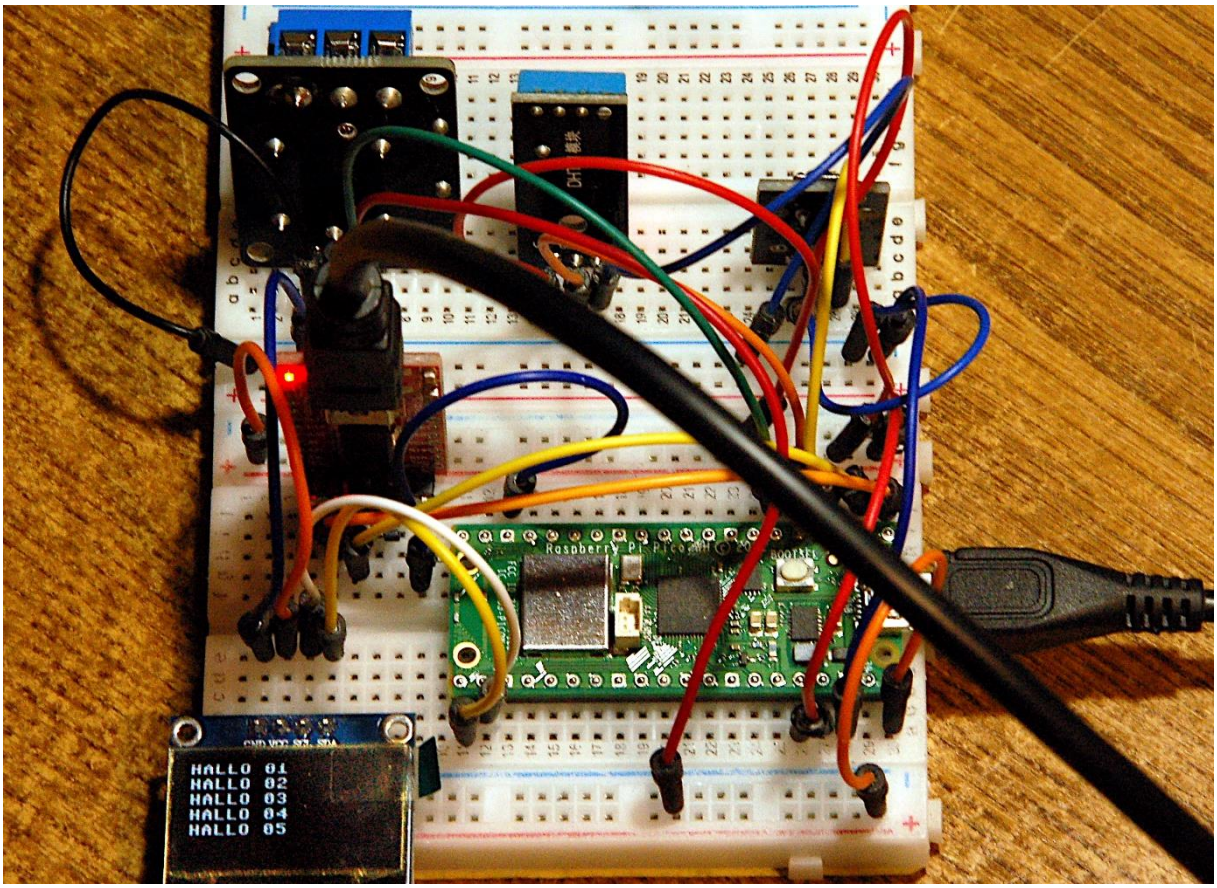

Raspberry Pi Pico W als Messknecht

Dieser Beitrag ist auch als PDF-Dokument verfügbar.

Dass Python auf dem PC mehr kann, als MicroPython auf einem ESP32, ESP8266 oder Raspberry Pi Pico habe ich in der letzten Blog-Folge schon gesagt. Aber Python kann noch bedeutend mehr. Auf dem PC kann Python auch Fenster oder besser gesagt Windows mit den bekannten Steuerelementen wie Labels, Eingabefelder, Buttons und so weiter. Im MIT App Inventor 2 habe ich auch schon in einigen Beiträgen mit diesen Elementen gearbeitet, um Apps zu erstellen, die per WLAN mit den Controllern der ESP-Familie kommunizieren können. In Python gibt es ein GUI-Framework (Grafics-User-Interface-Werkzeug) mit dessen Hilfe sich Fenster mit den üblichen Bedienelementen erstellen lassen. Es heißt **Tkinter** und ist als Modul bereits im Kern von C-Python, kurz Python, enthalten. Die Steuerelemente heißen dort Widgets. Wir werden uns heute anschauen, wie wir damit eine Windows-App mit einem Window und entsprechenden Widgets erstellen können. Als Grundlage dient das zuletzt verwendete Programm ersterKontakt.py, mit dem wir Messwerte vom Raspberry Pi Pico über die serielle Verbindung angefordert haben. Diese Anwendung werden wir noch um ein Relais und ein OLED-Display erweitern. Stoßen Sie mit mir vor in die Welt der grafischen Oberflächen in einer neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

Heute:

Grafische Oberflächen

Was bisher geschah

Die Schaltung für unseren Aufbau sah zuletzt so aus.

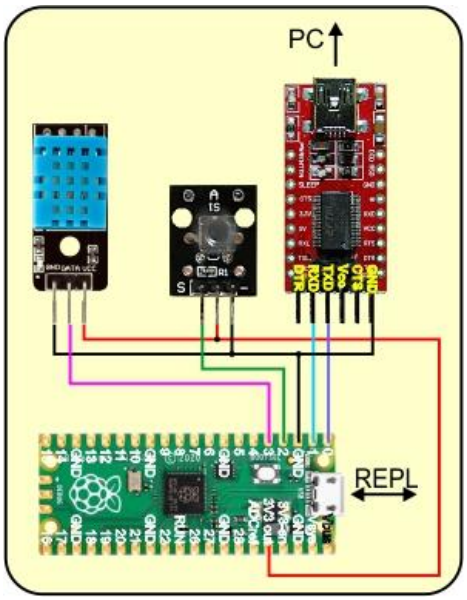

Abbildung 1: Raspberry Pi Pico als Messstation

Ich habe sie nun um ein Relais und eine OLED-Anzeige erweitert.

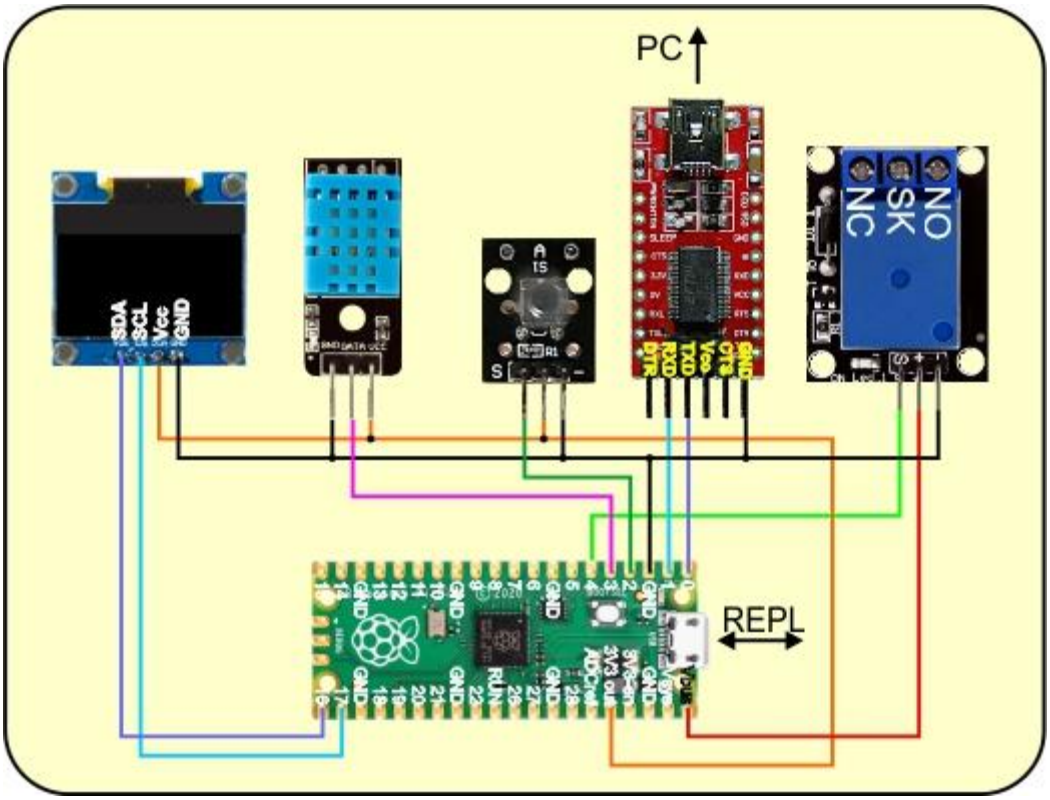

Abbildung 2: Raspberry Pi Pico als Messstation erweitert

Daraus ergibt sich folgende Teileliste für die Durchführung der nun geplanten Aktionen.

Hardware

1	Raspberry Pi Pico RP2040 Mikrocontroller-Board
1	KY-015 DHT 11 Temperatursensor Modul
1	KY-004 Taster Modul
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	1-Relais 5V KY-019 Modul High-Level-Trigger
1	FT232-AZ USB zu TTL Serial Adapter für 3,3V und 5V
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
diverse	Jumper Wire Kabel 3 x 40 STK
optional	Logic Analyzer

Die Softwarewerkzeuge sowie Firmware-Links und die Programme dazu finden Sie im nächsten Kapitel.

Die Software zum Projekt

Fürs Flashen und die Programmierung des ESP32/Raspberry Pi Pico:

Thonny oder
Betriebs-Software Logic 2 von SALEAE
PuTTY ein Terminalprogramm

Verwendete Firmware für einen ESP32:

Micropython Firmware Download
v1.19.1 (2022-06-18) .bin

Verwendete Firmware für einen ESP8266:

v1.19.1 (2022-06-18) .bin

Verwendete Firmware für den Raspberry Pi Pico:

RPI_PICO-20240105-v1.23.1.uf2

Verwendete Firmware für den Raspberry Pi Pico W:

RPI_PICO_W-20240105-v1.23.1.uf2

Die MicroPython-Programme zum Projekt:

ssd1306.py: Hardwaretreiber für das Display
oled.py: API für das Display
satellite.py Betriebssoftware für den Raspberry Pi Pico
satellite_new.py neue Betriebssoftware für den Raspberry Pi Pico
ersterKontakt.py altes Steuerprogramm auf dem PC
timeout.py Modul mit nichtblockierenden Software-Timern
gui.py Windows-App als neues Frontend auf dem PC

MicroPython - Sprache - Module und Programme

ESP32 und ESP8266 und Raspberry Pi Pico

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 07.01.2024) auf den ESP-Chip gebrannt wird. Wie Sie den **Raspberry Pi Pico** einsatzbereit kriegen, finden Sie hier.

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.

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.

Raspberry Pi Pico

Eine Beschreibung, wie Sie MicroPython als Firmware auf den Raspberry Pi Pico bekommen, finden Sie in MicroPython auf dem Raspberry Pi Pico.pdf.

Die Programme

Python auf dem PC

Werfen wir zu Beginn noch einmal einen Blick auf die Programme aus der letzten Folge: satellite.py und ersterKontakt.py. Das Programm **satellite.py** auf dem Raspi werden wir erweitern, wenn die grafische Oberfläche gui.py auf dem PC so weit fertig ist. Diese werden wir an dem Programm **ersterKontakt.py** ausrichten und erweitern.

```
# ersterKontakt.py
import serial
from time import sleep
from sys import exit

uart=serial.Serial("COM9",9600,parity=serial.PARITY_NONE,
timeout=0)

while 1:
    wahl=input("q|Q Query values\ne|E Exit program ")
    if wahl.upper()=="Q":
        uart.write(b"sendValues\n")
        sleep(2)
        temp=uart.readline().decode().strip("\n")
        hum= uart.readline().decode().strip("\n")
        print(temp,hum,"\n")
    elif wahl.upper() == "E":
        uart.close()
        print("\nProgramm beendet")
        exit()
```

Die fett markierte Eingabe-Zeile entscheidet darüber, welche Aufträge der Raspi bekommen soll. Zur Information welche Optionen zur Verfügung stehen, dient das "Menü", das wir als Eingabeprompt anzeigen lassen. Wollen wir weitere Punkte ergänzen, geht die Übersicht ziemlich schnell flöten. Deshalb bringt eine grafische Oberfläche die praktikabelste Lösung, und die ist leicht umzusetzen, zumal Tkinter alles beinhaltet, was das Programmiererherz zum Erstellen einer grafischen Oberfläche begehrt.

Starten wir also durch mit **IDLE** und **Tkinter**. Ich importiere das Modul **tkinter** und weise den Alias **tk** zu, weil ich nicht so viel tippen möchte.

```
>>> import tkinter as tk
>>> win=tk.Tk()
```

Mit dem Abschicken des zweiten Kommandos erscheint auch schon unser erstes Fenster, das wir jetzt bestücken werden. Das geht allerdings nicht so schön, wie im MIT App Inventor 2 (AI2), mit grafischen Bausteinen via drag and drop, sondern über die in Python üblichen Programmzeilen oder über REPL. Eingaben im Terminal werden hier fett, Ausgaben vom Controller kursiv formatiert.

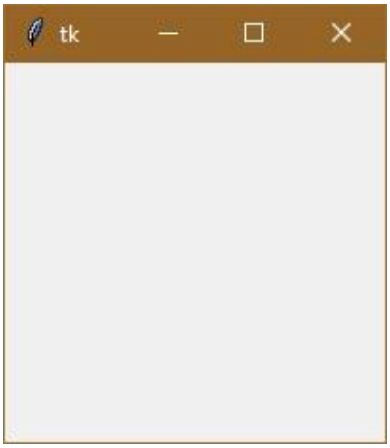

Abbildung 3: Unser erstes Fenster

Über die Methode **geometry**() können wir schon einmal die Größe des Fensters einstellen. Es kriegt eine Titelzeile und ein Limit für die Abmaße

```
>>> win.geometry("300x250")
>>> win.title("Steuerung Kellermodul")
>>> win.maxsize(width=500,height=250)
```

Der schalenmäßige Aufbau mit Frames in Tkinter entspricht den Arrangements von AI2 und erlaubt uns die Gliederung des Fensters.

Nun werden die Widgets erstellt. Wir brauchen Frames, Labels, Buttons und ein Eingabefeld, ein Entry, außerdem noch ein paar StringVars, Stringvariablen der besonderen Art. Damit die Tipparbeit nicht für die Katz ist, legen wir ein neues File an und schreiben alles gleich in eine Programmdatei. Wir speichern es schon mal unter dem Namen **gui.py** im Arbeitsverzeichnis ab.

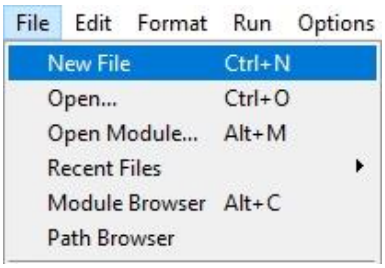

Abbildung 4: In IDLE ein neues File anlegen

Alles startet mit dem Import der benötigten Module.

```
# gui.py
# Bedienoberfläche fuer die Rspberry Pi Pico Steuerung
#
import tkinter as tk
import serial
from time import sleep
```

Dann instanziieren wir das UART-Objekt und legen den Schaltzustand, **relState**, für das Relais fest.

```
uart=serial.Serial("COM9",9600,
                   parity=serial.PARITY_NONE, timeout=0)
relState=0
```

Die Definition des Fenstercontainers **win** haben wir schon besprochen. Um die Fenstergröße übersichtlich an einer Stelle im Programm zu haben, definieren wir dafür die Variable **geo**. Breite und Höhe erhalten wir durch Splitten des Strings am "x".

```
# Window erzeugen
geo="300x250"
b,h=geo.split("x")
win=tk.Tk()
win.geometry(geo)
win.title("Steuerung Kellermodul")
win.maxsize(width=500,height=250)
```

Nun folgen die ersten beiden Widgets, zwei Frames. Damit die Schalenkonstruktion korrekt erfolgt und unseren Vorstellungen entspricht, müssen wir als Erstes die Container für den Rest der Steuerelemente schaffen und zwar von oben nach unten im Programm. Das schreibt uns der Packer vor, der die Objekte auf der Fensterfläche platziert. Jedes Objekt, also auch jeder der Frames, braucht mindestens zwei Zeilen. In der ersten wird das Objekt erzeugt, in der zweiten wird es platziert. Bei der Instanziierung geben wir in der Parameterliste immer zuerst das Vater-Objekt, den Master an, in dem das Objekt erscheinen soll. Die eingebetteten Objekte nennt man Slaves.

```
main=tk.Frame(master=win, width=int(b),
              height=int(h), bg="#74CFED")
main.pack(padx=5,pady=5, fill="both", expand=1)

slave=tk.Frame(master=win, width=int(b),
               height=50, bg="#a47F8D")
slave.pack(padx=5,pady=5, fill="both", expand=1)
```

Der Frame **main** hat als Master das Fenster **win**, erhält seine Breite und Höhe letztlich von der Definition in **geo** und bekommt als Hintergrundfarbe ein sanftes Hellblau über den RGB-Code, R:0x74, G:0xCF und B:0xED.

Der Packer wird angewiesen, die Frames mit einem Rand von 5 Pixeln in x- und y-Richtung auszustatten und den Frame in beide Richtungen nach Möglichkeit auszudehnen. Wenn wir jetzt das Programm starten, verlangt Python die

Speicherung. Wir bestätigen mit OK und sehen ein hellblaues Fenster mit hellem Rand. Wo ist der zweite Frame geblieben? Nun die Frames haben ja noch keinen Inhalt und weil der erste Frame so hoch ist wie das Fenster, hat er den zweiten ganz einfach von der Tanzfläche verdrängt. Aber keine Sorge, das gibt sich schon noch.

Dann legen wir in **main** ein Anzeigefeld, ein Label an. Es soll sich über die gesamte Breite des Frames erstrecken (fill="x") und 5 Pixel Rand zu ihm einhalten. Der Rand der Fläche wird dreidimensional dargestellt (tk.RAISED). Das Label wird im Frame **main** erscheinen.

```
headline=tk.Label(main,text="RS232-Remote Control",
                  relief=tk.RAISED)
headline.pack(padx=5,pady=5, fill="x")
```

Die erste Stringvariable heißt **t**. Felder, deren Text-Eigenschaft diese Stringvariable zugewiesen wird, übernehmen den Inhalt der Stringvariablen sofort, sobald deren Wert geändert wird. Sie Stringvariable **t** wird mit dem Text **Temperatur** vorbelegt.

```
t=tk.StringVar()
t.set("Temperatur")
```

Danach erzeugen wir gleich das Label in dem der Temperaturwert angezeigt werden soll. Wir weisen die Stringvariable t zu und machen den Hintergrund orange. Der Text soll vom Rand 5 Pixel Abstand haben (ipadx=5, ipady=5).

```
temp=tk.Label(main, textvariable=t, relief=tk.RAISED)
temp["bg"]="orange"
temp.pack(ipadx=5,ipady=5)
```

In der gleichen Vorgehensweise werden Stringvariable **h** und Label **hum** angelegt.

```
h=tk.StringVar()
h.set("Rel. Luftfeuchte")
hum=tk.Label(main, textvariable=h, relief=tk.RAISED)
hum["bg"]="cyan"
hum.pack(ipadx=5,ipady=5)
```

Für die Eingabe einer Nachricht brauchen wir ein **Entry**-Objekt und die dazugehörige Stringvariable **m**.

```
m=tk.StringVar()
m.set("")
mesg=tk.Entry(main, textvariable=m, relief=tk.FLAT)
mesg.pack(padx=5,pady=5,fill="x")
```

Damit der Benutzer weiß, was er mit dem weißen Feld tun soll, geben wir Ihm mit dem Label **reply** einen Hinweis. Damit das Label ohne Hintergrund erscheint, färben wir diesen mit der Hintergrundfarbe des Frames **main** ein. Das Label wird später die Antwort vom Raspberry Pi Pico anzeigen.

```
r=tk.StringVar()
r.set("Nachricht eingeben und OK")
reply=tk.Label(main,textvariable=r)
reply["bg"]="#74CFED"
reply["fg"]="black"
reply.pack(padx=5,pady=5)
```

In der nächsten Zeile sollen drei Buttons nebeneinander eingebaut werden und zwar von rechts nach links (side="right"). Wir beginnen mit dem Button, bei dessen Click die Werte vom Raspberry Pi Pico geholt werden sollen. Für einen Button brauchen wir eine sogenannte Callback-Routine, die ausgeführt wird, wenn der Button angeklickt wird. In unserem Fall ist das die Funktion **getValues**(), die wir gleich im Anschluss definieren werden. Dem Parameter **command** weisen wir den Bezeichner dieser Funktion zu.

```
query=tk.Button(main,text="Werte holen", bg="blue",
                fg="Yellow", command=getValues)
query.pack(side="right", expand=1,fill="x")
```

Wenn wir jetzt das Programm ausführen würden, bekämen wir im Terminalfenster eine Fehlermeldung, dass der Name **getValues** nicht definiert ist.

NameError: name 'getValues' is not defined

Dieselbe Fehlermeldung bekämen wir, wenn wir die Deklaration **nach** der Zuweisung der Funktion an **command** in den Programmtext schreiben würden. Funktionen schreibe ich daher, mit wenigen begründeten Ausnahmen, stets an den Anfang eines Programms. Gehen wir daher in Zeile 12 unter das Statement **relState=0**, um die Funktion zu codieren.

```
relState=0

# Callback-Routinen definieren
def getValues():
    uart.write(b"sendValues\n")
    sleep(3)
    t.set(uart.readline().decode().strip("\n")+" °C")
    h.set(uart.readline().decode().strip("\n")+" %")
    print(t.get(),h.get(),"\n")
```

Die Funktion muss das Codewort **sendValues** an den Raspberry Pi Pico schicken und die Antwort abwarten (sleep(3)). Die nächsten beiden Zeilen übernehmen wir aus dem Programm **ersterKontakt.py**. Deren Zustandekommen ist in der letzten Blogfolge genau erklärt. Wir ergänzen noch die Maßeinheiten und weisen die Strings den Stringvariablen **t** und **h** via der **set**-Methode zu. Im Terminal können wir, zur Kontrolle, bei der Programmentwicklung die Werte auslesen (t.get()) und in REPL ausgeben. Wir testen das Programm schon mal mit F5 – OK .

Abbildung 5: Der erste Button wird getestet

Ist der Aufbau mit dem Raspberry Pi Pico am PC mit dem USB-Kabel verbunden? Dann können wir die Wirkung eines Klicks auf den Button testen, wenn wir zunächst auf dem Raspi das Programm satellite.py gestartet haben. Ein paar Sekunden, bedingt durch die eingebauten Wartezeiten, ergibt sich eine Darstellung wie in Abbildung 5.

Lassen wir nun zwei weitere Buttons im Design folgen. Der Packer wird sie jeweils links neben dem zuvor deklarierten Button ansiedeln. Mit **OK** soll der Text im Entry **mesg** übernommen und an den Raspberry Pi Pico gesendet werden und mit **relais** werden wir das Relais am Raspi ein- und ausschalten. Die drei Buttons nehmen jetzt zusammen die gesamte Breite des Fensters ein.

```
OK=tk.Button(main,text="OK", bg="#108000",
                fg="Yellow", command=sendMessage)
OK.pack(side="right",  expand=1, fill="x")

relais=tk.Button(main,text="Relais AUS", bg="blue",
                fg="Yellow", command=switch)
relais.pack(side="left", expand=1,fill="x")
```

Die beiden Callback-Routinen **sendMessage**() und **switch**() schreiben wir unterhalb dem Code von **getValues**() in den Programmtext.

```
def sendMessage():
    text="message:"+m.get()+"\n"
    uart.write(text.encode())
    sleep(2)
    antwort=uart.readline().decode().strip("\n")
    r.set(antwort)
```

Den eingegebenen Text holen wir durch **get**() aus der Stringvariablen **m** ab und setzen ihn zu einem String-Objekt, text, zusammen. Es wird als bytes-Objekt codiert und an den Raspberry Pi Pico geschickt. Der soll seinerseits nun eine Antwort zurücksenden, die wir in der bekannten Weise, lesen, decodieren und im Label **reply** anzeigen lassen.

In switch wird die Zustandsvariable **relstate** verändert. Diese Änderung muss im Hauptprogramm gemerkt werden, daher deklarieren wir die Variable als **global**. Beim Klicken des Buttons soll der Schaltzustand geändert werden, aus 1 wir 0 und

umgekehrt. Angezeigt wird der Zustand durch die Beschriftung des Buttons und durch seine Hintergrundfarbe.

Wir bauen also einen Kommando-String zusammen, der aus dem Codewort **relais,** einem Doppelpunkt und dem Schatzustand besteht mit einem abschließenden Newline, **\n**. Wie bei **sendMessage**() steht das Kommando links und der Datenteil recht von einem Doppelpunkt. Der Raspberry Pi Pico kann die Anteile der Nachricht durch Splitten des Strings wieder aufdröseln, dazu später mehr.

```
def switch():
    global relState
    if relState==1:
        relState=0
    else:
        relState=1
    text="relais:"+str(relState)+"\n"
    uart.write(text.encode())
    sleep(2)
    antwort=uart.readline().decode().strip("\n")[-1]
    print(antwort)
    if antwort == "1":
        relais["text"]="Relais AN"
        relais["bg"]="red"
    else:
        relais["text"]="Relais AUS"
        relais["bg"]="blue"
```

Senden, warten, Antwort lesen und das letzte Zeichen im Antwortstring isolieren ([-1]). Je nachdem werden die Hintergrundfarbe und der Text des Buttons **relais** eingestellt.

Hängen wir doch gleich noch die Callback-Routinen für die beiden letzten Buttons hier mit dran. **Goodbye**() beendet das Programm und schließt das Fenster **win**, nachdem die serielle Verbindung gekappt und der UART COM9 freigegeben wurde.

```
def goodbye():
    uart.close()
    win.destroy()

def clear():
    uart.read(256)
```

Während der Entwicklungsphase gab es diverse Male fatal störende Reste im Empfangspuffer von COM9. Um den Puffer zu leeren, lesen wir einfach alle 256 möglichen Zeichen aus. Hierbei ist es wichtig, dass beim Öffnen der Verbindung der Parameter **timeout** auf 0 gesetzt wird, damit die Empfangsschleife den Programmlauf nicht blockiert.

Die Buttons **quitApp** und **clearBuffer** platzieren wir im zweiten Frame, in **slave**.

```
quitApp=tk.Button(slave,text="Close UART and QUIT",
bg="white",
                  fg="red", command=goodbye)
quitApp.pack(padx=5,pady=5, side="right", expand=1,fill="x")

clearBuffer=tk.Button(slave,text="Buffer leeren", bg="white",
                  fg="blue", command=clear)
clearBuffer.pack(padx=5,pady=5, expand=1,fill="x")
```

Damit sind wir mit dem Programm auf dem PC fast am Ende, es fehlt nur noch der Aufruf der Mainloop des Fensters.

```
win.mainloop()
```

Das Dienstprogramm auf dem Raspberry Pi Pico

Das Programm aus der vorangegangenen Folge hat für das Abholen der Messwerte gute Dienste geleistet, muss aber für die erweiterten Funktionen in **gui.py** etwas aufgemufft werden. Das neue Programm soll satellite_new.py heißen.

```
# satellite_new.py
# for raspberry pi pico
#
from timeout import TimeOutMs
from dht import DHT11
from machine import Pin, UART, SoftI2C
from sys import exit
from gc import collect
from oled import OLED

taste= Pin(2,Pin.IN,Pin.PULL_UP)
relais=Pin(4,Pin.OUT, value=0)

dht=DHT11(Pin(3))
u = UART(0, baudrate=9600)

i2c=SoftI2C(scl=17,sda=16,freq=100000)
d=OLED(i2c,heightw=64)

lines=[]

def warten(delay):
    genugGewartet=TimeOutMs(delay)
    while not genugGewartet():
        if taste.value() == 0:
            u.deinit()
            print("Programm beendet")
            exit()

def messen():
    dht.measure()
```

```
    warten(1000)
    dht.measure()
    temp=dht.temperature()
    hum=dht.humidity()
    print("Temperatur: {}; Rel. Luftfeuchte: {}". \
          format(temp,hum))
    u.write(("Temperatur: {}\nR. Luftfeuchte: {}\n".\
             format(temp,hum)).encode())

def nachrichtAnzeigen(data):
    u.write("Nachricht empfangen "+data+"\n")
    print(data)
    lines.append(data)
    while len(lines) >= 7:
        lines.pop(0)
    for i in range(len(lines)):
        d.writeAt(lines[i],0,i)


dht.measure()
temp=dht.temperature()
hum=dht.humidity()
print("Temperatur: {}; Rel. Luftfeuchte: {}".format(temp,hum))
# exit()
while 1:
    while not u.any():
        if taste.value() == 0:
            u.deinit()
            print("Programm beendet")
            exit()
    warten(1000)
    rec=u.readline().decode().strip("\n")
    print("rec:",rec)
    pos=rec.find(":")
    if pos == -1: # einfacher Befehl
        if rec=="sendValues":
            messen()
    else:
        command,data=rec.split(":")
        data=data.strip("\n")
        if command=="message":
            print("data",data)
            nachrichtAnzeigen(data)
        elif command=="relais":
            state=int(data)
            relais.value(state)
            u.write("Relais-Status ist {}".format(state))

#     except KeyboardInterrupt:
#         u.deinit()
```

Wie arbeitet das Programm?

Wir importieren den Millisekunden-Timer vom Modul **timeout**, das zuvor in den Flash des Raspberry Pi Pico hochgeladen werden muss. Dasselbe gilt für die Dateien **oled.py** und **ssd1306.py**. Auf letzteres Modul greift die Klasse **OLED** zurück, die für die Ansteuerung des Displays zuständig ist. Alle weiteren Module wohnen bereits im Kern von MicroPython. Die Namen der Klassen und Funktionen verraten deren Aufgaben.

```
from timeout import TimeOutMs
from oled import OLED
from dht import DHT11
from machine import Pin, UART, SoftI2C
from sys import exit
from gc import collect
```

Pin GP2 liest als Eingang mit Pullup-Widerstand die Taste ein und an GP4 geben wir die Signalpegel für die Relaissteuerung aus.

```
taste= Pin(2,Pin.IN,Pin.PULL_UP)
relais=Pin(4,Pin.OUT, value=0)
```

An GP3 habe ich das DHT11-Sensormodul angeschlossen. Der UART liegt standardmäßig an GP0 (TX) und GP1 (RX). Daher bedarf es keiner Nennung der Anschlüsse in der Parameterliste. Die Übertragungsgeschwindigkeit wird auf 9600 Baud festgelegt.

```
dht=DHT11(Pin(3))
u = UART(0, baudrate=9600)
```

Für das OLED-Display instanziieren wir den I2C-Bus an GP16 und GP17.

```
i2c=SoftI2C(scl=17,sda=16,freq=100000)
d=OLED(i2c,heightw=64)
```

Die Liste für die künftigen Display-Ausgaben wird erzeugt und geleert.

```
lines=[]
```

Dann deklarieren wir drei Funktionen, **warten**(), **messen**() und **nachrichtAnzeigen**(). Für diese Routinen ist der Name Programm.

```
def warten(delay):
    genugGewartet=TimeOutMs(delay)
    while not genugGewartet():
        if taste.value() == 0:
            u.deinit()
            print("Programm beendet")
            exit()
```

Die Funktion **warten**() verwendet die Closure **compare**(), die in der umgebenden Funktion **TimeOutMS**() deklariert ist, unter dem Alias **genugGewartet**, um nicht nur

Zeit totzuschlagen, sondern gleichzeitig die Betätigung des Tasters zu überprüfen. Die Wartezeit in Millisekunden wird im Parameter **delay** übergeben. **genugGewartet**() liefert True zurück, wenn diese Zeit um ist. Dann wird die serielle Verbindung geschlossen und das Programm mit **exit**() beendet.

```
def messen():
    dht.measure()
    warten(1000)
    dht.measure()
    temp=dht.temperature()
    hum=dht.humidity()
    print("Temperatur: {}; Rel. Luftfeuchte: {}". \
          format(temp,hum))
    u.write(("Temperatur: {}\nR. Luftfeuchte: {}\n".\
             format(temp,hum)).encode())
```

In **messen**() geben wir sofort einen Messauftrag an den DHT11. Nach einer Sekunde wiederholen wir das. Wird nämlich in der Hauptschleife das Eintreffen von Zeichen festgestellt, dann kann das seit dem letzten Zugriff schon eine Weile her sein. Der im DHT11 gespeicherte Wert ist dann schon fossil und entspricht gegebenenfalls nicht mehr der momentanen Klimasituation. Deshalb holen wir mit dem zweiten Aufruf den aktuellen Wert. Die Wartezeit dazwischen ist notwendig, weil der Sensor eine Weile braucht, um den Wert bereitzustellen. In dieser Zeit ist er nicht ansprechbar. Stören wir ihn bei seiner Arbeit durch zu rasch aufeinanderfolgende Anfragen sagt er uns das mit einer Fehlermeldung:

>>> dht.measure(); dht.temperature(); dht.humidity()
19
43
>>> dht.measure(); dht.temperature(); dht.humidity()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "dht.py", line 1, in measure
OSError: [Errno 110] ETIMEDOUT

Sonst holen wir die Messwerte ab und bauen Strings für die Ausgabe in REPL und Rückantwort an den PC zusammen.

```
def nachrichtAnzeigen(data):
    u.write("Nachricht empfangen "+data+"\n")
    lines.append(data)
    while len(lines) >= 7:
        lines.pop(0)
    for i in range(len(lines)):
        d.writeAt(lines[i],0,i,False)
    d.show()
```

In der Hauptschleife wird der empfangene String geparst und in Kommando- und Datenanteil aufgespaltet. Letzterer wird in **data** an die Funktion **nachrichtAnzeigen**() übergeben. Den String hängen wir an die Liste empfangener Texte hinten an. Wenn nun die Liste **lines** mehr als sechs Einträge hat, zwicken wir den Anfang, das nullte

Element, ab. Danach wird der Listeninhalt in den Anzeigepuffer geschrieben und abschließend mit **show**() zum Display geschickt.

```
dht.measure()
temp=dht.temperature()
hum=dht.humidity()
print("Temperatur: {}; Rel. Luftfeuchte: {}".format(temp,hum))
```

Vor dem Betreten der Hauptschleife lösen wir eine Probemessung aus, um die Funktion des Sensors zu testen.

In der Mainloop warten wir brav auf das Eintreffen von Bytes. Währenddessen haben wir nichts anderes zu tun, als auf die Abbruchtaste zu warten.

```
while 1:
    while not u.any():
        if taste.value() == 0:
            u.deinit()
            print("Programm beendet")
            exit()
```

Sind die ersten Bytes eingetrudelt, warten wir noch, bis wirklich alles da ist. Wir lesen das bytes-Objekt ein, decodieren es in einen String, von dem wir den Linefeed-Schwanz abzwicken.

```
    warten(1000)
    rec=u.readline().decode().strip("\n")
    print("rec:",rec)
```

Findet sich kein ":" in **rec**, dann handelt es sich um einen einfachen Befehl, ohne Datenanteil. **sendValues** ist so einer. Wenn der angekommen ist, geben wir das Ruder an die Funktion **messen**() weiter.

```
    pos=rec.find(":")
    if pos == -1: # einfacher Befehl
        if rec=="sendValues":
            messen()
```

Wurde ein Doppelpunkt gefunden, dann trennen wir den String in den Kommando- und den Datenteil auf. Den Anzeigetext schicken wir an die Anzeigefunktion weiter.

```
    else:
        command,data=rec.split(":")
        data=data.strip("\n")
        if command=="message":
            print("data",data)
            nachrichtAnzeigen(data)
```

Um das Relais kümmern wir uns im **elif**-Block. Den String im Datenteil wandeln wir in eine Ganzzahl um und steuern damit den Relaisausgang an. Den aktuellen Zustand schicken wir an den PC zurück.

```
        elif command=="relais":
            state=int(data)
            relais.value(state)
            u.write("Relais-Status ist {}".format(state))
```

Nun kann die gesamte Geschichte in Betrieb gehen. Starten wir das Programm **satellite_new.py** im Raspberry Pi Pico und **gui.py** auf dem PC. Das Ergebnis des Tests könnte dann etwa so ausschauen wie in Abbildung 6.

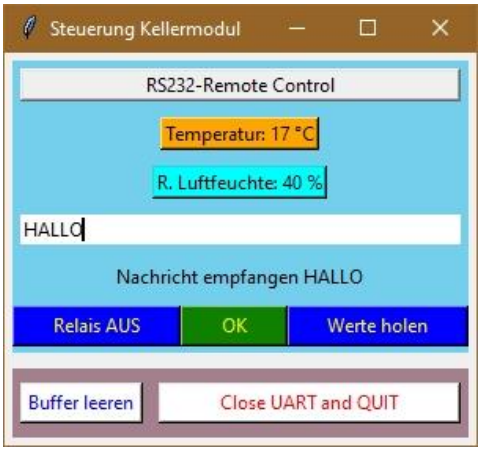

Abbildung 6: RS232-Gui im Einsatz

Ausblick:

Beide Teile des Projekts lassen sich natürlich erweitern. Das Programm auf dem Raspberry Pi Pico läuft mit geänderten Pin-Zuweisungen auch auf einem ESP8266 oder ESP32. Ich werde in einem der nächsten Beiträge die serielle Verbindung zwischen einem dieser Controller oder einem Raspberry Pi Pico nutzen, um ein Lernspiel und einen MP3-Player zu realisieren, denn das MP3-Modul DF-Player, wird über RS232 angesprochen.

Die WLAN-Eigenschaften dieser Controller lässt sich ferner nutzen, um die Daten über Funk zu übertragen und dann über RS232 weiterzuleiten oder umgekehrt. Auch auf dem PC kann man Netzwerkverbindungen verwenden, um zusammen mit den Grafikeigenschaften von Tkinter Applikationen zu bauen, die zum Beispiel Daten von einem Messknecht empfangen und in eine Datei schreiben, die von einer Tabellenkalkulation weiterverarbeitet werden kann. Sogar eine grafische dynamische Auswertung von Messreihen in Python ist mit **numpi** gut machbar.

Genug, genug, meinen Sie? OK, dann bis demnächst!