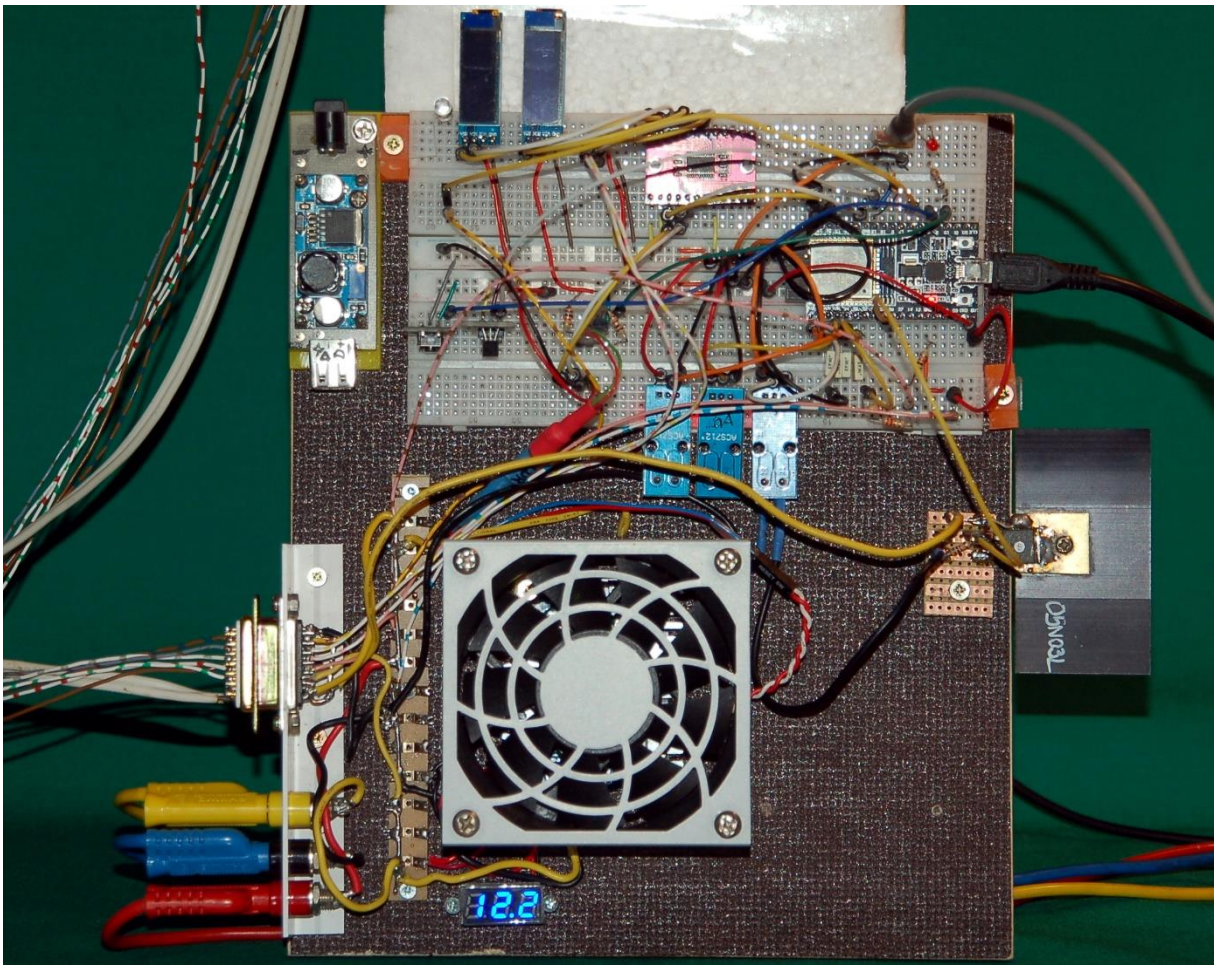


Abbildung 1: Testaufbau mit einer angeschlossenen Einheitf

Für den Beitrag gibt es eine deutsche und eine englische PDF-Fassung.

In der letzten Folge wurde eine Kühlbox mit Hilfe eines Peltierelements gebaut. Wir haben den Buck-Converter des Netzteils aus der ersten Folge so erweitert, dass die Ausgangsspannung durch einen ESP32 gesteuert werden konnte. Wir haben auch eine Formel gefunden, durch welche die Berechnung der Ausgangsspannung in Abhängigkeit vom Tastverhältnis des PWM-Signals berechnet werden konnte. Heute bauen wir das Ganze auf bis zu drei solcher Einheiten aus und entwickeln ein Programm zur Steuerung. Darin erfahren Sie, wie durch Listen gleichartige Eigenschaften verschiedener Objekte durch eine Methode verwaltet werden können und wie man Methoden einer Klasse verändert, ohne in die Klassendefinition einzugreifen – Stichwort: Decorator. Was Sie im Einzelnen erwartet, sage ich gleich, aber zuerst einmal willkommen zum 4. Teil der Reihe mit dem Titel

Peltierelemente - Der ESP32 steuert den Selbstbau-Kühlschrank

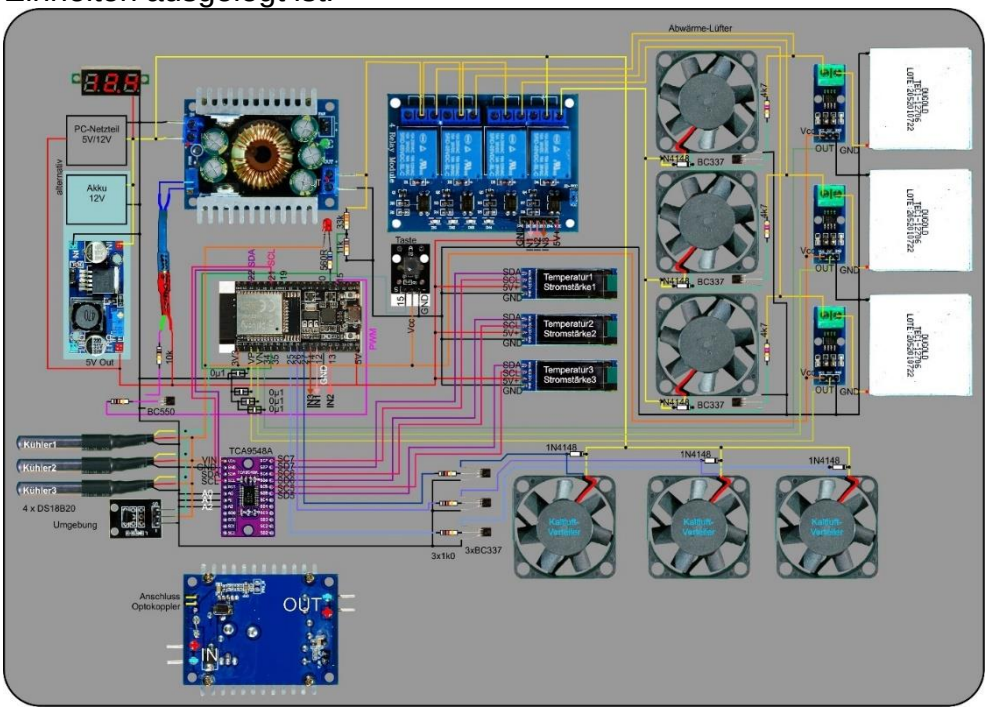
Ein ESP32 übernimmt die Überwachung der Temperaturen, die Steuerung der Stromstärke in den thermoelektrischen Wandlern und die Anzeige der wichtigsten Daten auf je einem OLED-Display an der jeweiligen Einheit. Natürlich müssen dabei Stromstärken bis ca. 4 A gemessen werden. Als Herausforderung der besonderen

Art, sind bis zu drei OLED-Displays, die alle dieselbe I2C-Geräteadresse haben, anzusteuern. Die Motoren der Abwärmelüfter sollen automatisch mit dem Einschalten der Spannung an den Peltierelementen anlaufen und die Lüfter in der Box sollen durch den ESP32 einzeln geschaltet werden können. Auch hier gehen die Kenndaten des Motors mit 12 V und 80 mA deutlich über die Kapazität eines GPIO-Anschlusses hinaus. Damit die geregelte Spannung des Buck-Converters, möglichst verlustlos geschaltet, an die Peltierelemente gelangen kann, hatte ich für jeden Zweig ein Relais vorgesehen. "Hatte" deshalb, weil ich im Lauf der Entwicklung gezwungen war, den Gedanken an die Relais aufzugeben, weshalb, das verrate ich weiter unten im Hardware-Kapitel.

Auch programmtechnisch gilt es, die umfangreiche Hardware möglichst effektiv zu bedienen, beim RAM-Speicher liegen wir wohl kurz vor dem Limit. Die Daten der Kühlereinheiten werden über Listen definiert und über Indizes gesteuert und verwaltet. Da kommt uns zugute, dass in einer Liste unter MicroPython alle möglichen Objekte untergebracht werden können. Die Arduino-IDE erlaubt in einem Array nur einfache Variablentypen. Einige der Funktionen sind für positive (HIGH-Level getriggert) sowie negative Logik (LOW-Level getriggert) geschrieben und daher sehr anpassungsfähig. Das erleichtert die Übertragbarkeit auf andere Anwendungen. Für Tests wird zunächst die PC-Tastatur eingesetzt, um Befehle an den Controller zu senden. In der nächsten Folge werden wir dafür entweder eine App auf dem Android-Smartphone oder ein Programm in Verbindung mit der AZ-Touch-Wandbox mit TFT-Display (320x240 Pixel) entwickeln und einsetzen.

Hardware

Zur besseren Orientierung stelle ich den Schaltplan bereits an dieser Stelle vor. Darauf erkennen Sie gut die verschiedenen Baugruppen. Es ist möglich, das Projekt mit einer, zwei oder allen drei Einheiten zu verwirklichen. Dementsprechend wird die Anzahl der benötigten Teile aus der nachfolgenden Liste gewählt, die für drei Einheiten ausgelegt ist.



Ja, das wäre der Schaltplan gewesen, wenn die Sache mit den Relais nicht gewesen wäre.

Abbildung 2: Thermobox_Schaltung 1

Ein Relais ist ein elektromagnetisch betätigter Schalter, und das Magnetfeld beim Einschalten der Relaisspule macht am Kunststoffgehäuse des Relais ebenso wenig Halt, wie der Restmagnetismus nach dem Ausschalten.

"Wo liegt dabei das Problem?", meinen Sie? Nun, wir wollen die Stromstärken in den Zweigen messen, und das soll mit den ACS712-Sensoren geschehen. Die arbeiten aber mit Hallsensoren, nicht mit Spannungsmessung an einem Messwiderstand. Genau diese Hallsensoren erfassen nun nicht alleine das Magnetfeld des zu messenden Stroms, sondern auch das der Relais und zwar sehr dominant auch noch auf eine Entfernung von mehreren Zig-Zentimetern.

Anstelle der Relais wurden nun drei MOSFET-Treiberstufen in Dienst gestellt. Von den Schalteigenschaften her nicht ganz so effektiv wie ein Relais, haben die MOSFETs den Vorteil, dass kein Kontaktverschleiß auftritt. Ein Exemplar des Schaltplans in DIN A4 steht zum Download bereit.

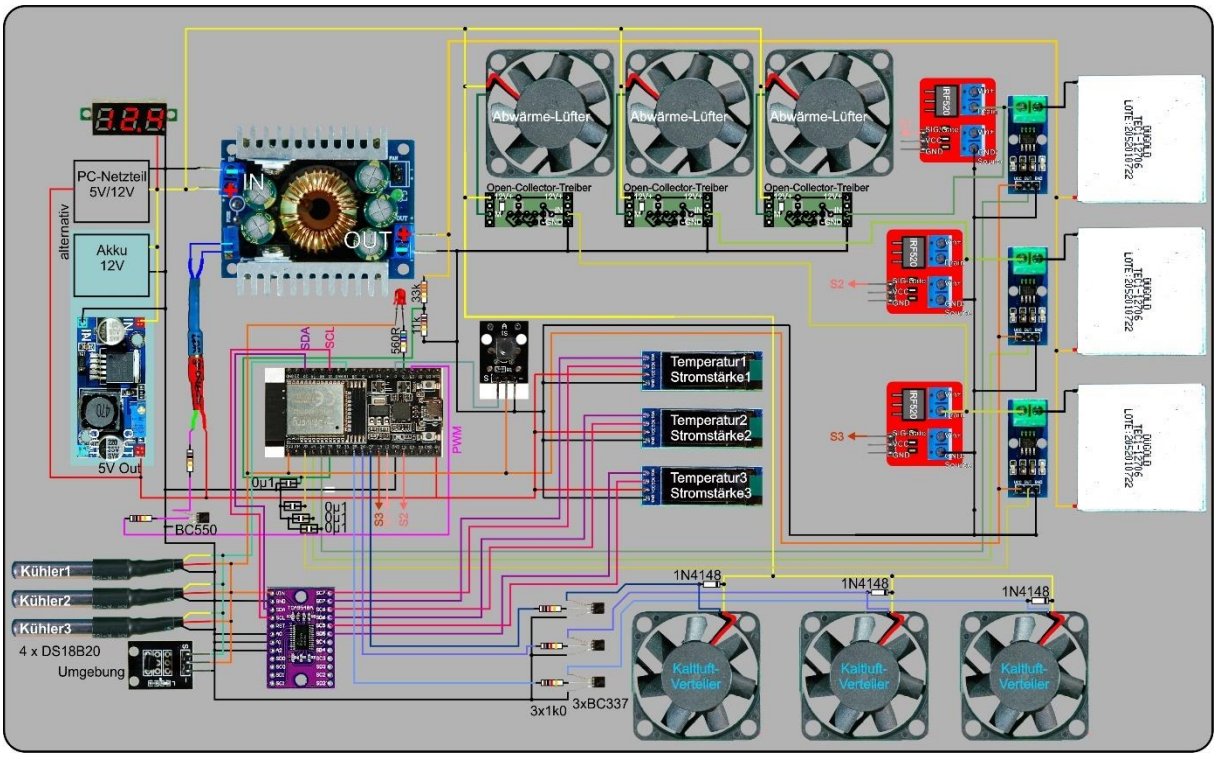


Abbildung 3: Thermobox_Schaltung mit MOSFET-Stufen

Hier kommt die Liste der Bauteile. Sie können das Projekt mit nur einer Cooler-Einheit aufbauen, dann benötigen Sie keinen I2C-Multiplexer, müssen aber das Programm an einigen Stellen gravierend ändern. Ab zwei Einheiten ist der I2C-Multiplexer notwendig, um die OLED-Displays getrennt anzusteuern. Wie oben bereits erwähnt, haben die Module eine fest zugeordnete Geräteadresse und können somit nicht parallel an einem I2C-Bus betrieben werden. Dieses Projekt zeigt auf, wie solche Probleme grundsätzlich bezüglich Hardware, aber auch programmtechnisch raffiniert gelöst werden können, ohne in bestehende Module (Programmbibliotheken) eingreifen zu müssen. Unter der Arduino-IDE sind mir derartige Tricks bislang noch nicht begegnet – MicroPython kann damit aufwarten.

Teileliste:

3	Thermoelektrischer Wandler 40 x 40 mm
1	DC-DC-Buck-Converter 8A für 2 Einheiten oder DC-DC-Buck-Converter 12A für 3 Einheiten
1	ESP32
1	DS18B20 als Modul oder Einzel-IC
3	DS18B20 mit 1m Kabel, wasserdicht
1	PCA9548A I2C IIC Multiplexer
1	Taster KY-004
3	0,91 Zoll OLED I2C Display 128 x 32 Pixel
3	ACS712 Stromsensor 5A Messbereich
3	IRF520 MOS Driver Modul 0-24V 5A
1	0,28 Zoll Mini Digital Voltmeter Spannungsmesser mit 7-Segment Anzeige
1	nur bei 12V-Akkubetrieb: LM2596S DC-DC Netzteil Adapter Step down Modul
1	LED weiß 5mm
1	LDR 5mm
1	20-25mm Messingrohr 6mmØ und 0,5mm Wandstärke
	Dichtmasse
	Schrumpfschlauch Stücke
1	Widerstand 1kΩ
1	Widerstand 10kΩ
1	NPN Standard Transistor zum Beispiel BC550
2	Zweipolige Stiftleiste gerade
6	Transistor BC337 (30V, 800mA)
3	Transistor BC548 (30V, 100mA)
3	Widerstand 4,7kΩ
9	Widerstand 1,0kΩ
6	Diode 1N4148
6	Stiftleiste 4-polig gerade
3	Stück Lochraster-Platine 4 x 11 Pins
3	Stück Lochraster-Platine 4 x 14 Pins
1	LED rot
1	Widerstand 560Ω
1	zweipolige Buchsenleiste
1	Widerstand 33kΩ
1	Widerstand 11kΩ (Ersatz 10kΩ + 1kΩ)
2	Breadboard
	diverse Jumperkabel

Für den Thermokopf

2*3	Rippen-Kühlkörper
2*3	dazu passende PC-Lüfter
4*3	Kunststoffwinkel 10 x 10 x 30mm
	Wärmeleitpaste
evtl.	einige Aluplatten Abschnitte (siehe Text)
	wasserfest verleimte Mehrschichtplatte
3	Stück Styroporplatte 126 x 126 x 10
	diverse Schrauben, Muttern,
	Zuleitungskabel für das Peltier-Element mit mindestens 1mm²

Für den Kühlbehälter

3	Styroporplatte 20 oder 30mm
	Paketklebeband

Der Aufbau des Thermokopfs und des Kühlbehälters wurde bereits in Folge 3 genau beschrieben, ebenso die Herstellung des Optokopplers als Ergänzung für den 8A-Buck-Converter, welche den Regler fernsteuerbar macht. In ähnlicher Weise kann auch der 12A-Converter aufgemufft werden. Im Schaltplan ist dieses Modul von oben dargestellt. Der Anschluss des LDR im Optokoppler ist auf der Platinenunterseite gelb markiert.

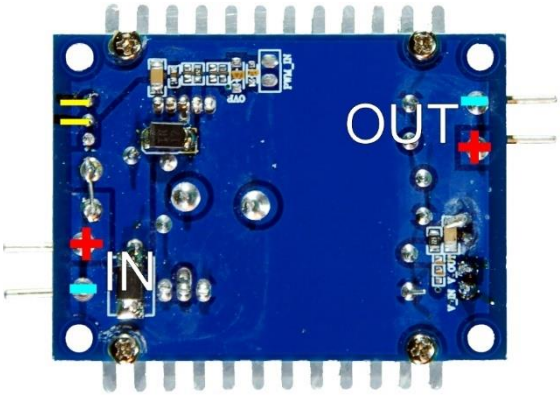


Abbildung 4: 12-Ampere-Regler_Unterseite

Gehen wir den Schaltplan kurz durch, und schauen wir uns die Funktionseinheiten an.

Das Voltmeter, ganz links oben, zeigt die Spannung am Akku oder PC-Netzteil an. Für beide Energielieferanten sollte noch ein Schalter vorgesehen werden, der im Schaltplan nicht eingezeichnet ist. Im Fall des Akkus muss der Schalter maximal 12A vertragen. Beim Netzteil genügt ein kleiner Schalter, der am Stecker den Anschluss 16 (PS-ON) mit einem Masse-Anschluss verbindet. Als Steckstifte lassen sich gut 1,3mm-Pfostenstifte verwenden (siehe Abbildungen 5 und 6).

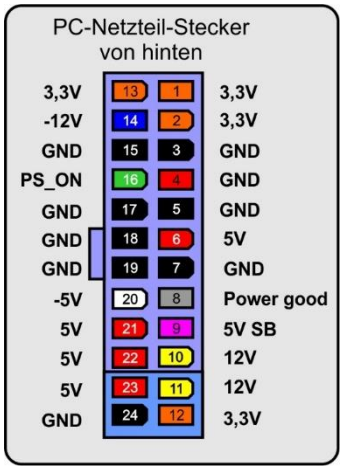


Abbildung 5:Netzteilstecker 24-polig von hinten

Abbildung 6:Netzteilstecker halb von vorne

Das PC-Netzteil muss selbstredend am 12V-Anschluss bis zu 12A liefern können, das heißt, der Anschluss muss eine Abgabeleistung von 144W aufwärts haben, damit drei Kühleinheiten angeschlossen werden können. Das PC-Netzteil hat außerdem den großen Vorteil, dass es auch einen 5V-Ausgang hat. Bei Akkubetrieb muss dafür ein separater Regler in Dienst genommen werden. Wenn wir für die 5V-Versorgung die 5V SB-Leitung (Pin 9) des Netzteils verwenden, dann kann der ESP32 über einen, der noch freien, GPIO-Anschlüsse die 12V-Versorgung selbst freigeben, indem er den Anschluss 16 auf GND-Potenzial zieht. Das Netzteil bleibt dann fest am 230V-Netz. Der Befehl zum Einschalten kann im Endausbau dann alternativ auch über die Funkverbindung erfolgen.

Die geregelte Spannung aus dem Buck-Converter wird den Peltier-Elementen am roten Anschluss zugeführt. Vom schwarzen Anschluss geht es zurück zum Drain-Anschluss des MOSFET-Transistors und von dessen Source über das Strommessmodul ACS712 05A auf GND. Jeweils eine invertierende Transistorschaltstufe sorgt auf der Warmseite dafür, dass immer dann, wenn durch das Peltierelement Strom fließt, auch der Lüfter auf der Warmseite automatisch anläuft. So sehen die Schaltung für die Lochrasterplatine und der zugehörige Schaltplan aus.

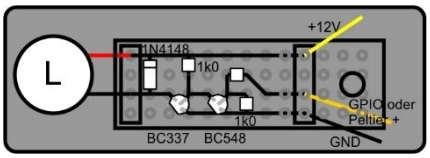


Abbildung 7: Transistortreiber invertierend

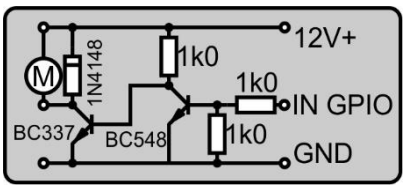


Abbildung 8: Transistortreiber invertierend_Schaltung

Die anderen drei einfachen Schaltstufen aktivieren, durch das Signal vom ESP32, die Lüftermotoren auf der Kaltseite. Die Diode dient auch hier zum Schutz des Transistors. Der Ring auf der Diode muss auf der Seite der positiven Spannungszuführung liegen. Eine Verpolung tötet die Diode und den Transistor.

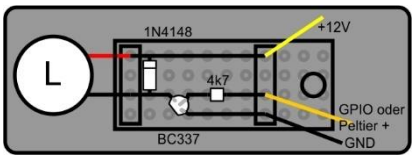


Abbildung 9: Transistortreiber

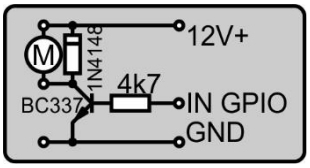


Abbildung 10: Transistortreiber_Schaltung

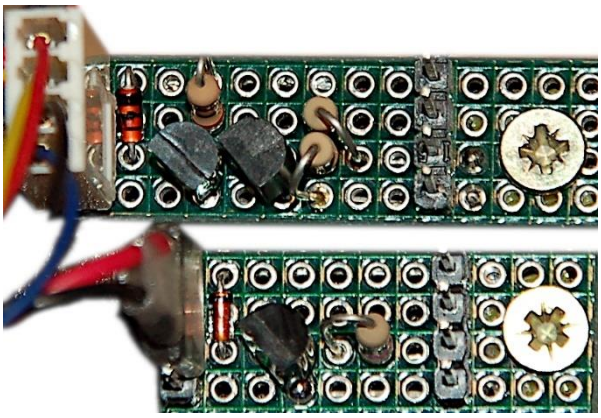


Abbildung 11: Transistor-Schaltstufen

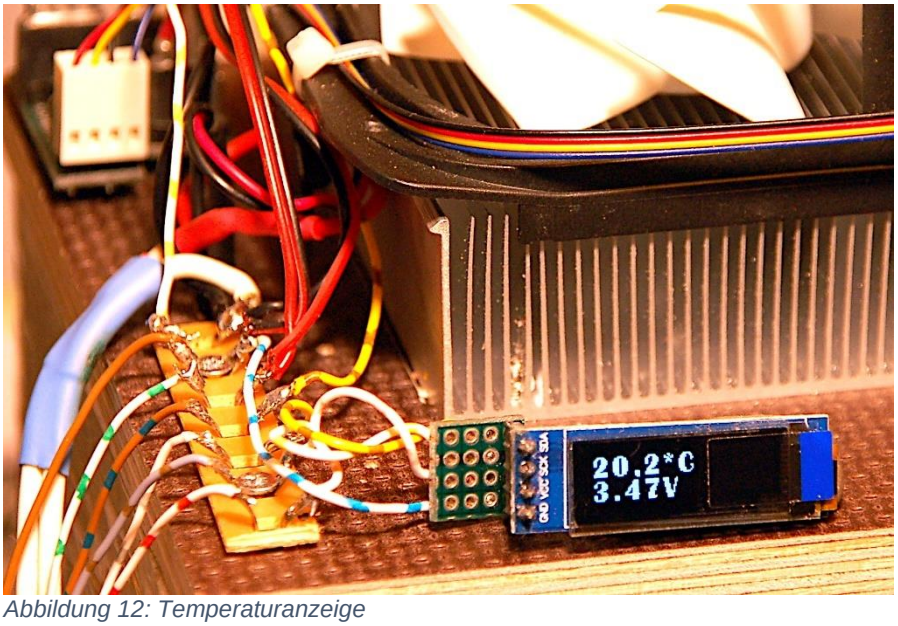


Abbildung 12: Temperaturanzeige

Die OLED-Displays werden am Cooler angebracht und geben Auskunft über Temperatur, Spannung oder Stromstärke. Das Problem, dass alle Displays eine nicht veränderbare Geräteadresse haben, wird durch den Einsatz eines PCA9548A gelöst. Dieser I2C-Multiplexer stellt das Signal vom ESP32 an die Ausgänge durch, die im Konfigurationsregister des Bausteins als Bit eine 1 stehen haben. Vor der Ausgabe an ein OLED-Modul muss also erst der Kanal geschaltet werden, danach erfolgt die Ansteuerung so, als wäre das OLED das einzige im Kosmos. Damit das transparent und ohne Änderung an den Methoden der Klassen OLED und CharSet erfolgen kann, werden die entsprechenden Methoden "verkleidet", in MicroPython heißt dieser Vorgang Dekorieren. Dazu gibt es ganz am Schluss vor dem Programmlisting noch Hinweise.

Zur Anzeige wird ein vergrößerter Zeichensatz verwendet. Sie können mit dem Paket font2py.rar eigene Zeichensätze aus dem Windows Font-Ordner clonen. Die im Paket enthaltene Batchdatei makecharset.bat erledigt das mit der Angabe der 4 Parameter font, size, chars und Quellenpfad in einem Arbeitsgang. Der String chars gibt die Zeichen aus dem Zeichensatz font an, die sie clonen möchten. Wie das im Einzelnen geht, steht in dem PDF-Dokument Erstellen von großen oder eigenen

Zeichensaetzen.pdf. Eine Kurzanleitung mit Quellenangaben finden Sie hier. Alternativ können Sie auch fürs Erste den Zeichensatz geometer_18.py downloaden.

Die drei Temperaturfühler mit den Kabeln werden jeweils durch den Deckel in die Kühlbox geführt und hängen zusammen mit dem Sensor für die Umgebungstemperatur alle am gleichen One-Wire-Bus.

Eine LOW-aktive Taste dient zum geordneten Rückzug. Sie kann jederzeit gedrückt werden, um das Programm abzubrechen. Geordnet bedeutet, dass vor dem "Aus" aufgeräumt wird. Die Stromzuführung zu den Peltierelementen wird unterbrochen, die Anzeigen werden ausgeschaltet.

Damit sind wir am ESP32 selbst angekommen. Noch nicht erwähnt wurden die Anzeige-LED an GPIO2, der PWM-Ausgang an GPIO15, die Messung der Ausgangsspannung am Buck-Converter über Analogeingang GPIO35 und die Eingänge für die Strommessung VN, VP und GPIO34. Zur Verminderung des Rauschens wird jeder Eingang mit einem Kondensator von 0µ1 gegen GND abgeblockt, ferner wird für die Messung der analogen Signale standardmäßig der Mittelwert aus 100 Einzelmessungen verwendet. Dennoch besteht trotz Korrekturformeln für die ADC-Kennlinie eine Messungenauigkeit von bis zu 10%.

Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
packetsender für Windows
font2py.rar

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

Die MicroPython-Programme zum Projekt:

thermobox.py
button.py
charset.py
geometer_18.py
i2cbus.py
oled.py
ssd1306.py
font2py.rar Font-Konverterpaket

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung. Darin gibt es auch eine Beschreibung, wie die MicropythonFirmware auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, bevor der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm compilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen, über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Macro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein compiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Das Betriebsprogramm

Im Programm für den ESP32 sind bereits alle Sequenzen enthalten, die sowohl für erste Tests dienen, aber auch schon für den Funkbetrieb vorgesehen sind. Verschiedene Funktionen lesen Sensoren aus und stellen die Spannung, beziehungsweise die Stromstärke oder Temperatur ein oder schalten die Peltier-Elemente oder die Lüfter im Kühlraum.

Alle, bei mehreren Einheiten, mehrfach vorhandene Teile werden durch Listen verwaltet. Dazu gehören Stromstärkemessung, Schaltstufen, OLED Displays, Thermosensoren sowie die Kaltluftwirbler. Die Zuordnung der GPIO-Pins wird in for-Schleifen erledigt. Entsprechende Funktionen erhalten, neben anderen eventuellen Parametern, den Index der zu bedienenden Einheit. Als Beispiel werfen wir einen Blick auf die Definition der MOSFET-Steuer-Pins. Die Bezeichnung "relais" oder "rel" erinnert noch an den ursprünglichen Ansatz mit der 4-fach-Relaiseinheit. Die MOSFET-Stufen sind übrigens, im Gegensatz zu den Relaisstufen, HIGH-getriggert.

```
relaisPin=(12,13,14)    # Relais0,1,2
rel0,rel1,rel2=0,1,2
on=1; off=0
lowTriggered=0
highTriggered=1
rel=[0,0,0]             # Liste der Relais-Pins
for i in range(3):      # Pin definieren + Relais aus
    rel[i]=Pin(relaisPin[i],Pin.OUT)
    rel[i].value(1)
```

In einem Tupel geben wir die Pinnummern an. Tupel haben, im Gegensatz zu Listen, den Vorteil, dass die Abarbeitung schneller ist. Der Nachteil gegenüber Listen ist, dass, letztere vielseitigere Methoden für die Verarbeitung zur Verfügung stellen und, dass Listeninhalte veränderbar (mutable) sind, was für Tuples und auch Strings nicht gilt (immutable). Elemente von Listen können nachträglich den Wert ändern, Elemente von Tuples oder Zeichen in Strings nicht. Aber jetzt zurück zur Definition der Relais-Pins.

Die Indizes werden den Variablen rel0 etc. zugewiesen. Die Festlegung einiger Variablen fördert die Lesbarkeit später im Programm. Dann erzeugen wir eine Liste, die gleich im Anschluss in der for-Schleife mit den Pinobjekten zum Schalten der MOSFETs gefüllt wird. Weil die MOSFETs HIGH-Level getriggert sind, schaltet eine 1 am GPIO-Pin jeden Transistor ein, eine 0 aus.

Zu den MOSFETs gehört die Funktion **relais**(). Diese nimmt als Parameter den Index n und den gewünschten Status state sowie das Flag trigLevel, welches optional mit dem Wert highTriggered (=1) vorbelegt ist.

```
def relais(n,state,trigLevel=highTriggered):
    status=rel[n].value()       #  Status merken
    s = state                   # Zielstatus übernehmen
    if trigLevel==lowTriggered: # Pegel tauschen, wenn lowtr.
        status=(0 if status==1 else 1)
```

```
        s=(0 if state==1 else 1)
    rel[n].value(s)
    return status # vorheriger Status zurueck
```

Der bestehende Status wird abgefragt und der Soll-Status der lokalen Variablen s zugewiesen. Das if-Konstrukt wandelt die logischen Pegel in die elektrischen um, falls die Schaltstufe LOW-Level getriggert ist, also durch eine 0 am ESP32-Ausgang eingeschaltet wird. In diesem Fall wird auch der zuvor ausgelesene Pegel invertiert, damit der Rückgabewert wieder logisch stimmt. Jeder der folgenden Aufrufe schaltet also die entsprechende Schaltstufe ein:

3 HIGH-Level getriggerte Transistorstufen:
```
   for n in range(3):
      relais(n,on)
```

LOW-Level getriggerte Relais-Stufe :
```
>>> relais(0,on,lowTriggered)
```

Im ersten Fall gilt der Defaultwert highTriggered. Im zweiten Fall muss das Triggerverhalten angegeben werden, damit das Relais korrekt schaltet.

Die Funkion **getRelaisState**() fragt den Schaltzustand einer Schaltstufe ab. Die Ausgabe wird wieder der Trigger-Eigenschaft angepasst.

```
def getRelState(n,trigLevel=highTriggered):
    status=rel[n].value()       # Status holen
    if trigLevel==lowTriggered: # Pegel tauschen, wenn lowtr.
        status=(0 if status==1 else 1)
    return status
```

Die Basisfunktion **readADC**() bedient Funktionen zur Strom und Spannungsmessung mit den rohen Zählwerten. Sie nimmt den Index und eine optionale Anzahl von Wiederholungsmessungen zur Filterung und Beruhigung des Ergebnisses, das sich als Mittelwert der Einzelmessungen ergibt, 100 Einzelmessungen sind Standard.

Die Stromsensoren liefern bei 0,0A am Ausgang eine Spannung vom Wert der halben Betriebsspannung ab. Die Funktion **getQuiescentCurrentRaw**() ermittelt die Zählwerte, nach dem die entsprechende Leitung sicher durch den MOSFET unterbrochen wurde. Nach einer Pause von 0,2 Sekunden wird **readADC**() aufgerufen und danach der Transistor in den vorherigen Schaltzustand versetzt. Die Null-Ampere-Zählwerte legt die Funktion in der Liste **I0[ ]** ab. Die ursprünglich eingesetzten ACS712 20A lieferten bei der Betriebsspannung von 3,3V zu kleine und sehr unsichere Werte, weil die Sensoren normalerweise an 5V arbeiten. Durch den Austausch mit 5A-Typen konnte die Genauigkeit gesteigert werden, wenngleich auch die Eichfaktoren, Spannung zu Stromstärke, aus dem Datenblatt für die Spannung 3,3V nicht zutreffen. Es musste selbst ein Kalibrierfaktor durch Zuhilfenahme eines DAM (Digital-Ampere-Meter) gefunden werden. Die analogen Eingänge des ESP32 vertragen leider keine Spannungen größer 3V, deswegen wurde von einer Spannungsversorgung mit 5V für die ACS712 vorsorglich abgesehen.

readAmps() nutzt die Rohwerte. Nachdem die Gültigkeit des Index überprüft wurde, bringt der Aufruf von **readADC**() den rohen Zählwert. Davon wird der I0-Wert des Kanals subtrahiert und die Differenz in einen Stromstärkewert umgerechnet. Dazu wird der Wert **mVpA** (Millivolt pro Ampere) verwendet. Die Werte für diesen Faktor sind auch vom Exemplar abhängig und aus diesem Grund müssen wir die einfache Variable **mVpA** in eine Liste der modulspezifischen Werte umwandeln. Das ist im Listing bereits geschehen. Wieder ermöglichen Listen die Verarbeitung einer Größe durch nur eine einzige Methode.

```
mVpA= [73,73,75]
```

….

```
def readAmps(n,repeat=100): # Stromwert n holen
    if n in range(3):
        Icnt=readADC(n,repeat)
        Icnt-=I0[n]
        return int(Icnt/4096*3600/mVpA[n]*100+0.5)/100
        #counts/maxcount*Uref/ApmV
    else:
        return 9999
```

War der angegebene Index nicht zulässig, kann anhand des Rückgabewerts die Gültigkeit des Stromstärkewerts gegebenenfalls geprüft werden.

Das Programm soll nur angeschlossene Einheiten steuern. Wir überprüfen die Anwesenheit, indem wir die entsprechende Stufe kurz einschalten und nachsehen, ob ein Strom fließt, welcher wenigstens der minimal an einem Peltierelement anliegenden Spannung entspricht. Das erledigt die Funktion **findCoolers**(), welcher die Triggereigenschaft der Schaltstufe übergeben werden muss. Wir rufen die Funktion **setVoltage**() mit dem gewünschten Spannungswert von 4V und bekommen den Duty Cycle des bisherigen Spannungswerts zurück, den wir uns merken. Das Einschalten jeder Stufe liefert dessen vorherigen Schaltzustand, den merken wir uns auch. Wenn der festgestellte Stromstärkewert größer als 0,3A ist, wird die Einheit in der Liste **coolerPressent** als True verzeichnet. Danach werden der Schaltzustand des Transistors und zum Schluss die vorherige Spannung wieder hergestellt.

In der vorangegangenen Blogfolge hatten wir den Zusammenhang zwischen dem Taktverhältnis D des PWM-Signals und der resultierenden Spannung Ua am Ausgang des Buck-Converters durch eine Formel dargestellt.

$$Ua = 17{,}8894 \cdot \left(\frac{1}{D}\right)^{0{,}39279}$$

Abbildung 13: Gleichung der Trendlinie:

Die Parameter **a=17,8894** und **b=0,39279** sind von den Bauteilen des SB-Optokopplers sowie deren Anordnung abhängig. Deswegen müssen sie durch eine Messreihe selbst ermittelt werden. Wie das geht, ist in Folge 3 beschrieben.

Allerdings brauchen wir jetzt die genau umgekehrte Zuordnung Ua -> D. Den Zusammenhang erhalten wir durch Umformen der Gleichung und Auflösung nach D.

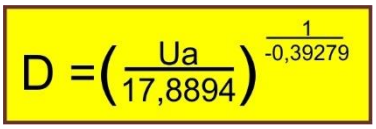


Abbildung 14: Duty Cycle aus Spannung

oder allgemein:

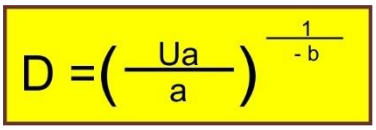


Abbildung 15: Duty Cycle aus Spannung_allgemein

Diese Formel wird von der Funktion **volt2duty**() benutzt. Zulässige Spannungswerte werden umgerechnet und gleichzeitig die Prozentangabe weiter an den Bereich 0 bis 1023 angepasst.

Den Dienst dieser Funktion nutzen wiederum die Funktionen **setCurrent**() und **setVoltage**(). **setCurrent**() versucht zunächst, die Spannung so einzustellen, wie es für die gewünschte Stromstärke nach der ohmschen Widerstandsformel notwendig wäre. Gelingt das nicht, weil die berechnete Spannung nicht im Bereich **Umin** bis 16V liegt, wird die minimal oder maximal mögliche Spannung eingestellt und die Funktion verlassen. Auch wenn die Ist-Stromstärke nicht mehr als 100mA von der Soll-Stromstärke abweicht, wird die Funktion verlassen. War eine Einstellung möglich, aber ergab sich eine zu große Differenz zwischen Ist- und Sollwert, versucht die Funktion zusätzlich eine sukzessive Approximation in zehn Schritten zu je 0,5V.

Die Ergebnisse der Spannungsmessung weichen im unteren und oberen Bereich erheblich von den Werten ab, die man mit dem DVM (Digital-Volt-Meter) misst. Um dieses Übel zu beseitigen, müssen wir wieder eine Messreihe aufnehmen, in der die Soll-Spannung vom DVM und die vom ESP32 gemessene Spannung erfasst werden.

Libre Office liefert dann mit der grafischen Auswertung eine Potenzfunktion, von dessen Funktionsterm uns die Koeffizienten interessieren, die wir gekürzt in unser Programm übertragen. Unsere Ansprüche an die Genauigkeit entscheiden darüber, ob wir eine Funktion zweiter oder dritter Ordnung wählen. Das Tabellenblatt steht zum Download bereit.

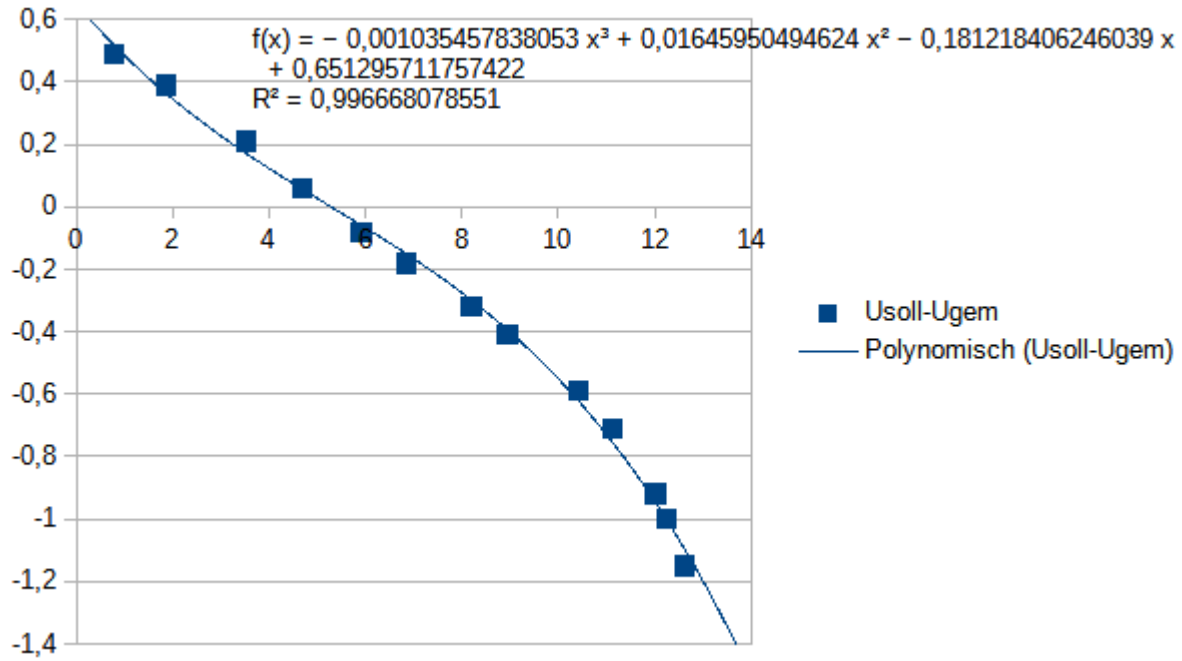


Abbildung 16: Grafik zur Fehlerkorrektur bei der Spannungsmessung

Im Programmtext sieht das dann so aus. Mit dem Faktor **3793** anstatt 3600 habe ich zunächst versucht, das Ergebnis zu trimmen, was aber nicht ausreichte. **UFaktor** berücksichtigt den Spannungsteiler 33kΩ:11kΩ am Analog-Eingang GPIO35, der die 12V auf ESP32-verträgliche 3,0V reduziert.

```
spannung=0
A=-0.00103545
B=0.0164595
C=-0.1812184
D=0.6512957
```

....

```
def readUist(repeat=100):
    global spannung
    Ucnt=readADC(Ubat,repeat) # counts/maxcount*Uref * 4V/Vmes
    U=int((Ucnt/4096*3793 * UFaktor)/10+0.5)/100
    spannung=A*U**3+B*U**2+(C+1)*U+D
    return spannung
```

Zum Einstellen und Überwachen der Temperatur in der Coolerbox dienen die Funktionen **setTemp**() und **holdTemp**(). Letztere versucht durch zyklischen Aufruf in der Hauptschleife die Solltemperatur einzustellen und zu halten. Das gelingt natürlich nur im Serverbetrieb. In der Testphase wird der Schleifendurchlauf durch den input-Befehl geblockt.

Die Kaltluftverteiler werden über die Funktion **setFanState**() geschaltet. Weil es sich um einfache Transistorstufen mit positiver Logik handelt, wurde von vornherein auf das Einfügen der Triggereigenschaft verzichtet.

getTemp() ist die Funktion, welche die Temperatur in der Kühlereinheit n bestimmt und mit einer Nachkommastelle zurückgibt. Eine ungültige Temperatur wird als 9999

im Fehlerfall zurückgegeben. Wird eine nicht vorhandene Cooler-Einheit angesprochen, kommt -9999 zurück.

Für die Anzeige von Werten in den OLED-Displays ist die Funktion **displayValues**() zuständig. Sie überprüft, ob die Stromstärkeregelung eingeschaltet ist (Iflag ist ungleich 0) und gibt in Abhängigkeit davon zur Temperatur die Stromstärke oder den Wert der eingestellten Spannung aus.

Als umfangreichste Funktion übernimmt **parse**() die Aufgabe, ankommende Befehle zu decodieren und die entsprechenden Aktionen einzuleiten. Weiterhin werden Rückmeldungen zur Aktion oder Fehlermeldungen erzeugt. Letzteres hilft sowohl dem Anwender als auch dem Programmierer, weil die Fehlerstelle leicht identifiziert werden kann.

Alle Befehle haben einen ähnlichen Aufbau. Ein Buchstabe steht für die Art des Jobs, gefolgt von einem Doppelpunkt. Bis auf das Einstellen der Spannung, wo jetzt sofort der Spannungswert folgt, kommt nach dem Doppelpunkt die Nummer der Kühlereinheit. Außer bei der Statusabfrage mit "S" folgt ein weiterer Doppelpunkt und danach ein Zahlenwert. Für die Wartung und Diagnose gibt es Ein-Tasten-Befehle.

```
# U:float           Spannung einstellen
# I:[0|1|2]:float   Stromstärke einstellen + halten
# T:[0|1|2]:float   Temperatur einstellen + halten
# C:[0|1|2]:[0|1]   Cooler-Schalter aus/ein
# F:[0|1|2]:[0|1]   Coller-Fan aus/ein
# S:[0|1|2]         Status melden
# R                 Nullstromwerte messen
# V                 Spannung messen
# P                 Flags und angeschlossene Einheiten
# A                 Stromstärken messen
```

Nach diesem Schema geht die jeweilige Abteilung vor, nachdem eine gültige Job-Kennung gefunden wurde. Zum Einsatz kommen die Methoden **find**() und **split**() des untersuchten **string-Objekts**. Bei der Nummer der Kühlereinheit und dem Schaltzustand wird der Gültigkeitsbereich überprüft. Die Umwandlung vom String in eine Fließkommazahl ist durch **try** und **except** abgesichert, um Programmabstürze zu verhindern. Die Rückmeldung erfolgt in dem Tupel **(art,act,value)**

Kommando:u:8.47
from 999.999.999.999:99999
Content = u:8.47
U:Aktuelle Spannung:8.43
:fehlerfrei:0

art enthält die Jobkennung, **act** die Meldung und **value** die Fehlernummer. Die 0 steht für OK. Die Fehlernummer steht am Ende des Strings, weil sie so vom auftraggebenden Prozess leicht gefunden und isoliert werden kann. Eine Sequenz der folgenden Art liefert den Index in eine Liste mit Klartextmeldungen oder Funktionsnamen, die im Fehlerfall angezeigt oder ausgeführt werden sollen.

Am Beginn des Deklarationsteils des Programms erfolgt die Auswahl des Modus zur Befehlseingabe. Zur Verfügung stehen lokales Funknetz (WLAN mit Accesspoint), der ESP32 als Accesspoint oder, wenn beides durch False abgewählt wurde, die PC-Tastatur. Natürlich können die Funktionen inclusive **parse**() auch via REPL von Hand aufgerufen werden.

```
# Auswahl der Betriebsart Netzwerk oder Tastatur:
# Netzwerk: Setzen Sie genau !_EINE_! Variable auf True
WLANconnect=False  # Netzanbindung ueber lokales WLAN
ownAP=False        # Netzanbindung ueber eigenen Accesspoint
# beide False ->> Befehlseingabe über PC + USB in Testphase
# Falls WLANconnect=True:
# Geben Sie hier die Credentials Ihres WLAN-Accesspoints an
#mySid = 'YOUR_SSID'; myPass = "YOUR_PASSWORD"
```

Der Verbindungsaufbau zum WLAN-Accesspoint sieht dann aus wie folgt.

```
# *******************************************************
# WLAN-Connection
# *******************************************************
if WLANconnect and (not ownAP):
    nic = network.WLAN(network.STA_IF) # erzeuge WiFi-Objekt
    nic.active(True)  # Objekt nic einschalten
    #
    MAC = nic.config('mac')  # binaere MAC-Adresse abrufen +
    myMac=hexMac(MAC)        # in Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
    # Verbindung mit AP im lokalen Netzwerk aufnehmen,
    # falls noch nicht verbunden, dann
    # connect to LAN-AP
    if not nic.isconnected():
      nic.connect(mySid, myPass)
      # warten bis die Verbindung zum Accesspoint steht
      print("connection status: ", nic.isconnected())
      while not nic.isconnected():
        blink(0.8,0.2,True)
        print("{}.".format(nic.status()),end='')
        sleep(1)
    # Wenn verbunden, zeige Verbindungsstatus & Config-Daten
    print("\nconnected: ",nic.isconnected())
    print("\nVerbindungsstatus: ",connectStatus[nic.status()])
    print("Weise neue IP zu:","10.0.1.101")
    nic.ifconfig(("10.0.1.101","255.255.255.0","10.0.1.20", \
                  "10.0.1.100"))
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
```

Hat die Verbindung geklappt, wird eine eigene, im WLAN noch nicht vorhandene, feste IP-Adresse vergeben, weil der ESP32 als **UDP-Server** läuft. Eine vom WLAN-Accesspoint über DHCP vergebene IP kann unterschiedliche Werte annehmen, das ist für einen Server keine gute Idee.

Falls die Verbindung über Funk erfolgen soll, wird jetzt ein Socket mit einem Timeout von 2 Sekunden eingerichtet. Der Timeout sorgt dafür, dass die Empfangsschleife auch noch andere Dingen tun kann, als nur auf eingehende Anfragen zu lauschen.

Ist kein Funkverkehr gewünscht, dann wartet eine input-Anweisung auf Kommandos, die dann ebenfalls dem Parser übergeben werden. Der Socket wird dabei auf 999.999.999.999:99999 gesetzt, um die Rückmeldung von einer Funkanfrage zu unterscheiden. Mit der Eingabe "e" wird das Programm sauber beendet. Das ist auch durch Drücken der Taste oder der Flash-Taste am ESP32 möglich.

Den Funkverkehr im WLAN können wir sehr gut mit Hilfe des Tools packetsender testen. Dort können die oben dargestellten Kommandos über die Tastatur eingegeben und gesendet werden. Die Antworten vom ESP32 werden auch ausgegeben. In der nächsten Folge werden wir uns näher damit beschäftigen.

Es lohnt noch ein Blick auf ein paar sehr interessante Zeilen am Anfang des Programms.

```
import britannic_18 as zs
from charset import CharSet
......
oledKanal=[5,6,7]
for i in range(3):
    iBus.writeToBus(1<<oledKanal[i])
    d=OLED(i2c,128,32)
    d.setYoffset(0)
    d.clearAll()

c=CharSet(zs,d) # stellt Routinen fuer grossen ZS bereit
# putValue und putSymbol werden dekoriert, damit das richtige
# Display mit der Nummer dpn angesteuert werden kann.
def switchToChannel(f):
    def g(dpn, *args, **kwargs):
        chnl=oledKanal[dpn]
        print("Kanal:",chnl,"calling",f.__name__)
        iBus.writeToBus(1<<chnl)
        xn=f(*args,**kwargs)
        return xn
    return g

c.putSymbol=switchToChannel(c.putSymbol)
c.putValue=switchToChannel(c.putValue)
d.clearAll=switchToChannel(d.clearAll)
d.writeAt=switchToChannel(d.writeAt)
```

Nach dem Import des Zeichensatzes britannic_18 und der Klasse CharSet definieren wir eine Liste, welche die Nummer der Cooler-Einheit (0..2) in die Kanalnummern des I2C-Multiplexers (5..7) übersetzt. Danach werden die drei OLED-Objekte erzeugt und initialisiert. Die for-Schleife schaltet den jeweiligen Kanal.

Der Deklaration einer Instanz c der Klasse CharSet, folgt die Deklaration der Funktion **switchToChannel**, die eine Funktion f als Argument nimmt und eine andere Funktion g, die in switchToChannel definiert wird, zurückgibt. switchToChannel ist ein sogenannter **Decorator**, der dazu dient, die ihm übergebenen Funktionen zu modifizieren. Das geschieht, ohne dass die Funktionen selbst verändert wird. Es werden lediglich vor oder nach dem Aufruf der übergebenen Funktion zusätzliche Anweisungen eingefügt. So gelingt es uns, mit Hilfe der letzten 4 dargestellten Zeilen, jeweils zwei Methoden der Klassen CharSet und OLED an unsere Bedürfnisse anzupassen. g stellt die modifizierte Funktion f dar. g stellt den gewünschten Kanal ein und ruft dann die übergebene Funktion f mit den Originalparametern auf. Die neue Funktion erhält nun einfach den Namen der Originalfunktion.

Sie erinnern sich, dass zum Ansteuern mehrerer OLED-Displays zuerst der richtige Kanal am TCA9548 eingestellt werden muss. Genau das erledigt also unser Decorator, indem er dem bisherigen Aufruf in der Parameterliste die Angabe der Kanalnummer n voranstellt - g(dpn, *args, **kwargs).

Aus d.clearAll() wird somit d.clearAll(n),
aus d.writeAt("test",0,0) wird d.writeAt(n,"test,0,0) und aus
d.putSymbol(c,xpos=0,ypos=0,show=True) wird
d.putSymbol(n,c,xpos=0,ypos=0,show=True) …

Eine sonst notwendige Anpassung der Klassen ist durch das Dekorieren der Methoden überflüssig geworden. Wenn Sie mehr über Closures und Decoraters erfahren möchten, dann empfehle ich Ihnen den Link zu nutzen und das PDF-Dokument herunterzuladen. Dort habe ich in mehreren Schritten den Sachverhalt beschrieben. Einfache Beispiele verdeutlichen den Nutzen der Programmierung, die hinter den beiden Begriffen steckt.

Hier folgt jetzt das gesamte Programmlisting:

Nein, das lasse ich lieber bleiben, denn ich meine, es ist viel besser, wenn Sie die ca. 13 Seiten des Programmtextes herunterladen und in einem eigenen Fenster, am besten in Thonny, darstellen. Dann können sie Blog und Programm parallel zueinander durchgehen und die bunte Darstellung in Thonny verbessert den Überblick.

Ich hoffe, der vorliegende Beitrag liefert nicht nur Anregungen und Know-how zum Thema Peltierelemente. Ich würde mich freuen, wenn Sie darin auch Anregungen für andere Anwendungen finden.

In der nächsten Episode verwenden wir einen ESP32 zusammen mit einem TFT-Farbdisplay mit Touchscreen (320x240 Pixel), um damit die Cooler-Batterie zu steuern. Einige Programmiertechniken aus dieser Folge, kommen auch dort wieder zum Einsatz. Bis dahin schon mal viel Spaß beim Bauen, Programmieren und natürlich Kühlen.