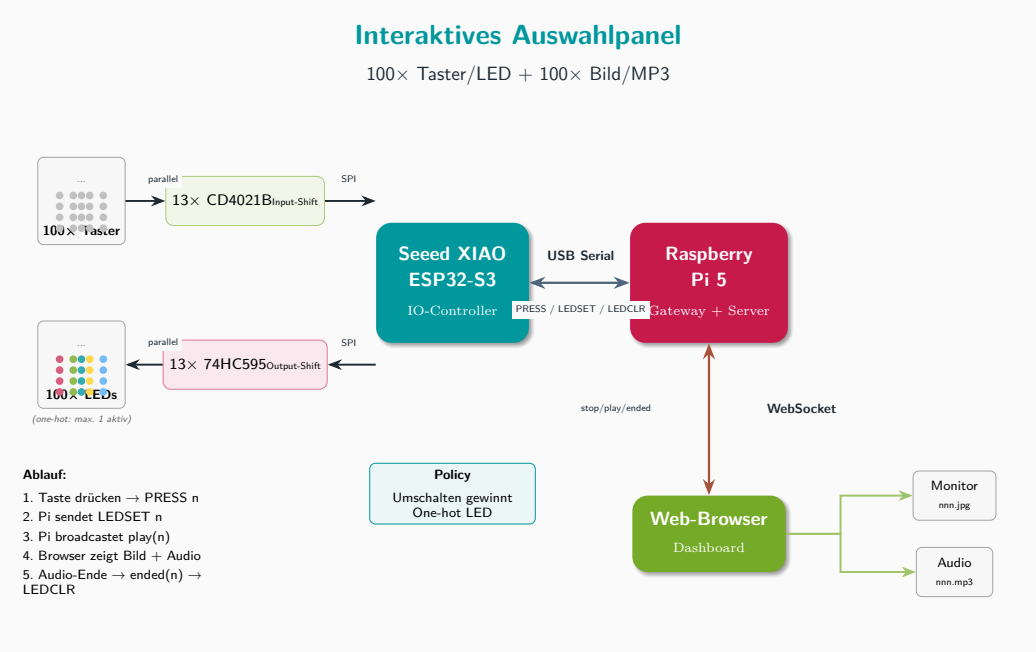
Interaktives Auswahlpanel

100× Taster/LED + 100× Bild/MP3

„Drücken – Leuchten – Abspielen"

ESP32S3 • Raspberry Pi 5 • Web-Dashboard



Technologie-Stack

CD4021BE • 74HC595 • PlatformIO • Python/aiohttp • WebSocket • HTML/JS • systemd

Stand: 8. Januar 2026
Auswahlpanel-Projekt

Jan Unger
Version: 2.5.3

Inhaltsverzeichnis

1 Voraussetzungen

Bevor wir mit dem Aufbau beginnen, werfen wir einen Blick auf die benötigte Hardware und Software. Die folgende Übersicht zeigt alle Komponenten, die wir für das Selection-Panel-Projekt benötigen.

1.1 Hardware

Table 8 listet die Kernkomponenten auf. Bei der Auswahl haben wir auf ein ausgewogenes Verhältnis zwischen Leistung und Kosten geachtet.

Tabelle 1: Hardware-Komponenten für das Selection-Panel

Komponente	Zweck	Bezugsquelle	Preis
Raspberry Pi 5, 4 GB RAM	Server, Media-Ausgabe	BerryBase (RPI5-4GB)	71,50 €
Raspberry Pi Active Cooler	Kühlung	BerryBase (RPI5-ACOOL)	5,90 €
Raspberry Pi 27 W USB-C Netzteil	Stromversorgung	BerryBase (RPI5NT5AW)	12,40 €
SanDisk Extreme microSDXC 128 GB	Betriebssystem	BerryBase (A2 UHS-I U3 V30)	∼18 €
HDMI Adapter Micro-D → A	Monitor	BerryBase (8007067)	1,10 €
Seeed XIAO ESP32-S3	Taster / LED	Reichelt	∼9 €

Bezugsquelle

Alle Raspberry-Pi-Komponenten sind bei BerryBase.de erhältlich. Preise Stand Januar 2026.

1.2 Software (Entwicklungsrechner)

Je nach Betriebssystem unterscheidet sich die Installation der Entwicklungswerkzeuge. Wir zeigen hier die Schritte für macOS, Windows und Linux.

1.2.1 macOS

Unter macOS nutzen wir Homebrew als Paketmanager:

```
/bin/bash -c "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install python git vim
```

1.2.2 Windows

Unter Windows installieren wir die Tools manuell:

1. **Git:** Download von git-scm.com/download/win
2. **Python:** Download von python.org/downloads – Option „Add Python to PATH“ aktivieren

1.2.3 Linux (Ubuntu/Debian)

Unter Linux greifen wir auf die Paketverwaltung zurück:

```
sudo apt update && sudo apt install python3 python3-pip
   python3-venv git vim
```

1.3 VS Code + PlatformIO

Für die Firmware-Entwicklung setzen wir auf VS Code mit der PlatformIO-Extension:

1. **VS Code** von code.visualstudio.com herunterladen und installieren
2. Die Extension `PlatformIO IDE` über den Marketplace installieren

Table 2 fasst die wichtigsten PlatformIO-Befehle zusammen. Mit diesen vier Kommandos decken wir den gesamten Entwicklungszyklus ab.

Tabelle 2: PlatformIO-Befehle für die ESP32-Entwicklung

Aktion	Befehl
Kompilieren	`pio run`
Flashen	`pio run -t upload`
Serial-Monitor	`pio device monitor`
Flash + Monitor	`pio run -t upload -t monitor`

1.4 Referenz-System

Die folgende Konfiguration dient als Referenz für die Dokumentation. Wenn wir auf Versionsnummern oder Verhaltensweisen verweisen, beziehen wir uns auf dieses Setup.

Tabelle 3: Hardware-Referenz

Hardware	Version
Board	Raspberry Pi 5 Model B Rev 1.1
Microcontroller	Seeed XIAO ESP32-S3

Tabelle 4: Software-Versionen

Software	Version
Pi OS	Debian 13 (trixie), Build 2025-12-04
Python	3.13+
aiohttp	3.9+
PlatformIO	6.x
Firmware	2.5.2
Server	2.5.2
Dashboard	2.5.1

1.5 Checkliste

Bevor wir zum nächsten Kapitel übergehen, prüfen wir kurz, ob alle Voraussetzungen erfüllt sind.

1.5.1 Hardware

- ☐ Raspberry Pi 5 + Active Cooler
- ☐ microSD-Karte (128 GB)
- ☐ ESP32-S3 XIAO
- ☐ USB-C Kabel (Daten, nicht nur Laden)
- ☐ Multimeter

1.5.2 Software

- ☐ Git: `git –version`
- ☐ Python: `python3 –version`

☐ VS Code + PlatformIO

☐ SSH-Zugang zum Pi

USB-Kabel beachten

Viele USB-C-Kabel übertragen nur Strom, keine Daten. Ein defektes oder reines Ladekabel ist eine häufige Fehlerquelle beim Flashen des ESP32.

1.6 Nächste Schritte

Mit der Hardware auf dem Tisch und der installierten Software können wir nun in die praktische Umsetzung einsteigen. Table 5 zeigt den empfohlenen Ablauf.

Tabelle 5: Weiterführende Dokumentation

Schritt	Dokument
Pi einrichten + SSH	→ Section 2
Repository klonen	→ Section 16
Server starten	→ Section 14
Löten	→ **??**

2 SSH-Setup

Bevor wir mit der Entwicklung beginnen können, richten wir den Raspberry Pi ein und konfigurieren einen passwortlosen SSH-Zugang. Das Ziel: Mit einem simplen `ssh rover` landen wir direkt auf dem Pi – ohne Passwortabfrage.

2.1 Pi OS installieren

Wir nutzen den offiziellen Raspberry Pi Imager, um das Betriebssystem auf die SD-Karte zu schreiben.

2.1.1 Raspberry Pi Imager

1. **Download:** raspberrypi.com/software
2. **OS auswählen:** Raspberry Pi OS Lite (64-bit)
3. **Einstellungen** über das Zahnrad-Symbol konfigurieren (table 6)

4. **Schreiben** – SD-Karte einlegen – Netzteil einstecken

Tabelle 6: Einstellungen im Raspberry Pi Imager

Einstellung	Wert
Hostname	`rover`
SSH	✓ aktivieren
Benutzer	`pi`
Passwort	(eigenes Passwort)
WLAN	SSID + Passwort
Zeitzone	`Europe/Berlin`

2.1.2 Nach dem ersten Boot

Sobald der Pi hochgefahren ist, verbinden wir uns erstmals per SSH und führen die Grundkonfiguration durch:

```
ssh pi@rover.local
sudo apt update && sudo apt upgrade -y
sudo usermod -aG dialout pi
```

Der letzte Befehl fügt den Benutzer `pi` zur Gruppe `dialout` hinzu – das benötigen wir später für den Zugriff auf den ESP32 über USB.

2.2 SSH-Key einrichten

Passwörter sind umständlich und unsicher. Wir generieren stattdessen ein Schlüsselpaar und kopieren den öffentlichen Schlüssel auf den Pi.

2.2.1 Mac / Linux

```
ssh-keygen -t ed25519
ssh-copy-id pi@rover.local
```

Der erste Befehl erzeugt ein Ed25519-Schlüsselpaar (aktueller Standard, kompakt und sicher). Der zweite kopiert den Public Key automatisch in die `authorized_keys` des Pi.

2.2.2 Windows (PowerShell)

Unter Windows müssen wir zunächst den OpenSSH-Client aktivieren:

```
# OpenSSH aktivieren (als Admin ausfuehren)
Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0

# Key erstellen
ssh-keygen -t ed25519

# Key manuell kopieren
type $env:USERPROFILE\.ssh\id_ed25519.pub | ssh pi@rover.local
   "mkdir -p ~/.ssh && cat >> ~/.ssh/authorized_keys && chmod 600
   ~/.ssh/authorized_keys"
```

Warum Ed25519?

Ed25519 bietet bei nur 256 bit Schlüssellänge eine Sicherheit vergleichbar mit RSA-3072. Die Schlüssel sind kompakter und die Signaturoperationen schneller.[1]

2.3 SSH-Config anlegen

Mit einer SSH-Konfigurationsdatei sparen wir uns künftig die Tipparbeit. Statt `ssh pi@rover.local` genügt dann `ssh rover`.

2.3.1 Mac / Linux

Wir erstellen oder erweitern die Datei `~/.ssh/config`:

```
Host rover
    HostName rover.local
    User pi
    IdentityFile ~/.ssh/id_ed25519
```

Die Berechtigungen müssen stimmen:

```
chmod 600 ~/.ssh/config
```

2.3.2 Windows

Die Config-Datei liegt unter `%USERPROFILE%\.ssh\config`:

```
Host rover
    HostName rover.local
    User pi
    IdentityFile ~/.ssh/id_ed25519
```

Verbindung testen

Nach der Konfiguration testen wir mit `ssh rover`. Wenn alles funktioniert, landen wir ohne Passwortabfrage direkt auf dem Pi.

2.4 GitHub-Zugang einrichten

Für den Zugriff auf GitHub-Repositories erstellen wir einen separaten Schlüssel:

```
ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519_github
```

Die SSH-Config erweitern wir um einen Eintrag für GitHub:

```
Host github.com
    HostName github.com
    User git
    IdentityFile ~/.ssh/id_ed25519_github
    IdentitiesOnly yes
```

Den Public Key hinterlegen wir bei GitHub unter *Settings* → *SSH Keys* → *New SSH Key*. Den Inhalt der Datei `~/.ssh/id_ed25519_github.pub` kopieren wir in das Textfeld.

```
# Verbindung testen
ssh -T git@github.com
# Erwartete Ausgabe: "Hi username! You've successfully
   authenticated..."
```

2.5 Serial-Port-Berechtigungen

Damit wir den ESP32 über USB ansprechen können, benötigt der Benutzer `pi` Zugriff auf die Serial-Ports:

```
# Auf dem Pi ausfuehren
sudo usermod -aG dialout pi
# Danach neu einloggen (oder reboot)

# Stabilen by-id Pfad pruefen
ls -la /dev/serial/by-id/usb-Espressif*
```

Neu einloggen erforderlich

Gruppenmitgliedschaften werden erst nach einem neuen Login aktiv. Ein einfaches `exit` und erneutes `ssh rover` genügt.

2.6 Troubleshooting

Table 7 listet die häufigsten Probleme und deren Lösungen.

Tabelle 7: SSH-Fehlerbehebung

Problem	Lösung
`Permission denied (publickey)`	`ssh-copy-id` erneut ausführen
`Could not resolve hostname`	IP direkt nutzen: `ssh pi@192.168.x.x`
`UNPROTECTED PRIVATE KEY FILE`	Berechtigungen setzen: `chmod 600 ~/.ssh/id_ed25519`
Serial-Port nicht zugänglich	Gruppe hinzufügen: `sudo usermod -aG dialout pi`
mDNS funktioniert nicht	Avahi prüfen: `sudo systemctl status avahi-daemon`

IP-Adresse ermitteln

Falls `rover.local` nicht auflöst, finden wir die IP über den Router oder – falls wir noch einen Monitor am Pi haben – mit `hostname -I`.

3 Hardware-Dokumentation

Wie ist die Hardware des Selection Panels aufgebaut? In diesem Kapitel dokumentieren wir alle Komponenten, Schaltpläne und Verdrahtungsdetails des 10-Button-Prototyps.

3.1 Komponentenübersicht

Tabelle 8: Komponenten des 10-Button-Prototyps

Komponente	Typ	Anzahl	Funktion
XIAO ESP32-S3	Mikrocontroller	1	Steuerlogik, USB-CDC
Raspberry Pi 5	SBC + Netzteil + microSD	1	Server, Dashboard, Medien
CD4021B	8-Bit PISO Schieberegister	2	Taster einlesen
74HC595	8-Bit SIPO Schieberegister	2	LEDs ansteuern
Taster	6×6 mm Tactile Switch	10	Benutzereingabe
LED	5 mm, verschiedene Farben	10	Statusanzeige
Widerstand	330 Ω–3 kΩ	10	LED-Strombegrenzung
Widerstand	10 kΩ	10	Pull-up für Taster
Kondensator	100 nF (Keramik)	4	Stützkondensatoren

3.2 Pin-Belegung XIAO ESP32-S3

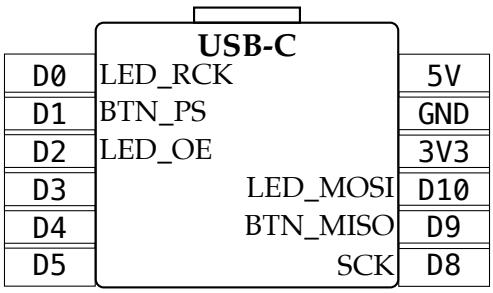


Abbildung 1: Pin-Belegung des XIAO ESP32-S3

Tabelle 9: Signal-Zuordnung der GPIO-Pins

Pin	Signal	Funktion	Ziel
D0	LED_RCK	Latch (STCP)	74HC595 Pin 12
D1	BTN_PS	Parallel/Serial Control	CD4021B Pin 9
D2	LED_OE	Output Enable (PWM, active-low)	74HC595 Pin 13
D8	SCK	SPI Clock	Beide Chip-Typen
D9	BTN_MISO	Daten von CD4021B	CD4021B #0 Pin 3 (Q8)
D10	LED_MOSI	Daten zu 74HC595	74HC595 #0 Pin 14 (SER)

3.3 74HC595 Detailschaltplan (LED-Ausgang)

Der 74HC595 ist ein Serial-In/Parallel-Out Schieberegister mit Latch. Zwei Chips in Daisy-Chain steuern die 10 LEDs.

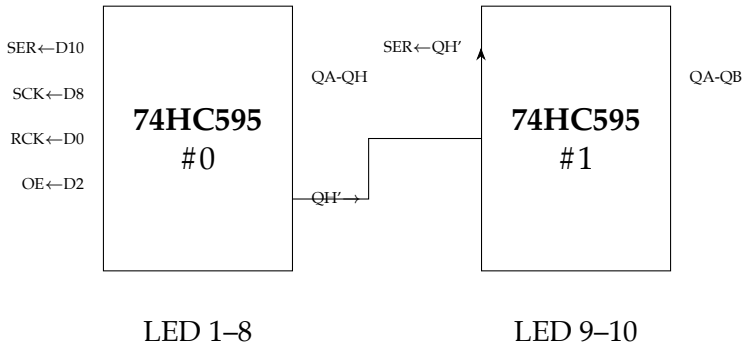


Abbildung 2: 74HC595 Daisy-Chain für LED-Ausgabe

CLR-Pin beachten

Der CLR-Pin (Pin 10) muss auf 3V3 gelegt werden, nicht floaten lassen! Sonst kann das Register ungewollt gelöscht werden.

3.3.1 LED-Beschaltung (Active-High)

Die LEDs sind Active-High beschaltet: Ein HIGH am Ausgang schaltet die LED ein.

Tabelle 10: LED-Stromberechnung bei 3,3 V und $U_f \approx 2{,}0\,\mathrm{V}$

Widerstand	Strom	Helligkeit
330 Ω	≈4 mA	Hell
1 kΩ	≈1,3 mA	Gedimmt
3 kΩ	≈0,4 mA	Schwach

Die Berechnung folgt dem Ohmschen Gesetz:

$$I = \frac{U_{CC} - U_f}{R} = \frac{3{,}3\,\mathrm{V} - 2{,}0\,\mathrm{V}}{R} \tag{1}$$

3.4 CD4021B Detailschaltplan (Taster-Eingang)

Der CD4021B ist ein Parallel-In/Serial-Out Schieberegister. Zwei Chips in Daisy-Chain lesen die 10 Taster ein.

Tabelle 11: CD4021B Pinout (DIP-16)

Pin	Name	Funktion
1	PI-8	Parallel Input 8 (erstes Bit nach Load)
3	Q8	Serial Output (Daten zum ESP32)
7	PI-1	Parallel Input 1
8	VSS	Ground
9	P/S	Parallel/Serial Control
10	CLOCK	Takteingang
11	DS	Serial Data Input (Daisy-Chain)
16	VDD	+3V3

3.4.1 Daisy-Chain Datenfluss

Die Daten fließen vom letzten Chip zum ersten und dann zum ESP32:

```
3V3 --> DS [CD4021B #1] --> Q8 --> DS [CD4021B #0] --> Q8 --> D9
   (MISO) --> ESP32
            |                          |
            |  BTN 9-10                |  BTN 1-8
            |  (Byte 1)                |  (Byte 0)
            |                          |
```

```
            +-- Spaeter gelesen -------+-- Zuerst gelesen -->
```

DS-Pin des letzten Chips

Der DS-Pin (Pin 11) des **letzten** CD4021B muss auf **+3V3** gelegt werden, nicht auf GND. Bei GND würden Nullen nachgeschoben, die als „gedrückt"fehlinterpretiert werden. Ungenutzte PI-x Pins ebenfalls auf +3V3 legen.

3.4.2 Taster-Beschaltung (Active-Low)

Die Taster sind Active-Low beschaltet: Ein gedrückter Taster zieht den Eingang auf LOW.

Tabelle 12: Taster-Zustände

Zustand	Pegel	Firmware-Interpretation
Losgelassen	HIGH (1)	nicht gedrückt
Gedrückt	LOW (0)	gedrückt

Die Firmware invertiert dies in `bitops.h`, sodass die Logik-Schicht mit „pressed = trueärbeitet.

3.5 SPI-Bus Verkabelung

Tabelle 13: SPI-Bus Verbindungen

ESP32 Pin	Signal	Verbindung
`D10` (MOSI)	LED-Daten	→ SER (Pin 14) von 74HC595 #0
`D8` (SCK)	Clock	→ SCK/CLK beider Chip-Typen
`D9` (MISO)	Taster-Daten	← Q8 (Pin 3) von CD4021B #0
`D0` (RCK)	LED-Latch	→ RCK (Pin 12) beider 74HC595
`D1` (PS)	Load-Control	→ P/S (Pin 9) beider CD4021B
`D2` (OE)	Output Enable	→ OE (Pin 13) beider 74HC595

3.6 Timing-Diagramme

3.6.1 CD4021B Lesevorgang

Der Lesevorgang beginnt mit einem Parallel-Load-Impuls (P/S: HIGH→LOW→HIGH). Danach liegt das erste Bit bereits am Q8-Ausgang, *bevor* der erste Clock kommt.

First-Bit-Problem

Nach dem Parallel-Load liegt PI-1 (das MSB) sofort am Q8-Ausgang. SPI samplet aber erst nach der ersten Clock-Flanke. Die Firmware löst dies durch einen `digitalRead()` vor dem SPI-Transfer.

3.6.2 74HC595 Schreibvorgang

Daten werden bei steigender SCK-Flanke ins Schieberegister übernommen. Der RCK-Impuls (Latch) überträgt die Daten auf die Ausgänge.

3.7 Stromlimits

Tabelle 14: Stromlimits der Komponenten

Komponente	Parameter	Wert	Bemerkung
ESP32-S3	GPIO Output (max)	40 mA	Drive Strength 3
	GPIO Output (default)	20 mA	Drive Strength 2
74HC595	Output pro Pin (max)	±35 mA	Absolutes Maximum
	Output pro Pin (empf.)	±6 mA	Dauerbetrieb
	VCC/GND gesamt	70 mA–75 mA	**Package-Limit!**
CD4021B	Input (pro Pin)	$< 1\,\mu$A	CMOS-Eingang

3.7.1 Stromversorgung

Tabelle 15: Stromaufnahme des Systems

Komponente	Typisch	Maximum
ESP32-S3	80 mA	500 mA (WiFi)
CD4021B (×2)	< 1 mA	1 mA
74HC595 (×2)	< 1 mA	70 mA
LEDs (×10 @ 4 mA)	40 mA	40 mA
Gesamt	**∼130 mA**	**∼620 mA**

USB-Versorgung

Die USB-CDC Versorgung vom Pi liefert bis zu 500 mA. Bei mehr als 8 LEDs gleichzeitig oder höheren Strömen: Helligkeit per PWM reduzieren oder externe 5V-Versorgung verwenden.

3.8 Stückliste (BOM)

Tabelle 16: Stückliste für den 10-Button-Prototyp

Pos	Komponente	Wert/Typ	Anz.	Bemerkung
1	XIAO ESP32-S3	Seeed Studio	1	Mikrocontroller
2	Raspberry Pi 5	4GB / 8GB	1	Mit Netzteil + microSD
3	CD4021B	DIP-16	2	PISO Schieberegister
4	74HC595	DIP-16	2	SIPO Schieberegister
5	Taster	6×6 mm	10	Tactile Switch
6	LED 5mm	Diverse Farben	10	2× weiß, blau, rot, gelb, grün
7	Widerstand	330 Ω–3 kΩ	10	LED-Vorwiderstand
8	Widerstand	10 kΩ	10	Pull-up für Taster
9	Kondensator	100 nF	4	Keramik, Stützkondensatoren
10	Lochrasterplatine	100×160 mm	1	Oder Breadboard
11	USB-C Kabel	Daten-fähig	1	ESP32 ↔ Pi

3.9 Hardware-Eigenheiten

3.9.1 First-Bit-Problem (CD4021B)

Nach dem Parallel-Load liegt das erste Bit (PI-1 → Q8) sofort am Ausgang, **bevor** der erste Clock kommt. Der ESP32 samplet aber erst **nach** der ersten Clock-Flanke. Die Firmware löst dies durch einen `digitalRead()` vor dem SPI-Transfer.

3.9.2 SPI-Bus Crosstalk

Wenn der ESP32 vom CD4021B liest, taktet er dabei Nullen durch den 74HC595. Die Ausgänge ändern sich erst beim Latch-Impuls, aber ein glitchender RCK-Pin könnte LEDs kurz ausschalten. Die Firmware kompensiert dies durch `LED_REFRESH_EVERY_CYCLE = true`.

3.9.3 Stützkondensatoren

Jeder IC benötigt einen 100 nF Keramikkondensator zwischen VCC und GND, möglichst nah am Chip. Ohne diese Kondensatoren können Störungen auf der Versorgung zu Fehlfunktionen führen.

3.10 Skalierung auf 100 Buttons

Für das 100-Button-System werden jeweils 13 Schieberegister in Daisy-Chain benötigt.

Tabelle 17: Komponentenbedarf: 10 vs. 100 Buttons

Komponente	10-Button	100-Button
CD4021B	2	13
74HC595	2	13
Taster	10	100
LEDs	10	100
R (LED)	10	100
R (Pull-up)	10	100
C (100 nF)	4	26

Timing-Budget

Die SPI-Transferzeit steigt von ∼20 µs auf ∼260 µs – weit unter dem 5 ms-Zyklus. Die Hardware skaliert problemlos.

4 Schieberegister: 74HC595 und CD4021B

4.1 Motivation: Warum Schieberegister?

Ein ESP32-S3 soll 100 LEDs steuern und 100 Taster einlesen. Direkt verdrahtet bräuchten wir 200 GPIO-Pins – doch der ESP32-S3 bietet nur etwa 20 nutzbare. Wie lösen wir dieses Missverhältnis?

Die Antwort liegt in der **seriellen Kommunikation**: Wir wandeln parallele Signale in serielle um und umgekehrt. Die Analogie: Statt 100 LKWs gleichzeitig durch eine enge Straße zu schicken (100 Spuren nötig), reihen wir die Container hintereinander auf einem Gleis auf. Am Ziel verteilen wir sie wieder parallel.

Tabelle 18: Übersicht der Schieberegister-Bausteine

Baustein	Richtung	Typ	Funktion
74HC595	MCU → Außenwelt	SIPO	Serial-In, Parallel-Out
CD4021B	Außenwelt → MCU	PISO	Parallel-In, Serial-Out

4.2 Der 74HC595 – Daten hinausschreiben

4.2.1 Architektur

Werfen wir einen Blick auf das Blockdiagramm in Abbildung 3. Wir erkennen drei Stufen, die wie ein Staffellauf funktionieren:

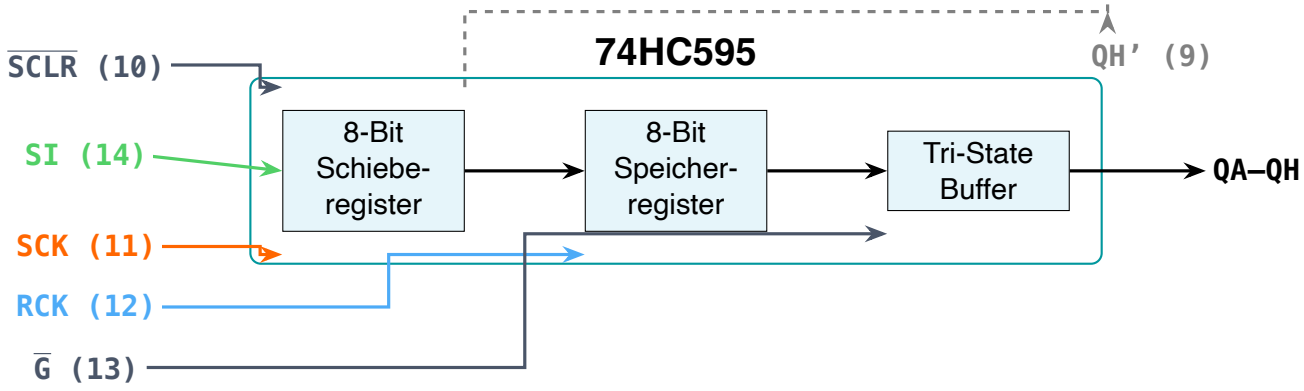


Abbildung 3: Blockdiagramm des 74HC595

Das Prinzip dahinter: Daten wandern Bit für Bit in das Schieberegister. Erst wenn alle 8 Bits angekommen sind, kopiert ein Latch-Impuls den gesamten Inhalt ins Speicher-

register – und die Ausgänge ändern sich *gleichzeitig*. Diese Entkopplung verhindert Flackern während des Schiebens.

4.2.2 Signalbeschreibung

Welche Signale brauchen wir, um den 74HC595 anzusteuern? Tabelle 19 zeigt die wichtigsten Pins:

Tabelle 19: 74HC595 Pinbelegung und Funktion

Pin	Symbol	Funktion	Aktiv	Beschreibung
14	SI	Serial Input	–	Serieller Dateneingang
11	SCK	Shift Clock	↑	Positive Flanke übernimmt SI
12	RCK	Register Clock	↑	Kopiert Schieberegister → Speicher
10	$\overline{\text{SCLR}}$	Shift Clear	LOW	Löscht das Schieberegister
13	$\overline{\text{G}}$	Output Enable	LOW	Aktiviert die Ausgänge
9	QH′	Serial Out	–	Für Kaskadierung

4.2.3 Timing-Sequenz

Doch wie sieht das Zusammenspiel dieser Signale konkret aus? Um das Muster `0b10110001` auszugeben, durchlaufen wir folgende Sequenz – das Timing-Diagramm in Abbildung 4 macht den Ablauf sichtbar:

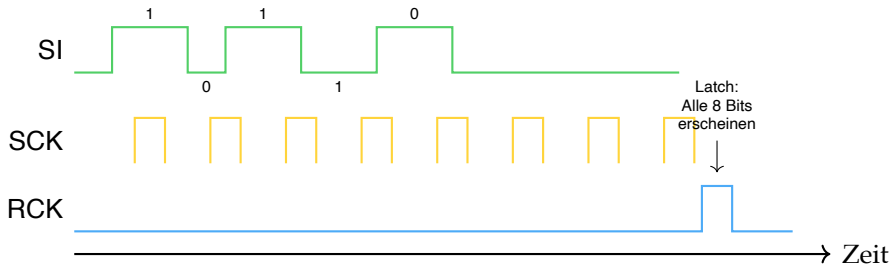


Abbildung 4: Timing-Diagramm: Daten in den 74HC595 schieben

Kritischer Punkt

Die Ausgänge ändern sich erst bei der RCK-Flanke – nicht während des Schiebens. Das verhindert „Geisterbilder“ auf den LEDs.

4.2.4 Kaskadierung

Was aber, wenn 8 Ausgänge nicht reichen? Der `QH'`-Ausgang (Pin 9) liefert das „herausgeschobene“ Bit. Verbinden wir `QH'` des ersten ICs mit `SI` des zweiten, entsteht eine 16-Bit-Kette. Abbildung 5 zeigt das Prinzip:

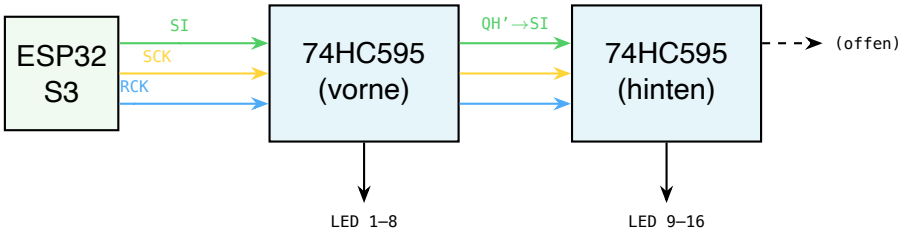


Abbildung 5: Kaskadierung zweier 74HC595

Reihenfolge beachten

Wir schieben zuerst die Bits für das **hintere** IC, dann für das vordere. Das erste Bit „rutscht“ durch bis zum Ende der Kette.

4.3 Der CD4021B – Daten hereinlesen

4.3.1 Architektur

Der CD4021B arbeitet spiegelverkehrt zum 74HC595: Er liest 8 parallele Eingänge gleichzeitig ein und schiebt sie seriell heraus. Das Blockdiagramm in Abbildung 6 verdeutlicht den zweistufigen Aufbau:

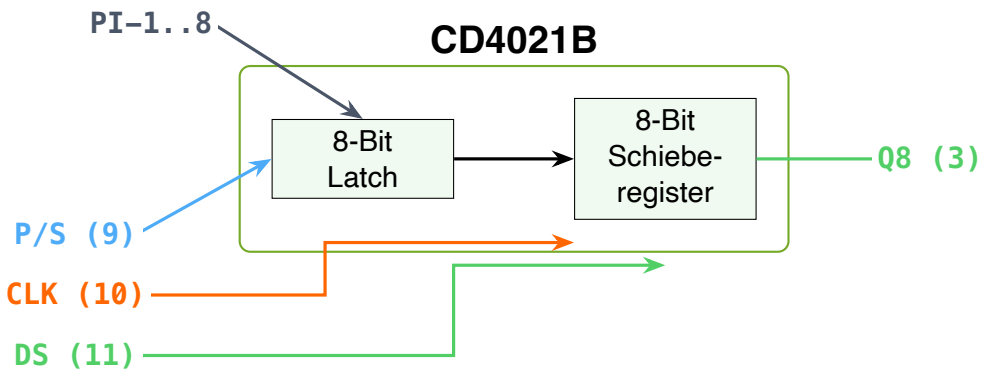


Abbildung 6: Blockdiagramm des CD4021B

4.3.2 Signalbeschreibung

Tabelle 20: CD4021B Pinbelegung und Funktion

Pin	Symbol	Funktion	Aktiv	Beschreibung
1,4–7,13–15	PI-8..1	Parallel Inputs	–	8 parallele Eingänge
9	P/S	Parallel/Serial	HIGH	HIGH = Load, LOW = Shift
10	CLK	Clock	↑	Positive Flanke schiebt
11	DS	Serial Input	–	Für Kaskadierung
3	Q8	Serial Output	–	Serieller Datenausgang

Invertierte Logik!

Der CD4021B hat eine **invertierte** Load-Logik: P/S = HIGH für Parallel Load, P/S = LOW für Serial Shift. Das ist das Gegenteil dessen, was man intuitiv erwarten würde!

4.3.3 Timing-Sequenz und First-Bit-Problem

Beim CD4021B liegt nach dem Parallel Load bereits das erste Bit (PI-1, MSB) an Q8 an – **bevor** wir den ersten Clock-Puls geben. Abbildung 7 zeigt diesen kritischen Ablauf:

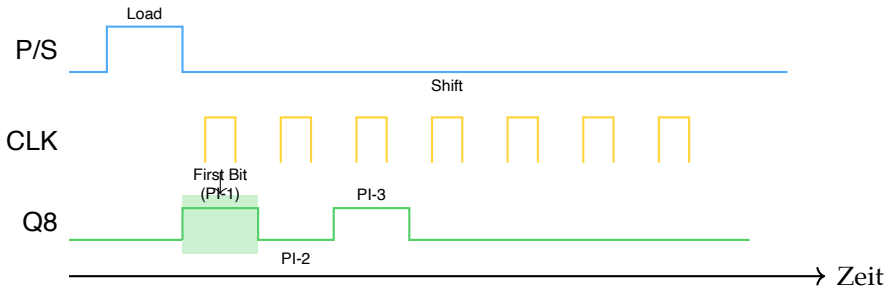


Abbildung 7: CD4021B Timing: First Bit liegt sofort nach Load an Q8

First-Bit-Rescue

Das erste Bit muss **vor** dem SPI-Transfer gelesen werden! Nach dem Parallel Load (P/S = HIGH → LOW) liegt PI-1 bereits an Q8. Der erste Clock-Puls würde es „wegschieben“, bevor wir es lesen können.

4.3.4 Bit-Reihenfolge: MSB-first

Der CD4021B gibt die Bits in MSB-first-Reihenfolge aus:

- Nach Load: PI-1 (MSB) an Q8
- Nach 1. Clock: PI-2 an Q8
- Nach 7. Clock: PI-8 (LSB) an Q8

Hardware-Verdrahtung beachten

Die physische Verdrahtung bestimmt das Bit-Mapping:

- **BTN 1** → PI-8 (Pin 1) → erscheint als Bit 0 im Datenstrom
- **BTN 8** → PI-1 (Pin 7) → erscheint als Bit 7 im Datenstrom

Die Firmware abstrahiert dies mit: `btn_bit(id) = 7 - ((id - 1) % 8)`

4.3.5 Kaskadierung

Für mehr als 8 Eingänge verbinden wir `Q8` des hinteren ICs mit `DS` des vorderen:

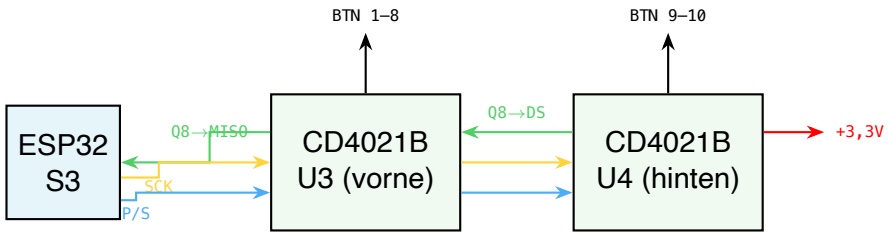


Abbildung 8: Kaskadierung: CD4021B U4 (hinten) → U3 (vorne) → ESP32

DS des letzten ICs auf VCC!

Der `DS`-Eingang des letzten ICs in der Kette **muss** auf +3,3V (VCC) gelegt werden! CMOS-Eingänge dürfen niemals floaten – das führt zu zufälligen Werten und erhöhtem Stromverbrauch.

4.4 Firmware-Implementierung

4.4.1 ESP32-S3 Pinbelegung

Wir nutzen einen gemeinsamen SPI-Bus für beide IC-Typen, mit separaten Steuerleitungen:

Tabelle 21: ESP32-S3 XIAO Pinbelegung

Signal	Pin	Ziel	SPI-Mode
MOSI	D10	74HC595 SER (Pin 14)	MODE0, 1 MHz
SCK	D8	Alle CLK (gemeinsam)	–
RCK	D0	74HC595 RCLK (Pin 12)	–
OE	D2	74HC595 OE (Pin 13)	PWM optional
MISO	D9	CD4021B Q8 (Pin 3)	MODE1, 500 kHz
P/S	D1	CD4021B P/S (Pin 9)	–

4.4.2 CD4021B mit First-Bit-Rescue

```
uint16_t cd4021_read(uint8_t* buffer, size_t numBytes) {
    // 1) Parallel Load: P/S HIGH -> LOW
    digitalWrite(BTN_LOAD_PIN, HIGH);
    delayMicroseconds(2);
    digitalWrite(BTN_LOAD_PIN, LOW);
    delayMicroseconds(2);

    // 2) First Bit Rescue: PI-1 liegt bereits an Q8!
    uint8_t firstBit = digitalRead(BTN_DATA_PIN);

    // 3) SPI Transfer (restliche Bits)
    SPI.beginTransaction(SPISettings(500000, MSBFIRST,
       SPI_MODE1));
    SPI.transfer(buffer, numBytes);
    SPI.endTransaction();

    // 4) First Bit einsetzen (MSB von Byte 0)
    buffer[0] = (buffer[0] >> 1) | (firstBit << 7);

    return combineBytes(buffer, numBytes);
}
```

Listing 1: CD4021B First-Bit-Rescue

Die kritische Stelle

Wir lesen **vor** dem SPI-Transfer, nicht danach. Nach dem Parallel Load liegt das erste Bit bereits an `Q8`. Der SPI-Clock würde das nächste Bit herbei schieben und das erste Bit wäre verloren.

4.4.3 Bit-Mapping: Hardware-Abstraktion

```
// Firmware-Abstraktion: btn_bit(id) = 7 - ((id - 1) % 8)
// BTN 1 -> PI-8 (Pin 1) -> Bit 0 im Datenstrom
// BTN 8 -> PI-1 (Pin 7) -> Bit 7 im Datenstrom

inline bool isButtonPressed(uint16_t stream, uint8_t btnId) {
    uint8_t byteIdx = (btnId - 1) / 8;
    uint8_t bitPos = 7 - ((btnId - 1) % 8);
    uint8_t mask = 1 << bitPos;

    // Active-Low: gedrückt = 0
    return ((stream >> (byteIdx * 8)) & mask) == 0;
}
```

Listing 2: Mapping physisch → logisch

Warum ein Mapping? Die physische Verdrahtung (BTN 1 → PI-8) entspricht nicht der logischen Nummerierung. Statt im Code zu rechnen, definieren wir eine klare Formel. Bei Verdrahtungsänderungen passen wir nur diese Formel an – der restliche Code bleibt unberührt.

4.4.4 Debouncing

Mechanische Taster prellen – sie schalten beim Drücken mehrfach ein und aus (typisch 5 ms bis 20 ms). Die Lösung: Wir warten, bis der Zustand **stabil** bleibt.

```
inline void buttonsTask(uint32_t nowMs) {
    const uint16_t raw = readButtonsPressedMask();

    // Rohaenderung: Zeitpunkt merken
    if (raw != g_btnRaw) {
        g_btnRaw = raw;
        g_btnChangedAt = nowMs;  // Timer neu starten
```

```
    }

    // Wenn lange genug stabil -> uebernehmen
    if ((nowMs - g_btnChangedAt) >= DEBOUNCE_MS &&
        g_btnStable != g_btnRaw) {
        const uint16_t prev = g_btnStable;
        g_btnStable = g_btnRaw;

        // Rising Edge: 0 -> 1 (Taste wurde gedrueckt)
        const uint16_t rising = (g_btnStable & ~prev);
        if (rising) {
            // Toggle-Logik hier...
        }
    }
}
```

Listing 3: Debouncing mit Flankenerkennung

4.5 Timing-Analyse

Funktioniert unser Code schnell genug? Schauen wir uns die Zahlen an.

4.5.1 Sind unsere Delays ausreichend?

Tabelle 22: Timing-Parameter CD4021B (bei 5 V)

Parameter	Symbol	Min	Unser Delay	Status
P/S Pulse Width	t_{WH}	80 ns	2000 ns	✓
P/S Removal Time	t_{REM}	140 ns	2000 ns	✓
Clock Pulse Width	t_W	40 ns	1000 ns	✓

Bei 3,3 V (statt 5 V) verlängern sich alle Zeiten um etwa Faktor 2. Unsere Delays haben einen Sicherheitsfaktor von ca. 25× – mehr als ausreichend.

4.5.2 Gesamtzeit für einen Durchlauf

$$t_{\text{Load}} = 2\,\mu\text{s} + 2\,\mu\text{s} = 4\,\mu\text{s} \tag{2}$$

$$t_{\text{SPI}} = 16\ \text{Bits} \times \frac{1}{500\,\text{kHz}} = 32\,\mu\text{s} \tag{3}$$

$$t_{\text{Summe}} \approx 40\,\mu\text{s pro Scan (Hardware-SPI)} \tag{4}$$

Bei `POLL_MS` $= 5\,$ms haben wir ca. $4960\,\mu$s Reserve – das System ist weit vom Limit entfernt.

4.6 Skalierung auf 100×

Für 100 LEDs und 100 Taster brauchen wir 13 ICs je Typ ($13 \times 8 = 104 \geq 100$). Doch was ändert sich im Code?

4.6.1 Änderungen im Code

```
// Vorher (10x)
constexpr uint8_t NUM_LEDS = 10;
constexpr uint8_t NUM_SHIFT_BYTES = 2;
static uint8_t g_ledState[NUM_SHIFT_BYTES] = {0};

// Nachher (100x)
constexpr uint8_t NUM_LEDS = 100;
constexpr uint8_t NUM_SHIFT_BYTES = 13;  // Aufrunden: (100+7)/8
static uint8_t g_ledState[NUM_SHIFT_BYTES] = {0};
```

Listing 4: Skalierung auf 100×

4.6.2 Transfer-Zeiten bei 100×

Tabelle 23: Geschwindigkeitsvergleich: 10× vs. 100×

Komponente	**10× (16 Bits)**	**100× (104 Bits)**
74HC595 @ 1 MHz	≈16 μs	≈104 μs
CD4021B @ 500 kHz	≈32 μs	≈208 μs
Gesamt	≈50 μs	≈320 μs

Selbst bei 100× bleibt die Scan-Zeit unter 500 μs – bei einem 5 ms-Polling-Intervall haben wir noch 90% Reserve.

Hardware-SPI ist Pflicht

Bei 100× ist Hardware-SPI Pflicht. Bit-Banging würde ca. 16× länger dauern und wäre nicht mehr praktikabel.

4.7 Zusammenfassung

Die Kombination aus 74HC595 (Ausgänge) und CD4021B (Eingänge) löst das Pin-Multiplex-Problem elegant. Tabelle 24 fasst den Vergleich zwischen dem aktuellen 10×-System und dem Ziel-System mit 100× zusammen:

Tabelle 24: Vergleich: 10× vs. 100× System

Aspekt	10× (aktuell)	100× (Ziel)	Änderung
GPIO-Pins	6	6	keine
ICs	4	26	+22
Scan-Zeit	≈50 μs	≈320 μs	6×
Code-Änderungen	–	Konstanten + Arrays	minimal

Das Grundprinzip bleibt identisch – nur die Skalierung ändert sich. Genau das macht gutes Hardware-Abstraktions-Design aus: Die Architektur skaliert, ohne dass wir das Rad neu erfinden müssen.

5 Architektur-Übersicht

Das Selection Panel verbindet 10 Taster und 10 LEDs über einen ESP32-S3 mit einem Raspberry Pi 5. Wie arbeiten die Komponenten zusammen? Der Pi übernimmt die Multimedia-Wiedergabe, während der ESP32 die zeitkritische I/O-Verarbeitung in Echtzeit bewältigt.

Tabelle 25: Metadaten der Architektur

Metadaten	Wert
Version	2.5.2
Phase	7 (Raspberry Pi Integration)
Stand	2026-01-08

5.1 Systemkontext

Figure 9 zeigt den Gesamtaufbau des Systems. Die Kommunikation zwischen Pi und ESP32 erfolgt über USB-CDC mit 115 200 baud.

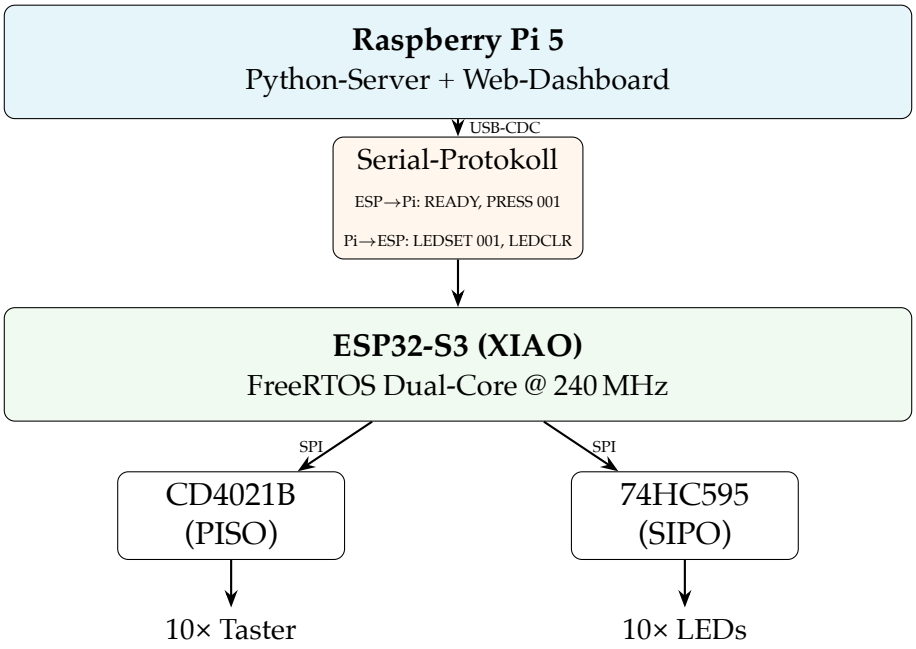


Abbildung 9: Systemkontext des Selection Panels

5.2 Schichtenmodell

Die Firmware folgt einem strikten Schichtenmodell. Jede Schicht kennt nur die darunterliegende – das erleichtert Tests und Wartung erheblich.

Tabelle 26: Firmware-Schichten und ihre Verantwortlichkeiten

Schicht	Verzeichnis	Verantwortung
Entry	`main.cpp`	Queue erstellen, Tasks starten
App	`src/app/`	FreeRTOS Tasks (io_task, serial_task)
Logic	`src/logic/`	Debouncing, Selection-Logik
Driver	`src/drivers/`	CD4021B, 74HC595 Ansteuerung
HAL	`src/hal/`	SPI-Bus Abstraktion mit Mutex

5.3 Datenfluss

Was passiert, wenn wir einen Taster drücken? Der Datenfluss durchläuft mehrere Stationen:

1. **Hardware → Driver:** Der CD4021B liest den Tasterzustand per Parallel Load ein
2. **Driver → Logic:** `readRaw()` liefert ein 2-Byte-Array mit den Rohwerten
3. **Logic (Debouncer):** Filtert Prellen heraus, liefert stabile Zustände
4. **Logic (Selection):** Ermittelt die aktive ID nach „Last Press Wins"
5. **App → Serial:** Ein `LogEvent` wird in die Queue geschrieben
6. **Serial → Pi:** Die Nachricht `PRESS 001\n` geht über USB

Parallel dazu aktualisiert die Logic-Schicht das LED-Array, das der 74HC595 ausgibt.

5.4 Timing

Die zeitlichen Zusammenhänge verdeutlichen, warum wir mit einem 5 ms-Zyklus arbeiten:

Tabelle 27: Timing-Parameter des IO-Tasks

Operation	Dauer	Beschreibung
CD4021B Read	∼20 µs	SPI @ 500 kHz
74HC595 Write	∼16 µs	SPI @ 1 MHz
Debounce-Zeit	30 ms	Pro Taster, zeitbasiert
IO-Periode	5 ms	Abtastrate 200 Hz

Timing-Budget

Bei 5 ms Zykluszeit und ∼40 µs SPI-Transferzeit bleiben 99,2% der CPU-Zeit für andere Tasks. Das System ist nicht annähernd ausgelastet.

5.5 Gemeinsamer SPI-Bus

CD4021B und 74HC595 teilen sich einen SPI-Bus, verwenden aber unterschiedliche Modi. Der `SpiGuard` (RAII-Pattern) stellt sicher, dass Transaktionen korrekt beendet werden.

Tabelle 28: SPI-Konfiguration der Schieberegister

Chip	Funktion	SPI-Modus	Takt	Bit-Ordnung
CD4021B	Taster einlesen	MODE1 (CPOL=0, CPHA=1)	500 kHz	MSB-first
74HC595	LEDs ansteuern	MODE0 (CPOL=0, CPHA=0)	1 MHz	MSB-first

```
{
    SpiGuard g(bus, settings);  // lock() + beginTransaction()
    SPI.transfer(data);
}  // Automatisch: endTransaction() + unlock()
```

Listing 5: SpiGuard garantiert saubere Transaktionen

5.6 Bit-Adressierung

Die Hardware-Verdrahtung bestimmt das Bit-Mapping. Dank korrekter Verkabelung ist die Formel für beide Chips identisch:

$$\text{byte}(id) = \lfloor (id-1)/8 \rfloor, \quad \text{bit}(id) = (id-1) \mod 8 \tag{5}$$

5.6.1 CD4021B (Taster)

BTN 1 liegt an PI-8 (Pin 1), BTN 8 an PI-1 (Pin 7):

Tabelle 29: Bit-Mapping der Taster

Taster-ID	CD4021B PI	Byte	Bit	Formel
1	PI-8	0	0	$(1-1) \mod 8 = 0$
2	PI-7	0	1	
8	PI-1	0	7	
9	PI-8	1	0	
10	PI-7	1	1	

5.6.2 74HC595 (LEDs)

LED 1 liegt an QA, LED 8 an QH:

Tabelle 30: Bit-Mapping der LEDs

LED-ID	74HC595 Q	Byte	Bit	Formel
1	QA	0	0	$(1-1) \mod 8 = 0$
2	QB	0	1	
8	QH	0	7	
9	QA	1	0	
10	QB	1	1	

```
// Identische Formel fuer beide Chips dank korrekter Verdrahtung
static inline uint8_t btn_byte(uint8_t id) { return (id - 1) / 8;
    }
static inline uint8_t btn_bit(uint8_t id)  { return (id - 1) % 8;
    }

static inline uint8_t led_byte(uint8_t id) { return (id - 1) / 8;
    }
static inline uint8_t led_bit(uint8_t id)  { return (id - 1) % 8;
    }
```

Listing 6: Bit-Operationen in bitops.h

5.7 First-Bit-Problem (CD4021B)

Nach dem Parallel-Load liegt PI-1 (das MSB, also BTN 8) sofort am Q8-Ausgang – bevor der erste Clock kommt. SPI samplet aber erst nach der ersten Flanke. Das erste Bit geht verloren!

Die Lösung: Wir lesen das erste Bit vor dem SPI-Transfer per `digitalRead()` und setzen es anschließend ein:

```
void Cd4021::readRaw(SpiBus& bus, uint8_t* out) {
    // 1. Parallel Load
    digitalWrite(PIN_BTN_PS, HIGH);
    delayMicroseconds(2);
    digitalWrite(PIN_BTN_PS, LOW);
    delayMicroseconds(2);

    // 2. First Bit Rescue: PI-1 liegt bereits an Q8!
    uint8_t firstBit = digitalRead(PIN_BTN_MISO);

    // 3. SPI Transfer (restliche Bits)
    SpiGuard g(bus, spi_);
    SPI.transfer(out, BTN_BYTES);

    // 4. First Bit einsetzen (MSB von Byte 0)
    out[0] = (out[0] >> 1) | (firstBit << 7);
}
```

Listing 7: First-Bit-Rescue im CD4021B-Treiber

Timing beachten

Der Load-Puls muss mindestens 2 μs dauern (CMOS-Anforderung). Zwischen Load und Read darf keine zu lange Pause liegen, sonst driftet der Zustand.

5.8 Zeitbasiertes Debouncing

Mechanische Taster prellen – ein einzelner Druck erzeugt mehrere Flanken. Unser Debouncer arbeitet zeitbasiert: Jeder Taster hat einen eigenen Timer. Bei Änderung des Rohwerts wird der Timer zurückgesetzt. Erst wenn der Timer 30 ms abgelaufen ist und der Rohwert stabil bleibt, wird der entprellte Zustand übernommen.

Vorteile des zeitbasierten Ansatzes

Diese Methode ist unabhängig von der Abtastrate und erlaubt schnelle Tastenfolgen. Ein Counter-basierter Ansatz würde bei höherer Abtastrate längere Entprellzeiten erzeugen.

5.9 Selection-Logik

Die Selection-Logik folgt dem **Last Press Wins**-Prinzip: Wird ein neuer Taster gedrückt, überschreibt er die vorherige Auswahl sofort. Mit `LATCH_SELECTION=true` bleibt die Auswahl nach Loslassen bestehen – die LED leuchtet weiter, bis ein neuer Taster gedrückt wird oder das Audio endet.

5.10 Konfigurationsparameter

Die wichtigsten Parameter finden sich in `config.h`:

Tabelle 31: Konfigurationsparameter der Firmware

Parameter	Wert	Beschreibung
`IO_PERIOD_MS`	5	Abtastrate des IO-Tasks (200 Hz)
`DEBOUNCE_MS`	30	Entprellzeit pro Taster
`LATCH_SELECTION`	true	Auswahl bleibt nach Loslassen
`PWM_DUTY_PERCENT`	50	LED-Helligkeit (0–100%)
`LED_REFRESH_EVERY_CYCLE`	true	Kompensiert SPI-Glitches
`SERIAL_PROTOCOL_ONLY`	true	Nur Protokoll, keine Debug-Logs

5.11 Pin-Zuordnung

Tabelle 32: Pin-Zuordnung ESP32-S3 XIAO

Pin	Funktion	Chip	Signal
`D10`	MOSI	74HC595	SER (Data In)
`D8`	SCK	Beide	SRCLK / CLK (shared)
`D0`	RCK	74HC595	RCLK (Latch)
`D2`	OE	74HC595	Output Enable (PWM)
`D9`	MISO	CD4021B	Q8 (Data Out)
`D1`	P/S	CD4021B	Parallel/Serial Load

5.12 Skalierung auf 100 Buttons

Die Architektur ist für 100 Taster/LEDs vorbereitet. Was müssen wir ändern?

1. `BTN_COUNT` und `LED_COUNT` auf 100 setzen
2. `BTN_BYTES` und `LED_BYTES` werden automatisch auf 13 berechnet
3. Zusätzliche Schieberegister in Daisy-Chain verkabeln
4. Die Logic-Schicht skaliert automatisch (Bit-Arrays)

Tabelle 33: Skalierungsverhalten der SPI-Transfers

Konfiguration	SPI-Transferzeit	Budget (5 ms)
10 Buttons	~40 µs	0,8%
100 Buttons	~320 µs	6,4%

Reichlich Reserven

Selbst mit 100 Tastern bleiben über 93% des Timing-Budgets frei. Die Architektur ist nicht der limitierende Faktor – die Hardware-Verdrahtung und das Löten von 100 Tastern sind die eigentliche Herausforderung. [1]

6 Spezifikation (SPEC)

Dieses Kapitel bildet die **Single Source of Truth** für Protokolle, Pinbelegung und Policies des Auswahlpanels. Wenn wir uns fragen, wie etwas funktionieren soll, finden wir hier die verbindliche Antwort.

Tabelle 34: Metadaten der Spezifikation

Metadaten	Wert
Version	2.5.2
Datum	2026-01-08
Status	✓ Prototyp funktionsfähig (10×)

6.1 Glossar

Bevor wir in die Details einsteigen, klären wir die wichtigsten Begriffe. Table 35 dient als Nachschlagewerk.

Tabelle 35: Begriffsdefinitionen

Begriff	Erklärung
One-hot	Genau ein Bit ist aktiv, alle anderen sind aus
Preempt	Neue Aktion unterbricht sofort die laufende
Race-Condition	Timing-Problem, wenn zwei Ereignisse fast gleichzeitig auftreten
FreeRTOS	Echtzeit-Betriebssystem für Mikrocontroller
Mutex	Sperre, die gleichzeitigen Zugriff auf Ressourcen verhindert
CMOS	Chip-Technologie – Eingänge nie unbeschaltet lassen
Pull-Up	Widerstand zieht Signal auf HIGH, Taster zieht auf LOW
Kaskadierung	ICs in Reihe schalten, um mehr Ein-/Ausgänge zu erhalten
DIP-16	IC-Gehäuse mit 16 Pins in zwei Reihen
USB-CDC	USB Communications Device Class (virtueller COM-Port)
1-basiert	Nummerierung beginnt bei 1 (nicht 0)
Preloading	Medien vorladen bevor sie benötigt werden

6.2 Policy

Zwei zentrale Regeln bestimmen das Verhalten des Systems: One-hot LED und Preempt.

6.2.1 One-hot LED

Zu jedem Zeitpunkt leuchtet **maximal eine LED**. Diese Einschränkung hat einen praktischen Grund: Der Maximalstrom beträgt so nur $1 \times 20\,\text{mA}$ statt $100 \times 20\,\text{mA} = 2\,\text{A}$.

Tabelle 36: LED-Befehle im One-hot-Modus

Befehl	Wirkung
`LEDSET n`	LED *n* an, alle anderen aus
`LEDCLR`	Alle LEDs aus

6.2.2 Preempt („Umschalten gewinnt")

Jeder neue Tastendruck unterbricht sofort die aktuelle Wiedergabe. Der Ablauf mit ESP32 v2.5.2 (lokale LED-Steuerung):

1. ESP32 setzt LED sofort (< 1 ms)
2. ESP32 → Pi: `PRESS 005`
3. Pi → Browser: `{"type:ßtop"}` + `{"type:"play",ïd:5}` (parallel)
4. Browser spielt aus Cache (< 50 ms)

Race-Condition-Schutz

Nach Audio-Ende meldet der Browser `{"type:ënded",ïd:5}`. Der Pi sendet `LEDCLR` **nur wenn** `id == current_id`. So bleibt die LED an, falls zwischenzeitlich ein neuer Taster gedrückt wurde.

6.3 Nummerierung (1-basiert)

Alle IDs sind 1-basiert (001–100), nicht 0-basiert (000–099). Diese Konvention zieht sich durch alle Schichten – keine Konvertierung nötig.

Tabelle 37: Durchgängig 1-basierte Nummerierung

Schicht	Format	Beispiel
Taster (physisch)	1–100	Taster 1, Taster 10
Serial-Protokoll	001–100	`PRESS 001`, `LEDSET 010`
WebSocket	1–100	`{"type:"play",ïd:1}`
Medien-Dateien	001–100	`001.jpg`, `010.mp3`
Dashboard-Anzeige	001–100	„001", „010"

6.4 Pinbelegung ESP32-S3 XIAO

Table 38 zeigt die Verbindungen zwischen ESP32 und den Schieberegistern.

Tabelle 38: ESP32-S3 XIAO Pinbelegung

Signal	Pin	Ziel-IC	Funktion
MOSI	`D10`	74HC595 Pin 14 (SER)	Serielle Daten für LEDs
SCK	`D8`	74HC595 Pin 11 + CD4021B Pin 10	Gemeinsamer SPI-Takt
RCK	`D0`	74HC595 Pin 12 (RCLK)	LED-Latch (Ausgabe freigeben)
OE	`D2`	74HC595 Pin 13 (OE)	Output Enable (PWM für Helligkeit)
MISO	`D9`	CD4021B Pin 3 (Q8)	Serielle Daten von Tastern
P/S	`D1`	CD4021B Pin 9 (P/S)	Parallel Load (HIGH = Load)

SPI-Modi beachten

Die beiden ICs verwenden unterschiedliche SPI-Modi:

- **74HC595:** MODE0 (CPOL=0, CPHA=0) @ 1 MHz
- **CD4021B:** MODE1 (CPOL=0, CPHA=1) @ 500 kHz

6.5 CD4021B vs. 74HC165

Warum nutzen wir den CD4021B statt des häufiger verwendeten 74HC165? Table 39 zeigt die Unterschiede.

Tabelle 39: Vergleich der Eingabe-Schieberegister

Aspekt	74HC165	CD4021B
Load-Signal	LOW-aktiv	**HIGH-aktiv**
Shift-Signal	HIGH-aktiv	LOW-aktiv
DIP-Verfügbarkeit	Schwer	Gut
Load-Puls	1 μs	**2 μs** (CMOS)
Clock-Puls	1 μs	1 μs

Firmware-Anpassung

Die invertierte Load-Logik und längeren Pulse des CD4021B sind in der Firmware berücksichtigt. Bei einem Wechsel zum 74HC165 müssten diese Parameter angepasst werden.

6.6 Verdrahtungsregeln

CMOS-ICs vertragen keine offenen Eingänge – das führt zu undefiniertem Verhalten und erhöhtem Stromverbrauch. Table 40 fasst die wichtigsten Regeln zusammen.

Tabelle 40: Verdrahtungsregeln für die Schieberegister

IC	Pin	Regel	Grund
CD4021B (letzter)	Pin 11 (DS)	→ VCC	CMOS-Eingänge nie floaten
74HC595 (letzter)	Pin 9 (QH')	offen OK	Ausgang treibt aktiv
74HC595 (alle)	Pin 10 (SRCLR)	→ VCC	Clear deaktiviert
74HC595 (alle)	Pin 13 (OE)	→ `D2` oder GND	Outputs aktiviert / PWM
Alle ICs	VCC / VDD	100 nF → GND	Abblockkondensator

6.6.1 Bit-Mapping

Die Hardware-Verdrahtung bestimmt, welcher Taster welchem Bit entspricht.

CD4021B (Eingabe): BTN 1 → PI-8 (Pin 1), BTN 8 → PI-1 (Pin 7)

$$\text{btn\_bit}(id) = (id - 1) \mod 8 \tag{6}$$

74HC595 (Ausgabe): LED 1 → QA, LED 8 → QH

$$\text{led\_bit}(id) = (id - 1) \mod 8 \tag{7}$$

6.7 Serial-Protokoll (ESP32 ↔ Pi)

Die Kommunikation erfolgt mit 115 200 baud, ASCII-kodiert und Newline-terminiert (`\n`).

Stabiler Device-Pfad

Statt `/dev/ttyACM0` (kann sich ändern) verwenden wir:
`/dev/serial/by-id/usb-Espressif*`

6.7.1 ESP32 → Pi

Tabelle 41: Nachrichten vom ESP32 zum Pi

Nachricht	Bedeutung
`READY`	Controller bereit
`FW SelectionPanel v2.5.2`	Firmware-Version
`PRESS 001`	Taster 1 gedrückt (001–100)
`RELEASE 001`	Taster 1 losgelassen
`OK`	Befehl erfolgreich
`PONG`	Antwort auf PING
`ERROR msg`	Fehlermeldung

6.7.2 Pi → ESP32

Tabelle 42: Befehle vom Pi zum ESP32

Befehl	Funktion
`LEDSET 001`	LED 1 ein (one-hot)
`LEDON 001`	LED 1 ein (additiv)
`LEDOFF 001`	LED 1 aus
`LEDCLR`	Alle LEDs aus
`LEDALL`	Alle LEDs ein
`PING`	Verbindungstest → PONG
`STATUS`	Zustand abfragen
`VERSION`	Firmware-Version
`HELP`	Befehlsliste

6.7.3 STATUS-Ausgabe (v2.5.2)

```
STATUS active=5 leds=0x10
LEDS 0000100000      # Bit-Vektor (MSB links)
BTNS 1111111111      # Bit-Vektor (Active-Low: 1 = losgelassen)
```

6.7.4 USB-CDC Besonderheit

Der ESP32-S3 XIAO nutzt USB-CDC statt UART. Ohne explizites `flush()` fragmentieren die Nachrichten:

```
Serial.println(buffer);
Serial.flush();  // Wichtig bei USB-CDC!
```

Listing 8: Serial.flush() verhindert Fragmentierung

6.8 WebSocket-Protokoll (Pi ↔ Browser)

Endpoint: `ws://rover:8080/ws`

Tabelle 43: WebSocket-Nachrichten

Richtung	Nachricht	Bedeutung
Pi → Browser	`{"type:ßtop"}`	Wiedergabe stoppen
Pi → Browser	`{"type:"play",ïd:5}`	Medien 5 abspielen
Browser → Pi	`{"type:ënded",ïd:5}`	Audio 5 beendet
Browser → Pi	`{"type:"ping"}`	Heartbeat

6.9 HTTP-Endpoints

Tabelle 44: HTTP-Endpoints des Servers

Pfad	Funktion
`/`	Dashboard (index.html)
`/ws`	WebSocket-Verbindung
`/static/*`	CSS, JavaScript
`/media/*`	Bilder und Audio
`/test/play/{id}`	Wiedergabe simulieren (1-basiert)
`/test/stop`	Wiedergabe stoppen
`/status`	Server-Status (JSON)
`/health`	Health-Check (200/503)

6.9.1 Status-Response (v2.5.2)

```
{
  "version": "2.5.2",
  "mode": "prototype",
  "num_media": 10,
  "current_button": 5,
  "ws_clients": 1,
  "serial_connected": true,
  "serial_port": "/dev/serial/by-id/usb-Espressif...",
  "media_missing": 0,
  "esp32_local_led": true
}
```

6.10 Medien-Konvention

Tabelle 45: Zuordnung von IDs zu Mediendateien

ID	Bild	Audio
1	media/001.jpg	media/001.mp3
5	media/005.jpg	media/005.mp3
10	media/010.jpg	media/010.mp3
100	media/100.jpg	media/100.mp3

Dateinamen-Konvention

IDs: 1–100 (1-basiert), Dateien: 001–100 (zero-padded, 3 Stellen).

6.11 Latenz-Budget

Wie schnell reagiert das System auf einen Tastendruck? Table 46 schlüsselt die einzelnen Komponenten auf.

Tabelle 46: Latenz-Budget vom Tastendruck bis zur Wiedergabe

Komponente	Latenz	Beschreibung
ESP32 LED	< 1 ms	Lokale Steuerung (v2.5.2)
Serial	~ 5 ms	USB-CDC Übertragung
Server	~ 1 ms	asyncio.gather
WebSocket	~ 5 ms	Netzwerk
Dashboard	< 50 ms	Aus Cache (Preloading)
Gesamt	**< 70 ms**	Tastendruck → Wiedergabe

6.12 Akzeptanztests

Tabelle 47: Akzeptanztests für den Prototyp

Test	Erwartung	Status
Preempt	Neuer Taster unterbricht sofort	✓
One-hot	Nur eine LED leuchtet	✓
Ende	Nach Audio: alle LEDs aus	✓
Race	LED bleibt an wenn neue ID aktiv	✓
Debounce	Nur ein Event pro Tastendruck	✓
Zuordnung	Taster 1 → Medien 001	✓
Alle Taster	10/10 erkannt (Prototyp)	✓
LED-Latenz	< 1 ms	✓
Dashboard-Latenz	< 50 ms	✓

6.13 Versionen

Tabelle 48: Aktuelle Komponenten-Versionen

Komponente	Version	Änderung
Firmware	2.5.2	FreeRTOS, Hardware-SPI, First-Bit-Rescue
Server	2.5.2	by-id Pfad, asyncio, ESP32_SETS_LED_LOCALLY
Dashboard	2.5.1	Preloading, Cache, Audio-Unlock

6.14 Bekannte Einschränkungen

Table 49 dokumentiert bekannte Probleme und deren Lösungen.

Tabelle 49: Bekannte Einschränkungen und Lösungen

Problem	Status	Lösung
ESP32-S3 USB-CDC fragmentiert	✓	`Serial.flush()`
pyserial funktioniert nicht	✓	`os.open` + `stty`
CD4021B braucht längere Pulse	✓	2 μs Load, 1 μs Clock
CD4021B First-Bit-Problem	✓	`digitalRead()` vor SPI
LED-Latenz durch Roundtrip	✓	Lokale LED-Steuerung
Dashboard-Latenz	✓	Medien-Preloading
Serial-Pfad instabil	✓	by-id Pfad verwenden

7 Protokoll-Referenz

Das Selection Panel verwendet zwei Protokolle: Ein zeilenbasiertes Serial-Protokoll zwischen ESP32 und Pi, sowie ein JSON-basiertes WebSocket-Protokoll zwischen Server und Browser. Schauen wir uns beide im Detail an.

7.1 Übersicht

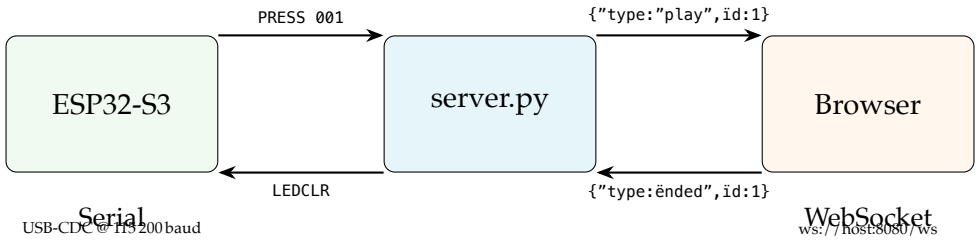


Abbildung 10: Protokoll-Übersicht: Datenfluss zwischen den Komponenten

7.2 Verbindungsaufbau

7.2.1 Serial (ESP32 ↔ Pi)

Nach dem Start sendet der ESP32 automatisch:

```
READY
```

```
FW SelectionPanel v2.5.2
```

Der Server wartet auf `READY`, bevor er Befehle sendet. Empfohlener Timeout: 5 s.

7.2.2 WebSocket (Server ↔ Browser)

Der Browser verbindet sich zu `ws://host:8080/ws`. Nach Verbindung ist der Client sofort empfangsbereit – kein Handshake auf Anwendungsebene nötig.

7.3 Serial-Protokoll (ESP32 ↔ Pi)

IDs sind 1-basiert und 3-stellig formatiert (001–100). Physikalisch: USB-CDC @ 115 200 baud, 8N1, LF-terminiert.

7.3.1 Nachrichten ESP32 → Pi

Tabelle 50: Nachrichten vom ESP32 zum Pi

Nachricht	Beispiel	Beschreibung
`READY`	`READY`	ESP32 betriebsbereit
`FW`	`FW SelectionPanel v2.5.2`	Firmware-Version
`PRESS`	`PRESS 001`	Taster gedrückt (nach Entprellung)
`RELEASE`	`RELEASE 001`	Taster losgelassen
`PONG`	`PONG`	Antwort auf `PING`
`OK`	`OK`	Befehl erfolgreich ausgeführt
`ERROR`	`ERROR invalid id`	Fehlermeldung

7.3.2 Befehle Pi → ESP32

Tabelle 51: Befehle vom Pi zum ESP32

Befehl	Beispiel	Beschreibung
`PING`	`PING`	Verbindungstest → `PONG`
`STATUS`	`STATUS`	Zustand abfragen
`VERSION`	`VERSION`	Firmware-Version abfragen
`HELP`	`HELP`	Verfügbare Befehle auflisten
`LEDSET`	`LEDSET 003`	One-Hot: nur diese LED an
`LEDON`	`LEDON 003`	LED additiv einschalten
`LEDOFF`	`LEDOFF 003`	LED ausschalten
`LEDCLR`	`LEDCLR`	Alle LEDs aus
`LEDALL`	`LEDALL`	Alle LEDs an

One-Hot vs. Additiv

`LEDSET` schaltet *nur* die angegebene LED an (alle anderen aus). `LEDON`/`LEDOFF` ändern einzelne LEDs, ohne andere zu beeinflussen.

7.3.3 Fehlerbehandlung

Tabelle 52: Fehlermeldungen und ihre Ursachen

Fehlermeldung	Ursache	Lösung
`ERROR unknown command`	Unbekannter Befehl	Befehlsname prüfen
`ERROR invalid id`	ID außerhalb 1–100	ID-Bereich prüfen
`ERROR missing id`	ID fehlt	`LEDSET <id>` mit ID

7.3.4 Timing

Tabelle 53: Timing-Parameter des Serial-Protokolls

Parameter	Wert	Beschreibung
Baudrate	115 200 baud	8N1
Event-Latenz	< 35 ms	Abtastung + Entprellung
Befehl-Antwort	< 5 ms	Typische Antwortzeit

Die Event-Latenz setzt sich zusammen aus: IO-Zyklus (5 ms worst case), Debounce (30 ms) und Serial (< 1 ms).

7.3.5 Protokoll-Muster

Drei typische Anwendungsmuster zeigen die Flexibilität des Protokolls.

Echo-Modus (LED folgt Taster)

```
while True:
    line = ser.readline().decode().strip()
    if line.startswith("PRESS"):
        btn_id = line.split()[1]
        ser.write(f"LEDSET {btn_id}\n".encode())
```

Listing 9: Echo-Modus: LED spiegelt Tastendruck

Toggle-Modus

```
led_state = [False] * 10

while True:
    line = ser.readline().decode().strip()
    if line.startswith("PRESS"):
        btn_id = int(line.split()[1])
        led_state[btn_id - 1] = not led_state[btn_id - 1]

        if led_state[btn_id - 1]:
            ser.write(f"LEDON {btn_id:03d}\n".encode())
        else:
            ser.write(f"LEDOFF {btn_id:03d}\n".encode())
```

Listing 10: Toggle-Modus: Tastendruck schaltet LED um

Sequenz-Modus

```
sequence = []
MAX_LEN = 5

while True:
    line = ser.readline().decode().strip()
    if line.startswith("PRESS"):
        btn_id = int(line.split()[1])
```

```
        sequence.append(btn_id)

        if len(sequence) > MAX_LEN:
            sequence.pop(0)

        # Alle LEDs der Sequenz anzeigen
        ser.write(b"LEDCLR\n")
        for led_id in sequence:
            ser.write(f"LEDON {led_id:03d}\n".encode())
```

Listing 11: Sequenz-Modus: Letzte N Tastendrücke anzeigen

7.3.6 Beispiel-Session

Eine vollständige Kommunikation zwischen Pi und ESP32:

```
# ESP32 startet
-> READY
-> FW SelectionPanel v2.5.2

# Pi prueft Verbindung
<- PING
-> PONG

# Pi fragt Status ab
<- STATUS
-> STATUS active=0 leds=0x00

# Benutzer drueckt Taster 3
-> PRESS 003

# Pi schaltet LED 3 an
<- LEDSET 003
-> OK

# Benutzer laesst Taster 3 los
-> RELEASE 003

# Benutzer drueckt Taster 7 (Preempt!)
-> PRESS 007
```

```
# Pi schaltet auf LED 7 um
<- LEDSET 007
-> OK

# Pi schaltet alle LEDs aus
<- LEDCLR
-> OK

# Ungueltiger Befehl
<- FOOBAR
-> ERROR unknown command
```

Listing 12: Beispiel-Session (← = Pi → ESP32, → = ESP32 → Pi)

Debug-Modus

Mit `SERIAL_PROTOCOL_ONLY = false` in `config.h` werden zusätzliche Debug-Informationen ausgegeben. Diese sollten vom Parser ignoriert werden. Für Produktivbetrieb: `true` verwenden.

7.4 WebSocket-Protokoll (Server ↔ Browser)

Der Server kommuniziert mit dem Browser-Dashboard über WebSocket. Die Verbindung erfolgt zu `ws://host:8080/ws`.

7.4.1 Server → Browser

Tabelle 54: WebSocket-Nachrichten vom Server zum Browser

Type	Beispiel	Beschreibung
`play`	`{"type:"play",id:3}`	Medien-Wiedergabe starten
`stop`	`{"type:ßtop"}`	Aktuelle Wiedergabe stoppen

Bei `play` zeigt der Browser das Bild `/media/003.jpg` und spielt `/media/003.mp3` ab.

7.4.2 Browser → Server

Tabelle 55: WebSocket-Nachrichten vom Browser zum Server

Type	Beispiel	Beschreibung
`ended`	`{"type:ënded",ïd:3}`	Audio beendet
`ping`	`{"type:"ping"}`	Heartbeat (optional)

Nach `ended` sendet der Server `LEDCLR` an den ESP32 – aber nur, wenn die ID noch aktuell ist (Race-Condition-Schutz).

7.4.3 WebSocket-Ablauf

Der typische Ablauf bei einem Tastendruck:

1. Browser verbindet sich zu `ws://host:8080/ws`
2. ESP32 sendet `PRESS 003` an Server
3. Server broadcastet `{"type:"play",ïd:3}` an alle Clients
4. Browser spielt Audio ab
5. Browser sendet `{"type:ënded",ïd:3}` nach Audio-Ende
6. Server sendet `LEDCLR` an ESP32 (falls ID noch aktuell)

7.5 HTTP-Endpoints

Neben WebSocket bietet der Server REST-Endpoints für Status und Tests.

Tabelle 56: HTTP-Endpoints des Servers

Endpoint	Methode	Beschreibung
`/`	GET	Dashboard (index.html)
`/ws`	WebSocket	Echtzeit-Events
`/status`	GET	Server-Status (JSON)
`/health`	GET	Health-Check (200/503)
`/test/play/{id}`	GET	Tastendruck simulieren
`/test/stop`	GET	Wiedergabe stoppen
`/static/`	GET	JavaScript, CSS
`/media/`	GET	Bilder und Audio

7.5.1 Status-Endpoint

```
curl http://rover:8080/status | jq
```

```
{
  "version": "2.5.2",
  "mode": "prototype",
  "num_media": 10,
  "current_button": 3,
  "ws_clients": 2,
  "serial_connected": true,
  "serial_port": "/dev/serial/by-id/...",
  "media_missing": 0,
  "esp32_local_led": true
}
```

Listing 13: Beispiel-Antwort von /status

7.6 Lokale LED-Steuerung

ESP32 setzt LED lokal

Mit `ESP32_SETS_LED_LOCALLY = true` (Standard) setzt der ESP32 bei Tastendruck die LED selbst. Der Server sendet dann nur `LEDCLR` wenn das Audio beendet ist. Dies reduziert die Latenz auf unter 5 ms.

Der Vorteil: Die LED-Reaktion ist unabhängig von der Netzwerk- und Server-Latenz. Der Benutzer sieht sofortiges Feedback.

8 Embedded Firmware mit Arduino C++

Wie programmiert man einen Mikrocontroller? Dieses Kapitel erklärt die Grundlagen der Embedded-Entwicklung mit Arduino C++ – von den Unterschieden zur Desktop-Programmierung bis zu den Best Practices für robusten Code.

8.1 Was ist Embedded Firmware?

Firmware ist Software, die direkt auf Hardware läuft – ohne Betriebssystem dazwischen. Sie steuert Mikrocontroller, die in Geräten „eingebettet"(embedded) sind.

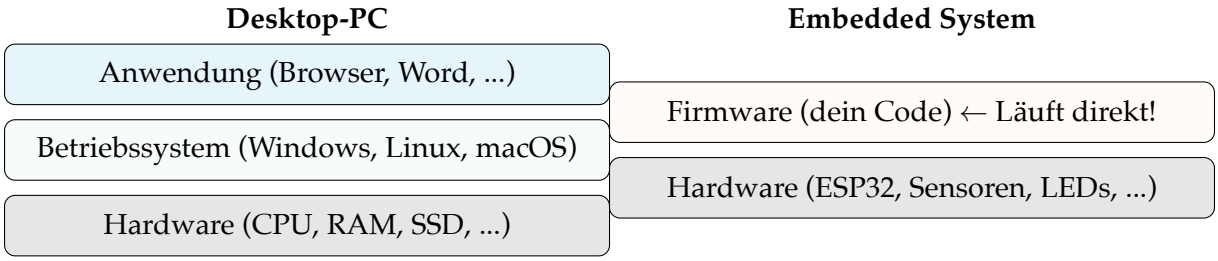


Abbildung 11: Vergleich: Desktop vs. Embedded System

Konsequenzen für den Code:

Tabelle 57: Unterschiede zwischen Desktop- und Embedded-Entwicklung

Aspekt	Desktop	Embedded
Speicher	GBs RAM	KBs RAM
Dateisystem	Ja	Meist nein
Multitasking	OS regelt	Du regelst
Timing	Unkritisch	Oft kritisch
Fehler	Absturz → Neustart	Kann Hardware beschädigen

8.2 Arduino C++: Welche Version?

8.2.1 Der Arduino-Core

Arduino verwendet **C++11** mit Erweiterungen. Der ESP32-Arduino-Core (Espressif) basiert auf GCC 8.4 und unterstützt C++11 vollständig, C++14 und C++17 teilweise.

8.2.2 Was Arduino hinzufügt

Arduino ist kein eigener Compiler, sondern ein **Framework** auf C++:

```
// Klassisches C/C++: Du schreibst main()
int main() {
    // Hardware initialisieren
    while (1) {
        // Endlosschleife
    }
    return 0;
}

// Arduino: Das Framework versteckt main()
void setup() {
    // Wird einmal ausgefuehrt
}

void loop() {
    // Wird endlos wiederholt
}
```

Listing 14: Klassisches C++ vs. Arduino

Hinter den Kulissen

Arduino's `main()` ruft `init()` für die Hardware-Initialisierung, dann `setup()`, und schließlich `loop()` in einer Endlosschleife auf.

8.3 Programmierdogma: Embedded Best Practices

8.3.1 Die goldenen Regeln

1. **KEIN** dynamischer Speicher (`new`, `malloc`) in der Hauptschleife
2. **KEINE** Blockierung (`delay` nur wenn unvermeidbar)
3. **DETERMINISMUS:** Jeder Durchlauf dauert gleich lang
4. **FEHLERTOLERANZ:** Hardware kann jederzeit „spinnen"
5. **RESSOURCEN-BEWUSSTSEIN:** Jedes Byte zählt

8.3.2 Speicher-Dogma

```
// SCHLECHT: Dynamische Allokation
void loop() {
    String text = "Hello";        // String alloziert auf Heap
    text += " World";             // Re-Allokation!
}
// -> Speicherfragmentierung, irgendwann Absturz

// GUT: Statische Allokation
static char text[32];             // Feste Groesse, einmal
   alloziert
void loop() {
    snprintf(text, sizeof(text), "Hello World");
}
```

Listing 15: Speicherverwaltung: Schlecht vs. Gut

8.3.3 Timing-Dogma

```
// SCHLECHT: Blockierendes Warten
void loop() {
```

```
    if (buttonPressed()) {
        doSomething();
        delay(1000);              // CPU macht 1 Sekunde NICHTS
    }
}

// GUT: Nicht-blockierendes Warten
static uint32_t lastAction = 0;
void loop() {
    if (buttonPressed() && (millis() - lastAction >= 1000)) {
        doSomething();
        lastAction = millis();
    }
}
```

Listing 16: Timing: Blockierend vs. Nicht-blockierend

delay() vermeiden

`delay()` blockiert die gesamte CPU. In dieser Zeit können keine anderen Aufgaben ausgeführt werden – keine Taster-Abfrage, keine Serial-Kommunikation, nichts.

8.4 Code-Aufbau: Die Anatomie der Firmware

8.4.1 Dateistruktur

```
src/
|-- config.h      <-- Konfiguration (Konstanten, Pins)
+-- main.cpp      <-- Hauptprogramm (Logik)
```

8.4.2 Aufbau von main.cpp

```
// 1. INCLUDES - Externe Bibliotheken einbinden
#include "config.h"
#include <SPI.h>

// 2. KONSTANTEN & KONFIGURATION
static const SPISettings spiButtons(500000, MSBFIRST, SPI_MODE1);
```

```
7
8  // 3. GLOBALE VARIABLEN (ZUSTAND)
9  // static = nur in dieser Datei sichtbar
10 static uint8_t btnRaw[BTN_BYTES];
11 static uint8_t activeId = 0;
12
13 // 4. HILFSFUNKTIONEN - Kleine, wiederverwendbare Bausteine
14 static inline bool btnIsPressed(const uint8_t *state, uint8_t id)
      { /* ... */ }
15
16 // 5. HARDWARE-FUNKTIONEN - Direkter Zugriff auf Peripherie
17 static void readButtons() { /* ... */ }
18
19 // 6. LOGIK-FUNKTIONEN - Verarbeitung, Entscheidungen
20 static void debounceButtons() { /* ... */ }
21
22 // 7. DEBUG-FUNKTIONEN - Ausgaben fuer Entwicklung
23 static void printState() { /* ... */ }
24
25 // 8. ARDUINO ENTRY POINTS
26 void setup() { /* Einmalige Initialisierung */ }
27 void loop() { /* Hauptschleife */ }
```

Listing 17: Struktur einer Firmware-Datei

8.5 C++ Syntax im Detail

8.5.1 Präprozessor-Direktiven

Der Präprozessor läuft **vor** dem Compiler und manipuliert den Quelltext:

```
1  // Include: Fuegt Dateiinhalt ein
2  #include <SPI.h>          // Sucht in System-Pfaden
3  #include "config.h"       // Sucht im Projekt-Ordner
4
5  // Include Guard: Verhindert doppeltes Einbinden
6  #pragma once              // Moderne Variante (eine Zeile)
7
8  // Alternativ (klassisch):
9  #ifndef CONFIG_H
```

```
#define CONFIG_H
// ... Inhalt ...
#endif
```

Listing 18: Präprozessor-Direktiven

8.5.2 Datentypen

```
// Embedded Best Practice: Exakte Groessen verwenden!
#include <cstdint>

int8_t   a;       //  8 Bit mit Vorzeichen:    -128 bis 127
uint8_t  b;       //  8 Bit ohne Vorzeichen:      0 bis 255
int16_t  c;       // 16 Bit mit Vorzeichen: -32768 bis 32767
uint16_t d;       // 16 Bit ohne Vorzeichen:     0 bis 65535
int32_t  e;       // 32 Bit mit Vorzeichen
uint32_t f;       // 32 Bit ohne Vorzeichen

// size_t: Fuer Groessen und Indizes (plattformabhaengig, immer
   positiv)
size_t arraySize = 10;
```

Listing 19: Exakte Datentypen für Embedded

Warum exakte Größen?

`int` ist auf verschiedenen Plattformen unterschiedlich groß (16 oder 32 Bit). Mit `uint32_t` ist die Größe immer 32 Bit – egal welche Plattform.

8.5.3 Konstanten

```
// C-Style (veraltet, aber funktioniert)
#define LED_COUNT 10      // Textersetzung, kein Typ!

// C++ Style (empfohlen)
const int LED_COUNT = 10;           // Zur Laufzeit, braucht RAM
constexpr int LED_COUNT = 10;       // Zur Compile-Zeit, kein RAM!

// constexpr vs const:
```

```
constexpr int COMPILE_TIME = 5 * 2;     // Berechnung zur Compile-Zeit
const int RUNTIME = analogRead(A0);      // Kann nur const sein (Laufzeit)
```

Listing 20: Konstanten: #define vs. constexpr

8.5.4 Variablen-Deklaration

```
// Speicherklassen
int globalVar;                  // Global: ueberall sichtbar
static int fileVar;             // Datei-lokal: nur in dieser .cpp
void func() {
    int localVar;               // Lokal: nur in dieser Funktion
    static int persistentVar;   // Lokal, aber behaelt Wert zwischen Aufrufen!
}

// static bei lokalen Variablen:
void countCalls() {
    static int counter = 0;   // Initialisierung nur beim ersten Aufruf!
    counter++;
    Serial.println(counter);
}
// Aufruf 1: Ausgabe "1", Aufruf 2: "2", Aufruf 3: "3"
```

Listing 21: Speicherklassen und static

8.5.5 Funktionen

```
// static: Funktion nur in dieser Datei sichtbar
static void readButtons() { }

// inline: Compiler soll Code direkt einsetzen statt Funktionsaufruf
inline bool isValid(int x) { return x > 0; }

```

```
// static inline: Beides kombiniert (haeufig fuer kleine Hilfsfunktionen)
static inline uint8_t byteIndex(uint8_t id) { return (id - 1) / 8; }

// Parameter und Rueckgabe
static inline bool btnIsPressed(const uint8_t *state, uint8_t id) {
    // state: Zeiger auf Array (wird nicht veraendert wegen const)
    // id: Kopie des Wertes (call by value)
    return !(state[byte] & (1u << bit));
}
```

Listing 22: Funktionen mit Schlüsselwörtern

8.5.6 Zeiger und Referenzen

```
uint8_t buffer[10];           // Array
uint8_t *ptr = buffer;        // Zeiger auf erstes Element

*ptr = 42;                    // Dereferenzierung: Wert schreiben
uint8_t x = *ptr;             // Dereferenzierung: Wert lesen
ptr++;                        // Zeiger auf naechstes Element

// const-Korrektheit
const uint8_t *readOnly = buffer;    // Daten nicht aenderbar

// Funktionsparameter:
void modify(int x) { x = 99; }           // Call by Value: Kopie
void modify(int *x) { *x = 99; }         // Call by Pointer: Original
void modify(int &x) { x = 99; }          // Call by Reference: Original
void print(const String &s) { }          // Const Reference: effizient
```

Listing 23: Zeiger, Arrays und Parameterübergabe

8.5.7 Bit-Operationen

Bit-Operationen sind essentiell für Embedded-Entwicklung!

```
uint8_t a = 0b11001010;
uint8_t b = 0b10101100;

a & b     // AND:  0b10001000  (beide 1 -> 1)
a | b     // OR:   0b11101110  (mindestens einer 1 -> 1)
a ^ b     // XOR:  0b01100110  (genau einer 1 -> 1)
~a        // NOT:  0b00110101  (invertiert)
a << 2    // Links-Shift: 0b00101000 (x 4)
a >> 2    // Rechts-Shift: 0b00110010 (/ 4)

// Praktische Anwendungen:
value |= (1u << bitNr);       // Bit setzen (auf 1)
value &= ~(1u << bitNr);      // Bit loeschen (auf 0)
value ^= (1u << bitNr);       // Bit umschalten (toggle)
if (value & (1u << bitNr)) {} // Bit pruefen
```

Listing 24: Bit-Operatoren und praktische Anwendungen

```
// Taster gedrueckt pruefen (Active-Low: 0 = gedrueckt)
return !(state[byte] & (1u << bit));
//       ^              ^
//       Invertieren    Bit-Maske

// LED einschalten
ledState[byte] |= (1u << bit);

// LED ausschalten
ledState[byte] &= ~(1u << bit);
```

Listing 25: Bit-Operationen im Selection Panel

8.5.8 Kontrollstrukturen

```
// for-Schleife
for (int i = 0; i < 10; i++) {
    // i laeuft von 0 bis 9
```

```
}

// Rueckwaerts (wichtig fuer Daisy-Chain!)
for (int i = LED_BYTES - 1; i >= 0; --i) {
    SPI.transfer(ledState[i]);
}

// Fruehes Verlassen
for (int i = 0; i < 100; i++) {
    if (found) break;          // Schleife sofort verlassen
    if (skip) continue;        // Zum naechsten Durchlauf springen
}

// Ternaerer Operator
int max = (a > b) ? a : b;
```

Listing 26: Schleifen und Kontrollfluss

8.5.9 Speicher-Funktionen

```
#include <cstring>

uint8_t src[10] = {1, 2, 3};
uint8_t dst[10];

memcpy(dst, src, 10);          // 10 Bytes von src nach dst
   kopieren
memset(dst, 0x00, 10);         // 10 Bytes mit 0x00 fuellen
memset(dst, 0xFF, 10);         // 10 Bytes mit 0xFF fuellen

if (memcmp(src, dst, 10) == 0) {
    // Identisch
}
```

Listing 27: memcpy, memset, memcmp

8.6 Arduino-spezifische Funktionen

Tabelle 58: Wichtige Arduino-Funktionen

Funktion	Beschreibung
Digitale I/O	
`pinMode(PIN, MODE)`	MODE: INPUT, OUTPUT, INPUT_PULLUP
`digitalWrite(PIN, val)`	HIGH oder LOW setzen
`digitalRead(PIN)`	HIGH oder LOW lesen
Timing	
`delay(ms)`	Warten (BLOCKIEREND!)
`delayMicroseconds(us)`	Warten in μs (BLOCKIEREND!)
`millis()`	Millisekunden seit Start
`micros()`	Mikrosekunden seit Start
Serial	
`Serial.begin(baud)`	Baudrate setzen
`Serial.print(x)`	Ohne Zeilenumbruch
`Serial.println(x)`	Mit Zeilenumbruch
`Serial.printf(...)`	Formatiert (ESP32)
SPI	
`SPI.begin(SCK, MISO, MOSI, SS)`	Pins konfigurieren
`SPI.beginTransaction(settings)`	Transaktion starten
`SPI.transfer(byte)`	Byte senden/empfangen
`SPI.endTransaction()`	Transaktion beenden
PWM (ESP32)	
`ledcSetup(ch, freq, res)`	Kanal konfigurieren
`ledcAttachPin(pin, ch)`	Pin zuweisen
`ledcWrite(ch, duty)`	Duty Cycle setzen

8.7 Der Code im Kontext

8.7.1 Ablaufdiagramm

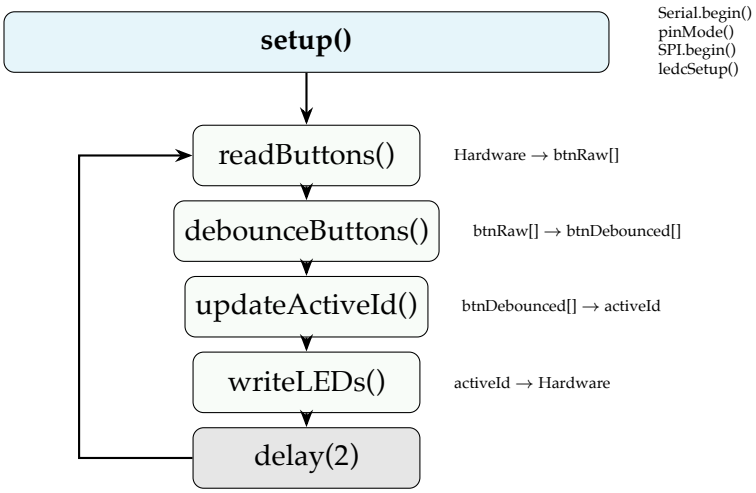


Abbildung 12: Ablauf der Firmware-Hauptschleife

8.7.2 Datenfluss

```
Hardware          Firmware                    Hardware
---------         --------                    --------
Taster ----------> btnRaw[]
                      |
                      v
                  btnDebounced[] (zeitverzoegert, stabil)
                      |
                      v
                  activeId (0–10)
                      |
                      v
                  ledState[]
                      |
                      v
                               ---------------> LEDs
```

8.8 Zusammenfassung

8.8.1 Checkliste für guten Embedded-Code

- ☐ Exakte Datentypen verwenden (`uint8_t`, `uint32_t`, ...)
- ☐ `static` für datei-lokale Variablen und Funktionen
- ☐ `constexpr` für Compile-Zeit-Konstanten
- ☐ Kein dynamischer Speicher in `loop()`
- ☐ Blockierende Wartezeiten vermeiden
- ☐ Bit-Operationen statt Division/Multiplikation wo möglich
- ☐ Fehlerprüfung bei Parametern (Range-Checks)
- ☐ Aussagekräftige Namen (nicht `x`, `temp`, `data`)
- ☐ Kommentare erklären WARUM, nicht WAS

8.8.2 Schnellreferenz: Schlüsselwörter

Tabelle 59: C++ Schlüsselwörter für Embedded

Schlüsselwort	Bedeutung
`static`	Datei-lokal (global) oder persistent (lokal)
`const`	Wert nicht änderbar (zur Laufzeit)
`constexpr`	Wert zur Compile-Zeit berechnet
`inline`	Compiler-Hinweis: Code direkt einsetzen
`volatile`	Wert kann sich „von außen"ändern (Hardware, ISR)
`void`	Kein Rückgabewert / generischer Zeiger

9 Firmware Code Guide

Wie ist die Firmware des Selection Panels aufgebaut? Dieser Guide dokumentiert die Architektur, die Module und die Designentscheidungen der ESP32-S3-Firmware.

9.1 Projektstruktur

```
firmware/
|-- include/
|   |-- bitops.h        # Bit-Adressierung fuer Taster/LEDs
|   |-- config.h        # Zentrale Konfiguration
|   +-- types.h         # Gemeinsame Datentypen
|-- src/
|   |-- main.cpp        # Entry Point
|   |-- app/
|   |   |-- io_task.cpp/.h      # I/O-Verarbeitung
|   |   +-- serial_task.cpp/.h  # Protokoll-Handler
|   |-- drivers/
|   |   |-- cd4021.cpp/.h       # Taster-Schieberegister
|   |   +-- hc595.cpp/.h        # LED-Schieberegister
|   |-- hal/
|   |   +-- spi_bus.cpp/.h      # SPI-Abstraktion
|   +-- logic/
|       |-- debounce.cpp/.h     # Entprellung
|       +-- selection.cpp/.h    # Auswahl-Logik
+-- platformio.ini
```

9.2 Schichtenmodell

Die Firmware folgt einem strikten Schichtenmodell. Abhängigkeiten zeigen nur nach unten – eine Schicht kennt nur die darunterliegende.

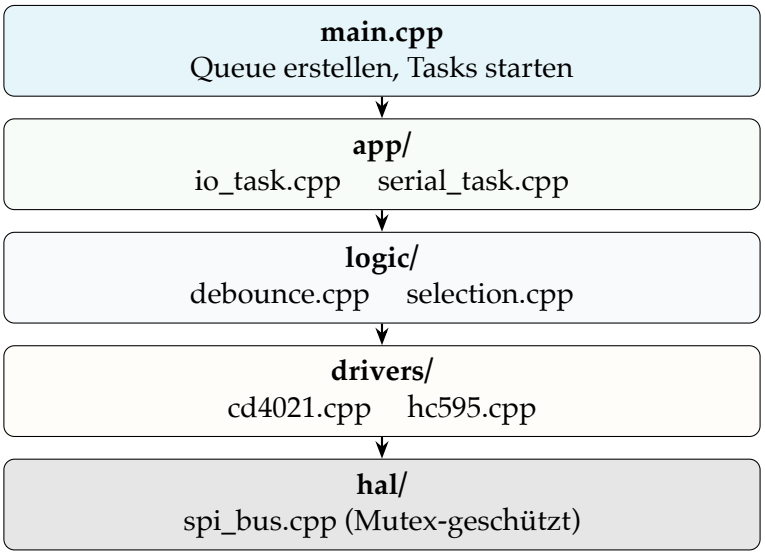


Abbildung 13: Schichtenmodell der Firmware

9.3 Modul-Referenz

9.3.1 config.h – Zentrale Konfiguration

Alle Hardware- und Timing-Parameter an einer Stelle:

```
// Anzahl der Ein-/Ausgaenge
constexpr uint8_t BTN_COUNT = 10;
constexpr uint8_t LED_COUNT = 10;

// Bytes fuer Bit-Arrays (aufrunden: 10 Bits -> 2 Bytes)
constexpr size_t BTN_BYTES = (BTN_COUNT + 7) / 8;
constexpr size_t LED_BYTES = (LED_COUNT + 7) / 8;

// Pin-Zuordnung (gemeinsamer SPI-Bus)
constexpr int PIN_SCK      = D8;   // SPI-Takt (shared)
constexpr int PIN_BTN_PS   = D1;   // CD4021B: Parallel/Serial
   Select
constexpr int PIN_BTN_MISO = D9;   // CD4021B: Daten (Q8)
constexpr int PIN_LED_MOSI = D10;  // 74HC595: Daten (SER)
constexpr int PIN_LED_RCK  = D0;   // 74HC595: Latch (RCLK)
constexpr int PIN_LED_OE   = D2;   // 74HC595: Output Enable (PWM)

// Timing
constexpr uint32_t IO_PERIOD_MS = 5;   // 200 Hz Abtastrate
constexpr uint32_t DEBOUNCE_MS = 30;   // Entprellzeit

// SPI-Einstellungen
constexpr uint32_t SPI_HZ_BTN = 500000UL;  // 500 kHz (CD4021B)
constexpr uint32_t SPI_HZ_LED = 1000000UL; // 1 MHz (74HC595)
constexpr uint8_t SPI_MODE_BTN = SPI_MODE1; // CPOL=0, CPHA=1
constexpr uint8_t SPI_MODE_LED = SPI_MODE0; // CPOL=0, CPHA=0
```

Listing 28: Zentrale Konfiguration in config.h

9.3.2 bitops.h – Bit-Adressierung

Dieses Modul abstrahiert die Hardware-Verdrahtung der Schieberegister.

```
// CD4021B Button Bit-Mapping
// Hardware-Verdrahtung: BTN 1 -> PI-8 (Pin 1), BTN 8 -> PI-1
```

```
   (Pin 7)
// Formel: btn_bit(id) = (id - 1) % 8
static inline uint8_t btn_byte(uint8_t id) { return (id - 1) / 8;
   }
static inline uint8_t btn_bit(uint8_t id)  { return (id - 1) % 8;
   }

// 74HC595 LED Bit-Mapping
// Hardware-Verdrahtung: LED 1 -> QA (Bit 0), LED 8 -> QH (Bit 7)
static inline uint8_t led_byte(uint8_t id) { return (id - 1) / 8;
   }
static inline uint8_t led_bit(uint8_t id)  { return (id - 1) % 8;
   }

// Active-Low Hilfsfunktionen (Taster: 0 = gedrueckt, Pull-up!)
static inline bool activeLow_pressed(const uint8_t* arr, uint8_t
   id) {
    return !((arr[btn_byte(id)] >> btn_bit(id)) & 1);
}

// LED-Hilfsfunktionen
static inline void led_set(uint8_t* arr, uint8_t id, bool on) {
    uint8_t mask = 1 << led_bit(id);
    if (on) arr[led_byte(id)] |= mask;
    else    arr[led_byte(id)] &= ~mask;
}
```

Listing 29: Bit-Adressierung für Taster und LEDs

9.3.3 types.h – Gemeinsame Datentypen

```
struct LogEvent {
    uint32_t ms;                // Zeitstempel
    uint8_t raw[BTN_BYTES];     // Rohzustand der Taster
    uint8_t deb[BTN_BYTES];     // Entprellter Zustand
    uint8_t led[LED_BYTES];     // LED-Ausgabezustand
    uint8_t activeId;           // Aktive Auswahl (0 = keine,
       1-10 = ID)
    bool rawChanged;            // Flag: Raw hat sich geaendert
```

```
    bool debChanged;            // Flag: Debounced hat sich
       geaendert
    bool activeChanged;         // Flag: Auswahl hat sich
       geaendert
};
```

Listing 30: LogEvent-Struktur für die Queue

9.3.4 spi_bus.h – HAL-Schicht

```
class SpiBus {
public:
    void begin(int sck, int miso, int mosi);
    void lock();    // Mutex nehmen
    void unlock();  // Mutex freigeben
private:
    SemaphoreHandle_t mtx_;
};

// RAII Guard fuer automatisches Cleanup
class SpiGuard {
public:
    SpiGuard(SpiBus& bus, const SPISettings& settings);
    ~SpiGuard();  // Automatisch: endTransaction() + unlock()
};
```

Listing 31: Mutex-geschützte SPI-Abstraktion

9.3.5 cd4021.h – Taster-Treiber

```
void Cd4021::readRaw(SpiBus& bus, uint8_t* out) {
    // 1. Parallel Load
    digitalWrite(PIN_BTN_PS, HIGH);
    delayMicroseconds(2);
    digitalWrite(PIN_BTN_PS, LOW);
    delayMicroseconds(2);

    // 2. First Bit Rescue: PI-1 liegt bereits an Q8!
    uint8_t firstBit = digitalRead(PIN_BTN_MISO);
```

```
10
11      // 3. SPI Transfer (restliche Bits)
12      SpiGuard g(bus, spi_);
13      SPI.transfer(out, BTN_BYTES);
14
15      // 4. First Bit einsetzen (MSB von Byte 0)
16      out[0] = (out[0] >> 1) | (firstBit << 7);
17  }
```

Listing 32: CD4021B-Treiber mit First-Bit-Rescue

First-Bit-Problem

Nach dem Parallel-Load (P/S HIGH→LOW) liegt das erste Bit (PI-1) bereits an Q8 an – bevor der erste Clock kommt. Die Lösung: `digitalRead()` vor dem SPI-Transfer.

9.3.6 debounce.h – Entprellung

Der Algorithmus ist zeitbasiert:

1. Bei Rohwert-Änderung: Timer zurücksetzen
2. Wenn Timer ≥ 30 ms UND Rohwert $\neq$ entprellt: übernehmen

```
1  class Debouncer {
2  public:
3      void init();
4      bool update(uint32_t nowMs, const uint8_t* raw, uint8_t* deb);
5  private:
6      uint8_t rawPrev_[BTN_BYTES];
7      uint32_t lastChange_[BTN_COUNT];  // Timer pro Taster
8  };
```

Listing 33: Debouncer-Klasse

9.3.7 selection.h – Auswahl-Logik

Prinzip: „Last Press Wins"– der zuletzt gedrückte Taster wird aktiv.

```
class Selection {
public:
    void init();
    bool update(const uint8_t* debNow, uint8_t& activeId);
private:
    uint8_t debPrev_[BTN_BYTES];
};
```

Listing 34: Selection-Klasse

Latch-Modus

Mit `LATCH_SELECTION = true` bleibt die Auswahl nach Loslassen bestehen.

9.4 FreeRTOS-Konfiguration

Tabelle 60: FreeRTOS Tasks

Task	Core	Priorität	Periode	Funktion
io_task	1	5	5 ms	Hardware-I/O (Taster, LEDs)
serial_task	1	2	Event-driven	Protokoll-Handler

Core 0 bleibt für WiFi/BLE reserviert (falls später benötigt).

9.5 Datenfluss im Detail

Was passiert in jedem I/O-Zyklus? Schauen wir uns die Hauptschleife an:

```
void io_loop() {
    TickType_t lastWake = xTaskGetTickCount();

    while (true) {
        uint32_t now = millis();

        // 1. Taster einlesen
        cd4021.readRaw(bus, raw);
        bool rawChg = memcmp(raw, rawPrev, BTN_BYTES) != 0;

        // 2. Entprellen
```

```
        bool debChg = debouncer.update(now, raw, deb);

        // 3. Auswahl aktualisieren
        bool selChg = selection.update(deb, activeId);

        // 4. LEDs setzen (activeId -> One-Hot)
        memset(led, 0, LED_BYTES);
        if (activeId > 0) led_set(led, activeId, true);
        hc595.write(bus, led);

        // 5. Bei Aenderung: Event senden
        if (debChg || selChg) {
            LogEvent ev = {now, raw, deb, led, activeId, ...};
            xQueueSend(queue, &ev, 0);
        }

        // 6. Auf naechsten Zyklus warten
        vTaskDelayUntil(&lastWake, pdMS_TO_TICKS(IO_PERIOD_MS));
    }
}
```

Listing 35: I/O-Task Hauptschleife (vereinfacht)

9.6 Design-Entscheidungen

9.6.1 Warum Queue statt direktem Serial-Aufruf?

Die Queue entkoppelt I/O von Serial-Ausgabe:

- io_task blockiert nie (5 ms Deadline)
- serial_task kann langsam sein (USB-Puffer voll)
- Atomare Snapshots (keine Race-Conditions)

9.6.2 Warum SpiGuard (RAII)?

Garantiert korrektes Cleanup auch bei frühem Return:

```
{
    SpiGuard g(bus, settings);
```

```
    if (error) return;  // endTransaction() + unlock() trotzdem!
    SPI.transfer(data);
}
```

Listing 36: RAII-Pattern für SPI

9.6.3 Warum zeitbasiertes Debouncing?

- Unabhängig von Abtastrate
- Jeder Taster hat eigenen Timer
- Schnelle Tastenfolgen möglich

9.6.4 Warum LED_REFRESH_EVERY_CYCLE?

Beim CD4021B-Read werden Nullen durch den 74HC595 getaktet (gemeinsamer SCK). Ein glitchender Latch-Pin könnte LEDs kurz ausschalten. Der Refresh nach jedem Zyklus kompensiert dies.

9.7 Skalierung auf 100 Buttons

Die Architektur skaliert durch Änderung zweier Konstanten:

```
constexpr uint8_t BTN_COUNT = 100;
constexpr uint8_t LED_COUNT = 100;
// BTN_BYTES und LED_BYTES werden automatisch auf 13 berechnet
```

Listing 37: Skalierung auf 100 Buttons

Alle Schleifen und Bit-Arrays passen sich automatisch an. Die SPI-Transferzeit steigt von $\sim$40 μs auf $\sim$320 μs – weit unter dem 5 ms-Budget.

9.8 Build und Upload

```
# PlatformIO
cd firmware
pio run -t upload
pio device monitor
```

```
# Serial-Ausgabe
# READY
# FW SelectionPanel v2.5.2
# PRESS 001
# RELEASE 001
```

9.9 Debugging

9.9.1 Log-Level aktivieren

In `config.h`:

```
constexpr bool LOG_VERBOSE_PER_ID = true;   // Details pro Taster
constexpr bool LOG_ON_RAW_CHANGE = true;    // Rohwert-Aenderungen
constexpr bool SERIAL_PROTOCOL_ONLY = false; // Debug-Ausgabe
   erlauben
```

Listing 38: Debug-Optionen

9.9.2 Bit-Mapping verifizieren

```
# STATUS-Befehl zeigt Bit-Arrays:
echo "STATUS" > /dev/serial/by-id/usb-Espressif*

# Ausgabe:
# LEDS 0000000001    <-- LED 1 an (Bit 0)
# BTNS 1111111110    <-- BTN 1 gedrueckt (Active-Low: Bit 0 = 0)
```

Tabelle 61: Serial-Protokoll für Debugging

Befehl	Antwort	Beschreibung
`STATUS`	`LEDS .../BTNS ...`	Bit-Arrays anzeigen
`PING`	`PONG`	Verbindungstest
`LEDSET 005`	`OK`	LED 5 setzen (One-Hot)
`LEDCLR`	`OK`	Alle LEDs aus

10 Raspberry Pi Integration

In Phase 7 wird der ESP32-S3 zum reinen I/O-Controller. Der Raspberry Pi 5 übernimmt die Anwendungslogik: Er empfängt Taster-Events über Serial, sendet sie per WebSocket an das Web-Dashboard und steuert bei Bedarf die LEDs. Wie funktioniert diese Arbeitsteilung?

10.1 Systemübersicht

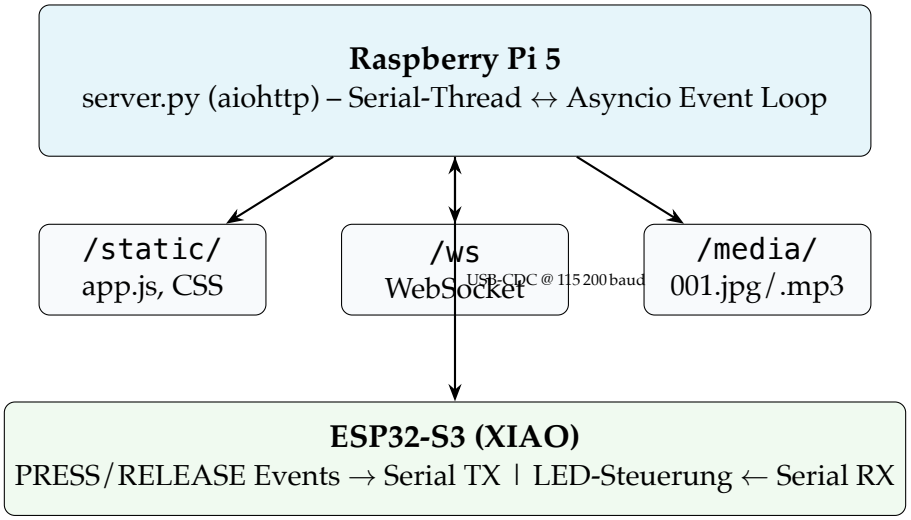


Abbildung 14: Systemarchitektur der Pi-Integration

Lokale LED-Steuerung

Mit `ESP32_SETS_LED_LOCALLY = true` setzt der ESP32 die LED bei Tastendruck selbst. Der Server muss dann nur noch `LEDCLR` nach Audio-Ende senden. Das reduziert die Latenz auf unter 5 ms.

10.2 Serial-Verbindung

10.2.1 Stabiler Device-Pfad (by-id)

Der ESP32-S3 erscheint als USB-CDC-Gerät. Der Pfad `/dev/ttyACM0` kann sich nach einem Reboot ändern – ein klassisches Problem. Die Lösung: Wir verwenden den stabilen by-id-Pfad.

```
# Stabilen Pfad ermitteln
ls -la /dev/serial/by-id/
```

```
# Ausgabe:
   usb-Espressif_USB_JTAG_serial_debug_unit_98:3D:AE:EA:08:1C-if00

# Dieser Pfad bleibt stabil
SERIAL_PORT="/dev/serial/by-id/usb-Espressif_USB_JTAG_serial_debug_unit_98:3D:A
```

10.2.2 Berechtigungen

```
# Benutzer zur dialout-Gruppe hinzufuegen
sudo usermod -aG dialout $USER
# Ausloggen und wieder einloggen!

# Verbindung testen
screen $SERIAL_PORT 115200
# Erwartete Ausgabe:
# READY
# FW SelectionPanel v2.5.2
```

Neu einloggen erforderlich

Gruppenmitgliedschaften werden erst nach einem neuen Login aktiv. Ein einfaches `exit` und erneutes `ssh rover` genügt.

10.3 Server-Architektur

Der Python-Server verwendet **aiohttp** für asynchrone HTTP/WebSocket-Verarbeitung. Table 62 zeigt die Komponenten.

Tabelle 62: Server-Komponenten und ihre Technologien

Komponente	Technologie	Funktion
HTTP-Server	aiohttp	Statische Dateien, API-Endpoints
WebSocket	aiohttp	Echtzeit-Kommunikation mit Browser
Serial-Reader	Threading	Liest ESP32-Events im Hintergrund
Media-Validator	Startup	Prüft ob alle Medien vorhanden

10.3.1 HTTP-Endpoints

Tabelle 63: HTTP-Endpoints des Servers

Endpoint	Methode	Beschreibung
`/`	GET	Web-Dashboard (index.html)
`/ws`	WebSocket	Echtzeit-Events
`/static/`	GET	JavaScript, CSS, Favicon
`/media/`	GET	Bilder und Audio (001.jpg, 001.mp3)
`/status`	GET	Server-Status als JSON
`/health`	GET	Health-Check (für Monitoring)
`/test/play/{id}`	GET	Simuliert Tastendruck
`/test/stop`	GET	Stoppt Wiedergabe

10.3.2 Konfiguration

```
VERSION = "2.5.2"

# Build-Modus
PROTOTYPE_MODE = True   # True = 10 Medien, False = 100 Medien
NUM_MEDIA = 10 if PROTOTYPE_MODE else 100

# Serial-Port (stabiler by-id Pfad!)
SERIAL_PORT = "/dev/serial/by-id/usb-Espressif_USB_JTAG_..."
SERIAL_BAUD = 115200

# HTTP-Server
HTTP_HOST = "0.0.0.0"
HTTP_PORT = 8080

# ESP32 setzt LED selbst bei Tastendruck
ESP32_SETS_LED_LOCALLY = True
```

Listing 39: Wichtige Einstellungen in server.py

10.4 Protokolle

10.4.1 WebSocket-Protokoll (Server ↔ Browser)

Tabelle 64: WebSocket-Nachrichten

Richtung	Type	Beispiel	Beschreibung
Server → Browser	play	`{"type:"play",ïd:3}`	Wiedergabe starten
Server → Browser	stop	`{"type:ßtop"}`	Wiedergabe stoppen
Browser → Server	ended	`{"type:ënded",ïd:3}`	Audio beendet
Browser → Server	ping	`{"type:"ping"}`	Heartbeat

10.4.2 Serial-Protokoll (ESP32 ↔ Pi)

Tabelle 65: Serial-Nachrichten zwischen ESP32 und Pi

Richtung	Nachricht	Beispiel	Bedeutung
ESP32 → Pi	READY	`READY`	ESP32 bereit
ESP32 → Pi	FW	`FW SelectionPanel v2.5.2`	Firmware-Version
ESP32 → Pi	PRESS	`PRESS 001`	Taster gedrückt
ESP32 → Pi	RELEASE	`RELEASE 001`	Taster losgelassen
Pi → ESP32	LEDSET	`LEDSET 001`	One-Hot LED (optional)
Pi → ESP32	LEDCLR	`LEDCLR`	Alle LEDs aus

LED-Steuerung

Mit `ESP32_SETS_LED_LOCALLY = true` setzt der ESP32 die LED bei Tastendruck selbst. Der Server sendet dann nur `LEDCLR` nach Audio-Ende.

10.5 Datenfluss

Was passiert, wenn wir Taster 3 drücken? Der Datenfluss durchläuft mehrere Stationen:

1. **ESP32:** Entprellt den Taster, schaltet LED 3 an, sendet `PRESS 003`
2. **Serial-Thread:** Empfängt die Nachricht, gibt sie an den Event Loop
3. **Event Loop:** Broadcastet `{"type:"play",ïd:3}` an alle WebSocket-Clients
4. **Browser:** Zeigt Bild 003.jpg, spielt 003.mp3 ab

5. **Browser:** Sendet nach Audio-Ende `{"type:ënded",ïd:3}`
6. **Server:** Sendet `LEDCLR` an ESP32 (falls ID noch aktuell)
7. **ESP32:** Schaltet alle LEDs aus

10.6 Web-Dashboard

Das Dashboard (`index.html` + `app.js`) bietet folgende Features:

- **Audio-Unlock:** Button zum Entsperren der Browser-Autoplay-Policy
- **Medien-Preload:** Lädt alle Bilder / Audio nach Unlock vor
- **Echtzeit-Anzeige:** Aktuelles Bild + Audio-Fortschritt
- **Keyboard-Shortcuts:** Space = Play / Pause, Ctrl+D = Debug
- **Debug-Panel:** Zeigt alle Events (ausklappbar)

10.6.1 Zugriff

```
# Server starten
cd /home/pi/selection-panel
python3 server.py

# Browser oeffnen
# Lokal:   http://localhost:8080/
# LAN:     http://rover:8080/
# IP:      http://192.168.1.24:8080/
```

10.6.2 Status-API

```
curl http://rover:8080/status | jq
```

```
{
  "version": "2.5.2",
  "mode": "prototype",
  "num_media": 10,
  "current_button": null,
```

```
  "ws_clients": 1,
  "serial_connected": true,
  "serial_port": "/dev/serial/by-id/...",
  "media_missing": 0,
  "esp32_local_led": true
}
```

Listing 40: Beispiel-Antwort von /status

10.7 USB-Port-Verwaltung (AMR-Koexistenz)

Auf dem Pi läuft auch das AMR-Projekt, das denselben ESP32-Port nutzen kann. Ein **flock-basiertes Locking** verhindert Konflikte.

10.7.1 Lock-Mechanismus

```
# Lock-Datei
/var/lock/esp32-serial.lock

# Selection Panel: Non-blocking (startet nicht wenn belegt)
flock -n /var/lock/esp32-serial.lock python3 server.py

# AMR Agent: Blocking (wartet bis frei)
flock /var/lock/esp32-serial.lock micro_ros_agent ...
```

10.7.2 Schneller Wechsel

```
# --> Selection Panel Modus
sudo systemctl stop selection-panel.service  # Falls AMR laeuft
sudo systemctl start selection-panel.service
sudo journalctl -u selection-panel.service -f

# --> AMR Modus
sudo systemctl stop selection-panel.service
cd /home/pi/amr/docker
sudo docker compose -p docker up -d microros_agent
```

10.7.3 Kontrolle

```
# Wer haelt den USB-Port?
sudo fuser -v /dev/ttyACM0

# Wer haelt den Lock?
sudo lslocks | grep esp32-serial
```

10.8 systemd-Service

10.8.1 Service-Datei

```
[Unit]
Description=Selection Panel Server
After=network.target

[Service]
Type=simple
User=pi
WorkingDirectory=/home/pi/selection-panel
# flock -n: Startet nur wenn Lock frei
ExecStart=/usr/bin/flock -n /var/lock/esp32-serial.lock
   /usr/bin/python3 server.py
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Listing 41: /etc/systemd/system/selection-panel.service

10.8.2 Aktivierung

```
# Service registrieren
sudo systemctl daemon-reload

# Manueller Start
sudo systemctl start selection-panel.service
```

```
# Autostart aktivieren
sudo systemctl enable selection-panel.service

# Status pruefen
sudo systemctl status selection-panel.service

# Logs verfolgen
journalctl -u selection-panel.service -f
```

10.9 Medien-Struktur

```
media/
|-- 001.jpg    # Bild fuer Taster 1
|-- 001.mp3    # Audio fuer Taster 1
|-- 002.jpg
|-- 002.mp3
|-- ...
|-- 010.jpg
+-- 010.mp3
```

10.9.1 Validierung beim Start

Der Server prüft beim Start, ob alle erwarteten Medien vorhanden sind:

```
2026-01-08 [INFO] Medien-Validierung: 10/10 vollstaendig

# Oder bei fehlenden Dateien:
2026-01-08 [WARNING] Fehlende Medien: 2 Dateien
2026-01-08 [WARNING]   - 005.jpg
2026-01-08 [WARNING]   - 005.mp3
```

10.9.2 Test-Medien generieren

```
cd /home/pi/selection-panel
./scripts/generate_test_media.sh
```

10.10 Troubleshooting

Tabelle 66: Häufige Probleme und Lösungen

Problem	Lösung
ESP32 nicht erkannt	`lsusb | grep Espressif`, `ls /dev/serial/by-id/`
Keine READY-Nachricht	`screen /dev/ttyACM0 115200`, ESP32 per Reset neu starten
WebSocket verbindet nicht	`sudo systemctl status selection-panel.service`, Port 8080 prüfen
Audio spielt nicht	„Sound aktivieren"Button klicken, Browser-Konsole prüfen
Port-Konflikt mit AMR	`sudo fuser -v /dev/ttyACM0`, Selection Panel stoppen

10.11 Latenz-Analyse

Tabelle 67: Latenz-Analyse der Signalkette

Station	Latenz	Beschreibung
Taster → ESP32 (Debounce)	30 ms	Entprellzeit
ESP32 → Serial TX	< 1 ms	USB-CDC
Serial → Server	< 1 ms	Python-Thread
Server → WebSocket	< 1 ms	aiohttp
Browser → Audio Start	5 ms–50 ms	Gecached: ∼5 ms
Gesamt (Preloaded)	**∼40 ms**	Mit Medien-Cache
Gesamt (Nicht gecached)	**∼200 ms**	Ohne Preloading

Medien-Preloading

Das Preloading im Browser reduziert die Latenz erheblich. Nach dem Klick auf „Sound aktivieren"werden alle Medien vorgeladen – danach startet die Wiedergabe in unter 50 ms.

11 Python-Code-Guide

Wie ist der Server aufgebaut? Dieser Guide erklärt die Architektur von `server.py` und die Designentscheidungen dahinter. Wir folgen dabei dem Prinzip: Jede Komponente hat *eine* Hauptverantwortung mit klaren Schnittstellen.

11.1 Architektur in einem Satz

Ein **aiohttp-Server** bridged **ESP32-Serial** → **asyncio** und broadcastet Events per **WebSocket** an Browser-Clients. Bei jedem Tastendruck gilt **„Umschalten gewinnt"** (Preempt) und **One-Hot-LED**.

11.2 Schichten und Verantwortlichkeiten

Die Architektur folgt einer klaren Schichtentrennung. Table 68 zeigt die Komponenten und ihre Aufgaben.

Tabelle 68: Server-Schichten und ihre Verantwortlichkeiten

Schicht	Verantwortung
Konfiguration	Ports, Pfade, Mode, Timeouts (`FRAGMENT_TIMEOUT_MS = 50`, Reconnect 5 s)
Zustand	`AppState` hält `current_id`, WebSocket-Clients, Serial-FD, Media-Status
Serial-Pfad	`serial_reader_task()` liest Bytes im Thread, bildet Zeilen, übergibt an Event-Loop
Event-Logik	`handle_button_press()` setzt `current_id`, sendet `stop` + `play`
HTTP/WebSocket	Routes für `/ws`, `/status`, `/health`, `/test/*`, Static/Media-Serving

Erweiterung nach Verantwortlichkeit

Neue Funktionalität immer dort hinzufügen, wo die Verantwortung liegt:

- Neues Serial-Kommando → `handle_serial_line()`
- Neue WebSocket-Message → `handle_ws_message()`
- Neue HTTP-Route → `create_app()` + eigener Handler

11.3 asyncio-Grundmuster

Die goldene Regel: Im Event-Loop keine Blocker. Kein `time.sleep()`, kein blocking IO. Blockende Operationen gehören in einen Thread oder Process – die Ergebnisse kommen per Callback zurück in den Loop.

11.3.1 Umsetzung im Code

- Serial-IO läuft bewusst im Thread (poll + nonblocking FD)
- Der Event-Loop übernimmt nur: Parsing-Resultate verarbeiten, broadcasten, optional Serial-TX

11.3.2 Praxis-Patterns

```
# Broadcast an alle WebSocket-Clients
await asyncio.gather(
    *[ws.send_json(msg) for ws in state.ws_clients],
    return_exceptions=True  # Fehlerhafte Clients nicht abbrechen
)
```

Listing 42: Parallele Sends mit asyncio.gather()

Resilientes Broadcasting

Mit `return_exceptions=True` sammeln wir fehlerhafte Clients und entfernen sie anschließend aus `ws_clients`. So bricht ein disconnected Client nicht den gesamten Broadcast ab.

11.4 Serial-Parsing: Fragmentierung behandeln

USB-CDC kann Nachrichten fragmentieren – `PRESS` und `003` kommen getrennt an. Wie gehen wir damit um?

11.4.1 Das Problem

Tabelle 69: Beobachtung und Lösung bei Serial-Fragmentierung

Aspekt	Beschreibung
Beobachtung	USB-CDC kann `PRESS` und `003` getrennt liefern
Daten	Pending-Fragment + Timeout-Vervollständigung (50 ms)
Regel	Bytes puffern → Zeilen bilden → Fragmente kombinieren → erst dann Event erzeugen

11.4.2 Best Practice für neue Befehle

Wenn wir neue Serial-Kommandos hinzufügen, verwenden wir **eindeutige Prefixe** (z. B. `SENSOR`, `ACK`). So werden Fragmente nicht versehentlich als Zahlen-ID interpretiert. Die Funktion `parse_button_id()` bleibt strikt: nur `isdigit()`, Range-Check.

11.5 Preempt und One-Hot: Race-Conditions kontrollieren

Bei konkurrierenden Events brauchen wir eine **monotone Wahrheit**. In unserem Fall ist das `state.current_id`. Alles, was später reinkommt, muss gegen diese Wahrheit geprüft werden.

11.5.1 Umsetzung im Code

- **Preempt:** Neuer Tastendruck setzt sofort `state.current_id` und sendet `stop` + `play`
- **Playback-Ende:** `handle_playback_ended()` löscht LEDs nur, wenn `ended_id == current_id`

```
async def handle_playback_ended(ended_id: int) -> None:
    # Nur LEDs loeschen, wenn keine neue Auswahl aktiv
    if ended_id == state.current_id:
        state.current_id = None
        await send_serial("LEDCLR")
    # Sonst: ignorieren (neuer Taster hat bereits uebernommen)
```

Listing 43: Race-Condition-Schutz beim Playback-Ende

Erweiterung mit Sequenznummern

Für robustere Zuordnung bei Multi-Tab oder hoher Latenz: Ergänze eine laufende Sequenznummer (`event_seq += 1`) und sende sie mit `play`. Der Browser kann `ended` dann eindeutig zuordnen.

11.6 Medien-Validierung

Teure Validierung früh (Startup), schnelle Checks zur Laufzeit – das ist die Faustregel.

Tabelle 70: Validierungsstrategie für Medien

Zeitpunkt	Funktion	Prüfung
Startup	`validate_media()`	Prüft pro ID `.jpg` und `.mp3`, zählt, loggt fehlende
Runtime	`check_media_exists()`	Liefert Status für eine ID
Health	`/health`	„degraded"wenn Serial down oder Medien fehlen

Fail Fast für Produktion

Für Produktionsbetrieb: Fehlende Medien optional als harte Startbedingung (Exitcode $\neq 0$), wenn „fail fast"gewünscht ist.

11.7 Code-Qualität: Leitplanken

Diese Regeln zahlen sich in der Praxis aus:

1. **Typen durchziehen:** `Optional[int]`, Return-Types konsequent nutzen – auch für WebSocket-Payload-Schemas
2. **Logging statt Print:** Strukturiertes Logging mit `LOG_FORMAT` und Levels. Für Debug-Phasen: gezielte `logging.debug` in Parser/State-Übergängen
3. **Konstanten zentral:** Mode und Anzahl Medien über `PROTOTYPE_MODE` und `NUM_MEDIA`
4. **Schnittstellen schmal halten:** `handle_button_press(button_id)` ist der zentrale Eingang für „User-Intent"

11.8 Protokoll-Übersicht

11.8.1 Serial-Protokoll (ESP32 ↔ Pi)

Tabelle 71: Serial-Nachrichten zwischen ESP32 und Pi

Richtung	Nachricht	Bedeutung
ESP32 → Pi	`READY`	ESP32 bereit
ESP32 → Pi	`FW SelectionPanel v2.5.2`	Firmware-Version
ESP32 → Pi	`PRESS 001`	Taster 1 gedrückt
ESP32 → Pi	`RELEASE 001`	Taster 1 losgelassen
ESP32 → Pi	`PONG`	Antwort auf PING
Pi → ESP32	`LEDSET 001`	LED 1 an (One-Hot)
Pi → ESP32	`LEDCLR`	Alle LEDs aus
Pi → ESP32	`PING`	Verbindungstest

11.8.2 WebSocket-Protokoll (Server ↔ Browser)

Tabelle 72: WebSocket-Nachrichten

Richtung	Message	Beschreibung
Server → Browser	`{"type:"play",ïd:n}`	Wiedergabe starten
Server → Browser	`{"type:ßtop"}`	Wiedergabe stoppen
Browser → Server	`{"type:ënded",ïd:n}`	Audio beendet
Browser → Server	`{"type:"ping"}`	Heartbeat

11.8.3 HTTP-Endpoints

Tabelle 73: HTTP-Endpoints des Servers

Endpoint	Methode	Beschreibung
`/`	GET	Web-Dashboard
`/ws`	WebSocket	Echtzeit-Events
`/static/`	GET	JavaScript, CSS
`/media/`	GET	Bilder und Audio
`/status`	GET	Server-Status (JSON)
`/health`	GET	Health-Check (200/503)
`/test/play/{id}`	GET	Tastendruck simulieren
`/test/stop`	GET	Wiedergabe stoppen

11.9 Glossar

Tabelle 74: Begriffe aus der Server-Entwicklung

Begriff	Erklärung
aiohttp	Python-Webframework für HTTP-Server und WebSockets auf asyncio-Basis
async/await	Syntax für asynchrone Funktionen (Coroutines), nicht-blockierend
asyncio	Standardbibliothek für kooperatives Multitasking (Event-Loop, Tasks)
Broadcast	Senden derselben Nachricht an mehrere Empfänger
Coroutine	Asynchrone Funktion, die pausiert und fortgesetzt werden kann
Daemon-Thread	Thread, der das Programm nicht am Beenden hindert
Event-Loop	Zentrale Schleife, die Coroutines plant und IO-Ereignisse verarbeitet
File Descriptor	Integer-Handle einer geöffneten OS-Ressource (Datei, Serial)
Fragmentierung	Aufteilung logisch zusammengehöriger Daten in mehrere Chunks
gather	asyncio-Funktion für parallele Ausführung mehrerer Awaitables
Health-Check	Endpoint für Monitoring/Orchestrierung (gesund/degraded)
non-blocking IO	Lese/Schreiboperationen blockieren nicht, liefern sofort Ergebnis
One-Hot	Kodierung, bei der genau ein Element aktiv ist
poll	Systemcall zum Warten auf IO-Events mehrerer FDs
Preempt	Neue Aktion verdrängt sofort die laufende
Race-Condition	Timing-abhängiger Fehler bei konkurrierenden Abläufen
Reconnect-Loop	Wiederholtes Verbinden nach Fehler, meist mit Backoff
WebSocket	Dauerhafte bidirektionale Verbindung für Echtzeit-Events

12 JavaScript-Code-Guide

Wie ist das Dashboard aufgebaut? Dieser Guide erklärt die Architektur von `app.js` und `index.html` und die Designentscheidungen dahinter. Die zentrale Regel: UI-Code wird wartbar, wenn der Datenfluss eindeutig ist.

12.1 Architektur: Datenfluss in 4 Schritten

Der Datenfluss folgt einem klaren Muster: **Input** → **Parse** → **State** → **Render**. Schauen wir uns die einzelnen Schritte an:

1. **Input:** WebSocket empfängt `{"type:ßtop"}` oder `{"type:"play",id:n}`
2. **Parse:** `handleServerMessage()` macht `JSON.parse` + Switch auf `message.type`
3. **State:** `state.currentId`, `state.isPlaying`, `state.audioUnlocked`, `state.preloaded`
4. **Render:** `handleStop()` / `handlePlay(id)` aktualisieren DOM

Leitplanke für Erweiterungen

Jede neue Funktion (z. B. „pause", „volume", „shuffle") sollte entweder **State ändern** oder **rendern** – nicht beides quer verteilt.

12.2 Konfiguration und globaler Zustand

Konstanten zentralisieren, Zustand minimal halten, Zustandsänderung an wenigen Stellen – das sind die Grundregeln.

```
const CONFIG = {
    reconnectInterval: 5000,
    numMedia: 10,
    preloadConcurrency: 3,
    wsUrl: `${location.protocol === 'https:' ? 'wss:' :
       'ws:'}//${location.host}/ws`
};

const state = {
    ws: null,
    audioUnlocked: false,
    currentId: null,
    isPlaying: false,
    preloaded: false,
    preloadProgress: 0
};

const mediaCache = {
    images: {},  // images[id] = HTMLImageElement
    audio: {}    // audio[id] = HTMLAudioElement
};
```

Listing 44: Konfiguration und State-Struktur

Skalierung auf 100 Medien

Für 100 Medien: nur `CONFIG.numMedia = 100` ändern und ggf. `preloadConcurrency` an Netzwerk/Server anpassen.

12.3 WebSocket: Robust verbinden, sauber senden

Reconnect ist Teil des Normalbetriebs. Fehler sollen sichtbar sein, aber nicht „crashen".

```
function connectWebSocket() {
    state.ws = new WebSocket(CONFIG.wsUrl);

    state.ws.onopen = () => {
        log('WebSocket verbunden');
        updateConnectionStatus(true);
    };

    state.ws.onclose = () => {
        log('WebSocket getrennt, Reconnect in 5s...');
        updateConnectionStatus(false);
        setTimeout(connectWebSocket, CONFIG.reconnectInterval);
    };

    state.ws.onmessage = (event) => {
        handleServerMessage(JSON.parse(event.data));
    };
}

function sendMessage(msg) {
    if (state.ws?.readyState === WebSocket.OPEN) {
        state.ws.send(JSON.stringify(msg));
    } else {
        console.warn('WebSocket nicht verbunden');
    }
}
```

Listing 45: WebSocket-Verbindung mit Reconnect

Protokoll-Disziplin

Wenn du zusätzliche Message-Typen einführst, halte das Protokoll strikt (`type` Pflicht, Payload validieren). Sonst werden UI-Bugs zu „Netzwerkproblemen".

12.4 Medien-Preloading

Preload parallel, aber begrenzt – sonst überlastest du Browser und Server. Die Beobachtung: Viele gleichzeitige Requests können „stottern". Die Lösung: Eine Semaphore begrenzt die Concurrency.

```
class Semaphore {
    constructor(max) {
        this.max = max;
        this.current = 0;
        this.queue = [];
    }

    async acquire() {
        if (this.current < this.max) {
            this.current++;
            return;
        }
        await new Promise(resolve => this.queue.push(resolve));
        this.current++;
    }

    release() {
        this.current--;
        if (this.queue.length > 0) {
            this.queue.shift()();
        }
    }
}

async function preloadAllMedia() {
    const sem = new Semaphore(CONFIG.preloadConcurrency);
    const promises = [];

    for (let id = 1; id <= CONFIG.numMedia; id++) {
```

```
        promises.push(preloadMedia(id, sem));
    }

    await Promise.all(promises);
    state.preloaded = true;
}
```

Listing 46: Preloading mit begrenzter Concurrency

Tabelle 75: Empfohlene Concurrency-Werte

Szenario	Concurrency	Begründung
LAN / Pi lokal	4–8	Schnelle Verbindung
WLAN / Handy	2–4	Weniger Peaks
Mobiles Netz	1–2	Bandbreite schonen

12.4.1 Preload-Details

- **Bilder:** `new Image()` + `onload/onerror`, speichern in Cache
- **Audio:** `new Audio()` + `oncanplaythrough`, Timeout-Fallback nach 5000 ms

12.5 Playback-State-Machine: Preempt + Race-Fix

Bei schnellem Umschalten brauchen wir eine eindeutige Zuordnung von Events zur aktuellen ID. Sonst führen „alte"`ended`-Events zu falschen LED-Clears.

```
function handlePlay(id) {
    // Preempt: Vorheriges Audio stoppen
    if (state.currentId !== null) {
        stopCurrentAudio();
    }

    state.currentId = id;
    state.isPlaying = true;

    const cachedAudio = mediaCache.audio[id];
    cachedAudio.currentTime = 0;

```

```
    // Race-Fix: ended-Event an diese ID binden
    cachedAudio.onended = () => handleAudioEnded(id);

    cachedAudio.play();
    updateUI(id);
}

function handleAudioEnded(endedId) {
    // Ignorieren wenn nicht mehr aktuelle ID
    if (endedId !== state.currentId) {
        log(‘Ignoriere ended fuer ${endedId}, aktuell:
           ${state.currentId}‘);
        return;
    }

    state.isPlaying = false;
    state.currentId = null;
    sendMessage({ type: ’ended’, id: endedId });
}
```

Listing 47: Preempt und Race-Condition-Schutz

Erweiterte Absicherung

Für noch robustere Zuordnung bei Multi-Tab oder hoher Latenz: Ergänze eine Sequenznummer (`seq`) in `play`/`ended`. Der Server kann dann „alte endedßicher ignorieren.

12.6 Audio-Unlock: iOS/Autoplay-Policies

Audio darf erst nach User-Geste zuverlässig starten – besonders iOS/Safari. Daher: Unlock-Button + „silent play”.

```
async function unlockAudio() {
    try {
        // Methode 1: AudioContext
        const ctx = new (window.AudioContext ||
           window.webkitAudioContext)();
        if (ctx.state === ’suspended’) {
            await ctx.resume();
```

```
        }

        // Methode 2: Silent WAV abspielen
        const silentWav =
           'data:audio/wav;base64,UklGRigAAABXQVZFZm10...';
        const audio = new Audio(silentWav);
        await audio.play();
        audio.pause();

        state.audioUnlocked = true;
        elements.unlockBtn.hidden = true;

        // Jetzt Medien vorladen
        await preloadAllMedia();

    } catch (err) {
        log('Audio-Unlock fehlgeschlagen: ' + err.message);
    }
}
```

Listing 48: Audio-Unlock für iOS/Safari

Unlock-Prüfung

Alle Audio-Starts müssen hinter `state.audioUnlocked === true` liegen. Im `handlePlay` ist das so geprüft.

12.7 DOM-Integration

DOM-Zugriffe einmal bündeln, Rendering über klar definierte UI-Aktionen.

```
const elements = {
    unlockBtn: document.getElementById('unlock-btn'),
    waiting: document.getElementById('waiting'),
    mediaContainer: document.getElementById('media-container'),
    currentId: document.getElementById('current-id'),
    imageContainer: document.getElementById('image-container'),
    audio: document.getElementById('audio'),
    progressBar: document.getElementById('progress-bar'),
    debugPanel: document.getElementById('debug')
```

```
};
```

Listing 49: DOM-Element-Referenzen

12.7.1 Debug-Logging

```
function log(message) {
    const timestamp = new Date().toLocaleTimeString();
    console.log(`[${timestamp}] ${message}`);

    // In Debug-Panel schreiben (max 50 Zeilen)
    const line = document.createElement('div');
    line.textContent = `[${timestamp}] ${message}`;
    elements.debugPanel.appendChild(line);

    while (elements.debugPanel.children.length > 50) {
        elements.debugPanel.removeChild(elements.debugPanel.firstChild);
    }
}
```

Listing 50: Debug-Funktion mit UI-Output

Sicherheitshinweis

`innerHTML` ist ok, solange du nur kontrollierte Inhalte einsetzt. Bei künftig „freiem Textäus dem Server: auf `textContent` wechseln (XSS-Risiko vermeiden).

12.8 Protokoll-Übersicht

12.8.1 Server → Dashboard (WebSocket)

Tabelle 76: WebSocket-Nachrichten vom Server

Message	Beschreibung
`{"type:"play",id:n}`	Starte Wiedergabe für Taste n
`{"type:ßtop"}`	Stoppe aktuelle Wiedergabe

12.8.2 Dashboard → Server (WebSocket)

Tabelle 77: WebSocket-Nachrichten vom Dashboard

Message	Beschreibung
`{"type:ended",id:n}`	Audio für Taste n beendet

12.8.3 HTTP-Endpoints

Tabelle 78: Vom Dashboard genutzte HTTP-Endpoints

Endpoint	Beschreibung
`GET /`	Dashboard HTML
`GET /media/{id}.jpg`	Bild für Taste id
`GET /media/{id}.mp3`	Audio für Taste id
`GET /status`	Server-Status (JSON)
`GET /test/play/{id}`	Tastendruck simulieren

12.9 Checkliste für Erweiterungen

- ☐ Neues Protokollfeld: in `handleServerMessage()` validieren (Typ/Range)
- ☐ Neue UI-Anzeige: erst `elements` erweitern, dann dedizierte Render-Funktion
- ☐ Preload bei 100 Medien: Concurrency und Timeout realistisch wählen (5000 ms–15 000 ms)

12.10 Glossar

Tabelle 79: Begriffe aus der Dashboard-Entwicklung

Begriff	Erklärung
AudioContext	WebAudio-API-Kontext; wird genutzt, um Audio auf iOS nach User-Geste freizuschalten
Autoplay Policy	Browser-Regeln, die automatisches Abspielen ohne Nutzerinteraktion blockieren
Base64	Kodierung von Binärdaten als Text (hier: Silent-WAV als Data-URL)
Cache	Zwischenspeicher für Ressourcen (Bild/Audio) ohne Netz-Latenz
CloneNode	DOM-Methode zum Duplizieren eines Elements
Concurrency	Anzahl gleichzeitig laufender Operationen/Requests
DOMContentLoaded	Event, wenn das HTML geparst ist und DOM verfügbar
Event Listener	Registrierte Callback-Funktion für Events
Preempt	Neue Wiedergabe ersetzt sofort die laufende
Progress Bar	UI-Element für Audio-Fortschritt (`currentTime/duration`)
Race-Condition	Timing-Problem bei konkurrierenden Events
Reconnect	Automatisches Wiederverbinden nach Verbindungsabbruch
Semaphore	Synchronisationsmechanismus zur Begrenzung paralleler Tasks
WebSocket	Persistente bidirektionale Verbindung; `wss` ist TLS-verschlüsselt

13 USB-Port-Verwaltung

Auf dem Raspberry Pi 5 teilen sich zwei Projekte denselben ESP32: das Selection Panel und die AMR Platform. Doch der Serial-Port verträgt nur einen Zugriff gleichzeitig – sonst gehen Daten verloren oder Reads brechen ab. Wie lösen wir dieses Problem?

13.1 Das Exklusivitätsprinzip

Die Lösung liegt in einem gemeinsamen Lock via `flock` auf die Datei `/var/lock/esp32-serial.`
Beide Projekte respektieren diesen Lock, verhalten sich aber unterschiedlich:

- **Selection Panel (systemd):** Nutzt `flock -n` – startet nur, wenn der Lock frei ist, sonst Abbruch.

- **AMR micro-ROS Agent (Docker):** Nutzt `flock` ohne `-n` – wartet geduldig, bis der Lock frei wird.

Stabiler Device-Pfad

Statt `/dev/ttyACM0` (kann sich ändern) empfehlen wir den by-id-Pfad:
`/dev/serial/by-id/usb-Espressif_USB_JTAG_serial_debug_unit_98:3D:AE:EA:0`

Table 80 zeigt die beiden Projekte und ihre Zugriffsmethoden im Überblick.

Tabelle 80: Serial-Port-Nutzung der beiden Projekte

Projekt	Prozess	Startart	Serial-Port
Selection Panel	`server.py`	systemd (`selection-panel.service`)	by-id (stabil)
AMR Platform	`micro_ros_agent`	Docker Compose (`microros_agent`)	by-id (empfohlen)

13.2 Nach dem Reboot: Standard-Ablauf

Nach einem Neustart des Pi müssen wir entscheiden, welches Projekt den Port nutzen soll. Schauen wir uns beide Modi an.

13.2.1 Selection Panel Modus (UI + Taster/LEDs)

Wir starten den Service und prüfen, ob alles läuft:

```
sudo systemctl start selection-panel.service
sudo systemctl status selection-panel.service --no-pager
```

Das Dashboard erreichen wir im Browser unter `http://rover.local:8080/`. Falls mDNS nicht funktioniert, ermitteln wir die IP manuell:

```
hostname -I
# Ausgabe z.B.: 192.168.1.24 172.17.0.1
# Browser: http://192.168.1.24:8080/
```

IP-Adressen verstehen

Die erste Adresse (hier `192.168.1.24`) ist die LAN/WLAN-IP für den Browser. Die `172.17.0.1` gehört zur Docker-Bridge und ist für den externen Zugriff nicht relevant.

Für die Live-Logs nutzen wir:

```
sudo journalctl -u selection-panel.service -f
```

Wenn wir jetzt Taster 1–10 drücken, sehen wir im Log Meldungen wie `Button X gedrueckt`.

13.2.2 AMR Modus (micro-ROS Agent)

Bevor der Agent starten kann, müssen wir das Selection Panel sauber beenden – das gibt Lock und Port frei:

```
sudo systemctl stop selection-panel.service

cd /home/pi/amr/docker
sudo docker compose -p docker up -d microros_agent
```

Die Agent-Logs verfolgen wir mit:

```
sudo docker compose -p docker logs -f microros_agent
```

Falls das Selection Panel noch laufen sollte, wartet der Agent dank `flock` geduldig, statt den Port zu blockieren.

13.3 Schneller Wechsel ohne Reboot

Im Entwicklungsalltag wechseln wir häufig zwischen beiden Modi. Die folgenden Befehle ermöglichen einen sauberen Übergang.

13.3.1 Wechsel zum Selection Panel

```
cd /home/pi/amr/docker
```

```
sudo docker compose -p docker stop microros_agent

sudo systemctl start selection-panel.service
sudo journalctl -u selection-panel.service -f
```

13.3.2 Wechsel zur AMR Platform

```
sudo systemctl stop selection-panel.service

cd /home/pi/amr/docker
sudo docker compose -p docker up -d microros_agent
sudo docker compose -p docker logs -f microros_agent
```

13.4 Autostart konfigurieren

Soll das Selection Panel beim Boot automatisch starten?

```
# Autostart aktivieren
sudo systemctl enable selection-panel.service

# Autostart deaktivieren
sudo systemctl disable selection-panel.service
```

Empfehlung für AMR

Den AMR-Agent starten wir bewusst manuell mit `docker compose up -d microros_agent`. So ist immer klar definiert, wer den Port belegt.

13.5 Sanity Checks: Port und Lock prüfen

Wenn etwas nicht funktioniert, helfen diese Befehle bei der Diagnose:

```
# Wer haelt den USB-Port?
sudo fuser -v /dev/ttyACM0 || true

# Wer haelt den Lock?
sudo lslocks | grep esp32-serial || true
```

```
ls -l /var/lock/esp32-serial.lock || true
```

13.6 One-time Setup: Docker Compose mit Serial-Lock

Damit der AMR-Agent den Lock respektiert, passen wir die Docker-Compose-Konfiguration einmalig an. Die Datei liegt unter `/home/pi/amr/docker/docker-compose.yml`.

13.6.1 Backup erstellen

```
cd /home/pi/amr/docker
cp -a docker-compose.yml docker-compose.yml.bak.$(date +%F_%H%M%S)
```

13.6.2 Service-Definition anpassen

Wir modifizieren nur den Service `microros_agent`:

```
services:
  microros_agent:
    image: microros/micro-ros-agent:humble
    container_name: amr_agent
    network_mode: host
    privileged: true
    restart: always

    volumes:
      - /dev:/dev
      - /var/lock:/var/lock

    entrypoint: ["/bin/sh", "-lc"]
    command: >
      DEV="/dev/serial/by-id/usb-Espressif_USB_JTAG_serial_debug_unit_
      98:3D:AE:EA:08:1C-if00";
      echo "[microros_agent] waiting for lock (DEV=$DEV)";
      exec flock /var/lock/esp32-serial.lock
      /bin/sh /micro-ros_entrypoint.sh serial --dev "$DEV" -b
         921600
```

Listing 51: Lock-Wrapper für den microros_agent

13.6.3 Änderungen anwenden

```
cd /home/pi/amr/docker
sudo docker compose -p docker config >/dev/null
sudo docker compose -p docker up -d --force-recreate
   microros_agent
```

13.7 Troubleshooting

13.7.1 Selection Panel startet nicht

Die häufigste Ursache: Der Lock ist belegt, weil der AMR-Agent läuft. Das Selection Panel bricht dann absichtlich ab.

```
sudo journalctl -u selection-panel.service -n 120 --no-pager
sudo lslocks | grep esp32-serial || true
sudo fuser -v /dev/ttyACM0 || true
```

Fix: Den jeweils anderen Prozess stoppen – für Selection Panel also `sudo docker compose -p docker stop microros_agent`.

13.7.2 Device-Pfad hat sich geändert

Nach einem Firmware-Update oder bei einem neuen ESP32 kann sich der by-id-Pfad ändern:

```
ls -l /dev/serial/by-id/
ls /dev/ttyACM*
```

Fix: Den neuen Pfad in `server.py` (Selection Panel) und im `DEV="..."` der Compose-Datei eintragen.

13.7.3 Docker-Status prüfen

```
cd /home/pi/amr/docker
sudo docker compose -p docker ps
sudo docker compose -p docker logs --tail=120 microros_agent
```

14 Quickstart

In diesem Kapitel bringen wir das Selection Panel zum Laufen – Server und Dashboard in 5 Minuten. Wir gehen davon aus, dass SSH eingerichtet, das Repository geklont und der ESP32 geflasht ist.

Tabelle 81: Quickstart-Metadaten

Metadaten	Wert
Stand	2026-01-08
Version	2.5.2
Status	✓ Prototyp funktionsfähig

14.1 Voraussetzungen

Bevor wir starten, prüfen wir kurz die Voraussetzungen:

- ☐ SSH eingerichtet → Section 2
- ☐ Repository geklont → Section 16
- ☐ ESP32 geflasht → Section 4.4

14.2 Setup (einmalig)

Zunächst erstellen wir eine virtuelle Python-Umgebung und installieren die Abhängigkeiten:

```
ssh rover
cd ~/selection-panel

python3 -m venv venv
venv/bin/pip install -r requirements.txt
```

Minimale Abhängigkeiten

Nur `aiohttp` wird benötigt – kein pyserial mehr. Der Server nutzt direkte Dateizugriffe auf den Serial-Port.

14.3 Server starten

Jetzt starten wir den Server:

```
cd ~/selection-panel
source venv/bin/activate
python server.py
```

Bei erfolgreichem Start sehen wir folgende Ausgabe:

```
================================================
Auswahlpanel Server v2.5.2 (PROTOTYPE)
================================================
Medien: 10 erwartet (IDs: 001-010)
Taster: 1-10 (1-basiert)
Serial: /dev/serial/by-id/usb-Espressif...
HTTP:   http://0.0.0.0:8080/
ESP32 lokale LED: aktiviert
================================================
Medien-Validierung: 10/10 vollstaendig
================================================
Serial verbinde: /dev/serial/by-id/usb-Espressif...
Serial verbunden
```

Das Dashboard erreichen wir unter `http://rover:8080/`.

14.4 Dashboard nutzen

Das Dashboard führt uns durch den Startvorgang:

1. Dashboard öffnen: `http://rover:8080/`
2. **„Sound aktivieren"** Button klicken – wichtig für die Audio-Wiedergabe!
3. Warten auf Preload: „Lade Medien... 5/10"→ „Warte auf Tastendruck..."
4. Taster drücken → Bild und Ton werden sofort abgespielt

Latenz-Verhalten

Die LED leuchtet sofort (< 1 ms), die Wiedergabe erfolgt aus dem Cache (< 50 ms). Das System fühlt sich instantan an.

14.5 Testen ohne Hardware

Auch ohne angeschlossene Taster können wir das System testen. Die HTTP-API simuliert Tastendrücke:

```
# Wiedergabe simulieren (1-basiert!)
curl http://rover:8080/test/play/1
curl http://rover:8080/test/play/5
curl http://rover:8080/test/play/10

# Status abfragen
curl http://rover:8080/status | jq

# Health-Check
curl http://rover:8080/health | jq

# Wiedergabe stoppen
curl http://rover:8080/test/stop
```

14.6 Serial direkt testen

Für die Fehlersuche ist es hilfreich, die Serial-Kommunikation direkt zu beobachten:

```
# Stabilen Port ermitteln
SERIAL_PORT=$(ls /dev/serial/by-id/usb-Espressif* 2>/dev/null |
   head -1)
echo "Port: $SERIAL_PORT"

# Port konfigurieren
stty -F $SERIAL_PORT 115200 raw -echo

# Daten empfangen (Ctrl+C zum Beenden)
cat $SERIAL_PORT
```

Befehle können wir direkt an den ESP32 senden:

```
echo "PING" > $SERIAL_PORT
echo "STATUS" > $SERIAL_PORT
echo "LEDSET 001" > $SERIAL_PORT
echo "LEDCLR" > $SERIAL_PORT
```

14.7 Autostart einrichten

Damit der Server nach einem Reboot automatisch startet, installieren wir den systemd-Service:

```
sudo cp selection-panel.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable --now selection-panel.service
```

Status und Logs prüfen wir mit:

```
sudo systemctl status selection-panel
journalctl -u selection-panel -f
```

14.8 Troubleshooting

Table 82 listet die häufigsten Probleme und deren Lösungen.

Tabelle 82: Häufige Probleme und Lösungen

Problem	Lösung
`ModuleNotFoundError`	`venv/bin/pip install aiohttp`
`Permission denied: /dev/ttyACM0`	`sudo usermod -aG dialout $USER` → Neu einloggen
Port blockiert	`sudo fuser /dev/ttyACM0` → Prozess beenden
Kein Ton	„Sound aktivierenButton im Browser klicken
Server startet nicht	`journalctl -u selection-panel -f`
Taster nicht erkannt	Serial testen: `cat /dev/serial/by-id/usb-Espressif*`
Falsche Medien	Prüfen: `ls media/` (001.jpg bis 010.jpg)
Preload dauert lange	Medien komprimieren oder Concurrency erhöhen

14.9 Medien-Struktur

Die Medien folgen der 1-basierten Nummerierung mit Zero-Padding auf 3 Stellen:

```
media/
|-- 001.jpg  001.mp3
|-- 002.jpg  002.mp3
|-- ...
+-- 010.jpg  010.mp3
```

Für Tests generieren wir Platzhalter-Medien mit:

```
./scripts/generate_test_media.sh 10
```

14.10 Latenz-Budget

Tabelle 83: Latenz-Budget vom Tastendruck bis zur Wiedergabe

Komponente	Latenz
ESP32 LED	< 1 ms
Serial + Server	$\sim$10 ms
Dashboard (aus Cache)	< 50 ms
Gesamt	**< 70 ms**

14.11 Referenz-System

Tabelle 84: Referenz-System für diese Dokumentation

Komponente	Version
Raspberry Pi 5	4 GB RAM
Pi OS	Debian 13 (trixie)
Python	3.13+
aiohttp	3.9+
ESP32 Firmware	2.5.2
Server	2.5.2
Dashboard	2.5.1

14.12 Schnellreferenz

Table 85 fasst die wichtigsten Befehle zusammen.

Tabelle 85: Befehls-Schnellreferenz

Aktion	Befehl
Server starten	`python server.py`
Dashboard	`http://rover:8080/`
Status	`curl http://rover:8080/status`
Health	`curl http://rover:8080/health`
Test Play	`curl http://rover:8080/test/play/5`
Serial Monitor	`cat /dev/serial/by-id/usb-Espressif*`
Service Status	`sudo systemctl status selection-panel`
Service Logs	`journalctl -u selection-panel -f`

15 Befehlsreferenz

Diese Referenz fasst alle wichtigen Befehle für Deployment, Build, Test und Diagnose zusammen. Wir beginnen mit einer Schnellreferenz und gehen dann ins Detail.

15.1 Schnellreferenz

Tabelle 86: Wichtigste Befehle auf einen Blick

Aktion	Befehl
Server starten	`python server.py`
Dashboard öffnen	`http://rover:8080/`
Status abfragen	`curl http://rover:8080/status | jq`
Health-Check	`curl http://rover:8080/health`
Test-Play	`curl http://rover:8080/test/play/5`
Serial-Monitor	`cat /dev/serial/by-id/usb-Espressif*`
Firmware flashen	`pio run -t upload`
Service Status	`sudo systemctl status selection-panel`
Service Logs	`journalctl -u selection-panel -f`

15.2 Server starten

URL-Hinweis

`rover` funktioniert auf allen Geräten (Mac, iPhone, iPad) dank mDNS. Alternative: IP-Adresse direkt verwenden (`http://192.168.1.24:8080/`).[1]

```
# Auf dem Pi
ssh rover
cd ~/selection-panel
source venv/bin/activate
python server.py
```

Erwartete Ausgabe:

```
==================================================
Auswahlpanel Server v2.5.2 (PROTOTYPE)
==================================================
Medien: 10 erwartet (IDs: 001-010)
Serial:
   /dev/serial/by-id/usb-Espressif_USB_JTAG_serial_debug_unit_...
HTTP:   http://0.0.0.0:8080/
ESP32 lokale LED: aktiviert
==================================================
Serial verbunden
```

15.3 Serial direkt testen

15.3.1 Serial-Port finden

```
# Stabiler Pfad (empfohlen)
ls -la /dev/serial/by-id/usb-Espressif*

# Fallback
ls -la /dev/ttyACM*
```

15.3.2 Empfangen (ohne Server)

```
# Port konfigurieren (by-id Pfad verwenden)
SERIAL_PORT=$(ls /dev/serial/by-id/usb-Espressif* 2>/dev/null |
   head -1)
stty -F $SERIAL_PORT 115200 raw -echo

# Serial-Monitor (Ctrl+C zum Beenden)
cat $SERIAL_PORT
```

15.3.3 Befehle senden

```
# Verbindungstest
echo "PING" > $SERIAL_PORT
echo "STATUS" > $SERIAL_PORT
echo "VERSION" > $SERIAL_PORT
echo "HELP" > $SERIAL_PORT
```

15.3.4 LED-Befehle (1-basiert!)

```
# Einzelne LED (one-hot)
echo "LEDSET 001" > $SERIAL_PORT   # LED 1 ein
echo "LEDSET 005" > $SERIAL_PORT   # LED 5 ein
echo "LEDSET 010" > $SERIAL_PORT   # LED 10 ein

# Additiv
echo "LEDON 001" > $SERIAL_PORT    # LED 1 ein (additiv)
echo "LEDON 002" > $SERIAL_PORT    # LED 2 ein (additiv)
echo "LEDOFF 001" > $SERIAL_PORT   # LED 1 aus

# Alle LEDs
echo "LEDALL" > $SERIAL_PORT       # Alle ein
echo "LEDCLR" > $SERIAL_PORT       # Alle aus

# LED-Test (Lauflicht)
echo "TEST" > $SERIAL_PORT
echo "STOP" > $SERIAL_PORT         # Test stoppen
```

15.3.5 Diagnose-Befehle

```
# Status zeigt CURLED (aktuelle LED)
echo "STATUS" > $SERIAL_PORT
# Ausgabe:
# LEDS 0000100000
# CURLED 5          <-- Aktuelle LED (1-basiert)
# BTNS 0000000000
# HEAP 372756
# QOVFL 0
# MODE PROTOTYPE
```

15.3.6 Screen (interaktiv)

```
screen $SERIAL_PORT 115200
```

Screen beenden

Ctrl+A, dann K, dann Y

15.4 HTTP-Endpoints

```
# Status (JSON)
curl http://rover:8080/status | jq

# Health-Check (200 = healthy, 503 = degraded)
curl -w "%{http_code}" http://rover:8080/health

# Tastendruck simulieren (1-basiert!)
curl http://rover:8080/test/play/1
curl http://rover:8080/test/play/5
curl http://rover:8080/test/play/10

# Wiedergabe stoppen
curl http://rover:8080/test/stop
```

15.4.1 Status-Response (v2.5.2)

```
{
  "version": "2.5.2",
  "mode": "prototype",
  "num_media": 10,
  "current_button": 5,
  "ws_clients": 1,
  "serial_connected": true,
  "serial_port": "/dev/serial/by-id/usb-Espressif_USB_JTAG_...",
  "media_missing": 0,
  "missing_files": [],
  "esp32_local_led": true
}
```

Listing 52: Beispiel-Antwort von /status

15.5 Deployment (Mac → Pi)

15.5.1 Mit rsync

```
cd ~/selection-panel

rsync -avz --delete \
    --exclude='firmware' \
    --exclude='hardwaretest_firmware' \
    --exclude='venv' \
    --exclude='.git' \
    --exclude='__pycache__' \
    . pi@rover:/home/pi/selection-panel/
```

Tabelle 87: rsync-Flags

Flag	Bedeutung
-a	Archiv-Modus (Rechte, Zeiten erhalten)
-v	Verbose (zeigt Dateien)
-z	Komprimiert Übertragung
--delete	Löscht Dateien auf Ziel, die lokal fehlen

15.5.2 Mit Git (empfohlen)

```
# Auf dem Mac
git add -A && git commit -m "..." && git push

# Auf dem Pi
ssh rover "cd ~/selection-panel && git pull && sudo systemctl
   restart selection-panel"
```

15.6 Server-Steuerung (systemd)

```
# Starten
sudo systemctl start selection-panel

# Stoppen
sudo systemctl stop selection-panel

# Neu starten
sudo systemctl restart selection-panel

# Status pruefen
sudo systemctl status selection-panel

# Autostart aktivieren/deaktivieren
sudo systemctl enable selection-panel
sudo systemctl disable selection-panel
```

15.6.1 Logs (journalctl)

```
# Live-Logs
journalctl -u selection-panel -f

# Letzte 50 Zeilen
journalctl -u selection-panel -n 50

# Letzte Stunde
journalctl -u selection-panel --since "1 hour ago"
```

```
# Heute
journalctl -u selection-panel --since today
```

15.7 Firmware flashen (Mac)

```
cd ~/selection-panel/firmware

# Kompilieren
pio run

# Flashen
pio run -t upload

# Serial-Monitor (PlatformIO)
pio device monitor

# Flash + Monitor
pio run -t upload -t monitor

# Clean Build
pio run -t clean
```

15.8 Medien verwalten

15.8.1 Prüfen (1-basiert: 001–010)

```
# Auf dem Pi
cd ~/selection-panel

# Medien auflisten
ls -la media/

# Medien-Check
for i in $(seq -w 1 10); do
    echo -n "0$i: "
    [ -f "media/0$i.jpg" ] && echo -n "JPG OK " || echo -n "JPG
       -- "
```

```
    [ -f "media/0$i.mp3" ] && echo -n "MP3 OK" || echo -n "MP3 --"
    echo
done

# Anzahl pruefen
ls media/*.jpg 2>/dev/null | wc -l
ls media/*.mp3 2>/dev/null | wc -l
```

15.8.2 Generieren

```
# Test-Medien generieren (auf Mac)
./scripts/generate_test_media.sh 10    # Prototyp
./scripts/generate_test_media.sh 100   # Produktion
```

15.9 Diagnose

15.9.1 USB / Serial

```
# USB-Geraete
ls -la /dev/serial/by-id/usb-Espressif*
lsusb | grep -i espressif

# Wer nutzt den Serial-Port?
sudo fuser /dev/serial/by-id/usb-Espressif*

# Port freigeben (falls blockiert)
sudo systemctl stop selection-panel
sudo systemctl stop microros-agent.service  # falls vorhanden
```

15.9.2 Netzwerk

```
# IP-Adressen
ip addr | grep 192.168
hostname -I

# Hostname
```

```
hostname
```

15.9.3 Python

```
# Python-Version
python3 --version

# Pakete im venv
~/selection-panel/venv/bin/pip list

# Detailliert
~/selection-panel/venv/bin/pip freeze
```

15.9.4 System-Info

```
# Pi Hardware-Modell
cat /proc/device-tree/model

# Pi OS Version
cat /etc/os-release

# Kernel
uname -r

# Speicherplatz
df -h /

# RAM
free -h
```

15.10 Schnelltest (Komplettablauf)

Der folgende Ablauf testet das gesamte System:

1. **Server starten** (Terminal 1):

   ```
   ssh rover
   ```

```
cd ~/selection-panel && source venv/bin/activate && python
   server.py
```

2. **Dashboard öffnen** (Mac):

```
open http://rover:8080/
```

3. **Sound aktivieren:** Button im Browser klicken → Medien werden vorgeladen
4. **Alle 10 Taster durchdrücken:**
   - Jeder Taster sollte Bild + Ton abspielen
   - LED leuchtet sofort (< 1 ms)
   - Wiedergabe startet aus Cache (< 50 ms)
5. **Status prüfen:**

```
curl http://rover:8080/status | jq
```

15.11 Erwartete Ausgaben

15.11.1 Server-Log (erfolgreich)

```
Button 1 gedrueckt
GET /media/001.mp3 HTTP/1.1 206
Wiedergabe 1 beendet -> LEDs aus
```

15.11.2 Dashboard Debug-Panel

```
Preloading 10 Medien...
Preload abgeschlossen: 10/10 OK (1823ms)
RX: {"type": "play", "id": 1}
Bild aus Cache: 001 (instant)
Audio aus Cache gestartet: 001 (12ms)
Audio beendet: 1
TX: {"type":"ended","id":1}
```

15.11.3 Serial-Monitor

```
PRESS 001
RELEASE 001
```

15.11.4 Status-Endpoint

```
{
  "version": "2.5.2",
  "mode": "prototype",
  "num_media": 10,
  "current_button": null,
  "ws_clients": 1,
  "serial_connected": true,
  "esp32_local_led": true
}
```

16 Git-Workflow

Wie verwalten wir den Quellcode des Selection Panels? Dieses Kapitel beschreibt den Git-Workflow für die Zusammenarbeit zwischen Entwicklungsrechner, Raspberry Pi und GitHub.

Repository: `github.com:unger-robotics/selection-panel`

16.1 Repository klonen

```
git clone git@github.com:unger-robotics/selection-panel.git
cd selection-panel
```

Voraussetzung

SSH-Key für GitHub muss eingerichtet sein → Section 2.4

16.2 Erstmaliges Setup (neues Repo)

Falls wir ein neues Repository anlegen:

```
cd ~/selection-panel
git init
git add .
git commit -m "feat: Initial commit"
git branch -M main
git remote add origin
   git@github.com:unger-robotics/selection-panel.git
git push -u origin main
```

16.3 Täglicher Workflow

Der typische Arbeitsablauf besteht aus vier Schritten:

```
git pull                    # Aenderungen holen
git add .                   # Aenderungen stagen
git commit -m "feat: ..."   # Committen
git push                    # Pushen
```

So bleiben alle drei Ebenen synchron: Mac – Rover (Pi) – GitHub.

```
# Status pruefen
git fetch origin && git status
```

16.4 Commit-Konventionen

Wir verwenden Conventional Commits für konsistente Commit-Messages.

Tabelle 88: Commit-Präfixe und ihre Verwendung

Präfix	Verwendung	Beispiel
`feat`	Neue Funktion	`feat(server): WebSocket-Broadcast`
`fix`	Bugfix	`fix(firmware): LED-Index korrigiert`
`docs`	Dokumentation	`docs: HARDWARE.md erweitert`
`perf`	Performance	`perf: Latenz-Optimierung`
`refactor`	Umstrukturierung	`refactor: shift_register modularisiert`
`chore`	Build, Tooling	`chore: requirements.txt vereinfacht`

Scopes: `firmware`, `server`, `dashboard`, `docs`

16.5 Branching

Für größere Features erstellen wir einen eigenen Branch:

```
# Feature-Branch erstellen
git checkout -b feature/led-animation
git add . && git commit -m "feat: LED-Animation"
git push -u origin feature/led-animation

# Nach Review: Mergen
git checkout main
git pull
git merge feature/led-animation
git push
git branch -d feature/led-animation
```

Branch-Namenskonvention

Wir verwenden das Muster `feature/beschreibung` für neue Features, `fix/beschreibung` für Bugfixes und `docs/beschreibung` für Dokumentationsänderungen.

16.6 Häufige Befehle

Tabelle 89: Git-Schnellreferenz

Befehl	Aktion
`git status`	Was hat sich geändert?
`git log –oneline -10`	Letzte 10 Commits
`git diff`	Änderungen anzeigen
`git stash`	Änderungen parken
`git stash pop`	Änderungen zurückholen
`git commit –amend -m "..."`	Letzten Commit korrigieren
`git tag -a v2.5.2 -m "..."`	Release-Tag erstellen
`git push origin v2.5.2`	Tag pushen

16.7 Deployment (Mac → Pi)

Zwei Optionen für das Deployment auf den Raspberry Pi:

16.7.1 Option 1: rsync

```
rsync -avz --delete \
  --exclude='firmware' \
  --exclude='hardwaretest_firmware' \
  --exclude='venv' \
  --exclude='.git' \
  --exclude='__pycache__' \
  . pi@rover:~/selection-panel/

ssh rover 'sudo systemctl restart selection-panel'
```

16.7.2 Option 2: Git (empfohlen)

```
ssh rover 'cd ~/selection-panel && git pull && sudo systemctl
   restart selection-panel'
```

Empfehlung

Die Git-Variante ist sauberer, weil sie nur committete Änderungen überträgt und die Historie konsistent bleibt.

16.8 .gitignore

Diese Dateien und Verzeichnisse schließen wir von der Versionskontrolle aus:

```
firmware/.pio/
firmware/.vscode/
hardwaretest_firmware/*/.pio/
hardwaretest_firmware/*/.vscode/
__pycache__/
*.pyc
venv/
.DS_Store
docs/_site/
```

16.9 Git-Aliase

Für häufige Befehle lohnen sich Aliase. In `~/.gitconfig`:

```
[alias]
    st = status
    lg = log --oneline --graph
    co = checkout
    br = branch
```

Dann genügt: `git st`, `git lg`, etc.

16.10 Vim + Git

Git verwendet standardmäßig Vim für Commit-Messages. Die wichtigsten Befehle:

Tabelle 90: Vim-Befehle für Git

Tastenkombination	Aktion
`i`	Insert-Modus (Text eingeben)
`Esc`	Normal-Modus
`:wq`	Speichern und beenden (Commit ausführen)
`:q!`	Beenden ohne Speichern (Commit abbrechen)

Editor ändern

Wer VS Code bevorzugt: `git config –global core.editor ”code –wait”`

A Glossar

Dieses Glossar erklärt die wichtigsten Begriffe des Selection-Panel-Projekts. Die Einträge sind thematisch gruppiert und folgen dem Muster: Regel, Beispiel, Anwendung.

Begriffskategorien

- **JSON** = Datenformat (Text-Notation für Schlüssel/Wert und Listen)
- **WebSocket / UART / SPI** = Transportwege (Kommunikationsprotokolle)
- **Server / Raspberry Pi / ESP32-S3** = Rollen/Computer
- **Schieberegister** = Hardware-Trick (Serial ↔ Parallel)

A.1 WebSocket

WebSocket ist eine **dauerhafte TCP-Verbindung** zwischen Browser und Server. Der Server kann sofort pushen, ohne dass der Browser ständig pollt (Polling = zyklisches Nachfragen per HTTP).

Beispiel: Browser verbindet sich auf `/ws`. Server sendet:

```
{”type”:”play”,”id”:42}
```

Anwendung: Der Pi-Server broadcastet das Event an alle verbundenen Browser – Bild/Audio startet ohne neue HTTP-Anfrage pro Tastendruck.

A.2 JSON

JSON ist **Text**, der Daten als Schlüssel/Wert-Paare und Listen darstellt. Leicht im Browser (JavaScript) und in Python zu parsen (parsen = Text in Datenstruktur umwandeln).

```
{"type":"PRESS","id":42,"t_ms":123456}
```

Anwendung: Einheitliches Format für Events (`PRESS`, `RELEASE`, `play`, `stop`) zwischen Pi und Browser.

A.3 Server

Ein Server ist ein Programm, das **Anfragen annimmt** (HTTP/WebSocket) und **Antworten/Ereignisse liefert**.

Beispiel: `server.py` macht typischerweise:

- `GET /` → liefert `index.html`
- `GET /media/...` → liefert Mediendatei
- `WS /ws` → hält Verbindung offen und sendet Events

Anwendung: Der Raspberry Pi ist der Koordinator: nimmt UART-Events vom ESP an, verwaltet Medien und verteilt `play`/`stop` an Browser.

A.4 UART (Serial)

UART ist eine **asynchrone serielle** Punkt-zu-Punkt-Verbindung (TX/RX + GND). Üblich: 115 200 baud, 8N1.

Tabelle 91: UART-Parameter

Parameter	Bedeutung
Baud	Symbole pro Sekunde (hier praktisch Bitrate)
8N1	8 Datenbits, kein Paritätsbit, 1 Stopbit

Berechnung: Bei 115 200 baud und 8N1:

$$\frac{115\,200\,\text{bit/s}}{10\,\text{bit/B}} \approx 11\,520\,\text{B/s} \approx 11{,}5\,\text{kB/s} \tag{8}$$

Anwendung: ESP32-S3 → Pi sendet kurze Textzeilen wie `PRESS 042\n`. Das ist schnell genug, weil pro Event nur wenige Bytes übertragen werden.

A.5 SPI (und „SPI-ähnlich")

SPI ist eine **synchrone** serielle Verbindung: Master erzeugt Clock, Daten werden pro Takt geschoben.

Tabelle 92: SPI-Signale

Signal	Funktion
SCLK	Clock (Taktgeber)
MOSI	Daten zum Slave
MISO	Daten zurück zum Master
CS/Latch	Rahmen/Übernahme

Anwendung: Der ESP32 taktet Bits in/aus die Schieberegister. Das ist „SPI-ähnlich", weil zusätzlich Latch/Load-Signale benötigt werden.

A.6 Schieberegister (74HC595 / CD4021B)

Schieberegister wandeln **Serial** ↔ **Parallel**. Mehrere lassen sich kaskadieren (QH' → nächstes SER).

Beispiel für 100 Buttons:

- 13× 74HC595 → $(13 \cdot 8 = 104)$ LED-Ausgänge
- 13× CD4021B → $(13 \cdot 8 = 104)$ Taster-Eingänge

Anwendung:

- **74HC595 (Output):** ESP schiebt 100 Bits → Latch → alle LEDs aktualisieren gleichzeitig
- **CD4021B (Input):** ESP löst Parallel-Load aus → schiebt 100 Bits raus und liest sie ein

A.7 Raspberry Pi

Ein Raspberry Pi ist ein **Linux-Single-Board-Computer**: Dateisystem, Netzwerk, HDMI, Audio, Python-Server.

Anwendung im Projekt:

- Hostet Web-App (HTML/JS/CSS) + Medien-Dateien
- Läuft `server.py` (aiohttp): WebSocket + Datei-Serving
- Liest USB-Serial vom ESP32-S3
- Verteilt Events an Browser und triggert Medienwiedergabe

A.8 ESP32-S3 (Seeed XIAO)

ESP32-S3 ist ein **Mikrocontroller**: sehr schnell für I/O, deterministisch, echtzeit-nah. Das Seeed XIAO ist das Board-Layout mit USB, Pins, Regler.

Anwendung im Projekt:

- Scannt Taster über CD4021B (schnell, periodisch)
- Setzt LEDs über 74HC595 (schnell, gelatcht)
- Entprellt und erzeugt Events
- Sendet Events über UART an den Pi
- FreeRTOS Dual-Core: io_task und serial_task auf Core 1

A.9 74HC595 Pinout (DIP-16)

Tabelle 93: 74HC595 Pin-Verbindungen

Pin	Name	Verbindung
14	SER	← `D10` (MOSI) oder vorheriger QH'
11	SRCLK	← `D8` (SCK, shared)
12	RCLK	← `D0` (Latch)
9	QH'	→ nächster SER oder offen
10	SRCLR	→ VCC (nicht löschen)
13	OE	← `D2` (PWM) oder → GND

SPI-Mode: MODE0 (CPOL=0, CPHA=0) @ 1 MHz

A.10 CD4021B Pinout (DIP-16)

Tabelle 94: CD4021B Pin-Verbindungen

Pin	Name	Verbindung
3	Q8	→ `D9` (MISO) oder nächster DS
10	CLK	← `D8` (SCK, shared)
9	P/S	← `D1` (Load-Signal)
11	DS	← **VCC** (letzter IC!) oder vorheriger Q8

P/S = HIGH für Load

Der CD4021B hat invertierte Load-Logik im Vergleich zum 74HC165!

SPI-Mode: MODE1 (CPOL=0, CPHA=1) @ 500 kHz

A.11 Bit-Mapping

A.11.1 Button-Verdrahtung (CD4021B)

Tabelle 95: Taster-Bit-Zuordnung

Taster	Pin	PI-Eingang	Bit
BTN 1	Pin 1	PI-8	Bit 0
BTN 2	Pin 15	PI-7	Bit 1
BTN 3	Pin 14	PI-6	Bit 2
BTN 4	Pin 13	PI-5	Bit 3
BTN 5	Pin 4	PI-4	Bit 4
BTN 6	Pin 5	PI-3	Bit 5
BTN 7	Pin 6	PI-2	Bit 6
BTN 8	Pin 7	PI-1	Bit 7

Formel: $\text{btn\_bit}(id) = (id - 1) \mod 8$

A.11.2 LED-Verdrahtung (74HC595)

Tabelle 96: LED-Bit-Zuordnung

LED	Pin	Ausgang	Bit
LED 1	Pin 15	QA	Bit 0
LED 2	Pin 1	QB	Bit 1
LED 3	Pin 2	QC	Bit 2
LED 4	Pin 3	QD	Bit 3
LED 5	Pin 4	QE	Bit 4
LED 6	Pin 5	QF	Bit 5
LED 7	Pin 6	QG	Bit 6
LED 8	Pin 7	QH	Bit 7

Formel: $\text{led\_bit}(id) = (id - 1) \mod 8$

A.12 Taster-Input Decoder (Active-Low)

Active-Low: gedrückt $\Rightarrow$ Bit $= 0$ (Pull-up Widerstand $10\,\text{k}\Omega$)

Beispiel: BTN 5 gedrückt

```
Bit:    7    6    5    4    3    2    1    0
BTN:    8    7    6    5    4    3    2    1
Wert:   1    1    1    0    1    1    1    1
                      ^
                 BTN 5 gedrueckt (Bit 4 = 0)
```

Hex: `0xEF` (1110 1111)

```
1 bool isPressed = !((stream >> btn_bit(5)) & 1);  // btn_bit(5) = 4
```

Listing 53: Taster-Abfrage im Code

A.13 LED-Output Decoder (Active-High)

Active-High: an $\Rightarrow$ Bit $= 1$

Beispiel: LED 5 an

```
Bit:    7    6    5    4    3    2    1    0
LED:    8    7    6    5    4    3    2    1
Wert:   0    0    0    1    0    0    0    0
                      ^
                 LED 5 an (Bit 4 = 1)
```

Hex: `0x10` (0001 0000)

```
led_set(ledBytes, 5, true);  // setzt Bit 4
```

Listing 54: LED setzen im Code

A.14 Skalierungsbeispiel: LED 100

100-LED-Config: `NUM_LEDS = 100` → `NUM_BYTES = 13`

Ziel: Nur LED 100 an

A.14.1 Berechnung

$$\text{idx0} = 100 - 1 = 99 \tag{9}$$

$$\text{ic} = \lfloor 99/8 \rfloor = 12 \quad \text{(IC \#12, letzter in der Kette)} \tag{10}$$

$$\text{bit} = 99 \mod 8 = 3 \quad \text{(Bit 3)} \tag{11}$$

$$\text{mask} = 1 \ll 3 = \texttt{0x08} \tag{12}$$

A.14.2 Array-Zustand und SPI-Transfer

```
ledBytes[12] = 0x08;  // LED100 an
ledBytes[0..11] = 0x00;

// Hinten zuerst, vorne zuletzt
for (int i = NUM_BYTES - 1; i >= 0; i--) {
    SPI.transfer(ledBytes[i]);
}
digitalWrite(PIN_LED_RCK, HIGH);  // Latch
digitalWrite(PIN_LED_RCK, LOW);
```

Listing 55: LED 100 setzen und übertragen

A.15 Binär-Hex-Tabelle

Tabelle 97: Binär-Hexadezimal-Dezimal Umrechnung

Binär	Hex	Dezimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

A.16 Firmware-Architektur (FreeRTOS)

Die Firmware verwendet **FreeRTOS Dual-Core** auf dem ESP32-S3.

Tabelle 98: FreeRTOS Tasks

Task	Core	Priorität	Periode	Funktion
io_task	1	5	5 ms	Hardware-I/O (Taster, LEDs)
serial_task	1	2	Event-driven	Protokoll-Handler

Design-Prinzipien:

- **Queue-basiert:** io_task sendet LogEvents über FreeRTOS-Queue an serial_task
- **Mutex-geschützt:** SPI-Bus wird durch Mutex vor gleichzeitigem Zugriff geschützt

- **RAII:** SpiGuard für automatisches Cleanup
- **Zeitbasiertes Debouncing:** Jeder Taster hat eigenen Timer (30 ms)
- **Bitfeld-basiert:** LEDs/Taster als Byte-Arrays mit Maskenoperationen

Core-Zuordnung

Core 0 bleibt für WiFi/BLE reserviert (falls später benötigt). Die I/O-Tasks laufen auf Core 1.