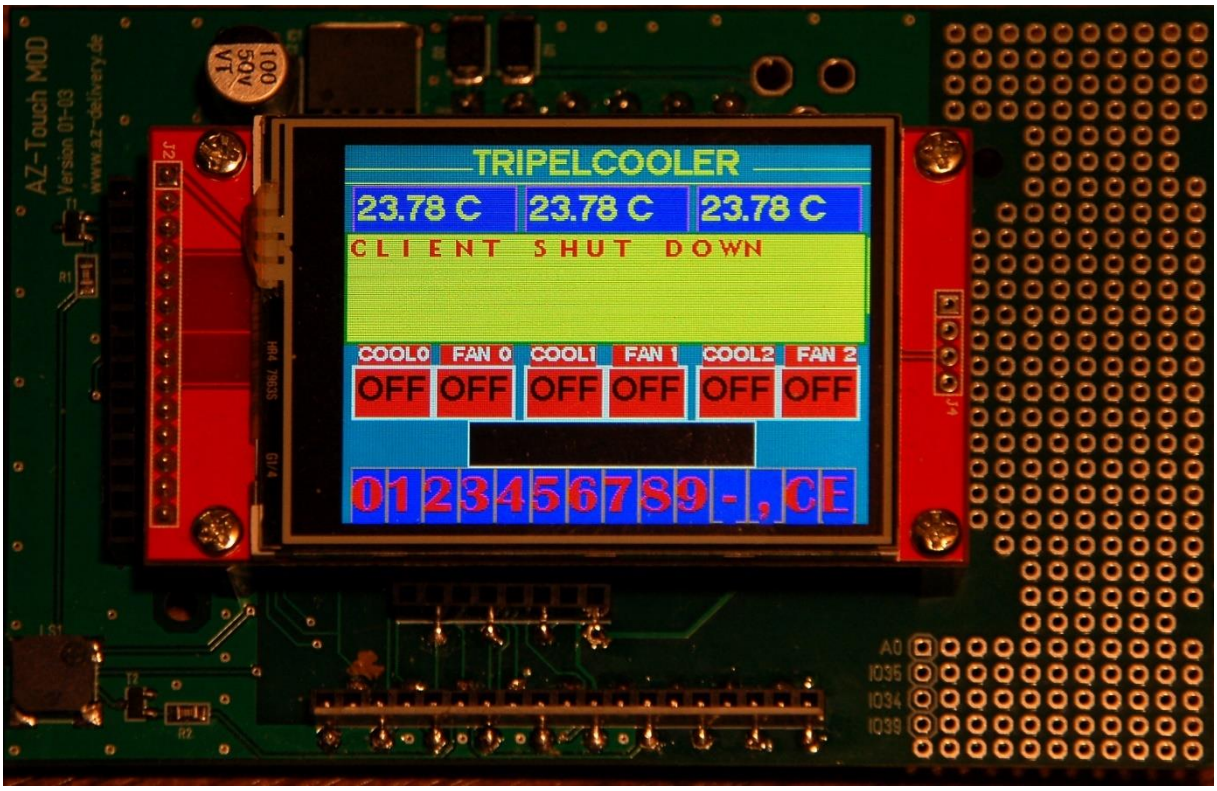

Abbildung 1: Display im Wandmodul

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

Ein Kühlschrank ohne Temperatureinstellung und -regelung ist nur ein halber Kühlschrank. Daher kommt jetzt, nach den bisherigen Entwicklungsschritten der Blogteile 1, 2, 3 und 4,mit dem Teil 5 die Vorstellung einer Steuereinheit auf der Basis einer UDP-Funkübertragung. Das ist notwendig, weil der Cooler keine eigenen Knöpfchen oder Regler besitzt. Gesteuert wird heute nicht über's Handy, sondern mit einem ESP32 in Zusammenarbeit mit einem TFT-Farbdisplay mit Touchscreen via UDP. Der Beitrag zeigt neben der grundlegenden Anwendung eines größeren TFT-Farbdisplays mit 320 x 240 Pixeln und dem Einsatz des damit verbundenen resistiven Touchpads die Programmierung einer Touch-Zehner-Tastatur, die Verwendung von Touch-Schalttasten, Eingabefeldern und einem Meldungsfenster, das seinerseits berührungstechnisch in Unterabschnitte aufgeteilt ist. Außerdem wird sich die Funkübertragung via UDP als sehr praktisch erweisen, gerade, was Rückmeldungen von der Kühleinheit betrifft. Damit herzlich willkommen beim 5. Teil der Reihe mit dem Titel

Peltierelemente –
Wir sprechen mit unserem Kühlschank

Hardware

Die Hardware für diese Folge besteht aus zwei Teilen.

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.

Seien Sie nochmals gewarnt, was die Ausführung des ESP32-Boards angeht. Weil nichts anderes zu haben war, hatte ich ESP32 Lolin LOLIN32 bestellt und musste zu meinem Bedauern feststellen, dass das LOLIN leider nicht zusammen mit der Basisplatine des Displays verwendbar ist. Die Verdrahtung ist über die Leiterplatte fest vorgegeben. Deswegen sind nur die beiden genannten Controllerboards verwendbar. Ein ESP8266 kommt für diese Anwendung überhaupt nicht in Frage, weil bereits der ESP32 an der RAM-Speichergrenze arbeitet.

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 - 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.

Erweiterung der Treiber

Für das bequemere Arbeiten habe ich die Treibermodule angepasst und erweitert. Vor allem bekam die Klasse ILI9341 im Modul ili934xnew Zuwachs um die Klasse FRAME. Hier sind Methoden angesiedelt, die es erlauben, Strukturen zu erstellen, die an die Fenster von Winzig-Weich erinnern. Ich habe sie Rahmen oder eben Frames getauft. Damit ist es möglich, den Bildschirm in kleinere Bereiche aufzuteilen, die sich selbst verwalten können.

Die Basisplatine des Displays enthält, neben einem 5V-Buckconverter mit dem LM2576, auch einen passiven Buzzer, der am besten über einen PWM-Ausgang angesteuert wird. Die Klasse ILI9341 erhielt daher für den Konstruktor einen zusätzlichen, optionalen Parameter, dem ein PWM-Objekt für das GPIO-Pin 21 übergeben werden kann. Geschieht das nicht, dann erzeugt er sich selbst eines. Die Methode beep() nimmt einen Frequenzwert und eine optionale Pulsdauer.

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

Die Klasse für die Bedienung des resistiven Touchpads XPT2046 erhielt Zuwachs um die Methoden **saveCalibration**() und **loadCalibration**(). Damit eine Zuordnung eines Berührpunkts zu einer Bildschirmposition möglich ist, muss eine Kalibrierung durchgeführt werden. Das geschieht mit Hilfe des Programms **calibrate**.py. Dieses Programm musste auch erst auf die Bildschirmverwendung im Querformat angepasst werden. Aus dem Kalibriervorgang gehen 8 Zahlenwerte hervor, die entweder direkt im Anwenderprogramm eingebunden oder, wie hier, in einer Datei (**calibrationdata**.txt) auf dem ESP32 abgelegt werden. Das ermöglicht uns eine spätere Änderung, ohne dass in das Anwenderprogramm eingegriffen werden muss. Die beiden neuen Methoden erlauben das auf einfache Weise. Der Konstruktor eines Touch-Objekts versucht, die Datei zu lesen. Findet er sie nicht, dann nimmt er die Standard-Werte des Moduls. Alternativ kann beim Instanziieren eines Touch-Objekts ein Tuple mit den Kalibrierungsdaten als optionaler Parameter übergeben werden.

Die Zeichensätze

Für die Textausgabe werden TTF-Zeichensätze in MicroPython-Module umgewandelt. Das erledigt das Tool font_to_py.py von Peter Hinch (MIT-Lizenz), von dem allerdings zwei Versionen existieren. Wir hatten dieses Programm schon einmal für die Erstellung von reduzierten Zeichensätzen für ein Schwarz-Weiß-OLED-Display eingesetzt. Wenn Sie selbst Zeichensätze für MicroPython herstellen möchten, laden Sie bitte das Tool von hier (nicht von GitHub!) in ein beliebiges Verzeichnis herunter. Kopieren Sie anschließend die gewünschten TTF-Dateien ebenfalls dort hin. Navigieren Sie nun im Explorer zu diesem Pfad und rechtsklicken Sie mit gedrückter Shift-Taste auf den Verzeichnisnamen. Im Kontextmenü klicken Sie auf **PowerShell-Fenster hier öffnen**.

Angenommen, der Pfad lautet F:_font2py und die TTF-Datei sei britannic.ttf. Dann muss der Aufruf in einem Powershellfenster wie folgt aussehen:

```
.\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>
```

Bei fehlerfreier Ausführung finden wir die Datei **britannic18**.py im gleichen Verzeichnis. Sie wird als Modul **britannic18** ins Anwenderprogramm importiert und mittels der Methode **set_font**(**britannic18**) für die nächste Ausgabe ausgewählt.

Das Zeichensatzmodul bringt einige Funktionen für die Zeichenauswahl und die Bestimmung der Zeichenbreite mit. Letztere braucht ILI9341 für die Darstellung. Hier unterscheiden sich die Versionen des Tools **font_to_py**.py. Fehlt im Modul die Funktion **get_width**(s), dann bricht die Klasse ILI9341 mit Fehlermeldung ab.

Die Schaltung

Die Schaltung ist über die gedruckte Verdrahtung der Basisplatine fest vorgegeben. Der Schaltplan ist vom Hersteller zu haben. Ein kostenloses e-Book beschreibt den Aufbau und die Inbetriebnahme des Moduls. Die mitgelieferten Buchsenleisten werden alle auf der Platinenseite montiert, auf der sich auch die Teile des Step-Down-Reglers befinden. Der ESP32 wird **von der Unterseite her durch die Platine** gesteckt. Dadurch ergeben sich auf der Oberseite spiegelsymmetrische Belegungen der Leisten. Der Abstandshalter rechts vorne dient nur zum Abstützen des Displays. Für die untere Befestigung ist keine Bohrung vorhanden und auch kein Platz, weil genau an dieser Stelle der Stecker des USB-Kabels liegt.

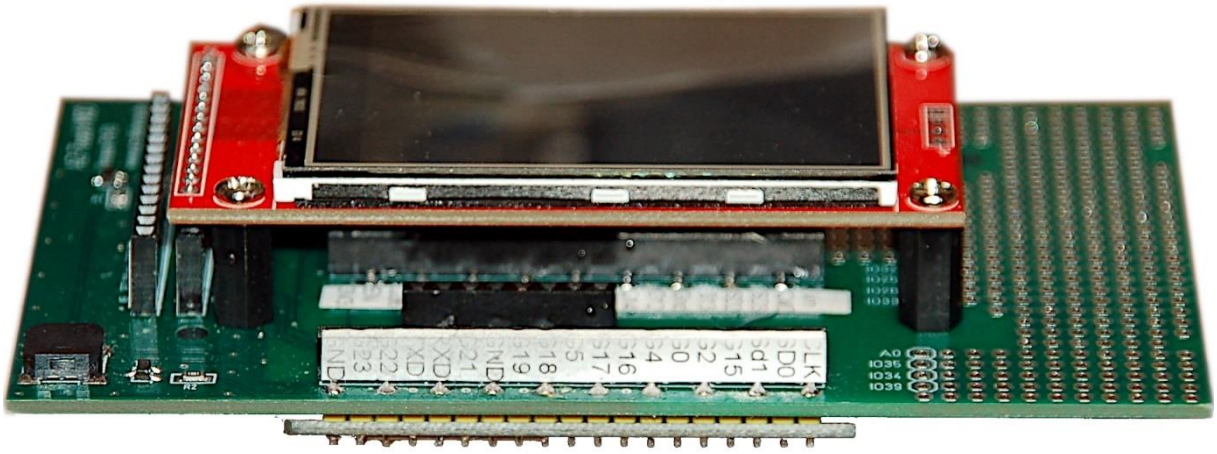

Abbildung 2: Oberseite

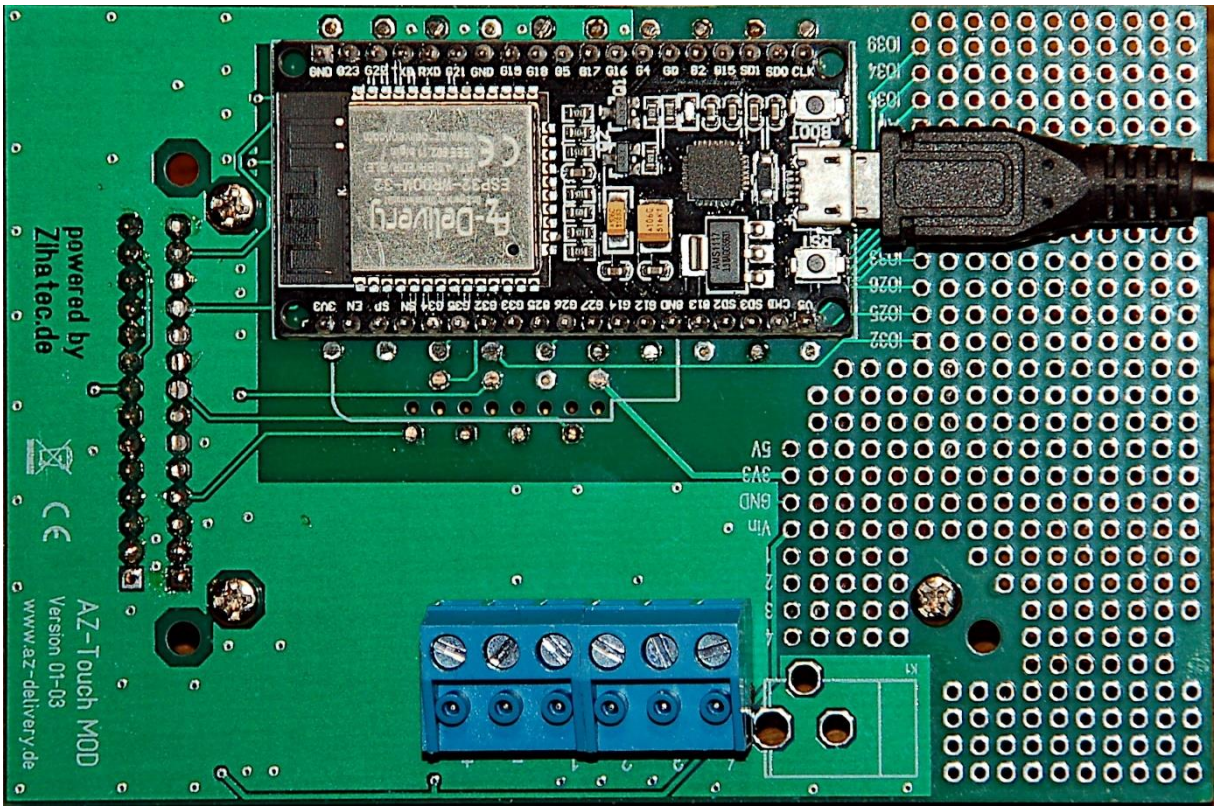

Abbildung 3: Unterseite

Das Programm

Es hat ohne die Treiber- und Zeichensatzmodule einen Umfang von 490 Zeilen und besteht neben dem Import von Modulen und der Definition der Datenstrukturen aus drei wesentlichen Blöcken:

- Einrichtung der Funkstrecke
- Parser und Auftragsübermittler (Talker) an den Server in der Coolbox
- Listener für die Rückmeldungen

Die Kommunikation zwischen ESP32 und Display sowie Touchpad erfolgt via SPI-Bus. Der SPI-Bus ist, wie der I2C-Bus, eine getaktete synchrone Verbindung. Im Gegensatz zum I2C-Bus, der nur eine Datenleitung benutzt und Bausteine durch das Versenden einer Geräteadresse auswählt, verwendet der SPI-Bus zwei Datenleitungen und für jeden Baustein eine eigene Chipselect-Leitung. Über diese Leitung werden durch LOW-legen die Bausteine ausgewählt, mit denen kommuniziert werden soll. Unser Display braucht zwei Auswahlleitungen, eine für das Anzeigemodul ILI9341 und eine für das Touchpad XPT2046.

Das erzeugte SPI-Objekt, spi, arbeitet mit 5MHz. Das ist ein Kompromiss zwischen zuverlässig arbeitendem Touchpad und leidlich schneller Darstellung am Display, das bis 32 MHz funktionieren würde. Display und Touchpad liegen am selben Bus, aber natürlich an unterschiedlichen GPIOs für cs (chip select).

Damit auf dem Display überhaupt etwas sichtbar wird, brauchen wir einen weiteren GPIO-Pin für die Hintergrundbeleuchtung. Achtung! Aus unerklärlichen Gründen wird

die Beleuchtung über einen PNP-Transistor angesteuert und ist daher LOW-aktiv! Zum Ausschalten der Beleuchtung nach einer gewissen Zeit (timerDelay=30000) kann an bestimmten Stellen das Timerobjekt, **tirq=Timer(0)**, aktiviert werden. Diese Stellen befinden sich an mehreren Positionen im Programm. Die Konstante autoAus = 1 aktiviert das Feature grundsätzlich.

Das Display-Objekt heißt d, das Touchpad-Objekt t.
Zur Ansteuerung des Buzzers erzeugen wir das PWM-Objekt buzz.

Wir brauchen eine Reihe von gleichartigen Schalterflächen. Die nötigen Instanzen erstellen wir mit der Funktion **schalter**(). Sie nimmt ausschließlich Positionsparameter und zwar die x-, y-Koordinaten der linken oberen Ecke, Breite b und Höhe h des Buttons. Die Pixelentfernung der nächsten Schaltfläche in x-Richtung steht in versatz. Es folgen Randstärke und Beschriftungstext. Den Abschluss bilden Schriftfarbe, Rahmenfarbe und Hintergrundfarbe, alle drei sind RGB-Werte. Für die Anzeige müssen die 24-Bit-Farbwerte der Form 0xRRGGBB in den 16-Bit-Farbraum umgerechnet werden. Das macht die Funktion **rgbTo565**(), welche ich dem Modul ili934xnew hinzugefügt habe. Im Ergebnis stehen die ersten 5 hochwertigsten Bits für rot, es folgen 6 Bit für grün und die restlichen 5 Bit stellen den Blauwert dar.

```
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
```

Der Rückgabewert von **schalter**() ist die erzeugte Liste von Frameobjekten. Jedes Objekt kennt seine Eigenschaften und kann daher leicht und übersichtlich über die

Methoden der Klasse FRAME bedient werden, ohne dass außer dem Display-Objekt d weitere Parameter übergeben werden müssen.

Die Funktion **daten**() erstellt in ähnlicher Weise Labelfelder zur Anzeige von Texten und Werten her.

```
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
```

Tastenfelder werden durch die Funktion **tastenfeld**() erzeugt. Der übergebene String enthält die darzustellenden Tastenbeschriftungen. Jedem Zeichen entspricht eine Taste. **keyPressed**() ermittelt die gedrückte Taste und **getNumber**() setzt daraus eine Zahl zusammen. Die Taste C löscht rückwärts, E übernimmt. Die Zahl kann inklusive Komma bis zu 10 Stellen haben. Während der Zahleneingabe ist das automatische Ausblenden der Anzeige durch **tirq.deinit**() ausgeschaltet.

```
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()
```

Die Funktion **isrTirq**() ist die Serviceroutine des Timerinterrupts, durch die die Hintergrundbeleuchtung ausgeschaltet wird.

get_stringsize() ermittelt die Länge eines Strings s in Abhängigkeit des angegebenen Zeichensatzes font.

print_centered() gibt den String s so aus, dass dessen Mitte an der Position x,y liegt.

Die Temperatur der Coolereinheit n wird durch **getTempIst**() aus dem von dort zurückgesandten Antwort-String **resp** herausgefiltert.

```
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]
```

Die Funktion **xmit**() sendet den String **msg** an die, im **s.sendto**-Aufruf angegebenen Adressaten, nachdem Reste aus dem Empfangspuffer entfernt wurden. Einer der Adressaten sollte die Coolereinheit sein. Weitere Ziele können zur Kontrolle angegeben sein. Diese Vorgehensweise ist möglich, weil UDP Nachrichten

versendet, ohne eine gesicherte Verbindung wie bei TCP aufzubauen. Das ist vergleichbar mit einer RS232-Schnittstelle, wo man sich nicht darum kümmern muss, ob eine Gegenstelle auch wirklich mithört, wer Ohren hat, der höre!

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

readMessage() liest Nachrichten bis zu 250 Zeichen Länge als Bytesobjekt aus dem Empfangspuffer. Zeilenvorschub und Wagenrücklauf werden entfernt und der Rest in einen String aus ASCII-Zeichen umgewandelt, der zurückgegeben wird.

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

showMessage() zeigt die Zeichenkette **m** im Meldungsfenster an. Farbe und Zeichensatz können als Parameter übergeben werden. Nach **tout** Sekunden wird das Meldungsfenster gelöscht, es sei denn, die letzten beiden Zeilen der Funktion werden auskommentiert. Der Default-Zeichensatz ist geometer16. Er kommt zum Einsatz, wenn der Parameter font nicht angegeben wird.

```
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() arbeitet ähnlich wie **getTempIst**(), ist aber flexibler, weil Suchstring und Endezeichen mit übergeben werden können. Außerdem ist der Rückgabewert ein String und keine Zahl.

Es folgen die Befehle zur Bildschirmeinteilung durch die Erzeugung der diversen Frame-Objekte.

```
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()
```

Die Netzverbindung

Die Anwendung ist funktechnisch vom Prinzip her ein Client. Die Verbindung zu einem Accesspoint kann sowohl über einen WLAN-Router als auch direkt zum optionalen Accesspoint in der Coolerbox erfolgen. Wenn die Verbindung zum Accesspoint steht, wird der Socket des UDP-Clients instanziiert. IP-Daten und Portnummer müssen natürlich an die örtliche Gegebenheit angepasst werden, ebenso die Zugangsdaten des Accesspoints.

Der Parser

Der Parser, in Form der Funktion **doJobs**(), nimmt die Koordinaten als ein Tuple, das der Aufruf der **t.get_touch**()-Methode in der Hauptschleife ermittelt hat. Durch die Rotation des Displays um 270° (r=3 im Konstruktoraufruf des Display-Objekts d) sind x und y vertauscht. Beim Entpacken der Koordinaten wird das korrigiert. Dann werden einfach der Reihe nach alle Frame-Objekte abgefragt, ob die Koordinaten des Berührpunkts aus der Fläche des Objekts stammen. Das macht die FRAME-Methode **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))
```

Die Schalter werden, weil als Liste vorliegend, in einer for-Schleife abgefragt. Kann das Programm die Koordinaten einer Schalterfläche zuordnen, dann wird der vorher leere String **senden** mit Sendedaten gefüllt, nachdem das Erscheinungsbild der Schaltfläche geändert wurde. **senden** enthält jetzt den Befehlscode, den wir in der in der vorangehenden Blogfolge schon über die Tastatur getestet haben. Ist der String nicht leer, dann wird er an den Cooler geschickt und **senden** wieder geleert.

Ein Tipp auf den schwarzen Rahmen, in dem eingegebene Zahlen erscheinen, beendet das Programm. Das Serverprogramm im Cooler wird davon nicht beeinflusst.

Ein Tipp auf das Meldungsfenster ruft den Status einer Coolereinheit ab. Der Parser decodiert den zugehörigen Bereich anhand einer genaueren Betrachtung des x-Werts und wählt dadurch die entsprechende Einheit aus, deren Daten zurückgeschickt werden. Die Vergleichs-x-Werte nehmen wir von den Temperatureingabefeldern.

Wird ein Temperaturfenster angetippt, geht ein g-Befehl an den Cooler, welcher dort bewirkt, dass die aktuelle Temperatur zurückgeschickt wird.

Der Listener

Während der Parser die Senderolle übernimmt, spielt die Hauptschleife den Listener, der auf eingehende Nachrichten, also die Antworten vom Server lauscht. Diese Trennung hat sich als sehr nützlich erwiesen, weil die Zeiten zwischen Auftragsvergabe und Eintreffen der Antwort nicht unerhebliche Verzögerungen im

Programmablauf hervorrufen. Die while-Schleife ist auf möglichst raschen Durchlauf optimiert, wodurch Tipp-Ereignisse schneller erkannt werden und auch rascher auf eintreffende Meldungen reagiert werden kann.

```
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()
```

Die zurückkommenden Antworten vom Server dienen nicht nur der Anzeige im Meldungsrahmen, sondern lösen auch teilweise wieder neue Befehle an den Server aus. Aber gehen wir einfach der Reihe nach vor.

Am Anfang steht die Abfrage des Touchpads. Wurde es angetippt, dann ist der Rückgabewert c der Methode **t.get_touch**() ein Tuple der normierten Koordinaten. Das Programm reagiert darauf mit dem Aufruf des Parsers, dem das Koordinaten-Tuple übergeben wird. Was der Parser damit tut, das hatten wir schon.

Ist kein Tipp erfolgt, dann schauen wir nach, ob eine Antwort vom Server vorliegt, die abzuarbeiten ist. Wurde die Empfangsschleife ohne Ergebnis mit Timeout verlassen, dann wirft MicroPython eine Exception, die es abzufangen gilt, weil sonst hier das Programm abgebrochen würde. Wir setzen die Variable **rec** in diesem Fall auf None. Sonst erhält sie den Inhalt des Empfangsbuffers.

Wenn **rec** nicht None ist, wird der String im Meldungsframe angezeigt und dann der Inhalt geparst.

Ist das erste Zeichen ein "G", dann ist es die Antwort auf einen G-Befehl mit der Teil-Zeichenfolge T:….*C; als Payload (aka wesentlicher Datenanteil). Das zweite Zeichen in **rec** ist die Kanalnummer der angesprochenen Coolereinheit. Die Funktion **getTemplst**() extrahiert den Temperaturwert, den wir uns in der Liste **Tist** merken.

Wir löschen den Label-Frame und geben dort den bisherigen Sollwert der Temperatur aus. Die Einheit **Cs** lässt dies erkennen. Solange der Wert angezeigt wird, haben wir die Möglichkeit mit einem Tipp auf "E" in der untersten Zeile die Eingabe eines neuen Zielwerts einzuleiten. Die Zifferneingabe wird durch erneutes Tippen auf "E" beendet. Die "Taste" "C" löscht den Ziffernstring rückwärts.

Die erzeugte Zahl schreiben wir als String in den Label-Frame f. Dann senden wir den Wert in einem T-Befehl an den Server, der daraufhin versucht die Temperatur im Regelkreis einzustellen. Dafür setzt er das Tflag des Kanals, geht auf volle Leistung und startet die Lüfter.

Um aus dem Regelkreis herauszukommen, können wir die Solltemperatur 0 setzen. Danach lässt sich auch die Einheit wieder von Hand schalten, ebenso die Lüfter. Zum Umsetzen all dieser Feinheiten war ein nachträgliches Anpassen des Programms **thermobox**.py notwendig. Für das Ausprobieren der Steuerung via Wandbox ist also das neue Programm nötig.

Den Abschluss der Hauptschleife bildet wie immer die Notbremse mit der Flashtaste, die allerding in unserem Fall schlecht zugängig ist und deshalb durch den Touch auf das Zifferneingabefeld ersetzt wurde.

Die Aufteilung der Programmteile in Listener und Talker, die beide neben einander her laufen, hat zwei große Vorteile. Wir brauchen nicht nach jedem abgesetzten Befehl auf die Antwort vom Server warten. Das beschleunigt den Schleifendurchlauf erheblich. Dazu kommt als Zweites, dass der Server der Anzeigeeinheit jederzeit auch von sich aus Daten senden kann. Es muss dann nur in der while-Schleife eine Sequenz geben, die darauf reagiert. So wäre es zum Beispiel denkbar, dass die Temperaturanzeige auf diesem Weg laufend aktualisiert wird, ohne dass dieses die Steuereinheit angefragt hat. Mit UDP ist so etwas ohne großen Aufwand möglich.

Bisher haben wir die Schaltung am USB-Bus betrieben. Im Wandgehäuse soll aber eine eigene Spannungsversorgung eingesetzt werden. Das kann ein kleines Schaltnetzteil mit 5V bis 12V Ausgangsspannung sein. Die Kabelenden werden auf korrekte Polung geprüft und dann an die Schraubklemme auf der Unterseite der Platine gelegt.

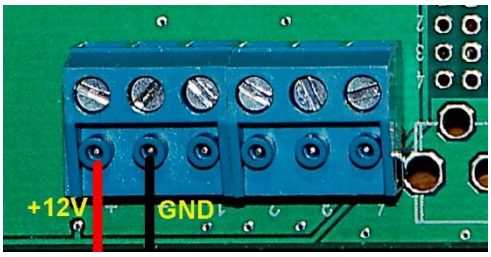

Abbildung 4: Externe Spannung

Spätestens jetzt speichern wir unser Programm **thermoXmit.py** als **boot.py** im Workspace ab und senden diese Datei dann an den ESP32. Nun kann das USB-Kabel abgezogen werden. Ab sofort startet das Programm bei jedem Einschalten automatisch.

Das und manches Andere aus dieser Blogreihe wartet nun auf die Umsetzung in Ihren eigenen Projekten. Dazu wünsche ich Ihnen viel Erfolg und viel Freude am Programmieren und Entdecken.