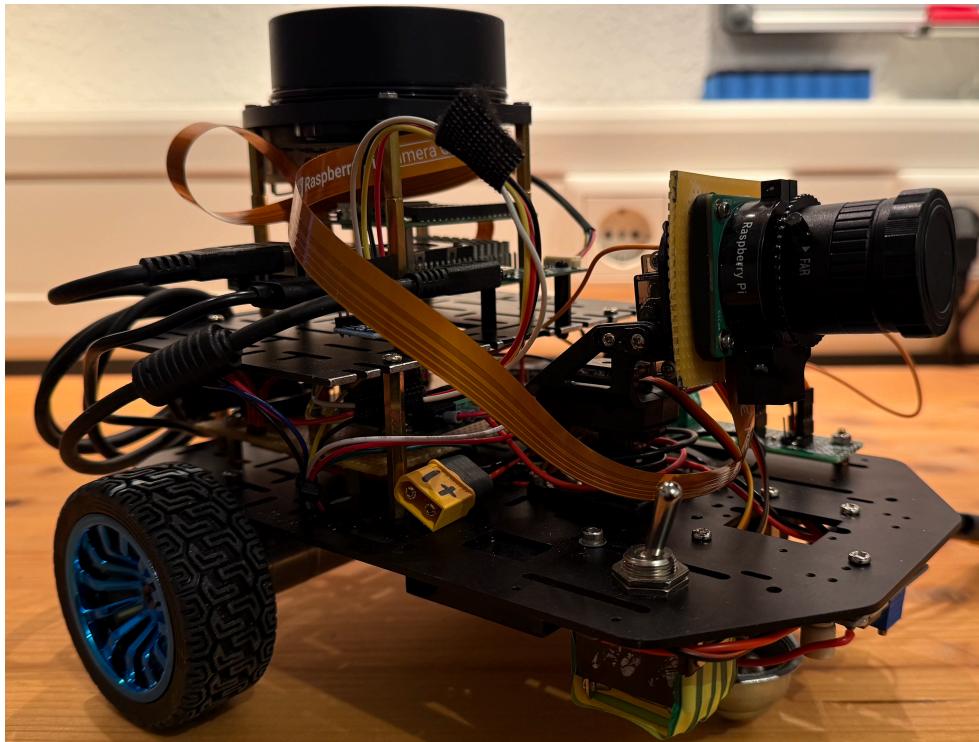


Autonomous Mobile Robot

AMR-Roadmap

„Vom Lötkolben zum SLAM“

Interdisziplinäres Projekt (Mechanik • Elektronik • Software)



Technologie-Stack

ESP32-S3 • FreeRTOS • micro-ROS • ROS 2 Jazzy • Nav2

Inhaltsverzeichnis

1 Phase 1: micro-ROS auf ESP32-S3 (USB-Serial)	1
1.1 Zielbild & Definition of Done	1
1.2 Testergebnisse (2025-12-20)	2
1.3 Systemübersicht	3
1.4 Topics (verifiziert)	4
1.5 Hardware (Phase 1)	4
1.6 Firmware – Parameter (v3.2.0)	5
1.7 Build / Flash / Monitor (PlatformIO)	6
1.8 Docker-Setup (Pi 5)	6
1.9 Smoke-Tests	7
1.10 Troubleshooting	9
1.11 Bekannte Einschränkungen	9
1.12 Nächste Schritte	10
1.13 Changelog	10
2 Phase 1 – Befehlsreferenz	10
2.1 Nach Pi Reboot	10
2.2 Nach ESP32 Reboot (USB umgesteckt)	11
2.3 Notfall-Stop (falls Räder drehen)	11
2.4 Smoke-Tests	11
2.5 Motor-Tests	13
2.6 Failsafe-Test	14
2.7 Odom nach Fahrt prüfen	14
2.8 Checkliste Phase 1	14
2.9 Troubleshooting	15
3 Phase 2: ROS 2 Humble auf Raspberry Pi 5 (Docker) + micro-ROS Agent	15
3.1 Zielbild & Definition of Done	15
3.2 Docker-Images (Regel)	16
3.3 Host-Voraussetzungen	16
3.4 Repo-Struktur	17
3.5 Docker Compose	17
3.6 Befehle	18
3.7 Smoke-Tests	19
3.8 Troubleshooting	20
3.9 Verifizierte Konfiguration	21
3.10 Changelog	21

4 Phase 3: RPLidar A1 Integration (+ /scan)	21
4.1 Zielbild & Definition of Done	21
4.2 Hardware-Info (verifiziert)	22
4.3 Docker-Integration	23
4.4 Installation	23
4.5 RPLidar starten & Smoke-Tests	24
4.6 Troubleshooting	25
4.7 Statischer TF (temporär für RViz2)	26
4.8 Nächste Schritte (Phase 4)	26
4.9 Changelog	27
5 Phase 3: RPLidar A1 – Befehlsreferenz	27
5.1 Quick Start	27
5.2 Erwartete Topics	28
5.3 Scan-Daten Referenz	28
5.4 Troubleshooting	29
5.5 Hardware-Info (RPLidar A1)	29
6 AMR Implementierungsplan	30
6.1 Grundprinzip: Vertikale Scheiben statt horizontaler Schichten	30
6.2 Phasen-Übersicht	31
6.3 Phase 1: micro-ROS auf ESP32-S3	32
6.4 Phase 2: Docker-Infrastruktur	35
6.5 Phase 3: RPLidar A1 Integration	36
6.6 Phase 4: EKF Sensor Fusion	37
6.7 Phase 5: SLAM (slam_toolbox)	37
6.8 Phase 6: Nav2 Autonome Navigation	38
6.9 Zukünftige Erweiterungen (optional)	38
6.10 Hardware-Übersicht	39
6.11 Risikomatrix	39
6.12 Zeitplan (8 Wochen, grob)	40
6.13 Checkliste pro Phase	40
6.14 Changelog	40
7 Entwicklerdokumentation: AMR Low-Level Controller	41
7.1 Architektur-Design	41
7.2 Hardware Abstraction Layer (HAL)	44
7.3 Firmware-Logik (<code>main.cpp</code>)	44
7.4 Schnittstellen & Datenfluss	46

7.5	Konfiguration (<code>config.h</code>)	47
7.6	Deployment	48
7.7	Testing	49
7.8	Troubleshooting	50
7.9	Bekannte Einschränkungen	50
7.10	Nächste Entwicklungsschritte	51
7.11	Changelog	51
8	Systemdokumentation: AMR Low-Level Controller	52
8.1	Architektur-Übersicht	52
8.2	Firmware-Architektur (Dual-Core)	54
8.3	ROS 2 Schnittstelle (API)	55
8.4	Konfiguration & Parameter	56
8.5	Inbetriebnahme	57
8.6	Testergebnisse (20.12.2025)	59
8.7	Known Issues & Lösungen	59
8.8	Bekannte Einschränkungen	59
8.9	Projektstruktur	60
8.10	Changelog	60
8.11	Nächste Schritte	61
9	Git-Workflow: Mac ↔ GitHub ↔ Raspberry Pi	62
9.1	Übersicht	62
9.2	Initiales Setup	62
9.3	Täglicher Workflow	64
9.4	Commit-Konventionen	66
9.5	Branching-Strategie (optional)	66
9.6	Synchronisation Mac ↔ Pi	67
9.7	Häufige Szenarien	67
9.8	Nützliche Aliase (optional)	68
9.9	Backup-Strategie	69
9.10	Projekt-Struktur	70
9.11	Checkliste: Git richtig nutzen	71
9.12	Zusammenfassung: Der goldene Pfad	71
9.13	Quick Reference Card	72
10	Industriestandards für AMR-Entwicklung	72
10.1	ROS-Standards: REPs (ROS Enhancement Proposals)	72
10.2	Sicherheitsstandard (Safety)	73

10.3 Architektur-Standard: Hybrid Master-Slave	74
10.4 Code-Qualität (Best Practices)	74
10.5 Projekt-Checkliste (minimal)	74
11 Kostenübersicht AMR-Projekt	75
11.1 Übersicht nach Händler	75
11.2 Detaillierte Aufstellung	75
11.3 Kostenstruktur nach Kategorie	77
11.4 Bewertung	78
11.5 Nicht enthaltene Kosten (Schätzung)	78
12 ToDo-Liste AMR-Projekt	79
12.1 Phasen-Übersicht	79
12.2 Phase 1: micro-ROS ESP32-S3 (abgeschlossen)	79
12.3 Phase 2: Docker-Infrastruktur (abgeschlossen)	80
12.4 Phase 3: RPLidar A1 (abgeschlossen)	80
12.5 Phase 4: URDF + TF + EKF (nächste Phase)	80
12.6 Phase 5: SLAM	81
12.7 Phase 6: Nav2	81
12.8 Aktuelle Hardware-Ports	82
12.9 Quick Reference	82

1 Phase 1: micro-ROS auf ESP32-S3 (USB-Serial)

Status & Version	
<ul style="list-style-type: none">• Status: completed• Updated: 2025-12-20• Version: 3.2.0• Source: <code>firmware/src/main.cpp</code>, <code>firmware/include/config.h</code>, <code>firmware/platformio.ini</code>	1

1.1 Zielbild & Definition of Done

Zielbild:

- ESP32-S3 läuft als **micro-ROS Client** über **USB-CDC (Serial)**.
- `/cmd_vel` steuert Motoren über **Cytron MDD3A Dual-PWM**.
- `/odom_raw` wird publiziert (`geometry_msgs/Pose2D`) und ist plausibel.
- **Failsafe** stoppt Motoren nach `FAILSAFE_TIMEOUT_MS = 2000`.

DoD (verifiziert 2025-12-20):

- ✓ Agent verbindet stabil (Reconnect reproduzierbar).
- ✓ `/cmd_vel` wirkt (vor/zurück/rotieren).
- ✓ `/odom_raw` plausibel (x steigt vorwärts, θ bei Drehung).
- ✓ Timeout-Failsafe stoppt deterministisch nach ≈ 2 s.
- ✓ `/esp32/heartbeat` läuft (≈ 1 Hz).

1.2 Testergebnisse (2025-12-20)

Test	Befehl	Ergebnis	Status
Agent-Verbindung	-	fd: 3 stabil	✓
Heartbeat	ros2 topic echo /esp32/heartbeat	≈ 1 Hz	✓
Vorwärts	linear.x: 0.15	Räder drehen vorwärts	✓
Rückwärts	linear.x: -0.15	Räder drehen rückwärts	✓
Drehen links	angular.z: 0.5	Roboter dreht links	✓
Drehen rechts	angular.z: -0.5	Roboter dreht rechts	✓
Failsafe	Ctrl+C, 2s warten	Motoren stoppen	✓
Odom	ros2 topic echo /odom_raw	x, y, θ plausibel	✓

Odom-Beispiel nach Testfahrt:

```
x: 0.899
y: -0.329
theta: 6.09
```

1.3 Systemübersicht

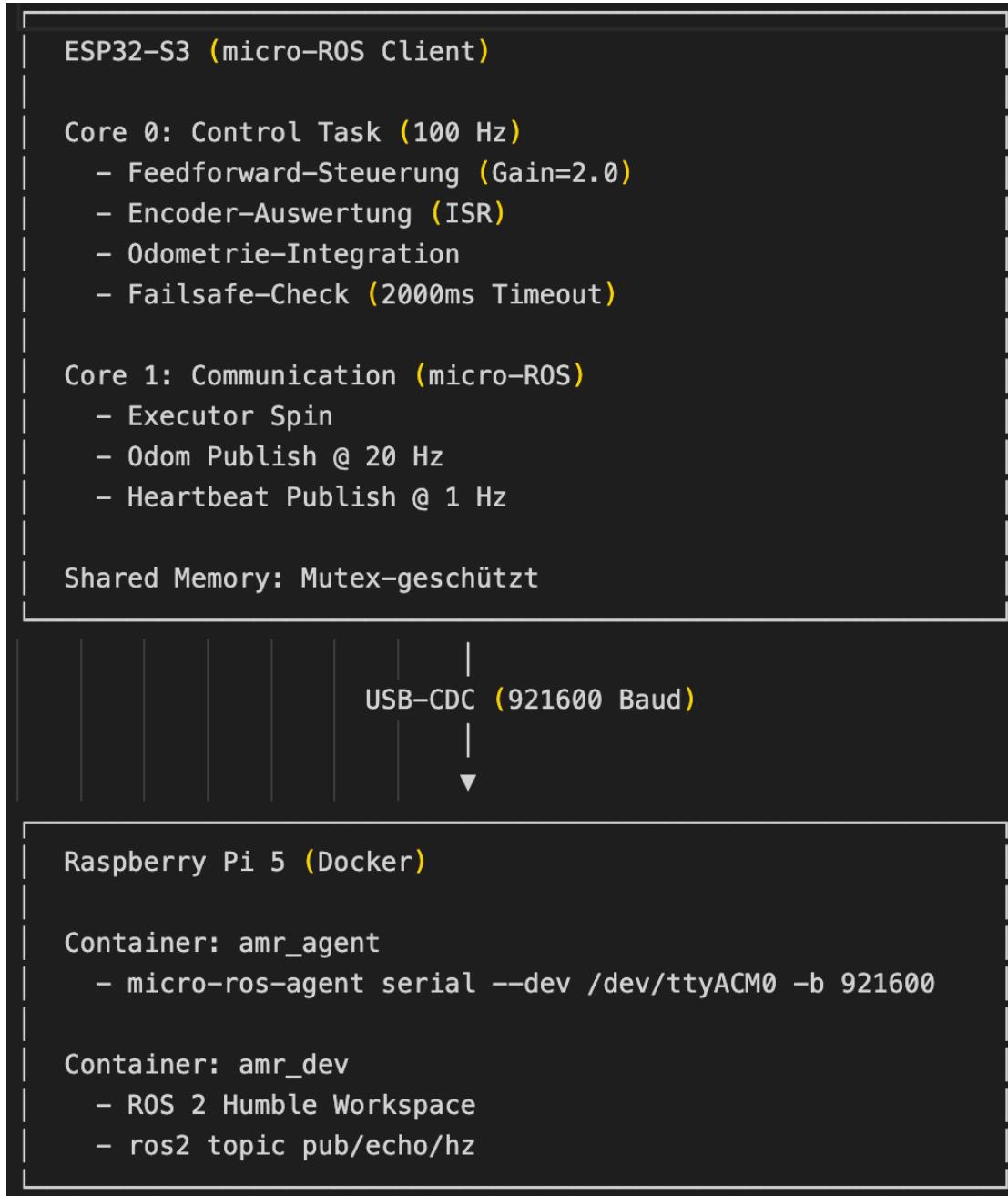


Abbildung 1: Phase-1-Architektur (micro-ROS über USB-Serial): Der ESP32-S3 arbeitet als micro-ROS-Client und führt auf Core 0 die Regel-/Odometrie-Schleife (100 Hz) inkl. Failsafe (2 s Timeout) aus, während Core 1 die micro-ROS-Kommunikation bedient. Über USB-CDC bei 921600baud ist der Raspberry Pi 5 (Docker) mit dem `micro-ros-agent` verbunden; im ROS 2-Workspace werden Topics wie `/cmd_vel` (Motorsteuerung) und `/odom_raw` (Pose2D) getestet/überwacht.

1.4 Topics (verifiziert)

Topic	Typ	Richtung	Funktion
/cmd_vel	geometry_msgs/Twist	Sub	Geschwindigkeitsbefehle
/odom_raw	geometry_msgs/Pose2D	Pub	Odometrie (x, y, θ)
/esp32/heartbeat	std_msgs/Int32	Pub	Lebenszeichen
/esp32/led_cmd	std_msgs/Bool	Sub	LED-Steuerung

1.5 Hardware (Phase 1)

Komponente	Spezifikation	Rolle
Seeed XIAO ESP32-S3	Dual-Core Xtensa LX7, USB-CDC	micro-ROS Client + Control
Cytron MDD3A	Dual-PWM, 4 V bis 16 V	Motortreiber
JGA25-370 (2×)	12 V DC + Hall-Encoder	Antrieb + Odometrie
Raspberry Pi 5	ROS 2 Humble (Docker)	micro-ROS Agent + Host

1.5.1 Pin-Mapping

Funktion	Pin	Typ	Hinweis
Motor Left A	D0	PWM	→ PWM_CH 1 (getauscht)
Motor Left B	D1	PWM	→ PWM_CH 0 (getauscht)
Motor Right A	D2	PWM	→ PWM_CH 3 (getauscht)
Motor Right B	D3	PWM	→ PWM_CH 2 (getauscht)
Encoder Left A	D6	IRQ	A-only
Encoder Right A	D7	IRQ	A-only
LED/MOSFET	D10	GPIO	Status

Hinweis

Die PWM-Kanäle wurden getauscht (A↔B), um die korrekte Fahrtrichtung zu erreichen.



1.6 Firmware – Parameter (v3.2.0)

1.6.1 config.h

Parameter	Wert	Beschreibung
L0OP_RATE_HZ	100 Hz	Control-Zyklus (10 ms)
ODOM_PUBLISH_HZ	20 Hz	Odom Publish (50 ms)
FAILSAFE_TIMEOUT_MS	2000 ms	Heartbeat-Timeout
MOTOR_PWM_FREQ	20 kHz	PWM-Frequenz (unhörbar)
MOTOR_PWM_BITS	8	Auflösung (0–255)
PWM_DEADZONE	35	Mindest-PWM
WHEEL_DIAMETER	0,065 m	Raddurchmesser
WHEEL_BASE	0,178 m	Spurbreite

1.6.2 PWM-Kanäle (getauscht für korrekte Richtung)

```

1 #define PWM_CH_LEFT_A 1 // war 0
2 #define PWM_CH_LEFT_B 0 // war 1
3 #define PWM_CH_RIGHT_A 3 // war 2
4 #define PWM_CH_RIGHT_B 2 // war 3

```

Listing 1: PWM-Channel Mapping (A/B getauscht)

1.6.3 Regelung (Open-Loop mit Feedforward)

Parameter	Wert	Beschreibung
PID_KP	0,0	Deaktiviert
PID_KI	0,0	Deaktiviert
PID_KD	0,0	Deaktiviert
feedforward_gain	2,0	Direkte Ansteuerung

Hinweis

PID wurde deaktiviert, da die Encoder-Polarität invertiert ist. Feedforward ermöglicht stabile Open-Loop-Steuerung. PID-Tuning ist sinnvoll ab Phase 4+, nachdem die Encoder-Richtungsheuristik validiert ist.

1.6.4 main.cpp – Feedforward-Berechnung

```
1 float feedforward_gain = 2.0f;
2 float pwm_l = feedforward_gain * set_v_l +
   pid_left.compute(set_v_l, v_enc_l, dt);
3 float pwm_r = feedforward_gain * set_v_r +
   pid_right.compute(set_v_r, v_enc_r, dt);
4
5 // Begrenzen auf PWM-Bereich
6 pwm_l = constrain(pwm_l, -1.0f, 1.0f);
7 pwm_r = constrain(pwm_r, -1.0f, 1.0f);
```

Listing 2: Feedforward + (optional) PID

1.7 Build/Flash/Monitor (PlatformIO)

1.7.1 Firmware kompilieren und flashen (Mac)

```
cd ~/daten/start/IoT/AMR/amr-platform/firmware
pio run -e seeed_xiao_esp32s3 -t upload
```

1.7.2 Serial Monitor (Debug)

```
pio device monitor -b 921600
```

1.8 Docker-Setup (Pi 5)

1.8.1 docker-compose.yml (Auszug)

```
services:
  microros_agent:
    image: microros/micro-ros-agent:humble
    container_name: amr_agent
    network_mode: host
    privileged: true
    restart: always
    command: serial --dev /dev/ttyACM0 -b 921600
```

```
devices:
  - /dev/ttyACM0:/dev/ttyACM0

amr_dev:
  build: .
  container_name: amr_base
  network_mode: host
  privileged: true
  volumes:
    - ../ros2_ws:/root/ros2_ws
  command: tail -f /dev/null
```

1.8.2 Container starten

```
cd ~/amr-platform/docker
docker compose up -d
docker compose ps
```

1.8.3 Agent-Logs prüfen

```
docker compose logs microros_agent --tail 10
```

Erwartete Ausgabe:

```
amr_agent | [timestamp] info | TermiosAgentLinux.cpp | init |
running... | fd: 3
```

1.9 Smoke-Tests

1.9.1 1) In Container gehen

```
docker compose exec amr_dev bash
source /opt/ros/humble/setup.bash
```

1.9.2 2) Topics prüfen

```
ros2 topic list
```

Erwartung:

```
/cmd_vel  
/esp32/heartbeat  
/esp32/led_cmd  
/odom_raw  
/parameter_events  
/rosout
```

1.9.3 3) Motor-Tests (Räder aufbocken!)

Sicherheit

Vor Motor-Tests Räder aufbocken: der Roboter darf nicht unkontrolliert losfahren.

```
# Vorwärts  
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: 0.15}, angular: {z: 0.0}}" -r 10  
  
# Rückwärts (Ctrl+C, dann:)  
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: -0.15}, angular: {z: 0.0}}" -r 10  
  
# Drehen links  
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: 0.0}, angular: {z: 0.5}}" -r 10  
  
# Drehen rechts  
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: 0.0}, angular: {z: -0.5}}" -r 10
```

Listing 3: Motor-Tests via /cmd_vel

1.9.4 4) Failsafe-Test

1. Motor-Befehl senden (Räder drehen)
2. **Ctrl+C** drücken
3. 2s warten
4. **Erwartung:** Motoren stoppen automatisch

1.9.5 5) Odometrie prüfen

```
ros2 topic echo /odom_raw --once
```

1.10 Troubleshooting

Problem	Ursache	Lösung
Serial port not found	ESP32 nicht angeschlossen	USB-Kabel prüfen, <code>ls /dev/ttyACM*</code>
Topics fehlen	Agent nicht verbunden	Agent-Logs prüfen, ESP32 Reset
Räder drehen falsche Richtung	PWM-Kanäle falsch	A↔B tauschen in <code>config.h</code>
Motor reagiert nicht	Feedforward zu niedrig	<code>feedforward_gain</code> erhöhen
PID eskaliert	Encoder-Polarität invertiert	PID deaktivieren ($K_p = 0$)
Failsafe greift nicht	Timeout zu kurz	<code>FAILSAFE_TIMEOUT_MS</code> erhöhen

1.11 Bekannte Einschränkungen

1. **Open-Loop-Steuerung:** PID deaktiviert, keine Geschwindigkeitsregelung.
2. **Encoder A-only:** Richtung wird aus Soll-Geschwindigkeit abgeleitet.
3. **Odom-Rate:** effektiv $\approx 3\text{ Hz}$ bis 6 Hz durch Serial-Transport.

1.12 Nächste Schritte

Phase	Beschreibung	Status
Phase 1	micro-ROS ESP32-S3	abgeschlossen
Phase 2	Docker-Infrastruktur	vorhanden
Phase 3	RPLidar A1 Integration (/dev/ttyUSB0)	bereit
Phase 4	EKF Sensor Fusion	offen
Phase 5	SLAM (<code>slam_toolbox</code>)	offen
Phase 6	Nav2 autonome Navigation	offen

1.13 Changelog

Version	Datum	Änderungen
v1.0	2025-12-19	Initiale Dokumentation
v3.1.0	2025-12-20	PID aktiviert, Baudrate 921 600 baud
v3.2.0	2025-12-20	PWM-Kanäle getauscht, Feedforward (Gain=2,0), PID deaktiviert, alle Tests bestanden

2 Phase 1 – Befehlsreferenz

Zweck

Schnelle Befehls-Sammlung für Phase 1 (micro-ROS Agent im Docker auf dem Pi, ESP32 per USB-Serial). Arbeitsverzeichnis: `~/amr-platform/docker`.

2.1 Nach Pi Reboot

```
cd ~/amr-platform/docker
docker compose up -d
sleep 5
docker compose logs microros_agent --tail 5
```

Erwartung

running... | fd: 3

2.2 Nach ESP32 Reboot (USB umgesteckt)

```
cd ~/amr-platform/docker
docker compose restart microros_agent
sleep 5
docker compose logs microros_agent --tail 5
```

Erwartung

```
running... | fd: 3
```

2.3 Notfall-Stop (falls Räder drehen)

Sicherheit

Wenn sich der Rover unkontrolliert bewegt: *sofort* Stop senden. Bei Motor-Tests Räder aufbocken.

```
docker compose exec amr_dev bash -c "source
/opt/ros/humble/setup.bash && \
ros2 topic pub --once /cmd_vel geometry_msgs/msg/Twist
'{linear: {x: 0.0}, angular: {z: 0.0}}'"
```

2.4 Smoke-Tests

2.4.1 1. In Container gehen

```
docker compose exec amr_dev bash
source /opt/ros/humble/setup.bash
```

2.4.2 2. Topics prüfen

```
ros2 topic list
```

Erwartung

```
/cmd_vel  
/esp32/heartbeat  
/esp32/led_cmd  
/odom_raw  
/parameter_events  
/rosout
```

2.4.3 3. Heartbeat prüfen

```
ros2 topic echo /esp32/heartbeat
```

Erwartung

Counter inkrementiert ca. $1 \times /s$ (mit **Ctrl+C** beenden).

**2.4.4 4. Odometrie prüfen**

```
ros2 topic echo /odom_raw --once
```

Erwartung

Ausgabe enthält plausible x, y, theta Werte.

**2.4.5 5. Frequenzen messen**

```
ros2 topic hz /esp32/heartbeat
```

```
ros2 topic hz /odom_raw
```

Erwartung

Heartbeat $\approx 1 \text{ Hz}$, Odom $\approx 3 \text{ Hz}$ bis 6 Hz .



2.5 Motor-Tests

Sicherheit

Räder aufbocken (kein Bodenkontakt). Notfall-Stop bereithalten.



2.5.1 Vorwärts

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: 0.15}, angular: {z: 0.0}}" -r 10
```

2.5.2 Rückwärts

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: -0.15}, angular: {z: 0.0}}" -r 10
```

2.5.3 Drehen links

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: 0.0}, angular: {z: 0.5}}" -r 10
```

2.5.4 Drehen rechts

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: 0.0}, angular: {z: -0.5}}" -r 10
```

2.5.5 Manueller Stop

```
ros2 topic pub --once /cmd_vel geometry_msgs/msg/Twist \  
  "{linear: {x: 0.0}, angular: {z: 0.0}}"
```

Hinweis

Nach **Ctrl+C** stoppt der Failsafe die Motoren automatisch nach ca. 2 s.



2.6 Failsafe-Test

1. Motor-Befehl senden (Räder drehen).
2. **Ctrl+C** drücken.
3. 2 s warten.
4. **Erwartung:** Motoren stoppen automatisch.

2.7 Odom nach Fahrt prüfen

```
ros2 topic echo /odom_raw --once
```

Erwartung

Nach Vorwärtsfahrt: $x > 0$. Nach Drehung: **theta** ändert sich.



2.8 Checkliste Phase 1

Test	Erwartung	Status
Agent verbindet	fd: 3	<input type="checkbox"/>
Topics vorhanden	6 Topics	<input type="checkbox"/>
Heartbeat	$\approx 1 \text{ Hz}$	<input type="checkbox"/>
Odom publiziert	x, y, theta Werte	<input type="checkbox"/>
Vorwärts	Räder drehen vorwärts	<input type="checkbox"/>
Rückwärts	Räder drehen rückwärts	<input type="checkbox"/>
Drehen links	Roboter dreht links	<input type="checkbox"/>
Drehen rechts	Roboter dreht rechts	<input type="checkbox"/>
Failsafe	Stop nach 2 s	<input type="checkbox"/>

3 Phase 2: ROS 2 Humble auf Raspberry Pi 5 (Docker) + micro-ROS Agent

2.9 Troubleshooting

Problem	Lösung
Serial port not found	USB-Kabel prüfen, <code>ls /dev/ttyACM*</code>
Topics fehlen	<code>docker compose restart microros_agent</code>
Räder reagieren nicht	Feedforward-Gain prüfen (2.0)
Falsches Verhalten	USB ziehen, Firmware prüfen

3 Phase 2: ROS 2 Humble auf Raspberry Pi 5 (Docker) + micro-ROS Agent

Status & Version	1
<ul style="list-style-type: none">• Status: completed• Updated: 2025-12-20• Version: 2.0• Depends on: Phase 1 (micro-ROS auf ESP32-S3)• Next: Phase 3 (RPLidar A1)	1

3.1 Zielbild & Definition of Done

3.1.1 Zielbild

- Raspberry Pi 5 (Raspberry Pi OS **64-bit**) betreibt **ROS 2 Humble in Docker**.
- micro-ROS Agent läuft reproduzierbar (Container) und verbindet sich über **USB-Serial** zum ESP32-S3.
- Host-ROS kann:
 - `/cmd_vel` publizieren → Motor reagiert
 - `/odom_raw` empfangen → Werte plausibel
 - `/esp32/heartbeat` empfangen → Agent-Verbindung verifiziert

3.1.2 DoD (verifiziert 2025-12-20)

- ✓ `docker compose up` startet Container ohne manuelle Nacharbeit.

3 Phase 2: ROS 2 Humble auf Raspberry Pi 5 (Docker) + micro-ROS Agentenwerk

- ✓ Agent verbindet stabil über `/dev/ttyACM0` mit 921600 Bd.
- ✓ ROS Smoke-Tests sind grün:
 - ✓ `ros2 topic list` zeigt `/cmd_vel`, `/odom_raw`, `/esp32/heartbeat`, `/esp32/led_cmd`
 - ✓ `ros2 topic pub /cmd_vel ...` bewegt den Rover
 - ✓ `ros2 topic echo /odom_raw` liefert kontinuierliche Werte
- ✓ Failsafe stoppt Motoren nach 2 s Timeout.

3.2 Docker-Images (Regel)

Service / Container	Image	Funktion
<code>microros_agent</code> / <code>amr_agent</code>	<code>microros/micro-ros-agent:humble</code>	Serial Agent
<code>amr_dev</code> / <code>amr_base</code>	Custom (ROS 2 Humble)	Workspace

Warum Humble statt Jazzy:

- micro-ROS Agent für Humble stabiler auf `arm64`
- Kompatibilität mit bestehenden Packages (Nav2, `slam_toolbox`)

3.3 Host-Voraussetzungen

3.3.1 System

- Raspberry Pi 5, Raspberry Pi OS **64-bit** (Bookworm)
- Docker Engine + Docker Compose Plugin

3.3.2 USB-Serial prüfen

```
ls -l /dev/ttyACM*
# Erwartung: /dev/ttyACM0 (ESP32-S3)

ls -l /dev/ttyUSB*
# Erwartung: /dev/ttyUSB0 (RPLidar A1)
```

3.4 Repo-Struktur

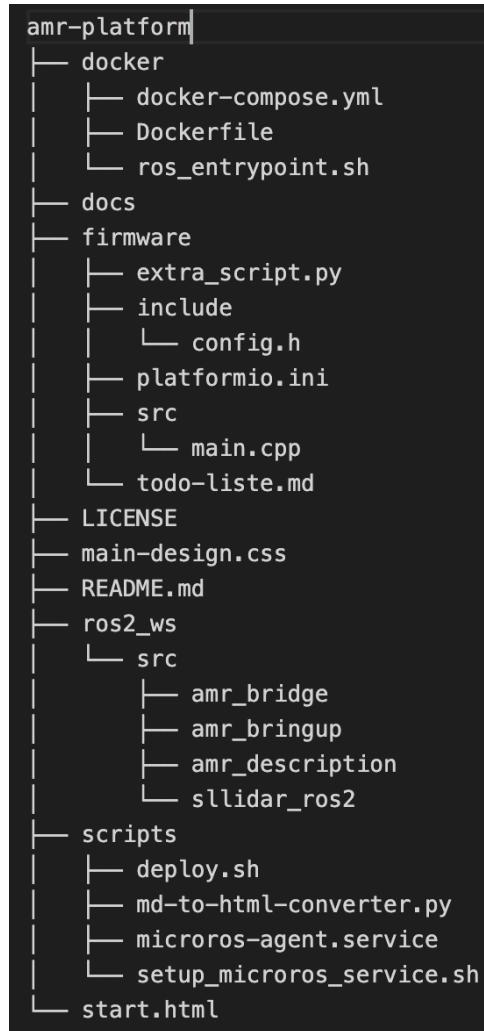


Abbildung 2: Projektstruktur des Repositories `amr-platform`: `firmware/` enthält die ESP32-S3-Firmware (PlatformIO mit `src/main.cpp` und `include/config.h`), `ros2_ws/src/` den ROS 2-Workspace mit Paketen für Bridge/Bringup/Description sowie `sllidar_ros2`. Die Laufzeitumgebung ist in `docker/` (Dockerfile, Compose, Entrypoint) gekapselt. Automatisierung und Betrieb liegen in `scripts/` (Deployment, micro-ROS-Agent-Service, Dokumentations-Converter). `docs/` bündelt Projektdokumentation; `README.md`, `LICENSE`, `start.html` und `main-design.css` bilden Einstieg und Styling der Doku.

3.5 Docker Compose

3.5.1 docker-compose.yml

```

services:
  microros_agent:

```

3 Phase 2: ROS 2 Humble auf Raspberry Pi 5 (Docker) + micro-ROS Agentenwerk

```
image: microros/micro-ros-agent:humble
container_name: amr_agent
network_mode: host
privileged: true
restart: always
command: serial --dev /dev/ttyACM0 -b 921600
devices:
  - /dev/ttyACM0:/dev/ttyACM0

amr_dev:
  build: .
  container_name: amr_base
  network_mode: host
  privileged: true
  volumes:
    - ../ros2_ws:/root/ros2_ws
  command: tail -f /dev/null
```

3.5.2 Dockerfile

```
FROM ros:humble-ros-base

RUN apt-get update && apt-get install -y \
    python3-colcon-common-extensions \
    ros-humble-tf2-tools \
    ros-humble-xacro \
    ros-humble-rviz2 \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /root/ros2_ws
```

3.6 Befehle

3.6.1 Nach Pi Reboot

```
cd ~/amr-platform/docker
docker compose up -d
sleep 5
```

3 Phase 2: ROS 2 Humble auf Raspberry Pi 5 (Docker) + microros Agentenwerk

```
docker compose logs microros_agent --tail 5
```

Erwartung

```
running... | fd: 3
```

3.6.2 Nach ESP32 Reboot

```
docker compose restart microros_agent  
sleep 5  
docker compose logs microros_agent --tail 5
```

3.6.3 In Container gehen

```
docker compose exec amr_dev bash  
source /opt/ros/humble/setup.bash
```

3.7 Smoke-Tests

3.7.1 Topics prüfen

```
ros2 topic list
```

Erwartung

```
/cmd_vel  
/esp32/heartbeat  
/esp32/led_cmd  
/odom_raw  
/parameter_events  
/rosout
```

3.7.2 Heartbeat

```
ros2 topic echo /esp32/heartbeat
```

3 Phase 2: ROS 2 Humble auf Raspberry Pi 5 (Docker) + micrOROS Agentenwerk

Erwartung

Counter steigt ca. 1 Hz.



3.7.3 Odometrie

```
ros2 topic echo /odom_raw --once
```

Erwartung

Ausgabe enthält plausible x, y, theta.



3.7.4 Motor-Test (Räder aufbocken!)

Sicherheit

Motor-Tests nur mit aufgebockten Rädern (kein Bodenkontakt). !

```
# Vorwärts
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \
    "{linear: {x: 0.15}, angular: {z: 0.0}}" -r 10

# Ctrl+C -> Failsafe stoppt nach 2s
```

3.8 Troubleshooting

Problem	Ursache	Lösung
Agent sieht ESP32 nicht	Device-Pfad falsch	ls /dev/ttyACM* prüfen
Topics fehlen	Agent nicht verbunden	docker compose restart microros_agent
ROS sieht keine Topics	Domain-ID Mismatch	network_mode: host nutzen
Motor reagiert nicht	Failsafe greift	Timeout prüfen (2000 ms)

3.9 Verifizierte Konfiguration

Parameter	Wert
ROS-Version	Humble
Agent-Image	<code>microros/micro-ros-agent:humble</code>
Baudrate	921600 Bd
Device	<code>/dev/ttyACM0</code>
Services/Container	<code>microros_agent (amr_agent), amr_dev (amr_base)</code>
Network	host

3.10 Changelog

Version	Datum	Änderungen
v2.0	2025-12-20	Humble statt Jazzy, 921600 Bd, Container-Namen aktualisiert, Status: abgeschlossen
v1.0	2025-12-19	Initiale Jazzy-Version (überholt)

4 Phase 3: RPLidar A1 Integration (+ /scan)

Status & Version	
<ul style="list-style-type: none"> Status: completed Updated: 2025-12-20 Version: 3.0 Depends on: Phase 1 (micro-ROS ESP32-S3), Phase 2 (Docker ROS 2 Humble) Next: Phase 4 (URDF + TF + EKF) 	1

4.1 Zielbild & Definition of Done

4.1.1 Zielbild

- RPLidar A1 läuft am Raspberry Pi 5 (Docker/ROS 2 Humble) als Laser-Treiber.
- Topic `/scan` (`sensor_msgs/msg/LaserScan`) ist stabil und liefert plausible Werte.
- `frame_id` ist `laser` (TF-Anbindung an `base_link` folgt in Phase 4).

4.1.2 DoD (verifiziert 2025-12-20)

- ✓ `ros2 topic hz /scan` zeigt stabile Frequenz $\approx 7,6 \text{ Hz}$
- ✓ `ros2 topic echo /scan -once` liefert plausible Werte
- ✓ `frame_id: laser`
- ✓ Scan-Range: $-\pi$ bis $+\pi$, 0,05 m bis 12,0 m

4.2 Hardware-Info (verifiziert)

4.2.1 Sensor

Parameter	Wert
Modell	RPLidar A1
S/N	74A5FA89C7E19EC8BCE499F0FF725670
Firmware	1.29
Hardware Rev	7
Scan Mode	Sensitivity
Sample Rate	8 kHz

4.2.2 Gemessene Werte

Parameter	Wert
Frequenz	$\approx 7,6 \text{ Hz}$
<code>range_min</code>	0,05 m
<code>range_max</code>	12,0 m
<code>angle_min</code>	$-3,14 \text{ rad}$
<code>angle_max</code>	$+3,14 \text{ rad}$

4.2.3 Anschluss

Device	Port	Hinweis
RPLidar A1	/dev/ttyUSB0	USB-Adapter: cp210x

4.3 Docker-Integration

Wichtig

Kein separater `rplidar`-Container. Das erzeugt Port-Konflikte. RPLidar wird im `amr_dev`-Container gestartet.

4.3.1 docker-compose.yml (relevanter Auszug)

```
services:  
  microros_agent:  
    image: microros/micro-ros-agent:humble  
    container_name: amr_agent  
    network_mode: host  
    privileged: true  
    restart: always  
    command: serial --dev /dev/ttyACM0 -b 921600  
    devices:  
      - /dev/ttyACM0:/dev/ttyACM0  
  
  amr_dev:  
    build: .  
    container_name: amr_base  
    network_mode: host  
    privileged: true  
    stdin_open: true  
    tty: true  
    volumes:  
      - ../ros2_ws:/root/ros2_ws  
    devices:  
      - /dev/ttyUSB0:/dev/ttyUSB0  
    command: tail -f /dev/null
```

4.4 Installation

4.4.1 Repository klonen (im Container)

```
cd ~/amr-platform/docker  
docker compose exec amr_dev bash
```

```
cd /root/ros2_ws/src  
git clone https://github.com/Slamtec/sllidar_ros2.git
```

4.4.2 Build

```
cd /root/ros2_ws  
source /opt/ros/humble/setup.bash  
colcon build --packages-select sllidar_ros2  
source install/setup.bash
```

4.5 RPLidar starten & Smoke-Tests

4.5.1 Terminal 1: Lidar-Node starten

```
cd ~/amr-platform/docker  
docker compose exec amr_dev bash  
source /opt/ros/humble/setup.bash  
source /root/ros2_ws/install/setup.bash  
ros2 launch sllidar_ros2 sllidar_a1_launch.py  
    serial_port:=/dev/ttyUSB0
```

Erwartete Log-Infos (Auszug)

```
SLLidar S/N: 74A5FA89C7E19EC8BCE499F0FF725670  
Firmware Ver: 1.29  
Hardware Rev: 7  
health status : OK  
current scan mode: Sensitivity, sample rate: 8 Khz,  
max_distance: 12.0 m
```

4.5.2 Terminal 2: Smoke-Tests

```
cd ~/amr-platform/docker  
docker compose exec amr_dev bash  
source /opt/ros/humble/setup.bash
```

```
ros2 topic list  
ros2 topic hz /scan  
ros2 topic echo /scan --once
```

4.5.3 Erwartete Topics

Topic-Liste

```
/cmd_vel  
/esp32/heartbeat  
/esp32/led_cmd  
/odom_raw  
/scan  
/parameter_events  
/rosout
```

4.5.4 Prüfpunkte für /scan

Feld	Erwartung
header.frame_id	laser
angle_min/angle_max	-3.14 rad bis +3.14 rad
range_min/range_max	0,05 m bis 12,0 m
ranges []	Werte 0,05 m bis 12,0 m, inf bei keiner Reflexion
Frequenz	≈ 7,6 Hz

4.6 Troubleshooting

4.6.1 „Operation timeout“

Typisch: Port ist von anderem Prozess belegt.

```
# Auf Pi Host prüfen:  
sudo fuser /dev/ttyUSB0  
  
# Falls belegt:  
sudo kill <PID>
```

4.6.2 /scan nicht vorhanden

- Lidar-Node muss in Terminal 1 weiterlaufen.
- Wenn mit Ctrl+C gestoppt: kein /scan.

4.6.3 Device nicht vorhanden

```
ls -l /dev/ttyUSB*
dmesg | grep -i usb | tail -10
```

4.6.4 Container sieht Device nicht

```
# docker-compose.yml:
devices:
- /dev/ttyUSB0:/dev/ttyUSB0
```

4.6.5 Baudrate falsch

Standard ist 115 200 baud. Falls Timeout:

```
ros2 launch sllidar_ros2 sllidar_a1_launch.py \
serial_port:=/dev/ttyUSB0 serial_baudrate:=256000
```

4.7 Statischer TF (temporär für RViz2)

Bis Phase 4 (URDF/TF/EKF) fertig ist:

```
ros2 run tf2_ros static_transform_publisher \
0.12 0.0 0.15 0 0 0 base_link laser
```

4.8 Nächste Schritte (Phase 4)

1. URDF erstellen inkl. Transform `base_link` → `laser`
2. `robot_state_publisher` für TF-Baum

3. `odom_converter.py` Bridge Node (Pose2D → Odometry/TF)
4. Optional: EKF (`robot_localization`)

4.9 Changelog

Version	Datum	Änderungen
v3.0	2025-12-20	Status: completed; kein separater RPLidar-Container; Hardware-Info verifiziert; Troubleshooting erweitert
v2.0	2025-12-20	Humble statt Jazzy
v1.0	2025-12-19	Initiale Version

5 Phase 3: RPLidar A1 – Befehlsreferenz

Status

- Status: **abgeschlossen**
- Stand: **2025-12-20**

I

5.1 Quick Start

5.1.1 1) Container starten (nach Pi Reboot)

```
cd ~/amr-platform/docker
docker compose up -d
docker compose ps
```

5.1.2 2) RPLidar starten (Terminal 1)

```
docker compose exec amr_dev bash
source /opt/ros/humble/setup.bash
source /root/ros2_ws/install/setup.bash
ros2 launch sllidar_ros2 sllidar_a1_launch.py
    serial_port:=/dev/ttyUSB0
```

5.1.3 3) Smoke-Tests (Terminal 2)

```
docker compose exec amr_dev bash
source /opt/ros/humble/setup.bash

# Topics prüfen
ros2 topic list

# Frequenz prüfen (~7.6 Hz)
ros2 topic hz /scan

# Daten prüfen
ros2 topic echo /scan --once
```

5.2 Erwartete Topics

Topics
/cmd_vel
/esp32/heartbeat
/esp32/led_cmd
/odom_raw
/scan
/parameter_events
/rosout

5.3 Scan-Daten Referenz

Feld	Erwartung
frame_id	laser
angle_min	-3.14 rad
angle_max	+3.14 rad
range_min	0.05 m
range_max	12.0 m
Frequenz	≈ 7.6 Hz

5.4 Troubleshooting

5.4.1 Fehler: „Operation timeout“

```
# Port blockiert? Prüfen:  
sudo fuser /dev/ttyUSB0  
  
# Falls belegt, Prozesse beenden:  
sudo kill <PID>
```

5.4.2 Device nicht vorhanden

```
ls -l /dev/ttyUSB*  
dmesg | grep -i usb | tail -10
```

5.4.3 Container sieht Device nicht

docker-compose.yml prüfen:

```
devices:  
  - /dev/ttyUSB0:/dev/ttyUSB0
```

5.5 Hardware-Info (RPLidar A1)

Parameter	Wert
Modell	RPLidar A1
S/N	74A5FA89C7E19EC8BCE499F0FF725670
Firmware	1.29
Hardware Rev	7
Scan Mode	Sensitivity
Sample Rate	8 kHz
Scan Frequency	≈ 7.6 Hz

6 AMR Implementierungsplan

Meta	1
<ul style="list-style-type: none">• Version: 2.0• Stand: 2025-12-20• Firmware: v3.2.0	

6.1 Grundprinzip: Vertikale Scheiben statt horizontaler Schichten

Ein häufiger Fehler bei Robotik-Projekten ist ein *Big-Bang-Integrationsversuch*: zuerst komplette Hardware, dann komplette Firmware, dann komplette Treiber — und beim ersten Integrationstest ist alles gleichzeitig neu. Fehlersuche wird dann unverhältnismäßig teuer.

Ansatz: System in *vertikale Scheiben* schneiden. Jede Phase liefert ein lauffähiges, testbares Teilsystem, bevor die nächste Komplexitätsstufe hinzukommt.

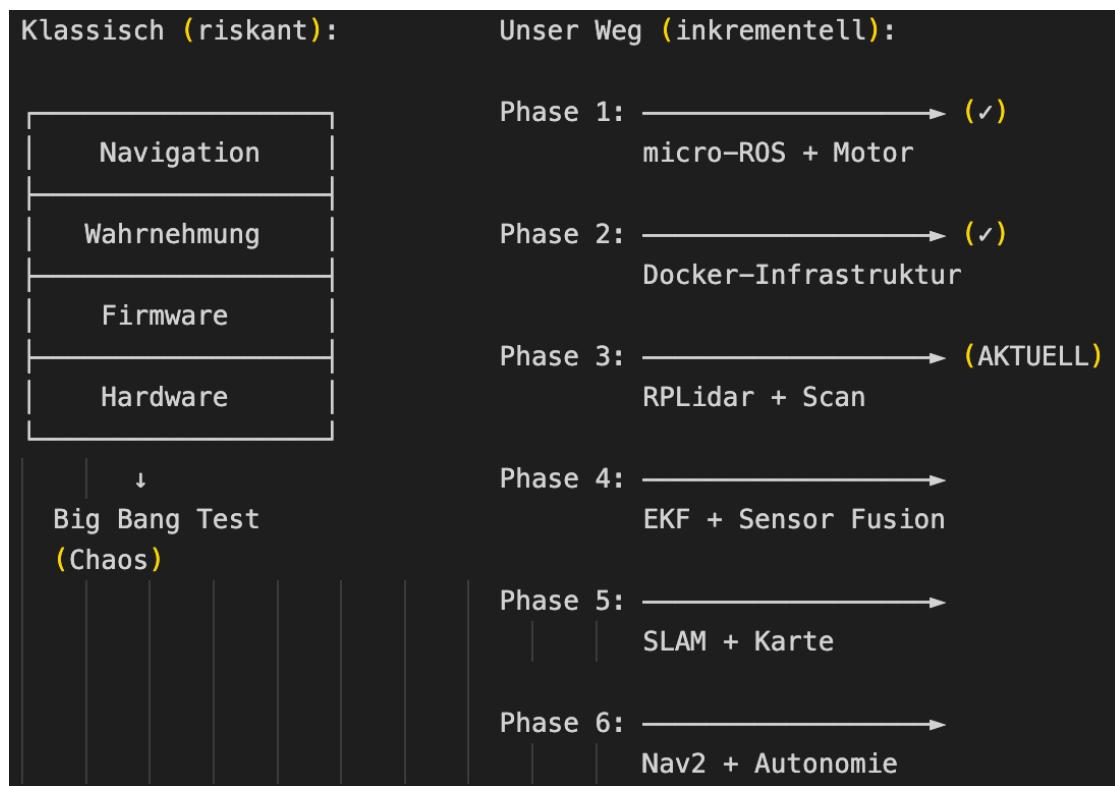


Abbildung 3: Grundprinzip „vertikale Scheiben“: Statt Hardware, Firmware, Wahrnehmung und Navigation separat „fertig“ zu bauen und erst am Ende im Big-Bang-Test zu integrieren, wird das System phasenweise als lauffähiger End-to-End-Strang erweitert. Jede Phase liefert ein testbares Teilsystem (z. B. Phase 1: micro-ROS + Motorsteuerung, Phase 2: Docker / ROS-Tooling, Phase 3: LiDAR-Scan), bevor mit EKF / Sensorfusion, SLAM und Nav2 die nächste Komplexitätsstufe hinzukommt.

6.2 Phasen-Übersicht

Phase	Beschreibung	Status
Phase 1	micro-ROS auf ESP32-S3 (USB-Serial)	✓ Abgeschlossen
Phase 2	Docker-Infrastruktur	✓ Vorhanden
Phase 3	RPLidar A1 Integration	AKTUELL
Phase 4	EKF Sensor Fusion	□
Phase 5	SLAM (<code>slam_toolbox</code>)	□
Phase 6	Nav2 Autonome Navigation	□

6.3 Phase 1: micro-ROS auf ESP32-S3

Ziel: Native ROS 2 Kommunikation über USB-Serial mit Dual-Core FreeRTOS Architektur.

Status: ✓ Abgeschlossen (2025-12-20) | Firmware **v3.2.0**

6.3.1 Architektur (Kurz)

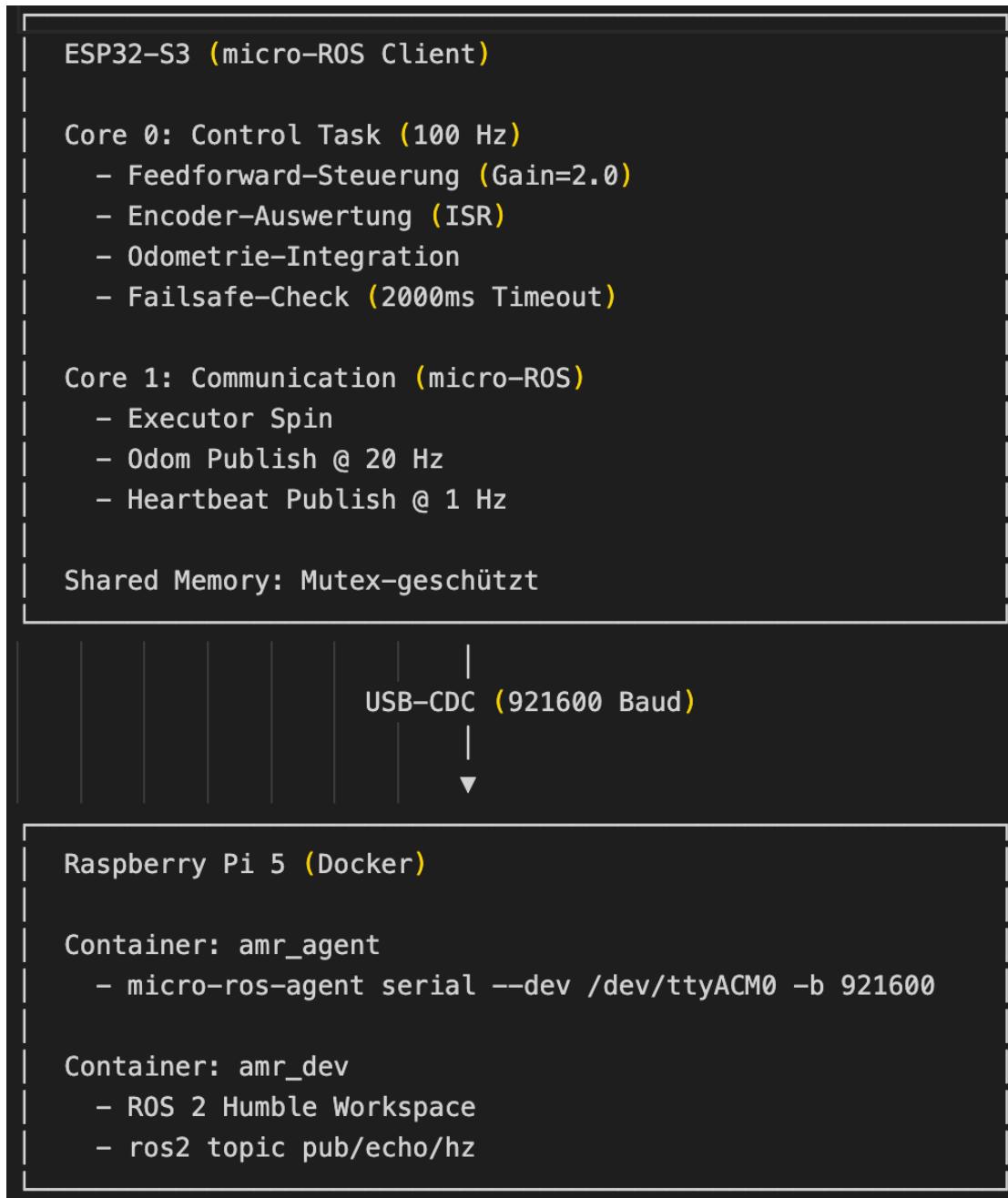


Abbildung 4: Phase-1-Architektur (micro-ROS über USB-Serial): Der ESP32-S3 arbeitet als micro-ROS-Client und führt auf Core 0 die Regel-/Odometrie-Schleife (100 Hz) inkl. Failsafe (2 s Timeout) aus, während Core 1 die micro-ROS-Kommunikation bedient. Über USB-CDC bei 921600baud ist der Raspberry Pi 5 (Docker) mit dem `micro-ros-agent` verbunden; im ROS 2-Workspace werden Topics wie `/cmd_vel` (Motorsteuerung) und `/odom_raw` (Pose2D) getestet/überwacht.

6.3.2 Topics

Topic	Typ	Richtung	Beschreibung
/cmd_vel	geometry_msgs/Twist	Sub	Geschwindigkeitsbefehle
/odom_raw	geometry_msgs/Pose2D	Pub	Odometrie (x, y, θ)
/esp32/heartbeat	std_msgs/Int32	Pub	Lebenszeichen
/esp32/led_cmd	std_msgs/Bool	Sub	LED-Steuerung

6.3.3 Konfiguration (Kernwerte)

Parameter	Wert
Baudrate	921600 Bd
Feedforward Gain	2.0
PID	deaktiviert ($K_p = 0$)
Failsafe Timeout	2000 ms
PWM-Kanäle	getauscht (A↔B)

6.3.4 Testergebnisse

Test	Status
Vorwärts	✓
Rückwärts	✓
Drehen links	✓
Drehen rechts	✓
Failsafe (2 s)	✓
Odom plausibel	✓

6.3.5 Cytron MDD3A – Dual-PWM (kritisch)

Kritisch

Der MDD3A verwendet **keinen** DIR-Pin, sondern **zwei PWM-Signale pro Motor**.

M1A (PWM)	M1B (PWM)	Ergebnis
200	0	Vorwärts
0	200	Rückwärts
0	0	Coast (Auslaufen)

6.4 Phase 2: Docker-Infrastruktur

Ziel: Container-basierte ROS 2 Umgebung für einfaches Deployment.

Status: ✓ Vorhanden

6.4.1 Container

Container	Image	Funktion
amr_agent	microros/micro-ros-agent:humble	Serial Agent
amr_dev	Custom (ROS 2 Humble)	Workspace

6.4.2 docker-compose.yml (Auszug)

```

services:
  microros_agent:
    image: microros/micro-ros-agent:humble
    container_name: amr_agent
    network_mode: host
    privileged: true
    restart: always
    command: serial --dev /dev/ttyACM0 -b 921600
    devices:
      - /dev/ttyACM0:/dev/ttyACM0

  amr_dev:
    build: .
    container_name: amr_base
    network_mode: host
    privileged: true
    volumes:
      - ../ros2_ws:/root/ros2_ws
    command: tail -f /dev/null

```

6.4.3 Quick Start

```
cd ~/amr-platform/docker
docker compose up -d
docker compose exec amr_dev bash
source /opt/ros/humble/setup.bash
ros2 topic list
```

6.5 Phase 3: RPLidar A1 Integration

Ziel: 360° Laserscan für Umgebungswahrnehmung.

Status: aktuell (Port /dev/ttyUSB0 erkannt)

6.5.1 Hardware

Komponente	Port	Status
RPLidar A1	/dev/ttyUSB0	✓ erkannt

6.5.2 Aufgaben

- rplidar_ros Package installieren
- Launch-File erstellen
- /scan verifizieren
- TF: laser → base_link
- RViz2 Visualisierung

6.5.3 Geplante Topics

Topic	Typ	Frequenz
/scan	sensor_msgs/LaserScan	5 Hz bis 10 Hz

6.5.4 Launch (geplant)

```
ros2 launch rplidar_ros rplidar_a1_launch.py
```

Meilenstein Phase 3: /scan publiziert, Daten in RViz2 sichtbar.

6.6 Phase 4: EKF Sensor Fusion

Ziel: Robuste Odometrie durch Fusion von Encoder-Daten (später + IMU).

6.6.1 Aufgaben

- robot_localization Package
- EKF Node konfigurieren
- /odom_raw → /odometry/filtered
- TF: odom → base_link
- Optional: IMU Integration (MPU6050)

6.6.2 Geplante Topics

Topic	Typ	Quelle
/odometry/filtered	nav_msgs/Odometry	EKF
/tf	tf2_msgs/TFMessage	EKF

Meilenstein Phase 4: TF-Baum korrekt, gefilterte Odometrie stabil.

6.7 Phase 5: SLAM (slam_toolbox)

Ziel: Der Roboter baut eine Karte seiner Umgebung.

6.7.1 Aufgaben

- slam_toolbox konfigurieren
- Online Async SLAM
- Testraum kartieren

- Karte speichern (PGM + YAML)

6.7.2 Launch

```
ros2 launch slam_toolbox online_async_launch.py  
params_file:=slam_params.yaml
```

Meilenstein Phase 5: Eine speicherbare Karte des Testraums existiert.

6.8 Phase 6: Nav2 Autonome Navigation

Ziel: Ziel auf Karte setzen, Roboter fährt autonom hin.

6.8.1 Nav2 Stack (Überblick)

Komponente	Funktion
AMCL	Lokalisierung auf bekannter Karte
Planner Server	Globaler Pfad (A* / Dijkstra)
Controller Server	Lokale Hindernisvermeidung
Costmap	Hinderniskarte aus Sensordaten
BT Navigator	Verhaltenssteuerung

Meilenstein Phase 6: Roboter navigiert autonom, weicht Hindernissen aus.

6.9 Zukünftige Erweiterungen (optional)

6.9.1 Kamera & AI

- IMX296 Global Shutter Kamera
- YOLOv8 auf Hailo-8L
- Personen-Erkennung → Stopp-Verhalten

6.9.2 PID-Regelung

Aktuell: Feedforward (Open-Loop). Für präzisere Regelung:

- Encoder-Polarität korrigieren (Quadratur-Encoder oder Richtungs-Heuristik verbessern)
- PID aktivieren (Beispielwerte aus früheren Tests: $K_p = 13.0$, $K_i = 5.0$, $K_d = 0.01$)

6.10 Hardware-Übersicht

Komponente	Spezifikation	Status
Seeed XIAO ESP32-S3	Dual-Core, USB-CDC	aktiv
Cytron MDD3A	Dual-PWM, 4 V bis 16 V	aktiv
JGA25-370 (2×)	12 V DC + Encoder	aktiv
Raspberry Pi 5	8 GB, ROS 2 Humble	aktiv
RPLidar A1	360° 2D Lidar	erkannt
Hailo-8L	AI Accelerator	später
IMX296	Global Shutter	später
MPU6050	IMU (I ² C)	später

6.11 Risikomatrix

Risiko	Wahrscheinlichkeit	Impact	Status
micro-ROS inkompatibel	früher hoch	hoch	✓ gelöst
MDD3A-Ansteuerung	früher hoch	hoch	✓ Dual-PWM geklärt
PID-Eskalation	früher mittel	mittel	✓ Feedforward
Motor-Richtung falsch	früher mittel	mittel	✓ PWM getauscht
Failsafe greift zu früh	früher mittel	niedrig	✓ 2000 ms
RPLidar-Treiber	niedrig	mittel	Phase 3
Nav2-Tuning aufwändig	hoch	mittel	Zeit einplanen

6.12 Zeitplan (8 Wochen, grob)

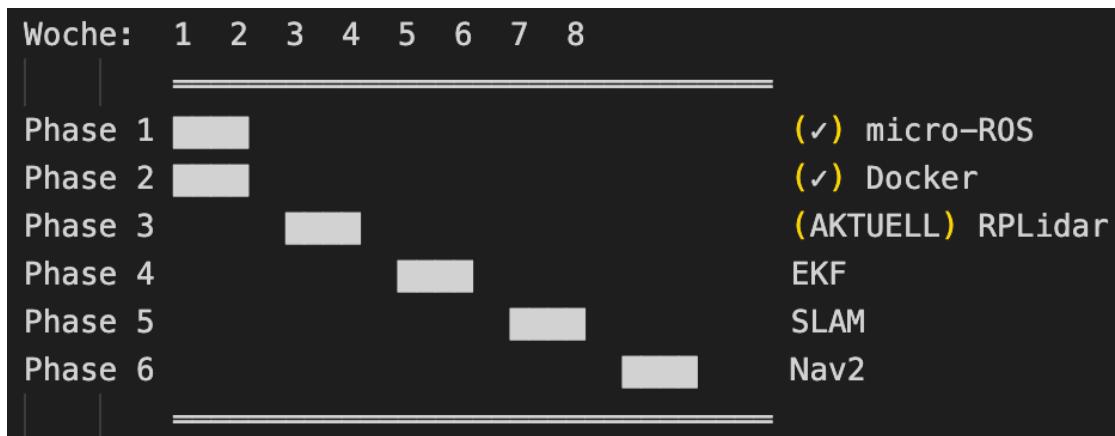


Abbildung 5: Grobzeitplan über 8 Wochen: Die Roadmap ist in aufeinander aufbauende Phasen mit je ca. 2 Wochen Fokuszeit gegliedert. Phase 1 (micro-ROS) und Phase 2 (Docker / ROS-Tooling) sind abgeschlossen; Phase 3 (RPLidar: Scan-Erfassung und Stabilisierung) ist aktuell. Danach folgen Phase 4 (EKF / Sensorfusion), Phase 5 (SLAM: Karte + Lokalisierung) und Phase 6 (Nav2: Pfadplanung und autonome Navigation).

6.13 Checkliste pro Phase

Jede Phase ist erst abgeschlossen, wenn:

- ✓ die definierten Tests bestanden sind
- ✓ der Code committet und dokumentiert ist
- ✓ die Konfigurationsdateien versioniert sind
- ✓ ein kurzes Protokoll die Ergebnisse festhält
- ✓ der nächste Schritt klar ist

Phase 1: ✓ alle Punkte erfüllt (2025-12-20)

Phase 2: ✓ alle Punkte erfüllt

6.14 Changelog

6.14.1 v2.0 (2025-12-20)

- Phase 1: micro-ROS statt Serial-Bridge
- Firmware: v3.2.0 mit Feedforward

- Architektur: Dual-Core FreeRTOS
- Docker: Container-basiertes Deployment
- Phasen: reorganisiert (6 statt 7)

6.14.2 v1.3 (2025-12-12)

- Phase 2 (Odometrie + PID) abgeschlossen
- Serial-Bridge Architektur (Legacy)

Merksatz

Lieber weniger Features, die funktionieren, als viele Features, die zusammen craschen.



7 Entwicklerdokumentation: AMR Low-Level Controller

Meta

1

- Version: 3.2.0
- Stand: 20.12.2025
- Status: Phase 1 abgeschlossen

7.1 Architektur-Design

Das System folgt einer **Hybrid-Echtzeit-Architektur**: zeitkritische Regelung und nicht-zeitkritische Kommunikation werden strikt getrennt durch Dual-Core-Nutzung des ESP32-S3.

7.1.1 Dual-Core Aufteilung (ESP32-S3)

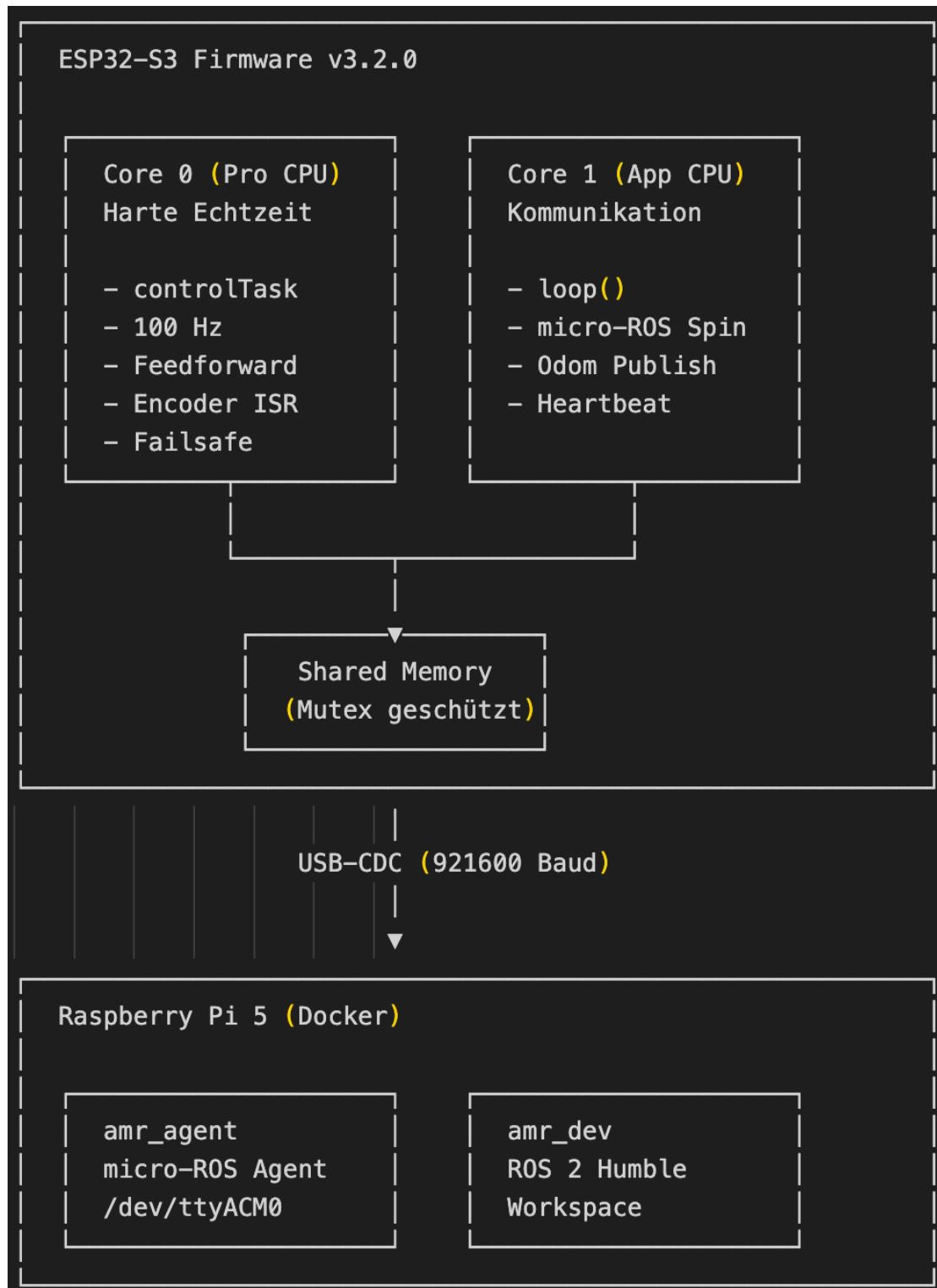


Abbildung 6: Dual-Core-Aufteilung auf dem ESP32-S3 (Hybrid-Echtzeit-Architektur): Core 0 übernimmt die zeitkritische Regel-/Odometrie-Schleife (100 Hz) inklusive Encoder-ISR, Feedforward und deterministischem Failsafe. Core 1 ist für nicht-zeitkritische Kommunikation reserviert (micro-ROS Spin/Executor, periodisches Publish von Odometrie und Heartbeat). Beide Kerne tauschen Zustände über mutex-geschützte Shared-Memory-Strukturen aus. Die Anbindung an den Raspberry Pi 5 erfolgt über USB-CDC mit 921 600 baud; dort laufen `micro-ros-agent` (`/dev/ttyACM0`) und die ROS 2-Humble-Entwicklungsumgebung in Docker-Containern.

7.2 Hardware Abstraction Layer (HAL)

Die Pin-Belegung ist für vollständige Hardware-Nutzung refaktoriert. Alle Pins werden initialisiert, um Floating-States zu vermeiden.

Ressource	Pin	PWM-CH	Core	Funktion	Status
Motor L-A	D0	CH 1	Core 0	PWM Vorwärts	✓ Aktiv
Motor L-B	D1	CH 0	Core 0	PWM Rückwärts	✓ Aktiv
Motor R-A	D2	CH 3	Core 0	PWM Vorwärts	✓ Aktiv
Motor R-B	D3	CH 2	Core 0	PWM Rückwärts	✓ Aktiv
Encoder L	D6	–	Core 0	ISR (Rising Edge)	✓ Aktiv
Encoder R	D7	–	Core 0	ISR (Rising Edge)	✓ Aktiv
Safety/LED	D10	–	Core 0	MOSFET (Not-Aus)	✓ Aktiv
I2C SDA	D4	–	Core 1	IMU (MPU6050)	□ Reserviert
I2C SCL	D5	–	Core 1	IMU (MPU6050)	□ Reserviert
Servo Pan	D8	–	Core 1	Kamera Pan	□ Reserviert
Servo Tilt	D9	–	Core 1	Kamera Tilt	□ Reserviert

7.2.1 PWM-Kanäle (A↔B getauscht)

```

1 // config.h
2 #define PWM_CH_LEFT_A 1 // war 0
3 #define PWM_CH_LEFT_B 0 // war 1
4 #define PWM_CH_RIGHT_A 3 // war 2
5 #define PWM_CH_RIGHT_B 2 // war 3

```

7.3 Firmware-Logik (main.cpp)

Die Firmware basiert auf zwei parallelen Tasks.

7.3.1 Task: controlTask (Core 0)

Dies ist das “Rückenmark” des Roboters.

Eigenschaft Wert

Frequenz 100 Hz (fixiert via vTaskDelayUntil)

Priorität configMAX_PRIORITIES - 1

Stack 4096 B

Ablauf (pro Tick):

1. Mutex Lock: Zielwerte aus Shared Memory lesen
2. Atomic Read: Encoderwerte lesen (Interrupts gesperrt)
3. Odometrie: x, y, θ integrieren
4. Richtungs-Heuristik: Encoder-Vorzeichen aus Soll-Geschwindigkeit
5. Feedforward + PID: Stellgrößen berechnen (PID aktuell deaktiviert)
6. Safety: last_cmd_time prüfen → Stop bei Timeout
7. Mutex Lock: Odometrie in Shared Memory schreiben

7.3.2 Steuerungslogik (Feedforward)

```

1 // Inverse Kinematik: Twist → Radgeschwindigkeiten
2 float set_v_l = target_v - (target_w * WHEEL_BASE / 2.0f);
3 float set_v_r = target_v + (target_w * WHEEL_BASE / 2.0f);
4
5 // Feedforward + PID (PID aktuell deaktiviert: Kp=Ki=Kd=0)
6 float feedforward_gain = 2.0f;
7 float pwm_l = feedforward_gain * set_v_l +
   pid_left.compute(set_v_l, v_enc_l, dt);
8 float pwm_r = feedforward_gain * set_v_r +
   pid_right.compute(set_v_r, v_enc_r, dt);
9
10 // Begrenzen auf PWM-Bereich
11 pwm_l = constrain(pwm_l, -1.0f, 1.0f);
12 pwm_r = constrain(pwm_r, -1.0f, 1.0f);
13
14 // Hardware ansteuern
15 hal_motor_write(pwm_l, pwm_r);

```

Warum Feedforward statt PID?

1

Encoder liefern nur Single-Channel-Signale (A-only), daher keine echte Richtungserkennung. Die Richtung wird aus der Soll-Geschwindigkeit abgeleitet. Mit aktivem PID kam es zur Eskalation (Vorzeichen-Fehlinterpretation). Feedforward umgeht das Problem.

7.3.3 Task: loop (Core 1)

Dies ist das "Sprachzentrum".

Eigenschaft	Wert
Frequenz	Best Effort
Odom Publish	20 Hz (alle 50 ms)
Heartbeat	1 Hz

Ablauf:

1. `rclc_executor_spin_some`: neue `/cmd_vel` prüfen
2. Odom Publish: alle 50 ms aus Shared Memory lesen und senden
3. Heartbeat: alle 1000 ms Counter inkrementieren und senden
4. Shared Memory Zugriff nur via `xSemaphoreTake`

7.4 Schnittstellen & Datenfluss**7.4.1 ROS 2 Topics**

Topic	Typ	Richtung	QoS	Freq.	Beschreibung
<code>/cmd_vel</code>	<code>geometry_msgs/Twist</code>	Pub	Reliable	-	Geschwindigkeitsbefehle
<code>/odom_raw</code>	<code>geometry_msgs/Pose</code>	Pub	Best Effort	20 Hz	Odometrie (x, y, θ)
<code>/esp32/heartbeat</code>	<code>std_msgs/Int32</code>	Pub	Best Effort	1 Hz	Lebenszeichen
<code>/esp32/led_cmd</code>	<code>std_msgs/Bool</code>	Sub	Reliable	-	LED/MOSFET-Steuerung

7.4.2 Nachrichtenformate (Beispiele)

```
/cmd_vel (Input)
linear:
  x: 0.15    # [m/s] Vorwärts (+) / Rückwärts (-)
angular:
  z: 0.5     # [rad/s] Links (+) / Rechts (-)
```

```
/odom_raw (Output)
x: 0.899    # [m] Position X
y: -0.329   # [m] Position Y
theta: 6.09  # [rad] Orientierung
```

7.4.3 Shared Memory Struktur

```
1 struct SharedData {
2     // Input (Core 1 → Core 0)
3     float target_lin_x;           // Soll-Linear-Geschwindigkeit
4                           [m/s]
5     float target_ang_z;          // Soll-Winkel-Geschwindigkeit
6                           [rad/s]
7     bool led_cmd_active;         // LED-Status
8     unsigned long last_cmd_time; // Zeitstempel für Failsafe
9
10    // Output (Core 0 → Core 1)
11    float odom_x;               // Position X [m]
12    float odom_y;               // Position Y [m]
13    float odom_theta;            // Orientierung [rad]
14};
```

7.5 Konfiguration (config.h)

7.5.1 Timing

Parameter	Wert	Beschreibung
LOOP_RATE_HZ	100	Control-Loop Frequenz
ODOM_PUBLISH_HZ	20	Odometrie Publish Rate
FAILSAFE_TIMEOUT_MS	2000	Heartbeat-Timeout

7.5.2 Kinematik

Parameter	Wert	Beschreibung
WHEEL_DIAMETER	0,065 m	Raddurchmesser
WHEEL_BASE	0,178 m	Spurbreite
TICKS_PER_REV_LEFT	374.3	Encoder-Ticks pro Umdrehung
TICKS_PER_REV_RIGHT	373.6	Encoder-Ticks pro Umdrehung

7.5.3 Regelung

Parameter	Wert	Beschreibung
PID_KP	0.0	deaktiviert
PID_KI	0.0	deaktiviert
PID_KD	0.0	deaktiviert
feedforward_gain	2.0	Direkte Ansteuerung

7.5.4 PWM

Parameter	Wert	Beschreibung
MOTOR_PWM_FREQ	20000	20 kHz (unhörbar)
MOTOR_PWM_BITS	8	0–255 Auflösung
PWM_DEADZONE	35	Mindest-PWM

7.6 Deployment

7.6.1 Firmware Update (Mac)

```
cd ~/daten/start/IoT/AMR/amr-platform/firmware
pio run -e seeed_xiao_esp32s3 -t upload
```

7.6.2 Docker starten (Pi)

```
cd ~/amr-platform/docker
docker compose up -d
```

```
sleep 5  
docker compose logs microros_agent --tail 5
```

Erwartung

```
running... | fd: 3
```

7.6.3 Nach ESP32 Reboot

```
docker compose restart microros_agent  
sleep 5
```

7.7 Testing

7.7.1 Verbindung prüfen

```
docker compose exec amr_dev bash  
source /opt/ros/humble/setup.bash  
ros2 topic list
```

Erwartung

```
/cmd_vel  
/esp32/heartbeat  
/esp32/led_cmd  
/odom_raw
```

7.7.2 Heartbeat prüfen

```
ros2 topic echo /esp32/heartbeat
```

7.7.3 Odometrie prüfen

```
ros2 topic echo /odom_raw --once
```

7.7.4 Motor-Test (Räder aufbocken!)

Sicherheit

Motor-Tests nur mit aufgebockten Rädern (kein Bodenkontakt). !

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \
  "{linear: {x: 0.15}, angular: {z: 0.0}}" -r 10
```

Ctrl+C → Failsafe stoppt nach 2s

7.7.5 Richtungen testen

Befehl	Erwartung
linear.x: 0.15	Vorwärts
linear.x: -0.15	Rückwärts
angular.z: 0.5	Drehen links
angular.z: -0.5	Drehen rechts

7.8 Troubleshooting

Problem	Ursache	Lösung
Motor reagiert nicht	Feedforward zu niedrig	feedforward_gain erhöhen
PID eskaliert	Encoder-Polarität invertiert	PID deaktivieren ($K_p=0$)
Räder drehen falsch	PWM-Kanäle	A↔B tauschen
Failsafe greift zu früh	Timeout zu kurz	FAILSAFE_TIMEOUT_MS erhöhen
Topics fehlen	Agent nicht verbunden	docker compose restart microros_agent
Keine Odom-Daten	QoS Mismatch	Best Effort QoS nutzen

7.9 Bekannte Einschränkungen

1. Open-Loop-Steuerung: PID deaktiviert, keine Geschwindigkeitsregelung

2. Encoder A-only: Richtung aus Soll-Geschwindigkeit abgeleitet
3. Odom-Rate: effektiv 3 Hz bis 6 Hz durch Serial-Transport
4. Keine TF: `odom` → `base_link` extern erforderlich

7.10 Nächste Entwicklungsschritte

Phase	Aufgabe	Status
Phase 3	RPLidar A1 Integration	<input type="checkbox"/>
Phase 4	EKF Sensor Fusion + TF	<input type="checkbox"/>
Phase 4	<code>odom_converter.py</code> Bridge Node	<input type="checkbox"/>
Phase 5	SLAM (<code>slam_toolbox</code>)	<input type="checkbox"/>
Phase 6	Nav2 Navigation	<input type="checkbox"/>

7.11 Changelog

7.11.1 v3.2.0 (20.12.2025) – Phase 1 Abschluss

- Motor-Richtung: PWM-Kanäle getauscht (A↔B)
- Steuerung: Feedforward (Gain = 2.0) statt PID
- PID: deaktiviert wegen Encoder-Polarität
- Failsafe: Timeout auf 2000 ms erhöht
- Tests: alle Richtungen validiert

7.11.2 v3.1.0 (20.12.2025)

- Baudrate: 921600 Bd
- PID: aktiviert ($K_p = 1.0$)
- Problem: PID-Eskalation

7.11.3 v3.0.0 (14.12.2025)

- Architektur: Dual-Core
- RTOS: FreeRTOS Tasks + Mutex

Hinweis

Diese Dokumentation ist als “Single Source of Truth” für die Firmware-Entwicklung gedacht.



8 Systemdokumentation: AMR Low-Level Controller

Meta

- Version: 3.2.0
- Datum: 20.12.2025
- Status: ✓ Phase 1 abgeschlossen

8.1 Architektur-Übersicht

Das System implementiert eine **Hybrid-Echtzeit-Architektur**. Harte Echtzeit-Anforderungen (Motorsteuerung) sind strikt von Kommunikations-Aufgaben (micro-ROS) getrennt durch Dual-Core-Nutzung des ESP32-S3.

8.1.1 Datenfluss (Übersicht)

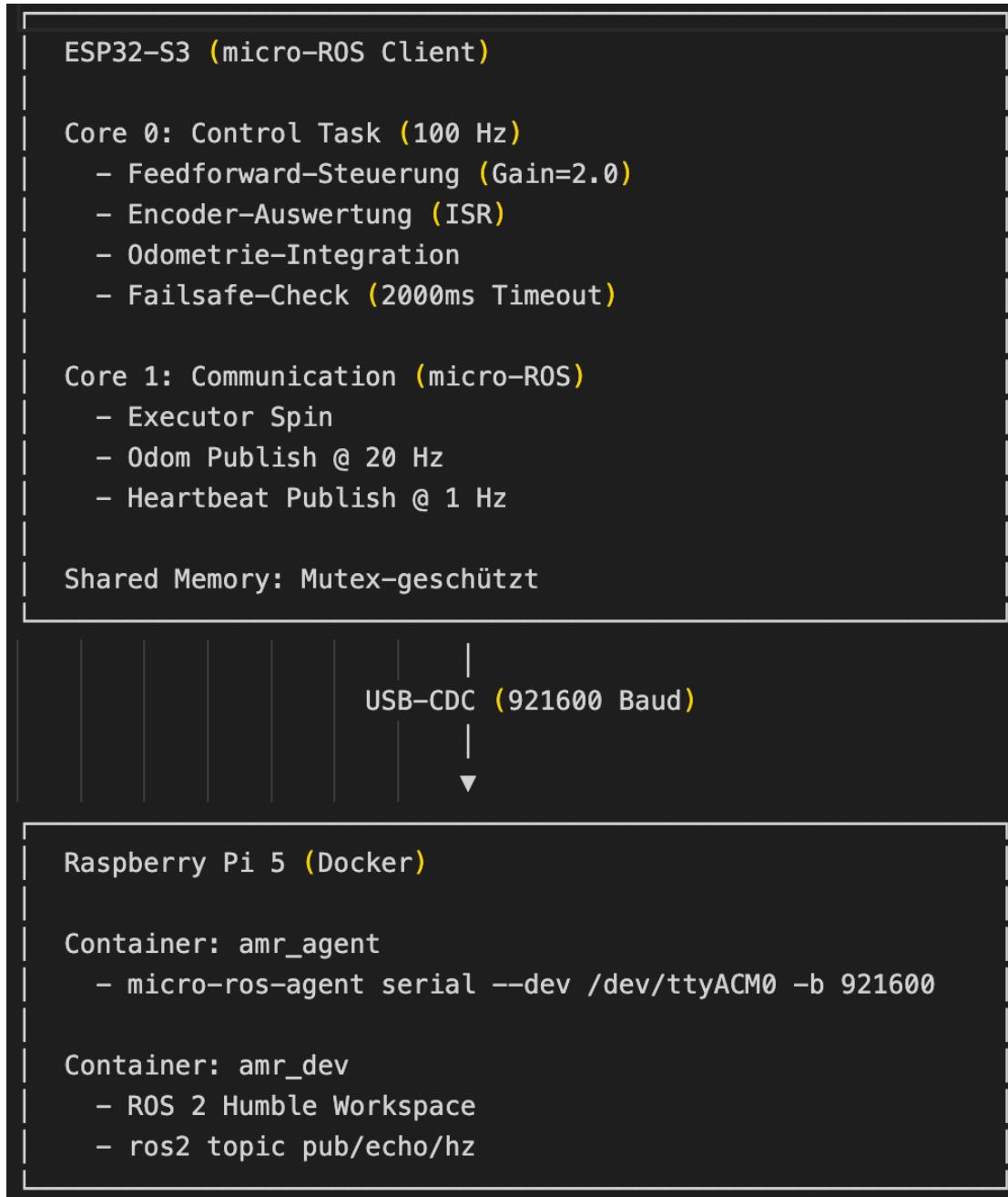


Abbildung 7: Phase-1-Architektur (micro-ROS über USB-Serial): Der ESP32-S3 arbeitet als micro-ROS-Client und führt auf Core 0 die Regel-/Odometrie-Schleife (100 Hz) inkl. Failsafe (2 s Timeout) aus, während Core 1 die micro-ROS-Kommunikation bedient. Über USB-CDC bei 921600baud ist der Raspberry Pi 5 (Docker) mit dem `micro-ros-agent` verbunden; im ROS 2-Workspace werden Topics wie `/cmd_vel` (Motorsteuerung) und `/odom_raw` (Pose2D) getestet/überwacht.

8.2 Firmware-Architektur (Dual-Core)

Die Firmware nutzt **FreeRTOS**, um zwei parallele Tasks auf den physischen CPU-Kernen auszuführen.

8.2.1 Core 0: controlTask (Hard Real-Time)

Eigenschaft	Wert
Task Name	controlTask
Frequenz	100 Hz (deterministisch via vTaskDelayUntil)
Priorität	hoch (configMAX_PRIORITIES – 1)

Aufgaben:

1. Encoder-Interrupts atomar auslesen
2. Odometrie integrieren (x, y, θ)
3. Feedforward-Stellgrößen berechnen (Gain = 2.0)
4. Safety: Heartbeat-Timeout (2000 ms) \Rightarrow Not-Halt

8.2.2 Core 1: loop() (Communication)

Eigenschaft	Wert
Task	loop() (Arduino Standard)
Odom Publish	20 Hz
Heartbeat	1 Hz

Aufgaben:

1. micro-ROS Executor Spin (Empfang/Versand)
2. Serialisierung der DDS-Nachrichten
3. I²C-Kommunikation (geplant: IMU)

Datenaustausch: über **SharedData** Struct, geschützt durch Mutex/Semaphore.

8.2.3 Steuerungslogik (Feedforward, PID deaktiviert)

```

1 // Feedforward + PID (PID aktuell deaktiviert)
2 float feedforward_gain = 2.0f;
3 float pwm_l = feedforward_gain * set_v_l +
   pid_left.compute(set_v_l, v_enc_l, dt);
4 float pwm_r = feedforward_gain * set_v_r +
   pid_right.compute(set_v_r, v_enc_r, dt);
5
6 // Begrenzen auf PWM-Bereich
7 pwm_l = constrain(pwm_l, -1.0f, 1.0f);
8 pwm_r = constrain(pwm_r, -1.0f, 1.0f);

```

Hinweis

PID ist deaktiviert ($K_p = K_i = K_d = 0$), da die Encoder-Polarität invertiert ist. Feedforward ermöglicht stabile Open-Loop-Steuerung.

8.3 ROS 2 Schnittstelle (API)

8.3.1 Topics

Topic	Typ	Richtung	Frequenz	QoS	Beschreibung
/cmd_vel	geometry_msgs/Twist	Pub	–	Reliable	Geschwindigkeitsbefehle
/odom_raw	geometry_msgs/Pose2D	Pub	20 Hz	Best Effort	Odometrie (x, y, θ)
/esp32/heartbeat	std_msgs/Int32	Pub	1 Hz	Best Effort	Lebenszeichen
/esp32/led_cmd	std_msgs/Bool	Sub	–	Reliable	LED/MOSFET- Steuerung

8.3.2 Nachrichtenformate

/cmd_vel (Input)

```

linear:
  x: 0.15    # [m/s] Vorwärts (+) / Rückwärts (-)
  y: 0.0      # Nicht verwendet
  z: 0.0      # Nicht verwendet
angular:
  x: 0.0      # Nicht verwendet

```

```
y: 0.0      # Nicht verwendet
z: 0.5      # [rad/s] Links (+) / Rechts (-)
```

/odom_raw (Output)

```
x: 0.899    # [m]
y: -0.329   # [m]
theta: 6.09  # [rad]
```

8.4 Konfiguration & Parameter

8.4.1 config.h

Parameter	Wert	Beschreibung
LOOP_RATE_HZ	100	Control-Zyklus (10 ms)
ODOM_PUBLISH_HZ	20	Odom Publish (50 ms)
FAILSAFE_TIMEOUT_MS	2000	Heartbeat-Timeout
MOTOR_PWM_FREQ	20000	20 kHz (unhörbar)
MOTOR_PWM_BITS	8	Auflösung 0–255
PWM_DEADZONE	35	Mindest-PWM
WHEEL_DIAMETER	0.065	Raddurchmesser [m]
WHEEL_BASE	0.178	Spurbreite [m]

8.4.2 PWM-Kanäle (getauscht für korrekte Richtung)

```
1 #define PWM_CH_LEFT_A 1 // war 0
2 #define PWM_CH_LEFT_B 0 // war 1
3 #define PWM_CH_RIGHT_A 3 // war 2
4 #define PWM_CH_RIGHT_B 2 // war 3
```

8.4.3 Regelung

Parameter	Wert	Beschreibung
PID_KP	0.0	deaktiviert
PID_KI	0.0	deaktiviert
PID_KD	0.0	deaktiviert
feedforward_gain	2.0	Direkte Ansteuerung

8.4.4 HAL-Pins (Übersicht)

Pin	Funktion	Modus	Hardware
D0	Motor Left A	PWM → CH 1	Cytron MDD3A
D1	Motor Left B	PWM → CH 0	Cytron MDD3A
D2	Motor Right A	PWM → CH 3	Cytron MDD3A
D3	Motor Right B	PWM → CH 2	Cytron MDD3A
D6	Encoder Left	ISR (Rising)	JGA25-370
D7	Encoder Right	ISR (Rising)	JGA25-370
D10	LED/MOSFET	Digital Out	IRLZ24N
D4, D5	I ² C	Wire	reserviert (MPU6050)
D8, D9	Servo	PWM	reserviert (Kamera)

8.5 Inbetriebnahme

8.5.1 Nach Pi Reboot

```
cd ~/amr-platform/docker
docker compose up -d
sleep 5
docker compose logs microros_agent --tail 5
```

Erwartung

running... | fd: 3

8.5.2 Nach ESP32 Reboot

```
cd ~/amr-platform/docker
docker compose restart microros_agent
sleep 5
docker compose logs microros_agent --tail 5
```

8.5.3 Verifikation

Topics prüfen

```
docker compose exec amr_dev bash
source /opt/ros/humble/setup.bash
ros2 topic list
```

Erwartung

```
/cmd_vel
/esp32/heartbeat
/esp32/led_cmd
/odom_raw
/parameter_events
/rosout
```

Heartbeat prüfen

```
ros2 topic echo /esp32/heartbeat
```

Odometrie prüfen

```
ros2 topic echo /odom_raw --once
```

Motor-Test (Sicherheit)

Sicherheit

Motor-Tests nur mit aufgebockten Rädern (kein Bodenkontakt). !

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \
  "{linear: {x: 0.15}, angular: {z: 0.0}}" -r 10
```

8.6 Testergebnisse (20.12.2025)

Test	Ergebnis	Status
Agent-Verbindung	fd: 3 stabil	✓
Heartbeat	≈ 1 Hz	✓
Vorwärts	Räder drehen vorwärts	✓
Rückwärts	Räder drehen rückwärts	✓
Drehen links	Roboter dreht links	✓
Drehen rechts	Roboter dreht rechts	✓
Failsafe	Stop nach ≈ 2 s	✓
Odom	x, y, θ plausibel	✓

8.7 Known Issues & Lösungen

Symptom	Ursache	Lösung
Roboter ruckelt	Failsafe greift	FAILSAFE_TIMEOUT_MS erhöhen (aktuell 2000 ms)
Keine Odom-Daten	QoS Mismatch	Best Effort QoS nutzen
Motor reagiert nicht	Feedforward zu niedrig	feedforward_gain erhöhen
PID eskaliert	Encoder-Polarität invertiert	PID deaktivieren ($K_p = 0$)
Räder drehen falsch	PWM-Kanäle	A↔B tauschen in config.h
Topics fehlen	Agent nicht verbunden	docker compose restart microros_agent

8.8 Bekannte Einschränkungen

1. Open-Loop-Steuerung: PID deaktiviert, keine Geschwindigkeitsregelung
2. Encoder A-only: Richtung aus Soll-Geschwindigkeit abgeleitet
3. Odom-Rate: effektiv 3 Hz bis 6 Hz durch Serial-Transport

8.9 Projektstruktur

```

amr-platform
├── docker
│   ├── docker-compose.yml
│   ├── Dockerfile
│   └── ros_entrypoint.sh
├── docs
├── firmware
│   ├── extra_script.py
│   ├── include
│   │   └── config.h
│   ├── platformio.ini
│   ├── src
│   │   └── main.cpp
│   └── todo-liste.md
├── LICENSE
├── main-design.css
└── README.md
├── ros2_ws
│   └── src
│       ├── amr_bridge
│       ├── amr_bringup
│       ├── amr_description
│       └── sllidar_ros2
└── scripts
    ├── deploy.sh
    ├── md-to-html-converter.py
    ├── microros-agent.service
    └── setup_microros_service.sh
└── start.html

```

Abbildung 8: Projektstruktur des Repositories `amr-platform`: `firmware/` enthält die ESP32-S3-Firmware (PlatformIO mit `src/main.cpp` und `include/config.h`), `ros2_ws/src/` den ROS 2-Workspace mit Paketen für Bridge/Bringup/Description sowie `sllidar_ros2`. Die Laufzeitumgebung ist in `docker/` (Dockerfile, Compose, Entrypoint) gekapselt. Automatisierung und Betrieb liegen in `scripts/` (Deployment, micro-ROS-Agent-Service, Dokumentations-Converter). `docs/` bündelt Projektdokumentation; `README.md`, `LICENSE`, `start.html` und `main-design.css` bilden Einstieg und Styling der Doku.

8.10 Changelog

8.10.1 v3.2.0 (20.12.2025) – Phase 1 Abschluss

- Motor-Richtung: PWM-Kanäle getauscht (A↔B)
- Steuerung: Feedforward (Gain = 2.0) statt PID

- PID: deaktiviert ($K_p = K_i = K_d = 0$) wegen Encoder-Polarität
- Failsafe: Timeout auf 2000 ms erhöht
- Tests: alle Richtungen validiert

8.10.2 v3.1.0 (20.12.2025)

- Baudrate: 921600 Bd
- PID: aktiviert ($K_p = 1.0$)
- Problem: PID-Eskalation durch Encoder-Polarität

8.10.3 v3.0.0 (14.12.2025) – Major Release

- Architektur: Wechsel auf Dual-Core (App/Pro CPU Trennung)
- RTOS: FreeRTOS Tasks und Mutex-Synchronisation
- Daten: Optimierung auf Pose2D (Bandbreite reduziert)
- Hardware: Initialisierung aller Pins

8.11 Nächste Schritte

Phase	Beschreibung	Status
Phase 1	micro-ROS ESP32-S3	✓
Phase 2	Docker-Infrastruktur	✓
Phase 3	RPLidar A1 Integration	□
Phase 4	EKF Sensor Fusion	□
Phase 5	SLAM (<code>slam_toolbox</code>)	□
Phase 6	Nav2 Autonome Navigation	□

9 Git-Workflow: Mac \leftrightarrow GitHub \leftrightarrow Raspberry Pi

Prinzip

- Mac = Entwicklung
- GitHub = *Single Source of Truth*
- Raspberry Pi = Deployment/Runtime
- Stand: 2025-12-12 | Firmware: v0.3.0-serial (Quelle)

9.1 Übersicht

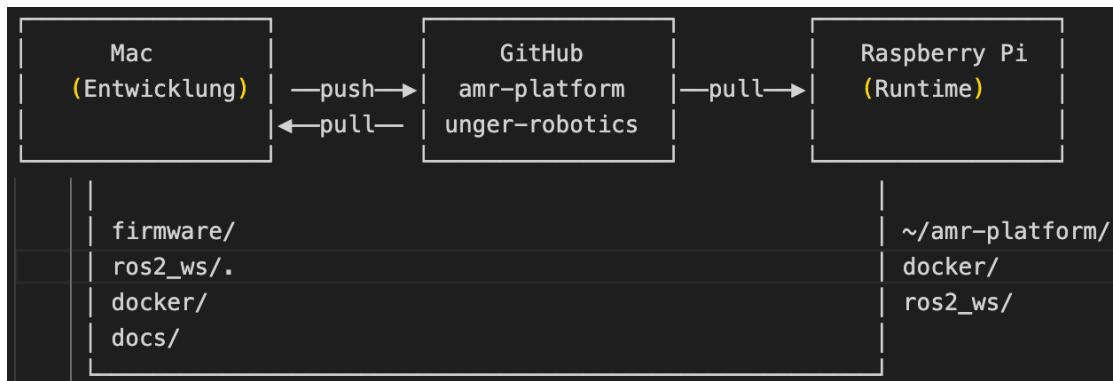


Abbildung 9: Git-Workflow zwischen Mac, GitHub und Raspberry Pi: Entwicklung und Dokumentation erfolgen auf dem Mac; Änderungen werden per `git push` nach GitHub übertragen, das als *Single Source of Truth* dient. Der Raspberry Pi zieht freigegebene Stände per `git pull` für Deployment und Runtime nach. Typische Projektpfade umfassen `firmware_serial/`, `ros2_ws/src/amr_serial_bridge/`, `docker/` und `docs/`; auf dem Pi liegt das Checkout z. B. unter `~/amr-platform/` mit separaten Laufzeit-/Container-Verzeichnissen.

9.2 Initiales Setup

9.2.1 Auf dem Mac (Development)

```

cd /Users/jan/daten/start/IoT/AMR/amr-platform

# 1) Identität setzen (WICHTIG!)
git config --global user.name "Jan Unger"
git config --global user.email "unger.robotics@gmail.com"

# 2) Falls noch nicht initialisiert
git init

```

```
# 3) Remote hinzufügen (SSH bevorzugt)
git remote add origin
    git@github.com:unger-robotics/amr-platform.git

# 4) .gitignore erstellen (WICHTIG!)
cat > .gitignore << 'EOF'
# === Build-Artefakte ===
.pio/
.vscode/
*.o
*.elf
*.bin
*.map

# === Python ===
__pycache__/
*.pyc
.pytest_cache/

# === ROS 2 Build ===
ros2_ws/build/
ros2_ws/install/
ros2_ws/log/

# === Docker ===
*.log

# === Hailo (große Dateien!) ===
*.hef
hailo_models/

# === macOS ===
.DS_Store
*.swp

# === Secrets (niemals committen!) ===
*.key
*.pem
secrets/
```

```
# === IDE ===
*.code-workspace
.idea/
EOF

# 5) Ersten Commit & Upload
git add .
git commit -m "feat: Serial-Bridge Firmware v0.3.0"
git branch -M main
git push -u origin main
```

9.2.2 Auf dem Raspberry Pi (Runtime)

```
# 1) Identität setzen (damit Hotfixes zugeordnet werden)
git config --global user.name "Jan Unger"
git config --global user.email "unger.robotics@gmail.com"

# 2) Backup der aktuellen Konfiguration
cp -r ~/amr-platform ~/amr_backup_$(date +%Y%m%d)

# 3) Repository klonen
cd ~
git clone git@github.com:unger-robotics/amr-platform.git

# 4) Symlinks für Kompatibilität erstellen (optional)
ln -s ~/amr-platform ~/amr
```

9.3 Täglicher Workflow

9.3.1 Auf dem Mac entwickeln

```
cd /Users/jan/daten/start/IoT/AMR/amr-platform

# 1) Vor der Arbeit: aktuellen Stand holen
git pull origin main

# 2) Arbeiten (Code schreiben, testen)
```

```
# ... Änderungen ...

# 3) Status prüfen
git status

# 4) Änderungen stagern
git add firmware_serial/src/main.cpp
# Oder alles:
# git add .

# 5) Commit mit aussagekräftiger Nachricht
git commit -m "fix: Deadzone-Kompensation für kleine
Geschwindigkeiten"

# 6) Hochladen
git push origin main
```

9.3.2 Auf dem Pi deployen

```
cd ~/amr-platform

# 1) Änderungen holen
git pull origin main

# 2) Docker neu starten
cd docker
docker compose down
docker compose up -d

# 3) Logs prüfen
docker compose logs -f serial_bridge
```

9.4 Commit-Konventionen

Präfix	Verwendung	Beispiel
feat:	Neue Funktion	feat: Encoder-ISR für Phase 2
fix:	Bugfix	fix: Failsafe-Timeout auf 500ms
docs:	Dokumentation	docs: README aktualisiert
refactor:	Code-Umbau	refactor: HAL in separate Datei
test:	Tests	test: Motor-Kalibrierung Sketch
chore:	Build/Tooling	chore: PlatformIO auf 6.1.15

Beispiele für gute Commits

```
git commit -m "feat(firmware): Serial-Bridge Protokoll V:x,W:y"
git commit -m "fix(docker): serial_bridge Container
Device-Mapping"
git commit -m "docs: Phase 1 abgeschlossen"
```

9.5 Branching-Strategie (optional)

```
# Neuen Feature-Branch erstellen
git checkout -b feature/encoder-odometry

# Arbeiten, committen...
git add .
git commit -m "feat: Odometrie-Publisher implementiert"

# Branch hochladen
git push -u origin feature/encoder-odometry

# Zurück zu main und mergen
git checkout main
git merge feature/encoder-odometry
```

```
git push origin main

# Branch löschen (lokal + remote)
git branch -d feature/encoder-odometry
git push origin --delete feature/encoder-odometry
```

9.6 Synchronisation Mac ↔ Pi

9.6.1 Schneller Weg: Nur Firmware (ohne Git)

```
# Von Mac aus (rsync über SSH)
rsync -avz --progress \
    /Users/jan/daten/start/IoT/AMR/amr-platform/firmware_serial/ \
    pi@rover:~/amr-platform/firmware_serial/
```

9.6.2 Empfohlener Weg: Über GitHub

Flow

```
Mac: git push → GitHub (unger-robotics) → Pi: git pull
```

9.6.3 Ein-Zeilens-Deploy (Mac → Pi)

```
git add . && git commit -m "fix: Beschreibung" && git push && \
ssh pi@rover "cd ~/amr-platform && git pull && cd docker &&
docker compose up -d"
```

9.7 Häufige Szenarien

9.7.1 “Ich habe auf dem Pi etwas getestet und will es behalten”

```
# Auf dem Pi
cd ~/amr-platform
git add docker/docker-compose.yml
git commit -m "tune: Serial-Bridge Timeout angepasst"
```

```
git push origin main

# Auf dem Mac (später)
git pull origin main
```

9.7.2 “Ich habe lokale Änderungen, will aber den neuesten Stand”

Option A: Änderungen temporär speichern (Stash)

```
git stash
git pull origin main
git stash pop
```

Option B: Änderungen verwerfen (Hard Reset – Vorsicht!)

```
git checkout -- .
git pull origin main
```

9.7.3 “Merge-Konflikt!”

```
# Git zeigt an: CONFLICT in firmware_serial/src/main.cpp

# 1) Datei öffnen, Konflikte manuell lösen
#     Suche nach: <<<<< HEAD ... ====== ... >>>>>

# 2) Gelöste Datei stagern
git add firmware_serial/src/main.cpp

# 3) Merge abschließen
git commit -m "merge: Konflikt in main.cpp gelöst"
```

9.8 Nützliche Aliase (optional)

```
# In ~/.gitconfig hinzufügen:
[alias]
    st = status
    co = checkout
    br = branch
```

```
ci = commit
lg = log --oneline --graph --decorate --all
sync = !git pull && git push
```

9.9 Backup-Strategie

9.9.1 Lokales Backup (Mac)

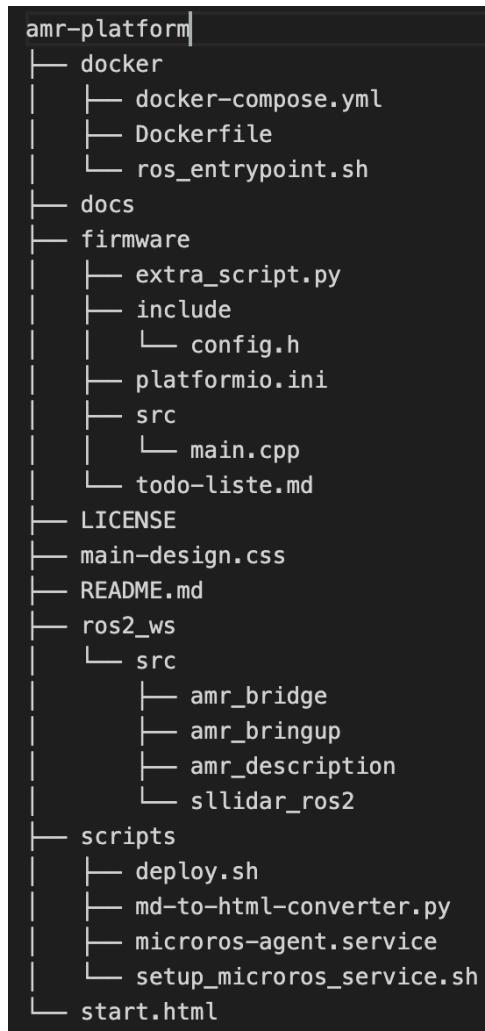
```
# Bare-Repository als Backup
git clone --bare . ~/Backups/amr-platform.git
```

9.9.2 Automatisches Backup (Pi)

```
# Cron-Job für tägliches Backup
crontab -e

# Eintrag hinzufügen:
0 3 * * * cd ~/amr-platform && git add -A && git commit -m "auto:
Daily backup" && git push origin main 2>/dev/null || true
```

9.10 Projekt-Struktur



```
amr-platform
├── docker
│   ├── docker-compose.yml
│   ├── Dockerfile
│   └── ros_entrypoint.sh
├── docs
├── firmware
│   ├── extra_script.py
│   ├── include
│   │   └── config.h
│   ├── platformio.ini
│   ├── src
│   │   └── main.cpp
│   └── todo-liste.md
├── LICENSE
├── main-design.css
└── README.md
├── ros2_ws
│   └── src
│       ├── amr_bridge
│       ├── amr_bringup
│       ├── amr_description
│       └── sllidar_ros2
├── scripts
│   ├── deploy.sh
│   ├── md-to-html-converter.py
│   ├── microros-agent.service
│   └── setup_microros_service.sh
└── start.html
```

Abbildung 10: Projektstruktur des Repositories `amr-platform`: `firmware/` enthält die ESP32-S3-Firmware (PlatformIO mit `src/main.cpp` und `include/config.h`), `ros2_ws/src/` den ROS 2-Workspace mit Paketen für Bridge/Bringup/Description sowie `sllidar_ros2`. Die Laufzeitumgebung ist in `docker/` (Dockerfile, Compose, Entrypoint) gekapselt. Automatisierung und Betrieb liegen in `scripts/` (Deployment, micro-ROS-Agent-Service, Dokumentations-Converter). `docs/` bündelt Projektdokumentation; `README.md`, `LICENSE`, `start.html` und `main-design.css` bilden Einstieg und Styling der Doku.

9.11 Checkliste: Git richtig nutzen

Regel	Warum
Vor der Arbeit: <code>git pull</code>	Konflikte vermeiden
Kleine Commits	Leichter nachvollziehbar
Aussagekräftige Nachrichten	Hilft beim Debuggen
Secrets nie committen	Sicherheit (keine Passwörter)
Identität prüfen	<code>git config user.email</code>
<code>firmware_serial/</code> für ESP32	Nicht <code>firmware/</code>

9.12 Zusammenfassung: Der goldene Pfad

```
# 1) Updaten
git pull origin main

# 2) Arbeiten
# ... code, test, debug ...

# 3) Committen
git add .
git commit -m "feat: Beschreibung der Änderung"

# 4) Synchronisieren
git push origin main

# 5) Auf Pi deployen
ssh pi@rover "cd ~/amr-platform && git pull && cd docker &&
docker compose up -d"
```

9.13 Quick Reference Card

Aktion	Mac	Pi
ESP32 flashen	cd firmware_serial && pio run -t upload	-
Serial Monitor	pio device monitor	screen /dev/ttyACM0 115200
Docker starten	-	docker compose up -d
Logs anzeigen	-	docker compose logs -f serial_bridge
Git sync	git pull && git push	git pull
Teleop testen	-	docker exec -it amr_perception ros2 run teleop_twist_keyboard teleop_twist_keyboard

10 Industriestandards für AMR-Entwicklung

Warum das wichtig ist

Für ein AMR-System (Raspberry Pi 5 + ESP32 + ROS 2) sind saubere Standards nicht “nice to have”: SLAM (`slam_toolbox`) und Navigation (Nav2) hängen direkt an korrekten Einheiten, Frames und Safety-Regeln.

10.1 ROS-Standards: REPs (ROS Enhancement Proposals)

10.1.1 REP-103: Einheiten & Koordinaten

Regel: In ROS wird konsistent in SI gerechnet und das Koordinatensystem ist festgelegt.

- **Einheiten:** Strecke in m, Winkel in rad, Zeit in s
- **Achsen:** x vorwärts, y links, z oben
- **Rotation:** Rechte-Hand-Regel, gegen Uhrzeigersinn ist positiv

Beispiel: Radstand / Spurbreite wird in der Firmware als

$$WHEEL_BASE = 0.178 \text{ m}$$

und nicht als 178 (mm) codiert.

Anwendung im Projekt:

- Encoder-Ticks in m umrechnen, bevor Odometrie publiziert wird.
- `/cmd_vel` immer in m/s und rad/s interpretieren.

10.1.2 REP-105: Koordinaten-Frames (TF-Tree)

Regel: Navigation braucht einen konsistenten TF-Baum.



Zuordnung im System:

- ESP32: publiziert `odom` → `base_link` (Odometrie driftet)
- SLAM/Localization auf Pi: publiziert `map` → `odom` (globale Karte)
- Robot-Description/Static TF: `base_link` → `laser_frame` (Montage)

10.2 Sicherheitsstandard (Safety)

10.2.1 Heartbeat / Dead Man's Switch

Regel: Wenn die Master-Seite (Pi / ROS) ausfällt, müssen die Motoren innerhalb kurzer Zeit stoppen.

- Heartbeat-Intervall: typisch ≈ 100 ms
- Timeout: > 500 ms ohne gültige Nachricht ⇒ **Motor Stop** (PWM = 0)

Anwendung im Projekt:

- Failsafe direkt in der ESP32-Firmware implementieren (Hard Real-Time).
- Timeout-Wert bewusst wählen (z. B. 500 ms bis 2000 ms, je nach Kommunikationspfad).

10.3 Architektur-Standard: Hybrid Master–Slave

Regel: Echtzeit-Aufgaben gehören auf den ESP32, “Denken” auf den Pi.

Teil	Rolle	Beispiele
ESP32 (Hard RT)	Physiknah, zeitkritisch, sicherheitsrelevant	PWM, Encoder-ISR, Not-Aus/Failsafe
Pi 5 (Soft RT)	Rechenlastig, tolerant gegen kurze Lags	SLAM, Nav2, Vision, Planung

10.4 Code-Qualität (Best Practices)

10.4.1 Non-Blocking Code

Regel: Keine blockierenden Wartezeiten in Kommunikationspfaden.

- kein `delay()` im Control-/Comm-Loop
- stattdessen Timer über `millis()` oder FreeRTOS Tasks (`vTaskDelayUntil`)

10.4.2 Topic Naming (ROS-Konvention)

- `/cmd_vel` (Steuerung)
- `/odom` bzw. `/odom_raw` (Odometrie)
- `/scan` (LiDAR)
- `/diagnostics` (Status/Fehler)

10.5 Projekt-Checkliste (minimal)

- SI-Einheiten überall (m, rad, s)
- TF-Baum: `map->odom->base_link->laser_frame`
- Failsafe: Timeout \leq 500 ms (oder begründet größer)
- Keine blockierenden Delays in Echtzeit-/Kommunikationspfaden
- Topic-Namen nach ROS-Konvention

11 Kostenübersicht AMR-Projekt

Kurzfazit	1
Gesamtsumme: 482,48 € (< 500 €, Reserve: 17,52 €)	

11.1 Übersicht nach Händler

Händler	Rechnung-Nr.	Datum	Betrag
BerryBase	RE-18181726	28.11.2025	253,90 €
Welectron	RE211899	28.11.2025	78,85 €
Botland	FOS/183/12/2025	02.12.2025	42,99 €
Botland	FOS/197/12/2025	02.12.2025	25,49 €
AliExpress	PayPal 7438758967648570K	28.11.2025	45,34 €
AliExpress	PayPal 7S180172G6404035R	28.11.2025	19,22 €
AliExpress	PayPal 7DE836060A880374E	28.11.2025	5,57 €
Amazon	DS-AEU-INV-DE-2025-596219183	28.11.2025	11,12 €
Gesamt			482,48 €

11.2 Detaillierte Aufstellung

11.2.1 A. Rechenleistung & Sensorik (Intelligence)

Position	Einzelpreis
Raspberry Pi 5 (8GB RAM)	82,90 €
Seeed RPLIDAR A1 (360°, 2D, 12 m)	89,90 €
Raspberry Pi Global Shutter Kamera	58,90 €
Netzteil 27 W USB-C (weiß)	12,40 €
Active Cooler (Pi 5)	6,30 €
Kamerakabel Standard-Mini 500 mm	3,50 €

BerryBase (RE-18181726) – 253,90 €

Position	Netto	Brutto
Raspberry Pi AI Kit (Hailo-8L TPU)	62,10 €	73,90 €
DHL Warenpost	4,16 €	4,95 €

Welectron (RE211899) – 78,85 €

11.2.2 B. Antrieb & Elektronik (Aktorik)

Position	Preis
Cytron MDD3A (Dual-Motortreiber 16 V / 3 A)	8,50 €
Objektiv PT361060M3MP12 CS-Mount 6 mm	27,50 €
Verbindungskabel Buchse–Buchse 30 cm (40 St.)	2,00 €
Versand	4,99 €

Botland (FOS/183/12/2025) – 42,99 €

Position	Preis
SanDisk microSDXC 128 GB Extreme (190 MB / s)	20,50 €
Versand	4,99 €

Botland (FOS/197/12/2025) – 25,49 €

Position	Menge	Preis
JGA25-370 Getriebemotor 12 V 170 RPM mit Encoder	2	20,37 €
Roboduino Roboter-Chassis (Alu 2 mm, doppelt)	1	16,44 €
PTZ-Halterung 2-Achsen Pan/Tilt	1	3,85 €
MG90S Servo (Metallgetriebe)	2	3,69 €
Kugelrolle/Caster (Carbon-Stahl)	5	0,99 €

AliExpress Robotik (PayPal 7438758967648570K) – 45,34 €

Position	Menge	Preis
BMS Schutzplatine 3S 25A Li-Ion Balance	2	5,57 €

AliExpress BMS (PayPal 7DE836060A880374E) – 5,57 €

11.2.3 C. Infrastruktur & Sonstiges

Position	Preis
DC/DC Wandler 12 V / 24 V → 5 V USB-C (25 W)	11,12 €

Amazon (DS-AEU-INV-DE-2025-596219183) – 11,12 €

Position	Preis
Hitzebeständiger Schrumpfschlauch (15 m)	1,44 €
PVC-Hülle für 18650-Zellen	1,26 €
Inspektionsspiegel 360°	2,62 €
18650 Lithium-Batteriehalter (100 St.)	2,26 €
Schweißdraht/Litzen	0,80 €
Wasserdichte Steckverbinder	1,28 €
Schraubstock magnetisch	3,52 €
Werkzeugtasche Hüftgurt	3,16 €
Diamant-Trennscheiben 22 mm (12 St.)	0,99 €
Edelstahl-Lineal	1,89 €

AliExpress Werkzeug (PayPal 7S180172G6404035R) – 19,22 €

11.3 Kostenstruktur nach Kategorie

Kategorie	Betrag	Anteil
Intelligence & Vision	332,75 €	69 %
Mechanik & Antrieb	93,90 €	19 %
Infrastruktur & Werkzeug	55,83 €	12 %

11.4 Bewertung

11.4.1 Budget-Einhaltung

Kriterium	Ziel	Erreicht
Gesamtkosten	< 500 €	482,48 € ✓
Reserve	> 0 €	17,52 € ✓

11.4.2 Kosten–Nutzen (Kurz)

Hohe Investitionen (gerechtfertigt)

- RPLIDAR A1 (89,90 €): SLAM-Grundlage, Qualität/Support besser als sehr günstige Alternativen
- Raspberry Pi 5 (82,90 €): Nav2 + SLAM + ROS 2 auf arm64
- Hailo-8L (78,85 €): ≈ 13 TOPS für Vision (optional, später)

Kostenoptimierungen

- Motoren (AliExpress, 20,37 €) statt EU-Preisniveau
- Chassis (AliExpress, 16,44 €) statt Fertigplattform
- BMS (5,57 €) statt fertigem Akkupack

11.5 Nicht enthaltene Kosten (Schätzung)

Position	geschätzt	Bemerkung
Samsung 35E 18650-Zellen (3S)	≈ 15 €	bereits vorhanden
ESP32-S3 XIAO	≈ 8 €	bereits vorhanden
MPU6050 IMU	≈ 3 €	bereits vorhanden
Lötmaterial, Schrauben	≈ 5 €	pauschal

Realistische Gesamtkosten inkl. Vorhandenem: ≈ 513 €

MetadatenKostenübersicht erstellt: 2025-12-12 | Projekt: AMR Bachelor-Thesis ✓

12 ToDo-Liste AMR-Projekt

Stand

- Stand: 2025-12-20
- Aktuelle Phase: **Phase 4 (URDF/TF/EKF)**

1

12.1 Phasen-Übersicht

Phase	Beschreibung	Status
Phase 1	micro-ROS auf ESP32-S3	✓ Abgeschlossen
Phase 2	Docker-Infrastruktur	✓ Abgeschlossen
Phase 3	RPLidar A1 Integration	✓ Abgeschlossen
Phase 4	URDF + TF + EKF	▫ NÄCHSTE
Phase 5	SLAM (<code>slam_toolbox</code>)	□
Phase 6	Nav2 Autonome Navigation	□

12.2 Phase 1: micro-ROS ESP32-S3 (abgeschlossen)

12.2.1 Firmware v3.2.0 (2025-12-20)

- ✓ micro-ROS Client über USB-CDC (Serial)
- ✓ Dual-Core FreeRTOS (Core 0: Control, Core 1: Comms)
- ✓ `/cmd_vel` → Motorsteuerung
- ✓ `/odom_raw` → Odometrie (Pose2D)
- ✓ `/esp32/heartbeat` → Lebenszeichen (1 Hz)
- ✓ Failsafe (2000 ms Timeout)
- ✓ Feedforward-Steuerung (Gain = 2.0)

12.3 Phase 2: Docker-Infrastruktur (abgeschlossen)

- ✓ `amr_agent` Container (micro-ROS Agent)
- ✓ `amr_dev` Container (ROS 2 Humble Workspace)
- ✓ `docker compose up -d` funktioniert
- ✓ Volumes für `ros2_ws` gemountet

12.4 Phase 3: RPLidar A1 (abgeschlossen)

12.4.1 Verifiziert (2025-12-20)

- ✓ `sllidar_ros2` gebaut
- ✓ `/scan` publiziert ($\approx 7,6$ Hz)
- ✓ Scan-Daten plausibel (0,05 m bis 12 m)
- ✓ `frame_id: laser`
- ✓ `docker-compose.yml` aktualisiert

12.4.2 Hardware-Info

Parameter	Wert
S/N	74A5FA89C7E19EC8BCE499F0FF725670
Firmware	1.29
Scan Mode	Sensitivity
Sample Rate	8 kHz
Frequenz	$\approx 7,6$ Hz

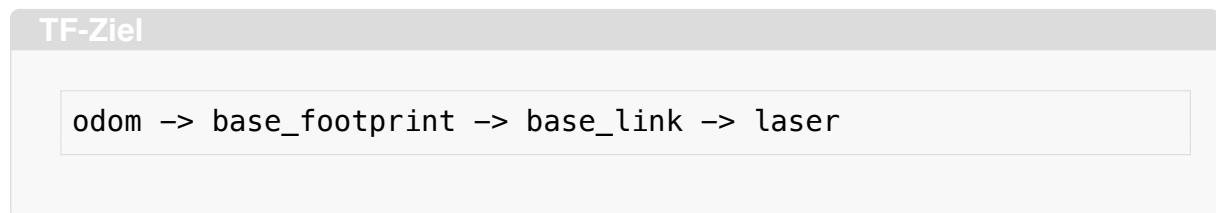
12.5 Phase 4: URDF + TF + EKF (nächste Phase)

12.5.1 ToDos

- URDF erstellen
 - `base_footprint` (Boden)
 - `base_link` (Chassis)

- `laser` Frame
- `robot_state_publisher` für statische TFs
- `odom_converter.py` Bridge Node
 - `/odom_raw` → `/odom`
 - TF: `odom` → `base_footprint`
- Optional: EKF (`robot_localization`)

12.5.2 TF-Baum Ziel



12.6 Phase 5: SLAM

- `slam_toolbox` installieren
- Online Async SLAM
- Testraum kartieren
- Karte speichern

12.7 Phase 6: Nav2

- Nav2 Stack
- AMCL Lokalisierung
- Costmap
- Autonome Navigation

12.8 Aktuelle Hardware-Ports

Device	Port	Funktion
ESP32-S3	/dev/ttyACM0	micro-ROS (921600 Bd)
RPLidar A1	/dev/ttyUSB0	LaserScan (/scan)

12.9 Quick Reference

12.9.1 Container starten

```
cd ~/amr-platform/docker  
docker compose up -d
```

12.9.2 RPLidar starten

```
docker compose exec amr_dev bash  
source /opt/ros/humble/setup.bash  
source /root/ros2_ws/install/setup.bash  
ros2 launch sllidar_ros2 sllidar_a1_launch.py  
    serial_port:=/dev/ttyUSB0
```

12.9.3 Topics prüfen

```
ros2 topic list  
ros2 topic hz /scan  
ros2 topic echo /scan --once
```