



Hochschule für Technik,  
Wirtschaft und Kultur Leipzig

FAKULTÄT INGENIEURWISSENSCHAFTEN

E203 - ECHTZEITSYSTEME UND MOBILE ROBOTIK

---

**Belegarbeit Echtzeitsysteme und Mobile Robotik**

---

*Vorgelegt von:* Philipp Nöcker  
*Studiengang:* Elektrotechnik und Informationstechnik  
*Seminargruppe:* 24EIM-AT  
*Matrikelnummer:* 85980  
*Emailadresse:* philipp.noecker@stud.htwk-leipzig.de  
*Git-Repository:* <https://github.com/phlnoc/BelegEchtzeitSystemeMobileRobotik>

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>ii</b>
<b>1 Einführung ROS2</b>	<b>1</b>
1.1 Installation . . . . .	1
1.2 Zusammenfassung Grundlagen ROS 2 . . . . .	1
<b>2 Erstellung eigenes ROS2 Paket</b>	<b>3</b>
2.1 Erstellung eines Arbeitsbereiches und ROS 2 Paket . . . . .	3
2.2 Erstellung einer Publisher-Subscriber Kommunikation mit Python . . . . .	4
2.3 Erstellung eines eigenen Nodes zur Bewegungssteuerung . . . . .	5
<b>3 Programmierung eigener Roboter</b>	<b>9</b>
<b>4 Erweiterung durch Sensorik und eigene Projekte</b>	<b>13</b>
4.1 Distanzmessung und Displayausgabe . . . . .	13
4.2 „Diebstahlschutz“ . . . . .	16
4.3 Zusammenführung der Projekte . . . . .	19
<b>5 Arbeiten mit einem professionellen Roboter: TurtleBot 4</b>	<b>20</b>
<b>6 Fazit und Ausblick Praktikum</b>	<b>22</b>
<b>7 Sockets/ Netcode</b>	<b>23</b>
7.1 Teilaufgabe A und B . . . . .	23
7.2 Teilaufgabe C . . . . .	25
7.3 Teilaufgabe D . . . . .	26
<b>8 Interprozesskommunikation</b>	<b>27</b>
8.1 Simple IPC Infrastruktur . . . . .	27
8.2 IPC Listenmanipulation . . . . .	31
8.3 IPC Peer Struktur . . . . .	33
<b>9 Fazit Programmieraufgaben</b>	<b>33</b>
<b>Quellenverzeichnis</b>	<b>34</b>

## Abbildungsverzeichnis

1	Überprüfung der ROS 2 Installation durch Aufruf eines <code>talker</code> -Knotens (linkes Fenster) und eines <code>listener</code> -Knotens (rechtes Fenster) in separaten Terminals . . . . .	1
2	Vergleich der Turtlesim mit angepasstem Overlay im Workspace (linkes Fenster) und normal installierter Turtlesim (rechtes Fenster) . . . . .	4
3	Testung der erstellten Publisher Subscriber Kommunikation . . . . .	5
4	Ergebnis des einmaligen Ausführens des Square-Nodes . . . . .	7
5	Ergebnis des zweifachen Ausführens des Square-Nodes . . . . .	7
6	Vergleich der Ausführung des Square Nodes in der Turtlesim . . . . .	7
7	Ergebnis des Heart-Nodes in der Turtlesim . . . . .	8
8	Beispielanzeige des Displays nach publish auf das Display Topic . . . . .	16
9	Visualisierung der Sensordaten des /scan Topics . . . . .	21
10	Aufgenommene Karte durch die Kartierung . . . . .	21
11	Caption . . . . .	24
12	Test der Socket Peer Anwendung mit drei Peers . . . . .	26
13	Terminalauszug des Clients bei Testung Serverantwort . . . . .	29
14	Terminalauszug des Hörer-Clients bei der Testung der Kommunikation . . . . .	29
15	Übersicht der Tests mit ipcs . . . . .	30
16	Konsole mit Tests der Funktionalität zur Manipulation einer Liste über Shared-Memory IPC . . . . .	32

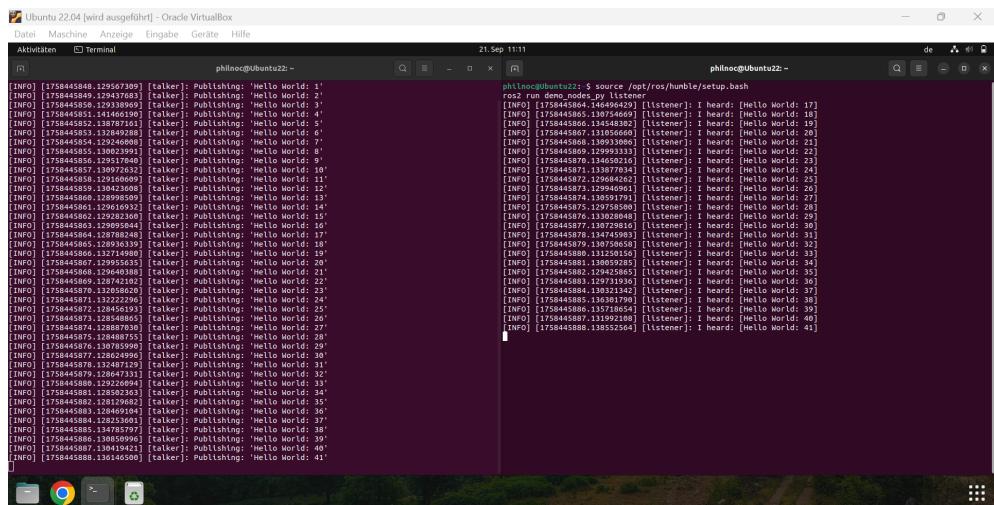
# 1 Einführung ROS2

## 1.1 Installation

Die Installation wurde gemäß der Praktikumsanleitung durchgeführt. Da ein Windows-Rechner verwendet wurde, wurde eine virtuelle Maschine erstellt, auf welcher ein Ubuntu-22.04-System erstellt wurde.

Nach der erfolgreichen Installation des Betriebssystems wurde gemäß der Anleitung die vollständige Desktop-Version von ROS 2 Humble installiert. Auch das Build Tool colcon wurde erfolgreich installiert.

Zu Überprüfung der Installation wurden, wie in der Praktikumsanleitung gefordert, die mitgelieferten Beispieldaten getestet. Dafür wurden zwei Terminals geöffnet und jeweils das Setup-File gesourct. Anschließend wurde in Terminal (linkes Fenster) ein **talker**-Knoten ausgeführt und in Terminal 2 (rechtes Fenster) ein **listener**-Knoten. Wie in Abb. 1 zusehen ist, sendet der **talker**-Knoten erfolgreich Nachrichten und der **listener**-Knoten empfängt diese erfolgreich. Es kann somit davon ausgegangen werden, dass die Installation von ROS 2 Humble erfolgreich verlief und nun die nächsten Schritte des Praktikums erfolgen können.



The screenshot shows two terminal windows on a Linux desktop. The left terminal window, titled 'phlinoc@Ubuntu22: ~', contains the command 'ros2 run demo\_nodes\_py talker' followed by a series of log messages from the 'talker' node publishing 'Hello World' messages at various time intervals. The right terminal window, also titled 'phlinoc@Ubuntu22: ~', contains the command 'ros2 run demo\_nodes\_py listener' followed by a series of log messages from the 'listener' node receiving 'Hello World' messages. Both terminals show the same sequence of messages, indicating successful communication between the two nodes.

```
[INFO] [1758445848.129567309] [talker]: Publishing: Hello World: 1'
[INFO] [1758445848.129567309] [talker]: Publishing: Hello World: 2'
[INFO] [1758445848.129567309] [talker]: Publishing: Hello World: 3'
[INFO] [1758445851.141466190] [talker]: Publishing: Hello World: 4'
[INFO] [1758445852.138787161] [talker]: Publishing: Hello World: 5'
[INFO] [1758445852.138787161] [talker]: Publishing: Hello World: 6'
[INFO] [1758445854.129246098] [talker]: Publishing: Hello World: 7'
[INFO] [1758445855.130923991] [talker]: Publishing: Hello World: 8'
[INFO] [1758445855.130923991] [talker]: Publishing: Hello World: 9'
[INFO] [1758445857.130972392] [talker]: Publishing: Hello World: 10'
[INFO] [1758445857.130972392] [talker]: Publishing: Hello World: 11'
[INFO] [1758445857.130972392] [talker]: Publishing: Hello World: 12'
[INFO] [1758445859.129619099] [talker]: Publishing: Hello World: 13'
[INFO] [1758445861.129616932] [talker]: Publishing: Hello World: 14'
[INFO] [1758445861.129616932] [talker]: Publishing: Hello World: 15'
[INFO] [1758445863.129695441] [talker]: Publishing: Hello World: 16'
[INFO] [1758445864.128782480] [talker]: Publishing: Hello World: 17'
[INFO] [1758445864.128782480] [talker]: Publishing: Hello World: 18'
[INFO] [1758445865.132714980] [talker]: Publishing: Hello World: 19'
[INFO] [1758445867.12995533] [talker]: Publishing: Hello World: 20'
[INFO] [1758445867.12995533] [talker]: Publishing: Hello World: 21'
[INFO] [1758445867.128742102] [talker]: Publishing: Hello World: 22'
[INFO] [1758445870.132058620] [talker]: Publishing: Hello World: 23'
[INFO] [1758445870.132058620] [talker]: Publishing: Hello World: 24'
[INFO] [1758445872.128456193] [talker]: Publishing: Hello World: 25'
[INFO] [1758445872.128456193] [talker]: Publishing: Hello World: 26'
[INFO] [1758445872.128456193] [talker]: Publishing: Hello World: 27'
[INFO] [1758445875.129795856] [listener]: I Heard: [Hello World: 1]
[INFO] [1758445875.129795856] [listener]: I Heard: [Hello World: 2]
[INFO] [1758445877.130729816] [listener]: I Heard: [Hello World: 3]
[INFO] [1758445878.134745980] [listener]: I Heard: [Hello World: 31]
[INFO] [1758445880.131259159] [listener]: I Heard: [Hello World: 32]
[INFO] [1758445880.131259159] [listener]: I Heard: [Hello World: 33]
[INFO] [1758445881.130805928] [listener]: I Heard: [Hello World: 34]
[INFO] [1758445881.129791930] [listener]: I Heard: [Hello World: 35]
[INFO] [1758445881.129791930] [listener]: I Heard: [Hello World: 36]
[INFO] [1758445884.130332134] [listener]: I Heard: [Hello World: 37]
[INFO] [1758445884.130332134] [listener]: I Heard: [Hello World: 38]
[INFO] [1758445886.1357518654] [listener]: I Heard: [Hello World: 39]
[INFO] [1758445887.131992180] [listener]: I Heard: [Hello World: 40]
[INFO] [1758445887.131992180] [listener]: I Heard: [Hello World: 41]
```

Abbildung 1: Überprüfung der ROS 2 Installation durch Aufruf eines **talker**-Knotens (linkes Fenster) und eines **listener**-Knotens (rechtes Fenster) in separaten Terminals

## 1.2 Zusammenfassung Grundlagen ROS 2

Es wurde sich, wie in der Praktikumsanleitung gefordert mit den Grundlagen von ROS2 vertraut gemacht. Dafür wurden die entsprechenden Ressourcen der Praktikumsanleitung und der ROS2 Dokumentation bearbeitet. Für den ersten Einstieg mit ROS wurden zudem Videos auf Youtube verwendet. Um die ermittelten Kenntnisse zu festigen, sollten diese in diesem Kapitel zusammenfassend notiert werden, sodass dieses Kapitel auch für die spätere Verwendung als eine Art Spickzettel dienen kann.

### Nodes

- **Node** = ausführbare Einheit (Programm)
- Jeder Node hat einen eindeutigen Namen
- Starten:

```
ros2 run turtlesim turtlesim_node
```

### Topics

- 
- Datenkanäle für Kommunikation
  - Publisher ↔ Subscriber
  - Beispiele:

```
ros2 topic list
ros2 topic echo /cmd_vel
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist {linear: {x: 1.0}, angular: {z: 0.5}}
```

## Services

- Anfrage ↔ Antwort (synchron)
- Beispiel:

```
ros2 service list
ros2 service call /clear std_srvs/srv/Empty {}
```

## Actions

- Länger laufende Tasks mit Feedback
- Beispiel: Navigation

```
ros2 action list
```

## Parameter

- Konfigurationswerte für Nodes
- Beispiel:

```
ros2 param list
ros2 param set /turtlesim background_r 255
```

## Remapping

- Umbenennen von Topics/Nodes ohne Codeänderung

```
ros2 run my_pkg my_node --ros-args -r /cmd_vel:=/robot/cmd_vel
```

## Nachrichten

- Wichtige Typen:
  - `geometry_msgs/Twist` (Bewegung)
  - `std_msgs/String` (Text)
  - `sensor_msgs/Imu`, `Image`, `LaserScan`
- Anzeigen:

```
ros2 interface show geometry_msgs/msg/Twist
```

## Workspaces & Packages

- Workspace = Projektordner (`src/`, `build/`, `install/`)

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

- Package erstellen:

```
ros2 pkg create my_pkg --build-type ament_python --license Apache-2.0
```

## Sourcen

```
source /opt/ros/humble/setup.bash
source ~/ros2_ws/install/setup.bash
```

---

## 2 Erstellung eigenes ROS2 Paket

In diesem Abschnitt soll gemäß der Praktikumsanleitung ein eigenes ROS 2 Paket erstellt werden. Dafür wird anhand der bereitgestellten Links der ROS 2 Dokumentation vorgegangen. Nachdem eine Publisher-Subscriber Kommunikation erzeugt wurde, soll in einem Abschließenden Schritt ein Node erstellt werden, welcher die Turtle in der Turtlesim steuern kann.

### 2.1 Erstellung eines Arbeitsbereiches und ROS 2 Paket

Für die Erstellung eines neuen Arbeitsbereiches wurde zunächst ein neuer Ordner `ezmr_ws` erzeugt, welcher zudem den leeren Ordner `src` beinhaltet. Anschließend wurde in einem neuen Terminal in den Pfad „`ezmr_ws/src`“ navigiert und anschließend mit dem Befehl

Code 1: Clonen des ROS Tutorials in den Workspace Ordner

```
git clone https://github.com/ros/ros_tutorials.git -b humble
```

das ROS Beispiel Repository für die Humble Version eingefügt. Anschließend konnte mit dem Befehl

Code 2: Installation der ROS 2 Turtorial Abhängigkeiten in den Arbeitsbereich

```
rosdep install -i --from-path src --rosdistro humble -y
```

die notwendigen ROS Abhängigkeiten installiert werden. Im Anschluss wurde es somit möglich über „`colcon build`“ den Arbeitsbereich zu erstellen. Die erfolgreiche Installation wurde durch eine Benachrichtigung visualisiert und zudem waren nun im Verzeichnis „`ezmb_`“ zusätzlich zu „`src`“ die Ordner „`build`“, „`install`“ und „`log`“. Es konnte anschließend in einem neuen Terminal der Workspace gesourcet und die Turtlesim gestartet werden. Vorteile solcher Arbeitsbereiche sind, dass beispielsweise an der Erscheinung, also dem Overlay, von der Turtlesim Veränderungen vorgenommen werden können, ohne dass diese bei der originalen Installation angewandt werden. Dafür wurde das Overlay des Arbeitsbereiches angepasst. Es sollte der Fensternname, sowie die Hintergrundfarbe der Turtlesim verändert werden. Dafür wurde in das Verzeichnis „`ezmr_ws/src/ros_tutorials/turtlesim/src`“ navigiert und anschließend mit dem Befehl

Code 3: Öffnen Datei „`turtle_frame.cpp`“ mit dem Editor

```
nano turtle_frame.cpp
```

ein Editor geöffnet welcher die „`turtle_frame.cpp`“ öffnet wodurch Anpassungen an der grafischen Benutzeroberfläche von Turtlesim gemacht werden können. Für den Fensternamen wurde der Parameterwert der Funktion `setTitle()` auf „Turtlesim Phil“ geändert. Für die Hintergrundfarbe wurden die Standardwerte des Hintergrundes mit folgendem Code festgelegt

Code 4: Definition der Hintergrundfarben der Turtlesim mit Hex-Werten

```
#define DEFAULT_BG_R 0x45  
#define DEFAULT_BG_G 0x56  
#define DEFAULT_BG_B 0xff
```

Anschließend wurde die Datei gespeichert und der Workspace mit dem Befehl, s. Code 5, neu gebaut, wobei nur die Turtlesim aktualisiert wurde, wodurch die Buildzeit verringert wurde.

Code 5: Builden des Workspace bei exklusiver Aktualisierung der Turtlesim

```
colcon build --packages-select turtlesim
```

Dann wurden durch zwei verschiedene Terminals die Turtlesim der Standardinstallation ausgeführt und die Turtlesim des Workspaces mit dem angepassten Overlay, s. Abb. 2. Dabei sind die Veränderung klar zu erkennen.

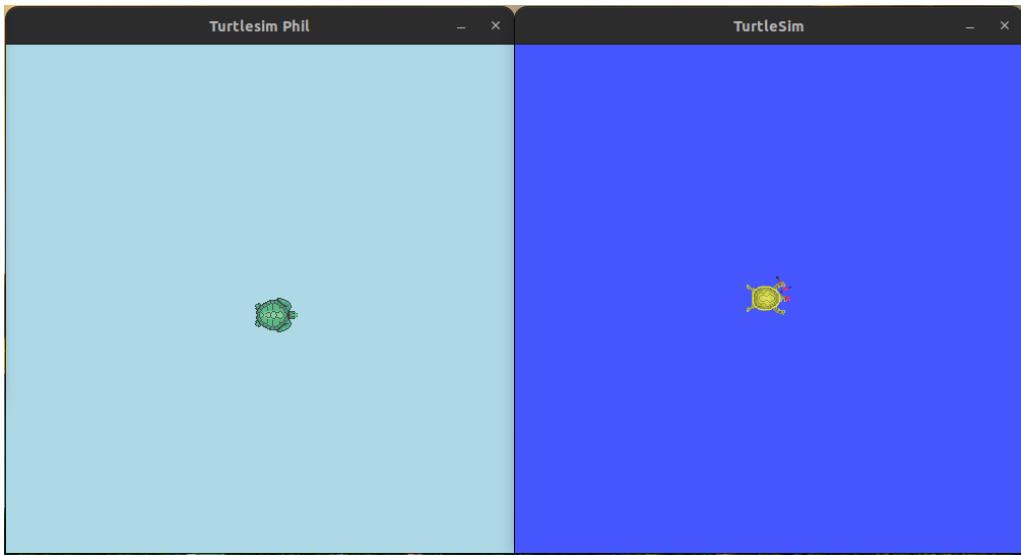


Abbildung 2: Vergleich der Turtlesim mit angepasstem Overlay im Workspace (linkes Fenster) und normal installierter Turtlesim (rechtes Fenster)

Nachdem nun der Arbeitsbereich erfolgreich erstellt wurde, kann nun die Erstellung eines eigenen ROS 2 Pakets beginnen. Dafür wurde entsprechend der Praktikumsanleitung und der darin referenzierten ROS 2 Dokumentation vorgegangen. Nachdem in einem neuen Terminal ROS 2 wieder gesourcet wurde, konnte mit dem folgendem Befehl ein neues ROS 2 Paket namens „phil\_pkg“ erstellt werden, s. Code 6.

Code 6: Erstellung eines neuen ROS 2 Pakets

```
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name
    ↪ HelloPhil phil_pkg
```

Dabei wurde auch wie im Tutorial ein simpler Hello World Node automatisch erzeugt, durch die Verwendung des optionalen Parameters –„node\_name“. Anschließend konnte, nachdem in das Wurzelverzeichnis des Arbeitsbereichs gewechselt wurde, der Workspace erneut gebuildt werden, zur Zeitoptimierung wurde dabei wieder der optionale Parameter „–packages-select“ verwendet, sodass das builden weniger Zeit benötigt. In einem neuen Terminal konnte dann, nachdem der Workspace gesourcet wurde, konnte der automatisch erzeugte ROS 2 knoten „HelloPhil“ mit dem Folgenden Befehl ausgeführt werden, s. Code 7.

Code 7: Aufruf des HelloPhil Knotens zum Test des erzeugten ROS 2 Pakets

```
ros2 run phil_pkg HelloPhil
```

Da der Aufruf „Hi from phil\_pkg.“ ausgab, kann von einer erfolgreichen Erstellung ausgegangen werden. Im Anschluss wurden noch gemäß der ROS 2 Dokumentation in den Dateien „package.xml“ und „setup.py“ die entsprechenden Einträge für Beschreibung sowie Maintainer und Maintainer-Mail angepasst.

## 2.2 Erstellung einer Publisher-Subscriber Kommunikation mit Python

In einem nächsten Schritt sollte gemäß der ROS 2 Dokumentation eine Publisher Subscriber Kommunikation erstellt werden. Es wurde sich dabei für die Umsetzung mit Python entschieden. Es wurde innerhalb des „ezmr\_ws“ ein neues Paket erstellt, welches den Namen „py\_pubsub\_phil“ trug. Es konnte dann ein Publisher erzeugt werden, indem folgender Befehl im Verzeichnis „/ezmr\_ws/src/py\_pubsub\_phil/py\_pubsub\_phil“ ausgeführt wurde, s. Code 8.

Code 8: Erzeugung eines Publisher Knoten, durch Kopieren des ROS 2 Beispiels

```
wget https://raw.githubusercontent.com/ros2/examples/humble/rclpy/topics/
    ↪ minimal_publisher/examples_rclpy_minimal_publisher/
    ↪ publisher_member_function.py
```

Nach der erfolgreichen Erstellung konnte in der „publisher\_member\_function.py“ noch individuelle Anpassungen vorgenommen werden, sodass die spätere Ausgabe leicht verändert wurde. Im Anschluss wurde es noch notwendig innerhalb des erstellten Pakets Abhängigkeiten hinzuzufügen, was bedeutet, dass definiert wurde, dass das Paket für die Ausführung des Knotens die Abhängigkeiten `rclpy` und `std_msgs` benötigt. Anschließend musste noch in der „`setup.py`“ der Entry Point festgelegt werden für den Publisher, wobei auch definiert wurde, wie der Knoten aufgerufen wird. Danach konnte der Subscriber Knoten erzeugt werden. Dafür wurde wieder das entsprechende Beispielprojekt kopiert, indem der Befehl, s. Code 9, ausgeführt wurde.

Code 9: Erzeugung eines Subscriber Knoten, durch Kopieren des ROS 2 Beispiels

```
wget https://raw.githubusercontent.com/ros2/examples/humble/rclpy/topics/
    ↪ minimal_subscriber/examples_rclpy_minimal_subscriber/
    ↪ subscriber_member_function.py
```

Dieser Subscriber musste dann auch noch in der „`setup.py`“ in die Entry Points hinzugefügt werden. Anschließend konnte der Arbeitsbereich erneut gebuildt werden. Nachdem die Steup Datein gesourcet wurden in einem neuen Terminal konnte die Publisher Subscriber Kommunikation erfolgreich getestet werden, s. Abb. 3.

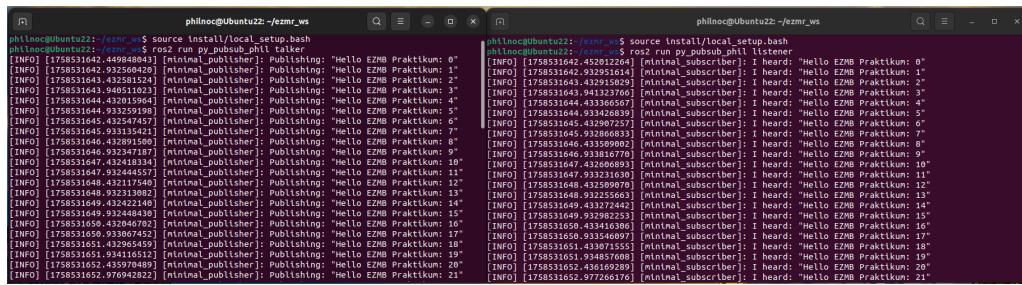


Abbildung 3: Testung der erstellten Publisher Subscriber Kommunikation

Wie in Abb. 3 zu sehen ist, Publiziert der Publisher erfolgreich Daten auf ein Topic und der Subscriber liest die Daten aus diesem Topic aus.

### 2.3 Erstellung eines eigenen Nodes zur Bewegungssteuerung

Es sollte nun ein eigener Node erstellt werden, welcher auf das Topic `turtle1/cmd_vel` publizieren soll und somit die Schildkröte in der Turtlesim steuern soll. Zunächst wurde dafür im „src“ Verzeichnis des Arbeitsbereich „ezmr\_ws“ ein neues Paket namens „`drive_rob`“ angelegt, s. Code 10.

Code 10: Erzeugung eines Pakets zur Bewegungssteuerung

```
ros2 pkg create --build-type ament_python --license Apache-2.0 drive_rob
```

Anschließend konnte im Verzeichnis „`drive_rob/drive_rob`“ eine neue Pythondatei namens „`robot_square`“ erzeugt werden, welche den Code für den zu erstellen Node beinhalten sollte. Die Datei konnte anschließend mit dem Texteditor geschrieben werden. Zu Beginn wurden die notwendigen Bibliotheken importiert, welche die ROS Instanzen und Kommunikation erlaubten (`rclpy`), die Erstellung des Nodes (Node) und das Verwenden von Twist Nachrichten zur Bewegungssteuerung (Twist), s. Code 11.

---

Code 11: Importierung der notwendigen Python Bibliotheken für die Nodeerstellung

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
```

Im Anschluss konnte eine Node Instanz deklariert werden, welche „TurtleSquare“ benannt wurde. Innerhalb der Initialisierungsmethode wurde dabei definiert, dass ein Publisher erzeugt wird, welcher auf das Topic „/turtle1/cmd\_vel“ publishen soll, die Warteschlange soll dabei eine Länge von 10 besitzen, was bedeutet, dass 10 Nachrichten zwischengespeichert werden, falls schneller publiziert wird, als der Subscriber lesen kann. Zusätzlich wurde ein Timer deklariert, welcher für die spätere Bewegungsrealisierung notwendig ist. Zudem wurden Statusvariablen erzeugt, sowie Zeitwerte für die Bewegungssteuerung, s. Code 12.

Code 12: Instanzierung des Nodes zur Bewegungssteuerung

```
class TurtleSquare(Node):
    def __init__(self):
        super().__init__('square')
        self.publisher = self.create_publisher(Twist, '/turtle1/cmd_vel', 10)
        self.timer_period = 0.1 # 10 Hz
        self.timer = self.create_timer(self.timer_period, self.move)
        self.state = 'straight1'
        self.step_count = 0
        self.turn_duration = int(1.8 / self.timer_period)
        self.straight_duration = int(2.5 / self.timer_period)
```

Darauf folgend wurde eine Methode definiert, welche die konkreten Bewegungswerte und somit die Bewegungsabfolge definiert. Ziel war es den Roboter in einem Viereck fahren zu lassen. Dafür sollte eine Bewegung zum Vorwärtsfahren und eine Drehbewegung um 90 ° implementiert werden, welche jeweils vier mal durchgeführt werden sollen. Dies sollte anschließend eine Bewegung entlang eines Quadrates ergeben. Es wurde für die Zustände `straight`, `turn` und `finish` implementiert. Bei den Bewegungszuständen wurden solange Nachrichten auf das Topic „/turtle1/cmd\_vel“ veröffentlicht, bis entsprechende Timer abgelaufen waren. Es wurde dann noch eine Zählvariable eingefügt, welche sicherstellen sollte, dass die beiden Bewegungsarten jeweils vier Mal ausgeführt werden und anschließend der Zustand `finish` eingenommen wird, s. Code 13.

Code 13: Methode zum Bewegungsablauf der quadratischen Bewegung

```
def move(self):
    msg = Twist()
    if self.state == 'straight':
        if self.loopcount==4:
            self.state='finish'
        elif self.step_count < self.straight_duration:
            msg.linear.x = 1.0
            msg.angular.z = 0.0
            self.step_count += 1
        else:
            self.state = 'turn'
            self.step_count = 0
    elif self.state == 'turn':
        if self.step_count < self.turn_duration:
            msg.linear.x = 0.0
            msg.angular.z = 1.0
            self.step_count += 1
        else:
            self.state = 'straight'
            self.step_count=0
            self.loopcount+=1
    else:
        self.step_count = 0
        msg.linear.x = 0.0
```

---

```

        msg.angular.z = 0.0
        self.publisher.publish(msg)
        self.destroy_timer(self.timer)
        self.destroy_node()
        return
    self.publisher.publish(msg)

```

Abschließend sollte noch die Hauptfunktion implementiert werden. Diese initialisiert das ROS2-System und erstellt die `TurtleSquare` Node-Instanz. Diese wird an den ROS Executer übergeben. Solange der Node aktiv ist, verarbeitet der Executor Callbacks, beispielsweise den Timer für die Bewegungssteuerung. Sobald der Node sich selbst beendet, kehrt `spin` zurück. Danach wird mit `rclpy.shutdown()` das ROS-2-System sauber heruntergefahren, s. Code 14.

Code 14: Hauptfunktion des Bewegungsablaufs der quadratischen Bewegung

```

def main(args=None):
    rclpy.init(args=args)
    node = TurtleSquare()
    rclpy.spin(node)
    rclpy.shutdown()

```

Nach Fertigstellung konnte dann innerhalb der „`setup.py`“ der Node als Entrypoint hinzugefügt werden. Im Anschluss wurde das Paket mit dem Befehl „`colcon build --packages-select drive_rob`“ gebaut und konnte anschließend getestet werden. In Abb. 6 können die Ergebnisse der Turtlesim eingesehen werden. Abb. 4 zeigt die einmalige Ausführung des Nodes, dabei lässt sich ein gefahrenes Viereck gut erkennen. Es fällt jedoch auf, dass es kein sauberes Quadrat ist, welches aufgezeichnet wurde. Deutlich kann man es in Abb. 5 erkennen, dabei sieht man, dass bei einer zweiten Ausführung des Nodes zwei Vierecke entstehen, welche jedoch in der Positionierung verschieden sind. Es kann darauf zurückgeführt werden, dass der Winkel nicht korrekt bei  $90^\circ$  eingestellt wurde. Es ist jedoch zu sagen, dass bei mehrfacher Ausführung festgestellt wird, dass die erschienenen Vierecke stets etwas unterschiedlich sind. Vermutlich hängt dies mit dem Timer des Publizierens zusammen. Falls ein korrektes Quadrat stattdessen gefahren werden soll, wäre die Wahl mit Koordinaten vermutlich eine besser Methode anstatt der Relativbewegungen.



Abbildung 4: Ergebnis des einmaligen Ausführrens des Square-Nodes



Abbildung 5: Ergebnis des zweifachen Ausführrens des Square-Nodes

Abbildung 6: Vergleich der Ausführung des Square Nodes in der Turtlesim

Zusätzlich wurde noch ein weiterer Node erstellt. Dabei wurde versucht eine herzförmige Bewegung auszuführen. Dies sollte durch die Verkettung zweier geradliniger Bewegungen sowie zwei Kreis-

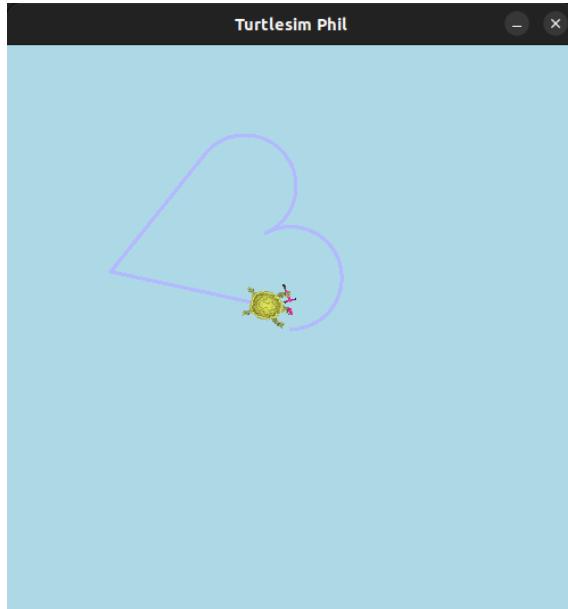


Abbildung 7: Ergebnis des Heart-Nodes in der Turtlesim

bewegungen und Drehungen dazwischen durchgeführt werden. Die Erstellung des Nodes erfolgte dabei analog zu der Erstellung des Nodes für die Quadratbewegung. Im Unterschied wurden weitere Zustände implementiert, da wie beschrieben mehr notwendig waren. Diese wurden anschließend durch eine `if-elif-else`-Verzweigung verknüpft. Der konkrete Code kann aus dem beigefügten Git-Repository entnommen werden. Zur Reduzierung des Umfangs, wird dieser in dem Beleg nicht abgebildet. Nach der Erstellung und der Hinzufügung des Entrypoints konnte das Paket erneut gebaut werden und anschließend der entwickelte Node getestet werden. Die Tests ergaben auch hier, dass bei jeder Zeichnung geringfügige Unterschiede sich ergaben. Dennoch kann die grundlegende Herzform stets erkannt werden, s. Abb. 7. Es kann abschließend gesagt werden, dass dieser Abschnitt erfolgreich dazu beigetragen haben, dass der Student die Begriffe Workspace und Paket innerhalb von ROS verstanden hat. Zudem konnte der Student erfolgreich eigene Pakete erstellen, sowie eigene Nodes schreiben, deren Auswirkung erfolgreich in der Turtlesim betrachtet werden konnte.

---

### 3 Programmierung eigener Roboter

Nachdem die Installation von Micro-ROS, sowie von VS-Code und PLattform-IO durchgeführt wurde, konnte das Beispielprojekt der Praktikumsanleitung auf den Roboter geflasht werden. Dabei konnte in einem mit ROS2 das Topic „/micro\_ros\_plattformio\_node\_publisher“ über den echo Befehl abgehört werden und die Daten empfangen werden. Dies sollte eine funktionelle Installation anzeigen, sodass in den nächsten Schritten die Programmierung des Roboters angegangen werden konnte.

Um die Ansteuerung der Motoren besser kennenzulernen und zu verstehen, wurde dafür zunächst ein PlattformIO Projekt erzeugt, welches keine ROS2 Implementierung beinhalten sollte, sondern lediglich den Roboter etwas zu bewegen. Für die Steuerung der Motoren wurde UART2 initialisiert, da über diese laut Praktikumsanleitung Befehle an die Motoren gesendet werden können. Für die UART2 Verbindung wurde es notwendig den GPIO 17 als Transmitter Pin zu verwenden. Im Programm wurde noch ein Rx Pin definiert, welcher jedoch den Wert -1 besaß, dies bedeutet, dass der Pin nicht verbunden wird. Da die Motoren keine Rückmeldung über die UART2 geben ist dies ein sinnvoller Schritt, um trotzdem einen vollständigen Funktionsaufruf auszuführen. Nach einigen Tests, um die Befehlssyntax über die UART für die Motoren kennenzulernen, wurde eine Funktion geschrieben, mit welcher solch ein Motorenbefehl gesendet werden kann. Bei der Funktion `sendeMotorBefehl` wird ein Telegramm zur Steuerung erzeugt, welches das Nachrichtenformat der Praktikumsanleitung als Byte-Array kodiert und anschließend an den Motorenkontroller sendet, sodass bei einem Funktionsaufruf jeder Parameter festgelegt werden kann und das Start Byte automatisch hinzugefügt wird. Innerhalb dieses Testprogramms konnte so der Roboter zunächst eine Drehbewegung nach rechts 5 s durchführen, dann eine Linksdrehbewegung für 5 s und abschließend eine geradlinige Bewegung nach vorn für 5 s. Es konnte mit diesem Programm auch herausgefunden werden wie die Motoren bei verschiedenen Parametern reagieren. Die ID kann die Werte 1 und 2 annehmen, dabei ist 1 der linke Motor, betrachtet als Fahrer, und ID 2 ist der rechte Motor. Wie in der Praktikumsanleitung beschrieben kann der Betriebsmodus zwischen Schrittmodus und Dauerbetrieb wechseln. Die Drehrichtung ist bei beiden Motoren ähnlich, bei 1 dreht sich der Motor im Uhrzeigersinn, bei 0 gegen die Uhr. Zu beachten ist dabei, dass die beiden Motoren spiegelsymmetrisch angeordnet sind. Das bedeutet, dass für eine geradlinige Bewegung die Drehrichtung der Motoren entgegen gerichtet sein muss. Der modusabhängige Parameter bestimmt beim Schrittmodus die Anzahl der Schritte. Beim Dauerbetrieb bestimmt er die Geschwindigkeit. Bei der Geschwindigkeit ist zu beachten, dass ein geringerer Wert zu einer höheren Geschwindigkeit führt.

Nachdem die Motorensteuerung nun untersucht und getestet wurde, konnte die Implementierung der Motorensteuerung mit ROS 2 angegangen werden. Laut Aufgabenstellung sollte der Roboter auf das Topic `cmd_vel` hören.

Zu Beginn wurde eine Datei „motor.cpp“, und eine dazugehörige Header-Datei angelegt. In die .cpp Datei wurde dann die zuvor geschriebene `sendeMotorBefehl`-Funktion geschrieben. Um die Funktionalität noch zu verbessern, wurde eine `stoppeMotoren`-Funktion geschrieben, welche bei Aufruf an beide Motoren den Befehl zu einer Geschwindigkeit von 0 sendet. Außerdem wurde eine Funktion `motorControl` geschrieben, welche einen Linear-Wert und einen Angular-Wert annimmt und daraus entsprechende Befehle an den Motor sendet. Diese Funktion soll somit als Callback für das `cmd_vel`-Topic verwendet werden. Innerhalb der Funktion wird zunächst ein Differential berechnet, es werden also die Vorwärtsbewegung und die Drehbewegung kombiniert und entsprechend eine passende Geschwindigkeit berechnet, s. Code 15.

Code 15: Antriebsdifferential der Motoren

```
// Differenzial-Antrieb: linke und rechte Geschwindigkeit
float left = linear - angular;
float right = linear + angular;
```

Je nach Geschwindigkeitswert, wird zudem in einer `if`-Verzweigung definiert, welche Drehrichtung die Motoren besitzen sollen, s. Code 16.

Code 16: Festlegen der Drehrichtungen der Motoren, je nach Geschwindigkeitswerten

---

```
// Richtung setzen
uint8_t leftDir = (left >= 0) ? 0x01 : 0x00;
uint8_t rightDir = (right >= 0) ? 0x00 : 0x01;
```

Danach soll der Geschwindigkeitswert der ROS2 Benachrichtigung in ein entsprechendes Format übersetzt werden. Dafür wurde eine Mapping Funktion geschrieben. Bei den Tests mit den Motoren wurde schließlich festgestellt, dass geringere Werte bei der Geschwindigkeit zu einer schnelleren Bewegung führen und dass wenn der Wert kleiner als 2 sinkt, Probleme auftreten. Nach dem Mapping werden zum Abschluss noch die Motorenbefehle über die UART2 gesendet. Mit dieser Funktion sollte es also möglich werden, durch Eingabe von Werten für `linear.x` und `angular.z` eine Motorensteuerung zu gewährleisten.

Im Anschluss wurde die „motor.h“ geschrieben, bei welcher die Funktionsprototypen der grundlegenden Funktionen für die Motorensteuerung definiert wurden, sodass diese in der „main.cpp“ verwendet werden können. Als Grundlage für die Main Datei wurde dabei das Micro-ROS Testbeispiel der Praktikumsanleitung genommen und entsprechend erweitert. Zu Beginn der Datei wurde die Header Datei der Motoren eingebunden. Anschließend wurde auch die Twist Benachrichtigung eingebunden, da diese das Format für das Topic `cmd_vel` definiert. Außerdem wurden noch die entsprechenden Pins und die Baudrate für die Motorensteuerung definiert, s. Code 17.

Code 17: Definition und Einbindung von Komponenten für die Motorenintegration

```
...
#include "motor.h"

#include <rcl/rcl.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
#include <geometry_msgs/msg/twist.h>
#include <std_msgs/msg/string.h>

#define UART_TX_PIN 17
#define UART_RX_PIN -1
#define UART_BAUDRATE 9600
```

Anschließend wurden die ROS Objekte definiert und es wurden Variablen für eine Timeout Logik deklariert. Es wurde sich entschieden solch eine Timeout Logik zu integrieren, sodass eine Bewegung des Roboters aktiv gestartet werden muss und nach Ablaufend er Zeit die Motoren automatisch gestoppt werden. Dafür wurde ein Timer mit einer Dauer von 1 s implementiert. Anschließend wurde der Subscriber implementiert, welcher auf das Topic `cmd_vel` hören soll, s. Code 18.

Code 18: Definition des Subscribers des `cmd_vel` Topics

```
// Subscriber Callback
void cmd_vel_callback(const void *msgin) {
    const geometry_msgs__msg__Twist *msg = (const geometry_msgs__msg__Twist *)msgin;
    float linear = msg->linear.x;
    float angular = msg->angular.z;
    last_cmd_time = millis(); // Zeitstempel aktualisieren
    motorControl(linear, angular);
}
```

Im Setup Block wird dann nach Initialisierung der Micor-Ros Kommunikation die Motoren Kommunikation initialisiert. Der Roboter hat das Node „phil\_robot“. Im Anschluss wird festgelegt, dass der Roboter auf das Topic `cmd_vel` hört und dass bei Werten auf dem Topic anschließend der entsprechende Callback mit der `motorControl`-Funktion aufgerufen wird, s. Code 19.

Code 19: Setup-Block des ESP für die Robotersteuerung

```
void setup() {
    Serial.begin(115200); //Serielle Verbindung zum PC
    set_microros_serial_transports(Serial); //Ros Kommunikation ueber seriell
    delay(2000);
```

---

```

allocator = rcl_get_default_allocator(); //Ros Client Library Initialisierung
RCCHECK(rclc_support_init(&support, 0, NULL, &allocator)); //Initialisierung Ros Support Struktur
RCCHECK(rclc_node_init_default(&node, phil_robot, , &support));//Initialisierung Ros2 Node

initMotorSerial(UART_RX_PIN, UART_TX_PIN, UART_BAUDRATE);

RCCHECK(rclc_subscription_init_default(
    &subscriber_cmd_vel,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(geometry_msgs, msg, Twist),
    cmd_vel)); //ESP hoert auf das Topic cmd_vel

RCCHECK(rclc_executor_init(&executor, &support.context, 5, &allocator));
// subscriber_cmd_vel (fuer cmd_vel)
RCCHECK(rclc_executor_add_subscription(
    &executor,
    &subscriber_cmd_vel,
    &twist_msg,
    &cmd_vel_callback,
    ON_NEW_DATA));
}

```

Im abschließenden Loop-Block wird der Event-Loop von Micro-ROS aufgerufen. Das bedeutet, dass überprüft wird, ob neue Nachrichten auf den entsprechenden Topics publiziert wurden, falls ja werden die entsprechenden Callbacks ausgeführt. Zudem werden die Funktionen bis zu 100 ms ausgeführt, falls notwendig. Im Anschluss wird der Timer überprüft, bei einem Callback aufruf wird ein Zeitstempel gesetzt und falls dieser länger als 1000 ms her ist, dann wird automatisch die Funktion `stoppeMotoren` ausgeführt, s. Code 20.

Code 20: Loop-Block des ESP für die Robotersteuerung

```

void loop() {
    RCSOFTCHECK(rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100)));
    unsigned long now = millis();
    // Timeout-Ueberpruefung fuer Stop
    if (now - last_cmd_time > CMD_TIMEOUT_MS) {
        stoppeMotoren();
    }
    delay(10);
}

```

Nach der Fertigstellung des Programms sollte dieses nun auch getestet werden. Dafür wurde das Programm innerhalb von VS-Code gebaut und anschließend auf den ESP-32 geflasht. Währenddessen wurde ein neues Terminal geöffnet, ROS2 gesourcet und der Micro-ROS Agent gestartet. Anschließend wurde in einem neuen Terminal der zuvor erstellte Workspace gesourcet und anschließend der Turtlesim Node „turtle\_teleop\_key“ ausgeführt, dabei wurde der folgende Befehl verwendet, mit welchem ein Remapping des Topics von `trutle/cmd_vel` auf `cmd_vel` ausgeführt wird, s. Code 21.

Code 21: Aufruf des Befehls zur Steuerung des Roboters mit den Pfeiltasten durch Topic-Remapping

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

Damit ließ sich der eigens programmierte Roboter gut steuern. Dabei jedoch nur mit einer einheitlichen Geschwindigkeit und stets nur entweder linear oder mit einer Drehbewegung. Um die Bewegungen und Geschwindigkeiten besser zu untersuchen, wurde der `teleop_twist_keyboard` Node verwendet. Mit diesem besteht die Möglichkeit die Geschwindigkeit sowohl linear als auch angular zu verändern. Der Node wurde durch den folgenden Befehl aufgerufen, s. Code 22.

Code 22: Robotersteuerung durch Verwendung des `teleop_twist_keyboard` Nodes

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

---

Durch die Verwendung dieses Nodes konnte festgestellt werden, dass der Roboter auf verschiedene eingestellte Geschwindigkeiten verschieden reagiert und ab einem Geschwindigkeitswert von  $x=2$  abgeriegelt ist. höhere Werte werden entgegengenommen, aber die Geschwindigkeit wird nicht höher. Zudem konnte auch festgestellt werden, dass wenn eine Nachricht sowohl linear, als auch angular Anteil besitzen, dann der Roboter auch eine entsprechende Bewegung durchführt. Die Implementierung der grundlegenden Fahrfunktionalität kann somit als gewährleistet betrachtet werden.

Abschließend sollten noch die zuvor erstellten Nodes für die Turtlesim ausgeführt werden und das Verhalten des Roboters beobachtet werden. Es wurde dafür ein neues Terminal geöffnet im Workspace „ezmr\_ws“ und es wurde das Arbeitsverzeichnis gesourcet. Anschließend wurde der „turtle\_square“ Node des „drive\_rob“ Paketes ausgeführt. Dabei wurde auch ein Remapping durchgeführt, sodass der Node auf das Topic „cmd\_vel“ schreibt, s. Code 23.

Code 23: Robotersteuerung durch Remapping und verwendung des turtle\_square Nodes

```
ros2 run drive_rob turtle_square --ros-args -r /turtle1/cmd_vel:=/cmd_vel
```

Die Ausführung des Befehls führte dazu, dass der Roboter sich bewegte. Dabei ist jedoch festzuhalten, dass der Roboter nicht wie in der Turtlesim ein Rechteck fuhr, sondern ein Vieleck begann. Die Geraden wurden zwar korrekt gefahren, jedoch hat sich der Roboter bei den Drehbewegungen nicht wie die Turtlesim um  $90^\circ$  gedreht, sondern eher um  $30^\circ$ . Dies wird darin vermutet, dass die Drehbewegungen des Roboters nicht genau auf die gleiche Empfindlichkeit programmiert sind wie die der Simulation. Es wäre jedoch möglich durch Anpassung der Parameter für die Geschwindigkeit und Anpassung der Motorenbefehle je nach ROS 2 Nachricht, eine gleiche Bewegung in der Simulation und mit dem realen Roboter durchzuführen. Dies ist jedoch nicht Aufgabe des Praktikums weshalb es nicht weiter gemacht wird. Es wurde zudem auch noch der „turtle\_heart“ Node ausgeführt. Auch bei diesem wurde die Zielform nicht nachgefahren. Dennoch kann festgehalten werden, dass der Roboter erfolgreich durch das cmd\_vel Topic zu steuern ist.

---

## 4 Erweiterung durch Sensorik und eigene Projekte

Nachdem nun die Grundfunktionalität der Bewegung implementiert wurde, soll der Roboter und das ROS2 Projekt um weitere Sensoren erweitert werden. Die Praktikumsanleitung hat dazu eine Auswahl an den verfügbaren Sensoren, sowie deren Pinbelegung bereitgestellt, wie beispielsweise einen Ultraschallsensor, eine IMU, ein Display, einen Buzzer, sowie zwei Bumper.

### 4.1 Distanzmessung und Displayausgabe

In einem ersten Projekt soll nun der Ultraschallsensor integriert werden. Dabei soll kontinuierlich der Abstand zwischen Sensor und Objekten gemessen werden und über das ROS Netzwerk ausgegeben werden. Der Ultraschallsensor ist ein HC-SR04, welcher über die Pins 27 für den Trigger und 25 für das Echo angeschlossen ist. Der Sensor sendet ein Ultraschallsignal mit 40 kHz und kann somit Entferungen zwischen 0,02 und 4 m erfassen. Die Messung basiert auf einem Echo, weshalb sich die Berechnung der Entfernung zu

$$\text{Entfernung [cm]} = \frac{\text{Zeit [\mu s]} \cdot 0.0343}{2}$$

ergibt. Der Divisor 2 entsteht, da der Schall den Weg zweimal durchführen muss. Der Faktor 0,0343 entspricht der Schallgeschwindigkeit umgerechnet in  $\frac{\text{cm}}{\mu\text{s}}$ .

Wie bei der Integration des Motors wurden zunächst Funktionen in einer Testumgebung geschrieben, bei welcher keine ROS2 Komponenten integriert wurde. Es wurden dabei die Funktionen `initUltrasonic`, zur Initialisierung des Sensors, und `readDistance`, zur Messdatenaufnahme, entwickelt. Bei der Initialisierungsfunktion wird zudem für die ROS2 Nachricht einige Konstanten deklariert, wie beispielsweise der Messbereich, s. Code 24.

Code 24: Initialisierungsfunktion für das Ultraschallmodul

```
void initUltrasonic(sensor_msgs__msg__Range &range_msg) {
    pinMode(TRIGGER_PIN, OUTPUT);
    pinMode(ECHO_PIN, INPUT);

    range_msg.radiation_type = sensor_msgs__msg__Range__ULTRASOUND;
    range_msg.field_of_view = 0.5;
    range_msg.min_range = 0.02;
    range_msg.max_range = 4.0;
    range_msg.header.frame_id.data = (char *)"ultrasonic_sensor";
    range_msg.header.frame_id.size = strlen("ultrasonic_sensor");
    range_msg.header.frame_id.capacity = range_msg.header.frame_id.size + 1;
}
```

Innerhalb der Funktion für die Messdatenaufnahme wurde der Triggerpin für 10 Mikrosekunden aktiviert und anschließend auf das Echo gewartet. Für die Berechnung der Distanz wird dabei eine vereinfachte Formel genutzt, in welcher der Wert für die Schallgeschwindigkeit nicht als Faktor, sondern als Divisor mit dem Wert 29 genutzt wird, was gerundet dem Kehrwert von 0,0343 entspricht. Die Funktion gibt dann die berechnete Distanz in cm zurück, s. Code 25.

Code 25: Funktion für die Distanzmessung per Ultraschall

```
long readDistanceCm() {
    digitalWrite(TRIGGER_PIN, LOW);
    delayMicroseconds(2);
    digitalWrite(TRIGGER_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(TRIGGER_PIN, LOW);

    long duration = pulseIn(ECHO_PIN, HIGH, 30000); // Timeout 30ms

    if (duration == 0) return -1;

    return duration / 29 / 2;
}
```

---

In der „main.cpp“ des ROS Projektes wurde anschließend zunächst der entwickelte Header-File des Ultraschallmoduls eingebunden. Im Anschluss wurden das ROS2 Objekt des Range Nachrichtenobjektes der Sensoren eingebunden. Danach wurden solch ein Range Nachrichtenobjekt deklariert und auch Publisher, da die Sensorwerte in das ROS2 Netzwerk veröffentlicht werden sollen, s. Code 26.

Code 26: Funktion für die Distanzmessung per Ultraschall

```
#include <sensor_msgs/msg/range.h>
...
sensor_msgs__msg__Range range_msg;
rcl_publisher_t range_publisher;
```

In der Setup Schleife der Datei wurde anschließend die Initialisierungsfunktion der Ultraschallmoduls ausgeführt und der Publisher für die Ultraschallwerte initialisiert, s. Code 27.

Code 27: Initialisierung des Ultraschallmoduls und des Ultraschall Publishers

```
void setup(){
    ...
    initUltrasonic(range_msg);
    ...
    RCCHECK(rclc_publisher_init_default(
        &range_publisher,
        &node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(sensor_msgs, msg, Range),
        "ultrasonic_range"));
}
```

Nach dem Setup wurde in der Loop-Funktion noch die Messwertaufnahme durch aufruf der entsprechenden Funktion ausgeführt. Die Messwertaufnahme soll dabei in einem Intervall von 200 ms durchgeführt werden. Der aufgenommene Distanzwert wird anschließend in die Einheit Meter umgerechnet und in das ROS2 Netzwerk publiziert, s.Code 28.

Code 28: Messwertaufnahme und Publizierung in das ROS Netzwerk

```
void loop(){
    ...

    static unsigned long last_pub_time = 0;
    unsigned long now = millis();
    if (now - last_pub_time >= 200) { // alle 200 ms publizieren
        last_pub_time = now;
        long dist = readDistanceCm();
        range_msg.header.stamp.sec = now / 1000;
        range_msg.header.stamp.nanosec = (now % 1000) * 1000000;
        range_msg.range = (dist > 0) ? dist / 100.0 : 0.0;
    }
    rcl_publish(&range_publisher, &range_msg, NULL);
    ...
}
```

Anschließend wurde getestet, ob die Werte korrekt aufgenommen und ins ROS Netzwerk publiziert werden. Dafür wurde das Projekt auf den EPS32 geflasht und der MicroRos Agent gestartet. Im Anschluss wurde in einem neuen Terminal der Arbeitsbereich gesourct. Mit dem Befehl, **ros2 Topic list**, konnte dann überprüft werden, ob das Topic für die Ultraschallwerte veröffentlicht ist, was der Fall war. Im Anschluss wurde der Echo Befehl genutzt um die veröffentlichten Werte auf dem Topic **ultrasonic\_range** abzufragen. Die führte zu einer erfolgreichen Ausgabe der Abstandswerte. Durch fahren des Roboters auf ein Ziel zu oder von einem Ziel weg, konnte damit auch bestätigt werden, dass sich die Werte in einem sinnvollen Maße änderten. Die Integration des Ultraschallmoduls kann somit als erfolgreich betrachtet werden. Die Integration konnte dabei ohne Komplikationen durchgeführt werden.

Das nächstes Ziel sollte die Integration des Displays sein. Das höhere Ziel sollte es sein einen Python Node zu schreiben, bei welchem die empfangenen Distanzwerte auf dem Display angezeigt werden und der Roboter bei zu nah entfernten Objekten eine simple Ausweichbewegung ausführen soll.

Der erste Umgang mit dem Display wurde wieder in einer Datei gemacht, ohne ROS Integration, so konnte die prinzipielle Einbindung des Displays zunächst fokussiert werden. Am Ende der Testphase konnten wieder eine Initialisierungsfunktion, `initDisplay` und eine Anzeigefunktion, `showMessage` erstellt werden. Innerhalb der Initialisierungsfunktion wird die I2C Schnittstelle zum Display initialisiert und zunächst ein schwarzes Display mit dem Text „Display bereit!“ angezeigt. Zudem wird innerhalb dieser Funktion auch Speicherplatz für die String Benachrichtigung reserviert, die Stringlänge soll dabei auf 100 Zeichen beschränkt werden. Das Ziel sollte es sein den Text auf dem Display durch Senden eines Strings zu verändern. Es wurde bei dieser Implementierung darauf verzichtet, weitere Einstellungen und Parameter des Displays zu integrieren. Es wäre dennoch sinnvoll gewesen, möglicherweise Bilder auf dem Display durch senden von Bitmaps zu ermöglichen. Nach Einbindung der Header Datei in das Main Projekt musste anschließend die Standardnachricht String noch als Objekt eingebunden werden und ein Objekt deklariert werden. Zudem wurde auch ein Subscriber für das Display deklariert, s. Code 29.

Code 29: Vorbereitung zur Integration des Display als ROS Komponente

```
#include "display.h"
...
#include <std_msgs/msg/string.h>
...
std_msgs__msg__String string_msg_display;
rcl_subscription_t subscriber_display;
```

Danach wurde eine Subscriber Callback Funktion für das Display analog zum Motor geschrieben. Dabei wird die Funktion `showMessage` aufgerufen und die Daten der String Benachrichtigung werden dabei übergeben, s. Code 30.

Code 30: Callbackfunktion für den Display Subscriber

```
//Subscriber Callback Display
void display_string_callback(const void *msgin) {
    const std_msgs__msg__String *msg = (const std_msgs__msg__String *)msgin;
    showMessage(msg->data.data);
}
```

Innerhalb des Setup Blocks wurde dann das Display über die `initDisplay` Funktion und der Subscriber über die entsprechende ROS Funktion initialisiert sowie die Callback Funktion verknüpft, s. Code 31.

Code 31: Callbackfunktion für den Display Subscriber

```
void setup(){
    ...
    initDisplay(string_msg_display);
    ...
    CCHECK(rclc_subscription_init_default(
        &subscriber_display,
        &node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, String),
        "display"));
    ...
    RCCHECK(rclc_executor_add_subscription(
        &executor,
        &subscriber_display,
        &string_msg_display,
        &display_string_callback,
        ON_NEW_DATA));
}
```

Anschließend konnte wieder der Code gebuilded und geflasht werden, um die Implementierung in das ROS Netzwerk zu überprüfen. Die Auflistung der verfügbaren Topics zeigte das Display Topic. Um die Funktionalität nun weiter zu testen, wurde ein String auf das Topic publiziert, s. Code 32.

Code 32: Testung des Display Topics durch simplem publish

```
ros2 topic pub --once /display std_msgs/String {data: 'Display Integration
→ erfolgreich'}
```

---

Anschließend wurde der publizierte String erfolgreich auf dem Display angezeigt, s. Abb. 8.



Abbildung 8: Beispielanzeige des Displays nach publish auf das Display Topic

In der Abbildung ist zu erkennen, dass das dritte Wort getrennt dargestellt ist. Wie erwähnt könnte die Anzeige von den Wörtern oder Strings deutlich besser formatiert sein und so, dass über das ROS Netzwerk noch mehr mögliche Einstellungen, wie Schriftfarbe und -größe eingestellt werden können. In diesem Praktikum wurde jedoch darauf verzichtet, da der Bearbeiter bisher nur grundlegende Kenntnisse im Umgang mit Mikrocontrollern und zuvor keine Grundlagen im Umgang mit ROS hatte und deshalb dieses Praktikums nutzen wollte, um bei den eigenen Sensorprojekten verschiedene Sensoren ausprobieren und testen wollte, anstatt einen möglichst komplex einzubinden.

Es sollte nun also noch ein Node geschrieben werden, welcher den Roboter steuern sollte, indem die zuvor erstellten Topics dafür genutzt werden. Der Node sollte auf dem Host-PC laufen und mit dem Roboter nur über das ROS Netzwerk kommunizieren. Dafür wurde innerhalb des „ezmr\_ws“ ein neues ROS Paket erstellt, mit dem Namen „sensors\_add“. Innerhalb des Pakets wurde eine neue Python Datei erzeugt, in welcher die Robotersteuerung unter Anbindung des Ultraschallsensors und des Displays genutzt werden sollte. Für die Erstellung wurde sich am erstellten Publisher-Subscriber Nodes orientiert. In der `init` Funktion wurden zwei Publisher erzeugt, welche auf das Topic `cmd_vel` und auf das Topic `display` publizieren sollen. Zudem wurde ein Subscriber erzeugt, welcher auf das Topic `ultrasonic_range` hören soll. Zudem wurde eine Methode `range_callback` erzeugt, welche die Range Nachrichten des Ultraschallmoduls abfängt, anschließend den Abstandswert extrahiert und diesen dann auf das Displaytopic publiziert, sodass die aktuelle Entfernung auf dem Display ausgegeben wird. Für die Bewegung des Roboters wurde eine Move Methode geschrieben. Diese publiziert Twist Nachrichten auf das `cmd_vel` Topic. Es wurde so programmiert, bei Methodenaufruf der letzte Distanzwert aufgerufen wird. Falls dieser einen Wert von 0,3 unterschreitet, also ein Objekt in unter 30 cm Entfernung detektiert, dann soll der Roboter eine Drehbewegung nach links (`angular.z>0`). Falls die die Entfernung größer als der gewählte Schwellwert ist, soll der Roboter mit einer geradlinigen Bewegung nach vorn fahren. Das Testen dieses Nodes führte auch zum Erfolg, so wurde die Entfernung stets auf dem Display wiedergegeben und je nach Entfernung führte der Roboter eine entsprechende Bewegung aus. Beim Testen des Nodes gab es jedoch einige Probleme gerade in Bezug auf das Beenden des Nodes. Nach Beenden des Nodes mit strg + c führte dazu, dass in der Konsole der Node beendet wurde, jedoch der Roboter seine vorherige Bewegung behielt und den letzten Distanzwert weiterhin auf dem Display darstellte. Es konnte durch eine `except`-Logik ein verbessertes Beenden implementiert werden, bei welcher die Motoren gestoppt werden. Dennoch gab es beim Ausführen des Programms zum Teil zu Fehlern, sodass der Roboter nach wie vor weiterfuhr. Der konkrete Fehler konnte leider bisher nicht gefunden werden.

## 4.2 „Diebstahlschutz“

In einem nächsten Schritt wurde sich zur Aufgabe gemacht die IMU Einheit, also das MPU 9260, welches am ESP angeschlossen ist einzusetzen. Wie zuvor wurde dafür ein Projekt ohne ROS Inhalte erstellt um die Sensoranbindung zu ermöglichen. Die Einbindung des MPU Sensors führte

zunächst zu Probleme, da Treiberprobleme entstanden und somit der Sensor nicht konkret an gesteuert werden konnte. Nach einiger Recherche und vielen Tests konnte ein passender Treiber aus einem Git Repository entnommen werden. Somit konnte die Integration und Ansteuerung des Sensors erfolgreich durchgeführt werden. Es wurden dabei wieder eine Initialisierungsfunktion geschrieben und eine Funktion welche die Sensorwerte ausliest. Mit der entwickelten Funktion lassen sich die 3 Werte des Beschleunigungssensors auslesen, 3 Werte des Gyroskops, 3 Werte des Magnetsensors und noch die Temperatur des Sensors. In einem nächsten Schritt wurde auch der Piezo Buzzer getestet. Dessen Testung und Einbindung verlief ohne größere Probleme. Bei dem Soundmodul wurde zunächst eine Funktion geschrieben, welche als Eingabe ein Array aus Tönen in Hz und Spielzeiten in Millisekunden annimmt. Dabei wird ein Pointer übergeben und durch Übergabe einer Längenanzahl kann durch das Array iteriert werden. Anschließend wird bei der Iteration bei jedem Ton durch die Arduino Funktion `tone` die entsprechende Frequenz mit der entsprechenden Länge gespielt. Das Programm wurde erweitert, sodass zwei festgelegte Sound implementiert wurde, welche durch Aufruf ihrer Funktion jeweils gespielt werden können. Zum einen wurde somit die Thememelodie von Super Mario integriert, welches beim Aufruf der Funktion `SuperMario` gespielt wird und beim Aufruf der Funktion `IndianaJones` wird die Thememelodie von Indiana Jones abgespielt. Zusätzlich ist es jedoch möglich über den Aufruf von `playCostumMelody`, dass der Nutzer einen String eingibt, welcher dem Format „notes: . , .;durations: . , .“ entspricht, sodass dieser String als Eingabe geparsst wird und daraus die Melodie gespielt wird. Dies wurde zur Integration in das ROS Netzwerk geschrieben, sodass über ein Topic ein String gesendet werden kann, welcher dann zu einer Tonausgabe führt. Des Weiteren wurde eine Initialisierungsfunktion geschrieben, welche für solch einen String eine Speicherreservierung durchführt.

Anschließend mussten der Sensor und der Buzzer wieder in das ROS Netzwerk integriert werden. Es wurde dafür zunächst ein neues Projekt erzeugt und die Grundlegende ROS Funktionalität gemäß Praktikumsanleitung integriert. Anschließend wurden die Anpassungen gemacht, sodass der Buzzer und der IMU integriert werden können. ROS bietet für IMU ein passendes Nachrichtenformat sodass dieses auch für die Übertragung der Sensordaten gewählt wurde. Dabei wurden jedoch nur die Beschleunigungs- und Gyroskop-werte verwendet. Nach einbindung der benötigten ROS Objekte, sowie der Module für Buzzer und IMU konnten dann auch der Subscriber Callback für den Sound definiert werden, s. Code 33.

Code 33: Deklarationen für die Integartion der IMU in das ROS Netzwerk

```
#include "imu.h"
#include "sound.h"
...
#include <sensor_msgs/msg/imu.h>
#include <std_msgs/msg/string.h>
...
cl_publisher_t imu_publisher;
...
sensor_msgs__msg__Imu imu_msg;
std_msgs__msg__String string_msg_sound;
rcl_subscription_t subscriber_sound;

xyzFloat gValue;
xyzFloat gyr;
xyzFloat magValue;
float tempValue;
...
//Subscriber Callback Sound
void sound_callback(const void *msgin) {
    const std_msgs__msg__String *msg = (const std_msgs__msg__String *)msgin;
    String Sound = msg->data.data;

    if (Sound == "Supermario") {
        SuperMario();
    } else if (Sound == "IndianaJones") {
        IndianaJones();
    } else {
        playCustomMelody(Sound.c_str()); // Neue dynamische Variante
    }
}
```

---

Es musste dann wieder innerhalb des Setup-Blocks die Initialisierung des Sensors und des Buzzers durchgeführt werden, weshalb die geschriebene Initialisierungsfunktion aufgerufen wurden. Anschließend wurde noch der Publisher für das IMU Topic initialisiert und der die Callback Funktion für den Sound Subscriber verknüpft, s. Code 34.

Code 34: Setup Block für die Initialisierung der Sensoren, Subscriber und Publisher für Sound und IMU

```
void setup(){
    ...
    init_IMU();
    initSound(string_msg_sound);
    ...
    RCCHECK(rclc_subscription_init_default(
        &subscriber_sound,
        &node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, String),
        "sound"));

    RCCHECK(rclc_executor_add_subscription(
        &executor,
        &subscriber_sound,
        &string_msg_sound,
        &sound_callback,
        ON_NEW_DATA));

    RCCHECK(rclc_publisher_init_default(
        &imu_publisher,
        &node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(sensor_msgs, msg, Imu),
        "imu_values"));
}
```

Abschließend mussten dann innerhalb des Loop-Blocks die Sensorwerte in die IMU Nachricht eingetragen werden und veröffentlicht werden. Wie im vorherigen Projekt werden die Werte wieder alle 200 ms publiziert, s. Code 35.

Code 35: Loop Block für die Veröffentlichung der Sensorwerte

```
void loop(){
    ...
    static unsigned long last_pub_time = 0;
    unsigned long now = millis();

    if (now - last_pub_time >= 200) { // alle 200 ms publizieren
        last_pub_time = now;
        ...

        imu_msg.header.stamp.sec=now/1000;
        imu_msg.header.stamp.nanosec = (now % 1000) * 1000000;
        getValuesIMU(&gValue, &gyr, &magValue, &tempValue);
        imu_msg.angular_velocity.x = gyr.x * M_PI / 180.0f;
        imu_msg.angular_velocity.y = gyr.y * M_PI / 180.0f;
        imu_msg.angular_velocity.z = gyr.z * M_PI / 180.0f;

        imu_msg.linear_acceleration.x=gValue.x*9.8f;
        imu_msg.linear_acceleration.y=gValue.y *9.8f;
        imu_msg.linear_acceleration.z=gValue.z*9.8f;

        rcl_publish(&imu_publisher, &imu_msg, NULL);
    }
}
```

Man kann zudem erkennen, dass noch eine Wertumwandlung durchgeführt wurde. Die zuvor aufgenommenen Sensorwerte der IMU waren bei der Beschleunigung in G. Um es etwas standardisierter zu gestalten, wurden diese Werte in  $\frac{m}{s^2}$ . Bei den Werten des Gyroskopes wurden diese zuvor in  $\frac{\circ}{s}$  ausgegeben, die Werte wurden anschließend noch in  $\frac{rad}{s}$  ausgegeben und auf das ROS Topic publiziert. Die Topics wurden anschließend getestet und es gelang auf sie zuzugreifen und somit die Werte der IMU auszulesen und Melodien den Buzzer zu spielen.

Anschließend sollte ein Node zum „Diebstahlschutz“ geschrieben werden. Dafür sollte der Robo-

---

ter kontinuierlich eine Kreisbewegung fahren. Sobald der Roboter jedoch angehoben wird, soll ein Warnsignal ausgegeben werden. So soll ein möglicher Diebstahl durch Warntöne verhindert werden, natürlich nur rein theoretisch. Innerhalb des zuvor erstellten „sensors\_add“ Pakets wurde deshalb eine weitere Python Datei erzeugt, in welcher diese Funktionalität abgebildet werden sollte. Innerhalb der Initialisierungsmethode des Nodes wurde dafür zwei Publisher für den Motor und den Sound erstellt und einen Subscriber für die IMU Werte. Zusätzlich wurde ein Schwellenwert von  $8.0 \frac{\text{m}}{\text{s}^2}$  für den z-Wert des Beschleunigungswertes festgelegt. Bei Unterschreitung dieses Wertes, soll das Warnsignal ausgegebene werden. Der Werte wurde dabei durch wenige Tests festgelegt. Zudem wurde ein Timer initialisiert, damit bei der Anhebung der Warnton nicht nur einmal abgespielt wird, sondern kontinuierlich bis, der Schwellwert wieder überschritten wurde. Anschließend wurde die Methode `drive_circle` erzeugt, welche Twist Befehle auf das Topic `cmd_vel` sendet, welche einen linear.x Wert von 1,2 und angular.z Wert von 0.5 besitzen. Somit wird eine Kreisbewegung gegen die Uhr durchgeführt. Des Weiteren wurde die `imu_callback` Methode erstellt, welche kontinuierlich die Sensordaten abfragt aus dem Topic und mit dem Schwellwert abgleicht und bei unterschreiten die Methoden zum Abspielend es Warnsignals abspielt. Für das Abspielen des Warnsignals wurden drei Methoden erzeugt. Eine zum Starten, welche einen Timer startet. Die `play_warning` Methode publiziert dann nach Abklingen des Timers eine Nachricht auf das Sound Topic, wodurch ein Warnsignal mit 4040 Hz und einer Dauer von 1 s abgespielt wird. Falls der Schwellwert wieder unterschritten wird, wird eine `stop_warning` Methode aufgerufen, welche das Abspielen des Warnsignals wieder stoppt. Zusätzlich wurde eine Funktion implementiert, welche 2 s nach Aufruf des Nodes das Super Mario Theme abspielt. Dies wurde lediglich aus Demonstrationszwecken und aus Gründen der Freude implementiert. So wird ein String mit „Supermario“ auf das Topic Sound gesendet.

Das Testen des Nodes führte zum Erfolg. Sodass das Anheben des Roboters zu einem Warnsignal führte, welches aufhörte sobald der Roboter wieder abgesetzt wurde. Auch die Super Mario Melodie wurde erfolgreich abgespielt. Bei diesem Node ist jedoch aufgefallen, dass bei einer längeren Ausführung starke Verzögerungen auftreten. So wird auch erst verspätet auch das Beenden des Nodes reagiert. Dennoch wurde die Funktionalität implementiert und ein Minimalbeispiel erstellt.

### 4.3 Zusammenführung der Projekte

Zum Abschluss des Kapitels der Erweiterung des eigenen Roboters mit Sensorintegration, wurden die bisher integrierten Sensoren in einem Projekt zusammen integriert, da vorher zwei separate Projekte erzeugt wurden. Für die Zusammenführung wurden die entsprechenden Codezeilen in eine Datei kopiert. Um das Projekt anschließend noch zu erweitern wurden abschließend noch die beiden Bumper integriert. Auf deren Integration wird nur geringfügig eingegangen, da kein weiteres Projekt in dieser Dokumentation aufgelistet wird, in welchem diese verwendet wurden. Für die Integration der Bumper wurde aus das Bool Nachrichtenobjekte von ROS importiert. So sollte jeder Bumper ein eigenes Topic besitzen auf welches der ESP publiziert. Anschließend wurden noch die Bumper Pins 12, und 14 definiert und es wurden jeweils zwei Publisher und zwei Bool Nachrichteninstanzen deklariert für die Bumper. Im Setup-Block wurden anschließend die Publisher initialisiert und die Bumper-Pins auf Input festgelegt. Im abschließenden loop-Block werden dann die Digitalwerte der Bumper alle 200 ms ausgelesen, wobei die Werte negiert werden, sodass ein Bumper Kontakt den boolschen Wert true zurückgibt. Es gilt zu beachten, dass die Jumper Buchsen der Bumper, erst nach dem Flashen des ESP angeschlossen werden dürfen. Bei einer vorherigen Verbindung kommt es zu einer Fehlermeldung beim Flashvorgang.

Es kann somit abschließend gesagt werden, dass in diesem Kapitel erfolgreich verschiedene Sensoren auf dem Roboter in das ROS Netzwerk integriert wurden und gleichermaßen erfolgreich die Verarbeitung der Daten innerhalb von ROS Nodes durchgeführt wurde, welche mit Python erzeugt wurden. Diese Arbeit hat sehr stark das Interesse des Studenten gefördert. Für die Sensorintegration kann gesagt werden, dass mit die komplexeste Arbeit für den Studenten darin bestand die Sensoren einzubinden und die geforderten Daten auszulesen. Bei der Arbeit mit den ROS Objekten konnte festgestellt werden, dass der Student immer sicherer im Umgang damit wurde, auch wenn ohne eigens erstellten Spickzettel größere Probleme entstanden wären.

---

## 5 Arbeiten mit einem professionellen Roboter: TurtleBot 4

In diesem Abschnitt wird die Arbeit mit dem Turtlebot4 vorgestellt. Wie in der Praktikumsanleitung angegeben wurden zunächst als Vorbereitung die Aufgaben für Zuhause durchgegangen. Dabei wurde sich zu Beginn mit dem Benutzerhandbuch beschäftigt. Im Anschluss wurde die Simulationsumgebung um Gazebo installiert und darin auch versucht den Turtlebot zu navigieren und die Sensordaten über RViz zu visualisieren. Es kann gesagt werden, dass dies prinzipiell funktioniert hat, jedoch die Simulation sehr langsam war, sodass keine flüssige Bewegung vom Roboter angezeigt werden konnte. Aus diesem Grund wurden keine konkreten Aufgaben oder Versuche mit der Simulation durchgeführt, sondern dann eher Zeit mit dem realen Turtlebot verbracht.

Der Turtlebot sollte dabei zunächst manuell gesteuert werden. Davor wurde sich zuerst ein Überblick über die verschiedenen Topics verschafft und anschließend zunächst der Batteriezustand über das Topic `/battery_state` mit dem `echo` Befehl überprüft. Anschließend wurde sich zudem ein Überblick über die vorhandenen Actions gemacht und dann die Action `/undock` ausgeführt, welche den Turtlebot von seiner Ladestation abdocken lies. Dafür musste der folgende Befehl gesendet werden, s. Code 44.

Code 36: Aufruf der `/undock` action zum Abdocken des Turtlebot von seiner Ladestation

```
ros2 action send_goal /undock irobot_create_msgs/action/Undock "{}"
```

Dann konnte die die `/dock` Action getestet werden, welche Analog zur `/undock` Action ausgeführt werden konnte. Im Anschluss sollten sollten auch über RViz Sensordaten visualisiert werden, dafür wurden die Daten des `/scan` Topics verwendet. In Abb. 9 kann gesehen werden, dass die Begrenzungen des Arbeitsbereiches erfasst wurden. Auch die Andeutung der Tür zum kleineren abgeschlossenen Bereich konnten erfasst werden. Es ist zudem zu erkennen, dass auf der Rückseite des Roboters es drei Lücken auf der Begrenzungslinie gibt, diese könnten auf die Kabel und die Kamerahalterung zurückzuführen sein, welche den gescannten Bereich an diesen Stellen beeinflussten. Mit Hilfe der Kommandozeile konnten die Inhalte des Nachrichten analysiert werden. Dabei konnte festgestellt werden, dass der Laserbereich zwischen 15 cm und 12 m aufgenommen werden kann. Dies bekräftigt die These, dass die Lücken durch die Halterung und die Kabel entstehen, da diese sich in der Totzone befinden. Es kann auch erkannt werden, dass der Sensor von  $-\pi$  bis  $\pi$  scannt, was aber auch aus dem Benutzerhandbuch entnommen werden konnte. Im Anschluss wurde eine Kartierung durchgeführt, das Ergebnis kann in Abb. 10 eingesehen werden. Dort ist der abgesteckte Arbeitsbereich deutlich zu erkennen. Mit der aufgenommen Karte konnte der Roboter autonom navigiert werden. Dabei ging die Bedienung sehr einfach. So konnte nach setzen der Initialposition die Zielpositionen durch auswählen auf der Karte ausgewählt werden. Dabei ist aufgefallen, dass die Navigation im großen Bereich problemlos verlief. Auch die Navigation vom großen Bereich in den kleineren Bereich, welcher durch eine Tür abgetrennt werden konnte, verlief ohne Probleme und der Turtlebot hatte keine Probleme bei der Navigation. Bei der Anforderung der Bewegung aus dem kleinen Bereich in den größeren Bereich kam es jedoch zu Problemen, zumindest wenn der Roboter aus dem kleinen Bereich in den Quadranten der Ladestation geschickt wurde. Dabei steuert der Roboter nicht zunächst den Durchgang an, sondern navigierte im kleinen Bereich vermeintlich ziellos umher, bis als Feedback aborted ausgegeben wurde. In RViz wurde jedoch die Bahnplanung korrekt durch den Durchgang angezeigt.

Anschließend sollte eine Beispielanwendung mit dem Turtlebot erzeugt werden. Es wurde sich dazu entschieden verschiedene Funktionen des Turtlebot zu verwenden und dabei aus den zuvor entwickelten Beispielanwendungen Elemente zu integrieren. Es wurden verschiedenen Tests durchgeführt, weshalb zuvor ein neues ROS-Paket erstellt wurde. In der Hauptanwendung wurde ein Node geschrieben, bei welchem vorausgesetzt wird, dass der Roboter sich auf seiner Dockingstation befindet. Das Programm beginnt damit, dass die Abdock-Action ausgeführt wird. Im Anschluss fährt der Turtlebot geradeaus, bis ein Objekt detektiert wird. Es wird also dafür auf das `/cmd_vel` Topic eine lineare Geschwindigkeit in X von 0.15 publiziert. Zur Objektdetektion wird das Topic `/hazard_detection` abonniert, bei diesem wird nur bei Bumper-Kontakt eine Nachricht publiziert. Bei so einer Nachricht soll die Vorwärtsbewegung aufhören und eine Rückwärtsbewegung eintreten, weshalb eine negative Geschwindigkeit für linear.x auf das `/cmd_vel` Topic publiziert wird.

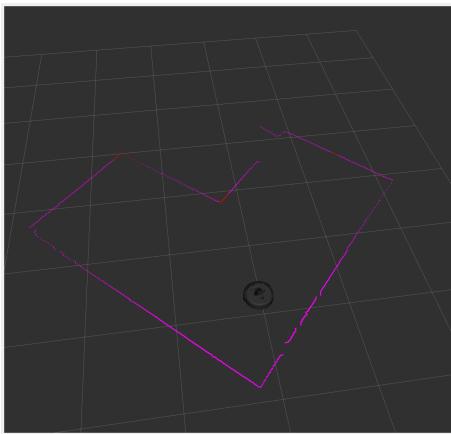


Abbildung 9: Visualisierung der Sensordaten des /scan Topics

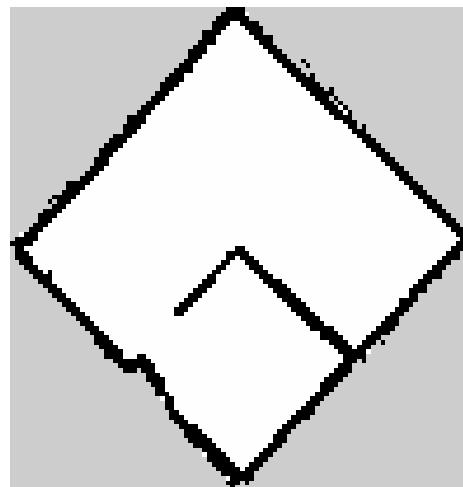


Abbildung 10: Aufgenommene Karte durch die Kartierung

nach kurzem zurückfahren, wird anschließend das Bild der Kamera ausgelesen und abgespeichert, es soll also somit ein Bild vom detektierten Objekt aufgenommen und abgespeichert werden. Im Anschluss wird eine 180° Drehung durchgeführt und der Roboter bewegt sich vorwärts in Richtung Dockingstation zurück. Dafür wurde die Zeit zwischen erstmaligem losfahren nach Abdocken und Objektkontakt gemessen und dient als Orientierung für die Zeit, welche der Roboter zurückfahren soll. Für die Bewegung zurück zur Dockingstation wurde jedoch aufgrund der verkürzten Strecke die Zeit auch verkürzt. Nach abstreichen der Zeit wird dann abschließend noch der Dockingvorgang durch Aufrufen der Docking Action ausgeführt. Der Node wird dann nach Beendigung der Action beendet. Diese Beispieldatenwendung wurde als eine Art Abenteuermodus des Turtlebot entwickelt. Um den abenteuerlichen Charakter zu unterstreichen ertönt bei der Bewegung bis zur Objektdetektion die Melodie von Indiana Jones, also der „Raiders March.“ Die Ausführung des Nodes zeigte, dass die grundlegende Funktionalität erfolgreich umgesetzt werden konnte. Es ist jedoch aufgefallen, dass es viel Verbesserungspotential bei der Anwendung gibt. Die Beispieldatenwendung konnte dabei dennoch helfen verschiedene Topics des Turtlebot einzubinden und in unterhaltender Weise zu kombinieren. Auch die Aufnahme des Bildes konnte erfolgreich durchgeführt werden, in dem durchgeföhrten Versuch wurde der Roboter mit dem Fuß gestoppt, was auf dem aufgenommenen Bild erkannt werden kann. Der entwickelte Quellcode wurde im Github bereitgestellt. Dort ist auch ein Video des Prozesses verlinkt. In dem Video ist zu erkennen, dass sich der Roboter zunächst erfolgreich von der Ladestation abdockt. Im Anschluss wird eine geradlinige Bewegung ausgeführt und der Raiders March abgespielt, bis der Roboter aufgrund der Detektion mit dem Fuß des Studenten anhält. Anschließend bewegt sich der Roboter zurück und nimmt ein Foto auf. Das aufgenommene Bild kann auch im Repository im Workspace Ordner des Turtlebot eingesehen werden. Im Anschluss wird eine Drehbewegung ausgeführt. Diese sollte zukünftig angepasst werden, da der Roboter sich bei dieser Drehung nicht auf der Stelle dreht, wie am Ende beim Undock-Vorgang, sondern eher eine Art Kreisförmige Bewegung ausführt. Somit ist die Position nach dem Zurückfahren vom Roboter Richtung Ladestation wieder versetzt. Im Video ist zudem zu sehen, dass die Docking-Action erfolgreich ausgeführt wird. Dabei führt der Roboter zwar eine unübliche und nicht besonders zielführende Bewegung zu Beginn aus, als er versucht die Ladestation zu lokalisieren, dennoch kann gesehen werden, dass der Roboter am Ende an seinem Ziel ankommt.

---

## 6 Fazit und Ausblick Praktikum

Zusammenfassend kann gesagt werden, dass das Praktikum wirklich viel Wissen vermittelt hat und gleichzeitig auch sehr viel Freude bereitet hat. Es konnte sich gut in die ROS2 Grundlagen eingearbeitet werden und verschiedenen Bereiche kennengelernt werden. Es kann auf jeden Fall gesagt werden, dass dem Studenten hierbei der Fokus vor allem darauf lag möglichst viele verschiedenen Aspekte kennenzulernen. Dies kann auch in den Beispielanwendungen erkannt werden. Diesen mangelt es an einigen Stellen an Robustheit und Optimierung, was bei mehreren Ausführungen bemerkt wurde. Einige Probleme sind durchaus während des Praktikums aufgetreten, vor allem auch bei den Installationen. Eine andere Hürde war auf jeden Fall auch die Motorensteuerung, da dabei einiges an Zeit für das Herausfinden des korrekten Nachrichtenformates für die serielle Schnittstelle für den Motorencontroller aufgewendet werden musste. Es kann dennoch gesagt werden, dass mit der bereitgestellten Praktikumsanleitung die verschiedenen Themen verständlich und zielführend bearbeitet werden konnten und dabei auch in Einzelarbeit. Die Arbeit mit ROS hat auf jeden Fall das Interesse des Studenten gefördert. Gerade die Arbeit mit dem kleinen Roboter konnte sowohl die Fähigkeiten des Studenten in Bezug auf Sensorintegration und Arbeiten mit einem Mikrocontroller fördern, also auch die Begeisterung an (mobiler-) Robotik.

---

## 7 Sockets/ Netcode

In diesem Abschnitt wird der erarbeitete Lösungsversuch der Socket-Programmieraufgaben vorgestellt. Teilaufgabe A und B werden in einem gemeinsamen Abschnitt vorgestellt und getestet. Die erarbeiteten Programme können aus dem bereitgestellten Github Repository entnommen werden. Ein entsprechendes Makefile ist darin auch enthalten.

### 7.1 Teilaufgabe A und B

In Teilaufgabe A sollte eine Anwendung zur Entgegennahme von Anfragen erzeugt werden. Dies wurde durch einen TCP-Socket-Server realisiert. Zunächst wurde ein passiver TCP-Socket angelegt. Die Option `SO_REUSEADDR` erlaubt, den Port nach einem Neustart sofort wieder zu verwenden, ohne dass der Zustand `TIME_WAIT` den Neustart blockiert. Die Struktur `sockaddr_in` wurde mit der gewünschten IP-Adresse und dem Port befüllt. Mit `inet_nton()` wurde die Zeichenkette "127.0.0.1" in binäres Format umgewandelt, das für den Socket benötigt wird. Der Socket wurde an die lokale Adresse gebunden und in den Lauschen-Zustand versetzt. Über die Konsolenausgabe wird dem Nutzer signalisiert, dass der Server bereit ist.

Code 37: Bind und Listen vom TCp-Server

```
bind(server_fd, (struct sockaddr*)&addr, sizeof(addr));
listen(server_fd, 16);
puts("Server hört auf 127.0.0.1:5000 ...");
```

Die Hauptschleife wartet kontinuierlich auf neue Verbindungen. Mit `accept()` wird eine Anfrage entgegengenommen und ein eigener Socket für den jeweiligen Client erzeugt. Die Funktion `inet_ntop()` wandelt die binäre IP-Adresse wieder in eine lesbare Zeichenkette um, um sie im Log auszugeben.

Code 38: Accept und Ausgabe der Clientadresse nach Verbindungsanfrage vom Client

```
for (;;) {
    struct sockaddr_in peer; socklen_t len = sizeof(peer);
    int cfd = accept(server_fd, (struct sockaddr*)&peer, &len);

    char rip[64];
    inet_ntop(AF_INET, &peer.sin_addr, rip, sizeof(rip));
    printf("Verbunden mit %s:%u (fd=%d)\n", rip, ntohs(peer.sin_port), cfd);
    ...
}
```

Damit mehrere Clients gleichzeitig bedient werden können, wird nach jeder akzeptierten Verbindung ein neuer Prozess erzeugt. Der `fork()`-Aufruf liefert im Kindprozess den Wert 0. Über diesen Vergleich wird entschieden, ob der aktuelle Prozess das Kind ist (bearbeitet den Client) oder der Elternprozess (nimmt weitere Verbindungen an). Im Kindprozess wird der Listening-Socket geschlossen, da er dort nicht benötigt wird. Der Elternprozess schließt dagegen den Client-Socket und geht sofort zurück zu `accept()`, um weitere Clients anzunehmen. Innerhalb des Kindprozesses werden die vom Client gesendeten Daten empfangen, ausgegeben und anschließend eine Antwort versendet.

Code 39: Kommunikation mit dem Client, Ausgabe Clientanfrage und automatische Hello-Antwort vom Server

```
ssize_t n = recv(cfd, buf, BUFSZ, 0);
if (n > 0) {
    buf[(n < BUFSZ) ? n : BUFSZ] = '\0';
    printf("Nachricht: %s", buf);

    const char reply[] = "Hello from Server\n";
    send_all(cfd, reply, sizeof(reply) - 1);
}
close(cfd);
_exit(0);
```

```
$ ./simpleServer
Server hört auf 127.0.0.1:5000 ...
[child 4303] Verbunden mit 127.0.0.1:44782 (fd=4)
[child 4305] Verbunden mit 127.0.0.1:41452 (fd=4)
[child 4364] Verbunden mit 127.0.0.1:41456 (fd=4)
[child 4364] Nachricht: Guten Tag
[child 4305] Nachricht: ich bin es
[child 4303] Nachricht: Wer denn?
```

Abbildung 11: Caption

Die Hilfsfunktion `send_all()` sorgt dafür, dass auch bei partiellen Send-Vorgängen alle Bytes zuverlässig übertragen werden. Wenn ein Kindprozess beendet ist, wird durch einen `SIGCHLD`-Handler aufgeräumt. Der Nutzer kann das Programm durch `Ctrl+C` (`SIGINT`) beenden, sodass der Server-Socket geschlossen wird.

In Teilaufgabe B wurde der zugehörige Client entwickelt, der mit dem zuvor entwickelten Server kommuniziert. Auch hier wird ein TCP-Socket verwendet, diesmal allerdings im aktiven Modus. Der Client stellt die Verbindung zum Server her, sendet eine Nutzereingabe und empfängt anschließend die Antwort des Servers. Zunächst wird, analog zum Server, ein Socket erzeugt und mit den entsprechenden Parametern für Protokollfamilie und Übertragungstyp konfiguriert:

Code 40: Erzeugung des Client-Sockets analog zum Server-Socket

```
int client_fd = socket(AF_INET, SOCK_STREAM, 0);
```

Im Anschluss wird die Zieladresse vorbereitet. Die IP-Adresse "127.0.0.1" und der Port 5000 werden in die Struktur `sockaddr_in` eingetragen. Mit `inet_pton()` wird die IP-Adresse wieder aus der Textdarstellung in das binäre Netzwerkformat konvertiert. Nach erfolgreichem Verbindungsauftbau kann der Benutzer über die Konsole eine Nachricht eingeben, die an den Server gesendet wird. Dabei wird die tatsächliche Länge der Eingabe mit `strlen()` ermittelt, um nur die relevanten Bytes zu übertragen.

Code 41: Senden der Nutzereingabe

```
printf("Nachricht eingeben: ");
fgets(sendbuf, sizeof(sendbuf), stdin);

size_t len = strlen(sendbuf, sizeof(sendbuf));
send(client_fd, sendbuf, len, 0);
```

Nach dem Senden wartet der Client auf die Antwort des Servers. Diese wird mit `recv()` entgegengenommen und anschließend auf der Konsole ausgegeben.

Code 42: Empfang und Ausgabe der Serverantwort

```
ssize_t n = recv(client_fd, buffer, sizeof(buffer), 0);
buffer[n] = '\0';
printf("Antwort vom Server: %s", buffer);
```

Abschließend wird die Verbindung mit `close()` beendet, wodurch der Socket und alle zugehörigen Ressourcen freigegeben werden. Der gesamte Ablauf kann beliebig oft wiederholt werden, da der Server durch die in Teilaufgabe A implementierte `fork()`-Struktur mehrere gleichzeitige Clients bedienen kann. Der Client stellt nach Programmstart eine TCP-Verbindung zum Server her, überträgt eine vom Nutzer eingegebene Nachricht und gibt die vom Server zurückgesendete Antwort aus. Dies wurde auch getestet und wird in Abb. 11 abgebildet. Bei dem Test wurde der Server gestartet und drei Clients erzeugt. Anschließend haben die Clients beginnend bei dem jüngsten, Nachrichten gesendet. Dabei kamen alle drei Nachrichten in der korrekten Reihenfolge bei dem Server an. Sodass sichergestellt werden konnte, dass eine simple Kommunikation möglich ist und diese Kommunikation nicht durch Clients, welcher zuvor eine Verbindung erzeugt haben, blockiert wird. Anschließend wurden auch beide Programme mit Netcat überprüft. Zuerst wurde der Server getestet. Dafür wurde der Server wieder gestartet und anschließend in einem neuen Terminal der folgende Befehl ausgeführt:

---

Code 43: Testen des Socket-Server durch Netcat Client

```
nc 127.0.0.1 5000
```

So konnte in dem neuen Terminal eine Nachricht an den Server gesendet werden, welche auch im Serverterminal ausgegeben wurde. Zudem wurde auch die Serverantwort zurückgegeben. In einem weiteren Test sollte der Client getestet werden. Dafür wurde der folgende Befehl ausgeführt, mit welchem ein Listener Socket erzeugt wird:

Code 44: Testen des Socket-Server durch Netcat Client

```
nc -l 5000
```

Beim anschließenden Start eines Clients wurde eine erfolgreiche Verbindung mit dem Netcat Test-objekt erzeugt und die eingegebene Clientanfrage wurde in der Konsole ausgegeben. Eine Serverantwort konnte zudem auch per Konsoleneingabe generiert werden. Die Kommunikation wurde somit erfolgreich sowohl mit dem eigenen Server als auch mit netcat-Tests überprüft.

## 7.2 Teilaufgabe C

In Teilaufgabe C wurde die bestehende Client-Server-Struktur aus den vorherigen Aufgaben so erweitert, dass eine gemeinsam genutzte Liste über eine Socketverbindung gelesen und verändert werden kann. Die A-Anwendung stellt die Liste bereit, während mehrere B-Anwendungen (Clients) auf diese zugreifen und sie manipulieren können. Während in Teilaufgabe A und B jeder Client nur eine einzelne Nachricht sendet und anschließend die Verbindung geschlossen wird, wurde der Server in Teilaufgabe C so angepasst, dass er mehrere Befehle pro Client-Verbindung verarbeiten kann. Zur Realisierung wurde der Server um eine zusätzliche `select()`-basierte Ereignissesteuerung erweitert. Damit kann er auf mehreren aktiven Socketverbindungen gleichzeitig lesen, ohne dabei blockiert zu werden. Jede empfangene Zeile wird auf bekannte Befehle geprüft und anschließend ausgeführt. Der Befehl `/quit` führt dazu, dass der Server die jeweilige Verbindung schließt, während alle anderen Clients unbeeinflusst weiterlaufen. Dadurch ergibt sich ein robustes Mehrbenutzersystem, in dem mehrere Clients parallel mit der geteilten Liste interagieren können. Die Kommunikation folgt einem sehr einfachen textbasierten Protokoll. Jede Zeile, die der Client sendet, wird als Befehl interpretiert:

- `/read` – gibt die aktuelle Liste formatiert zurück (Index, Zeitstempel und Text).
- `/clear` – löscht alle Einträge der Liste.
- `/delete N` – entfernt den Eintrag mit der angegebenen Indexnummer.
- `/quit` – beendet nur die jeweilige Client-Verbindung.
- Alle anderen Texte – werden als neue Listeneinträge (`append`) interpretiert.

Die Datenstruktur zur Speicherung der Einträge wurde als einfacher statischer Array mit Zeitstempeln (`time_t`) umgesetzt. Jede Operation arbeitet direkt auf dieser globalen Liste im Serverprozess. Nach jeder ausgeführten Operation wird eine textuelle Antwort an den Client zurückgesendet. Der Client wurde ebenfalls erweitert und arbeitet nun als interaktive REPL-Anwendung (Read–Eval–Print–Loop). Er bleibt nach dem Start dauerhaft mit dem Server verbunden und akzeptiert wiederholt Nutzereingaben. Jede Eingabezeile wird über `send()` an den Server übermittelt, anschließend wartet der Client auf die vollständige Antwort bis zum Marker `--END--` und gibt diese auf der Konsole aus. Erst wenn der Nutzer den Befehl `/quit` eingibt, beendet sich der Client selbst und trennt die Verbindung. Durch die persistenten Verbindungen entfällt das wiederholte Auf- und Abbauen von TCP-Sockets, was die Kommunikation effizienter macht. Die Anwendungen wurden auch wieder getestet, dabei wurde wieder der Server gestartet und es wurden mehrere Clients erzeugt, welche die gemeinsame Liste manipulieren, dabei wurden keine Komplikationen festgestellt werden. Zudem wurde wieder über den Netcat Befehl aus Code 43 der Server getestet und auch mit dem Befehl konnte ein Client erzeugt werden, welcher die Liste manipulieren und auch auslesen konnte. Zudem wurde der Server auch mit den Clients aus Teilaufgabe B getestet, dies führte auch zu erfolgreichen Ergebnissen, auch wenn natürlich wie erwähnt der Client nur eine Manipulation möglich war.

### 7.3 Teilaufgabe D

In Teilaufgabe D gibt es keine Unterscheidung mehr zwischen Server und Client, nun gibt es nur noch Teilnehmer, sogenannte Peers. Beim Start eines Peers, dieser versucht auf auf 127.0.0.1:Port zu lauschen. Gelingt dies, ist der Peer im Server Modus. Ist der Port bereits belegt, verbindet er sich automatisch als Client und hält eine dauerhafte TCP-Verbindung zum aktiven Server. Die Befehle zur Bearbeitung der Liste sind gleich zur Teilaufgabe C. Damit die Liste beim Beenden des Leaders erhalten bleibt, ist ein *Handover* implementiert. Jeder verbundene Client-FD wird in einer Tracker-Struktur registriert; beim Handover wird der älteste verbundene Client als neuer Leader ausgewählt. Der alte Leader exportiert einen Snapshot der Liste und sendet ihn zusammen mit dem Ziel-Port im @handover-Frame. Der Ziel-Peer importiert die Liste, bestätigt mit @ready und übernimmt den Port (Retry bis 5 s). WICHTIG: Nach @ready schließt der alte Leader zunächst den Listening-Socket, sendet @newleader 127.0.0.1:Port an alle verbleibenden Clients und trennt danach die alten Verbindungen. So erhalten alle Clients den Reconnect-Hinweis zuverlässig und verbinden sich automatisch mit dem neuen Leader. Durch diese Abfolge bleibt der aktuelle Listeninhalt über Leaderwechsel hinweg konsistent und verfügbar. Ein Test der Funktionalität kann in Abb. 12 eingesehen werden.

```
philnuc@Ubuntu22:~/Echtzeitprogrammierung/Sockets_new/sockets$ ./peer
[peer] Server-Modus aktiv (127.0.0.1:5000)
[peer] Befehle: /read | /clear | /delete N | /quit
[peer] Server-Modus auf 127.0.0.1:5000
Apfel
OK: Eintrag (0) hinzugefuegt
/quit
[peer] Handover gesendet, warte auf @ready...
[peer] @ready empfangen
philnuc@Ubuntu22:~/Echtzeitprogrammierung/Sockets_new/sockets$ 
```

```
philnuc@Ubuntu22:~/Echtzeitprogrammierung/Sockets_new/sockets$ ./peer
[peer] Port 5000 belegt > Client-Modus
[peer] Client-Modus > 127.0.0.1:5000
[peer] Befehle: /read | /clear | /delete N | /quit
Banane
OK: Eintrag (1) hinzugefuegt
/read
(0) [2025-10-22 20:50:03] Apfel
(1) [2025-10-22 20:50:07] Banane
(2) [2025-10-22 20:50:10] Kiwi
[peer] Handover empfangen
[peer] Port 5000 uebernommen
[peer] Server-Modus auf 127.0.0.1:5000
/read
(0) [2025-10-22 20:50:03] Apfel
(1) [2025-10-22 20:50:07] Banane
(2) [2025-10-22 20:50:10] Kiwi
```

```
philnuc@Ubuntu22:~/Echtzeitprogrammierung/Sockets_new/sockets$ ./peer
[peer] Port 5000 belegt > Client-Modus
[peer] Client-Modus > 127.0.0.1:5000
[peer] Befehle: /read | /clear | /delete N | /quit
Kiwi
OK: Eintrag (2) hinzugefuegt
[peer] neuer Leader 5000
connect: Connection refused
[peer] verbunden
Birne
OK: Eintrag (3) hinzugefuegt
/delete 1
OK: Eintrag (1) geloescht
/quit
[peer] kein Client fuer Handover
philnuc@Ubuntu22:~/Echtzeitprogrammierung/Sockets_new/sockets$ 
```

Abbildung 12: Test der Socket Peer Anwendung mit drei Peers

Es ist zu erkennen, dass drei Peers gestartet werden. Anschließend schreibt jeder Peer eine Frucht in die Liste: Peer 1 - Apfel, Peer 2 - Banane und Peer 3 - Kiwi. Anschließend liest Peer 2 die Liste aus, welche alle drei Früchte beinhaltet. Danach wird Peer 1 mit dem /quit Befehl beendet. Peer 2 wird nun zum Leader und erhält die Liste über den Handover. Peer 3 wird über den Wechsel benachrichtigt und verbindet sich automatisch mit dem neuen Leader. Anschließend fügt Peer 3 Birne zur Liste hinzu. Peer 2 überprüft die Funktionalität mit dem /read und erhält eine Liste mit vier Früchten. Der Leaderwechsel und Reconnect war also erfolgreich. Anschließend entfernt Peer 3 den zweiten Eintrag der Liste. Peer 2 überprüft das anschließend. Danach beendet Peer 3 seinen Dienst und auch Peer 2 beendet seinen Dienst. Da kein Client mehr verbunden ist, wird kein Handover durchgeführt.

---

## 8 Interprozesskommunikation

In diesem Abschnitt wird der Lösungsversuch der Programmieraufgaben der Interprozesskommunikation vorstellt. Zur Durchführung wurde sich für die IPC-Variante Shared Memory entschieden. Verschiedene Prozesse greifen also auf einen gemeinsamen Speicherbereich zu.

### 8.1 Simple IPC Infrastruktur

Teilaufgabe A fordert die Erstellung einer IPC-Infrastruktur, bei dem ein Programm auf Anfragen antworten kann. In Teilaufgabe B sollen Anwendungen erzeugen, welche Kontakt mit der A-Anwendungen aufnehmen können. Die beiden Teilaufgaben sollen zusammen in diesem Abschnitt bearbeitet werden.

Der Server empfängt die Daten über ein Shared Memory Segment, welches er zuvor angelegt hat. Die Daten verarbeitet dann der Server und schreibt das Ergebnis in das gleiche Shared Memory Segment zurück. Zur Synchronisation der Zugriffe wird eine System-V-Semaphore eingesetzt, welche auch vom Server angelegt wird. Die Semaphore soll verhindern, dass mehrere Clients und der Server gleichzeitig den Speicherbereich verändern, wodurch ein sicherer Datenaustausch zwischen den parallelen Prozessen ermöglicht werden soll. Im entwickelten Beispiel soll sich der Datenaustausch auf die Berechnung der Quersumme einer Zahl fokussieren, ein Client schreibt eine Zahl in den Shared Memory, der Server liest die Zahl aus dem SHM aus, berechnet die Quersumme und schreibt das Ergebnis auch in das SHM. Es wurde die Umsetzung mit Shared Memory gewählt, da es eine sehr schnelle IPC-Variante ist, da keine Kopien zwischen den Anwendungen notwendig sind.

Der gemeinsam genutzte Speicher sollte eine kompakte und klar definierte Datenstruktur enthalten, welche die Protocol Data Unit zwischen Server und Client bildet. Bei der Entwicklung dieser Struktur sollte zudem auch schon auf die Weiterentwicklung geachtet werden, sodass in den weiteren Teilaufgaben mit der Struktur gearbeitet werden kann. Die Struktur sollte zwei Flags besitzen, eine sollte das Vorhandensein einer neuen Clientanfrage darstellen und die andere das Vorhandensein einer neuen Serverantwort. Zudem soll es eine Variable geben in welche der Client die Eingabezahl eintragen kann und eine Variable, in welche die Serverantwort geschrieben werden kann. Zur Übersichtlichkeit und aus Tracking Gründen wurde zudem eine Variable mit der Client-ID angelegt und auch eine Variable für Fehlermeldungen. In Hinblick auf die Teilaufgabe C wurde zudem eine Variable angelegt, welche einen Operationsmodus darstellen soll. In diesem Kapitel wird jedoch nur der Operationsmodus für die Quersumme benötigt.

Code 45: PDU Struktur für SHM-IPC Implementierung

```
typedef struct {
    volatile int req_ready; // Client: 1 = Anfrage liegt an
    volatile int resp_ready; // Server: 1 = Antwort liegt an

    int op; // OP_QSUM (kann später erweitert werden)
    pid_t sender; // PID des Clients (nur Info/Debug)

    int64_t value; // Anfragewert (vom Client gesetzt)
    int result; // Ergebnis (vom Server gesetzt)

    char info[128]; // kurze Info-/Fehlermeldung
} ShmBlock;
```

In der gemeinsamen Header-Datei sind sämtliche zentralen Definitionen und Strukturen für die IPC-Kommunikation zwischen Server und Clients zusammengefasst. Neben den benötigten System-Headern für System-V-IPC werden hier die Konstanten für die Erzeugung des gemeinsamen Schlüssels mittels ftok() (FTOK\_PATH und FTOK\_PROJ) definiert, um sicherzustellen, dass alle Prozesse auf dasselbe Shared-Memory-Segment zugreifen. Außerdem enthält die Datei die Definition des PDU-Datenblocks ShmBlock, der als gemeinsame Speicherstruktur für Anfragen und Antworten dient. Ergänzend sind ein enum OpCode zur Kennzeichnung der Operationen, eine eigene union semun für die Semaphorsteuerung sowie Hilfsfunktionen (die, sem\_lock, sem\_unlock, ensure\_ftok\_file) enthalten, welche die Fehlerbehandlung, Synchronisation und Initialisierung des

---

Shared Memory vereinfachen. Damit bildet die Header-Datei die zentrale Schnittstelle zwischen Server und Clients und stellt sicher, dass beide Seiten mit konsistenten Typen und IPC-Parametern arbeiten.

Wie erwähnt sollte eine Semaphore implementiert werden, welche die Synchronisation gewährleistet und somit Race Conditions verhindert. Es wurde eine binäre Semaphore mit einem Zähler verwendet, welche als Sperre für exklusive Abschnitte dienen soll. Mit der Hilfsfunktion `sem_lock(semid)` wird die Semaphore dekrementiert, diese wird aufgerufen, wenn ein Prozess einen kritischen Abschnitt betritt, also den SHM verändern möchte. Wenn der Prozess den Zugriff auf den SHM beendet, wird mit `sem_unlock` die Semaphore inkrementiert und somit die Ressource wieder freigegeben. Somit wird sichergetellt, dass nur ein Prozess gleichzeitig auf das SHM zugreift. Sowohl Server, als auch Client sollen diese Funktionen kurz vor und nach dem Schreiben aufrufen. Die Semaphor-Operationen nutzen das Flag `SEM_UNDO`, damit das System bei einem Absturz des Prozesses automatisch die Sperre wieder freigibt.

Das erzeugte Serverprogramm startet damit, dass über `ftok()` ein Schlüssel anhand von dem Pfad und der ID aus der Header Datei erzeugt wird. Anschließend wird mit `shmget()` das Shared Memory Segment erzeugt und an den Adressraum angehängt mit `shmat()`. Mit `semget()` wird eine Semaphore erzeugt und mit dem Wert 1 initialisiert. Nach der Initialisierung schläft der Server zyklisch und wartet dabei darauf, dass das Flag `req_ready` auf 1 gesetzt wird, somit also ein Client eine Anfrage gestellt hat. Der Server dekrementiert anschließend die Semaphore und Sperrt somit den Speicherzugriff für die anderen Prozesse. Der Server lässt die Anfrage aus und setzt das `req_ready` Flag wieder auf 0. Wenn als `op OP_QSM` gesetzt ist, dann wird die Quersumme aus dem `value` berechnet und in `result` geschrieben. Dann wird der Flag `resp_ready` auf 1 gesetzt und die Semaphore inkrementiert, also freigegeben. Falls der Prozess beendet wird durch `SIGINT` (Interrupt, bspw. durch `Ctrl + C`) oder `SIGTERM` (Terminate, bspw. durch `kill`), dann werden die SHM und die Semaphore entfernt, sodass keine Ressourcen nach der Beendigung im System zurückbleiben.

In Teilaufgabe B wurde ein Client gefordert, welcher an den Server Anfragen senden und Serverantworten empfangen kann. Der entwickelte Client verwendet die gleiche Header-Datei. Analog zum Server erzeugt der Client somit denselben Schlüssel über `ftok()` und bindet sich über `shmget()` und `shmat()` an das Shared-Memory-Segment. Über `semget()` wird die vom Server angelegte Semaphore geöffnet. Um eine Anfrage an den Server zu starten, gibt der Benutzer in der Konsole eine Zahl ein, deren Quersumme berechnet werden soll. Anschließend wird mit `sem_lock()` der exklusive Schreibzugriff auf den geteilten Speicher gesichert. Der Client befüllt dann den `ShmBlock` mit `op=OP_QSUM`, `sender=getpid()`, `value` mit der Nutzereingabe und anschließend setzt er noch `req_ready=1` und `resp_ready=0`, womit signalisiert wird, dass eine neue Anfrage bereit liegt. Danach verlässt der Client den kritischen Bereich (`sem_unlock()`) und wartet bis der Server `resp_ready=1` gesetzt hat, also der Server die Quersumme berechnet hat. Sobald `resp_ready` aktiv ist, liest der Client die Quersumme aus dem Feld `result` und noch die zusätzlichen Infos aus `info`. Dann setzt noch der Client nach dem Auslesen `resp_ready` auf 0. Beim Beenden des Clients durch Nutzereingabe von `/quit`, trennt sich der Client sauber vom Segment über `shmdt()`.

Zusätzlich wurde eine weitere Clientanwendung entwickelt, diese sollte im Gegensatz zu der zuvor beschriebenen jedoch eine reine Lesende Funktionalität besitzen, also quasi ein passiver Beobachter, des Datenverkehrs. Der Client sollte weder etwas in den geteilten Speicher schreiben, noch die Semaphore nutzen. Das Ziel war jede Clientanfrage zu erkennen und die dazugehörige Serverantwort zu protokollieren. Zu Beginn des Prozesses wird auch hier der geteilte Speicher eingehangen durch `ftok()`, `shmget()` und `shmat()`. Der Client arbeitet zustandslos und erkennt die Ereignisse über Flanken, durch abspeichern der vorherigen Zustände. Wenn also das Flag `req_ready` auf 1 geht, nimmt der Client einen Snapshot von `op`, `value` und `sender` auf und loggt diese. Das gleiche passiert mit der Serverantwort, wenn das Flag `resp_ready` auf 1 geht. Es wurde bewusst darauf verzichtet keinen Semaphore zu verwenden, da der Client somit den Datenfluss nicht beeinflusst.

---

Es ist jedoch zu beachten, dass dadurch theoretisch inkonsistente Zwischenstände sichtbar werden könnten. Zum Test der Funktionalität wurden die Programme für Client und Server gemäß Makefile kompiliert und gestartet. Anschließend wurden zwei Zahlen vom Nutzer eingegeben, von welchen der Server die Quersumme berechnen sollte. In Abb. 13 kann eingesehen werden, dass der Server auf die Clientanfrage mit einer korrekten Berechnung die Quersumme zurückgibt. Die Abb. 14 zeigt hingegen die andere Clientanwendung, welche lediglich die Daten des Shared-Memory ausliest und somit den Datenverkehr zwischen Clients und Server abhört.

```
Quersummen-Client. Zahl eingeben und Enter drücken.
Befehle: /quit beendet.
> 73645
SERVER:

PID=4176: Quersumme(73645) = 25
Ergebnis: 25

> 29450
SERVER:

PID=4176: Quersumme(29450) = 20
Ergebnis: 20
```

Abbildung 13: Terminalauszug des Clients bei Testung Serverantwort

```
[listener] verbunden. Beenden mit Ctrl+C.
[listener][2025-10-17 18:59:04] Anfrage erkannt:
  sender : 4176
  value   : 73645

[listener][2025-10-17 18:59:04] Antwort eingetroffen:
  (zu) sender : 4176
  (zu) value   : 73645
  result    : 25
  info      : PID=4176: Quersumme(73645) = 25

[listener][2025-10-17 18:59:12] Anfrage erkannt:
  sender : 4176
  value   : 29450

[listener][2025-10-17 18:59:12] Antwort eingetroffen:
  (zu) sender : 4176
  (zu) value   : 29450
  result    : 20
  info      : PID=4176: Quersumme(29450) = 20
```

Abbildung 14: Terminalauszug des Hörer-Clients bei der Testung der Kommunikation

Zudem sollte noch die IPC Infrastruktur mit ipcs und ipcrm überprüft werden. Mit ipcs kann angezeigt werden, welche IPC-Ressourcen im System existieren, also beispielsweise die Shared-Memory Segmente und die Semaphoren. Mit dem Befehl `ipcs -m -s` können die Shared Memory Segmente und die Semaphoren angezeigt werden. Vor dem Test wurden alle Prozesse geschlossen und dann der Befehl ausgeführt. Wie zu erwarten war, wurden somit keine Shared Memory oder Semaphoren angezeigt, s. Abb. 15a. Anschließend wurde der Server gestartet, dies sollte dazu führen, dass nun sowohl ein SHM-Segment, als auch ein Semaphoren-Objekt angezeigt werden, bei erneutem Ausführung des Befehls, s. Abb. 15b. Dabei ist zu erkennen, dass die Rechte 666 sind, dies bedeutet, dass sowohl Schreib, als auch Lesezugriff für alle Teilnehmer vorhanden ist. Zudem zeigt der Wert nattch, dass bisher nur ein Prozess an das Shared-Memory-Segment angehangen ist. In einem nächsten Schritt sollen noch die beiden Clients gestartet werden. Theoretisch sollten sich somit keine weiteren SHM-Segmente oder Semaphoren öffnen, sondern lediglich der Wert von nattch um jeweils 1 erhöhen. Nach dem Start von dem Nutzerclient wurde der Befehl erneut ausgeführt. Wie in Abb. 15c zu sehen ist, erhöhte sich der Wert von nattch um 1 auf 2. Anschließend wurde noch der Beobachterclient gestartet und der Befehl ausgeführt. Auch hier wurde wieder der Wert von nattch um 1 erhöht, s. Abb. 15d. Abschließend sollte noch getestet werden, was passiert, wenn der Server beendet wird, jedoch die Clients noch aktiv bleiben. Dafür wurde der Server durch `Ctrl + C` beendet und anschließend wieder `ipcs -m -s` ausgeführt. In Abb. 15e ist zusehen, dass nun der Status als zerstört beschrieben ist, jedoch immer noch 2 Prozesse an des Shared-Memory-Segment anhängt sind. Die Semaphore wurde jedoch auch entfernt. Nach Beendigung der beiden Prozesse kann festgestellt werden, dass kein SHM-Segment mehr vorhanden ist.

----- Gemeinsamer Speicher: Segmente -----						
Schlüssel	shmid	Besitzer	Rechte	Bytes	nattach	Status
----- Semaphorenfelder -----						
Schlüssel	SemID	Besitzer	Rechte	nsems		

(a) Initialer Test zur Überprüfung vorhandener SHM-Segmente und Semaphoren

----- Gemeinsamer Speicher: Segmente -----						
Schlüssel	shmid	Besitzer	Rechte	Bytes	nattach	Status
----- Semaphorenfelder -----						
Schlüssel	SemID	Besitzer	Rechte	nsems		

(b) Test nach Starten der Serveranwendung

----- Gemeinsamer Speicher: Segmente -----						
Schlüssel	shmid	Besitzer	Rechte	Bytes	nattach	Status
----- Semaphorenfelder -----						
Schlüssel	SemID	Besitzer	Rechte	nsems		

(c) Test nach Starten Clientanwendung

----- Gemeinsamer Speicher: Segmente -----						
Schlüssel	shmid	Besitzer	Rechte	Bytes	nattach	Status
----- Semaphorenfelder -----						
Schlüssel	SemID	Besitzer	Rechte	nsems		

(d) Test nach Starten Beobachteranwendung

----- Gemeinsamer Speicher: Segmente -----						
Schlüssel	shmid	Besitzer	Rechte	Bytes	nattach	Status
----- Semaphorenfelder -----						zerstört
Schlüssel	SemID	Besitzer	Rechte	nsems		

(e) Test nach Beendigung Serveranwendung

Abbildung 15: Übersicht der Tests mit ipcs

Zusammenfassend kann gesagt werden, dass in Teilaufgaben A und B eine funktionale und leicht erweiterbare Shared-Memory-IPC-Lösung entwickelt wurden. Es liegt ein klar definiertes PDU-Protokoll vor, welches von Server und Client-Anwendungen verwendet wird. Datenkorruptionen werden zudem durch die integrierte Semaphore verhindert. Zudem kann in den folgenden Aufgaben die Struktur aufgegriffen und durch entsprechende weitere Operationen ergänzt werden.

---

## 8.2 IPC Listenmanipulation

In Teilaufgabe C wurde die Client-Server-Architektur aus den Teilaufgaben A und B erweitert, um eine gemeinsame dynamische Textliste über Shared-Memory zu verwalten. Es sollten dabei weitere Operationsmodi eingefügt werden. So sollten es durch `/append` möglich werden neue Einträge hinzuzufügen, durch `/read` sollte die gesamte Liste ausgegeben werden, mit `/delete N` sollte ein einzelner Eintrag mit der ID N gelöscht werden können und mit `/clear` sollte die vollständige Liste gelöscht werden. So sollte also eine kooperative Mehrnutzer-Anwendung ermöglicht werden, welche einen gemeinsamen Speicher nutzt.

Die bisherige Struktur `ShmBlock` wurde um eine eingebettete Liste erweitert. Dafür wurden die Strukturen `SharedList` und `ListEntry` erzeugt, wobei `SharedList` die Zahl der aktuellen Listen-Einträge beinhaltet und ein Array aus Listeneinträgen `ListEntry`. `ListEntry` besitzt den Text des Listeneintrages und noch einen Zeitstempel, zu welchem Zeitpunkt der Listeneintrag hinzugefügt wurde. Der gemeinsame Speicher ist somit nicht mehr nur Datenkanal, sondern nun auch gemeinsamer Datencontainer.

Um die neuen Befehle zu integrieren wurde zunächst der Enum `OpCode` um die entsprechenden Operationen erweitert. Das Serverprogramm wurde anschließend nur in der Hauptschleife weiter angepasst. Es wurde die IF-Verzweigung um die neuen Operationen ergänzt. An dem Anlegen des SHM-Segmentes oder der Semaphore wurde nichts verändert, sodass die Synchronisation weiterhin über die bestehende Semaphore und deren Hilfsfunktionen abläuft. So kann nur ein Client gleichzeitig auf den gemeinsamen Speicher zugreifen, wodurch Race Conditions verhindert werden. Auch das Clientprogramm wurde geringfügig angepasst. Es wurde lediglich integriert, dass die Nutzereingaben korrekt verstanden werden nun mit den weiteren Befehlen. Der Client schreibt also weiterhin seine Anfrage in den Shared-Memory Block, setzt dann das `req_ready` Flag auf 1 und wartet, bis der Server das Ergebnis mit `resp_ready 1` signalisiert. Dann liest der Client die Antwort aus und gibt sie formatiert in der Konsole aus.

Zum Testen wurden die beiden Programme gemäß Makefile kompiliert und gestartet. Im Anschluss wurden mit einigen Nutzereingaben die Funktionalität getestet. Das Verhalten wurde in Abb. 16 dokumentiert. Zu Beginn wurden 9 Einträge zur Liste hinzugefügt, welche zusammen die Zutaten für ein einfaches Pancake-Rezept abbilden. Diese konnten durch die Eingabe in der Konsole und dem Bestätigen durch Enter hinzugefügt werden. Im Anschluss wurde die Liste mit dem Befehl `/read` ausgegeben. Dabei werden auch die entsprechenden ID und die Zeitstempel der Eintragung angezeigt. Im Anschluss wurde mit dem `/delete` Befehl der Listeneintrag 5 gelöscht. Durch einen anschließenden `/read` Befehl wird dies überprüft und es kann gesehen werden, dass Nutella entfernt wurde. Im Anschluss wird die Liste mit dem Befehl `/clear` vollständig gelöscht. Dies wird auch wieder mit `/read` erfolgreich überprüft.

```
Befehle: /read | /clear | /delete N | /quit
Textzeile ohne Slash = append
> 250 g Weizenmehl
SERVER: OK: Eintrag (0) hinzugefügt
> 2 EL Zucker
SERVER: OK: Eintrag (1) hinzugefügt
> 2 TL Backpulver
SERVER: OK: Eintrag (2) hinzugefügt
> 1 Prise Salz
SERVER: OK: Eintrag (3) hinzugefügt
> 2 Eier
SERVER: OK: Eintrag (4) hinzugefügt
> Nutella
SERVER: OK: Eintrag (5) hinzugefügt
> 200 ml Milch
SERVER: OK: Eintrag (6) hinzugefügt
> Speiseöl
SERVER: OK: Eintrag (7) hinzugefügt
> Ahornsirup
SERVER: OK: Eintrag (8) hinzugefügt
> Beeren
SERVER: OK: Eintrag (9) hinzugefügt
> /read
SERVER:

(0) [2025-10-17 21:08:43] 250 g Weizenmehl
(1) [2025-10-17 21:08:49] 2 EL Zucker
(2) [2025-10-17 21:08:55] 2 TL Backpulver
(3) [2025-10-17 21:09:01] 1 Prise Salz
(4) [2025-10-17 21:09:05] 2 Eier
(5) [2025-10-17 21:09:09] Nutella
(6) [2025-10-17 21:09:13] 200 ml Milch
(7) [2025-10-17 21:09:17] Speiseöl
(8) [2025-10-17 21:09:21] Ahornsirup
(9) [2025-10-17 21:09:25] Beeren
> /delete 5
SERVER: OK: Eintrag (5) gelöscht
> /read
SERVER:

(0) [2025-10-17 21:08:43] 250 g Weizenmehl
(1) [2025-10-17 21:08:49] 2 EL Zucker
(2) [2025-10-17 21:08:55] 2 TL Backpulver
(3) [2025-10-17 21:09:01] 1 Prise Salz
(4) [2025-10-17 21:09:05] 2 Eier
(5) [2025-10-17 21:09:13] 200 ml Milch
(6) [2025-10-17 21:09:17] Speiseöl
(7) [2025-10-17 21:09:21] Ahornsirup
(8) [2025-10-17 21:09:25] Beeren
> /clear
SERVER: Liste wurde gelöscht
> /read
SERVER:

(leer)
>
```

Abbildung 16: Konsole mit Tests der Funktionalität zur Manipulation einer Liste über Shared-Memory IPC

---

### 8.3 IPC Peer Struktur

In Teilaufgabe sollte keine Unterscheidung mehr zwischen A-Anwendungen und B-Anwendungen mehr vorliegen, es soll es nicht mehr eine Server-Client Architektur erzeugt werden. Dafür wird also eine Peer-Anwendung erzeugt, also Teilnehmer. Beim Start erzeugt oder öffnet jeder Teilnehmer wieder über den gleichen Key Mechanismus mit `ftok` und `shmget` ein gemeinsames Shared-Memory-Segment. vom Typ `ShmState`. Dieses besitzt eine referenzgezählte Liste der Teilnehmer und die Nachrichtenliste mit Zeitstempeln. Die Semaphore schützt alle schreibenden Zugriffe, sodass nur ein Peer den SHM bearbeiten darf. Jeder neue Peer erhöht den `refcount`, beim Beenden eines Peers wird dieser wieder dekrementiert. Der letzte Peer entfernt dann das SHM und die Semaphore. Die Befehle zur Listenbearbeitung haben sich im Vergleich zur vorherigen Aufgabe nicht verändert. Im Unterschied zur Teilaufgabe C, ist es nun jedoch so, dass die Ausgabe bei `/read` anders abläuft. Hier wird nun ein Snapshot im Lock gemacht und die Formatierung und Ausgabe geschieht wenn die Semaphore bereits wieder verlassen wurde.

Der Test des Peer Programms wurde durch Erzeugung von fünf Peers erzeugt, welche alle einen Eintrag in die Liste schreiben sollten. Anschließend wurden alle Peers beendet und überprüft, ob weiterhin ein SHM verwendet wird und ob bei einer erneuten Öffnung eines Peers die Liste noch verfügbar ist. Dies wurde wieder mit dem Befehl `ipcs -s -m` ausgeführt. Es konnte dabei festgestellt werden, dass sich die Liste durch die verschiedenen Teilnehmer ohne Komplikationen bearbeiten lassen. Es wurde auch nur ein SHM Segment erzeugt und nur eine Semaphore. Das schließen beziehungsweise beenden aller Peers führte dazu, dass das SHM Segment und die Semaphore durch den `ipcs -s -m` Befehl nicht mehr angezeigt wurden und somit sauber entfernt wurden. Auch beim Starten eines neuen Peers als ersten Peer führte dazu, dass eine leere Liste vorlag und nicht noch eine veraltete.

## 9 Fazit Programmieraufgaben

Die Programmieraufgaben konnten erfolgreich absolviert werden. Es wurden alle Teilaufgabenstellungen absolviert und die Anforderungen umgesetzt und die Programme auf verschiedenen Wegen getestet. Dabei kann gesagt werden, dass für die Entwicklung und das Verständnis von Sockets und Interprozesskommunikation die man pages sowie die Kapitel 9 und 11.4 aus Linux-UNIX-Programmierung [1] verwendet wurden. Es ist dennoch anzumerken, dass die Programme unter Zuhilfenahme von künstlicher Intelligenz geschrieben wurden. Die Ergebnisse wurde stets getestet und kritisch untersucht. Die Programmieraufgaben haben auf jeden Fall das Verständnis von Sockets und Shared Memory des Studenten erweitert und das Interesse daran gefördert. Der entwickelte Quellcode, sowie die Makefiles können aus dem bereitgestellten Github Repository entnommen werden.

---

## Quellenverzeichnis

- [1] Jürgen Wolf. *Linux-UNIX-Programmierung*. Openbook, 2., aktualisierte und erweiterte Auflage. 2006. URL: [https://openbook.rheinwerk-verlag.de/linux\\_unix\\_programmierung/index.htm](https://openbook.rheinwerk-verlag.de/linux_unix_programmierung/index.htm) (besucht am 22. Okt. 2025).