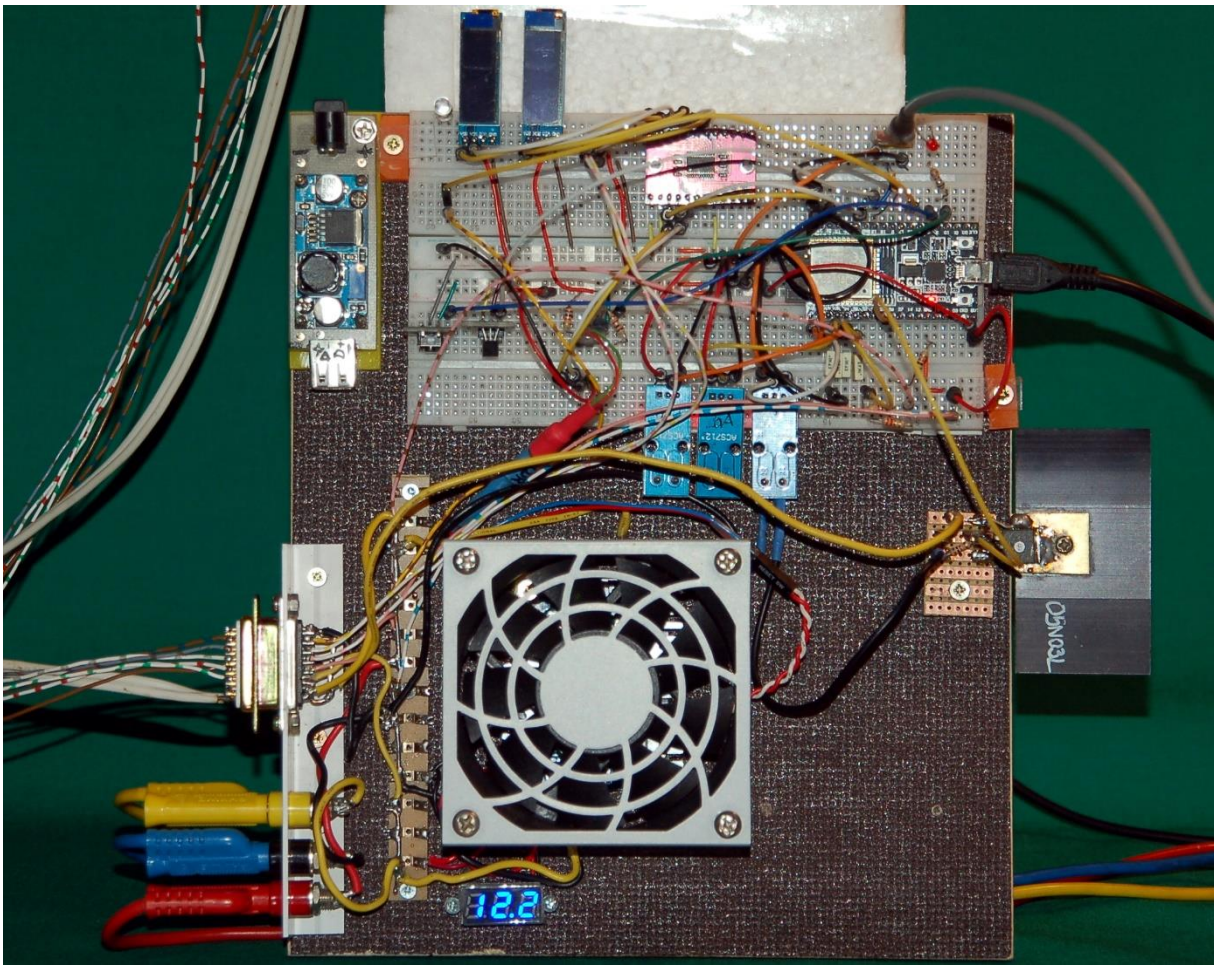


Abbildung 1: Testaufbau mit einer angeschlossenen Einheitf

This episode is available in German and English PDF version.

In the last episode, a cool box was built with the help of a Peltier element. We extended the buck converter of the power supply from the first episode so that the output voltage could be controlled by an ESP32. We also found a formula with which the calculation of the output voltage could be calculated depending on the duty cycle of the PWM signal. Today we are expanding the whole thing to up to three such units and developing a control program. You will learn how similar properties of different objects can be managed by a method using lists and how to change methods of a class without interfering with the class definition - keyword: decorator. I will tell you in a moment what you can expect in detail, but first of all welcome to the fourth part of the series with the title

Peltier elements - The ESP32 controls the do-it-yourself refrigerator

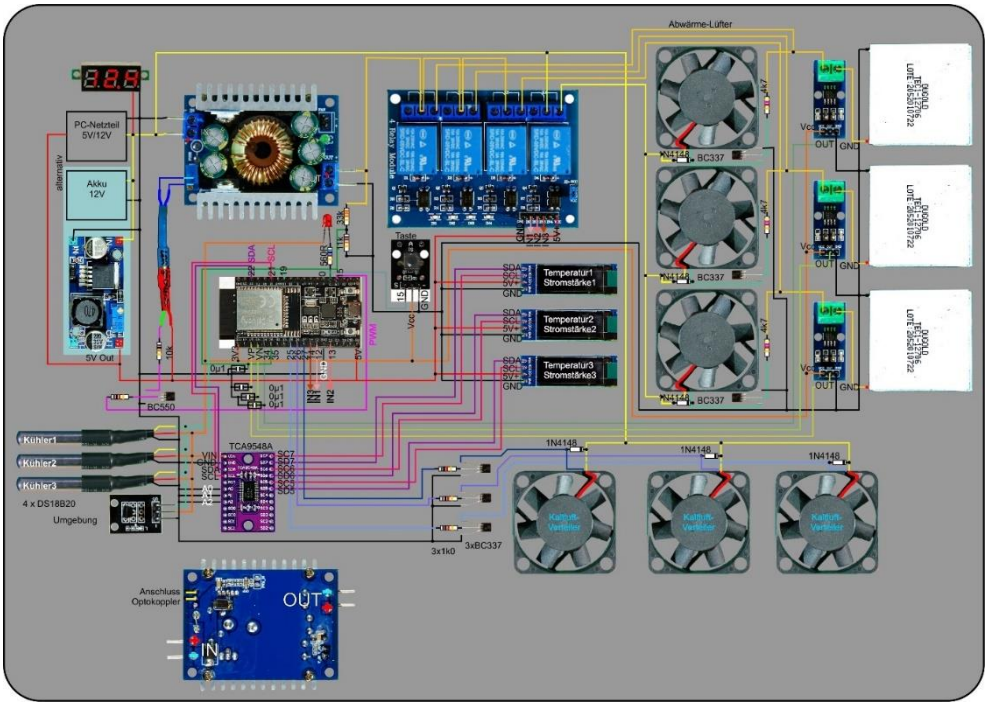
An ESP32 takes over the monitoring of the temperatures, the control of the current strength in the thermoelectric converters and the display of the most important data on each one OLED display on the respective unit. Of course, currents up to approx. 4 A have to be measured. A special kind of challenge is to control up to three OLED displays, all of which have the same I2C device address. The motors of the waste

heat fans should start automatically when the voltage is switched on at the Peltier elements and the fans in the box should be able to be switched individually by the ESP32. Here, too, the characteristics of the motor with 12 V and 80 mA go well beyond the capacity of a GPIO connection. So that the regulated voltage of the buck converter, switched with as little loss as possible, can reach the Peltier elements, I had provided a relay for each branch. "Had" because I was forced to give up the idea of relays in the course of development, which is why I'll tell you below in the hardware chapter.

In terms of programming, it is also important to operate the extensive hardware as effectively as possible, with RAM memory we are probably close to the limit. The data of the cooler units are defined using lists and controlled and managed using indices. We benefit from the fact that all possible objects can be accommodated in a list under MicroPython. The Arduino IDE only allows simple variable types in an array. Some of the functions are written for positive (HIGH level triggered) and negative logic (LOW level triggered) and are therefore very adaptable. This makes it easier to transfer them to other applications. For tests, the PC keyboard is first used to send commands to the controller. In the next episode, we will either develop and use an app on the Android smartphone or a program in connection with the AZ-Touch wall box with TFT display (320x240 pixels).

Hardware

For better orientation, I am already presenting the circuit diagram at this point. You can then easily see the various assemblies. It is possible to realize the project with one, two or all three units. Accordingly, the number of parts required is selected from the list below, which is designed for three units.



Yes, that would have been the wiring diagram if it hadn't been for the relay thing.

Abbildung 2: Thermobox_Schaltung 1

A relay is an electromagnetically operated switch, and the magnetic field when the relay coil is switched on does not stop at the plastic housing of the relay, nor does the residual magnetism after it is switched off.

"What's the problem with that?" You mean? Well, we want to measure the currents in the branches and that is to be done with the ACS712 sensors. But they work with Hall sensors, not with voltage measurement on a measuring resistor. It is precisely these Hall sensors that do not only detect the magnetic field of the current to be measured, but also that of the relays and do so very dominantly even at a distance of several tens of centimeters.

Instead of the relays, three MOSFET driver stages have now been put into service. Not quite as effective as a relay in terms of switching properties, the MOSFETs have the advantage that there is no contact weart. A copy of the circuit diagram in DIN A4 is available for download

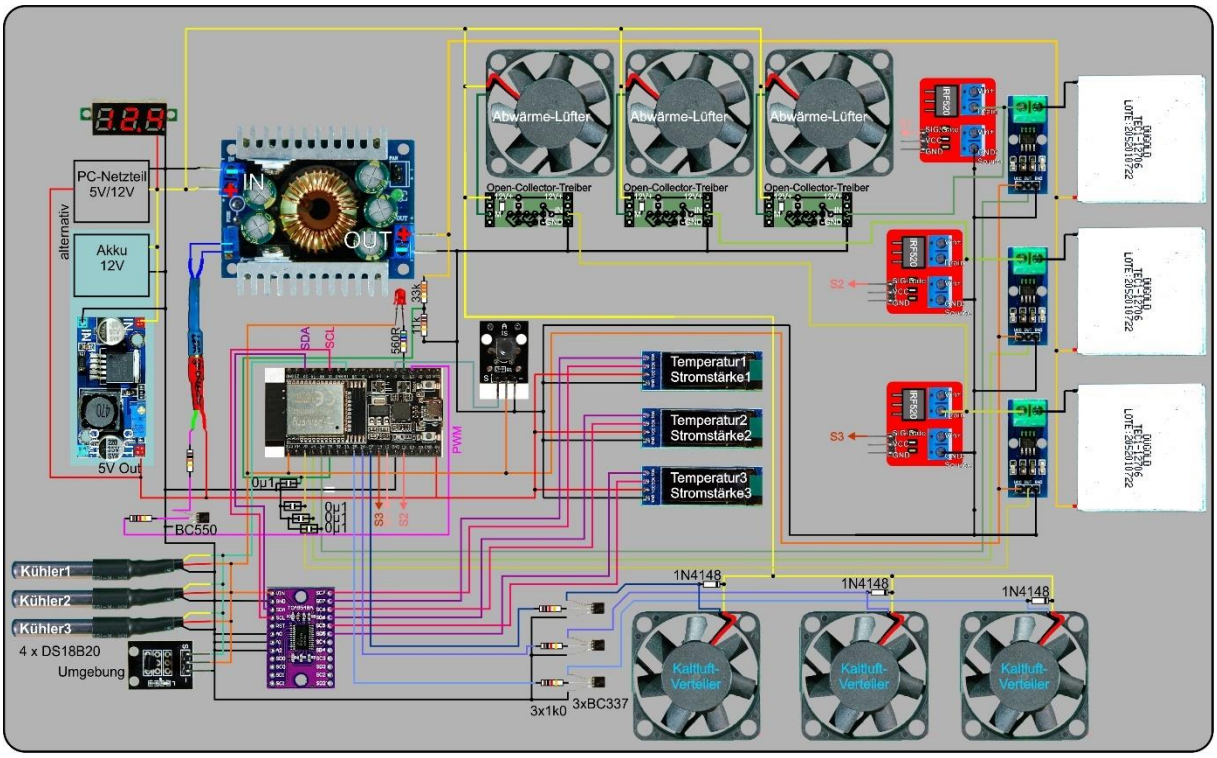


Abbildung 3: Thermobox_Schaltung mit MOSFET-Stufen

Here comes the list of parts. You can set up the project with just one cooler unit, then you do not need an I2C multiplexer, but you have to make major changes to the program in some places. The I2C multiplexer is required for two or more units in order to control the OLED displays separately. As already mentioned above, the modules have a permanently assigned device address and can therefore not be operated in parallel on an I2C bus. This project shows how such problems can be solved in a sophisticated way with regard to hardware, but also in terms of programming, without having to intervene in existing modules (program libraries). I have never encountered such tricks with the Arduino IDE - MicroPython can come up with them.

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

The structure of the thermal head and the cooling container has already been described in detail in Part 3, as well as the manufacture of the optocoupler as a supplement for the 8A buck converter, which makes the controller remotely controllable. The 12A converter can be cuffed in a similar way. This module is shown from above in the circuit diagram. The connection of the LDR in the optocoupler is marked yellow on the underside of the circuit board.

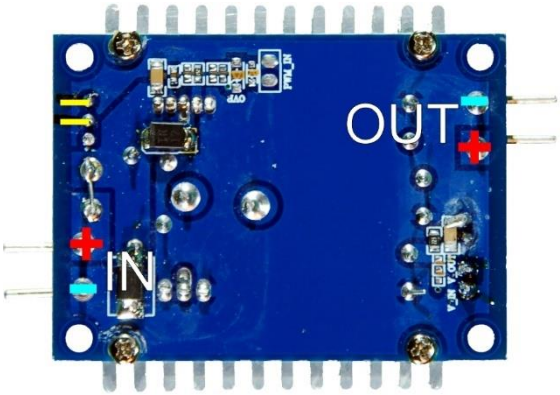


Abbildung 4: 12-Ampere-Regler_Unterseite

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

The voltmeter, at the top left, shows the voltage on the battery or PC power supply. A switch that is not shown in the circuit diagram should be provided for both energy suppliers. In the case of the battery, the switch must withstand a maximum of 12A. A small switch is sufficient for the power supply unit, which connects connection 16 (PS-ON) to a ground connection on the plug. 1.3mm post pins can be used as plug pins (see Figures 5 and 6).

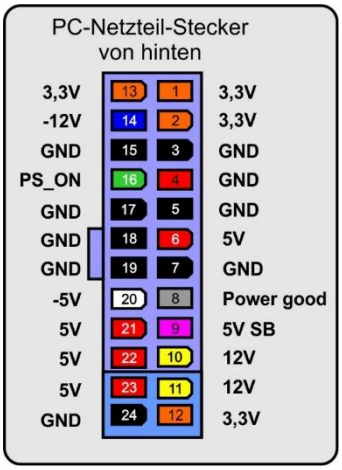


Abbildung 5:Netzteilstecker 24-polig von hinten

Abbildung 6:Netzteilstecker halb von vorne

The PC power supply unit must of course be able to deliver up to 12A at the 12V connection, that is, the connection must have an output power of 144W or more so that three cooling units can be connected. The PC power supply also has the great advantage that it also has a 5V output. In the case of battery operation, a separate controller must be used for this. If we use the 5V SB line (pin 9) of the power supply unit for the 5V supply, the ESP32 can enable the 12V supply itself via one of the still free GPIO connections by setting connection 16 to GND potential pulls. The power supply then remains firmly connected to the 230V network. In the final version, the command to switch on can alternatively also be given via the radio link.

The regulated voltage from the buck converter is fed to the Peltier elements at the red connection. From the black connection it goes back to the drain connection of the MOSFET transistor and from its source via the current measuring module ACS712 05A to GND. An inverting transistor switching stage on the warm side ensures that whenever current flows through the Peltier element, the fan on the warm side also starts up automatically. This is what the circuit for the breadboard and the associated circuit diagram look like.

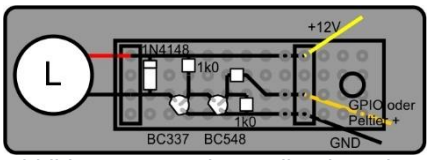


Abbildung 7: Transistortreiber invertierend

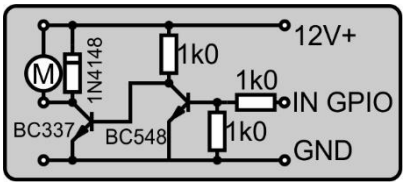


Abbildung 8: Transistortreiber invertierend_Schaltung

The other three simple switching stages activate the fan motors on the cold side using the signal from the ESP32. Here, too, the diode serves to protect the transistor. The ring on the diode must be on the positive voltage supply side. Reverse polarity kills the diode and the transistor.

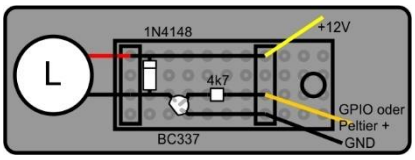


Abbildung 9: Transistortreiber

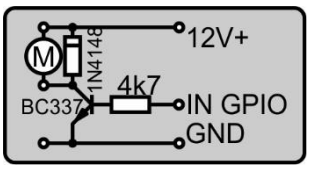


Abbildung 10: Transistortreiber_Schaltung

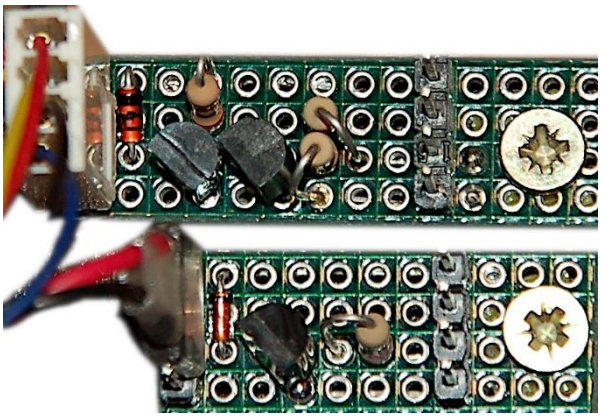


Abbildung 11: Transistor-Schaltstufen

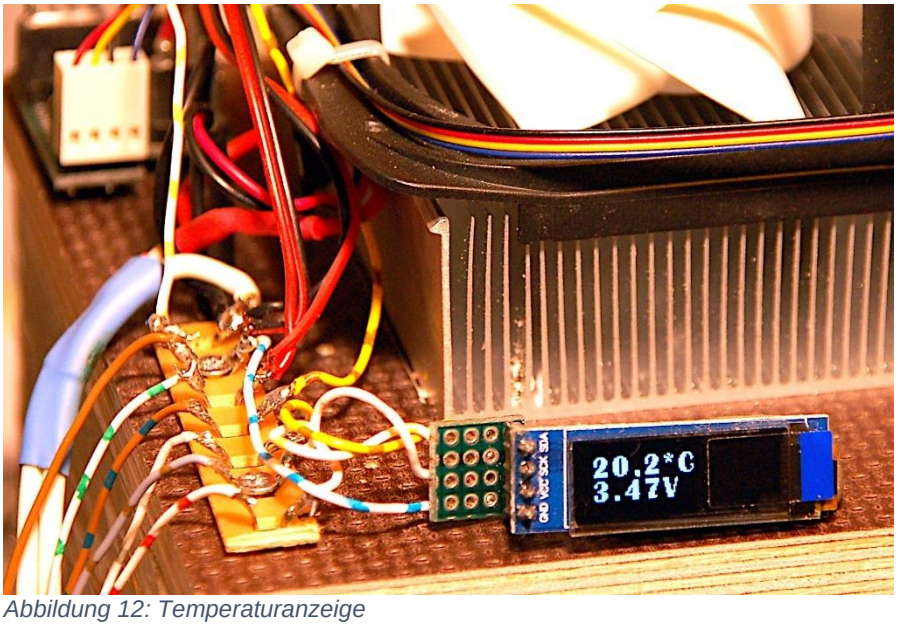


Abbildung 12: Temperaturanzeige

The OLED displays are attached to the cooler and provide information about temperature, voltage or amperage. The problem that all displays have a device address that cannot be changed is solved by using a PCA9548A. This I2C multiplexer puts the signal from the ESP32 through to the outputs that have a 1 bit in the configuration register of the module. Before outputting to an OLED module, the channel must first be switched, then control takes place as if the OLED were the only one in the cosmos. So that this can be done transparently and without changing the methods of the OLED and CharSet classes, the corresponding methods are "disguised"; in MicroPython this process is called decorating. There are hints on this at the very end before the program listing.

An enlarged font is used for display. You can clone your own fonts from the Windows font folder with the font2py.rar package. The batch file makecharset.bat contained in the package takes care of this in one step with the specification of the 4 parameters font, size, chars and source path. The string chars specifies the characters from the font character set that you want to clone. How this works in detail can be found in the PDF document Erstellen_von_großen_oder_eigenen Zeichensaetzen.pdf. A quick guide with sources can be found here. Alternatively, you can download the font geometer_18.py for now

The three temperature sensors with the cables are each led through the lid into the cool box and are all connected to the same one-wire bus together with the sensor for the ambient temperature.

A LOW-active button is used for an orderly withdrawal. It can be pressed at any time to cancel the program. Orderly means that before the "off" is tidied up. The power supply to the Peltier elements is interrupted, the displays are switched off.

With that we have arrived at the ESP32 itself. The display LED on GPIO2, the PWM output on GPIO15, the measurement of the output voltage on the buck converter via analog input GPIO35 and the inputs for current measurement VN, VP and GPIO34 have not yet been mentioned. To reduce the noise, each input is blocked with a capacitor of 0µ1 against GND, and the mean value of 100 individual measurements is used as standard for the measurement of the analog signals. Nevertheless, despite the correction formulas for the ADC characteristic, there is a measurement inaccuracy of up to 10%.

Software

For flashing and programming the ESP32:
Thonny oder
µPyCraft
packetsender for testing the ESP32 as a TCP/UDP server
font2py.rar

Used Firmware:

MicropythonFirmware
Please choose a stable version

MicroPython-Programs:

MicroPython-Programs:

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

MicroPython - Language - Modules and Programs

You can find detailed instructions for installing Thonny here. There is also a description of how the Micropython firmware is burned onto the ESP chip.

MicroPython is an interpreter language. The main difference to the Arduino IDE, where you always and exclusively flash entire programs, is that you only have to flash the MicroPython firmware once at the beginning on the ESP32 before the controller understands MicroPython instructions. You can use Thonny, µPyCraft or esptool.py for this. I have described the process for Thonny here.

As soon as the firmware is flashed, you can have a casual conversation with your controller, test individual commands and immediately see the answer without first having to compile and transfer an entire program. This is exactly what bothers me about the Arduino IDE. You simply save an enormous amount of time if you can do simple tests of the syntax and hardware through to trying out and refining functions and entire program parts via the command line before you knit a program out of it. For this purpose I also like to create small test programs over and over again. As a kind of macro, they combine recurring commands. From such program fragments, entire applications can develop.

Autostart

If the program is to start autonomously when the controller is switched on, copy the program text into a newly created blank file. Save this file under boot.py in the workspace and upload it to the ESP chip. The program starts automatically the next time it is reset or switched on.

Testing programs

If the program is to start autonomously when the controller is switched on, copy the program text into a newly created blank file. Save this file under boot.py in the workspace and upload it to the ESP chip. The program starts automatically the next time it is reset or switched on.

In between, Arduino IDE again?

If you want to use the controller together with the Arduino IDE again later, simply flash the program in the usual way. However, the ESP32 / ESP8266 then forgot that it ever spoke MicroPython. Conversely, every Espressif chip that contains a compiled program from the Arduino IDE or the AT firmware or LUA or ... can easily be provided with the MicroPython firmware. The process is always as described here.

The operating program

The program for the ESP32 already contains all the sequences that are used both for initial tests and are also intended for radio operation. Various functions read out sensors and set the voltage, the current strength or the temperature or switch the Peltier elements or the fans in the cold room.

All parts that are duplicated in the case of several units are managed by lists. This includes current measurement, switching levels, OLED displays, thermal sensors and the cold air vortex. The assignment of the GPIO pins is done in for loops. Corresponding functions receive, in addition to other possible parameters, the index of the unit to be operated. As an example, let's take a look at the definition of the MOSFET control pins. The designation "relais" or "rel" is reminiscent of the original approach with the 4-way relay unit. The MOSFET stages are, in contrast to the relay stages, HIGH-triggered.

```
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)
```

We enter the pin numbers in a tuple. In contrast to lists, tuples have the advantage that processing is faster. The disadvantage compared to lists is that the latter provide more versatile methods for processing and that list contents are mutable, which is not the case for tuples and strings (immutable). Elements of lists can change the value afterwards, elements of tuples or characters in strings cannot. But now back to the definition of the relay pins.

The indices are assigned to the variables rel0 etc. The definition of some variables promotes readability later in the program. Then we create a list, which is then filled in the for loop with the pin objects for switching the MOSFETs. Because the MOSFETs are triggered at HIGH level, a 1 on the GPIO pin switches each transistor on and a 0 off.

The relay () function belongs to the MOSFETs. This takes the index n and the desired status state as parameters as well as the flag trigLevel, which is optionally preassigned the value highTriggered (= 1).

```
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
```

The existing status is queried and the target status is assigned to the local variable s. The if construct converts the logical level into the electrical level if the LOW level is triggered, i.e. switched on by a 0 at the ESP32 output. In this case, the previously read level is also inverted so that the return value is logically correct again. Each of the following calls therefore switches on the corresponding switching stage:

3 HIGH level triggered transistor stages:

```
    for n in range (3):
        relay (n, on)
```

LOW level triggered relay stage:

```
>>> relay (0, on, low triggered)
```

In the first case, the default value highTriggered applies. In the second case, the trigger behavior must be specified so that the relay switches correctly.

The getRelaisState () function queries the switching status of a switching stage. The output is again adapted to the trigger property.

```
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
```

The basic function readADC () serves functions for current and voltage measurement with the raw counter values. It takes the index and an optional number of repeated measurements to filter and calm the result, which is the mean value of the individual measurements, 100 individual measurements are standard.

At 0.0A at the output, the current sensors deliver a voltage equal to half the operating voltage. The getQuiescentCurrentRaw () function determines the count values after which the corresponding line was safely interrupted by the MOSFET. After a pause of 0.2 seconds readADC () is called and then the transistor is set to the previous switching state. The function stores the zero-ampere count values in list I0 []. The originally used ACS712 20A delivered too small and very unsafe values at the operating voltage of 3.3V, because the sensors normally work on 5V. By exchanging with 5A types, the accuracy could be increased, even if the calibration factors, voltage to current strength, from the data sheet for the voltage 3.3V do not apply. A calibration factor had to be found by using a DAM (digital ampere meter). Unfortunately, the analog inputs of the ESP32 cannot tolerate voltages greater than 3V, which is why a 5V voltage supply for the ACS712 was not used as a precaution.

readAmps () uses the raw values. After the validity of the index has been checked, the readADC () call brings the raw count value. The I0 value of the channel is subtracted from this and the difference is converted into a current value. The value mVpA (millivolts per ampere) is used for this. The values for this factor also depend on the specimen and for this reason we have to convert the simple variable mVpA into a list of the module-specific values. This has already happened in the listing. Again, lists allow a size to be processed by only a single method.

```
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
```

If the specified index was not permitted, the validity of the current value can be checked using the return value.

The program should only control connected units. We check the presence by briefly switching on the corresponding stage and checking whether a current is flowing which corresponds at least to the minimum voltage applied to a Peltier element. This is done by the findCoolers () function, to which the trigger property of the switching stage must be passed. We call the setVoltage () function with the desired voltage value of 4V and get the duty cycle of the previous voltage value back, which we remember. Switching on each stage provides its previous switching status, which we also note. If the detected current value is greater than 0.3A, the unit is recorded as True in the coolerPressent list. Then the switching state of the transistor and finally the previous voltage are restored.

In the previous blog episode, we used a formula to show the relationship between the clock ratio D of the PWM signal and the resulting voltage Ua at the output of the buck converter.

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

Abbildung 13: Gleichung der Trendlinie:

The parameters a = 17.8894 and b = 0.39279 depend on the components of the SB optocoupler and their arrangement. That is why they have to be determined by a series of measurements. How to do this is described in episode 3.

However, we now need exactly the opposite assignment Ua -> D. We obtain the relationship by transforming the equation and solving it for D

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

Abbildung 14: Duty Cycle aus Spannung

or in general:

$$D = \left(\frac{Ua}{a}\right)^{\frac{1}{-b}}$$

Abbildung 15: Duty Cycle aus Spannung_allgemein

This formula is used by the volt2duty () function. Permissible voltage values are converted and at the same time the percentage is further adapted to the range 0 to 1023. The service of this function is in turn used by the functions setCurrent () and setVoltage (). setCurrent () first tries to set the voltage as it would be necessary for the desired current strength according to the ohmic resistance formula. If this does not succeed because the calculated voltage is not in the range Umin to 16V, the minimum or maximum possible voltage is set and the function is exited. The function is exited even if the actual current intensity does not deviate from the nominal current intensity by more than 100mA. If a setting was possible, but the difference between the actual and target value was too great, the function also attempts a successive approximation in ten steps of 0.5V each. The results of the voltage measurement deviate considerably in the lower and upper range from the values measured with the DVM (digital volt meter). In order to eliminate this evil, we have to start a series of measurements in which the target voltage from the DVM and the voltage measured by the ESP32 are recorded. Libre Office then provides a power function with the graphical evaluation, of whose function term we are interested in the coefficients, which we transfer to our program in abbreviated form. Our demands on accuracy decide whether we choose a function of the second or third order. The worksheet is [available for download]

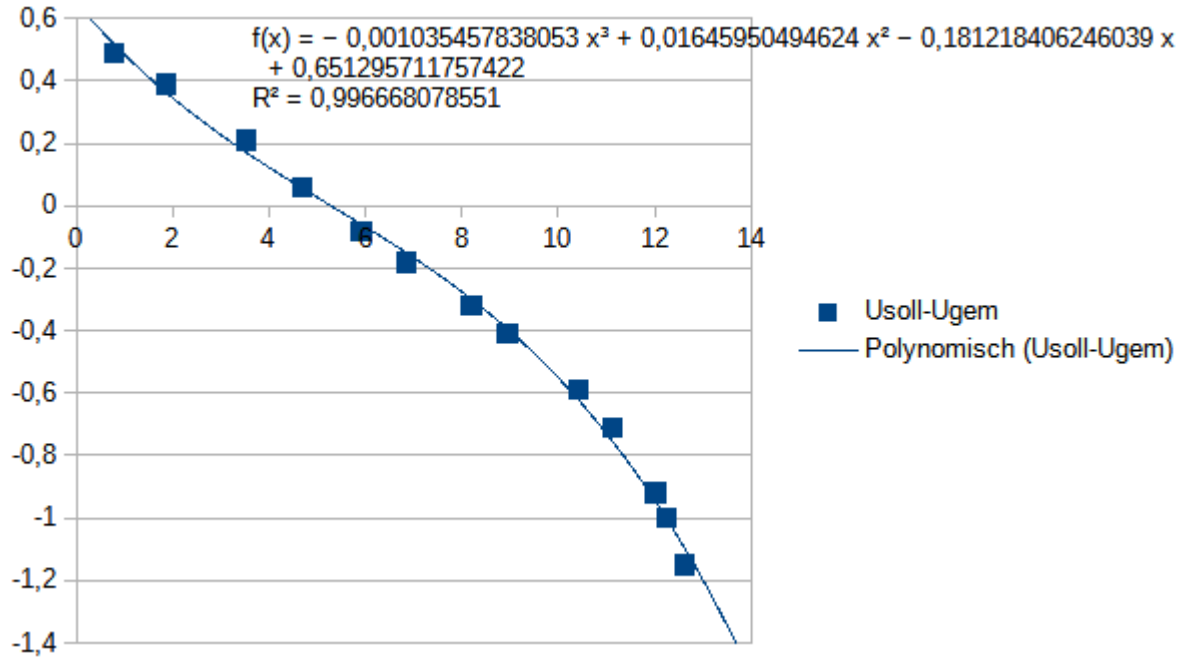


Abbildung 16: Grafik zur Fehlerkorrektur bei der Spannungsmessung

In the program text it looks like this. With the factor 3793 instead of 3600, I first tried to trim the result, but it was not enough. UFaktor takes into account the voltage divider 33kΩ: 11kΩ at the analog input GPIO35, which reduces the 12V to ESP32-compatible 3.0V.

```
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
```

The setTemp () and holdTemp () functions are used to set and monitor the temperature in the cooler box. The latter tries to set and maintain the target temperature by calling it cyclically in the main loop. Of course, this only works in server mode. In the test phase, the loop is blocked by the input command.

The cold air distributors are switched using the setFanState () function. Because these are simple transistor stages with positive logic, the trigger property was not added from the outset.

getTemp () is the function that determines the temperature in the cooler unit n and returns it with one decimal place. An invalid temperature is returned as 9999 in the event of an error. If a non-existent cooler unit is addressed, -9999 is returned.

The displayValues () function is responsible for displaying values in the OLED displays. It checks whether the current intensity control is switched on (Iflag is not equal to 0) and outputs the current intensity or the value of the set voltage depending on the temperature.

As the most extensive function, parse () takes on the task of decoding incoming commands and initiating the corresponding actions. Feedback on the action or error messages are also generated. The latter helps both the user and the programmer because the point of failure can be easily identified.

All commands have a similar structure. A letter stands for the type of job, followed by a colon. Except for setting the voltage, where the voltage value immediately follows, the number of the cooler unit comes after the colon. Except for the status query with "S", there is another colon and then a numerical value. There are one-button commands for maintenance and diagnosis.

```
    # 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
```

The respective department proceeds according to this scheme after a valid job ID has been found. The methods find () and split () of the examined string object are used. The area of validity is checked for the number of the cooler unit and the switching status. The conversion from a string to a floating point number is protected by try and except in order to prevent program crashes. The feedback takes place in the tuple (art, act, value)

Command: u: 8.47
from 999.999.999.999:99999
Content = u: 8.47
U: Current voltage: 8.43
: error free: 0

art contains the job ID, act the message and value the error number. The 0 stands for OK. The error number is at the end of the string because it can be easily found and isolated by the requesting process. A sequence of the following type provides the index in a list of plain text messages or function names that are to be displayed or executed in the event of an error.

The mode for entering commands is selected at the beginning of the declaration part of the program. A local radio network (WLAN with access point), the ESP32 as an access point or, if both were deselected with False, the PC keyboard are available. Of course, the functions including parse () can also be called manually via REPL.

```
# 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"
```

The connection to the WLAN access point then looks like this.

```
# *******************************************************
# 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='')
```

If the connection worked, a separate, fixed IP address that does not yet exist in the WLAN is assigned because the ESP32 is running as a UDP server. An IP assigned by the WLAN access point via DHCP can have different values, which is not a good idea for a server.

If the connection is to be made via radio, a socket is now set up with a timeout of 2 seconds. The timeout ensures that the receiving loop can do other things than just listening to incoming requests.

If no radio communication is desired, an input instruction waits for commands, which are then also passed to the parser. The socket is set to 999.999.999.999:99999 in order to distinguish the feedback from a radio request. Entering "e" ends the program properly. This is also possible by pressing the button or the flash button on the ESP32.

We can test the radio traffic in the WLAN very well with the help of the packetsender tool. The commands shown above can be entered and sent there using the keyboard. The responses from the ESP32 are also output. We'll take a closer look at this in the next episode.

It is worth taking a look at a few very interesting lines at the beginning of the program.

```
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)
```

After importing the character set britannic_18 and the CharSet class, we define a list that translates the number of the cooler unit (0..2) into the channel numbers of the I2C multiplexer (5..7). Then the three OLED objects are generated and initialized. The for loop switches the respective channel.

The declaration of an instance c of the CharSet class is followed by the declaration of the switchToChannel function, which takes a function f as an argument and returns another function g, which is defined in switchToChannel. switchToChannel is a so-called decorator, which is used to modify the functions transferred to it. This happens without the functions themselves being changed. Additional instructions are only inserted before or after the transferred function is called. With the help of the last 4 lines shown, we are able to adapt two methods of the CharSet and OLED classes to our needs. g represents the modified function f. g sets the desired channel and then calls the transferred function f with the original parameters. The new function is now simply given the name of the original function.

You will remember that in order to control several OLED displays, the correct channel must first be set on the TCA9548. This is exactly what our decorator does by adding the channel number n in front of the previous call in the parameter list - g (dpn, * args, ** kwargs).

D.clearAll () becomes d.clearAll (n),
d.writeAt ("test", 0,0) becomes d.writeAt (n, "test, 0,0) and off
d.putSymbol (c, xpos = 0, ypos = 0, show = True) becomes d.putSymbol (n, c, xpos = 0, ypos = 0, show = True) ...

An otherwise necessary adaptation of the classes has become superfluous due to the decoration of the methods. If you would like to learn more about closures and decoraters, I recommend that you use the link and download the PDF document. There I described the facts in several steps. Simple examples illustrate the benefits of the programming behind the two terms.

Here is the entire program listing:

No, I prefer not to insert it here, because I think it is much better if you download the approx. 13 pages of the program text and display them in a separate window, ideally in Thonny. Then you can go through the blog and the program in parallel and the colorful display in Thonny improves the overview.

I hope this article not only provides suggestions and know-how on the subject of Peltier elements. I would be delighted if you could find suggestions and inspirations for other applications in it.

In the next episode we will use an ESP32 together with a TFT color display with touchscreen (320x240 pixels) to control the cooler battery. Some programming techniques from this episode are also used there again. Until then, have fun building, programming and of course cooling.