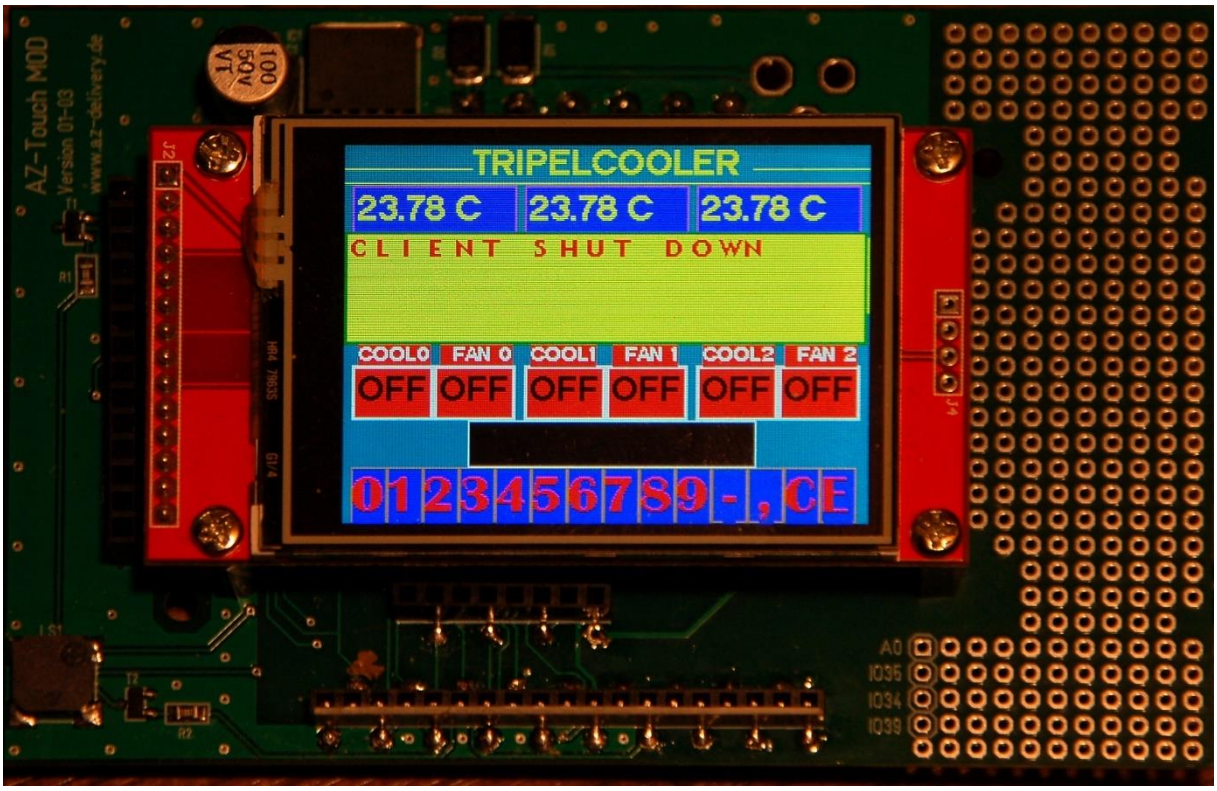

Abbildung 1: Display im Wandmodul

Diesen Beitrag gibt es auch als PDF-Dokument in Deutsch und Englisch.

A refrigerator without temperature setting and control is only half a refrigerator. Therefore, after the previous development steps of blog parts 1, 2, 3 and 4, part 5 introduces a control unit based on UDP radio transmission. This is necessary because the cooler does not have its own buttons or controls. Today it is not controlled via the mobile phone, but with an ESP32 in cooperation with a TFT color display with touchscreen via UDP. In addition to the basic application of a larger TFT color display with 320 x 240 pixels and the use of the associated resistive touchpad, the article shows the programming of a touch ten-key keyboard, the use of touch buttons, input fields and a message window, which in turn can be touched Is divided into subsections. In addition, radio transmission via UDP will prove to be very practical, especially when it comes to feedback from the cooling unit. So welcome to the 5th part of the series with the title

Peltierelements –
We're talking to our refrigerator

Hardware

The hardware for this episode consists of two parts.

1	AZ-Touch MOD Wandgehäuseset mit 2,4 Zoll Touchscreen darin enthalten sind das Gehäuse, die Leiterplatte incl. Buchsenleisten und das Display sowie zugehöriges Montagematerial
1	ESP32 Dev Kit C V4 oder ESP32 NodeMCU Module Achtung: Das ESP32 Lolin LOLIN32 ist nicht geeignet, weil es einen anderen Footprint besitzt. Stiftverteilung und Belegung passen also nicht zum Basisboard.

Again, be warned about the execution of the ESP32 board. Because nothing else was available, I had ordered ESP32 Lolin LOLIN32 and, to my regret, had to discover that the LOLIN unfortunately cannot be used together with the base board of the display. The wiring is fixed on the printed circuit board. Therefore only the two mentioned controller boards can be used. An ESP8266 is out of the question for this application because the ESP32 is already working at the RAM memory limit.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
packetsender zum Testen des ESP32/ESP8266 als UDP-Server

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

Die MicroPython-Programme zum Projekt:

thermobox.py Betriebssoftware der Coolereinheiten
thermoXmit.py Betriebssoftware der Steuereinheit
ili934xnew.py Anzeigetreiber (MIT-Lizenz)
xpt2046_syn.py Touchpadtreiber (MIT-Lizenz)
calibrate.py Kalibriert das Touchpad auf Display-Koordinaten

Zeichensätze:
tt14.py, tt24.py, tt32.py, britannic.py, geometer16.py, glcdfont.py
font_to_py.py zum Herstellen von Displayzeichensätzen

calibrationdata.txt die Datei mit den Kalibrierdaten des Touchpads

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.

Extension of the drivers

For more convenient work, I have adapted and expanded the driver modules. In particular, the ILI9341 class in the ili934xnew module was expanded to include the FRAME class. This is where methods are located that make it possible to create structures that are reminiscent of the windows of Winzig-Weich. I named them frames or frames. This makes it possible to divide the screen into smaller areas that can be managed by themselves.

In addition to a 5V buck converter with the LM2576, the base board of the display also contains a passive buzzer, which is best controlled via a PWM output. The class ILI9341 therefore received an additional, optional parameter for the constructor, to which a PWM object for the GPIO pin 21 can be transferred. If that doesn't happen, then he creates one for himself. The beep () method takes a frequency value and an optional pulse duration.

```
def __init__(self, spi, cs, dc, rst, w, h, r,buzz=None)
```

The class for operating the resistive touchpad XPT2046 has been expanded to include the saveCalibration () and loadCalibration () methods. A calibration must be carried out so that a contact point can be assigned to a screen position. This is done with the help of the calibrate.py program. This program also had to be adapted to the use of the screen in landscape format. The calibration process results in 8 numerical values that are either integrated directly in the user program or, as here, stored in a file (calibrationdata.txt) on the ESP32. This enables us to make changes later without having to intervene in the user program. The two new methods allow this to be done easily. The constructor of a touch object tries to read the file. If he doesn't find it, he takes the module's standard values. Alternatively, when instantiating a touch object, a tuple with the calibration data can be transferred as an optional parameter.

The character sets

TTF character sets are converted into MicroPython modules for text output. This is done by the font_to_py.py tool from Peter Hinch (MIT license), of which there are two versions. We had already used this program to create reduced character sets for a black and white OLED display. If you would like to create fonts for MicroPython yourself, please download the tool from here (not from GitHub!) Into any directory. Then copy the desired TTF files there as well. Now navigate to this path in Explorer and right-click on the directory name while holding down the Shift key. In the context menu click on Open PowerShell window here.

Suppose the path is F: \ _ font2py and the TTF file is britannic.ttf. Then the call in a Powershell window must look like this:

```
.\font_to_py F:\_font2py\britannic.ttf 18 britannic18.py
```

Ausgabe:

```
PS F:\_font2py> .\font_to_py F:\_font2py\britannic.ttf 18
britannic18.py
Writing Python font file.
Height set in 2 passes. Actual height 18 pixels.
```

```
Max character width 16 pixels.
britannic18.py written successfully.
PS F:\_font2py>
```

If the execution was not successful, we will find the britannic18.py file in the same directory. It is imported into the user program as the britannic18 module and selected for the next output using the set_font (britannic18) method.

The character set module brings some functions for the character selection and the determination of the character width. ILI9341 needs the latter for the display. The versions of the font_to_py.py tool differ here. If the get_width (s) function is missing in the module, the ILI9341 class aborts with an error message.

Die Schaltung

The circuit is fixed via the printed wiring on the base board. The circuit diagram is available from the manufacturer. A free e-book describes the structure and commissioning of the module. The supplied socket strips are all mounted on the side of the circuit board on which the parts of the step-down controller are also located. The ESP32 is inserted through the circuit board from the underside. This results in mirror-symmetrical assignments of the strips on the upper side. The spacer at the front right only serves to support the display. There is no hole and no space for the lower attachment, because the USB cable connector is located exactly at this point.

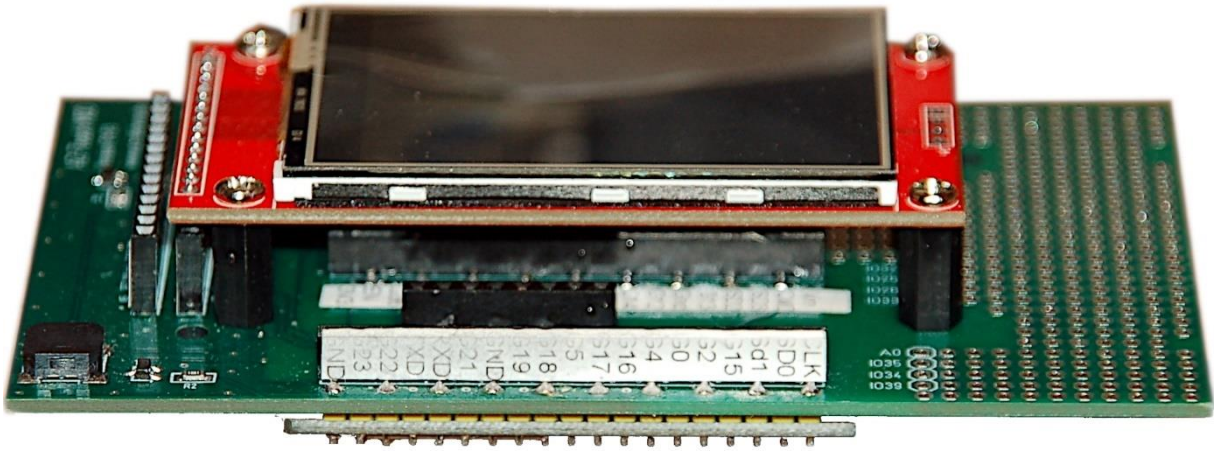

Abbildung 2: Oberseite

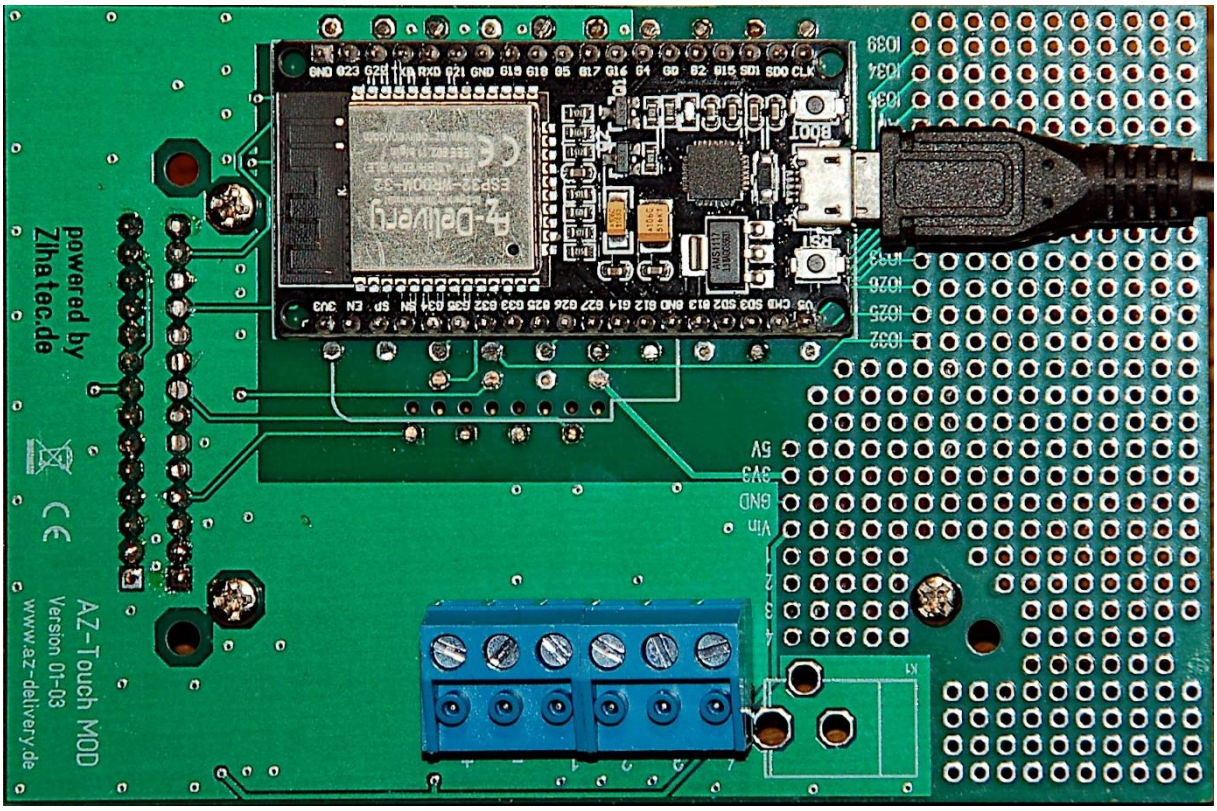

Abbildung 3: Unterseite

The Program

Without the driver and character set modules, it has a length of 490 lines and, in addition to the import of modules and the definition of the data structures, consists of three essential blocks:

- Establishing the radio link
- Parser and job sender (talker) to the server in the coolbox
- Listener for the feedback

The communication between the ESP32 and the display as well as the touchpad takes place via the SPI bus. Like the I2C bus, the SPI bus is a clocked synchronous connection. In contrast to the I2C bus, which only uses one data line and selects components by sending a device address, the SPI bus uses two data lines and a separate chip select line for each component. The blocks with which communication is to take place are selected via this line by placing LOW. Our display needs two selection lines, one for the display module ILI9341 and one for the touchpad XPT2046.

The generated SPI object, spi, works at 5MHz. This is a compromise between a reliable touchpad and a reasonably fast display that would work up to 32 MHz. The display and touchpad are on the same bus, but of course on different GPIOs for cs (chip select).

In order for anything to be visible on the display, we need another GPIO pin for the background lighting. Caution! For inexplicable reasons, the lighting is controlled via a

PNP transistor and is therefore LOW-active! To switch off the lighting after a certain time (timerDelay = 30000), the timer object, tirq = timer (0), can be activated at certain points. These locations are in several positions in the program. The constant autoAus = 1 always activates the feature.

The display object is called d and the touchpad object is called t.
To control the buzzer, we create the PWM object buzz.

We need a number of buttons of the same type. We create the necessary instances with the switch () function. It only takes position parameters, namely the x, y coordinates of the top left corner, width b and height h of the button. The pixel distance of the next button in the x-direction is offset. This is followed by the thickness of the border and the text to be written on. The final step is the font color, frame color and background color, all three of which are RGB values. For the display, the 24-bit color values in the form 0xRRGGBB must be converted into the 16-bit color space. This is done by the rgbTo565 () function, which I added to the ili934xnew module. As a result, the first 5 most valuable bits represent red, 6 bits follow for green and the remaining 5 bits represent the blue value.

```
def color565(r, g, b):
    return (r & 0xf8) << 8 | (g & 0xfc) << 3 | b >> 3

def rgbTo565(rgb):
    r=(rgb>>16)&0xff
    g=((rgb&0x00FF00)>>8)&0xff
    b=rgb&0x0000ff
    return color565(r,g,b)
```

```
def schalter(x,y,b,h,versatz,rand,text,textcolor,framecolor,\
             background):
    breite=b
    p=x
    switch=[0]*3
    f=d._font
    for i in range(3):
        d.set_font(tt24)
        switch[i]=FRAME(p+i*versatz,y,b,h,rand,0x000000,\
                        0xffffff,0xFF00ff,name=text+str(i))
        switch[i].show(d)
        switch[i].onoff(d,0)
        d.set_font(tt14)
        xt,yt=switch[i].textstart

switch[i].write(d,text+str(i),rgbTo565(0xffffff),x=xt,\
                        y=yt-rand-16)
    d.set_font(f)
    return switch
```

The return value of switch () is the generated list of frame objects. Each object knows its properties and can therefore be operated easily and clearly using the methods of the FRAME class without having to pass further parameters apart from the display object d.

The data () function creates label fields for displaying texts and values in a similar way.

```
def daten(x,y,b,h,versatz,rand,text,textcolor,framecolor,\
            background):
    breite=b
    p=x
    switch=[0]*3
    f=d._font
    for i in range(3):
        d.set_font(tt24)
        switch[i]=FRAME(p+i*versatz,y,b,h,rand,textcolor,\
                        framecolor,background,name=text+str(i))
        switch[i].show(d)
        switch[i].write(d,text,rgbTo565(textcolor))
    d.set_font(f)
    return switch
```

Keypads are created using the keypad () function. The transferred string contains the key labels to be displayed. A key corresponds to each character. keyPressed () determines the pressed key and getNumber () compiles a number from it. The key C deletes backwards, E takes over. The number can have up to 10 digits including the comma. The automatic hiding of the display by tirq.deinit () is switched off while the number is being entered.

```
def tastenfeld(x,y,h,rand,textcolor,framecolor,background,\
               beschriftung="0123456789-,CE"):
    breite=320//len(beschriftung)
    p=x
    n=0
    d.set_font(britannic)
    taste=[0]*len(beschriftung)
    for c in beschriftung:
        taste[n]=FRAME(p+n*breite,y,breite,35,rand,textcolor,\
                       framecolor, background)
        taste[n].show(d)
        taste[n].center(d,beschriftung[n],rgbTo565(textcolor))
        n+=1
    return taste
```

```
def keyPressed(x,y,keys):
    n=0
    for k in keys:
        if k.pressed(x,y): return n
        n+=1
    return None
```

```
def getNumber(keys,delay=2000):
    tirq.deinit()
    fs=d._font
    d.set_font(tt24)
```

```
    L=10
    number=[" "]*L
    f.show(d)
    ec=10
    def showNumber():
        s=""
        for c in number:
            s+=c
        f.show(d)
        d.set_pos(f.textstart[0],f.textstart[1])
        d.write(s)
    n=0
    while 1:
        c=t.get_touch(timeout=delay)
        if c is None:
            ec-=1
            print(ec)
            if ec < 0:
                d.set_font(fs)
                f.show(d)
                if autoAus:
                    tirq.init(period=timerDelay,\
                              mode=Timer.PERIODIC,\
                                callback=isrTirq)
                return None
        else:
            x=c[1]; y=c[0]
            k=keyPressed(x,y,keys)
            if k is not None and k in range(14):
                if k in range(10):
                    number[n]=str(k)
                    n=min(n+1,L-1)
                if k==10 and n==0:
                    number[n]="-"
                    n=min(n+1,L-1)
                if k==11 and n >=1:
                    number[n]="."
                    n=min(n+1,L-1)
                if k==12:
                    n-=1
                    n=max(n,0)
                    number[n]=" "
                if k==13 :
                    if n>=1:
                        if "." in number:
                            d.set_font(fs)
                            f.show(d)
                            if autoAus:
                                tirq.init(period=timerDelay, \
                                          mode=Timer.PERIODIC,
\
                                          callback=isrTirq)
```

```
                            return float("".join(number))
                        else:
                            d.set_font(fs)
                            f.show(d)
                            if autoAus:
                                tirq.init(period=timerDelay, \
                                          mode=Timer.PERIODIC,
\
                                          callback=isrTirq)
                            return int("".join(number))
                    else:
                        d.set_font(fs)
                        f.show(d)
                        if autoAus:
                            tirq.init(period=timerDelay, \
                                      mode=Timer.PERIODIC, \
                                      callback=isrTirq)
                        return None
                showNumber()
```

The isrTirq () function is the service routine of the timer interrupt, which switches off the background lighting.

get_stringsize () determines the length of a string s depending on the specified font.

print_centered () outputs the string s in such a way that its center is at position x, y.

The temperature of the cooler unit n is filtered out of the response string sent back from there by getTempIst ().

```
def getTempIst(resp,n):
    global Tist
    for i in range(3):
        p=0
        pos1=resp.find("T:",p)
        if pos1!=-1:
            pos2=resp.find("*",pos1)
            p=pos2
            Tist[i]=float(resp[pos1+3:pos2])
        else:
            Tist[i]=8888
    return Tist[n]
```

The function xmit () sends the string msg to the addressees specified in the s.sendto call after remnants have been removed from the receive buffer. One of the addressees should be the cooler unit. Additional goals can be specified for control purposes. This procedure is possible because UDP sends messages without establishing a secure connection as with TCP. This can be compared to an RS232 interface, where you don't have to worry about whether a remote station is actually listening in. Whoever has ears is hearing!

```
def xmit(msg):
```

```
    try:
        s.recvfrom(300)
    except:
        pass
    s.sendto(msg,target)
    s.sendto(msg,target2)
```

readMessage () reads messages up to 250 characters in length as a byte object from the receive buffer. The line feed and carriage return are removed, and the remainder is converted to a string of ASCII characters that is returned.

```
def readMessage():
    try:
        rec,addr=s.recvfrom(250)
        rec=(rec.strip("\n\r")).replace("\n"," ")
        return rec.decode()
    except:
        return None
```

showMessage () shows the string m in the message window. Color and font can be transferred as parameters. The message window is cleared after tout seconds, unless the last two lines of the function are commented out. The default character set is geometer16. It is used when the font parameter is not specified.

```
def showMessage(m,tout,color,font=geometer16):
    if m is not None:
        d.set_font(font)
        meldung.show(d)
        meldung.print(d,m,color)
        t.get_touch(timeout=tout)
        meldung.show(d)
```

holeWertAlsString() works in a similar way to getTempIst (), but is more flexible because the search string and terminator can also be passed. In addition, the return value is a string and not a number.

This is followed by the commands for dividing the screen by creating the various frame objects.

```
tirq=Timer(0)
timerDelay=30000
d.set_color(rgbTo565(0xFFFF00),rgbTo565(0x009999))
d.erase()
d.hLine(0,15,320)
d.hLine(0,24,320)
print_centered(width//2,0,"TRIPELCOOLER",tt24)
cool=schalter(5,height-100,50,35,105,2,"COOL",\
              0xFFFFFF,0xFFFFFF,0x0000ff)
fan=schalter(55,height-100,50,35,105,2,"  FAN ",\
             0xFFFFFF,0xFFFFFF,0x0000ff)
Temp=daten(5,26,100,30,105,1,"--.--*C",\
           0xFFFF00,0xFF00FF,0x0000CC)
```

```
Tist=[20]*3
Tsoll=[18.0]*3
f=FRAME(22*8//2-12,height-65,22*8,30,1,\
        0xffff00,0xFFFFFF,0x000000,name="f")
f.show(d)
taste=tastenfeld(5,height-35,35,1,0xff0000,0xCC9900,0x0033cc)
meldung=FRAME(0,55,319,72,2,0x000099,0x009900,0xffff00,name="m
sg")
meldung.show(d)
d.set_font(tt24)
ledOn()
```

The network connection

In terms of radio technology, the application is basically a client. The connection to an access point can be made via a WLAN router or directly to the optional access point in the cooler box. When the connection to the access point is established, the socket of the UDP client is instantiated. Of course, the IP data and port number must be adapted to the local conditions, as well as the access data for the access point.

The parser

The parser, in the form of the doJobs () function, takes the coordinates as a tuple that was determined by calling the t.get_touch () method in the main loop. By rotating the display by 270 ° (r = 3 in the constructor call of the display object d), x and y are swapped. This is corrected when the coordinates are unpacked. Then all frame objects are queried one after the other as to whether the coordinates of the contact point come from the surface of the object. This is what the FRAME method does pressed ().

```
def doJobs(c):
    if c is not None:
        ledOn()
        if autoAus:
            tirq.init(period=timerDelay, \
                      mode=Timer.PERIODIC, \
                      callback=isrTirq)
        y,x=c
        senden=""
        for i in range(3):
            if cool[i].pressed(x,y):
                d.set_font(tt24)
                cool[i].toggle(d)
                senden="c:"+str(i)+":"+str(cool[i].switch)
                # sende Cooler i an/aus
            if fan[i].pressed(x,y):
                d.set_font(tt24)
                fan[i].toggle(d)
                senden="f:"+str(i)+":"+str(fan[i].switch)
                # sende Fan i an/aus
```

```
        if senden:
            xmit(senden)
            senden=""

        if f.pressed(x,y):
            tirq.deinit()
            showMessage("CLIENT SHUT DOWN", 5000,\
                        rgbTo565(0xff0000))
            sys.exit()

        if meldung.pressed(x,y):
            for i in range(3):
                if Temp[i].x <= x <=Temp[i].x2:
                    xmit("s:"+str(i))

        for i in range(3):
            if Temp[i].pressed(x,y):
                xmit("g{}".format(i))
```

Because the switches are available as a list, they are queried in a for loop. If the program can assign the coordinates to a button, the previously empty string Send is filled with send data after the appearance of the button has been changed. Send now contains the command code that we tested using the keyboard in the previous blog sequence. If the string is not empty, it is sent to the cooler and emptied again.

A tap on the black frame in which the entered numbers appear ends the program. The server program in the cooler is not affected by this.

A tap on the message window calls up the status of a cooler unit. The parser decodes the associated area on the basis of a closer look at the x-value and thereby selects the appropriate unit whose data is sent back. We take the comparison x-values from the temperature input fields.

If a temperature window is tapped, a g-command is sent to the cooler, which causes the current temperature to be sent back.

The listener

While the parser takes on the sender role, the main loop plays the listener, which listens for incoming messages, i.e. the responses from the server. This separation has proven to be very useful because the times between the placing of the order and the receipt of the response cause not inconsiderable delays in the program flow. The while loop is optimized to run through as quickly as possible, which means that tip events can be recognized more quickly and incoming messages can be reacted to more quickly.

```
while 1:
    c=t.get_touch(initial=False,timeout=300)
    if c is not None:
        doJobs(c)
```

```
    try:
        rec,adr=s.recvfrom(150)
    except:
        rec=None
        pass
    if rec is not None:
        rec=rec.decode()
        rec=(rec.strip("\n\r")).replace("\n"," ")
        mf=geometer16
        showMessage(rec,3000,rgbTo565(0x006600),font=mf)
        if rec[0]=="G":
            i=int(rec[1])
            loctemp=getTempIst(rec,i)
            Tist[i]=loctemp
            Temp[i].show(d)
            d.set_font(tt24)

Temp[i].center(d,str(Tsoll[i])+"Cs",rgbTo565(0xffff00))
            c=t.get_touch(timeout=4000)
            if c is not None:
                y,x=c
                if taste[13].pressed(x,y):
                    z=getNumber(taste)
                    if z is not None:
                        Temp[i].show(d)

Temp[i].center(d,str(z)+"Cs",rgbTo565(0xffff00))
                        Tsoll[i]=z
                        xmit("t:{}:{}\n".format(i,z))
                        sleep(2)
            d.set_font(tt24)
            Temp[i].show(d)

Temp[i].center(d,str(Tist[i])+"Ci",rgbTo565(0xffff00))

        if rec[0]=="T":
            i=int(rec[1])
            h=holeWertAlsString(rec,"T:","*")
            Tist[i]=float(h)
            d.set_font(tt24)
            Temp[i].show(d)
            Temp[i].center(d,h+"Ci",rgbTo565(0xffff00))

    if cancel.value()==0:
        print("Mit Flashtaste abgebrochen")
        blinkLed.value(1)
        tirq.deinit()
        sys.exit()
```

The replies that come back from the server are not only used for display in the message frame, but also sometimes trigger new commands to the server. But let's just go ahead in order.

At the beginning there is the query of the touchpad. If it was tapped, the return value c of the method t.get_touch () is a tuple of the normalized coordinates. The program reacts to this by calling the parser to which the coordinate tuple is transferred. We already had what the parser does with it.

If there is no tip, we check whether there is a response from the server that needs to be processed. If the receive loop was left without a result with a timeout, then MicroPython throws an exception that needs to be caught, because otherwise the program would be aborted here. In this case we set the rec variable to None. Otherwise it receives the content of the receive buffer.

If rec is not None, the string is displayed in the message frame and then the content is parsed.

If the first character is a "G", then it is the answer to a G command with the substring T:…. * C; as a payload (aka a substantial part of the data). The second character in rec is the channel number of the addressed cooler unit. The getTempIst () function extracts the temperature value that we note in the Tist list.

We delete the label frame and output the previous setpoint of the temperature there. The unit Cs shows this. As long as the value is displayed, we have the option of entering a new target value by tapping "E" in the bottom line. Entering numbers is ended by tapping "E" again. The "key" "C" deletes the number string backwards.

We write the generated number as a string in the label frame f. Then we send the value in a T command to the server, which then tries to set the temperature in the control loop. To do this, he sets the Tflag of the channel, goes to full power and starts the fan.

To get out of the control loop, we can set the target temperature to 0. The unit can then be switched again by hand, as can the fans. To implement all these subtleties, it was necessary to subsequently adapt the thermobox.py program. The new program is therefore required to try out the control via the wall box.

As always, the main loop closes with the emergency brake with the flash button, which in our case is difficult to access and has therefore been replaced by touching the number input field.

The division of the program parts into listeners and talkers, which both run next to each other, has two major advantages. We do not need to wait for the response from the server after each command sent. This speeds up the loop cycle considerably. Secondly, the server of the display unit can also send data on its own at any time. Then there just has to be a sequence in the while loop that reacts to it. It would be conceivable, for example, that the temperature display is continuously updated in this way, without the control unit having been requested by it. With UDP, this is possible without great effort.

So far we have operated the circuit on the USB bus. However, a separate power supply should be used in the wall housing. This can be a small switching power supply with 5V to 12V output voltage. The cable ends are checked for correct polarity and then placed on the screw terminal on the underside of the circuit board.

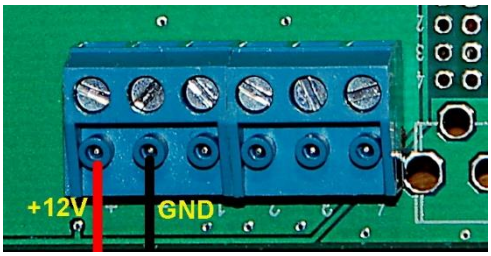

Abbildung 4: Externe Spannung

Now at the latest we save our program thermoXmit.py as boot.py in the workspace and then send this file to the ESP32. The USB cable can now be removed. From now on, the program starts automatically every time it is switched on.

This and many other things from this blog series are now waiting to be implemented in your own projects. I wish you every success and a lot of joy in programming and discovering.