

Projektaufgabe Embedded Systems SS2024

Tim Krüger, 1197031

13.08.2024

Inhaltsverzeichnis

1 Allgemeines	2
1.1 Persönliche Angaben	2
1.2 Eigenständigkeitserklärung	2
2 Einleitung	3
2.1 Motivation	3
2.2 Aufgabenstellung	3
3 Installation	4
3.1 Projektstruktur	4
3.2 Netzwerkboot	4
3.3 Buildroot	5
3.3.1 Installationshinweise	5
3.4 Kernel	5
3.4.1 Installationshinweise	5
3.5 WLAN-AP	5
3.6 Hardwareaufbau	6
3.7 Gerätetreiber	7
3.7.1 Allgemeines	7
3.7.2 Implementierung	7
3.7.3 Verwendung	9
3.8 MQTT-Konfiguration	9
4 Systemtest	10
4.1 Testplan	10
4.2 Komponententest	10
4.2.1 Boot und Entwicklung	10
4.2.2 WLAN	10
4.2.3 LED-Treiber	12
4.2.4 MQTT	12
4.3 Test des Gesamtsystems	13
5 Zusammenfassung	13

1 Allgemeines

1.1 Persönliche Angaben

Name: Tim Krüger

Matrikelnummer: 1197031

Studiengang: Master Informatik

Datum: 13.08.2024

1.2 Eigenständigkeitserklärung

Eidesstattliche Erklärung
zur Projektarbeit Eingebettete Systeme im SS2024

Name:

Matrikelnummer:

Ich versichere durch meine Unterschrift, dass die vorgelegte Arbeit ausschließlich von mir erstellt und verfasst wurde. Es wurden keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt.

Krefeld, 13.08.24

Ort, Datum

Krüger

Unterschrift

Abbildung 1: Eigenständigkeitserklärung

2 Einleitung

2.1 Motivation

2.2 Aufgabenstellung

3 Installation

3.1 Projektstruktur

Als Teil dieses Projekts werden alle notwendigen Dateien (Dokumentation, Configs, Skripte etc.) mitgeliefert. Die Struktur des Dateibaums ist im folgenden dargestellt:

```
Dev/ESY/
|-- buildroot/
|-- configs/
|-- devicetree/
|-- docs/
|   |-- res/
|-- kernel/
|-- modules/
|-- scripts/
|-- target/
`-- txts/
```

- *Dev/ESY/*: Das Root-Verzeichnis des Projekts
 - Bei einer anderen Verzeichnisstruktur müssen die Skripte *post-build.sh* und *post-image.sh* entsprechend angepasst werden
- *buildroot/*: Das Buildroot-Verzeichnis.
 - Genaue Hinweise folgen in Kapitel 3.3
- *configs/*: Verzeichnis für die Buildroot- und Kernel-Konfigurationsdateien
- *devicetree/*: Verzeichnis für den Device-Tree-Blob zur Ansteuerung der GPIOs
- *docs/*: Dokumentation des Projekts inkl. dem Abbildungsverzeichnis *res/*
- *kernel/*: Bereits vorkompilierte Kernel
 - Genaue Hinweise folgen in Kapitel 3.4
- *modules/*: Der Treiber bzw. das Kernelmodul zur Ansteuerung der LED
 - Genaue Hinweise folgen in Kapitel 3.7
- *scripts/*: Verschiedene Skripte (Teil des Entwicklungsprozesses)
- *target/*: Verschiedenen Skripte und Konfigurationsdateien, welche auf den Raspberry Pi kopiert werden
- *txt/*: Verschiedene Textdateien, hauptsächlich zur initialen TFTP-Konfiguration

3.2 Netzwerkboot

Für einen schnellen Entwicklungsprozess (Iterationsgeschwindigkeit + Deployment) wird Netzwerkboot via TFTP verwendet. Der Raspberry Pi zieht sich hierbei, bei korrekter Konfiguration, alle notwendigen Dateien des Bootprozesses von einem TFTP-Server, welcher auf dem Host-Rechner läuft.

Ein solches Setup kann mit den folgenden Schritten ans Laufen gebracht werden:

- Installation und Starten eines TFTP-Servers auf der Host-Maschine
 - Monitoren des Outputs bspw. via: *tail -f /var/log/syslog | grep tftp*
 - Empfehlung: Extensives Logging aktivieren
- Originale Pi4-Bootfiles herunterladen und ins TFTP-Verzeichnis kopieren
- Bootloader des Raspberry Pi's **muss** für Netzwerkboot angepasst werden
 - Dafür muss der Pi entsprechend geflashed werden
- Anpassung der Netzwerkschnittstellen
 - Zuweisung einer statischen IP ans Ethernet-Interface des Host-PC
 - Zuweisung und Konfiguration des Ethernet-Interface auf dem Pi (erfolgt über die Datei */target/interfaces*)

Es ist außerdem empfehlenswert das serielle Interface des Pi's für Output und Debugging zu verwenden. Die notwendigen Konfigurationsdateien werden bereitgestellt:

- Kopieren der Konfigurationsdateien
 - */txt/cmdline.txt* nach */srv/tftp/*
 - */txt/config.txt* nach */srv/tftp/*
- Installation eines beliebigen Terminalemulators, z.B. *Minicom*
 - Starten bspw. via: *sudo minicom -D /dev/ttyUSB0*

3.3 Buildroot

Als Systembuilder wird in diesem Projekt *Buildroot* verwendet. *Buildroot* ist im Endeffekt eine Sammlung von Makefiles, welche einem in einem kernelähnlichen Interface verschiedene Konfigurationsoptionen bieten. Buildroot wurde in diesem Projekt ausschließlich zum Bau des Userlands verwendet, der Kernel wurde *per Hand* kompiliert. Die exakten Konfigurationsschritte werden an dieser Stelle ausgelassen, da die Buildroot-Config zur Verfügung gestellt wird.

3.3.1 Installationshinweise

- Buildroot installieren
 - Git-Repository nach *buildroot/* clonen
 - *config/config.buildroot* nach *buildroot/* kopieren
 - *make menuconfig* aufrufen und Auswahl speichern
- Durch ein anschließendes Aufrufen von *make* wird das Userland gebaut
- Die Skripte *post-build.sh* und *post-image.sh* klinken sich in diesen Prozess ein und kopieren die notwendigen Dateien an die richtigen Stellen im Image des TFTP-Servers
- **Achtung:** Auf korrekte Pfade achten
 - Die Pfade zu den Skripten können per *make menuconfig* geändert werden
 - Die Pfade innerhalb der Skripte müssen natürlich ebenfalls stimmen (Root-Verzeichnis oben im Skript als Variable hinterlegt)
 - Extensives Logging ist aktiviert und sollte an dieser Stelle schnell Auskunft geben können

3.4 Kernel

Ein bereits crosskompilierter Kernel, welcher WLAN- und Modulefähig ist, findet sich im Verzeichnis *kernel/*. Es handel sich dabei um einen Kernel mit der Version 6.6.4. Dieser Kernel lief in meinen Tests am besten. Speziell die neueren Kernelversionen ≥ 6.9 haben bei mir Probleme gemacht. Bspw. lief der Arbeitsspeicher auf meiner Maschine beim Kompilieren von Kernel 6.9.1 voll, was schließlich zu einem Crash führte.

3.4.1 Installationshinweise

Falls Sie einen eigenen Kernel für dieses Projekt kompilieren wollen, stehen alle notwendigen Dateien zur Verfügung.

- Clonen des entsprechenden Kernels von Git
- Hinterlegen der notwendigen UmgebungsvARIABLEN durch: *source /scripts(exports).sh*
- *config/config.linux* ins Kernel-Verzeichnis kopieren
- Kompilieren durch: *time make -j ihr_name dtbs modules*
- Der neue Kernel muss dann noch namentlich im *post-image.sh* Skript hinterlegt werden
- Beim nächsten Aufruf von *make* innerhalb von *Buildroot* wird dieser dann kopiert

3.5 WLAN-AP

Ein Teil der Aufgabenstellung war die Konfiguration des Raspberry Pi's als WLAN Accesspoint. Der Kernel und das Userland wurden dafür entsprechend präpariert. Aufgespannt wird das WLAN *VimIsTheBest*. Verteilt werden IP-Adressen aus dem Netz 192.168.123.0/24. Skripte und Konfigurationsdateien finden sich im Verzeichnis *target/*. Alle Dateien werden nach *init.d* kopiert und automatisch beim Bootvorgang gestartet.

Die WLAN-AP-Konfiguration umfasst:

- WPA
- DNS und DHCP
- Firewall mit NAT und IP-Forwarding

Unter Umständen taucht beim Booten die Fehlermeldung *brcmfmac: brcmf_set_channel: set chanspec 0x100d fail, reason -52* auf. Diese wird von der WLAN-Hardware ausgelöst.

- Dabei handelt es sich um einen bekannten Bug, bisher ohne richtigen Fix:
 - Siehe z.B. [<https://github.com/raspberrypi/linux/issues/6049>]
- Der Bug betrifft sowohl Raspberry Pi's der Version 4 als auch der Version 5
- Er tritt, laut meiner Recherche, seit den Kernelversionen ≥ 6.3 auf
- Der Bug hat aktuell keinen Effekt auf uns, sollte aber beobachtet werden

3.6 Hardwareaufbau

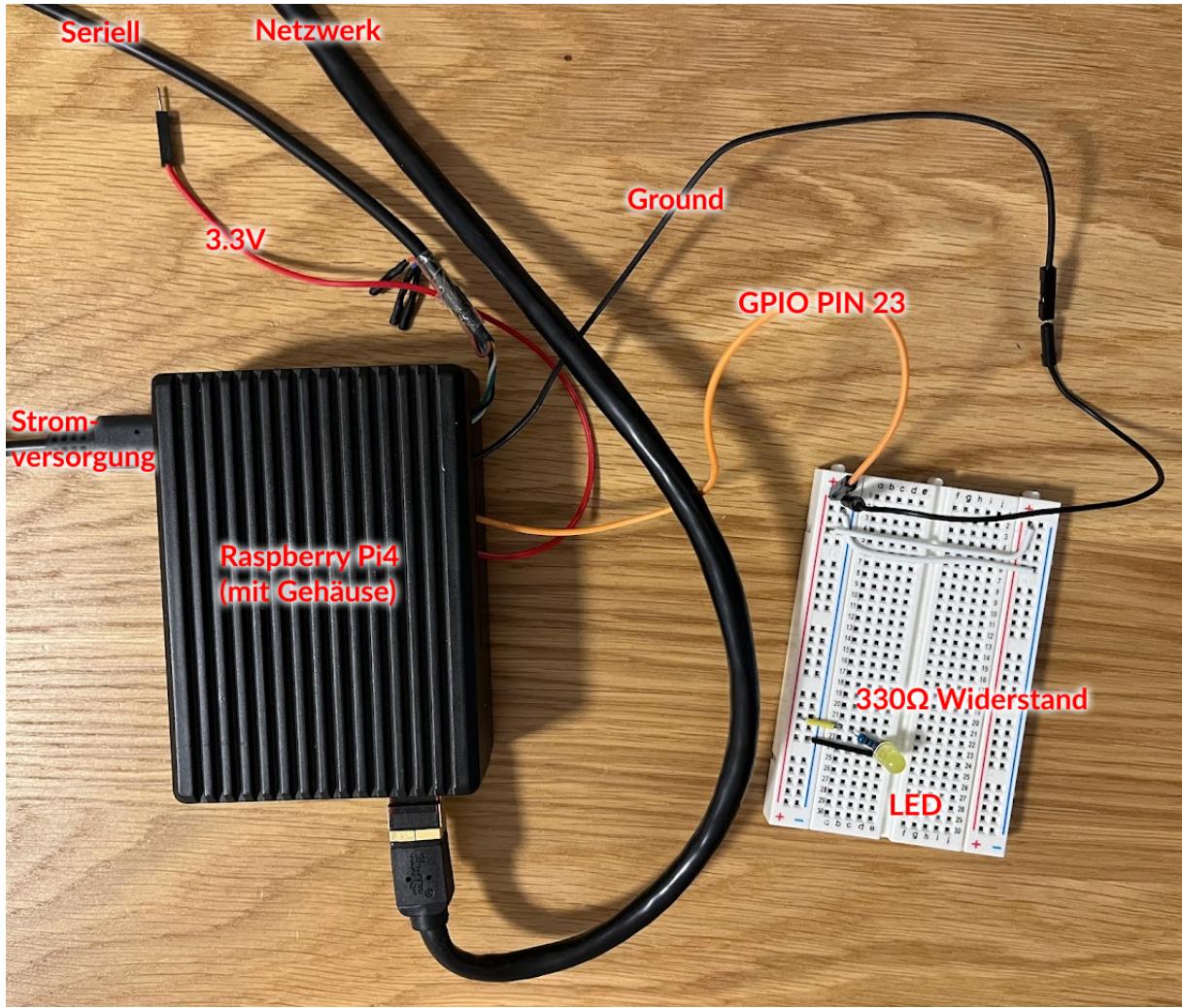


Abbildung 2: Hardwareaufbau

Der Hardware- bzw. Schaltungsaufbau kann *Abbildung 2* entnommen werden. Es handelt sich dabei um eine sehr einfache Schaltung welche Demonstrationszwecken dient. Folgende Aspekte können durch den Aufbau demonstriert werden:

- Logging via serieller Schnittstelle
- Booting via Netzwerk (TFTP)
- Ansteuerung eines GPIO Pin's mittels des Device-Tree's
 - Später: Integration in Kernelmodul und Fernsteuerung per MQTT
- Der Pi ist in meinem Fall noch mit einem optionalen Gehäuse ausgestattet, welches Schutz- und Kühlungszwecken dient

3.7 Gerätetreiber

Bei dem in diesem Projekt integrierten Gerätetreiber handelt es sich um ein Kernelmodul, welches die Ansteuerung einer LED über den Device-Tree implementiert.

Im folgenden wird kurz auf einige entwicklungstechnische Aspekte eingegangen, bevor Implementierung und Verwendung des Treibers erläutert werden.

3.7.1 Allgemeines

Der Gerätetreiber *signalru.c* befindet sich im Verzeichnis */modules/signalru/*. Zur Neukompilierung wird ein Makefile bereitgestellt. Zuvor müssen, identisch wie beim Kompilieren des Kernels, die UmgebungsvARIABLEN zum Crosskompilieren geladen werden (via *exports.sh*). Das generierte Kernelmodul wird beim Neuladen der Buildroot-Config (via *make*) automatisch gefetched und nach */lib/modules/* kopiert. Ein Startskript, welches das Kernelmodul automatisch beim Booten lädt, wird ebenfalls bereitgestellt. Bei Umbenennung oder Ergänzung von neuen Kernelmodulen müssen diese Skripte entsprechend angepasst werden.

Wie bereits erwähnt werden die GPIOs in diesem Projekt über den Device-Tree angesprochen. Die zuvor verwendete GPIO-API (*gpio_to_desc* und Kollegen) scheint deprecated zu sein. Auch mit Kernelversion 6.6 funktionierte bei mir nur die Version über den Device-Tree.

Zunächst wird ein Device-Tree-Blob-Overlay erstellt, welches GPIO PIN 23 benennt (*/device-tree/custom_gpio.dts*). Diese Datei wird dann mit dem Device-Tree-Compiler kompiliert (*device-tree/custom_gpio.dtbo*) und nach */srv/tftp/overlays* kopiert. Das Device-Tree-Blob-Overlay muss außerdem noch in die */txts/config.txt* eingetragen werden.

3.7.2 Implementierung

Die Implementierung des Kernelmoduls bestand aus dem Schreiben der folgenden Funktionen:

- Funktionen für Konfiguration und Speicherfreigabe der GPIO's
- *sig_driver_init* Funktion welche die grundlegende Initialisierung aller benötigten Komponenten beim Laden des Treibers vornimmt
 - Character device
 - Waitqueue
 - Kernelthread
 - ...
- *sig_driver_exit* Funktion welche alles wieder deallokiert und freigibt
- *sig_write* Funktion welche das Schreiben auf die Gerätedatei regelt
 - User-Input zum Setzen der LED-Frequenz
- *sig_read* Funktion welche das Lesen der Gerätedatei regelt
 - Ausgabe der aktuellen LED-Frequenz an den User
- *blink_thread_func* Kernelthread-Funktion welche in Abhängigkeit vom User-Input eine LED, mit einer bestimmten Frequenz, toggelt

sig_write erlaubt das An- und Ausschalten der LED, sowie das Setzen einer Blinkfrequenz. User-Input wird dabei als String entgegen genommen und intern konvertiert. *sig_write* ruft dafür die Funktion *validate_user_input* auf. Falls die Eingabe valide ist werden eine Zustandvariable und die Frequenz entsprechend neu gesetzt. Erst dann wird der Kernelthread via *wake_up_interruptible()* aufgeweckt. Um einen gesicherten asynchronen Zugriff zu gewährleisten, wurde ein Spinlock verwendet.

```
static void validate_user_input(char* buffer, unsigned long buf_size)
{
    // Convert and validate user input
    [...]

    // Enter critical section: Set global variable to provided user input
    spin_lock(&freq_lock);
    freqKrueger = user_input;
    new_input = true;
    spin_unlock(&freq_lock);

    // Wake up thread
    wake_up_interruptible(&freq_wq);
}
```

Realisiert wird das Blinken bzw. Toggeln der LED durch ein Schlafenlegen des Kernelthreads via `wait_event_interruptible_timeout()`. Der Timeout des Schlaflens entspricht der Frequenz des Blinkens, vorausgesetzt diese ist eingeschaltet und soll überhaupt blinken (mehr zur genauen Verwendung bzw. den erlaubten Inputs im nächsten Abschnitt). Falls neuer, gültiger User-Input bereitgestellt wurde, wird die LED entsprechend an- oder ausgeschaltet und die neue Frequenz bzw. Schlaflzeit wird berechnet.

```

static int blink_thread_func(void* params)
{
    int ret = 0, value = 0;
    long int provided_freq = 0, sleep_msecs = 1000;
    bool led_on = false;

    while(!kthread_should_stop())
    {
        ret = wait_event_interruptible_timeout
            ( freq_wq,
            ({
                spin_lock(&freq_lock);
                int cond = (new_input == true);
                provided_freq = freqKrueger;
                spin_unlock(&freq_lock);
                cond;
            }),
            msecs_to_jiffies(sleep_msecs)
        );

        if (ret == 0) // Timeout reached
        {
            if(led_on == true) // If LED is activated, toggle it accordingly
            {
                value ^= 1;
                gpiod_set_value(gpio_desc, value);
            }
        }
        else if (ret == -ERESTARTSYS) // Signal came in
        {
            pr_info("signalru: interrupted by signal ...\\n");
        }
        else // Condition is true
        {
            if(provided_freq == 0) // Turn off LED
            {
                led_on = false;
                gpiod_set_value(gpio_desc, 0);
            }
            else // Turn on LED and calculate new frequency aka sleep time
            {
                led_on = true;
                sleep_msecs = 1000 / provided_freq;
            }
        }

        // Reset state (no new user input is yet provided)
        spin_lock(&freq_lock);
        new_input = false;
        spin_unlock(&freq_lock);
    }
}

return 0;
}

```

3.7.3 Verwendung

Die Verwendung des Treibers läuft über die bereitgestellte Gerätedatei `led_onoff_ru`. Wie bereits erwähnt können Strings in diese gepiped werden, der Input wird intern konvertiert und validiert.

- Der Wert **0** schaltet die LED aus
- Ein Wert zwischen **1-10** schaltet die LED mit der entsprechenden Frequenz in Hertz an
- Negative Werte und Werte größer **10** werden verworfen

Die Gerätedatei kann außerdem ausgelesen werden. In diesem Fall returned sie die aktuell gesetzte Frequenz.

3.8 MQTT-Konfiguration

Ein weiterer Teil des Projekts ist das Betreiben des Pi's als Smarthome-Gateway. Als Kommunikationsprotokoll kommt dabei MQTT zum Einsatz. Als MQTT-Broker wird mosquitto verwendet, welcher bereits in Buildroot integriert ist. Die Konfigurationsschritte sahen im groben wie folgt aus:

- mosquitto in Buildroot aktivieren
- Die default mosquitto.conf in lokales Arbeitsverzeichnis kopieren
- mosquitto.conf editieren
 - Extensive Logging anschalten
 - Anonymen Access zulassen
 - Interface konfigurieren (IP-Adresse, Port, wlan0)
- mosquitto.conf standardmäßig aufs Target kopieren
- Startskript erstellen, welches mosquitto beim Booten automatisch startet
 - **Wichtig:** Erst nach der WPA/wlan0-interface Konfiguration!
 - Loopback-Interface muss ebenfalls aktiviert sein!

Als nächstes musste ein Skript (`target/S97topics`) geschrieben werden, welches in regelmäßigen Abständen (alle 60 Sekunden) verschiedene Topics published (siehe unten für einen Code-Ausschnitt). Alle Rückmeldungen werden dabei verworfen, da Fehlermeldungen separat in eine Loggingdatei geschrieben werden und nicht gepublished werden sollen. Auch dieses Skript wird standardmäßig beim Booten geladen.

Das regelmäßige Publishen **musste** über einen while-true-Loop in Bash gelöst werden, da die internen Parameter `-repeat` und `-repeat-delay` des `mosquitto_pub` Kommandos nur mit den initialen Werten der Skriptvariablen arbeiten. Das Kommando wird sozusagen nicht nochmal "richtig" ausgeführt, sondern es wird nur das Ergebnis des ersten Ausführens erneut gepublished. Diese Art der Implementierung wird bestimmt ihren Sinn haben, scheint mir aber zu kurz gegriffen. Zumindest die Option für ein erneutes "richtiges" Ausführen des Kommandos würde ich begrüßen.

```
# S97topics (Ausschnitt)

publish_topics()
{
    while true; do
        mosquitto_pub -h ${BROKER} -d -t ${HOST_TOPIC} -m ${HOST_MSG} > /dev/null 2>&1
        mosquitto_pub -h ${BROKER} -d -t ${eth0_TOPIC} -m ${eth0_MSG} > /dev/null 2>&1
        mosquitto_pub -h ${BROKER} -d -t ${wlan0_TOPIC} -m ${wlan0_MSG} > /dev/null 2>&1
        mosquitto_pub -h ${BROKER} -d -t ${DATE_TOPIC} -m ${DATE_MSG} > /dev/null 2>&1
        mosquitto_pub -h ${BROKER} -d -t ${NAME_TOPIC} -m ${NAME_MSG} > /dev/null 2>&1
        mosquitto_pub -h ${BROKER} -d -t ${UP_TOPIC} -m $(uptime | cut -d' ' -f2) > \
            /dev/null 2>&1
        mosquitto_pub -h ${BROKER} -d -t ${LED_TOPIC} -m $(cat /dev/led_onoff_ru) > \
            /dev/null 2>&1
        sleep ${INTERVAL}
    done &
}
```

Ein Topic welches gepublished aber auch angesteuert werden sollte, war die Blinkfrequenz der LED. Im signalru-Treiber wurden, wie bereits im vorhergegangen Kapitel erwähnt, die dafür notwendigen Funktionen implementiert.

Das Topic-Publishing-Skript subscribed per `mosquitto_sub` aufs Topic “`appl/frequency_set`” und piped `stdin` in ein LED-Kontrollskript (`target/ctrl_led_ru.sh`).

```
# S97topics (Ausschnitt)

if [ "$1" == "start" ]; then
    publish_topics
    echo "Started topic publishing service"
    (mosquitto_sub -h ${BROKER} -d -t ${LED_TOPIC_CTRL} | ${LED_SCRIPT_PATH}) &
    echo "Piped topic ${LED_TOPIC_CTRL} into script"
fi
```

Dieses LED-Kontrollskript läuft in einem while-true-Loop bis es Input erhält und schreibt diesen dann in die Gerätedatei.

```
# ctrl_led_ru.sh

#!/bin/sh

echo "Started LED control script"

while true; do
    read
    echo "${REPLY}" > /dev/led_onoff_ru
done
```

4 Systemtest

4.1 Testplan

Der Testplan ist relativ simpel. Im Sinne von *Rapid Prototyping* werden die Anforderungen an eine Komponente verschriftlicht (hier Teil der Aufgabenstellung) und dann wird diese implementiert. Im Anschluss wird geprüft ob alle Anforderungen erfüllt wurden und keine ungewünschten Nebeneffekte aufgetreten sind.

4.2 Komponententest

4.2.1 Boot und Entwicklung

4.2.2 WLAN

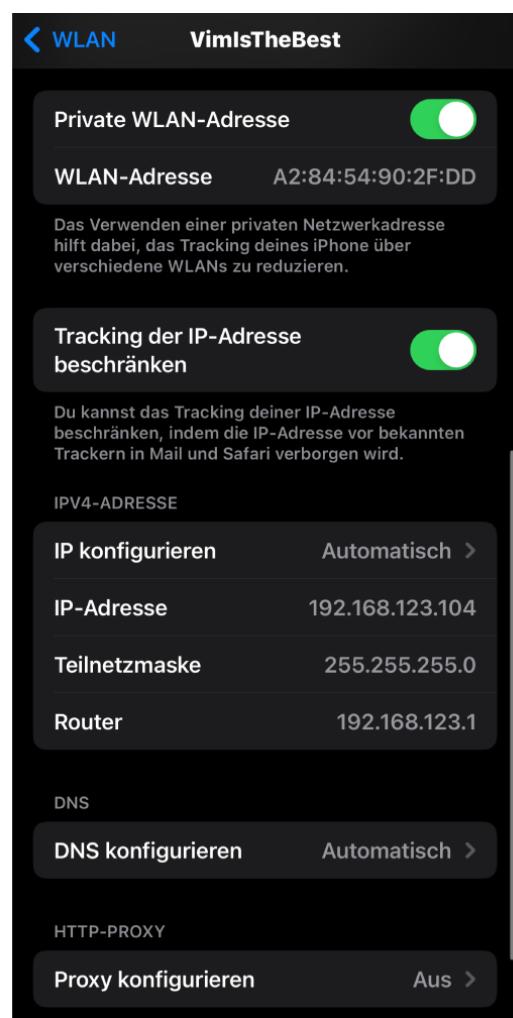


Abbildung 3: WLAN-Test

4.2.3 LED-Treiber

4.2.4 MQTT

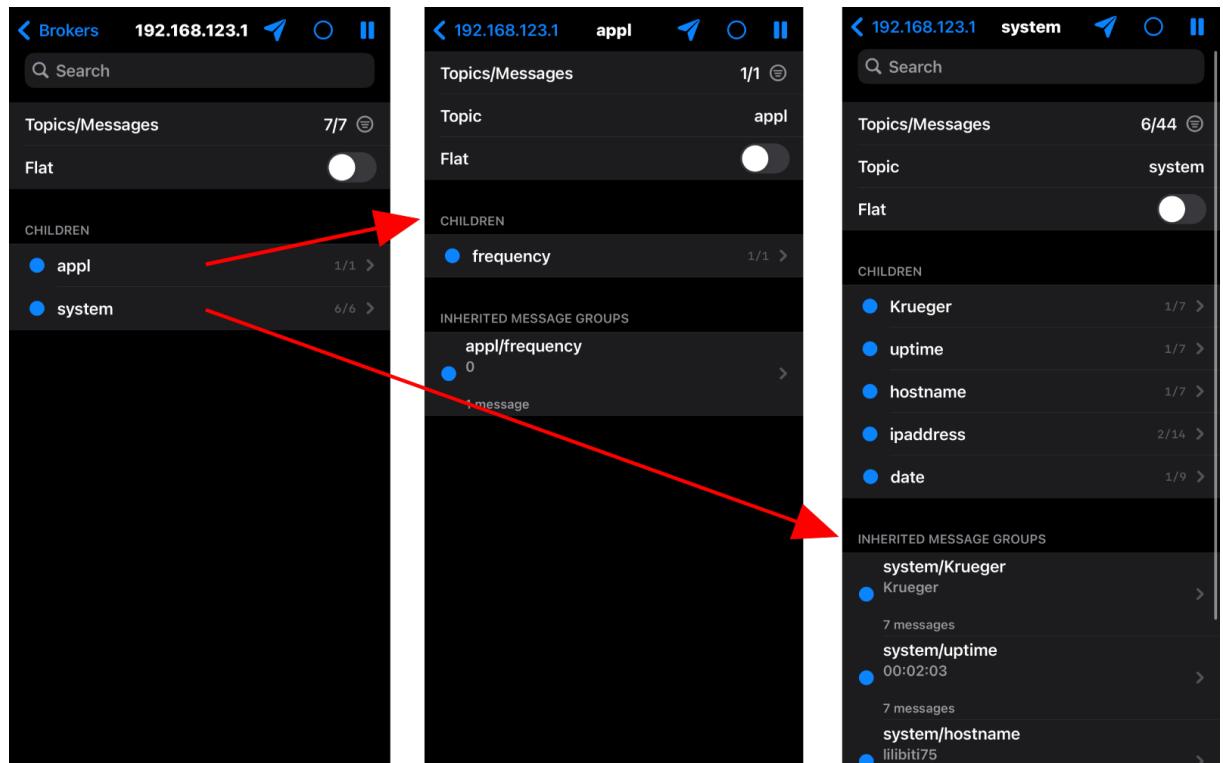
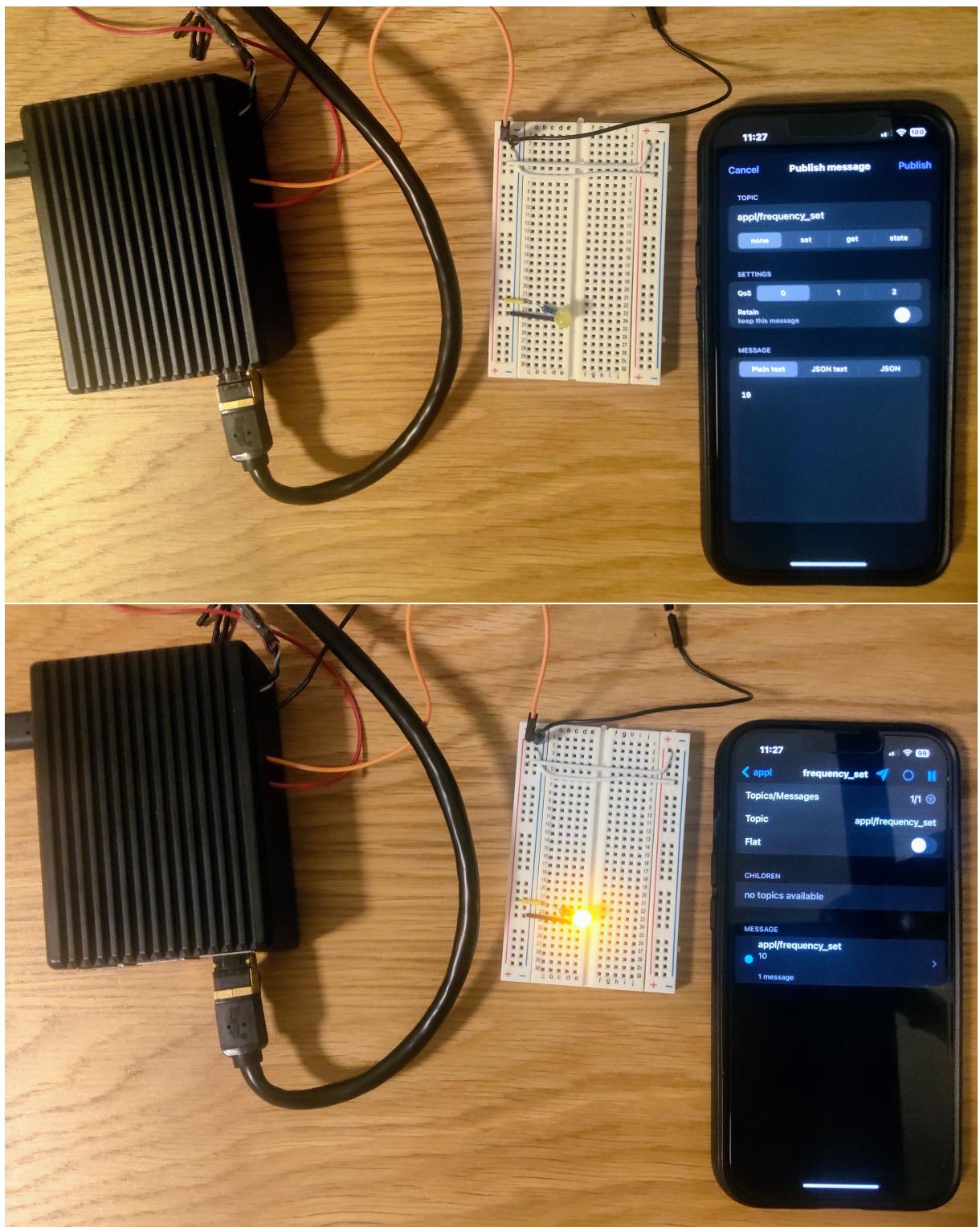


Abbildung 4: MQTT-Test

4.3 Test des Gesamtsystems



5 Zusammenfassung