1.3 Teilaufgabe C . . . . .	7
2.1.4 Teilaufgabe D . . . . .	7
2.2 Interprozesskommunikation . . . . .	9
2.2.1 Teilaufgabe A und B . . . . .	9
2.2.2 Teilaufgabe C . . . . .	9
2.2.3 Teilaufgabe D . . . . .	9
<b>3 Grundkonzepte von ROS2 mit einem Versuchsroboter</b>	<b>10</b>
3.1 Einrichtung der Arbeitsumgebung und ROS 2-Paket . . . . .	10
3.1.1 Installation von ROS 2 Humble . . . . .	10
3.1.2 Erstellung eines ROS 2-Workspaces . . . . .	11
3.1.3 Ergebnisse . . . . .	11
3.2 Erstinbetriebnahme . . . . .	12
3.2.1 Grundprinzip von Nodes und Topics . . . . .	12
3.2.2 Start der Turtlesim-Umgebung . . . . .	12
3.2.3 Untersuchung der ROS-Kommunikation . . . . .	13
<b>4 TurtleBot mit ESP32</b>	<b>15</b>
4.1 Arbeitsumgebung . . . . .	15
4.1.1 Betriebssystem und Systemkonfiguration . . . . .	15
4.1.2 Entwicklungsumgebung PlatformIO . . . . .	16
4.1.3 Netzwerk- und Agentumgebung . . . . .	16
4.1.4 Erkenntnisse . . . . .	17
4.2 Turtlesim Teleop . . . . .	17
4.2.1 Micro-ROS-Struktur und Node-Konfiguration . . . . .	17
4.2.2 Kommunikation Motorcontroller . . . . .	18
4.2.3 Implementierungsdetails . . . . .	18

4.2.4	Fehlerquellen und Besonderheiten . . . . .	19
4.3	Abfahren von Polygonen . . . . .	20
4.3.1	Ziel und Aufbau . . . . .	21
4.3.2	Steuerlogik . . . . .	21
4.3.3	Implementierungsdetails . . . . .	21
4.3.4	Aufruf und Parametrisierung . . . . .	22
4.3.5	Beobachtungen . . . . .	23
4.4	Sensorintegration: Bumper mit Host-seitiger Sicherheitslogik . . . . .	24
4.4.1	Zielsetzung . . . . .	24
4.4.2	Systemübersicht und Topics . . . . .	24
4.4.3	Implementierung . . . . .	25
4.4.4	Parameter der Safety-Node . . . . .	26
4.4.5	Ergebnis . . . . .	26
4.4.6	Probleme . . . . .	26
<b>5</b>	<b>Steuerung eines professionellen Roboters mit ROS2</b>	<b>28</b>
<b>Literatur</b>		<b>29</b>
<b>Anhang</b>		<b>30</b>

## **Abkürzungsverzeichnis**

A a

## **Abbildungsverzeichnis**

1	Funktionsprinzip der zeitbasierten Steuerung des <i>ShapeDriver</i> -Knotens. . . . .	22
2	Fahrspur der Turtlesim-Node . . . . .	23
3	Blockdiagramm der Systemarchitektur . . . . .	25

# 1 Einleitung

Mobile Roboter sind den meisten Menschen wohl in Form von Putzrobotern geläufig. In der Industrie werden sie ebenfalls weit verbreitet eingesetzt. Eine häufige Anwendung für mobile Roboter ist der Warentransport innerhalb der Produktion. Es können beispielsweise Waren aus Lagern geholt, in der Fertigung Teile an weiterverarbeitende Stationen geliefert oder auch wie im Heimbereich Reinigungsarbeiten durchgeführt werden. Der Unterschied zu einem Fahrerlosen Transportsystem ist die flexible Wegfindung der autonomen mobilen Roboter. Diese bewegen sich nicht auf festen vorgegebenen Pfaden sondern finden innerhalb ihrer Bewegungszonen den optimalen Weg zu ihrem Ziel. Mit Hilfe von Sensoren und Technologien wie Radar, erfolgt die Orientierung im Raum aber auch die Hinderniserkennung. Die mobilen Roboter umfahren im Weg befindliche Hindernisse selbstständig. Die Verwaltung der gesamten in einem Betrieb vorhandenen Roboter kann mit Software erfolgen. Dort können beispielsweise bestimmte Aufgaben zugewiesen und priorisiert werden [1].

In diesem Beleg sollen zunächst einige Grundkonzepte, die bei Echtzeitsystemen Anwendung finden anhand von Beispielen umgesetzt werden. Dazu gehören Sockets und die Interprozesskommunikation. Anschließend werden die Grundlagen des Roboter Betriebssystems ROS mit Hilfe eines Versuchsroboters angewendet. Im Rahmen einer kleinen Beispielanwendung soll das Verständnis der Funktionsweise von ROS belegt werden. Abschließend erfolgt die Erprobung der erlangten Erkenntnisse mit einem professionellen mobilen Roboter.

## 2 Programmieraufgaben

### 2.1 Sockets

Ein Socket ist eine Softwarestruktur, die zur Netzwerkkommunikation verwendet wird. Darüber hinaus werden Sockets auch zur Interprozesskommunikation eingesetzt. Darauf wird in der zweiten Aufgabe detaillierter eingegangen. Sockets sind bidirektional und der jeweilige Endpunkt des Kommunikationskanals. Über diesen Kanal können Anfragen gesendet und auch die entsprechenden Antworten empfangen werden. Client und Server besitzen einen eigenen Socket, dieser besteht aus Ziel- bzw. Quell-IP-Adresse, Ziel- bzw. Quellport sowie dem zu verwendenden Protokoll [2].

#### 2.1.1 Teilaufgabe A

In Teilaufgabe A wird die Grundstruktur für eine Socketkommunikation aufgebaut. Empfängt der gestartete Server ein PING Kommando antwortet dieser mit einem PONG.

Der erstellte Quellcode unter der eigenen IP-Adresse (localhost) einen Socket zu erstellen. Dazu wird weiterhin ein Port benötigt, der über ein define vorgegeben wird. In diesem Beispiel wird der Port 12345 genutzt. Nach der Socketerstellung wartet der Server auf eine eingehende Verbindung. Anschließend wird die Chat-Funktion aufgerufen, die auf eine eingehende Nachricht wartet, diese in einem Buffer speichert, auswertet in der Kommandozeile ausgibt.

Getestet werden kann dieser einfache Server mit netcat:

```
nc 192.168.178.123
```

Sendet man ein PING an den Server wird im Terminal PONG angezeigt. Alle anderen Nachrichten werden im Server angezeigt, senden aber keine Antwort an den Client.

#### 2.1.2 Teilaufgabe B

Im zweiten Aufgabenteil wird eine eigene Client Anwendung entwickelt. Außerdem wird der Server erweitert, sodass er mehrere Clienten verwalten kann. Über Fork wird jede Clientverbindung in einen neuen Kindprozess ausgelagert. Dort wird dann die Chatfunktion wie in Aufgabenteil A ausgeführt. Der Hauptprozess kann dann weiterhin neue Verbindungen akzeptieren, da er nicht mehr mit der Bearbeitung der Chatfunktion beschäftigt ist.

Die Clientanwendung nutzt die gleichen C-Bibliotheken wie der Server. Es wird ein eigener

Socket erstellt und anschließend eine Verbindung zum Server aufgebaut. Über defines ist dessen IP und der Port festgelegt. Bei mehreren Servern ist es sinnvoll den Code um eine Nutzereingabe zu ergänzen, die IP und PORT abfragt, wie es auch bei netcat umgesetzt ist.

Nach einer erfolgreichen Verbindung zum Server wartet das Clientprogramm auf eine Nutzereingabe und sendet diese an den Server. Gibt der Nutzer QUIT ein, wird die Verbindung geschlossen. Alle eventuellen Antworten vom Server werden auf der Kommandozeile ausgegeben. Die erstellte Clientanwendung kann mehrmals gestartet werden. Alle gestarteten Instanzen können Nachrichten an den Server senden und erhalten bei einem PING Kommando auch ein PONG.

### 2.1.3 Teilaufgabe C

Für die Implementierung der editierbaren Listenfunktion muss nur der Server angepasst werden. Der Nutzer muss dann nur die neuen Befehle kennen, die er im Clientprogramm eingeben muss. Im Server wird ein Log eingeführt, dass alle ein und ausgehenden Nachrichten mit einem Zeitstempel abspeichert. Der Client kann dann das Log vom Server anfordern und den Text von einzelnen Einträgen verändern. Geänderte Einträge werden im Log markiert.

Die Behandlung des Log wird in einer eigenen Funktion umgesetzt und die bisherige Chat Funktion wird zur comm Funktion. Das Log besteht aus einem Array in dem insgesamt 100 Nachrichten gespeichert werden können. Die Log Funktion wird bei Empfang einer neuen Nachricht bzw. beim versenden einer Antwort aufgerufen. Der Nutzer kann über GETLOG die Liste anfordern, EDITLOG *jid*; *jtext*; bearbeitet den angegebenen Eintrag.

Dadurch das für die Übertragung des Logs mehrere Nachrichten vom Server nacheinander gesendet werden müssen, muss das Verfahren im Client geändert werden. Bisher erfolgt immer ein Wechsel zwischen senden und empfangen. Der Empfang wird um ein Timeout erweitert. Trifft 200 ms keine Nachricht mehr beim Client ein, wird zur Eingabe übergegangen. Dies funktioniert im lokalen Netzwerk ohne Probleme. Sollen die Anwendungen über größere Entfernung kommunizieren muss der Timeout erhöht oder ein anderes Konzept verwendet werden.

### 2.1.4 Teilaufgabe D

In der Teilaufgabe D wird die Server-Client Architektur in eine Peer-to-Peer Kommunikation umgewandelt. Für die Sockets wird weiterhin ein Verwalter benötigt. Der erste Aufruf der Anwendung startet den Serversocket. Im gleichen Terminal kann dann noch

eine Clientverbindung aufgebaut werden, die die bekannten Befehle ausführen kann. Jeder weitere Aufruf gibt den Hinweis zurück, dass der Serversocket nicht erstellt werden konnte, da dieser schon belegt ist. Danach kann die Verbindung zum anderen Partner aufgebaut werden. Mit einem CONNECT `\ip\` kann die Verbindung hergestellt werden. Alternativ kann mit EXIT die Anwendung beendet werden.

Mit Threads werden die einzelnen zu erledigenden Aufgaben parallel ausgeführt. Der Serverthread wird nur vom ersten Aufruf gestartet, dort wird auch die bereits vorhanden comm Funktion aufgerufen. Alle weiteren Instanzen starten den Inputthread, der die Verbindungen zum Server aufbaut. Der Thread ruft dann eine Clientfunktion auf, die das bisherige separate Programm ersetzt.

## 2.2 Interprozesskommunikation

Für die IPC können mehrere Varianten gewählt werden. Für die Bearbeitung der nachfolgenden Aufgaben wurden Message Queues verwendet.

Message Queues verwalten Nachrichten wie verkettete Listen. Die Listen verfügen über eine Kennung durch die sie durch Prozesse identifiziert werden können. Von einem Prozess gesendete Nachrichten werden gespeichert und können vom Partnerprozess abgerufen werden. Der Prozess kann seinerseits auch Nachrichten in der Queue ablegen. Standardmäßig arbeiten die Queues nach dem FIFO-Prinzip. Es gibt jedoch die Möglichkeit Nachrichten zu priorisieren, indem man ihnen einen eigenen Nachrichtentyp zuweist. Der Prozess sucht dann in der Liste nach der ersten Nachricht dieses Typs auch wenn sie ganz am Ende der Queue stehen sollte [2].

### 2.2.1 Teilaufgabe A und B

Diese Teilaufgaben erstellen die Grundstruktur für die Kommunikation. Für den Test der main-Anwendung wird gleich ein Subscriber geschrieben, sodass die Aufgaben A und B gemeinsam gelöst werden.

Von der main-Anwendung wird eine Queue geöffnet über die die Subscriberprozesse den Hauptprozess erreichen können. Zum öffnen einer Queue wird ein eindeutiger Schlüssel benötigt. Der Erzeugerfunktion werden außerdem die Berechtigungen mitgegeben, die die geöffnete Queue haben soll. Bei Bedarf können so Listen erstellt werden, die nur gelesen werden können. Subscriber können dann über diesen Kanal keine Nachrichten senden. Die main-Anwendung wartet in einer Schleife auf eine neue Nachricht in der Queue. Die abgerufen Nachricht wird in einer Struktur gespeichert, die den Nachrichtentyp und den eigentlichen Nachrichtentext enthält. Der Nachrichtentext wird anschließend auf bestimmte Kommandos geprüft. Auf eine PING Nachricht wird mit einem PONG an den Subscriber geantwortet.

Die Subscriberanwendung greift auf die erstellte Queue von main zu. Der Nutzer kann über die Kommandozeile eine Nachricht eingeben, die dann in die Queue gesetzt wird. Anschließend wird auf die Antwort des Servers gewartet.

### 2.2.2 Teilaufgabe C

### 2.2.3 Teilaufgabe D

### **3 Grundkonzepte von ROS2 mit einem Versuchsroboter**

Ziel dieses Kapitels ist es, die grundlegenden Konzepte und Arbeitsweisen von ROS 2 praxisnah zu erlernen und anhand eines Versuchsroboters schrittweise umzusetzen. Dazu werden zunächst die für das Praktikum erforderliche Arbeitsumgebung eingerichtet, die wesentlichen ROS 2-Komponenten erläutert und ein erstes eigenes Paket erstellt. Im weiteren Verlauf werden die Prinzipien der Kommunikation zwischen Nodes, Topics und Nachrichtenformaten anhand der *turtlesim*-Simulation demonstriert. Dieses Beispiel dient als Grundlage, um das Zusammenspiel zwischen Steuerkommandos, Nachrichtenübertragung und der Bewegung eines Roboters nachzuvollziehen.

Darauf aufbauend werden die gewonnenen Erkenntnisse später auf reale Hardware übertragen, indem ein ESP32-Mikrocontroller als Micro-ROS-Node in das System integriert wird. So entsteht ein durchgängiger Übergang von der Simulation im ROS-Umfeld hin zur physikalischen Roboterplattform, bei dem Konzepte wie Publisher/Subscriber-Kommunikation, Timings, QoS-Profile und Topic-Strukturen angewendet werden.

#### **3.1 Einrichtung der Arbeitsumgebung und ROS 2-Paket**

##### **3.1.1 Installation von ROS 2 Humble**

Die Arbeit erfolgt auf Basis von Ubuntu 22.04 LTS, um die vom Praktikum vorgegebene Systemumgebung sicherzustellen. Anschließend wurde ROS 2 Humble gemäß der offiziellen Anleitung installiert.<sup>1</sup>

Die Installation erfolgte auszugsweise in folgenden Schritten:

1. Konfiguration der Systemlokalisierung:

```
sudo locale-gen en_US en_US.UTF-8  
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
```

2. Einbinden der ROS 2-Repositories und des GPG-Schlüssels:

```
sudo apt update
```

---

<sup>1</sup><https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html>

```
sudo apt install software-properties-common curl
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/
master/ros.asc |
sudo tee /usr/share/keyrings/ros-archive-keyring.gpg > /dev/null
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/
share/keyrings/ros-archive-keyring.gpg]
http://packages.ros.org/ros2/ubuntu jammy main" |
sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

3. Installation von ROS 2 Desktop und weiteren Tools:

```
sudo apt update
sudo apt install ros-humble-desktop
sudo apt install ros-dev-tools
```

4. Aktivierung der ROS-Umgebung:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Zusätzlich wurde das Build-Tool *colcon* installiert:

```
sudo apt install python3-colcon-common-extensions
```

### 3.1.2 Erstellung eines ROS 2-Workspaces

Zur strukturierten Organisation eigener Pakete wurde ein Arbeitsbereich eingerichtet:

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws
colcon build
source install/setup.bash
```

### 3.1.3 Ergebnisse

Die Installation von ROS 2 Humble verlief nach anfänglicher Konfiguration der Paketquellen und Systemlokalisierung problemlos. Durch die Nutzung der offiziellen Ubuntu-

Repositorys konnte eine stabile und vollständig kompatible Entwicklungsumgebung eingerichtet werden.

Mit der Einrichtung des ROS 2-Workspaces und der dauerhaften Aktivierung der ROS-Umgebungsvariablen in der `.bashrc`-Datei war die Grundlage für die Entwicklung und Ausführung eigener Pakete geschaffen. Darüber hinaus wurde mit der Installation des Build-Systems `colcon` eine einheitliche und reproduzierbare Build-Struktur sichergestellt.

Damit war die technische Umgebung vollständig vorbereitet, um im weiteren Verlauf die Simulationsaufgaben, die Micro-ROS-Kommunikation, die Integration des ESP32 und die praktische Steuerung des Roboters umzusetzen.

## 3.2 Erstinbetriebnahme

Nach der erfolgreichen Installation von ROS 2 Humble erfolgte die erste praktische Arbeit mit der vorinstallierten Simulationsumgebung *turtlesim*. Die *turtlesim*-Anwendung stellt eine virtuelle Schildkröte bereit, die über Geschwindigkeitkommandos bewegt werden kann.

### 3.2.1 Grundprinzip von Nodes und Topics

In ROS basiert die Kommunikation auf einem Publish/Subscribe-Prinzip. Ein Publisher sendet Nachrichten an ein bestimmtes Topic, während ein oder mehrere Subscriber diese Nachrichten empfangen. Jede laufende Anwendung ist dabei eine Node. Bei *turtlesim* gibt es beispielsweise die Node *turtlesim\_node*, die das grafische Fenster und die Turtle darstellt, sowie die Node *teleop\_turtle*, die Tastatureingaben des Benutzers in Geschwindigkeitskommandos übersetzt.

### 3.2.2 Start der Turtlesim-Umgebung

Zum Starten der Umgebung werden in zwei separaten Terminals die folgenden Befehle ausgeführt:

```
# Terminal 1: startet die Simulation  
ros2 run turtlesim turtlesim_node  
  
# Terminal 2: startet die Steuerung per Tastatur  
ros2 run turtlesim turtle_teleop_key
```

Mit den Tasten *W*, *A*, *S*, *D* lässt sich die Turtle nach vorne, links, rückwärts und rechts bewegen. Dabei publiziert *turtle\_teleop\_key* fortlaufend Nachrichten vom Typ *geometry\_msgs/msg/Twist* an das Topic */turtle1/cmd\_vel*, das wiederum von der Node *turtlesim\_node* abonniert wird. Dieses Zusammenspiel bildet das klassische ROS-Kommunikationsschema aus Publisher, Subscriber und Topic ab.

### 3.2.3 Untersuchung der ROS-Kommunikation

ROS 2 stellt zahlreiche Befehle zur Analyse und Kontrolle der Systemkommunikation bereit, die in den ersten Schritten besonders hilfreich sind:

- *ros2 node list*

Zeigt alle aktuell laufenden Nodes an, z. B.:

*/turtlesim*

*/teleop\_turtle*

- *ros2 topic list*

Listet alle aktiven Topics auf. Typischerweise erscheinen u. a.:

*/turtle1/cmd\_vel*

*/turtle1/pose*

- *ros2 topic info /turtle1/cmd\_vel*

Zeigt an, welcher Nachrichtentyp und welche Nodes an diesem Topic beteiligt sind (Publisher/Subscriber).

- *ros2 topic echo /turtle1/cmd\_vel*

Gibt die in Echtzeit übertragenen Nachrichten auf der Konsole aus. Dadurch kann beobachtet werden, welche linearen und rotatorischen Geschwindigkeiten beim Drücken einer Taste publiziert werden.

- *ros2 interface show geometry\_msgs/msg/Twist*

Zeigt den Aufbau der *Twist*-Nachricht, die aus den Vektoren *linear* (für Vorwärts-/Rückwärtsbewegung) und *angular* (für Rotationen) besteht.

Diese Befehle verdeutlichen die lose Kopplung der ROS-Komponenten und erleichtern das Verständnis, wie Daten in einem ROS-System fließen.

Nach dem erfolgreichen Start der *turtlesim*-Simulation wurden eigene Nodes erstellt, die selbstständig Nachrichten an das Topic */turtle1/cmd\_vel* senden. Dadurch konnte die Turtle automatisiert bewegt werden. Dieses Vorgehen bildet die Grundlage für komplexe Szenarien, wie etwa die spätere Übertragung derselben Steuerbefehle an den realen Roboter.

## 4 TurtleBot mit ESP32

### 4.1 Arbeitsumgebung

Die Implementierung und Inbetriebnahme des Micro-ROS-Systems erfolgte nativ unter **Ubuntu 22.04.5 LTS**. Die grundlegende Installation der benötigten Softwarekomponenten wie *ROS 2 Humble*, der *Micro-ROS-Agent* sowie die Einbindung von *PlatformIO* in *Visual Studio Code* erfolgte gemäß der offiziellen Praktikumsanleitung [3]. Im Folgenden werden daher weniger die Installationsschritte selbst beschrieben, sondern vielmehr die Erkenntnisse und Besonderheiten der praktischen Arbeitsumgebung zusammengefasst.

#### 4.1.1 Betriebssystem und Systemkonfiguration

Da die Arbeit auf einem nativen Ubuntu-System ausgeführt wurde, war keine zusätzliche Virtualisierung notwendig. Dies bot Vorteile hinsichtlich der direkten Hardwareambindung, insbesondere bei der seriellen Kommunikation mit dem ESP32. Während der Einrichtung stellte sich heraus, dass unter Ubuntu 22.04 standardmäßig bestimmte Kernelmodule geladen werden, die mit der seriellen Kommunikation über USB kollidieren. Insbesondere der *Braille-Treiber (brltty)* beansprucht häufig automatisch USB-Serial-Geräte (z.B. */dev/ttyUSB0*), wodurch keine Verbindung zu Mikrocontrollern aufgebaut werden kann. Das Deinstallieren bzw. Deaktivieren dieses Dienstes war zwingend erforderlich, um eine stabile Verbindung mit dem ESP32 herstellen zu können:

```
sudo apt remove brltty
```

Weiterhin muss der Benutzer dem System die Berechtigung geben, auf serielle Schnittstellen zuzugreifen. Dies geschieht über die Zuweisung zur Benutzergruppe *dialout*:

```
sudo usermod -a -G dialout $USER
```

Nach dieser Änderung ist eine Ab- und erneute Anmeldung erforderlich, damit die neuen Gruppenrechte aktiv werden.

#### 4.1.2 Entwicklungsumgebung PlatformIO

Die Firmware wurde in der Entwicklungsumgebung **PlatformIO** erstellt und über die serielle Schnittstelle direkt auf den ESP32 geflasht. PlatformIO bietet im Vergleich zur Arduino-IDE eine deutlich bessere Integration in ROS-Projekte, da Bibliotheken wie *micro\_ros\_platformio* komfortabel über die Projektkonfiguration (*platformio.ini*) eingebunden werden können.

Im praktischen Einsatz zeigte sich, dass beim Flashen des Mikrocontrollers über USB der serielle Monitor zunächst geschlossen sein muss, da andernfalls der Zugriff auf den Port blockiert ist. Ebenso hilfreich war es, den automatischen Neustart (*monitor\_dtr* und *monitor\_rts*) in der Konfiguration vorübergehend zu deaktivieren, falls das Board während des Uploads nicht zuverlässig in den Bootloader-Modus wechselte.

#### 4.1.3 Netzwerk- und Agentumgebung

Die Kommunikation zwischen Laptop und Roboter erfolgte über WLAN im *Station Mode* des ESP32. Als Micro-ROS-Agent diente das lokale Ubuntu-System, auf dem die UDP-Verbindung über den Port 8888 bereitgestellt wurde. Da der Laptop selbst als WLAN-Hotspot fungierte, konnte eine stabile Punkt-zu-Punkt-Verbindung hergestellt werden, ohne die Hochschulnetzwerke zu belasten. Diese Konfiguration erwies sich als zuverlässig und reproduzierbar, insbesondere für Tests mit wiederholtem Re-Flashen und Neustarten des Controllers.

Auf Seiten des ESP32 mussten die WLAN-Zugangsdaten und die Agent-IP-Adresse direkt im Programmcode angepasst werden, um die Verbindung herzustellen:

```
#define WIFI_SSID "ROS_Net"
#define WIFI_PASSWORD "EZMR"
#define AGENT_IP_STR "10.42.0.1"
#define AGENT_PORT 8888
```

Zusätzlich war in der Datei *platformio.ini* sicherzustellen, dass die Micro-ROS-Bibliothek eingebunden und der richtige Transportmechanismus aktiviert ist:

```
lib_deps = https://github.com/micro-ROS/micro_ros_platformio
board_microros_distro = humble
```

```
board_microros_transport = wifi
```

Im Code selbst erfolgte die Initialisierung des Micro-ROS-Transports anschließend über:

```
set_microros_wifi_transports((char*)WIFI_SSID, (char*)WIFI_PASSWORD,  
agent_ip, AGENT_PORT);
```

Diese Parameter legten fest, dass die Kommunikation nicht über USB-Serial, sondern direkt über WLAN mittels UDP-Paketen zwischen ESP32 und Micro-ROS-Agent auf dem Laptop abgewickelt wurde. Damit war der Datenaustausch über */cmd\_vel* in Echtzeit möglich und das System blieb auch nach Neustarts oder Re-Upserts des Controllers stabil verbunden.

#### 4.1.4 Erkenntnisse

Während der Arbeit wurde deutlich, dass die Einrichtung der Entwicklungsumgebung zwar durch die Anleitung gut beschrieben ist, in der Praxis aber häufig systemabhängige Details den Fortschritt behindern können. Insbesondere der Umgang mit USB-Geräten unter Linux erfordert systemseitige Konfiguration, da schon ein belegter Port oder ein falsch gesetztes Gruppenrecht dazu führen kann, dass der Micro-ROS-Agent keine Verbindung zum ESP32 aufbauen kann. Nach erfolgreicher Systemkonfiguration verlief die Entwicklung stabil, und PlatformIO erwies sich als zuverlässige Plattform für den kontinuierlichen Test- und Upload-Prozess.

## 4.2 Turtlesim Teleop

### 4.2.1 Micro-ROS-Struktur und Node-Konfiguration

Der ESP32 agiert als *Micro-ROS Node* und abonniert das Topic */cmd\_vel*, das Geschwindigkeitsbefehle vom ROS 2-System enthält. Diese werden als Nachrichten des Typs *geometry\_msgs/msg/Twist* per WLAN-Hotspot, mit welchem sich der TurtleBot verbündet, übertragen und bestehen aus einer linearen und einer angularen Komponente. Auf dem Host-PC wird der Micro-Ros Agent in einem separaten Terminal gestartet.

```
ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888
```

Die Node verarbeitet die Bewegungsdaten innerhalb einer Callback-Funktion, die bei jedem neuen Nachrichtenempfang ausgelöst wird. Dadurch werden nur dann Befehle an den Roboter gesendet, wenn tatsächlich eine neue Nachricht vorliegt. So wird die Kommunikationslast auf dem UART reduziert und vermeidet unnötige Wiederholungen.

#### 4.2.2 Kommunikation Motorcontroller

Die Steuerung der Antriebe erfolgt über zwei ATTiny-basierte Motorcontroller, die über UART (9600 Baud, 8N1) mit dem ESP32 verbunden sind. Beide Motoren besitzen eine eigene Adresse (*0x01* für links, *0x02* für rechts) und werden über ein einheitliches Protokoll angesprochen:

[Startbyte] [ID] [Mode] [Dir] [Param\_High] [Param\_Low]

Das Nachrichtenformat beginnt mit einem festen Startbyte *0xAA*, gefolgt von der Motor-ID und dem Betriebsmodus. *Dir* legt die Drehrichtung fest, während die *Param\_High* und *Param\_Low* die Geschwindigkeit kodieren. Dabei wird nur das niederwertige Byte (*Param\_Low*) aktiv genutzt, um Werte im Bereich von *0x10* (langsam) bis *0x02* (schnell) abzubilden.

Der Modus *Mode = 1* steht für den Dauerlauf, während *Mode = 0* den Schrittmodus aktiviert. Bei einem Stoppbefehl wird *Mode = 0* mit *Param\_Low = 0x00* übertragen, wodurch der Motor unmittelbar anhält, ohne eine Restbewegung auszuführen.

#### 4.2.3 Implementierungsdetails

Die Verarbeitung der */cmd\_vel* - Nachrichten erfolgt in der Callback-Funktion, die den linearen und den angularen Geschwindigkeitsanteil in Radgeschwindigkeiten umrechnet. Die Kinematik des Roboters basiert auf einem Differentialantrieb mit einem Radabstand von 0,168 m und einem Radradius von 0,042 m. Der folgende Zusammenhang wird verwendet:

$$\begin{aligned}v_r &= v + \omega \cdot b \\v_l &= v - \omega \cdot b\end{aligned}$$

wobei  $v$  die lineare und  $\omega$  die Winkelgeschwindigkeit ist,  $v_r$  und  $v_l$  die jeweiligen Radgeschwindigkeiten, und  $b$  der halbe Spurabstand.

Zusätzlich wurde eine Verstärkungslogik implementiert, um Drehbewegungen auf der Stelle zu beschleunigen. Wird erkannt, dass die lineare Geschwindigkeit nahezu null ist ( $|v| < 0.05$ ) und eine signifikante Drehgeschwindigkeit ( $|\omega| > 0.01$ ) anliegt, wird die Winkelgeschwindigkeit mit einem Faktor von drei multipliziert. Dadurch reagiert der Roboter beim Drehen deutlich agiler, ohne das allgemeine Fahrverhalten zu verändern.

#### 4.2.4 Fehlerquellen und Besonderheiten

Während der Entwicklung traten wiederholt Probleme mit der Synchronisation zwischen ROS-Agent und ESP32 auf, insbesondere bei WLAN-Reconnects oder unvollständigen Uploads. Hier waren einfache Neustarts der Hardware und das Drücken des Enable-Tasters des ESP32 wirksam.

Besonders zu beachten ist die korrekte Verwendung der Tastatursteuerung für die *turtlesim*-Simulation. Wird das Standardprogramm *turtle\_teleop\_key* gestartet, kommuniziert es direkt mit der *turtlesim\_node*, wodurch keine Nachrichten an das Topic */turtle1/cmd\_vel* weitergeleitet werden. Für die Integration mit dem eigenen Node (bzw. mit dem realen Roboter) muss daher folgender Befehl verwendet werden:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -r  
/cmd_vel:=/cmd_vel
```

Nur mit diesem Aufruf werden die Steuerbefehle als *geometry\_msgs/msg/Twist*-Nachrichten auf das Topic */turtle1/cmd\_vel* publiziert und können somit korrekt vom eigenen *cmd\_vel*-Subscriber empfangen und umgesetzt werden.

Alternativ kann das Problem auch durch ein sogenanntes Topic Remapping gelöst werden. Dabei wird beim Start einer Node das Ziel-Topic dynamisch umbenannt. So kann die Standardsteuerung von *turtlesim* weiterhin verwendet werden, während die Befehle automatisch an das gewünschte Topic weitergeleitet werden:

```
ros2 run turtlesim turtle_teleop_key --ros-args -r  
/turtle1/cmd_vel:=/cmd_vel
```

Diese Methode ist besonders praktisch, wenn dieselbe Node sowohl in der Simulation als auch auf einem realen Roboter eingesetzt werden soll, da keine Änderungen am Quellcode notwendig sind.

## Ergebnisse

Das System reagierte stabil und nachvollziehbar auf die von der Tastatursteuerung gesendeten Befehle. Die Integration über Micro-ROS erwies sich dabei als effizient, da nur bei tatsächlichem Empfang neuer Nachrichten UART-Kommandos an die Motorcontroller weitergeleitet wurden. Dies reduzierte die Kommunikationslast erheblich und trug zu einem gleichmäßigen Fahrverhalten bei.

Die Implementierung der zeitbasierten Logik für das Drehen auf der Stelle erwies sich als praktikabel und führte zu einer deutlich dynamischeren Reaktion des Roboters. Das gewählte Kommunikationsschema mit *Mode*- und *Param*-Bytes ermöglichte eine effektive Übersetzung der ROS-Twist-Nachrichten in serielle Steuerkommandos.

## 4.3 Abfahren von Polygonen

Zur Überprüfung der Funktionsfähigkeit der Implementierung wurde zunächst die *turtlesim*-Simulation genutzt und anschließend das gleiche Steuerungsskript auf den realen Roboter übertragen. Die entwickelte Node *shape\_ESP\_Turtle.py* wird auf dem PC gestartet und publiziert periodisch Geschwindigkeitsbefehle an das Topic */cmd\_vel*, wodurch die Turtle oder der Roboter ein vorgegebenes geometrisches Muster abfährt. Das Verhalten der Simulation diente dabei als Referenz für die physische Plattform.

Zur reproduzierbaren Bewegung der Turtle (bzw. eines realen Roboters mit identischem Interface) wurde ein Python-Knoten *ShapeDriver* implementiert. Der Knoten publiziert zeitbasiert Geschwindigkeitsbefehle an ein konfigurierbares */cmd\_vel* - Topic und lässt die Turtle wahlweise ein Dreieck oder ein Viereck abfahren. Das Skript ist generisch gehalten (Themenname, Kantenlänge, lineare- und Winkelgeschwindigkeit sind parametrierbar) und arbeitet vollständig timergesteuert.

### 4.3.1 Ziel und Aufbau

Der Knoten erbt von `rclpy.node.Node` und erstellt einen Publisher `geometry_msgs/msg/Twist` auf dem konfigurierten Topic (Standard: `/turtle1/cmd_vel`). Ein periodischer ROS-Timer (20 Hz) ruft die Steuerfunktion auf und publiziert in jeder Periode entweder eine reine Vorwärtsfahrt (gerade Strecke) oder eine reine Drehung (Ecken). Die gewünschte Form wird über das Argument `shape` gewählt (`square` oder `triangle`).

### 4.3.2 Steuerlogik

Die Bewegung wird als endliche Zustandsmaschine mit zwei Phasen modelliert:

- **straight:** Vorwärtsfahrt mit Geschwindigkeit  $v_{\text{lin}}$  während der Dauer  $t_{\text{straight}}$ .
- **turn:** Drehung auf der Stelle mit Winkelgeschwindigkeit  $v_{\text{ang}}$  mit Dauer  $t_{\text{turn}}$ .

Die Zeiten werden aus Geometrie und Geschwindigkeiten berechnet:

$$t_{\text{straight}} = \frac{\text{Kantenlänge}}{v_{\text{lin}}}, \quad \alpha = \frac{2\pi}{N}, \quad t_{\text{turn}} = \frac{\alpha}{v_{\text{ang}}},$$

wobei  $N \in \{3,4\}$  die Anzahl der Kanten (Dreieck/Viereck) ist und  $\alpha$  der Außenwinkel zwischen zwei Kanten. Die Phasen wechseln, sobald die jeweilige Sollzeit durch die Timer-Takte kumulativ erreicht ist. Nach  $N$  vollständigen Kanten wird explizit `Twist(0,0)` gesendet, der Timer gestoppt und die Node sauber beendet.

### 4.3.3 Implementierungsdetails

- **Zeitdiskretisierung:** Die Steuerfrequenz beträgt 20 Hz (`rate_hz=20`), d. h. die Timer-Periode ist  $\Delta t = 0.05$  s. Eine interne Uhr zählt `phase_time` in Schritten von  $\Delta t$ , bis  $t_{\text{straight}}$  bzw.  $t_{\text{turn}}$  erreicht ist.
- **Robustheit der Parameter:** Die lineare und Winkelgeschwindigkeit werden als Beträge verarbeitet (Vorzeichen wird nicht benötigt, da für Drehungen immer links/herum gedreht wird). Divisionen sind durch `max(v, 1e-6)` gegen  $v = 0$  abgesichert.
- **Formwahl:** `shape` akzeptiert `square/rectangle` ( $N = 4$ ) sowie `triangle` ( $N = 3$ ). Daraus wird  $\alpha = 2\pi/N$  abgeleitet.

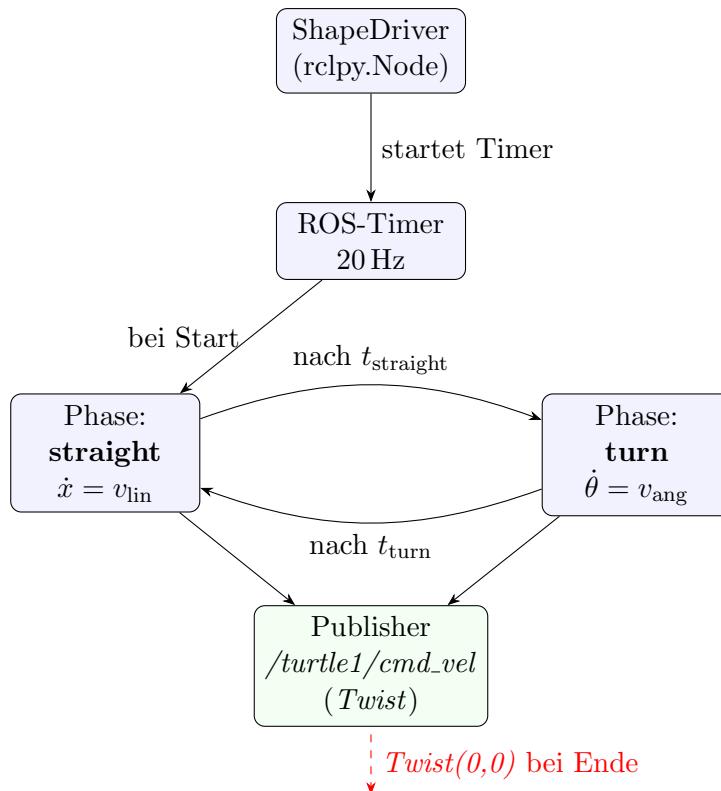


Abbildung 1: Funktionsprinzip der zeitbasierten Steuerung des *ShapeDriver*-Knotens.

- **Topic-Name:** Das Ausgabefeld ist frei wählbar (*-topic*), sodass derselbe Knoten sowohl die *turtlesim* (*/turtle1/cmd\_vel*) als auch reale Roboter (*/cmd\_vel*) bedienen kann.
- **Saubereres Beenden:** Nach Abschluss der letzten Drehphase wird ein Stopp (*linear.x=0, angular.z=0*) publiziert, der Timer gecancelt und *rclpy.shutdown()* aufgerufen, um Ressourcen freizugeben.

#### 4.3.4 Aufruf und Parametrisierung

Der Knoten akzeptiert Kommandozeilenargumente (über *argparse*):

- **-shape** (*square/viereck/triangle/dreieck*), Standard: *square*
- **-side** Kantenlänge (Turtlesim-Einheiten), Standard: *2.0*
- **-v\_lin** lineare Geschwindigkeit in m/s, Standard: *1.0*
- **-v\_ang** Winkelgeschwindigkeit in rad/s, Standard: *2.0*
- **-topic** Ausgabetopic (z. B. */turtle1/cmd\_vel* oder */cmd\_vel*)

Typische Aufrufe:

- Viereck, Kante 2.0, Standardgeschwindigkeiten, Turtlesim:

```
ros2 run <paketname> <executable> -- --shape square --side 2.0
--topic /turtle1/cmd_vel
```

- Dreieck, etwas langsamer, realer Roboter:

```
ros2 run <paketname> <executable> -- --shape triangle --side 0.8
--v_lin 0.5 --v_ang 1.5 --topic /cmd_vel
```

#### 4.3.5 Beobachtungen

In Abbildung 2 folgt die Turtle der Turtlesim der Form erwartungsgemäß. Kleine Abweichungen können durch die zeitdiskrete Approximation (Timer-Raster) und die Integration der Geschwindigkeiten entstehen. Auf realer Hardware weichen die Trajektorien erfahrungsgemäß stärker ab (u. a. durch Schlupf, Latenzen, nichtlineare Antriebskennlinien). Für reproduzierbarere Ergebnisse wäre eine schritt- oder sensorbasierte Regelung sinnvoll.

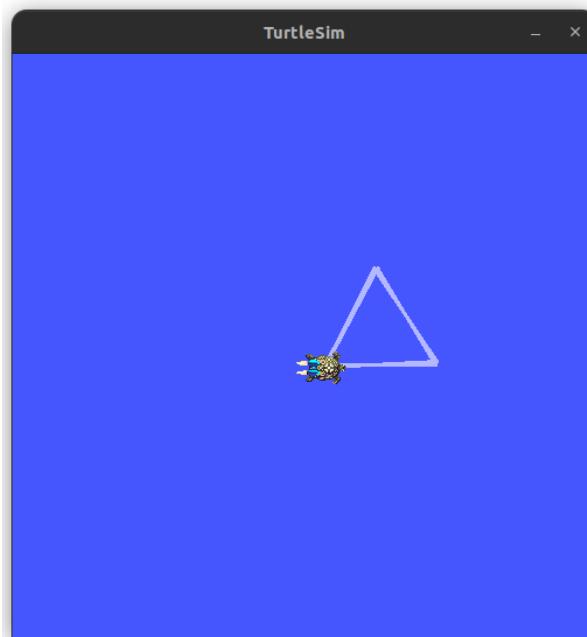


Abbildung 2: Fahrspur der Turtlesim-Node

## 4.4 Sensorintegration: Bumper mit Host-seitiger Sicherheitslogik

### 4.4.1 Zielsetzung

Ziel ist eine robuste Kollisionsreaktion, die ohne eigenständige lokale Autonomie auf dem Mikrocontroller auskommt. Der ESP32 übernimmt dabei ausschließlich die Erfassung der Bumper-Zustände und deren Publikation in das ROS-Netzwerk sowie die Ausführung eingehender Fahrbefehle vom Host-PC. Die *Safety Node* auf dem Host-System überwacht fortlaufend die Bumper-Signale, filtert die Teleop-Kommandos und greift im Kollisionsfall aktiv ein. Wird ein Bumper ausgelöst, führt die Node automatisch ein definiertes Ausweichmanöver aus: zunächst eine kurze Rückwärtsfahrt, anschließend ein gezieltes Wegdrehen, gefolgt von einem Stopp und der kontrollierten Rückgabe der Steuerung an den Agent.

### 4.4.2 Systemübersicht und Topics

- **Teleoperator** (Host): publiziert Rohbefehle auf `/cmd_vel_raw`.
- **Safety-Node** (Host):
  - Subscript `/cmd_vel_raw (geometry_msgs/Twist)` und `/bumper/state (std_msgs/UInt8)`.
  - Publiziert `/cmd_vel (geometry_msgs/Twist)` an den Roboter.
- **ESP32** (Turtle):
  - Publiziert `/bumper/state` (Bitmaske: Bit0=links, Bit1=rechts, 1=gedrückt).
  - Subscript `/cmd_vel` und setzt die Fahrbefehle per UART auf die Motorcontroller um.

Zur Sicherstellung einer stabilen Kommunikation werden alle relevanten Topics mit der Quality-of-Service-Einstellung *RELIABLE* und *KEEP\_LAST* betrieben. Damit wird garantiert, dass keine Nachrichten verloren gehen und stets nur die jeweils aktuellste Nachricht im Puffer gehalten wird. Der ESP32 veröffentlicht das Topic `/bumper/state` bei jeder Zustandsänderung der Bumper (Abfragefrequenz 200 Hz mit kurzer Entprellzeit). Die Safety-Node auf dem Host-PC sendet während einer aktiven Übersteuerung regelmäßig Befehle mit 100 Hz, um ein gleichmäßiges Bewegungsverhalten sicherzustellen.

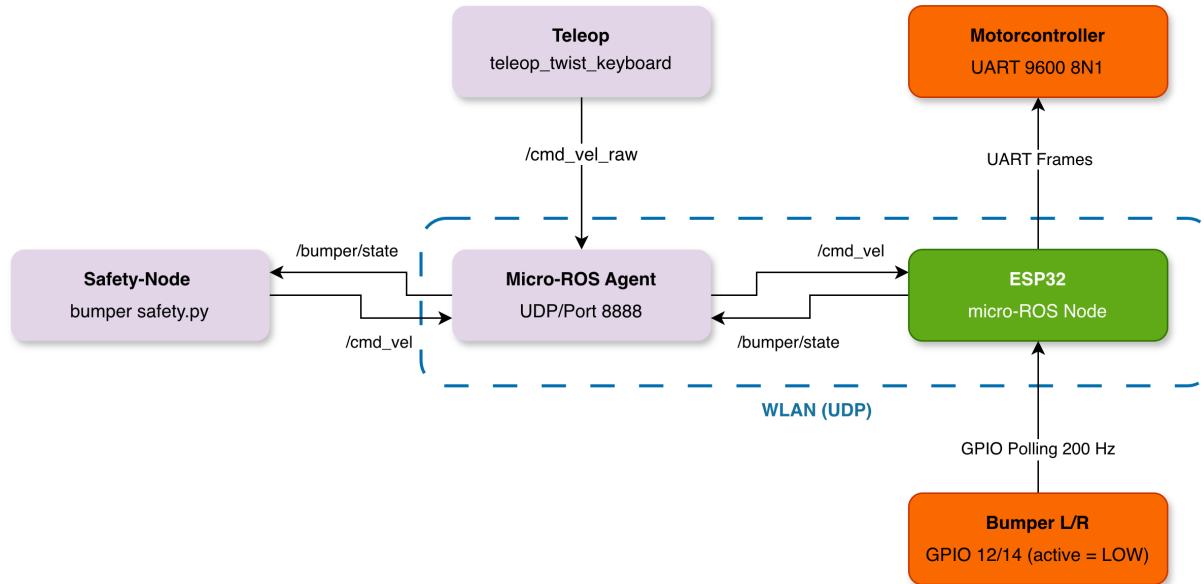


Abbildung 3: Blockdiagramm der Systemarchitektur

#### 4.4.3 Implementierung

##### ESP32 (Client)

- GPIOs: *Bumper links* = GPIO12, *Bumper rechts* = GPIO14, jeweils *INPUT\_PULLUP* (active-LOW).
- Publisher: `/bumper/state` (*std\_msgs/UInt8*, Bitmaske: *0x01* links, *0x02* rechts, *0x03* beide).
- Subscriber: `/cmd_vel` (*geometry\_msgs/Twist*); Umsetzung auf **UART-Frames** für die Motorcontroller (unverändert).
- Polling/Publish: Timer 5 ms (200 Hz), Debounce  $\approx$ 10 ms; Publish nur bei Zustandsänderung.

##### Safety-Node (Host, `bumper_safety.py`)

- Abonniert `/bumper/state` und `/cmd_vel_raw`, publiziert `/cmd_vel`.
- **Ablauf beim Event:** (1) Rückwärtsfahrt (konstantes  $v_{back}$ ) für  $t_{back}$ , (2) Drehung (konstantes  $\omega_{turn}$ ) für  $t_{turn}$ ; Drehrichtung gemäß Bumper: links gedrückt  $\rightarrow$  rechts drehen, rechts gedrückt  $\rightarrow$  links drehen, beide  $\rightarrow$  rechts drehen (Default).
- Am Ende: **Stop** publizieren, danach **Sperrphase**  $t_{hold}$  (Teleop wird ignoriert); optional erst Freigabe, wenn `/cmd_vel_raw` einmal 0/0 sendet.

- Taktung: 100 Hz während der Übersteuerung, sofortiges erstes Override-Kommando beim Event (minimale Latenz).

#### 4.4.4 Parameter der Safety-Node

Alle Parameter können zur Laufzeit per `-ros-args -p` gesetzt werden:

Name	Bedeutung	Default
<code>back_speed</code>	Rückwärtsgeschwindigkeit $v_{\text{back}}$ [m/s]	-0.25
<code>back_time</code>	Dauer der Rückwärtsfahrt $t_{\text{back}}$ [s]	2.25
<code>turn_speed</code>	Drehgeschwindigkeit $\omega_{\text{turn}}$ [rad/s]	1.2
<code>turn_time</code>	Dauer der Drehung $t_{\text{turn}}$ [s]	2.0
<code>hold_time</code>	Sperrphase nach Manöverende [s]	0.8
<code>require_zero_to_resume</code>	Freigabe erst nach Stoppkommando (0/0)	true
<code>raw_cmd_topic</code>	Eingangs-Topic (Teleop-Steuerung)	/cmd_vel_raw
<code>out_cmd_topic</code>	Ausgangs-Topic (Robotersteuerung)	/cmd_vel
<code>bumper_topic</code>	Eingang für Bumper-Zustände	/bumper/state

#### 4.4.5 Ergebnis

Mit der finalen Konfiguration reagiert der Roboter spürbar schneller auf Bumperkontakte. Das Event löst unmittelbar eine Rückwärtsbewegung aus, gefolgt von einer Ausweichdrehung in entgegengesetzte Richtung. Am Ende wird ein explizites Stop-Kommando gesendet, anschließend blockiert eine kurze Sperrphase neue Teleop-Befehle. Optional muss die Teleop einmal ein *Null*-Kommando senden, bevor die Steuerung wieder freigegeben wird. Die Fahrbefehle laufen ausschließlich über die Safety-Node (Teleop → `/cmd_vel_raw` → Safety → `/cmd_vel`).

#### 4.4.6 Probleme

Leider konnte die Ursprüngliche Idee, die Integration des Ultraschallsensors nicht erfolgen, da durch ungeklärte Umstände, keine Informationen an den Sensor gesendet werden

konnten. Der verwendete Ultraschallsensor war auf einer fest verlöteten Platine integriert, weshalb keine detaillierten Informationen zur internen Beschaltung vorlagen. Insbesondere die exakte Signalführung des *ECHO*-Pins konnte nicht überprüft werden, da kein Zugriff auf die Platinenrückseite bestand. Dadurch war es nicht möglich, das Pegelverhalten des Sensors direkt zu verifizieren oder gegebenenfalls mit einem Spannungsteiler anzupassen. In Folge dessen konnte mit dem Testskript kein gültiges Echo-Signal detektiert werden, obwohl die Ansteuerung über den *TRIG*-Pin funktionierte.

## **5 Steuerung eines professionellen Roboters mit ROS2**

## Literatur

- [1] OMRON. Autonome mobile Roboter. <https://industrial.omron.de/de/products/autonomous-mobile-robot> aufgerufen am 15.10.2025 12:35 Uhr.
- [2] Rene Krooß Jürgen Wolf. C von A bis Z. Rheinwerk, 2009.
- [3] Nico Beyer. Praktikum Echtzeitsysteme – Laborunterlagen. gespeichert am 22.08.2025.

## **Anhang**