

Technische Hochschule Deggendorf

Fakultät Angewandte Informatik

(Bachelor-Studiengang Angewandte Informatik)

Lüftersteuerung mit Temperatursensor

Studienarbeit in dem Sommersemester 2022 für das Fach

AI-B6-Systemprogrammierung

an der Technischen Hochschule Deggendorf

vorgelegt von:

Prillwitz, Robin (00805291)

Menzel, Sven (00804260)

am: 8. Juli 2022

Prüfer:

Professor Bösnecker

Inhaltsverzeichnis

Tabellenverzeichnis	ii
Abbildungsverzeichnis	ii
Abkürzungsverzeichnis	iii
Glossar	iii
1. Motivation	1
2. Hardware	2
2.1. Temperatursensor	2
2.1.1. Hardwareanschluss	2
2.1.2. I^2C Bus	2
2.2. PWM-Erzeugung	3
2.2.1. SPI Bus	3
2.2.2. MCP41XXX Digitales Potentiometer	4
2.2.3. 555 Timer	5
2.2.4. Lüfterkontrolle mit PWM	6
2.3. PCB	7
3. Software	8
3.1. Kernel Treiber	8
3.1.1. Temperatursensor über I^2C	8
3.1.2. Potentiometer über SPI	10
3.1.2.1. SPI Implementation	10
3.1.2.2. Devicetree Overlay	11
3.1.3. FOPS	11
3.1.4. IOCTL	12
3.1.5. Kompilierung und Verlinkung	12
3.2. Applikation	12
3.2.1. Funktion	12
3.2.2. IPC über IOCTL	13
3.2.3. Installation und Ausführung	14
Literaturverzeichnis	15
Anhang	16
A. Simulation	16
B. PCB	17

Tabellenverzeichnis

1.	<i>Inter-Integrated Circuit (I²C)</i> Pinout Tabelle	2
2.	TMP102 I ² C Add0 Adresse	3
3.	<i>Serial peripheral Interface (SPI)</i> Pinout Tabelle	4
4.	MCP41XXX Command-Byte Bits	5
5.	<i>Pulse width modulation (PWM)</i> Lüfter Pinbelegungs Tabelle	7
6.	Verfügbare IOCTL Kommandos	12

Abbildungsverzeichnis

1.	Blockschaltbild des Fanctrl Projekts	1
2.	Eine I ² C Datenübertragung.	3
3.	Eine SPI Datenübertragung.	4
4.	Eine SPI Datenübertragung für ein MCP41XXX Potentiometer	4
5.	NE555 Timer in PWM Konfiguration.	6
6.	Der Lüfter von <i>Noctua</i> [4].	6
7.	TOP-Seite des entworfenen <i>Printed circuit board (PCB)</i>	7
8.	Treiber Ordnerstruktur	8
9.	Integerdivisionsinduzierter Rundungsfehler	10
10.	Applikations LUT Beispiel	13
11.	2D Ansicht des PCB aus <i>KiCad</i>	17
12.	3D Ansicht des PCB aus <i>KiCad</i>	18

Abkürzungsverzeichnis

CS	Chip Select.
API	Application Programming Interface.
CSV	Comma-separated-values.
FOPS	File Operations.
FPU	Floating point unit.
GND	Ground.
I ² C	Inter-Integrated Circuit.
IOCTL	Input/Output Control.
IPC	Interprocess Communication.
LSB	Least significant bit.
LUT	Lookup-table.
MISO	Master In, Slave Out.
MOSI	Master Out, Slave In.
MSB	Most significant bit.
N/C	Not Connected.
PCB	Printed circuit board.
PWM	Pulse width modulation.
SBC	Single-Board Computer.
SCL	Serial Clock.
SCLK	Serial Clock.
SDA	Serial Data.
SPI	Serial peripheral Interface.
Vcc	Voltage at Common Collector.

Glossar

Breakout Board	Trägerplatine um externe Anschlüsse zu erleichtern.
Devicetree	Datenstruktur die Hardware für den Kernel beschreibt.
Kernelspace	Speicherregion des Kernels.
Raspberry Pi	Ein kleiner SBC für embedded Anwendungen.
Userspace	Speicherregion des Nutzers.

1. Motivation

Das Thema dieser Studienarbeit ist die Programmierung eines Gerätetreibers für Linux. Dieses Projekt implementiert eine Lüftersteuerung mit Temperatursensor. Zum Einsatz kommen ein *Raspberry Pi 4b*, ein *Noctua NF-A4x20 5V PWM* Lüfter mit vier Pins und ein digitaler Temperatursensor *TMP102*. Der Temperatursensor misst die momentane Umgebungstemperatur. Je nach Erwärmung wird die Drehgeschwindigkeit des Lüfters beeinflusst. Hierbei versucht der Lüfter die Temperatur um den Temperatursensor auf normale Umgebungstemperatur herunter zu kühlen.

Das Projekt besteht aus der Applikation im *Userspace* und dem Treiber im *Kernelspace*. Der Treiber kommuniziert mit *I²C* und *SPI* mit der Hardwareperipherie. Die Schnittstelle zwischen *Kernelspace* und *Userspace* wird sowohl durch *File Operations (FOPS)* und *Input/Output Control (IOCTL)* ermöglicht. Abbildung 1 beschreibt den abstrakten Aufbau des Projekts. Im Folgenden Dokument wird der Aufbau von der Software, als auch der Hardware erklärt.

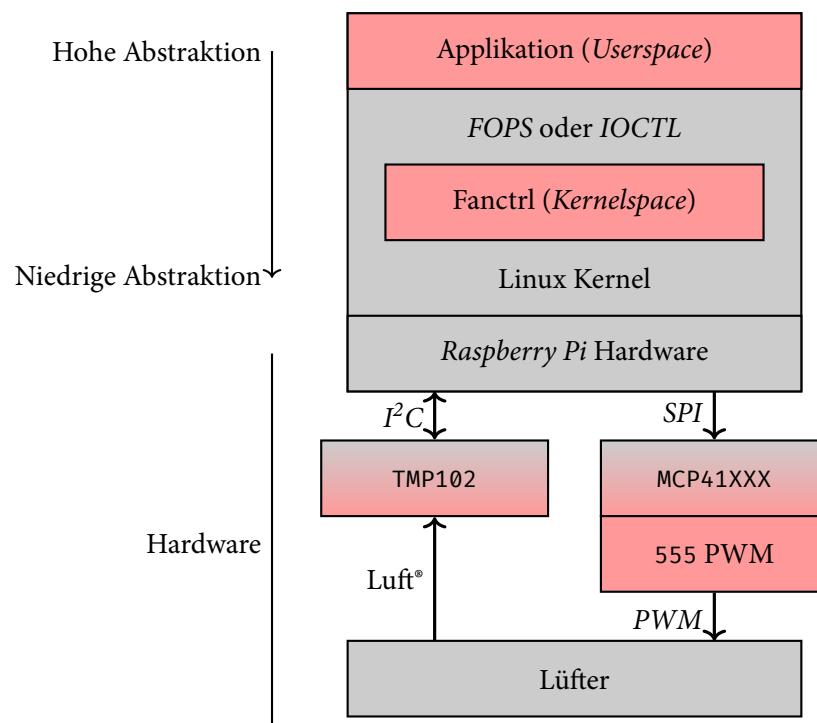


Abbildung 1: Blockschaltbild des Fanctrl Projekts. Die rot abgebildeten Lagen sind eigens implementiert.

2. Hardware

2.1. Temperatursensor

Beim Temperatursensor handelt es sich um einen TMP102 der Firma Texas Instruments auf einem *Breakout Board* von Sparkfun. Dieser hat einen Einsatzbereich von -40°C bis 125°C und wird mit I^2C angesteuert. Auf dem Sensor stehen folgende Pins zur Verfügung:

Serial Clock (SCL), *Serial Data (SDA)*, *Voltage at Common Collector (Vcc)*, *Ground (GND)*, *Alert* und schließlich *Add0*.

2.1.1. Hardwareanschluß

Der Sensor benötigt *Vcc* und *GND* für die Spannungsversorgung, sowie *SCL* und *SDA* für die Kommunikation. Die Pins werden mit dem *Raspberry Pi* wie in Tabelle 1 verbunden. Der *Alert* Pin ist unbenutzt und wird daher nicht verbunden. Der *Add0* Pin stellt das *Least significant bit (LSB)* der I^2C Adresse ein. Dieses hat einen Pull-Up auf dem *Breakout Board* und muss somit nicht extern angeschlossen werden, dieser kann *Not Connected (N/C)* bleiben.

Funktion	<i>SCL</i>	<i>SDA</i>	<i>Vcc</i>	<i>GND</i>	<i>Alert</i>	<i>Add0</i>
<i>Raspberry Pi Pin Name</i>	GPIO3	GPIO2			<i>N/C</i>	<i>N/C</i>
<i>Raspberry Pi Pin Number</i>	Pin5	Pin3	Pin2	Pin14	<i>N/C</i>	<i>N/C</i>

Tabelle 1: I^2C Pinout Tabelle

2.1.2. I^2C Bus

Der I^2C Bus ist ein Halb-Duplex Bus zur Kommunikation zwischen einem Master und beliebig vielen Slaves. I^2C Transaktionen bestehen immer aus 8 Bytes. Danach verschickt der Empfänger ein ACK oder NACK Acknowledgement Bit um zu kennzeichnen, ob die Transaktion erfolgreich war. Jede Transaktion wird von dem Busmaster gestartet. Dieser schickt zuerst eine 7 Bit Adresse. Das 8. R/W Bit, das *LSB*, gibt an, ob die folgende Operation ein Master Read oder Master Write ist. Ein generelles Muster einer I^2C Transaktion wird in Abbildung 2 angezeigt. Die 7-Bit I^2C Adresse des TMP102 verhält sich wie in Tabelle 2 angegeben.

Der TMP102 Sensor liefert Messdaten mit 13 Bit Auflösung. Dadurch I^2C jedoch nur je 8 Bit gelesen werden können, muss das Temperature Register (TR) an Position 00_H zwei mal gelesen werden. Die folgenden zwei Bytes müssen danach zu einem validen Messwert verkettet werden. Das erste Byte bildet das *Most significant bit (MSB)*. Das folgende Byte hat das *LSB* an 4. Stelle. Die Verkettung erfolgt nach Gleichung 1. Möglicherweise geht durch diese Operation das Sign-Bit verloren. Für unsere Anwendungen sind negative Temperaturen jedoch nicht relevant.

$$M_D := TR_0 \ll 4 \quad \text{OR} \quad TR_1 \gg 4 \quad (1)$$

Device Address	Add0 Pin
1001000	GND
1001001	Vcc
1001010	SDA
1001011	SCL

Tabelle 2: TMP102 I²C Add0 Adresse

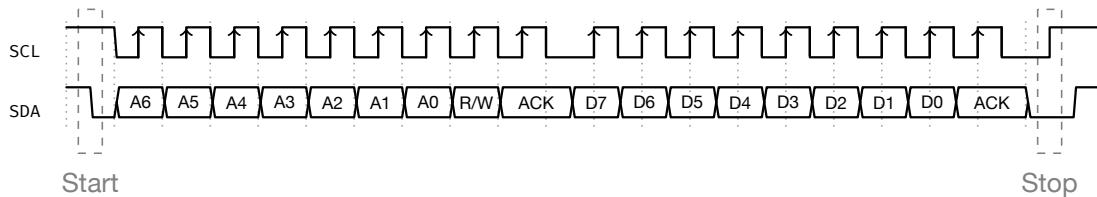


Abbildung 2: Eine I²C Datenübertragung aus 8 Adressbits und 8 Datenbits. Der Busmaster erstellt das SCL Signal und verschickt die Adresse, das Read/Write Bit und die Datenbits. Der ausgewählte Slave übernimmt die Acknowledgement Bits. Würde der Slave ein NACK übertragen, müsste der Master die vorherigen 8 Bits wiederholen.

2.2. PWM-Erzeugung

Um die Geschwindigkeit eines Lüfters zu regulieren, wird dieser mit einem *PWM* Signal gesteuert. Die Pulsweite ist proportional zur resultierenden Zielgeschwindigkeit. Der *Raspberry Pi* hat interne Hardware um *PWM* Signale zu generieren. Diese sind jedoch nicht trivial aus dem *Kernelspace* erreichbar.

Als Ersatzlösung wird externe Hardware genutzt. Ein 555 Timer erstellt ein rechteckiges Wechselsignal. Die Pulsweite wird durch ein MCP41XXX digitales Potentiometer über *SPI* eingestellt. Folglich ist die Lüftergeschwindigkeit proportional zur Pulsweite und proportional zum eingesetzten Widerstand.

2.2.1. SPI Bus

SPI ist ein synchroner, serieller Datenbus mit der Funktionsweise nach dem Master-Slave-Prinzip. Die vier wichtigsten Leitungen sind *Serial Clock (SCLK)*, *Master Out, Slave In (MOSI)*, *Master In, Slave Out (MISO)* und *Chip Select (CS)*. Die SCLK gibt einen Takt mit einer möglichen Frequenz bis mHz aus, welche zur Synchronisation vom Controller benötigt wird. Dieses Signal wird vom

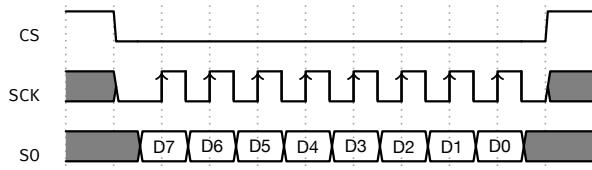


Abbildung 3: Eine SPI Datenübertragung von 8 Bits

Master ausgegeben. Mit \overline{CS} wird der jeweilige Slave ausgewählt, indem das Signal auf logisch 0 gesetzt wird. Daten werden mit der *MOSI* Leitung vom Master zum Slave und mit der *MISO* Leitung vom Slave zum Master gesendet.

In Abbildung 3 ist die Funktion von *SPI* dargestellt. Wenn der Slave ausgewählt wurde, toggelt der \overline{CS} von logisch 1 auf logisch 0. Mit jedem *SCLK* Takt wird ein Bit von Master auf Slave übertragen. Nach Beenden der Übertragung toggelt der \overline{CS} wieder auf logisch 1. Dies funktioniert mit der *MISO* Leitung vice versa.

2.2.2. MCP41XXX Digitales Potentiometer

Die Schaltung des digitalen Potentiometers wird direkt an den *Raspberry Pi* angeschlossen. Die Pins werden wie in Tabelle 3 verbunden. Der *MCP41XXX* hat zwei interne digital steuerbare Potentiometer. Um deren Widerstandswert einzustellen müssen zwei Bytes sequentiell über *SPI* gesendet werden. Das erste Byte stellt das Command-Byte dar, das zweite das Daten-Byte D_n . Die in Abbildung 4 aufgeführten Bits in der Datenübertragung setzen sich aus beiden Bytes zusammen. Die Bedeutung des Command-Byte wird in Tabelle 4 erklärt. Die resultierenden Widerstände können mit Gleichung 2 und Gleichung 3 berechnet werden.

Funktion	<i>MOSI</i>	<i>MISO</i>	<i>Vcc</i>	<i>SCLK</i>	\overline{CS}
<i>Raspberry Pi Pin Name</i>	GPIO10	GPIO9	5V Power	GPIO11	GPIO8
<i>Raspberry Pi Pin Nummer</i>	Pin19	Pin21	Pin2	Pin23	Pin24

Tabelle 3: SPI Pinout Tabelle

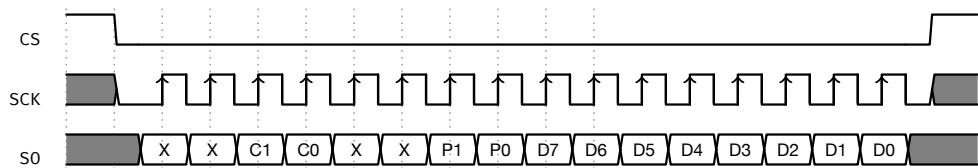


Abbildung 4: Eine SPI Datenübertragung für ein MCP41XXX Potentiometer

C1	C0	Kommando
0	0	Kein Kommando
0	1	Daten Schreiben
1	0	Ausschalten
1	1	Kein Kommando

a: Kommando Bits

P1	P0	Kommando
0	0	Kein Potentiometer
0	1	Nur Potentiometer 0
1	0	Nur Potentiometer 1
1	1	Beide Potentiometer

b: Potentiometer bits

Tabelle 4: MCP41XXX Command-Byte Bits

$$R_{WA}(D_n) = \frac{(R_{AB})(256 - D_n)}{256} + R_W \quad (2)$$

$$R_{WB}(D_n) = \frac{(R_{AB})(D_n)}{256} + R_W \quad (3)$$

wobei

R_{AB} : Widerstand zwischen Anschluss A und B, hier: $10\text{k}\Omega$

R_W : Wiper Widerstand, nominal: 52Ω

D_n : Dezimalwert des Wiperregisters: $0_{10} \leq D_n \leq 255_{10}$

2.2.3. 555 Timer

Der Timer ist in einer astabilen Konfiguration verbunden. Der Timer erwartet lediglich eine positive Spannung über $U+$ zu $U-$ von 3.3V. Ein rechteckiges Ausgangssignal mit einstellbarer Pulsweite wird an Pin, OUT, gegeben. Eine analytische Bestimmung von $U_C(t)$ oder $I(t)$ kann aufgrund der nicht linearen Übertragungsfunktion der Diode nicht bestimmt werden [5]. Folglich ist eine Berechnung von f nicht möglich¹. Die Frequenz kann lediglich empirisch durch Messung oder Simulation ermittelt werden. Der LTSpice Quellcode für die Partielle Bestimmung der Spannung an dem Kondensator wird in Anhang A gegeben. Der Duty-Cycle des PWM Signals kann nicht analytisch bestimmt werden, jedoch ist dieser proportional zu den Widerständen ($R_{WA} \propto D$, $R - R_{WB} \propto D$). Diese Annahme wird auch in dem Treiber genutzt.

¹Eine grobe Annäherung durch die Annahme, dass U_{fd} der Dioden konstant und unabhängig von I ist, ist möglich, jedoch sind die Resultate daraus sehr ungenau und nahezu unbrauchbar.

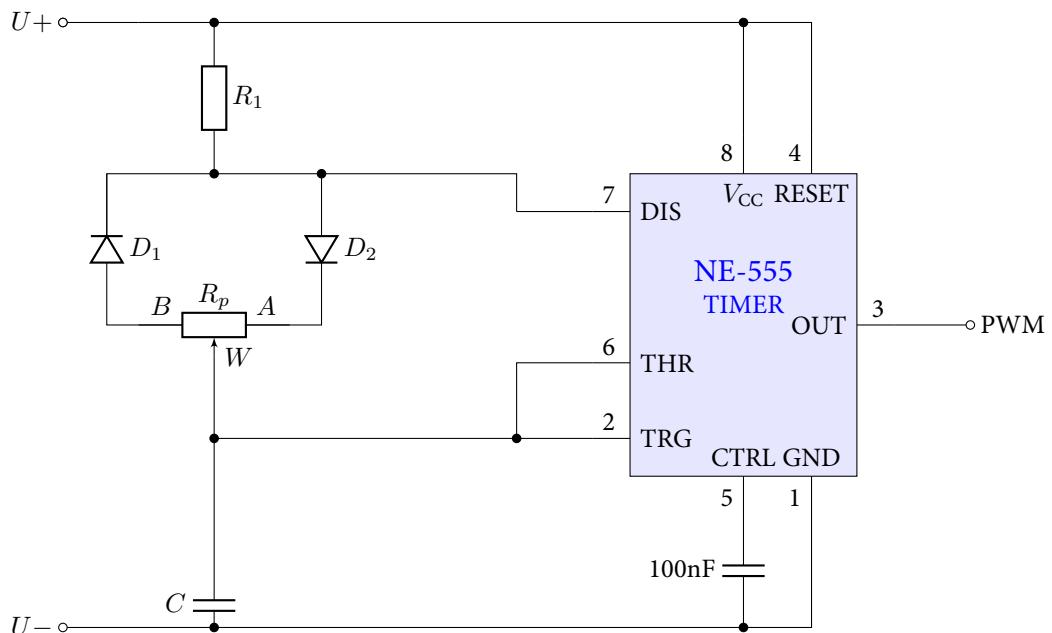


Abbildung 5: NE555 Timer in PWM Konfiguration.

2.2.4. Lüfterkontrolle mit PWM

Zur Kühlung wird ein *NF-A4x20* 4-Poliger PWM Lüfter von *Noctua* aus Abbildung 6 verwendet. Dieser benötigt eine Gleichspannung von 5V und wird mit einem PWM Signal mit einer Amplitude von 5V angesteuert. Das Tacho-Signal wird nicht verwendet. Die Pins werden wie in Tabelle 5 verbunden.



Abbildung 6: Der Lüfter von *Noctua* [4].

Lüfter Pin	Farbe	Funktion	Anschluss
1	■ Schwarz	GND	GND
2	■ Gelb	Vcc	5V
3	■ Grün	Tach	N/C
4	■ Blau	PWM	PWM von 555

Tabelle 5: PWM Lüfter Pinbelegungs Tabelle

2.3. PCB

Zu dieser Schaltung wurde auch ein *PCB* entworfen. Das *PCB* ist sowohl in Abbildung 7, als auch genauer unter Anhang B in Abbildung 11 und Abbildung 12 dargestellt. Der Schaltplan ist in Abbildung B inkludiert. Auf dem Board ist ein 555 Timer zur *PWM* Erzeugung und zwei LM75B Sensoren verbaut. Die beiden Sensoren sind dem TMP102 sehr ähnlich, sind jedoch vorrätig. Deren Adressen werden jedoch durch die Addresspins verändert. Der OS Pin der beiden Sensoren wird auch angeschlossen. Da dieser in Open-Drain Konfiguration benutzt werden kann, belegen beide Signale nur einen Pin. Momentan wird der Übertemperaturalarm nicht in der Software benutzt. Die Änderungen ab Treiber für die LM75B Sensoren ist trivial. Momentan besteht keine Softwareunterstützung für zwei Sensoren, jedoch ist diese Modifikation trivial. Das *PCB* ist *Raspberry Pi HAT* konform und respektiert mechanische und elektronische Anforderungen².

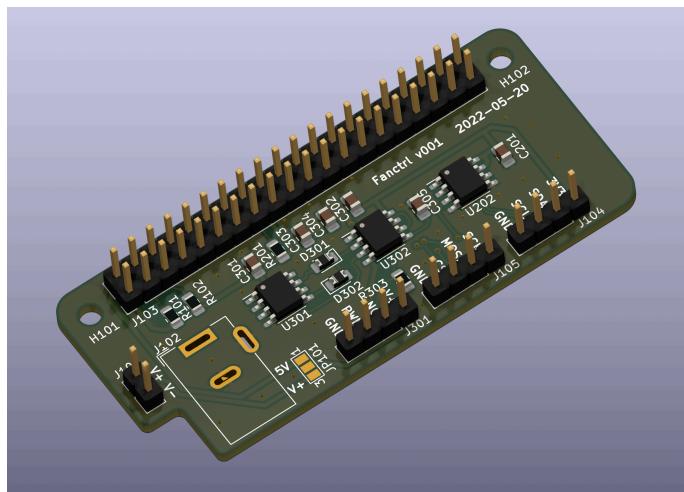


Abbildung 7: TOP-Seite des entworfenen *PCB*

²Diese Anforderungen spezifizieren beispielsweise den mechanischen Umriss, die nicht-leitenden Befestigungslöcher und die Belegung der 40-Pin Steckverbindung.

3. Software

3.1. Kernel Treiber

Der Code des Kernaltreibers ist in C99 geschrieben. Dieser ist Prozessoragnostisch. Es werden jedoch existierende low-level Treiber zwischen den Prozessorregistern und *Kernelspace* erwartet, die die Kommunikation über *I²C* und *SPI* abstrahieren. Das Modul an sich ist lediglich von Linux-Headern abhängig. Der Treiber ist auf verschiedene C-Dateien aufgeteilt. Die Struktur des Treibers ist in Abbildung 8 aufgeführt. Der Tatsächliche Quellcode für den Treiber befindet sich lediglich im *src* Ordner. Alle anderen Dateien sind notwendig jedoch kein direkter Teil des Treibers.

Die Hauptdatei mit dem Anfangspunkt des Treibers ist in *fanctrl.c*. Darin wird das Treibermodul initialisiert, registriert, ausgeführt und schließlich auch deinitialisiert. Der Kernaltreiber kann über, sowohl durch *FOPS*, als auch mit *IOCTL* mit dem *Userspace* kommunizieren. Dies wird in den *fops.c* und *ioctl.c* und deren jeweils dazugehörigen Header-Files implementiert. Die anderen zwei Hauptfunktionen, die Kommunikation über *I²C* und *SPI* werden jeweils in *temp.c* und *pwm.c* implementiert. In den folgenden Abschnitten werden die implementierten Funktionen, zusätzliche Dateien und der Kompilierungs/Installationsvorgang beschrieben.

3.1.1. Temperatursensor über *I²C*

Der Temperatursensor ist in *./src/temp.c* und *./src/temp.h* implementiert. Mit der Funktion *tempInit* wird der *I²C* Adapter initialisiert. Danach wird ein neuer *I²C Device struct* hinzugefügt und der Treiber wird dem Adapter registriert. Das Device enthält die Informationen über Device Name und Adresse. Ist der Adapter initialisiert wird dieser mit *tempDeinit* zum Programmende, falls er erfolgreich initialisiert wurde, wieder deinitialisiert. Dabei werden alle vom *I²C* Adapter genutzten Ressourcen für den Treiber frei gegeben. Auf dem *Raspberry Pi* muss *I²C* durch die *raspi-config* aktiviert werden.

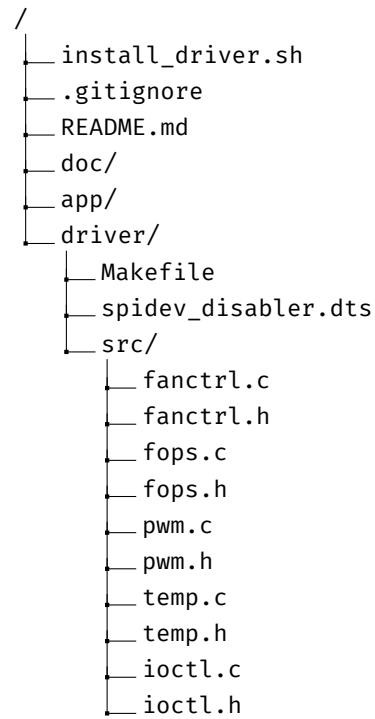


Abbildung 8: Treiber Ordnerstruktur

Der TMP102 Sensor gibt Messdaten mit 13-Bit Präzision bei einer Auflösung von 0.0625°C am *LSB* an. Um einen Messwert zu einer Temperatur zu konvertieren müssen zuerst zwei Byte gelesen werden. Diese müssen zu einem 16-Bit Wert, welcher den ursprünglichen 13-Bit Messwert beinhaltet, konkateniert werden. Anschließend muss der Messwert linear abgebildet werden, dass heißt mit der Auflösung pro *LSB* multiplizieren. Die *Floating point unit (FPU)* sollte unter allen Umständen nicht aus dem *Kernelspace* verwendet werden, da die Kontextänderung abträgliche Performanceimplikationen auf *Userspace* Applikationen mit sich führen kann. Die Berechnung wird folglich mit der Integerdivision des Kehrwerts wie in Gleichung 4 ausgeführt. Der Fehler der Berechnung wird in Gleichung 5 aufgeführt und in Abbildung 9 visualisiert. Der resultierende Rundungsfehler manifestiert sich als Rauschen. Da unsere Anwendung sich nominal bei und über Raumtemperatur in Einer-Stellen Präzision agiert sind die resultierenden Fehler vernachlässigbar. Die Implementation der Kommunikation mit dem Sensor und Wandlung der Daten ist in Sourcecode 1 gegeben.

$$\vartheta = M_D \cdot 0.0625^{\circ}\text{C} = \frac{M_D}{(0.0625^{\circ}\text{C})^{-1}} = \frac{M_D}{16^{\circ}\text{C}^{-1}} \approx \left\lfloor \frac{M_D}{16^{\circ}\text{C}^{-1}} \right\rfloor \quad (4)$$

$$\begin{aligned} \Delta_{M_D} &= \frac{M_D}{16^{\circ}\text{C}} - \left\lfloor \frac{M_D}{16^{\circ}\text{C}^{-1}} \right\rfloor \\ \delta_{M_D} &= \frac{\Delta_{M_D}}{\frac{M_D}{16^{\circ}\text{C}}} \end{aligned} \quad (5)$$

```

44 u16 readTemp(void) {
45     int temp;
46     u16 b1, b2;
47
48     i2c_smbus_write_byte(test_i2c_client, 0x00);
49     b1 = (u16)i2c_smbus_read_byte(test_i2c_client);
50     b2 = (u16)i2c_smbus_read_byte(test_i2c_client);
51
52     temp = (b1 << 4) | (b2 >> 4);
53     temp = temp / 16;
54
55     return temp;
56 }
```

Sourcecode 1: ./driver/src/temp.c – $I^2\text{C}$ Daten auslesen

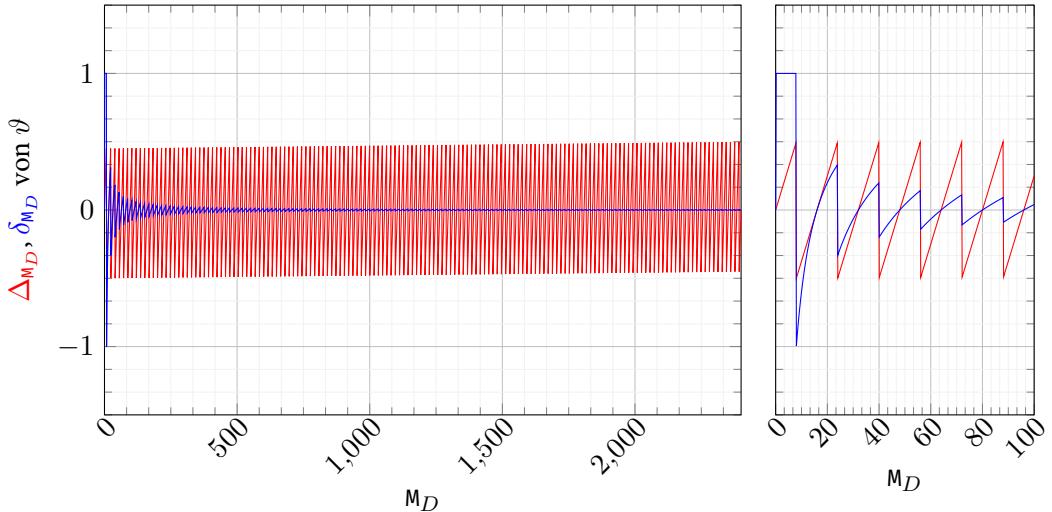


Abbildung 9: Integerdivisionsinduzierter Rundungsfehler der Temperatursensorkonversion aus Gleichung 5. Rot: Absoluter Fehler Δ_{MD} . Blau: Relativer Fehler δ_{MD} .

3.1.2. Potentiometer über SPI

3.1.2.1. SPI Implementation

Die Initialisierung des SPI Treibers läuft sehr ähnlich zu der des I²C Treibers ab. Die genaue Implementation ist in `./src/pwm.c` und `./src/pwm.h` zu finden. Es wird zuerst ein `spi_board_info` struct angelegt, welches mit den relevanten Daten des *Single-Board Computer* (SBC) gefüllt wird. Dazu zählen die Nummer des Kernel SPI Treibers, dessen CS Signal Nummer und dessen Geschwindigkeit. Diese ist hier als 4MHz festgelegt. Danach wird mit der Funktion `spi_busnum_to_master` ein geeigneter Bus Master gesucht. Folgend wird mit `spi_new_device` und schließlich `spi_setup` ein SPI Device angelegt und registriert. Zu jedem Schritt können Fehler auftreten, welche abgefangen werden. Die Deinitialisierung verläuft wieder ähnlich. Ist das SPI Device erfolgreich geladen, so wird dieser einfach mit `spi_unregister_device` abgemeldet.

Um den Widerstand des Potentiometers zu setzen, wird die in Sourcecode 2 gezeigte Funktion genutzt. Darin wird eine SPI Transaktion aus zwei Bytes erstellt. Der MCP Chip erwartet zuerst das *Command Byte* in Form von Tabelle 4. Im Code ist dieses als `00010011` festgelegt. Dadurch werden beide internen Potentiometer gesetzt. Da der Duty-Cycle proportional und relativ zu dem Widerstand ist, kann hier der Wert für das Widerstandsregister auch als Prozentbereich wie in Gleichung 6 interpretiert werden.

$$[0_H; FF_H] \rightarrow [0\%; 100\%] \quad (6)$$

```

67 void pwmSetDuty(int cycle)
68 {
69     u8 tx[2];
70
71     if (spi_device)
72     {
73         tx[0] = 0b00010011; // command byte
74         tx[1] = cycle;
75         spi_write(spi_device, &tx, 2);
76     }
77 }
```

Sourcecode 2: ./driver/src/pwm.c SPI – Potentiometer setzen

3.1.2.2. Devicetree Overlay

Auf dem *Raspberry Pi* muss der *SPI* Treiber durch *raspi-config* aktiviert werden. Dadurch werden unter */dev* zwei *SPI* Treiber angelegt und das Hardware *SPI* Subsystem aktiviert. Die angelegten Treiber stellen eine *Application Programming Interface (API)*-Verbindung zwischen *Kernelspace* und der Hardware dar. Jedoch werden durch die Treiber die \overline{CS} Signale belegt. Der Treiber zur Kommunikation über *SPI* funktioniert, kann jedoch aufgrund dessen nicht durch externe Programme im *Kernelspace* beansprucht werden. Um dies zu umgehen muss der Zugriff auf die \overline{CS} Signale durch die durch das System bereitgestellten Treiber unterbunden werden. Dazu wird der *Devicetree* zur Laufzeit mit einem Patch injiziert. Deswegen stehen die \overline{CS} Signale frei zur Verfügung, um durch unseren Treiber aus dem *Kernelspace* genutzt zu werden. Das *Devicetree* Overlay wird mit dem *Devicetree* compiler kompiliert:

```
dtc spidev_disabler.dts -O dtb > spidev_disabler.dtbo
```

Der Kernel kann nun mit dem Overlay gepatched werden:

```
sudo dtoverlay -d . spidev_disabler
```

3.1.3. FOPS

Die *FOPS* Operationen werden in *./src/fops.c* und *./src/fops.h* implementiert. Die Funktionen werden in *./src/fanctrl.c* im *file_operations* struct registriert. Der *FOPS* Treiber legt eine Pseudodatei unter */etc/fanctrl* an. Durch lesen der Datei wird der Sensor kontinuierlich ausgelesen und der Temperaturwert wird in Dezimalformat mit folgendem „°C“ Suffix

ausgegeben. Durch schreiben in die Datei wird der Duty-Cycle gesetzt. Dieser muss im Intervall zwischen $[0_H; FF_H]$ liegen. Der Code um einen beliebigen Dezimalwert in char* Repräsentation zu einem Byte zu konvertieren stammt von [3] mit eigenen Modifikationen.

3.1.4. IOCTL

Die *IOCTL* Treiber werden in `./src/ioctl.c` und `./src/ioctl.h` implementiert und auch in `./src/fanctrl.c` im `file_operations` struct registriert. Der Header definiert zwei *IOCTL* Kommandos. Diese sind in Tabelle 6 aufgeführt. Die Implementation findet in der dazugehörigen C-Datei in `dev_ioctl` statt. Durch `CMD_READ_TEMP` wird die Temperatur gelesen und zurückgegeben. Mit `CMD_SET_SPEED` wird der Duty-Cycle zu dem Übergabeparameter gesetzt.

<i>IOCTL</i> Name	Richtung	Typ	Nummer	Größe
<code>CMD_READ_TEMP</code>	Read	'A'	01_H	u32
<code>CMD_SET_SPEED</code>	Write	'A'	02_H	u32

Tabelle 6: Verfügbare *IOCTL* Kommandos

3.1.5. Kompilierung und Verlinkung

Der Treiber kann automatisch mit dem `Makefile` kompiliert und als Kernel Objekt verlinkt werden. Danach kann das erstelle Kernel Objekt auch in den Kernel geladen werden. Um den Entwicklungslebenszyklus zu vereinfachen besteht auch die Möglichkeit den gesamten Zyklus automatisch auszuführen. Mit dem `./install_driver.sh` Skript wird das Kernelmodul entladen, neu kompiliert, das `Devicetree` Overlay neu geladen und das Kernel Modul wieder neu geladen.

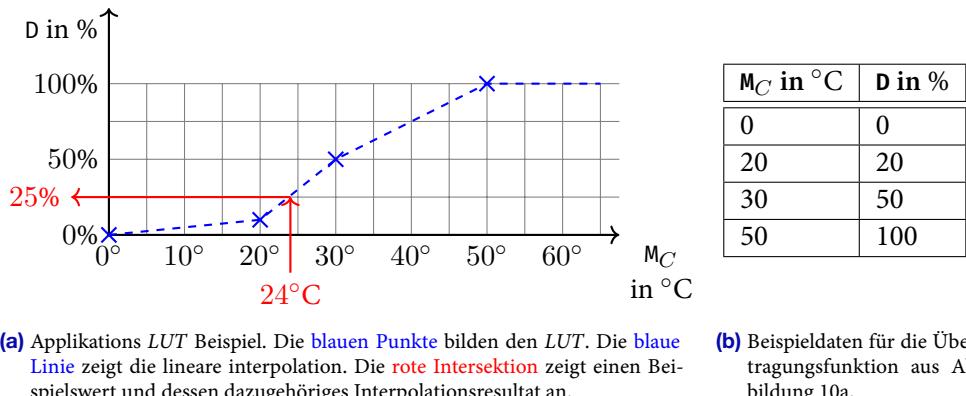
3.2. Applikation

3.2.1. Funktion

Die Applikation ist ein eigenständiges Programm. Dieses kommuniziert mit dem Treiber über *IOCTL*. Das Programm ist in *Python 3* implementiert. Das Programm liest periodisch einen Temperatur-Messwert $M_C[n]$ in °C aus und bildet diesen zu einem PWM-Duty-Cycle $D[n]$ nach dem Prinzip von Gleichung 7 um. Da der Treiber die Abtastrate weder vorgibt, noch künstlich limitiert, kann die Samplingfrequenz durch die Applikation frei gewählt werden. Der Parameter n steht hier für die Nummer eines Samples.

$$g(M_C[n]) \rightarrow D[n] \quad (7)$$

Grundsätzlich kann die Übertragungsfunktion g beliebig implementiert werden. Diese kann sogar von historischen Messdaten $M_C[n - m]$ abhängig sein. Dabei ist der Parameter m ein beliebiger Offset in die Vergangenheit. Damit bleibt sowohl die Einfachheit der Nutzung, als auch die Flexibilität komplexe Signalverarbeitung zu betreiben, bestehen. Unsere Implementation realisiert zur Demonstration einen einfachen, kontextfreien, *Lookup-table (LUT)*. Dieser beinhaltet einzelne Eingabe/Ausgabe Wert-Datenpaare aus einer *Comma-separated-values (CSV)* Datei und bestimmt den momentanen Ausgang anhand von linearer Interpolation am Eingangswert. Die resultierende Übertragungsfunktion ist eine stückweise lineare Funktion. Ein Beispiel dafür ist in Abbildung 10a mit den Datenpunkten aus Tabelle 10b dargestellt.



(a) Applikations LUT Beispiel. Die **blauen Punkte** bilden den LUT. Die **blaue Linie** zeigt die lineare Interpolation. Die **rote Intersektion** zeigt einen Beispieldwert und dessen dazugehöriges Interpolationsresultat an.

(b) Beispieldaten für die Übertragungsfunktion aus Abbildung 10a.

Abbildung 10: Applikations LUT Beispiel

3.2.2. IPC über IOCTL

Python unterstützt IOCTL durch die `fcntl` Standardbibliothek. Jedoch existieren die C-Makros zur Definition der IOCTL Konstanten nicht. Dazu wird die externe `ioctl_opt[9]` Bibliothek benutzt. Diese in Kombination der Standard `ctypes` Bibliothek ermöglicht eine ähnliche Nutzung zu den Nativen C-Makros. Die `ioctl_opt` Bibliothek stellt ähnliche Makros zur Verfügung. Die interne Implementation besteht aus logischen Operationen und Shifts. Die resultierenden IOCTL Konstanten können danach auf eine geöffnete Instanz der Pseudodatei in `/etc/fanctrl` durch `fcntl.ioctl` mit einem geeigneten Buffer angewendet werden. Um den Buffer zu schreiben oder zu lesen wird die native `Bytes` Klasse verwendet. Die Bytes benutzen *little Endianness*. Eine relevante Funktion ist in Sourcecode 3 aufgeführt.

```

27 # define CMD_READ_TEMP _IOR('A', 0x01, int32_t*)
28 CMD_READ_TEMP = IOR(ord('A'), 0x01, ctypes.c_uint32)
29
30 def read_temp():
31     buf = bytes(4)
32     fd = open(driver, "wb")
33     rt = ioctl(fd, CMD_READ_TEMP, buf, False)
34     temp = int.from_bytes(rt, "little")
35     fd.close()
36     return temp

```

Sourcecode 3: ./app/app.py IOCTL zum Auslesen der Temperatur

3.2.3. Installation und Ausführung

Zur Ausführung wird eine globale Instanz von *Python* Version 3.x.x benötigt. Es werden externe Abhängigkeiten benötigt. Es wird empfohlen diese in ein lokales *virtual environment* zu installieren um den globalen Scope nicht zu vermüllen. Der empfohlene Prozess zur Ausführung ist in Sourcecode 4 angegeben.

```

cd app
python3 -m venv venv
source ./venv/bin/activate
pip install -r requirements.txt
chmod +x ./app.py
./app.py

```

Sourcecode 4: Installationsvorgang der Applikation

Literaturverzeichnis

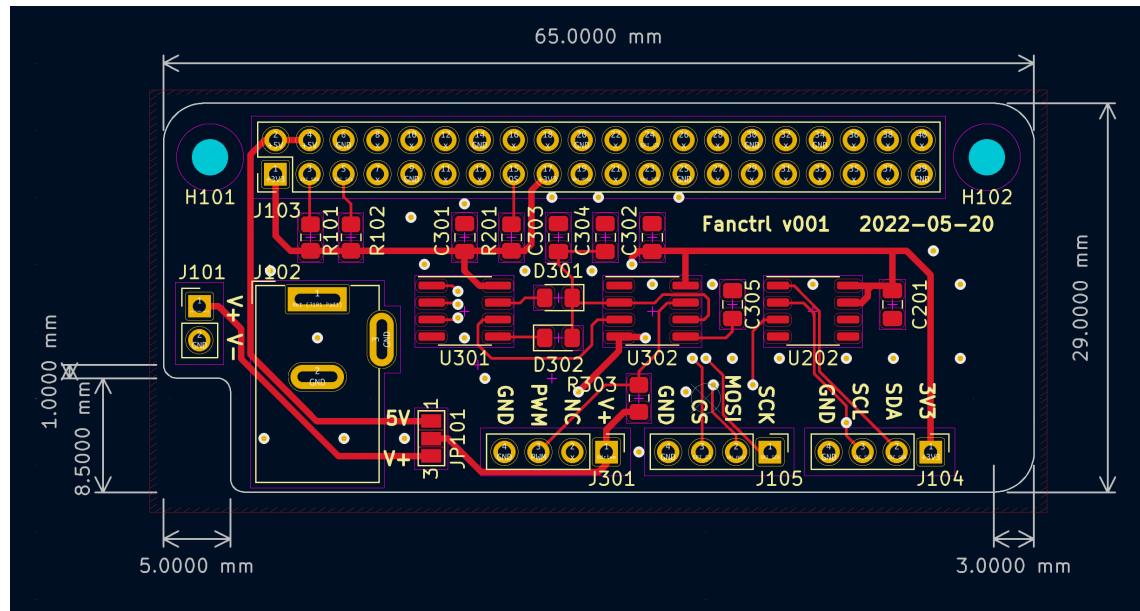
- [1] R.-H. bboyho. *Digital Temperature Sensor Breakout - TMP102*. Hrsg. von SparkFun. 2016. URL: https://github.com/sparkfun/Digital_Temperature_Sensor_Breakout--TMP102/tree/V_H13.0 (besucht am 30.05.2022).
- [2] J. Corbet. *Linux Device Drivers*. O'Reilly und Associates, 2005. ISBN: 978-0596005900.
- [3] Jacob. *How to convert a string to integer in C?* 2016-10-18. URL: <https://stackoverflow.com/a/40116476> (besucht am 16.06.2022) (siehe Seite 12).
- [4] Noctua, Hrsg. *NF-A4x20 PWM*. 2022. URL: <https://noctua.at/en/nf-a4x20-pwm> (besucht am 30.05.2022) (siehe Seite 6).
- [5] playduck. *Current through a Resistor, Diode and Capacitor in Series*. 2022. URL: <https://electronics.stackexchange.com/questions/621335/current-through-resistor-diode-and-capacitor-in-series> (besucht am 28.05.2022) (siehe Seite 5).
- [6] *spi-bcm2835_3f204000.spi: chipselect 0 already in use*. 2016. URL: <https://forums.raspberrypi.com/viewtopic.php?t=151423> (besucht am 27.05.2022).
- [7] „xx555 Precision Timers“. In: (1973). Hrsg. von Texas Instruments. URL: <https://www.ti.com/lit/ds/symlink/ne555.pdf> (besucht am 30.05.2022).
- [8] Texas Instruments, Hrsg. *Low Power Digital Temperature Sensor with SMBus*. 2008. URL: <https://www.sparkfun.com/datasheets/Sensors/Temperature/tmp102.pdf> (besucht am 30.05.2022).
- [9] vpelletier. *ioctl-opt* 1.2.2. 2018-03-21. URL: <https://pypi.org/project/ioctl-opt/> (besucht am 14.06.2022) (siehe Seite 13).

Anhang

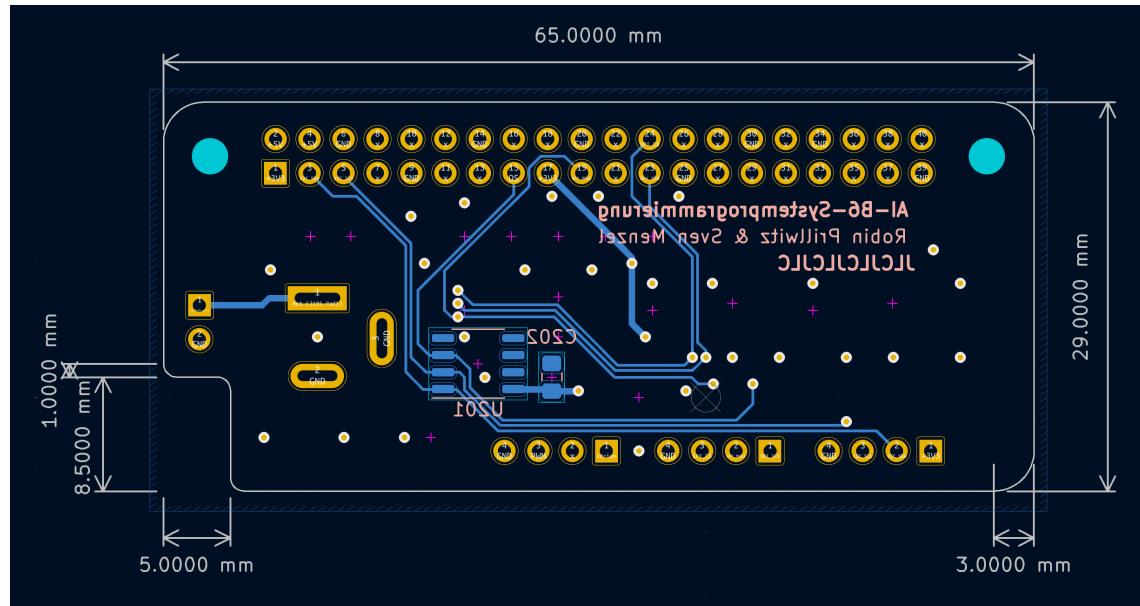
A. Simulation

```
1 Version 4
2 SHEET 1 880 680
3 WIRE 176 32 32 32
4 WIRE 176 48 176 32
5 WIRE 32 144 32 32
6 WIRE 176 160 176 128
7 WIRE 176 256 176 224
8 WIRE 32 336 32 224
9 WIRE 176 336 176 320
10 WIRE 176 336 32 336
11 WIRE 176 368 176 336
12 FLAG 176 368 0
13 SYMBOL res 160 32 R0
14 SYMATTR InstName R1
15 SYMATTR Value 100
16 SYMATTR SpiceLine tol=1 pwr=0.1
17 SYMBOL cap 160 256 R0
18 SYMATTR InstName C1
19 SYMATTR Value 1μ
20 SYMATTR SpiceLine V=2.5 Irms=0 Rser=0 Lser=0
21 SYMBOL voltage 32 128 R0
22 WINDOW 3 -37 53 Right 2
23 WINDOW 123 0 0 Left 0
24 WINDOW 39 0 0 Left 0
25 SYMATTR InstName V1
26 SYMATTR Value 5
27 SYMBOL diode 160 160 R0
28 SYMATTR InstName D1
29 SYMATTR Value 1N4148
30 TEXT -66 366 Left 2 !.tran 1m startup
```

B. PCB

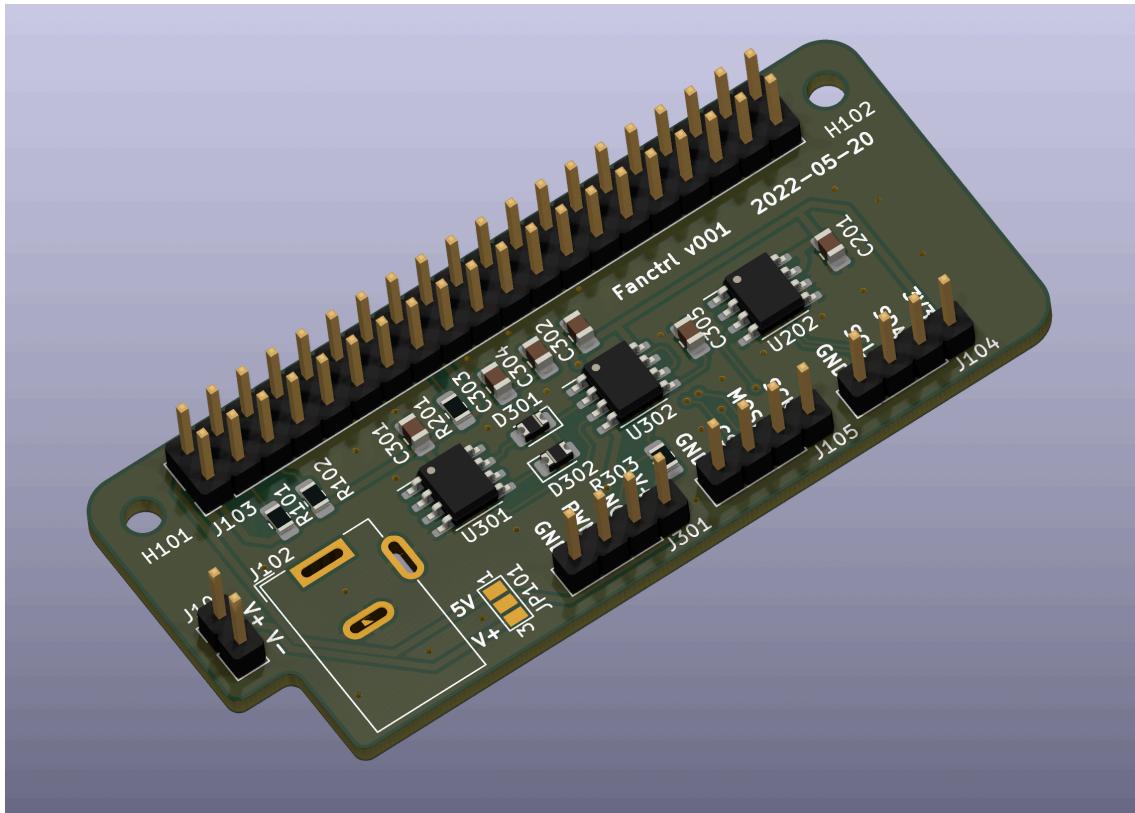


(a) TOP-Seite des PCB

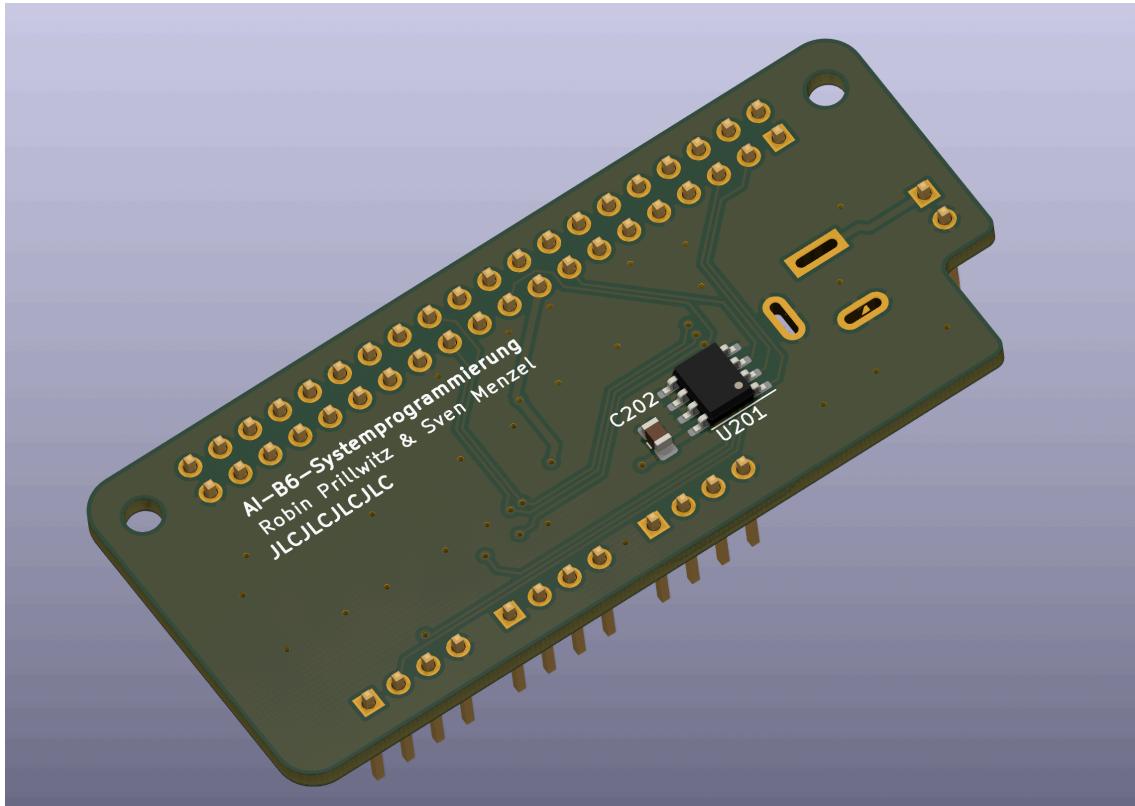


(b) BOTTOM-Seite des PCB

Abbildung 11: 2D Ansicht des PCB aus KiCad.



(a) TOP-Seite des *PCB*



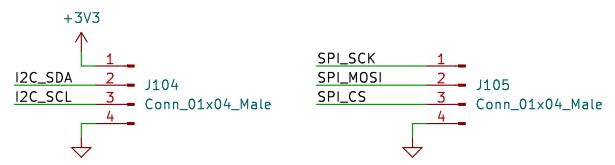
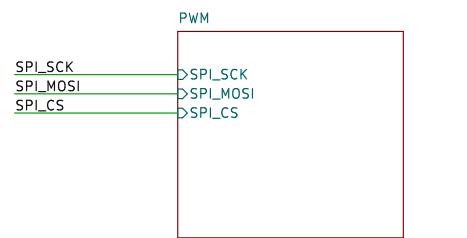
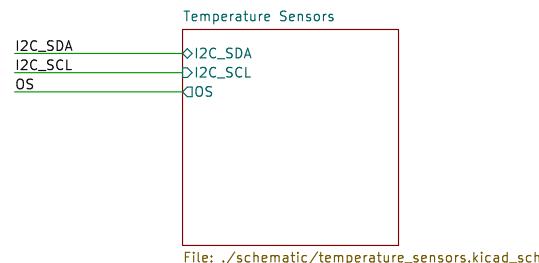
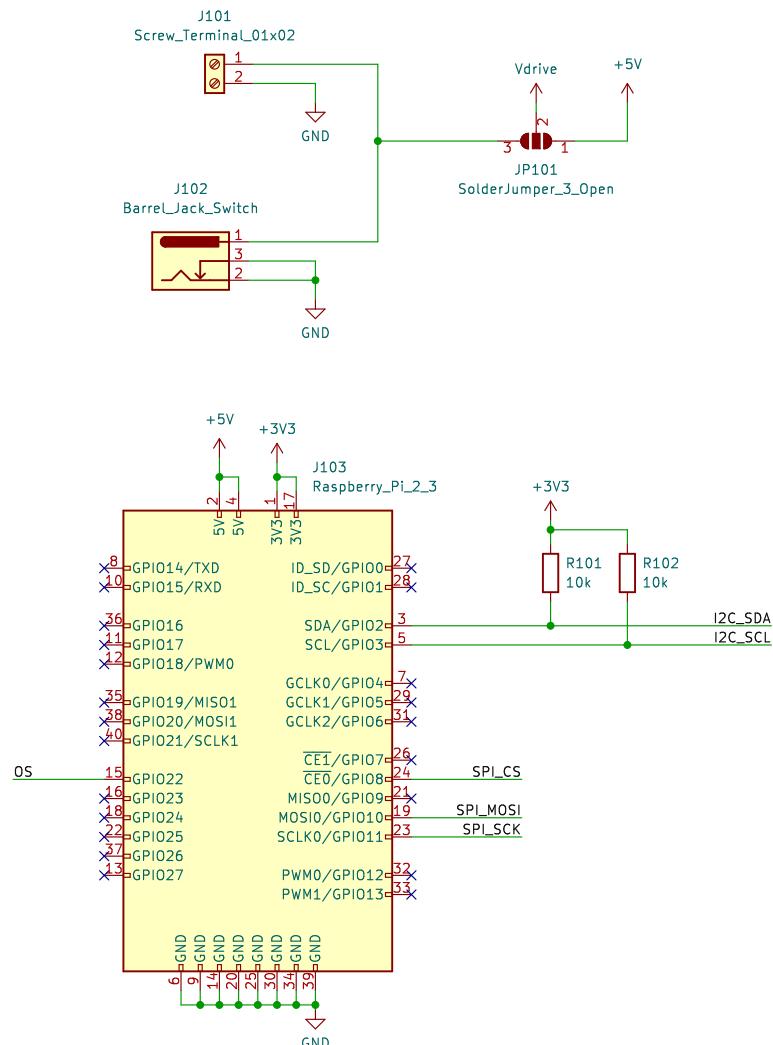
(b) BOTTOM-Seite des *PCB*

18

Abbildung 12: 3D Ansicht des *PCB* aus *KiCad*.

AI-B6-Systemprogrammierung

RPI Fanctrl



Sheet: / File: ai-b6-fanctrl.kicad_sch

Title:

Size: A4 Date:
KiCad E.D.A. kicad (6.0.0-0)

Rev:
Id: 1/3

A

A

B

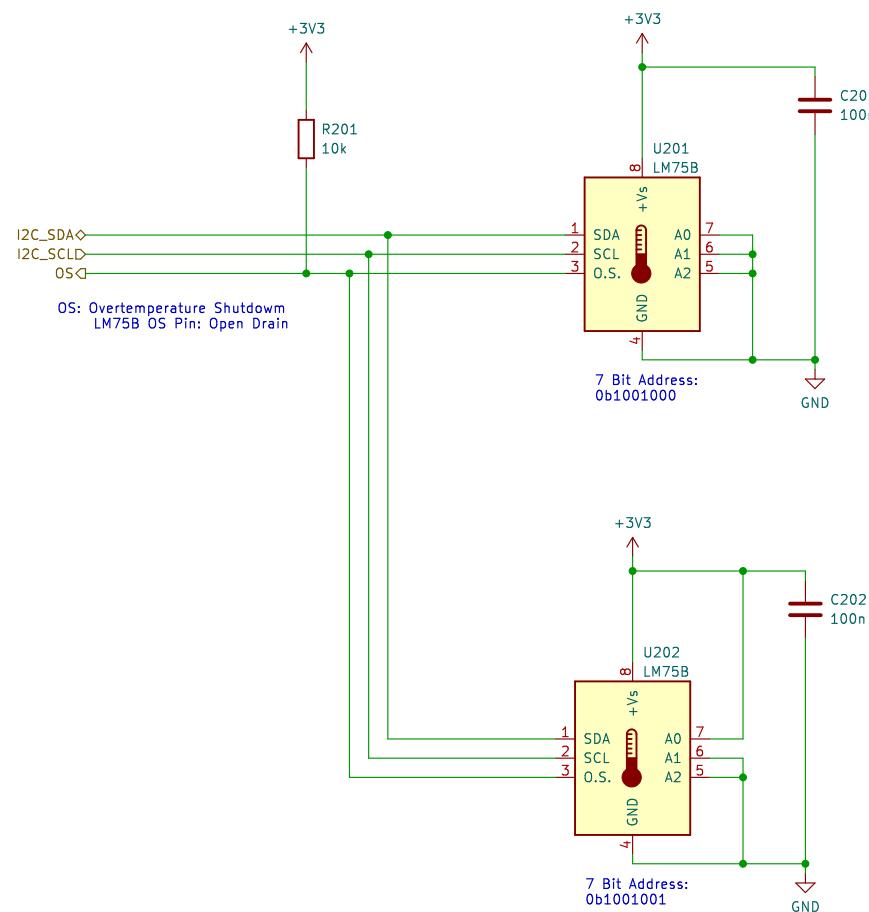
B

C

C

D

D



1 2 3 4 5 6

A

A

B

B

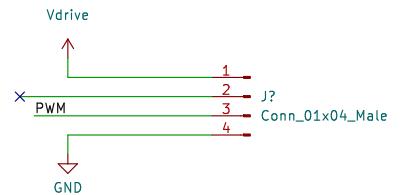
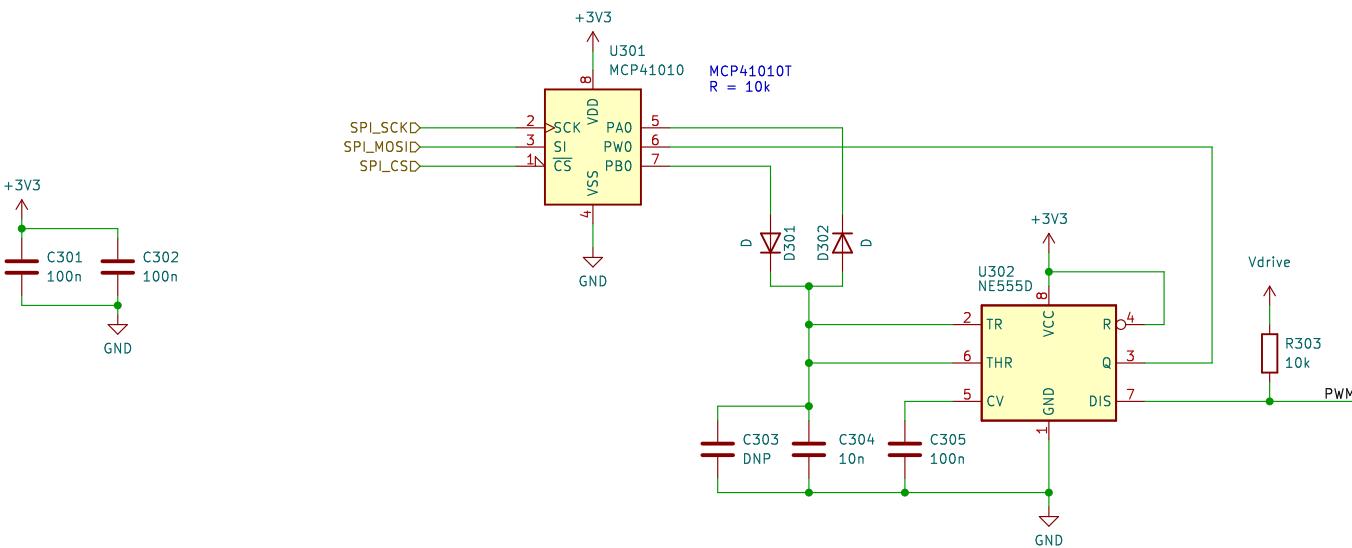
C

C

D

D

21



Title:
Sheet: /PWM/
File: gpio_expander.kicad_sch

Title:

Size: A4 Date:
KiCad E.D.A. kicad (6.0.0-0)

Rev:
Id: 3/3

1 2 3 4 5 6

Kolophon

Dieses Dokument ist ein $\text{\LaTeX} 2\epsilon$ (2022-06-01) Dokument der KOMA-Script Klasse unter der 3-Klausel-BSD Lizenz. Alle eigenen Zeichnungen sind mit TikZ und PGFPLOTS gesetzt. Kompiliert wurde es mit X \LaTeX und biber auf MacOs am 8. Juli 2022 von Prillwitz und Menzel.