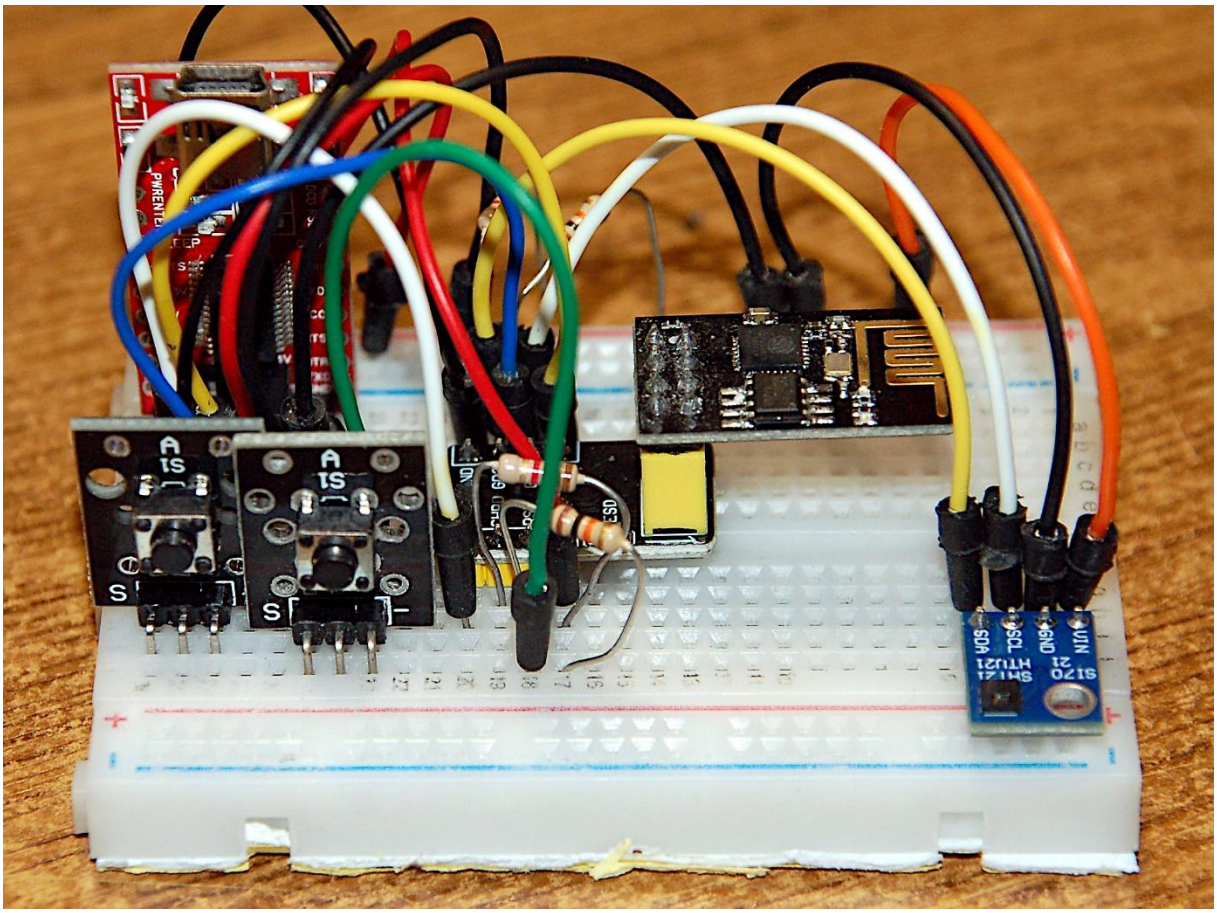

ESP8266-01 als Temperatur- und Feuchte-Sender

Dieser Beitrag ist auch als PDF-Dokument verfügbar.

Wenn Sie das Kabel zwischen dem Temperatur- und Feuchtesensor HTU21 und dem Matrixdisplay virtuell verlängern möchten oder wenn Sie einen anderen Zeichensatz für Ihr Matrix-Display wünschen, sind Sie hier richtig. Ich verrate Ihnen, wie Sie einen TTF-Zeichensatz von Windows in einen MicroPython-Zeichensatz umwandeln können und das Kabel ersetzen wir ganz einfach durch Funkübertragung. Willkommen zu

MicroPython auf dem ESP32 und ESP8266

heute

Mammut-Matrix-Display der HTU21 und WLAN

Ich beginne mal gleich mit der Hardware. Zum Mammut-Matrix-Display und dem zugehörigen Controller erfahren Sie mehr im ersten Teil dieser Reihe. Dort finden Sie den Schaltplan und eine Beschreibung der Arbeitsweise des Displays. Ferner wurde ein MicroPython-Modul für die Ansteuerung des Displays entwickelt, das in weiteren Projekten Anwendung finden kann und wird.

Im zweiten Teil ging es um den Temperatur- und Feuchtesensor HTU21 alias SHT21, für die Ansteuerung wurde ebenfalls ein MicroPython-Modul gebaut. Daneben erfahren Sie dort mehr über die Arbeitsweise des I2C-Busses und es wird gezeigt, wie man mit preiswerten Mitteln die Bussignale sichtbar machen kann. Der eingesetzte 8-Kanal-Logicanalyzer ist dafür bestens geeignet. Außerdem gibt es Informationen zur CRC-Prüfsummenberechnung.

In dieser Episode verwenden wir alle bisher eingesetzten Teile wieder. Zum Aufbau einer Funkstrecke wird aber ein zweiter Controller benötigt. In Frage kommen alle Mitglieder der ESP32/ESP8266- Familien, sogar ein ESP8266-01 kann den Sensor- und Sender-Part übernehmen. Und genau mit diesem habe ich den aktuellen Post vorbereitet.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F oder ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	ESP8266 ESP-01S WLAN WiFi Modul
1	FT232-AZ USB zu TTL Serial Adapter für 3,3V und 5V
1	Breadboardadapter für ESP-01 Breadboard-zu-ESP8266 01
2	KY-004 Taster Modul
1	Widerstand 10kΩ
1	AMS1117 3,3V Stromversorgungsmodul
	oder statt dessen
1	NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI WLAN unverlötet mit CP2102
nach Bedarf	MAX7219 8x8 1 Dot Matrix MCU LED Anzeigemodul oder MAX7219 8x32 4 in 1 Dot Matrix LED Anzeigemodul
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins
1	Sperrholzstreifen 5x 25cm … zur Display-Montage

Die Schaltung für den ESP8266-01S erscheint aufwändiger, weil Baugruppen benötigt werden, die zum Beispiel ein Amica bereits an Bord hat. Letztlich bleiben, wenn Entwicklung und Programmierung erledigt sind, im Betrieb auch nur der ESP8266-01S und das Modul GY21 mit dem HTU21 übrig. Das Netzteil kann auch durch zwei AA-Batterien ersetzt werden. Der Anschluss erfolgt dann an den 3,3V Pins und GND.

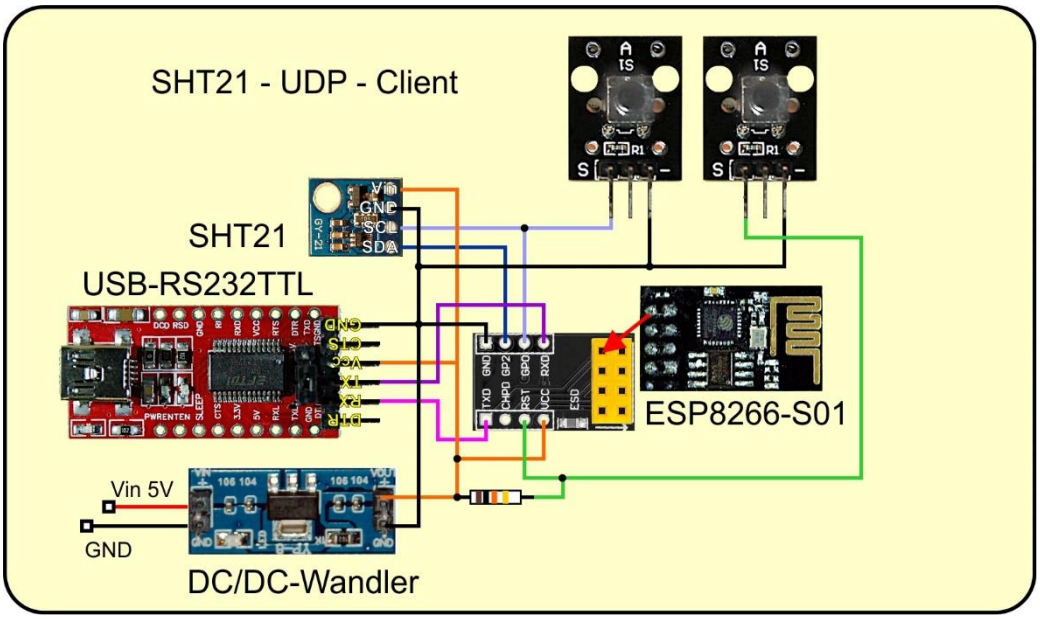

Abbildung 1: SHT21-UDP-Client

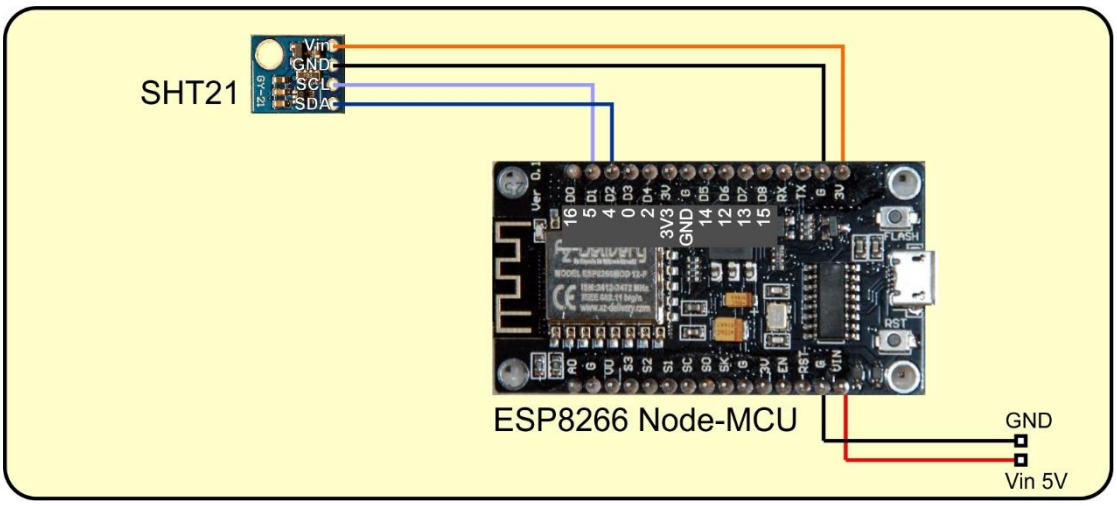

Abbildung 2: SHT-UDP-Client mit ESP8266-Node-MCU

Außer der Hardware ist für diese Blogfolge noch eine weitere vorbereitende Maßnahme nötig. Es geht um schlankere Zeichensätze, damit mehr Information in das Display passt. Wie Sie dazu kommen, das erkläre ich im folgenden Kapitel.

Clonen von Windows-Zeichensätzen für die Anwendung in MicroPython

Verwendete Software:

micropython-font-to-py

Wie es gemacht wird

Um einen Vektorzeichensatz von Windows ins Pixelformat zu übertragen, benutzen wir eine Freeware von Peter Hinch, die der MIT-Licence unterliegt. Sie heißt micropython-font-to-py und wird als ZIP-Datei von Git-Hub heruntergeladen.

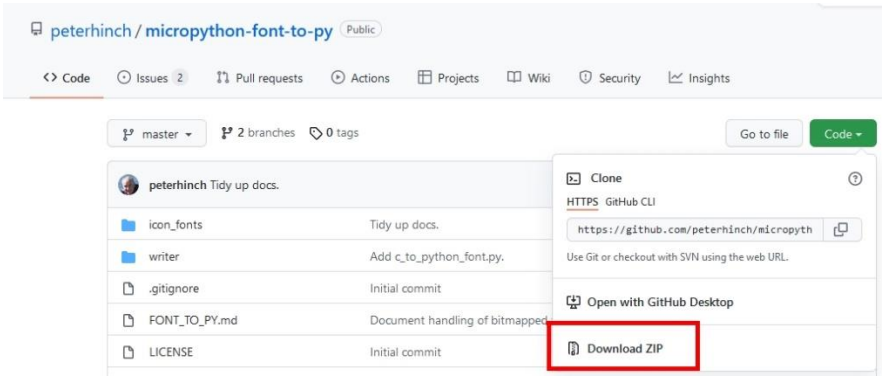

Abbildung 3: Software Download von Github

Für die Installation habe ich das Verzeichnis font2py im Root-Verzeichnis meiner Festplatte F: angelegt. Das und alles Weitere funktioniert übrigens auch zusammen mit einem USB-Stick, wenn die Festplatte sauber bleiben soll.

Die Datei **micropython-font-to-py-master.zip** habe ich in **F:\font2py** gespeichert und auch dort entpackt.

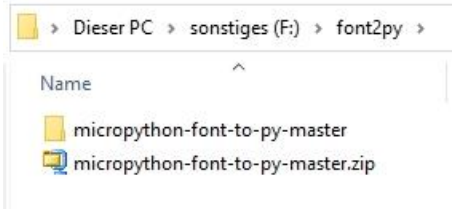

Abbildung 4: Nach dem Entpacken

Das entstandene Verzeichnis **micropython-font-to-py-master** habe ich in **f2p** umgetauft, der Name war mir zu lang. Wechseln wir jetzt in dieses Verzeichnis. Die beiden Pythonprogramme **font_to_py.py** und **font_test.py** werden wir in Kürze benutzen. Zuvor legen wir noch ein Verzeichnis **quellen** an.

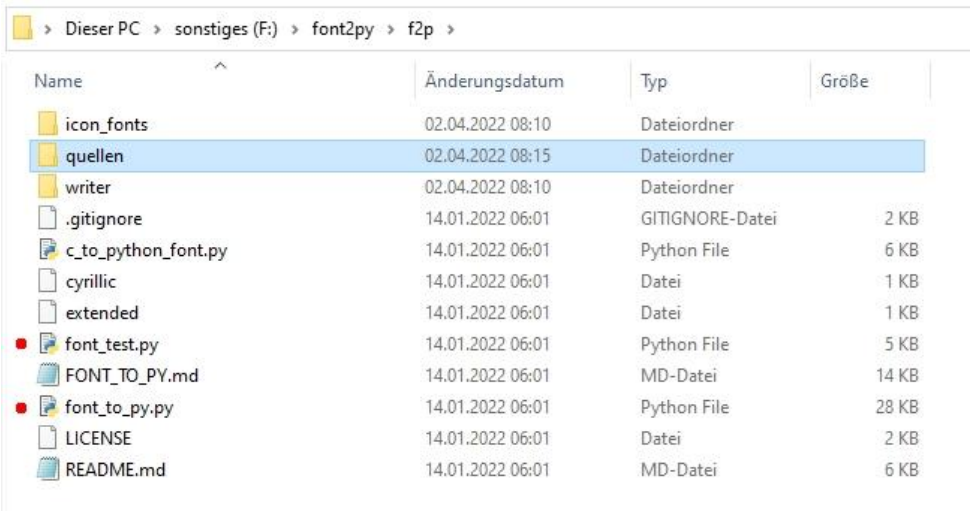

Abbildung 5: Arbeitsumgebung

Öffnen Sie nun den Schriftenordner von Windows, **C:\Windows\Fonts**. Suchen Sie nach einer möglichst klaren Schriftform und kopieren Sie die Datei in das Verzeichnis **F:\font2py\f2p\quellen**. Ich habe hier **FRADM.TTF** gewählt.

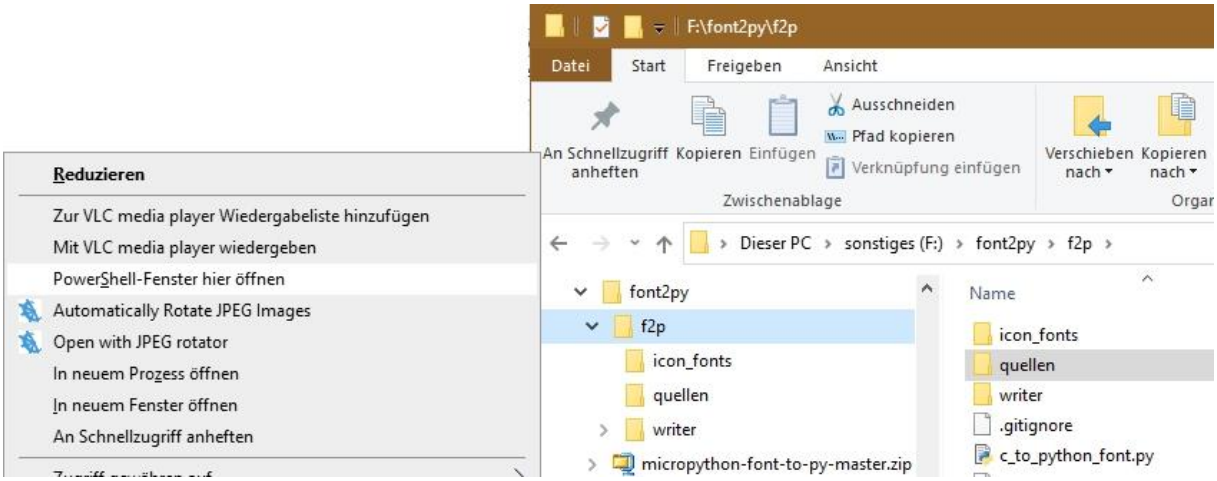

Abbildung 6: Powershell Fenster öffnen

Die nächsten beiden Schritte erfolgen in einem Powershell-Fenster. Wir öffnen das Contextmenü mit gedrückter **Umschalttaste** (Shift) und einem **Rechtsklick** auf den Ordner **f2y** und wählen **Powershell-Fenster hier öffnen**. Der Prompt zeigt uns, dass wir im richtigen Verzeichnis sind. Dann setzen wir folgendes Kommando ab:

.\font_to_py.py F:\font2py\f2p \quellen\FRADM.TTF 7 -x franklin8.py

Bitte beachten Sie, dass der Befehl mit einem ".\" eingeleitet werden muss, damit er in der aktuellen Pfadumgebung gefunden wird. - Nein, die 7 ist hier schon korrekt. Mit einer 8 als Zeichenhöhe würde **font_to_py.py** einen Zeichensatz mit 9 Pixel Höhe erzeugen. Warum? Das liegt wohl am geheimnisvollen Zusammenwirken von TTF-Font und **font_to_py.py**. Mit anderen Fonts klappt das nämlich wie erwartet – Jugend forscht!

Abbildung 7: Der erste MicroPython-Zeichensatz

Soeben haben wir den Zeichensatz **FRADM.TTF** mit 7 Pixel Zeichenhöhe in eine Python-Datei umgewandelt. Der Schalter **-x** sorgt dafür, dass Zeicheninformationen zeilenweise abgelegt werden. Das ist wichtig, weil unser Matrix-Display es so haben will. Ein Proportional-Zeichensatz wird mit unseren Angaben auch mit variablen Zeichenbreiten umgesetzt. Wenn Sie lieber gleiche Zeichenbreiten bevorzugen, fügen Sie einfach den Schalter **-f** noch mit in die Befehlszeile ein.

Natürlich interessiert es uns, wie der Zeichensatz aussieht.

.\font_test.py franklin8 "0123456789"

Abbildung 8: Fonttest

Der TTF-Font wurde in ein MicroPython-Modul **franklin8**.py umgewandelt, das wir im Ordner **F:\font2py\f2p** finden. Zur weiteren Verwendung muss es ins Arbeitsverzeichnis von Thonny kopiert werden.

Die Software zur Funk-Messstation

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen
ESP8266 mit 1MB Version 1.18 Stand: 25.03.2022 oder
ESP32 mit 4MB Version 1.18 Stand 25.03.2022

Die MicroPython-Programme zum Projekt:

matrix8x8.py Treibermodul für den MAX7219
matrixtest.py Demoprogramm für ein x mal 8 -Matrixdisplay
sht21.py Treibermodul für das GY-21-Modul
sht21_32_test.py Testprogramm für den ESP32
sht21_8266_test.py Testprogramm für den ESP8266
shtdisplay.py Thermo- Hygrometer Software mit Writer
shtdisplay+.py Thermo- Hygrometer Software mit Writer
WLANdisplay.py Anzeigeeinheit mit WLAN-Anbindung
APdisplay.py Anzeigeeinheit mit eigenem Accesspoint
sht21client.py Programm für den HTU21-Client
writer.py Der Treiber für die Anzeige mit den neuen Zeichen.

Sonstige Software:

Packet Sender Downloadseite
Packet Sender Windows Install Version
Packet Sender Windows Portable
Packet Sender Linux

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die MicropythonFirmware (Stand 05.02.2022) 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, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

An dieser Stelle ein paar Takte zum Flashen des ESP8266-01. Der hat nämlich, anders als seine größeren Geschwister, keine Flash-Automatik an Bord. Hier ist Handarbeit gefragt.

Der Flashvorgang gliedert sich in zwei Teile, erstens Flash-Speicher löschen und zweitens Firmware übertragen. Die folgende Liste ist ein Auszug aus der Beschreibung für den Flashvorgang:

a) In Thonny die Vorbereitungen erledigen
b) Reset- und Flash-Taste drücken
c) In Thonny den Flashvorgang starten
d) Reset-Taste lösen, Flash-Taste halten, bis der Fortschritt angezeigt wird
e) Flash-Taste lösen
f) Warten bis erneut der Zugriff auf die COM-Schnittstelle gemeldet wird
g) Dann erneut die Punkte b) bis f) durchlaufen und
h) abschließend das Installer-Fenster schließen und die Options mit OK beenden

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 kompilieren 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 Makro 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 kompiliertes 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.

Der Zeichensatztest

Nach allen Vorbereitungen wird es jetzt Ernst. Testen wir zuerst den neuen Zeichensatz. Im Modul FrameBuffer, von dem unsere Klasse MATRIX erbt, ist bereits ein Zeichensatz enthalten, auf den die Methode FrameBuffer.**text**() zugreift. Wenn wir einen, oder mehrere Zeichensätze hinzufügen wollen, brauchen wir dafür auch einen eigenen Treiber, wenn die Firmware MicroPython nicht neu kompiliert werden soll oder kann. Denn irgendwie müssen die Pixelinformationen ja in den Framebuffer kommen. Einen solchen Treiber hat Peter Hinch in seinem Paket **micropython-font-to-py-master** integriert.

Ich habe nun die besagte Treiberdatei mit Namen writer.py etwas verändert, um sie an meine Bedürfnisse anzupassen. Ferner habe ich zwei Routinen hinzugefügt, damit auch weiterhin die Befehle **roll**() und **center**() zur Verfügung stehen. Damit die Veränderungen greifen, dürfen Sie nicht die writer.py aus dem Paket micropython-font-to-py-master verwenden, sondern diese hier. Als Modul muss die Datei, wie auch die Zeichensatzdatei selbst, in den Flash des Controllers hochgeladen werden.

Nach dem Öffnen der Datei shtdisplay+.py in einem Editorfenster, starten wir das Programm mit F5. Im 3-Sekundentakt wird eine 7-stellige Temperaturanzeige in einem 4-stelligen Display dargestellt. Das Schriftbild der gewandelten Fonts ist teils gewöhnungsbedürftig, teils kann man es vergessen. Ihnen obliegt nun die Auswahl eines gut brauchbaren Zeichensatzes aus dem Windows-Font-Ordner. In Abbildung 9 wird ein Amica Node-MCU 3 als Controller für das Display verwendet.

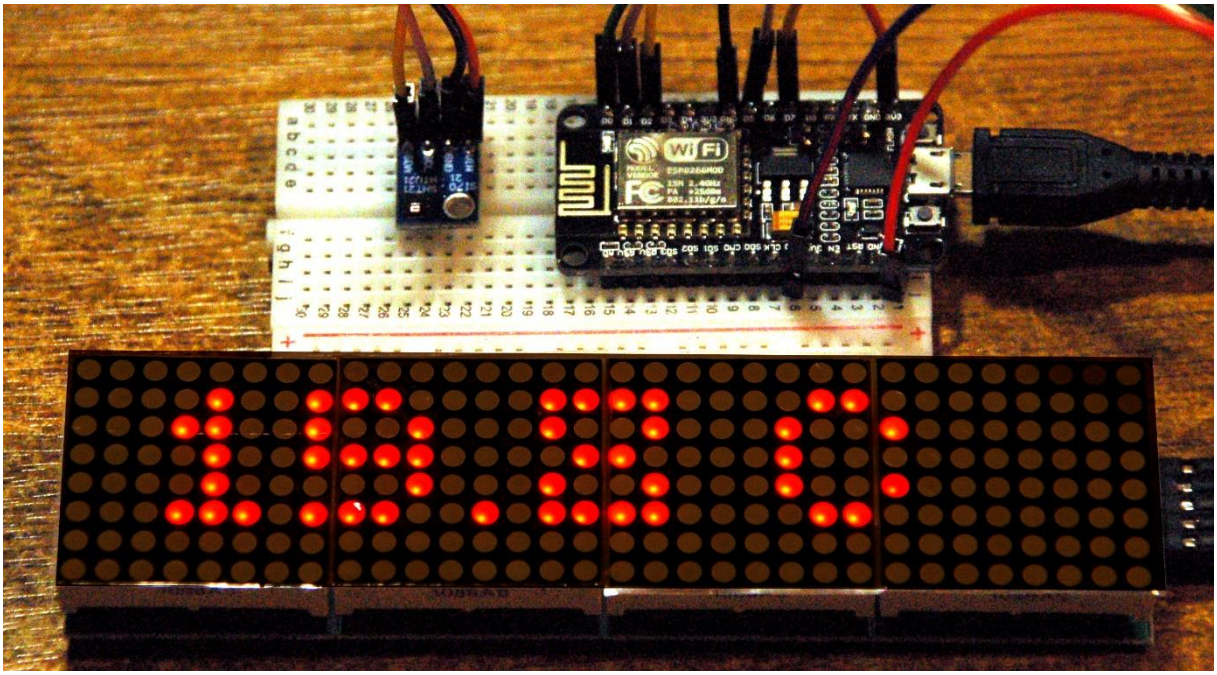

Abbildung 9: Temperaturanzeige 7-Stellig im 4-er Display

Die folgenden Stellen im abgewandelten Programm shtdisplay+.py im Vergleich zu dem aus dem vorigen Blog-Post shtdisplay.py habe ich im Listing hervorgehoben.

```
# _shtdisplay.py
import sys, os
from time import sleep_ms,sleep
from matrix8x8 import MATRIX
from machine import Pin, SoftI2C, SPI
from i2cbus import I2CBus
from sht21 import SHT21
import franklin8 as charset
from writer import Writer

chip=sys.platform
if chip == 'esp8266':
    # Pintranslator fuer ESP8266-Boards
    # LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
```

```
    # ESP8266 Pins 16  5  4  0  2 14 12 13 15
    #                SC SD
    bus = 1
    MISOp = Pin(12) # D6
    MOSIp = Pin(13) # D7
    SCKp  = Pin(14) # D5
    spi=SPI(1,baudrate=4000000)   #ESP8266
    # # alternativ virtuell mit bitbanging
    # spi=SPI(-1,baudrate=4000000,sck=SCK,mosi=MOSI,\
    #         miso=MISO,polarity=0,phase=0)  #ESP8266
    CSp = Pin(16, mode=Pin.OUT, value=1) # D0
    SCL=Pin(5) # D1
    SDA=Pin(4) # D2
elif chip == 'esp32':
    bus = 1
    MISOp= Pin(15)
    MOSIp= Pin(13)
    SCKp = Pin(14)
    spi=SPI(1,baudrate=10000000,sck=Pin(14),mosi=Pin(13),\
            miso=Pin(15),polarity=0,phase=0)  # ESP32
    CSp = Pin(4, mode=Pin.OUT, value=1)
    SCL=Pin(21)
    SDA=Pin(22)
    taste=Pin(0,Pin.IN,Pin.PULL_UP)
    blinkLed=Pin(2,Pin.OUT)
else:
    # blink(led,800,100,inverted=True,repeat=5)
    raise OSError ("Unbekannter Port")

print("Hardware-Bus {}: Pins fest vorgegeben".format(bus))
print("MISO {}, MOSI {}, SCK {},
CS{}".format(MISOp,MOSIp,SCKp,CSp))
print("SCL {}, SDA {}\n".format(SCL,SDA))

numOfDisplays=4
d=MATRIX(spi,CSp,numOfDisplays)
i2c=SoftI2C(SCL,SDA)
sleep_ms(15) # for booting SHT-Device
s=SHT21(i2c)
w = Writer(d, charset)

print("Install done")

delay=3

while 1:
    w.setTextPos( 0, 0)
    s.readTemperatureRaw()
    tempString="{0:5.1f} C"
    humString="{0:4.1f} %"
    s.calcTemperature()
    print(s.Temp,"°C")
```

```
    w.printString(tempString.format(s.Temp))
    d.show()
    sleep(delay)
    d.clear()
    w.setTextPos( 0, 0)
    s.readHumidityRaw()
    s.calcHumidity()
    print(s.Hum,"%RH")
    w.printString(humString.format(s.Hum))
    d.show()
    sleep(delay)
    d.clear()
```

Die Klasse **Writer** wird importiert und instanziiert. Dabei übergebe ich das bisherige Displayobjekt **d**. Das Writer-Objekt **w** greift über **d** auf die Klasse **FrameBuffer** zu. Wir verwenden die beiden Treiber parallel zueinander und nutzen so die Synergieeffekte.

Bevor es zur Funkabteilung weitergeht, noch eine Bemerkung zum Original-Treiber von Peter Hinch. Er hat das Programm so konzipiert, dass mehrere Schriftarten in derselben Anwendung möglich sind. Daher ist seine **set_textpos**()-Funktion Instanz übergreifend mit dem Decorator @staticmethod versehen. Jeder Schriftentreiber greift ja auf dasselbe Display zu und muss wissen, wo er oder ein anderer Treiber zu schreiben aufgehört hat. Nur eine Funktion, die für alle Instanzen zugänglich ist, kann das gewährleisten. Ich habe die Funktion als Instanz-Methode umgeschrieben, weil für die wenigen Pixel ein Zeichensatz reichen dürfte. Damit fällt die Mystic von Peters Ansatz weg und das Listing ist für einen Anfänger leichter zu durchdringen, wenigstens an dieser Stelle. Die Magie des Restes erschließt sich in der Hauptsache erst nach intensivem Studium des Listings und der Kenntnis des Innenlebens der Klasse FrameBuffer.

Der Funksensor und sein Display

Für den Funkverkehr haben wir eigentlich, mit der Einbindung des Sensors und der Ansteuerung der Anzeige, zwei von vier Jobs schon erledigt. Das Programm shtdisplay+.py oder sein Vorgänger shtdisplay.py muss aufgeteilt werden in Sensor- und Anzeige-Einheit, und die Teile sind mit einem UDP-Client beziehungsweise UDP-Server zu erweitern. Ferner muss entschieden werden, ob der Zugriff über ein lokales Netzwerk mit WLAN-Accesspoint erfolgen soll, oder ob eine Insellösung mit ESP-eigenem Accesspoint bevorzugt wird. Es ist aber auch nicht schwer, den einen Ansatz mit dem anderen zu vertauschen, weil dazu nur die Zeilen für das Funkpodest ausgetauscht werden müssen. Das Importieren der Module **network** und **socket** wird überall benötigt. Für die Anzeigeeinheit brauchen wir eine Serverschleife, die zusammen mit den zwei Funktionen **getData**() und **doJobs**() Aufträge empfängt und ausführt. Auch die Serverschleife ist für beide Ansätze dieselbe.

Für die WLAN-Lösung des Display-Teils sieht dieser Bereich so aus. Die Kommentare sagen, was gerade passiert.

```
# Unbedingt das AP-Interface ausschalten
nac=network.WLAN(network.AP_IF)
nac.active(False)
nac=None

# Wir erzeugen eine Netzwerk Interface-Instanz
nic = network.WLAN(network.STA_IF)
nic.active(False)
# Abfrage der MAC-Adresse zum Eintragen im Router,
# damit die Freigabe des Zugangs erfolgen kann
MAC = nic.config('mac')
myID=hexMac(MAC)
print("Client-ID",myID)

# Wir aktivieren das Netzwerk-Interface
nic.active(True)

# Aufbau der Verbindung
# Wir setzen eine statische IP-Adresse
nic.ifconfig(("10.0.1.96","255.255.255.0","10.0.1.20","10.0.1.
100"))

# Anmelden am WLAN-Router
nic.connect(mySSID, myPass)

if not nic.isconnected():
    # warten bis die Verbindung zum Accesspoint steht
    while not nic.isconnected():
        print("{}.".format(nic.status()),end='')
        d.text("."*n,0,0,1)
        d.show()
        sleep(1)

# Wenn verbunden, zeige Verbindungsstatus & Config-Daten
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
if nic.isconnected():
    # War die Konfiguration erfolgreich? Kontrolle
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
else:
    print("No AP found")
    d.roll("No AP found",cnt=2)
    d.clear()
    d.center("stoped")
```

Für die Display-Einheit mit eigenem AP fällt dieser Teil deutlich kürzer aus.

```
# ********************* AP einrichten ***********************
# Constructoraufruf erzeugt WiFi-Objekt nic
nic = network.WLAN(network.AP_IF)
#
nic.active(True)  # Objekt nic einschalten
#
MAC = nic.config('mac')  # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)        # in eine Hexziffernfolge umgewandelt
print("AP MAC: \t"+myMac+"\n") # ausgeben
#
ssid = 'choose a name'; passwd = "choose a passwd"
nic.ifconfig(("10.1.1.96","255.255.255.0","10.1.1.96","10.1.1.
96"))
print(nic.ifconfig())
#
# MicroPython unterstuetzt keine Authentifizierung
nic.config(authmode=0)
print("Authentication mode:",nic.config("authmode"))

# SSID konfigurieren und auf Aktivierung des AP-Interfaces
# warten, Zustand überprüfen
nic.config(essid=mySSID, password=myPass)
while not nic.active():
  print(".",end="")
  sleep(1)
print("NIC active:",nic.active())
```

In beiden Fällen muss ein Socket eingerichtet werden, auf dem der Datenaustausch ausgetragen wird.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
print("waiting on ",myPort)
s.settimeout(0.1)
```

socket.SO_REUSEADDR sorgt dafür, dass bei einem Neustart des Programms ohne Reset dieselben Socketdaten, IP-Adresse und Portnummer, wiederverwendet werden können. Der **bind**-Befehl bindet die bei der Netzwerkverbindung vergebene IP-Adresse an die Portnummer in **myPort**. Die Vereinbarung eines Timeouts verhindert, dass das Warten auf eingegangene Anfragen die Serverschleife blockiert.

Die Serverschleife ist sehr überschaubar und besteht im Wesentlichen aus den Aufrufen der Funktion **getData**(), die nachschaut, ob Nachrichten eingetroffen sind und dem Aufruf von **doJobs**(), wenn dem so war.

```
while 1:
    response=getData()
    if response is not None:
        doJobs(response)

    if taste()==0:
        sys.exit()
        #reset()
```

getData() prüft, ob Daten eingetroffen sind.

```
def getData():
    try:
        rec,adr=s.recvfrom(150)
        if rec is not None:
            resp=rec.decode()
            cmd,val=resp.split(":",1)
            return cmd,val
    except OSError:
        return None
```

Die Methode **recvfrom**() ist die Empfangsschleife. Sie wird so lange durchlaufen bis eine Nachricht eintrifft, oder bis der eingestellte Timeout abgelaufen ist. In letzterem Fall wird eine timeout-Exception geworfen, die wir mit **except OSError** abfangen. In diesem Fall wird von der Funktion ein **None** zurückgegeben. Exceptions (Ausnahme-Fehler) anderer Art, zum Beispiel ein fehlender Doppelpunkt in der Nachricht, führen trotzdem zum Programmabbruch.

Liegt eine Nachricht vor, dann enthält **rec** die Daten und **adr** die Socketdaten des Absenders. Da wir als Daten Texte erwarten, wandeln wir die bytes-Folge in ASCII-Text um. Die Syntax der Nachricht verlangt mindestens einen Doppelpunkt, der den Befehlsanteil von Datenteil trennt. Der aufgespaltene Inhalt wird in diesem Fall als Tuple zurückgegeben.

```
def doJobs(resp):
    try:
        cmd,val=resp
        cmd=cmd.upper()
        # linksbuendig
        if cmd == "L":
            d.clear()
            d.text(val,0,0,1)
            d.show()
        # zentriert
        if cmd == "C":
            d.clear()
            d.center(val)
            d.show()
        #  Rollen
        if cmd == "R":
            d.clear()
```

```
            n=1
            pos=val.find(":")
            if pos != -1:
                n=int(val[:pos])
                val=val[pos+1:]
            print(val,n)
            d.roll(val,cnt=n,delay=30)
            d.clear()
        if cmd == "E":
            d.clear()
    except:
        pass
```

Dieses Tuple bekommt **doJobs**() als Eingabe. Wir entpacken das Tuple **cmd,val=resp** und wandeln alle alphabetischen Zeichen in Großschrift um. Dann klappern wir die Kommandoliste durch, die jederzeit erweitert werden kann.

L steht für linksbündige Ausgabe, C für zentrieren und R für rollen. Nach jedem Eingang eines Befehls wird zuerst die Anzeige gelöscht, dann der Textbefehl ausgeführt und mit **show**() die Anzeige aktualisiert.

Der Befehl R kann einen zweiten Doppelpunkt im Datenteil enthalten. Davor würde die Anzahl der Durchgänge stehen. Wir stellen n schon mal auf 1 und suchen nach dem zweiten Doppelpunkt.

Ist keiner enthalten, dann hat **pos** den Wert -1 und der anzuzeigende Text steht in **val**. Wird ein Doppelpunkt gefunden, dann steht vor dem Doppelpunkt die Zahl der Durchläufe und nach dem ":" folgt der Text bis zum Stringende von **val**.

Um nicht alles zweimal darstellen zu müssen, habe ich WLANdisplay.py nur mit der Klasse MATRIX umgesetzt, während APdisplay.py die Writer-Variante vertritt. APdisplay.py arbeitet mit einem Display aus einer Vierergruppe (32 x 8 Pixel), die WLAN-Variante steuert ein Display mit 128 x 8 Pixel aus 4 Vierergruppen.

Nach dem Herunterladen der beiden Dateien müssen Sie sich entscheiden, welcher davon Sie für den Gebrauch den Vorzug geben. Wenn es um einen ersten Test geht, dann schlage ich die WLAN-Variante vor, denn zu deren Test brauchen Sie den SHT21-Client noch nicht unbedingt in Funktion.

Öffnen Sie die Datei **WLANdisplay.py** in einem Editor-Fenster von Thonny und starten Sie sie mit F5.

Abbildung 10: WLANdisplay.py ist gestartet

Laden Sie bitte das Tool Packetsender herunter. Installieren brauchen Sie nichts, wenn Sie die verlinkte Portable-Version nehmen. Entpacken Sie die ZIP-Datei in ein beliebiges Verzeichnis und starten Sie aus dem Ordner **PacketSenderPortable** die Datei **packetsender.exe**.

Als Erstes ändern Sie die lokale Portnummer des PCs. File – Settings …

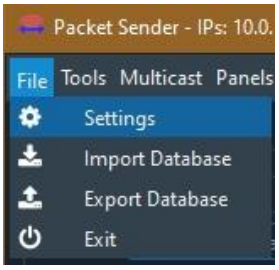

Abbildung 11: Einstellungen öffnen

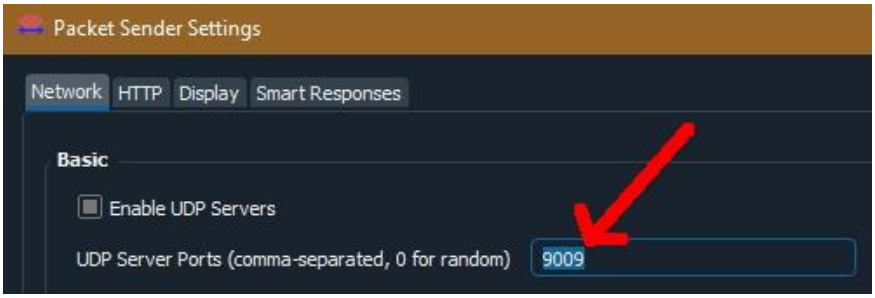

Abbildung 12: Portnummer angeben

Nach Angaben über die Zieladresse und das Protokoll schicken Sie die Nachricht an das Display ab (rote Punkte).

Abbildung 13: Weitere Angaben

Auf dem Display sollte jetzt linksbündig 1234 erscheinen. Versuchen Sie es auch einmal mit "C:Hallo auch!" oder "R:3:HEISSE SACHE". Wenn das alles funktioniert, dann ist es Zeit für ein anerkennendes Schulterklopfen.

Eine Kleinigkeit fehlt noch, und die kommt jetzt.

Der SHT21-Client

Nach der Trennung der Zuständigkeiten können wir aus dem ursprünglichen Programm **shtdisplay.py** für den Client alles entfernen, was mit der Ansteuerung des Displays zu tun hatte. Übrig bleibt die Abfrage des HTU21und ergänzt wird die schon bekannte Abteilung zur Verbindung mit einem Accesspoint. Der ist jetzt aber nicht mehr jener vom Router, sondern unsere Display-Einheit mit der IP 10.1.1.96. Nach den üblichen Verdächtigen, wie Pin, sleep, esp und so weiter, importieren wir noch SHT21, network und socket. Die Credentials und die Adressinformationen reihen sich danach ein. Die Taste an GPIO0 verschafft uns im Ernstfall einen geordneten Ausstieg aus der Mainloop. Der ESP8266-01 besitzt keine Tasten, also brauchen wir dafür das externe Tastenmodul.

```
from sht21 import SHT21

import network
import socket
mySSID = 'display'
myPass = 'guest'
myNetwork=("10.1.1.94","255.255.255.0","10.1.1.96","10.1.1.96")
myPort=9009
target=("10.1.1.96",9009)
taste=Pin(0,Pin.IN,Pin.PULL_UP)
```

Die Definition der Bus-Pins fällt um Etliches schlanker aus, beim Sender gibt es ja keinen SPI-Bus mehr.

```
chip=sys.platform
if chip == 'esp8266':
    # Pintranslator fuer ESP8266-Boards
    # LUA-Pins      D0 D1 D2 D3 D4 D5 D6 D7 D8
    # ESP8266 Pins 16  5  4  0  2 14 12 13 15
    #                 SC SD
```

```
#     SCL=Pin(5) # normaler 8266
#     SDA=Pin(4)
    SCL=Pin(0) # ESP8266-S01
    SDA=Pin(2)
elif chip == 'esp32':
    SCL=Pin(21)
    SDA=Pin(22)
else:
    raise OSError ("Unbekannter Port")

print("SCL: {}, SDA: {}".format(SCL,SDA))

i2c=SoftI2C(SCL,SDA)
sht=SHT21(i2c)

temperature=True # temp/hum switch
delay=3000 # ms
```

Dann instanziieren wir ein I2C-Objekt und übergeben es an das SHT21-Objekt **sht**. Das Flag **temperature** löst in der Mainloop das Versenden des Temperaturstrings an die Anzeigeeinheit aus und **delay** enthält die Intervalldauer für das Updaten der Messwerte.

getTemp() und **getHum**() holen, in schon bekannter Weise, aber dieses Mal getrennt, die Messwerte vom HTU21 und bereiten auch gleich die Strings für die Weitergabe an die Displayeinheit vor. Ich verwende dazu Formatstrings, wie an etlichen weiteren Stellen in den Programmen. Auf die Mystik, die dahintersteckt, will ich einmal kurz eingehen und einen Teil davon entzaubern.

Die eigentliche Formatanweisung steckt in dem Paar geschweifter Klammern {}. Außen herum können beliebige druckbare Zeichen stehen. Das Ganze wird in ein Paar von Hochkommas, " " oder ' ', eingeschlossen. Die Methode **format**() aus der Klasse **str** nimmt das Objekt aus der Parameterliste und fügt es in den Formatstring für die Ausgabe ein.

>>> "Temp {}".format(23.5)
'Temp 23.5'

>>> "Temp {}: {} C".format("innen",25.7)
'Temp innen: 25.7 C'

Verkettungen sind möglich. Die Reihenfolge der Argumente bestimmt die Reihenfolge für das Auffüllen der Platzhalter im Formatstring (automatic field numbering). Die Reihenfolge kann händisch verändert werden (manual field specification).

>>> "Temp {1} C, {0} ".format("innen",25.7)
'Temp 25.7 C, innen '

Die Verwendung von Namen und Variablen ist möglich.

>>> temp=23.8

>>> hum=45.3
>>> "Messwerte: {t} C, {h} %".format(h=hum,t=temp)
'Messwerte: 23.8 C, 45.3 %'

Ein **Doppelpunkt** trennt die Nummerierung oder die Namen von weiteren Formatangaben. Namen und Nummerierung können gemischt werden.

>>> temp=23.8762
>>> "Messwerte: {t:6.1f} C, {0:.3f} %".format(45.2,t=temp)
'Messwerte: 23.9 C, 45.200 %'

Die Temperatur wird hier auf eine minimale Breite von **6** Stellen durch Auffüllen mit führenden Leerzeichen (default) gebracht. Die Gleitkommazahl (**f**) wird auf eine Nachkommastelle (**.1**) gerundet. Die Zahl 45,2 steht an 0-ter Position in der Parameterliste und wird ohne Angabe einer minimalen Breite bündig an dem vorhergehenden Text ausgerichtet. Die 3 Nachkommastellen werden, falls nicht ausreichend vorhanden, durch Nullen aufgefüllt.

Für die Ausgabe eines Werts in anderen Zahlensystemen, als dem dezimalen, stehen entsprechende Symbole zur Verfügung, Beispiel Binärsystem.

Ausgabe als Binärzahl
>>> "{:b}".format(57)
'111001'

Ausgabe mit führenden Leerzeichen mit minimaler Breite von 8 Zeichen.
>>> "{:8b}".format(57)
' 111001'

Die Ausgabe mit führenden Nullen ist ideal für die Überprüfung von Registerinhalten während der Entwicklungs-Phase.

>>> "{:08b}".format(57)
'00111001'

>>> w=0xA2C
>>> w
2604
>>> "{:#06X}".format(w)
'0X0A2C'

Die Raute (#) führt zur Kennzeichnung als Hexadezimalwert bei der Ausgabe (**0X…**). Die minimale Breite von 6 lässt 4 Stellen für die Ziffernfolge 0A2C übrig, fehlende Stellen werden mit Nullen aufgefüllt (**06**). Das **X** lässt die hexadezimalen Ziffern A bis F als Großbuchstaben erscheinen.

Ich denke, fürs Erste reicht das, obwohl ich nur an der Oberfläche gekratzt habe. Zurück zum SHT-Client. Die Methode **doJobs**() bekommt den Messwertstring von **getTemp**() oder **getHum**(). Als Prefix wird das Kommando zum Zentrieren vorangestellt und das Ganze an die Adresse in **target** verschickt.

Die Funktion **TimeOut**() ist mein nicht blockierender Timer. Es ist eine sogenannte Closure, eine Funktion, welche in ihrem Inneren eine weitere Funktion definiert und statt eines Zahlenwerts eine Referenz auf diese Funktion zurückgibt. Normalerweise werden alle Variableninhalte, die innerhalb einer Funktion deklariert und verwendet wurden, beim Verlassen der Funktion eingestampft. Bei einer Closure bleiben alle Variablenreferenzen, die außerhalb der eingeschlossenen Funktion **compare**() definiert und innerhalb referenziert (aufgerufen) wurden nach dem Verlassen der äußeren Funktion **TimeOut**() erhalten. Das ermöglicht das periodische Abfragen des Funktionswerts von **compare**(), was uns mit **True** den Ablauf des Timers wissen lässt. Mehr über Closures erfahren Sie in dem PDF-Dokument Closures und Decorators.pdf.

```
def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_ms()-start) >= t
    return compare
```

Das Dictionary **connectStatus** liefert im Zusammenhang mit dem Aufruf der Methode **nic.status**() den Klartext zu deren Rückgabewert.

```
connectStatus = {
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
    5:    "GOT_IP"
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    }
```

Besonders beim ESP8266 ist es wichtig, das AP-Interface (Accesspoint-Interface) auszuschalten, wenn es nicht gebraucht wird.

```
# Unbedingt das AP-Interface ausschalten
nac=network.WLAN(network.AP_IF)
nac.active(False)
nac=None
```

Als Client brauchen wir das STA-Interface (Station-Interface), das wir auch gleich aktivieren.

```
# Wir erzeugen eine Netzwerk Interface-Instanz
nic = network.WLAN(network.STA_IF)
nic.active(False)

# Wir aktivieren das Netzwerk-Interface
nic.active(True)
```

Das Interface konfigurieren wir mit den weiter oben angegebenen Netzwerkdaten.
myNetwork=("10.1.1.94","255.255.255.0","10.1.1.96","10.1.1.96")
Die runden Klammern sind notwendig, denn die vier Strings stellen ein Tuple dar. Alle vier Angaben, die eigene IP-Adresse, die Netzwerkmaske, die Gateway-IP und die IP des DNS-Servers sind erforderlich. Bei unserer Insellösung gibt es kein Gateway und auch keinen DNS, es gibt auch keinen Internetzugang. Das Weglassen der Adressen würde aber zu einem Fehler und Programmabbruch führen, deswegen trage ich hier für beide die IP des Display-Accesspoints ein.

```
# Aufbau der Verbindung
# Wir setzen eine statische IP-Adresse
nic.ifconfig(myNetwork)
```

Für die Anmeldung am Accesspoint der Anzeigeeinheit übergeben wir die Credentials. Die Verwendung eines Passworts wird zwar nicht unterstützt, aber das Weglassen des Parameters **myPass** führt dennoch zu einer Fehlermeldung.

```
# Anmelden am WLAN-Router
nic.connect(mySSID, myPass)
```

Im Normalfall besteht an dieser Stelle noch keine Verbindung, **nic.isconnected**() liefert also **False** und mit dem **not** davor geht es deshalb in die Warteschleife, bis die Verbindung steht. Bis dahin wird jede Sekunde ein "." ausgegeben. Das **end=""** verhindert einen Zeilenwechsel.

```
if not nic.isconnected():
    # warten bis die Verbindung zum Accesspoint steht
    while not nic.isconnected():
        print("{}.".format(nic.status()),end='')
        sleep(1)
```

Wir lassen uns den Verbindungsstatus im Klartext sowie die Adressdaten anzeigen.

```
# Wenn verbunden, zeige Verbindungsstatus & Config-Daten
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
if nic.isconnected():
    # War die Konfiguration erfolgreich? Kontrolle
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
else:
    print("No AP found")
```

Dann bauen wir die UDP-Schnittstelle auf, so, wie weiter oben schon beschrieben.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
print("sending on port",myPort)
s.settimeout(0.1)
```

Die Mainloop schließt das Programm ab. Vor dem Eintritt starten wir den Timer **intervall**() mit der in **delay** definierten Ablaufzeit in Millisekunden. Die Funktion (!!!) **intervall**(), deren Bezeichner eine Referenz auf die Funktion **compare**() in **TimeOut**() enthält, gibt bis zum Ablauf der 3000 ms den Wert **False**, danach **True** zurück.

```
intervall=TimeOut(delay)
while 1:
    if intervall():
        if temperature:
            data=getTemp()
            temperature=False
        else:
            data=getHum()
            temperature=True
        intervall=TimeOut(delay)
        if data is not None:
            doJobs(data)
    if taste()==0:
        sys.exit()
        #reset()
```

In der Schleife prüfen wir auf den Rückgabewert. Bei False passiert nichts. Es wird nur noch die Taste abgefragt, deren Betätigung zum Programmende führt, alternativ zu einem Neustart.

Ist der Timer abgelaufen, gibt **intervall**() **True** zurück. Dann prüfen wir, ob das **temperature**-Flag gesetzt ist. In diesem Fall muss der Temperaturstring in die Variable **data** geholt werden. Das Flag wird auf **False** gesetzt, beim nächsten Mal ist dann die relative Luftfeuchte im **else**-Zweig dran. Dort wird das Flag wieder auf **True** gebracht, womit sich der Kreis schließt.

In jedem Fall muss der Timer neu gestartet werden. Wenn der Inhalt der Variablen **data** gültig ist, erledigt **doJobs**() die Sendung ans Display.

Wie bei der Anzeigeeinheit ist es nötig, das gesamte Programm **sht21client.py** als **boot.py** in den Flash des ESP8266-01 hochzuladen, damit ein autonomer Betrieb ohne PC stattfinden kann. Eine Möglichkeit der Durchführung ist die, **sht21client**.py über **File – Save as…** unter dem Namen **boot**.py neu abzuspeichern. Dann laden Sie boot.py wie üblich zum ESP8266 hoch.

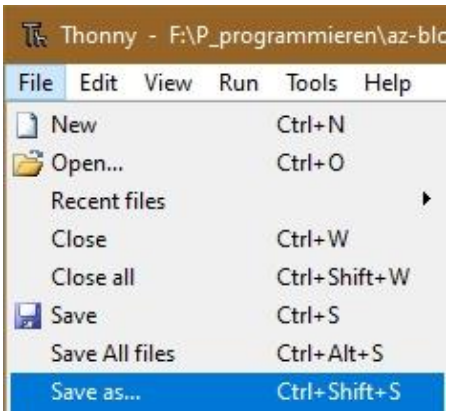

Abbildung 14: Abspeichern unter neuem Namen

Wenn jetzt die Displayeinheit noch läuft, sollten die Daten vom Client dort in der Anzeige erscheinen. Ich hoffe, Sie müssen nicht uchen, sondern können sich entspannt zurücklehnen und bei einer schönen Tasse Kaffee Ihr Werk bewundern. Wenn Sie bei uchen etwas unschlüssig über die Bedeutung dieses Wortes sind, versuchen Sie's doch mal mit dem Prefix "s" oder "fl".

Ausblick

In der nächsten Folge werden wir den Onlinedienst **Open Weather Map** (OWM) anzapfen, um mit diesen Daten und unserer Matrixanzeige einen Wetterticker zu programmieren. OWM stellt kostenlose Accounts zur Verfügung. Nach der Anmeldung bekommt man einen User-Schlüssel, mit dem man verschiedene Wetterdaten abfragen kann. Neugierig geworden?

Na dann, bis bald!