Inhaltsverzeichnis

1 Projektdokumentation

1.1 Timeline

1.1.1 Netzwerkboot

- TFTP-Server installieren und starten
  - Logging aktivieren
  - Monitoren des tftp-outputs via: tail -f /var/log/syslog | grep tftp
- Netzwerkschnittstelle auf Host-PC statische IP zuweisen
- Originale Bootfiles runterladen und ins tftp Verzeichnis kopieren
- Minicom installieren, konfigurieren und starten
  - sudo minicom -D /dev/ttyUSB0
- Bootloader des Pi ist für Netzwerkboot bereits konfiguriert
- config.txt nach /srv/tftp kopieren
- cmdline.txt nach /srv/tftp kopieren
  - Achtung Datei wurde verändert

1.1.2 Buildroot (initial)

- Buildroot installieren
- Pi4-Config erstellen
- Userland konfigurieren
  - Identisch zu Aufgabenblatt 5 (inkl. dropbear für SSH)
- Kernelerstellung abwählen

1.1.2.1 Skripte

- post-build.sh erstellen
  - target/interfaces anlegen und auf Target kopieren
  - Relevant für SSH
- post-image.sh erstellen
- Skripte in Buildroot-Config hinterlegen

1.1.3 Zusätzlicher User

- user_conf-Skript hinzugefügt welches mit Ausführungsrechten nach init.d installiert wird
  - Skript muss speziellen Namen tragen, hier “S99user”

1.1.4 WLAN

- Buildroot-Konfiguration nach Aufgabenblatt 7
- Skripte und Konfigurationsdateien für WPA, DHCP, Firewall und NAT anlegen und auf Target kopieren
- Alter Kernel ist nicht WLAN-fähig
  - Neubau von Kernel 6.6 mit WLAN-Support (absichtliche Wahl einer älteren Version)
- Falls Fehlermeldung beim Booten auftaucht: brcmfmac: brcmf_set_channel: set chanspec 0x100d fail, reason -52
  - Bekannter Fehler/Bug, bisher ohne Fix: [https://github.com/raspberrypi/linux/issues/6049]

1.1.5 Gerätetreiber

- exports.sh erstellen und anpassen
- Kernel für Pi 4 bauen (mit WLAN und modules support)
  - make bcm2709_defconfig
  - Kernel-Config aus Moodle als Grundlage
  - time make -j24 zImage dtbs modules
- hello.c crosskompilieren und unter /lib/modules ablegen
- Startskript schreiben, welches Kernelmodul beim booten lädt

1.1.5.1 Entwicklung eines ersten LED-Treibers

- Entwicklung auf Host mit Deployment per scp
  - scp led.ko root@192.168.42.69:/lib/modules
  - Logging via: watch -n 1 dmesg oder tail -f /var/log/kern.log
- consumer.h api mit Kernel 6.6 und 6.9 ausprobiert, beide Male kein Erfolg
  - Ansatz über Device-Tree
    - Device Tree Blob Overlay erstellen und GPIO PIN 23 definieren
    - .dts zu .dtbo kompilieren
    - .dtbo nach /srv/tftp/overlays verschieben
    - Device Tree Blob in config.txt eintragen
    - LED-Treiber auf DTBO-API anpassen

1.1.5.2 Entwicklung eines fortgeschrittenen LED-Treibers in eigenem Thread mit Ansteuerung via Gerätedatei

- sig_driver_init erzeugt Kernel-Thread
- user input wird über Gerätedatei entgegen genommen. Implementierung innerhalb von sig_write (mehr dazu unten)
  - Werte können als normale Strings eingegeben werden (werden intern konvertiert)
  - Der Wert 0 schaltet die LED aus
  - Ein Wert zwischen 1-10 setzt die LED mit der entsprechenden Frequenz in Hertz
  - Negative Werte oder Werte größer 10 werden verworfen
  - Im Anschluss wird ein gültiger Wert der globalen Variable zugewiesen (der Zugriff ist dabei über ein Spinlock geschützt)
  - Außerdem wird noch ein State gesetzt (auch geschützt), welcher angibt, dass der User neuen Input provided hat
- Kernel-Thread realisiert Blinklicht
  - Legt sich über wait_event_interruptible_timeout() schlafen
    - Timeout des Schlafens entspricht der Frequenz des Blinkens (voraussgesetzt die LED soll aktuell überhaupt blinken / ist eingeschaltet)
  - Ein Signal / eingehender Interrupt wird dabei nur gelogged, hat keine eigene Logik
  - Falls die Condition (neuer User-Input) == true ist, wird die LED entsprechend an- oder ausgeschaltet und die bereitgestellte Frequenz/Schlafenszeit wird berechnet
- sig_write erlaubt das Schreiben einer neuen Frequenz
  - Der Zugriff auf die globale Variable ist dabei durch ein Spinlock geschützt
  - User-Input wird entgegen genommen und konvertiert
  - Wenn der Input korrekt ist, werden State und Frequency entsprechend gesetzt
  - Dann wird der Thread via wake_up_interruptible aufgeweckt
- sig_driver_exit killt den Kernel-Thread
- Am Ende wird das crosskompilierte Modul noch unter /lib/modules auf das target abgelegt und beim Boot dynamisch geladen

1.1.6 MQTT

- Mosquitto in buildroot aktivieren
- Default mosqitto.conf besorgen
  - In lokalem Entwicklungsverzeichnis bearbeiten und dann aufs target kopieren
  - Extensives Logging anschalten
  - Anonymen Access zulassen
  - listener 1883 192.168.123.1
  - Nur ipv4-Zugriff und nur über wlan0 interface

- Startskript erstellen, welches mosquitto beim Hochfahren startet
  - Wichtig: Erst nach der wpa/wlan0-interface-Konfiguration!
- mosquitto_pub funktioniert nicht
  - Lag an ausgeschaltetem Loopback-Interface
  - S95wpa geändert, um dieses standardmäßig hochzufahren
- Test-Publishing erfolgreich via: mosquitto_pub -h 192.168.123.1 -d -t 'test/topic' -m 'Hello World!' –repeat 100 –repeat-delay 60
  - Sowohl pub, als auch sub werden erfolgreich in Datei gelogged
- Skript schreiben, welches in regelmäßigen Abständen (alle 60 Sekunden) verschiedene topics published
  - –repeat mit –repeat-delay funktioniert nicht, da nur die initialen Werte für die Variablen genommen werden
  - Das Kommando wird also nicht nochmal "richtig" ausgeführt
  - Lösung über while-true-Loop mit sleep
- Anpassung des signalru-Treibers durch Ergänzung einer read-Funktion
  - Einmalige Zurückgabe des Frequenzwertes
  - Ergänzung des topics in S97topics