# Funkverbindung zu WLAN-Routern oder Accesspoints herstellen

## Objekte und Funktionen

Folgende Module, Objekte und Methoden sind zu importieren. Die Anzeigentreiber werden weiter unten durch ein if-elif-Konstrukt behandelt.

Die Datei **credentials_poly.py** ist mit eigenen Verbindungsdaten zu füttern und dann in den Flash des Controllers hochzuladen.

Im Fall eines OLED-Displays muss zusätzlich zum Modul **oled.py** der Hardwaretreiber **ssd1306.py** in den Flash des Controllers hochgeladen werden. Ein LCD braucht die Treiber **lcd.py** und **hd44780.py** im Flash.

```
from time import sleep,sleep_ms
import network, socket
from credentials_poly import creds
from sys import exit, platform
```

## Einbinden eines Displays

Display ja oder nein

```
displayWanted=False
displayPresent=False
```

Entscheidung über die Art des Displays.

```
lcdUsed = False
oledUsed = not lcdUsed

if displayWanted:
    if platform == "esp8266":
        try:
            i2c=SoftI2C(scl=Pin(5),sda=Pin(4),freq=100000)
            displayPresent=True
        except:
            pass
        freq(160000000)
    elif platform == "esp32":
        try:
            i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=100000)
            displayPresent=True
        except:
            pass
        freq(240000000)
    else:
        raise UnkownPortError()
```

Im Fall eines OLED-Displays muss zusätzlich zum Modul oled.py der Hardwaretreiber ssd1306.py in den Flash des Controllers hochgeladen werden. Ein LCD braucht die Treiber lcd.py und hd44780.py im Flash.

Die Angaben für Spalten- und Zeilenanzahl müssen angepasst werden.

```
if displayPresent and oledUsed :
    from oled import OLED # holt ssd1306.py automatisch
    d=OLED(i2c,heightw=32,widthw=128) # Pixelmaße
    d.clearAll()
elif displayPresent and lcdUsed:
    from lcd import LCD # holt hd44780u.py automatisch
    d=LCD(i2c,adr=0x27,cols=16,lines=2) # Zeichenmaße
    d.cursorBlink(0)
    d.cursor(0)
    d.backLight(1)
```

# Schnittstellen-Objekte und Funktionen

Die hier beschriebenen Objekte sind alle in der Datei UDP-Bausteine_poly.py zu finden.

## Hilfsmittel

Mit den nächsten Objekten und Funktionen kümmern wir uns um die Funkverbindung. Das Dictionary **connectStatus** übersetzt die Statusnummern, die uns die Methode **status**() des WLAN-Moduls liefert, in freundlichen Klartext.

```
connectStatus = {
    1000: "STAT_IDLE",  # ESP32
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",  # ESP8266
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
    }
```

Nicht weniger mystisch ist die MAC-Adresse, die von der Methode **config**("mac") als **bytes**-Objekt abgeliefert wird. Was meist als Kauderwelsch ankommt, wird durch die Funktion **hexMac**() in üblichen Hexadezimal-Code umgesetzt. Was uns **config**("mac") liefert, sieht etwa so aus.

```
>>> import network, socket
>>> nic = network.WLAN(network.STA_IF)
>>> nic.active(True)
True
>>> MAC = nic.config('mac')
>>> MAC
b'x!\x84\xe0\x19t'

>>> hexMac(MAC)
'78-21-84-e0-19-74'
```

```
def hexMac(byteMac):
  macString =""
  for i in range(0,len(byteMac)):      # Fuer alle Bytewerte
    macString += hex(byteMac[i])[2:]   # String ab 2 bis Ende
    if i <len(byteMac)-1 :             # Trennzeichen
      macString +="-"                  # bis auf letztes Byte
  return macString
```

Wir holen die Bytes einzeln heraus und lassen den Wert in einen Hex-String umformen.

**>>> hex(MAC[0])**
*'0x78'*

Nach den ersten fünf Strings wird jeweils ein "-" angehängt. Den fertigen Hex-String geben wir zurück.

Alternative Lösung zu hexMac():

```
def hexMac(byteMac):
  macString =""
  for i in range(0,len(byteMac)):      # Fuer alle Bytewerte
    macString += "{:02X}".format(byteMac[i])  # String ab 2 bis Ende
    if i <len(byteMac)-1 :             # Trennzeichen
      macString +="-"                  # bis auf letztes Byte
  return macString
```

## Die Credentials-Datei

Jedes Dictionary in der Liste **creds** beginnt mit der Feststellung des Interface-Typs: "STA" (=WLAN-Verbindung) oder "AP" (Accesspoint auf dem ESP32). Es folgen SSID, Passwort, gewünschte IP und gewünschte Portnummer, ferner die Netzwerkmaske, Gateway-Adresse und DNS-Adresse.

Für das Einrichten einer WLAN-Verbindung wird die Funktion **connect2router**() verwendet. Die Funktion **setAccesspoint**() baut einen Accesspoint auf dem ESP32/ESP8266 auf.

In **credentials_poly.py** sind eine Reihe von Accesspoint-Daten enthalten, die der Reihe nach abgearbeitet werden, bis eine Verbindung zustande kommt. Klappt das nicht, dann baut der ESP32 selber einen Accesspoint auf, mit dem sich das Handy direkt verbinden kann, sofern der letzte Eintrag in **creds** den Typ "**AP**" gesetzt hat. Die Daten jeder Station stehen in jeweils einem Dictionary. Die Liste **creds** fasst alle Dictionarys zusammen.

```
# credentials_poly.py
# Geben Sie hier Ihre eigenen Zugangsdaten an
# mehrere Verbindungen sind möglich
creds=[
    {"myIF" : "STA",
     "mySSID" : "BELATRIX",
     "myPass" : "mockingbird",
     "myIP" : "10.0.1.199",
     "myPort" : 9002,
     "myMask" :"255.255.255.0",
     "myGW" : "10.0.1.20",
     "myDNS" : "10.0.1.100",
```

```
    },
    {"myIF" : "STA",
     "mySSID" : "EMPIRE_OF_ANTS",
     "myPass" : "nightingale",
     "myIP" : "10.0.1.183",
     "myPort" : 9001,
     "myMask" :"255.255.255.0",
     "myGW" : "10.0.1.20",
     "myDNS" : "10.0.1.100",
    },
     {"myIF" : "AP",
      "mySSID" : "ANTARES99",
      "myPass" : "greenCacadu",
      "myIP" : "10.0.1.200",
      "myPort" : 9003,
      "myMask" :"255.255.255.0",
      "myGW" : "10.0.1.20",
      "myDNS" : "10.0.1.100",
     },
     ]
```

## Controller mit einem Router oder Accesspoint verbinden

Die Funktion **connect2router**() wird beim Aufruf mit der Nummer eines Eintrags in dem Dictionary **creds** gefüttert. Wir schalten schon mal das AP-Interface gezielt aus, um danach ein Station-Interface-Objekt zu instanziieren, **nic**. Nach dem Aktivieren fragen wir die MAC ab und lassen sie uns ausgeben. Die brauchen wir, um am WLAN-Router oder Accesspoint unseren ESP32 anzumelden. Tun wir das nicht, wird der Cerberus des Accesspoints den ESP32 gewaltig in den Allerwertesten beißen und keinen Zutritt gewähren. Den Accesspoint oder Router ohne MAC-Filtering zu betreiben, ist übrigens keine besonders gute Idee, weil dann Hinz und Kunz einen Zugang bekommen, was der Sicherheit sicher nicht besonders zuträglich sein dürfte. Dann ist erst einmal eine kurze Rast empfehlenswert, bis sich das Interface häuslich eingerichtet hat.

Um mit dem ESP32/ESP8266 eine WLAN Verbindung zu einem Accesspoint aufzubauen, rufen wir die Funktion **connect2router**() mit einem Index in die Liste **creds** aus dem Modul **credentials_poly.py** auf. Der indizierte Eintrag muss mit dem Item `"myIF" : "STA"` beginnen

```
def connect2router(entry): # n = Connection-Set-Nummer
    # ************** Zum Router verbinden ******************
    nic=network.WLAN(network.AP_IF)
    nic.active(False)
    nic = network.WLAN(network.STA_IF)  # erzeugt WiFi-Objekt
    nic.active(True)                    # nic einschalten
    MAC = nic.config('mac')# binaere MAC-Adresse abrufen und
    myMac=hexMac(MAC)      # in Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
    sleep(2)
```

Wir verwenden für die Einrichtung der Netzverbindung die Daten aus dem Dictionary in **creds**, das durch die übergebene Nummer angesprochen wird. Die Vorgehensweise mit einer eigenen Credits-Datei erlaubt uns, nacheinander mehrere Verbindungen zu testen, ohne die Verbindungsdaten im Programm selbst ablegen zu

müssen. So kann der WLAN-Verkehr sicher und ohne Änderung des Programms den örtlichen Bedingungen angepasst werden.

```
nic.ifconfig((creds[entry]["myIP"],
              creds[entry]["myMask"],
              creds[entry]["myGW"],
              creds[entry]["myDNS"]))
```

In der Regel wird zu diesem Zeitpunkt beim ESP32 noch keine Verbindung bestehen, also bauen wir eine auf. Bei einem ESP8266 verhält sich das ein wenig anders. Wir lesen also die SSID und das Passwort aus **creds** ein und reichen die Daten an **connect**() weiter. Wenn wir **displayPresent** anfangs auf **True** gesetzt haben, wird jetzt im Display die SSID des Accesspoints ausgegeben. Dabei ist es egal, ob ein LCD-Display oder ein OLED-Display verbunden ist. Beide APIs sind so verfasst, dass kompatible Ausgabeanweisungen gleichlauten und dieselbe Syntax aufweisen.

```
if not nic.isconnected():
  nic.connect(creds[entry]["mySSID"],
              creds[entry]["myPass"])
  print("Status: ", nic.isconnected())
  if displayPresent:
      d.clearAll()
      d.writeAt(creds[entry]["mySSID"],0,0)
```

Wir erzeugen einen String mit 10 Punkten und setzen den Durchlaufzähler auf 0. Solange der ESP32 noch keine IP-Adresse bekommen hat, wird im Sekundentakt die Schleife durchlaufen. Bei jedem Durchlauf wird der Zähler erhöht und ein längeres Stück des Punkte-Strings im Display ausgegeben. Hat **n** den Wert 10 erreicht, geben wir den Verbindungversuch auf und die Meldung "NOT CONNECTED" aus. In diesem Fall beenden wir die Funktion indem wir None zurückgeben.

```
  points="." * 10
  n=1
  while nic.status() != network.STAT_GOT_IP:
    print(".",end='')
    if displayPresent: d.writeAt(points[0:n],0,1)
    n+=1
    sleep(1)
    if n >= 10:
        if displayPresent:
            d.writeAt("  NOT CONNECTED  ",0,1)
        print("\nNot connected")
        nic.active(False)
        nic=None
        return None
```

Sonst wird der Verbindungsstatus in REPL ausgegeben. Ein vorhandenes Display wird geputzt, wir holen die aktuelle Konfiguration ab und geben sie in REPL und am Display aus.

```
print("\nStatus: ",connectStatus[nic.status()])
if displayPresent: d.clearAll()
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
      STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
print()
```

```
    if displayPresent:
        d.writeAt(STAconf[0]+":"+str(creds[entry]["myPort"]),0,0)
        d.writeAt(creds[entry]["mySSID"],0,1)
    return nic
```

Das Interface-Objekt geben wir zurück, um auch händisch außerhalb der Funktion darauf Zugriff zu haben.

Mit dem Interface-Objekt steht zwar die Verbindung aber wir haben noch keinen Datenkanal. Den setzen wir mit dem Modul **socket** auf.

Hier entscheidet sich, ob wir eine UDP- oder einen TCP-Kanal wünschen. Mit SOCK_DGRAM wählen wir einen UDP-Kanal, auf dem in lockerer Weise und ohne Sicherung Datenpakete ausgetauscht werden. Es wird also durch das Schichtenmodell nicht überprüft, ob die gesendeten Daten auch angekommen sind und ob eventuell Fehler aufgetreten sind. Einen TCP-Kanal bräuchten wir zum Beispiel, um eine Webseite auf dem ESP32 zu hosten.

Wir übergeben die Nummer des **creds**-Eintrags, weil wir auf die Portnummer zugreifen müssen und einen Timeoutwert in Sekunden. Ferner sagen wir dem System, dass es bei einem Neustart die alte Socket-Adresse wiederverwenden soll. Dann binden wir die Portnummer an die vorhandene IP-Adresse und setzen den Timeout. Das ist später wichtig, damit beim Empfang von Daten und Kommandos die Hauptschleife nicht blockiert wird.

```
def setSocket(n,delay):
    # ***************** Socket aufsetzen ******************
#     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    sock.bind(('', creds[n]["myPort"]))
    print("sending on port",creds[n]["myPort"])
    sock.settimeout(delay)
    return sock
```

## Accesspoint auf dem ESP32/ESP8266/Raspberry Pi Pico einrichten

Um mit dem ESP32/ESP8266 einen Accesspoint aufzubauen, rufen wir die Funktion **setAccessPoint**() mit einem Index in die Liste **creds** aus dem Modul **credentials_poly.py** auf. Der indizierte Eintrag muss mit dem Item `"myIF" : "AP"` beginnen

Wir verwenden für die Einrichtung der Netzverbindung die Daten aus dem Dictionary in **creds**, das durch die übergebene Nummer angesprochen wird. Die Vorgehensweise mit einer eigenen Credits-Datei erlaubt uns, nacheinander mehrere Verbindungen zu testen, ohne die Verbindungsdaten im Programm selbst ablegen zu müssen. So kann der WLAN-Verkehr sicher und ohne Änderung des Programms den örtlichen Bedingungen angepasst werden.

Als Erstes erzeugen wir ein Accesspoint-Objekt und aktivieren es.

```
def setAccessPoint(entry):
    # ************** Zum Router verbinden ******************
    nic=network.WLAN(network.AP_IF)
```

```
nic.active(True)
```

Um die MAC-Adresse des Accesspoint-Interfaces zu erfahren, können wir die Methode config des Accesspoint-Objekts mit dem Argument "mac" aufrufen. Das Ergebnis ist meist recht kryptisch. Die Funktion hexMac() macht aus dem gelieferten bytes-Objekt einen üblichen Hexadezimal-String. Die Funktion ist oben beschrieben.

```
MAC = nic.config('mac')# binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)      # in Hexziffernfolge umwandeln
print("STATION MAC: \t"+myMac+"\n") # ausgeben
```

nach der obligatorischen Pause von einer Sekunde können wir das Interface konfigurieren. Wir verwenden dazu die Daten aus dem Dictionary, das mit **entry** indiziert wird. ifconfig() setzt die obligatorischen Netzwerkdaten und config() legt die Logindaten fest, mit denen wir uns beim Accesspoint anmelden können.

```
sleep(1)
nic.ifconfig((creds[entry]["myIP"],
              creds[entry]["myMask"],
              creds[entry]["myGW"],
              creds[entry]["myDNS"]))
nic.config(essid=creds[entry]["mySSID"],
           password=creds[entry]["myPass"])
```

Sofern ein Display eingebunden wurde, geben wir eine Meldung aus. Wir richte einen String mit 10 Punkten ein und setzen den Rundenzähler auf 1. Die Schleife fragt den Zustand des Accesspoint-Interfaces im Sekundenrhythmus ab und setzt jeweils einen Punkt mehr in das Display, so es vorhanden ist. Nach 10 Durchläufen brechen wir das Erstellen mit einer Fehlermeldung ab und geben None an das aufrufende Programm zurück.

```
if displayPresent: d.writeAt("Setting up AP",0,0)
points="." * 10
n=1
while not nic.active():
    print(".",end='')
    if displayPresent: d.writeAt(points[0:n],0,1)
    n+=1
    sleep(1)
    if n >= 10:
        if displayPresent:
            d.clearAll()
            d.writeAt("NOT CONNECTED")
        return None
```

Wenn alles geklappt hat, holen wir die aktuellen Verbindungsparameter und geben sie in REPL und im Display aus. Das Interface-Objekt geben wir an das aufrufende Programm zurück.

```
print("NIC active:",nic.active())
if displayPresent: d.clearAll()
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
      STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
print()
if displayPresent:
    d.writeAt(STAconf[0],0,0)
```

```
        d.writeAt(STAconf[1],0,1)
        d.writeAt(STAconf[2],0,2)
    return nic
```

## Der Kommando-Parser

Es geht jetzt noch um den Empfang und die Decodierung von Kommandos. Das erledigt die Funktion **parse**(), der wir den empfangenen Kommando-String übergeben. Die Variable **automatik** wird global deklariert, damit ein veränderter Wert die Funktion verlassen kann, **val** setzen wir auf 0, damit diese lokale Variable in jedem Fall existiert. Wir vermeiden damit den Fehler, dass **var** irgendwo in **parse**() referenziert wird, bevor ihr ein Wert zugewiesen wurde. Damit keinesfalls ein Programmabbruch aufgrund eines Fehlers passieren kann, setzen wir den Kommando-Parser in ein try-except-Konstrukt.

```
def parse(rec):
    global automatik
    val=0
    try:
```

Ein Kommandostring hat folgende Form:

**Befehl:Wert\n** oder nur
**Befehl\n**

Falls der Kommando-String einen ":" enthält, teilen wir ihn in Befehl und Wert auf. Das sich ergebende Tupel entpacken wir in derselben Zeile in **rec** und **val**. Dann wandeln wir den Inhalt von **rec** in Großbuchstaben um.

```
        if rec.find(":") != -1:
            rec,val=rec.split(":")
        rec=rec.upper()
```

Wir schauen jetzt einfach nach, ob der Inhalt von rec gleich einem String in der Liste ist. Finden wir eine Übereinstimmung, dann führen wir die entsprechende Aktion aus und geben einen String für eine Antwort an den Sender des Befehls zurück. Hat der Inhalt von **rec** keine Entsprechung ergeben, wird **None** zurückgegeben.

```
        if rec in ["EIN","AUS","VENT","AUTO","STATE","QUIT"]:
            if rec=="EIN":
                peltier(1)
                return "cooling:1\n"
            elif rec=="AUS":
                peltier(0)
                return "cooling:0\n"
            elif rec=="VENT":
                val=int(val)
                vent(val)
                return "vent:{}\n".format(val)
            elif rec=="AUTO":
                val=int(val)
                automatik=val
                return "auto:{}\n".format(val)
            elif rec=="STATE":
                return "state:{};{};{};{};{:4.2f}\n"\
                        .format(r1(),r2(),vent(),automatik,mean)
```

```
            elif rec=="QUIT":
                cancel()
                return "stoped:1\n"
        return None
    except:
        pass
```

## Vorbereitungen im Hauptprogramm

Wir belegen entry schon mal mit None vor. Dann holen wir nacheinander die Dictionarys, stellen die Art der gewünschten Schnittstell fest und versuchen das Interface zu instanziieren. Gelingt das, dann brechen wir die for-Schleife mit break ab, nachdem wir den Schleifenindex an die Variable entry übergeben haben.

```
entry=None
for i in range(len(creds)):
    print("\n\n",creds[i]["mySSID"])
    if creds[i]["myIF"] == "STA":
        nic = connect2router(i)
        if nic is not None:
            sleep(2)
            entry=i
            break
    if creds[i]["myIF"] == "AP":
        nic = setAccessPoint(i)
        if nic is not None:
            sleep(2)
            entry=i
            break
```

Ist jetzt **entry** nicht vom None-Typ, setzen wir den Socket auf. Andernfalls erfolgt Fehlanzeige bevor wir das Programm beenden.

```
if entry is not None:
    s = setSocket(entry,0.1)
else:
    if displayPresent:
        d.clearAll()
        d.writeAt("NOT CONNECTED",0,0)
    print("Got no connection")
    exit()
```

## Die Hauptschleife

Ein Kernpunkt in der Hauptschleife ist das Abfragen der Empfangsschleife. Sind Zeichen im Empfangspuffer vorhanden, werden sie mit **recvfrom**() abgeholt. Das gelieferte Tupel entpacken wir in die Payload und die Socketdaten des Senders, IP-Adresse und Portnummer. Die Nutzlast ist ein Bytes-Objekt, das wir in einen String umcodieren und von angehängten CR- und LF-Codes befreien. Dann rufen wir den Kommando-Parser auf. Die zurückgegebene Antwort geben wir zur Kontrolle in REPL aus und senden sie an den Auftraggeber als Quittung zurück.

```
while 1:
    try:
        rec,adr=s.recvfrom(150)
        rec=rec.decode().strip("\r\n")
        reply=parse(rec)
        print(rec,adr)
        s.sendto(reply,adr)
```

```
except KeyboardInterrupt:
    print("Cancelled")
    exit()
except OSError:
    pass
```

In der Hauptschleife können wir auch auf Strg + C, einen KeyboardInterrupt abfragen. Tritt einer auf, dann verlassen wir das Programm mit **exit**().

Trat in der try-Abteilung beim in der Empfangsschleife ein Timeout auf, dann wurde dadurch eine Exception geworfen, die wir mit **except OSError** abfangen. Es liegt ja kein eigentlicher Fehler vor, es waren halt nur keine Zeichen im Empfangspuffer. Die Empfangsschleife bricht nach 100ms, die wir beim Initialisieren des Sockets vorgegeben haben, den Empfang mit der Exception ab. Dadurch wird periodisch nachgeschaut ob Zeichen angekommen sind und gleichzeitig sorgt der Timeout dafür, dass die Empfangsschleife nicht die Hauptschleife blockiert. Das würde nämlich passieren, wenn wir keinen Timeout vereinbaren würden. Die Empfangsschleife würde dann nämlich warten, bis irgendwann Zeichen eingetroffen sind. Weiter Aktionen in der Hauptschleife würden dann nicht ausgeführt.