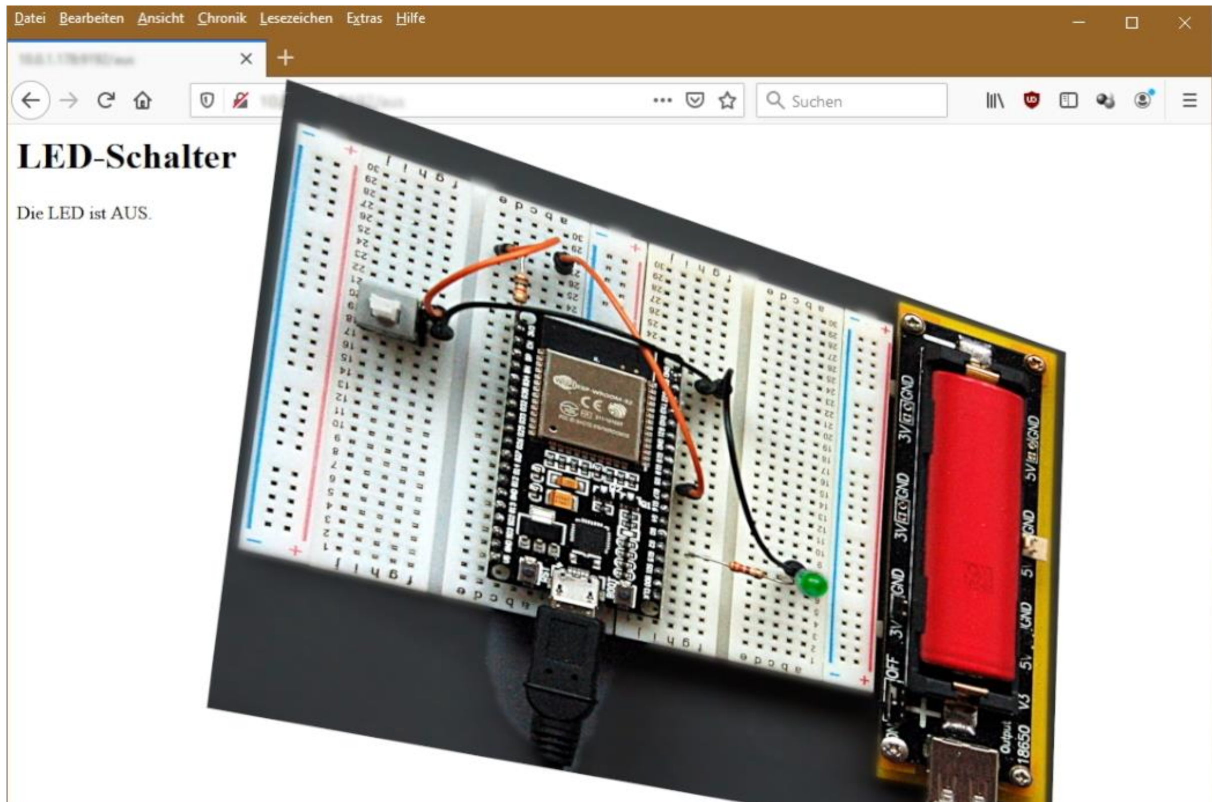


Projekte mit MicroPython und dem ESP8266/ESP32 (Teil 2)

Herzlich willkommen zum zweiten Teil unserer Reihe zu MicroPython.

Im **ersten Teil** haben wir die nötigen Vorbereitungen am PC sowie am Micro Controller von Espressif vorgenommen. Dabei liefen die Programme noch nicht autonom, sondern wurden vom PC aus via µPyCraft gestartet. So wollen wir es auch im 2. und 3. Teil handhaben, weil dieses Vorgehen Änderungen und Verbesserungen am Programmtext viel besser und effektiver zulässt. Für das fertige Produkt ändern wir das freilich, sodass der ESP32 auch ohne PC seine Aufgaben erfüllen kann. Die Steuerung des ESP32 wird dann, so viel sei schon verraten, über einen beliebigen Webbrowser vorgenommen.

Wer nach der Lektüre des ersten Teils zur Vertiefung die „Hausaufgaben“ gemacht hat, findet die Lösung am Ende dieses Teils. Die Dinge, die ich dort erklärt habe, werden hier im zweiten Teil des Blogs auch zum Einsatz kommen.

Für das weitere Vorgehen ist es hilfreich, zunächst einen genaueren Blick auf **O**bjekt-**O**rientierte-**P**rogrammierung (OOP) im Allgemeinen und speziell auf MicroPython zu werfen. Die Möglichkeit, den Kern der Sprache durch Module zu erweitern, werden wir immer wieder für unsere Lösungen nutzen. Dafür ist es nützlich, etwas über den Aufbau von Modulen zu wissen, Funktionen und Variablen unterscheiden und aufrufen zu können.

Ein Einblick in den Aufbau von MicroPython und OOP

MicroPython ist eine objektorientierte Sprache. Was das bedeutet, erläutere ich gleich auch noch anhand eines Beispiels. Eine Sprache ist objektorientiert, wenn in ihrer Definition "Baupläne" dafür vorhanden sind, wie später, während der Programmerstellung durch den User, seine Objekte aussehen können oder müssen. Objekte sind Datenstrukturen, mit deren Hilfe man Dinge aus dem Alltag in Computerprogrammen nachbilden (modellieren) kann. Sie haben Attribute oder Eigenschaften (aka Properties), welche die Erscheinungsform eines Objekts bestimmen. Der User kann mit Hilfe von Funktionen (aka Methoden) das Aussehen des Objekts verändern, indem er die Eigenschaftswerte verändert. Der direkte Zugriff auf die Attribute (Wertzuweisung, abfragen, verändern) ist in der OOP in der Regel nicht erwünscht. Stattdessen übergibt der User die Attributwerte als Parameter an die entsprechende Methode, welche die Änderungen am zugehörigen Objekt vornimmt. Damit kann z. B. durch Einsatz von Plausibilitätsabfragen verhindert werden, das ungültige Werte zur Verarbeitung kommen. Das klingt vielleicht kompliziert, ist es aber nicht wirklich.

Ein Beispiel aus einer alltäglichen Situation findet sich in der Baubranche. Der Architekt macht einen Plan für eine Reihenhaussiedlung, indem er das grobe Aussehen, innen wie außen, **eines** Hauses entwirft. Der **Plan** entspricht in der OOP einer **Klasse**. Jedes Haus hat Türen, Fenster, ein Dach, eine Raumaufteilung, Treppen usw. Für all diese Dinge gibt es wieder Pläne oder Klassen, die in die Klasse Haus integriert sind. Tür, Fenster, Treppe sind Unterklassen.

Wie die Handwerker ein Fenster herstellen sollen, unterliegt ganz grob einer **Vorschrift**, die in unserem Fall einer **Methode** entspricht. Wie der Fensterrahmen zu streichen ist, welche Art von Verglasung verwendet wird, das sind die **Eigenschaften** des Fensters, die aber nicht im Bauplan verankert sind, sondern die der Bauherr für sein **Objekt** eigens in Auftrag gibt. Gehen wir davon aus, dass der Bauherr nicht selbst Hand anlegt, sondern alles den Handwerkern machen lässt. Dann muss er die gewünschten Eigenschaften als **Parameter** an die Firma weiterleiten, damit diese den **Herstellungsprozess** wunschgemäß vollenden kann.

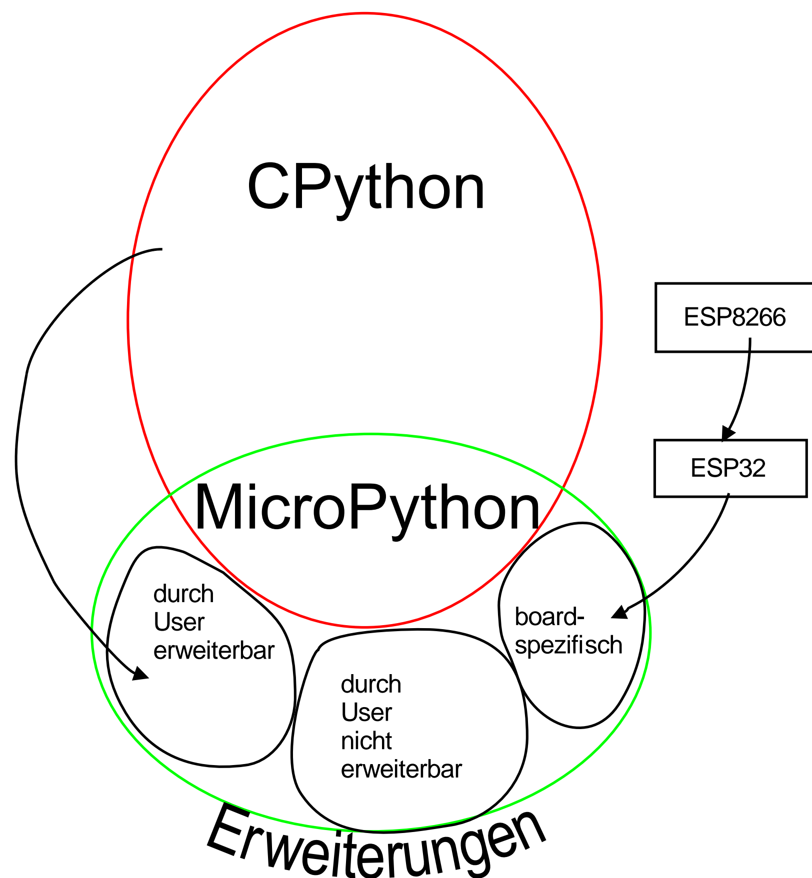
Wie der Bau eines Hauses sich aus mehreren Gewerken zusammensetzt, besteht auch der Kern von MicroPython aus einer Menge von eingebauten Modulen, die wiederum mindestens je eine Klasse enthalten. Eine Klasse als "Bauplan" für das daraus entstehende Programm-Objekt kann also als eine thematisch abgegrenzte Sammlung von Konstanten, Variablen und Funktionen verstanden werden. Die Funktionen werden in der Objekt-Orientierten-Programmierung als Methoden bezeichnet. Sie bestehen aus einer Zusammenstellung von einzelnen Anweisungen zu einem Block und sind stets an das Objekt (aka Instanz), das nach den Vorgaben dieser Klasse gebildet wird, gebunden (Instanzmethode). Methoden werden zur eindeutigen Unterscheidung zunächst nach dem Objekt benannt, auf das sie wirken sollen. Getrennt durch einen Punkt folgt dann der Name der Methode. Ich werde bei nächster Gelegenheit noch einmal näher darauf eingehen. Diese Punktnotation ist typisch für OOP. Was für Methoden gilt, gilt auch für Konstanten und Felder von Strukturen. Auch hier wird die Punktnotation verwendet. Thematisch zusammen

gehörige Attribute und Methoden können zu Libraries, sogenannten Modulen, zusammengefasst werden. Damit werden wir uns im dritten Teil beschäftigen.

Bei den Attributen unterscheiden wir zwischen Klassenattributen, die alle Objekte gleichermaßen betreffen und Instanzattributen, die auf einzelne Objekte in unterschiedlicher Weise wirken. Zur Verdeutlichung, die Anzahl an Stockwerken ist ein Klassenattribut, weil es für alle Häuser einer Klasse den gleichen Wert hat, der nur einmal festgelegt wird. Die Farbe der Fassade ist ein Instanzattribut, weil jedes Haus andersfarbig verputzt sein kann.

Damit beenden wir unseren Ausflug in die Baubranche. Ich denke, Sie haben die Arbeitsweise von OOP soweit verinnerlicht. Dann können wir mit MicroPython loslegen.

Was MicroPython von CPython unterscheidet, habe ich im ersten Teil schon kurz angerissen. Die folgende Grafik verdeutlicht das noch einmal. MicroPython ist ein kleiner Teil des Sprachumfangs von CPython. Das ist aber noch nicht die ganze Wahrheit. MicroPython wurde über den von CPython geerbten Sprachumfang hinaus um Module erweitert, die speziell auf das Zusammenwirken mit Microcontrollern abgestimmt sind. Ein Teil hiervon ist portspezifisch. Als Port (aka Plattform) wird das Board oder die Programmierumgebung (z. B. Linux) bezeichnet, auf dem oder in der MicroPython ausgeführt werden soll. Weil ESP8266/ESP32-Boards herstellerbedingt eine unterschiedliche Architektur haben können, sind portspezifische Erweiterungen des MicroPython-Kerns nicht portierbar, man kann sie also **nicht** ohne weiteres von einem Board auf ein anderes übertragen. Wenn man eigene Module aus CPython bildet, können diese natürlich ebenfalls den Sprachumfang von MicroPython erweitern.



Der folgende Befehl kann über das Terminalfenster eingegeben werden und liefert alle in MicroPython eingebauten Module. Damit ein Programm sie nutzen kann, muss man die gewünschten Module zu Beginn des Programms importieren, das macht sie dem Programm bekannt.

```
>>> help ('modules')
```

```
__main__          gc                        uasyncio/stream    upip_utarfile
_boot             inisetup                  ubinascii          upysh
_onewire          machine                   ubluetooth         urandom
_thread           math                      ucollections       ure
_uasyncio         micropython               ucryptolib         urequests
_webrepl          neopixel                  uctypes            uselect
apa106            network                   uerrno             usocket
btree             ntptime                    uhashlib          ussl
builtins          onewire                   uhashlib           ustruct
cmath             sys                       uheapq             utime
dht               uarray                    uio                utimeq
ds18x20           uasyncio/__init__ujson    uwebsocket
esp               uasyncio/core             umqtt/robust       uzlib
esp32             uasyncio/event            umqtt/simple       webrepl
flashbdev         uasyncio/funcs            uos                webrepl_setup
framebuf          uasyncio/lock             upip               websocket_helper
Plus any modules on the filesystem
>>>
```

Weitere Module finden Sie unter https://github.com/micropython/micropython-lib

Das zweite Projekt

Wir wollen Messdaten erfassen und über WLAN übertragen. Nun dafür brauchen wir erst einmal eine WLAN-Verbindung zum lokalen Accesspoint. Um das zu bewerkstelligen, importieren wir das Netzwerkmodul **network**.

Damit wir die MAC-Adresse des ESP32 erfahren können, muss die binär abgerufene Information (Bytecode) aus dem ESP32 in eine lesbare hexadezimale Schreibweise umgeformt werden. Das macht eine Methode aus dem Modul **ubinascii**. Das führende "u" zeigt an, dass es sich bei dem Modul um ein CPythonmodul handelt, das von CPython auf MicroPython adaptiert wurde.

```
import network
import ubinascii
```

Wir erzeugen nun als Erstes eine Instanz, ein Objekt der Netzwerkklasse. Ich habe es "nic" (von **N**etwork **I**nterface **C**onnector) genannt. Sie können dafür einen beliebigen anderen Namen wählen, welcher der Namenskonvention von Python entspricht. Als Schnittstelle soll das Station-Interface des ESP32 dienen. Alternativ könnten wir das Accesspoint-Interface wählen, mit etwas anderer Funktionalität.

Anders als in der Arduino-IDE oder in NodeMCU-LUA gibt es in MicroPython keinen Parallelbetrieb der beiden Interfaces.

Nach dem Kaltstart des ESP-Moduls bekommen wir nach Eingabe der folgenden Anweisung die darauffolgende Ausgabe im Terminalfenster. Mit der Eingabe weisen wir die Constructor-Methode der WLAN-Klasse, welche ein Teil des Moduls **network** ist, an, das Objekt **nic** zu erzeugen. Die Zugehörigkeit der WLAN-Klasse zum network-Modul deuten wir durch den Punkt zwischen den Bezeichnern an. Der Methode übergeben wir als Parameter die Konstante network.STA_IF und geben damit dem Wunsch Ausdruck, dass wir den ESP32 als STATION einrichten wollen. Die Schreibweise **network.STA_IF** deutet an, dass die Konstante STA_IF in der Klasse network deklariert ist.

```
>>> nic = network.WLAN(network.STA_IF)
```

Ausgabe:

I (8156610) wifi:wifi driver task: 3ffd1178, prio:23, stack:3584, core=0
[0;32mI (8179931) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE[0m
[0;32mI (8179931) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE[0m
I (8179971) wifi:wifi firmware version: 44aa95c
I (8179971) wifi:config NVS flash: enabled
I (8179971) wifi:config nano formating: disabled
I (8179971) wifi:Init dynamic tx buffer num: 32
I (8179971) wifi:Init data frame dynamic rx buffer num: 32
I (8179981) wifi:Init management frame dynamic rx buffer num: 32
I (8179981) wifi:Init management short buffer num: 32
I (8179991) wifi:Init static rx buffer size: 1600
I (8179991) wifi:Init static rx buffer num: 10
I (8180001) wifi:Init dynamic rx buffer num: 32

Nachdem der Constructor der Netzwerkklasse WIFI das Interface **nic** eingerichtet hat, müssen wir es durch folgende Eingabe im Terminalfenster aktivieren. Die Methode active() gehört zu einem Objekt aus der Klasse WLAN, von der unser Objekt **nic** abgeleitet ist. Also sprechen wir das Objekt nic an und übergeben dessen Methode active() den Parameter True. Die Methode schaltet darauf hin für uns das Interface ein. Ein "False" würde es abschalten.

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

Ausgabe:

[0;32mI (8189401) phy: phy_version: 4180, cb3948e, Sep 12 2019, 16:39:13, 0, 0[0m
I (8189401) **wifi:mode : sta (**11-11-11-11-11-11**)**
True
[0;32mI (8189401) wifi: STA_START>>> [0m

Die merkwürdigen Symbole, wie [0;32ml und [0m, sind Steuercodes für eine farbige Darstellung der Bildschirmausgaben. Allerdings kann µPyCraft damit nichts anfangen und deshalb verschandeln die Codes das Bild eher als es aufzuhübschen.

Die im Text verwendete MAC-Adresse 11-11-11-11-11-11 und der Pool von IP-Adressen der Form 8.8.8.x haben in diesem Text nur den Charakter von Platzhaltern. Bei Ihren Experimenten Ist die MAC-Adresse durch Ihr ESP32-Modul vorgegeben. Welche Werte für die IP-Adresse in Frage kommen, wird durch ihr lokales Netzwerk vorgegeben. Je nach Konfiguration desselben können Sie über den DHCP-Server Ihres Accesspoints/ Routers bestimmte Adressen gezielt angeben oder Sie bekommen eine aus dem dynamisch verwalteten Bereich des DHCP zugewiesen. Im letzteren Fall kann sich die IP-Adresse von Sitzung zu Sitzung ändern.

Die wichtigste Information in der letzten Bildschirmausgabe habe ich fett formatiert. Sie enthält die MAC-Adresse des ESP-Moduls.

wifi:mode : sta (11-11-11-11-11-11**)**

MAC-Adressen werden bei der Herstellung eines netzwerkfähigen Bauteils vergeben und sollten weltweit einmalig sein. Daher wird die MAC bei jedem ESP-Modul anders lauten. Sie besteht aber in jedem Fall aus sechs Hexadezimalwerten, die durch ein Trennzeichen, hier ist es ein Bindestrich, voneinander getrennt sind. Die MAC-Adresse benötigen wir für den Eintrag im lokalen Accesspoint/Router und evtl. im lokalen DHCP-Server, damit der ESP32 von anderen Geräten gefunden und angesprochen werden kann. Wie der Eintrag im Einzelnen durchzuführen ist, kann ich hier nicht umfassend darstellen, weil es dafür einfach zu viele Geräte und Plattformen gibt. Ich verweise Sie daher an dieser Stelle auf das Handbuch zu Ihrem Accesspoint.

Über die **config**()-Methode des network-Objekts **nic** kann man ebenfalls die MAC-Adresse abrufen, indem man den Parameter **mac** in Anführungszeichen übergibt. Diese Methode liefert einen teilweise unverständlichen Bytecode zurück, den wir uns durch Anwendung der Methode **hexlify**() aus der Klasse **ubinascii** übersetzen lassen. Das führende "u" des Klassen- oder Modulnamens deutet, wie schon erwähnt, darauf hin, dass dieses Modul ein für MicroPython zurechtgestutztes Modul aus CPython ist. Das u steht für das Sonderzeichen µ = micro. An hexlify() wird neben dem Bytecode der MAC-Adresse das Zeichen "-" übergeben, das als Trennzeichen zwischen den Hexcodes ausgegeben werden soll. Sie können dafür jedes beliebige andere ASCII-Zeichen einsetzen.

nic.config('mac') liefert die MAC.
ubinascii.hexlify(nic.config('mac'),"-") macht das Ganze lesbar.
print(ubinascii.hexlify(nic.config('mac'),"-")) gibt den lesbaren Text im Terminalfenster aus. Hier kommt die Eingabe, alles zusammen in einer Zeile.

```
>>> print(ubinascii.hexlify(nic.config('mac'),"-"))
```

b'11-11-11-11-11-11'

Diese Darstellung ist nicht gerade schön, genügt aber, um die MAC-Adresse (aka physikalische Adresse) des ESP32 zu erhalten. Ich zeige später noch, wie man die Darstellung aufhübschen kann.

Netzwerk einrichten und Verbindung zum lokalen Accesspoint aufnehmen.

Hinweise für das weitere Vorgehen:

1. Für die Definition konstanter Zeichenketten (Textkonstanten) können paarweise der Apostroph oder das gerade Anführungszeichen verwendet werden.
2. In MicroPython kann eine Befehlszeile mit einem ";" abgeschlossen werden, was aber nicht zwingend nötig ist.
3. Wenn in einer Zeile mehrere Anweisungen stehen sollen, **müssen** sie durch ein Semikolon ";" getrennt werden.

Füttern Sie nun zuerst das ESP-Modul mit Ihren persönlichen Credentials für den Netzzugang. <Ihre SSID> und <Ihr Passwort> ersetzen Sie bitte durch den Namen Ihres Accesspoints und Ihr dortiges Zugangspasswort.

```
>>> mySid = '<Ihre SSID>';  myPass = "<Ihr Passwort>"
>>> nic.connect(mySid, myPass)
```

Ausgabe im Terminalfenster:

```
I (8753761) wifi:state: init -> auth (b0)
I (8753761) wifi:state: auth -> assoc (0)
I (8753781) wifi:state: assoc -> run (10)
I (8753811) wifi:connected with <Ihr AP>, aid = 1, channel 6, BW20, bssid = <Ihre MAC>
I (8753811) wifi:security type: 3, phy: bgn, rssi: -63
I (8753811) wifi:pm start, type: 1

 [0;32mI (8753821) network: CONNECTED[0m
I (8753881) wifi:AP's beacon interval = 102400 us, DTIM period = 1
 [0;32mI (8754761) event: sta ip: 8.8.8.50, mask: 255.255.255.0, gw: 8.8.8.30[0m
 [0;32mI (8754761) network: GOT_IP[0m
```

Die fett gedruckte Zeile ist das Ergebnis, wenn

1. die Verbindung zum Accesspoint geklappt hat
2. von einem DHCP-Server im lokalen Netz eine IP-Adresse bezogen werden konnte.

Die Angaben zu IP, Netzmaske und Gateway haben bei Ihnen sicher andere Werte, bestehen aber auch aus Vierergruppen von Zahlen, die kleiner als 256 sind und durch Punkte gegliedert sind.

Für die Verbindung ist es erforderlich, dass die MAC-Adresse (im Weiteren einfach MAC) im Accesspoint in der Liste der zugelassenen Geräte aufgenommen wurde. Viel zu unsicher ist es, die Aufnahme neuer Geräte grundsätzlich frei zu schalten. Für die Vergabe einer IP-Adresse (IP) durch den Accesspoint muss auf diesem das Feature DHCP aktiviert sein, ferner muss dem

DHCP-Dienst ein Pool an Adressen für diesen Zweck zur Verfügung gestellt werden. Alternativ kann man einer MAC auf diesem Weg auch eine bestimmte feste IP zuweisen. Hinweise dazu entnehmen Sie bitte dem Handbuch zu Ihrem Accesspoint/Router.

Wir haben jetzt eine Verbindung zum Accesspoint im lokalen Netzwerk. Vom PC aus kann man nun die Netzwerkverbindung testen, indem man in einem DOS-Fenster (aka Eingabeaufforderung) folgenden Befehl absetzt.

Der ping-Befehl sendet Datenpakete an die dahinter aufgeführte IP und misst die Zeit, bis eine Antwort vom entfernten Netzwerkgerät zurückkommt.

```
C:\Users\roby>ping 8.8.8.50
```

Bildschirmausgabe:

Ping wird ausgeführt für 8.8.8.50 mit 32 Bytes Daten:
Antwort von 8.8.8.50: Bytes=32 Zeit=115ms TTL=255
Antwort von 8.8.8.50: Bytes=32 Zeit=16ms TTL=255
Antwort von 8.8.8.50: Bytes=32 Zeit=35ms TTL=255
Antwort von 8.8.8.50: Bytes=32 Zeit=30ms TTL=255

Ping-Statistik für 8.8.8.50:
    Pakete: Gesendet = 4, Empfangen = 4, Verloren = 0
    (0% Verlust),
Ca. Zeitangaben in Millisek.:
    Minimum = 16ms, Maximum = 115ms, Mittelwert = 49ms

Wenn die Rückmeldung bei Ihnen ähnlich aussieht, haben Sie die erste Hürde geschafft, der erste Kontakt mit dem ESP32 hat stattgefunden. Nun gehen wir zum nächsten Schritt über und erzeugen auf dem Board einen ersten, einfachen Webserver, durch den sich das Prinzip leicht verstehen lässt. Wenn alle 4 Pakete als 'verloren' angezeigt werden, überprüfen Sie bitte, ob Sie Passwort und SSID richtig angegeben und die IP korrekt eingetippt haben.

Fassen wir die bisherigen Maßnahmen noch einmal zusammen, am besten in einem Python-Script, das ich unter dem Namen **wifi_connect.py** abgespeichert und zum ESP32 hochgeladen habe. Sie können den folgenden Text kopieren oder herunterladen und auf Ihren ESP32 transferieren. Kopieren Sie dazu den Text des Scripts in die Zwischenablage und fügen Sie ihn in ein neues, leeres Dokument ein, das Sie mit File – New in µPyCraft erzeugt haben. Speichern Sie die Datei im Ordner Workspace von µPyCraft mit dem Namen **wifi_connect.py** ab. Damit sie im Workspace sichtbar wird, vergessen Sie bitte nicht, aus dem File-Menü **Reflush Directory** aufzurufen. Öffnen Sie nun die Datei mit Doppelklick und tragen Sie ihre Zugangsdaten für den Accesspoint ein. Nach dem erneuten Abspeichern ziehen Sie das Script ins device-Directory oder nutzen Sie zum Kopieren den **DownloadAndRun**-Button.

```
import os
from time import time,sleep
import network
import ubinascii
from machine import Pin

# Die Dictionarystruktur (dict) erlaubt später die Klartextausgabe
# des Verbindungsstatus anstelle der Zahlencodes
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
```

```
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND"
    }
#
# ---------------------------------------------------------------------
def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode entgegen und
  bildet daraus einen String für die Rückgabe
  """
  macString =""
  for i in range(0,len(byteMac)):     # Für alle Bytewerte
    macString += hex(byteMac[i])[2:]  # vom String ab Position 2 bis Ende
    if i <len(byteMac)-1 :            # Trennzeichen bis auf das letzte Byte
      macString +="-"
  return macString
#
# ---------------------------------------------------------------------
def zeige_ap_liste():
  """
  Scannt die Funkumgebung nach vorhandenen Accesspoints und liefert
  deren Kennung (SSID) sowie die Betriebsdaten zurück. Nach entsprechender
  Aufbereitung werden die Daten im Terminalfenster ausgegeben.
  """
  # Gib eine Liste der umgebenden APs aus
  liste = nic.scan()
  autModus=["open", "WEP", "WPA-PSK", "WPA2-PSK", "WPA/WPA2-PSK"]
  for AP in liste:
    print("SSID: \t\t",(AP[0]).decode("utf-8"))
    print("MAC: \t\t",ubinascii.hexlify(AP[1],"-").decode("utf-8"))
    print("Kanal: \t\t",AP[2])
    print("Feldstaerke: \t\t",AP[3])
    print("Autentifizierung: \t",autModus[AP[4]])
    print("SSID ist \t\t",end='')
    if AP[5]:
      print("verborgen")
    else:
      print("sichtbar")
    print("")
#
# ---------------------------------------------------------------------
# Netzwerk-Interface-Instanz erzeugen und ESP32-Stationmodus aktivieren;
# möglich sind network.STA_IF und network.AP_IF. Beide wie in LUA oder
# AT-based oder Adruino-IDE ist in MicroPython nicht möglich
nic = network.WLAN(network.STA_IF)  # Constructor
nic.active(True)
#
# binäre MAC-Adresse abrufen und in ein Hex-Tupel umgewandelt ausgeben
MAC = nic.config('mac')
myMac=hexMac(MAC)
```

```
print("meine MAC: \t"+myMac+"\n")
#
# Folgender Befehl geht auch, liefert aber kein schönes Ergebnis
# print(ubinascii.hexlify(MAC,"-")
#
# Zeige mir verfügbare APs
zeige_ap_liste()

# Verbindung mit AP im lokalen Netzwerk aufnehmen, falls noch nicht verbunden
if not nic.isconnected():
  # Geben Sie hier Ihre eigenen Zugangsdaten an
  mySid = '<Ihre SSID>'; myPass = "<Ihr Passwort>"
  # Zum AP im lokalen Netz verbinden und Status anzeigen
  nic.connect(mySid, myPass)
else:  # sonst,
  # Wenn bereits verbunden, zeige Verbindungsstatus und Config-Daten
  print("Verbindungsstatus: ",connectStatus[nic.status()])
  STAconf = nic.ifconfig()
  print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",STAconf[1],"\nSTA-
GATEWAY:\t",STAconf[2] ,sep='')
#
```

Datei: wifi_connect1.py
Nachdem die meisten Anweisungen bereits besprochen sind, werde ich nur noch auf die neu in das Script aufgenommenen Teile eingehen. Weitere Hinweise finden Sie im Script auch als Kommentare.

```
import os           # brauchen wir fuer Dateioperationen
from time import time, sleep
import network
import ubinascii
from machine import Pin
```

Das Modul **os** enthält Dateioperationen, die man nutzen kann, wenn das System über µPyCraft nicht mehr betretbar ist. Deshalb ist es wichtig, das Modul noch zur "Lebenszeit" im System zu integrieren. Aus dem Modul **time** importieren wir die Klassen **time** und **sleep**. In **network** ist alles enthalten, was für WiFi benötigt wird. **ubinascii** haben wir zum Decodieren schon benutzt und **Pin** aus dem Modul **machine** brauchen wir zur Steuerung von GPIO-Pins.

Das Script enthält zwei selbst definierte Funktionen. Funktionen haben in MicroPython keinen Typ wie in C. Nach dem reservierten Wort **def** folgt der Funktionsname nach der Python-Namenskonvention, danach die Parameterliste in runden Klammern. Die Parameter haben ebenfalls keine vorangestellte Typdeklaration wie in C. Der Funktionskopf wird mittels Doppelpunkt ":" abgeschlossen wie jeder Programmstrukturkopf. Der Funktionskörper ist wie bei allen anderen MicroPython-Strukturen eingerückt. Als Übergabeparameter byteMac wird hier eine Variable vom Typ bytes erwartet.

```
def hexMac(byteMac):
….
  for i in range(0,len(byteMac)):     # Für alle Bytewerte in der Struktur
```

Die for-Schleife besitzt den Laufindex i, der Werte zwischen 0 (inklusive) und der **Anzahl** an Bytes (**exklusive**) annimmt. Die Zählung von Listeninhalten wie in **byteMac** beginnt bei 0. Die Liste enthält 6 Bytes. Der Index läuft also von 0 bis 5, 6 exklusive.

```
  macString += hex(byteMac[i])[2:]  # vom String im Feld i ab Position 2 bis Ende
```

Jedes Byte wird in einen String umgewandelt, der den Hexcode des Bytes inklusive des Vorsatzes 0x enthält, z. B. 0xB7. Um nur die Hexziffern zu erhalten, verwenden wir von dem String nur die Zeichen ab der Position 2 bis Ende. Weil auch die Zeichen in Strings ab der Position 0 indiziert sind, erhalten wir b7 oder B7. Diese Zeichen verketten wir mit dem bisherigen String **macString**. Die Schreibweise += ist die Abkürzung für

```
  macString = macString + hex(byteMac[i])[2:]
```

```
  if i <len(byteMac)-1 :          # Trennzeichen bis auf das letzte Byte
    macString +="-"
```

Nach den ersten fünf Hexcodes soll jeweils ein "-" folgen, nach dem letzten nicht mehr.

```
return macString
```

Der Rückgabewert ist der komplett zusammengesetzte String **macString**. Das ist jetzt eine Zeichenkette, ein String, auf den auch alle Stringmethoden angewandt werden können.

```
def zeige_ap_liste():
```

Die Funktion **zeige_ap_liste**() benötigt keine Parameter, weil die Anweisung **nic.scan**() selbst für die benötigten Eingabedaten sorgt und die lokale Variable **liste** damit füllt, falls Accesspoints gefunden werden. Die Methode **scan**() des nic-Objekts liefert pro gefundenem Accesspoint ein sogenanntes Tupel aus 6 Feldern zurück: (SSID, MAC, Funkkanal, Feldstärke, Authentifizierungsmethode, hidden).

```
 liste = nic.scan()
```

Leere Liste erzeugen.

```
 autModus=["open", "WEP", "WPA-PSK", "WPA2-PSK", "WPA/WPA2-PSK"]
```

Die hiermit definierte Liste **autModus** erlaubt über die Indizierung ihrer Elemente die Klartextausgabe der Authentifizierungsmethode. Die folgende for-Schleife zerpflückt jedes Tupel eines Accesspoints in die einzelnen Felder und bereitet die enthaltenen Daten für die Ausgabe vor.

```
 for AP in liste:
  print("SSID: \t\t",(AP[0]).decode("utf-8"))
  print("MAC: \t\t",ubinascii.hexlify(AP[1],"-").decode("utf-8"))
  print("Kanal: \t\t",AP[2])
  print("Feldstaerke: \t\t",AP[3])
  print("Authentifizierung: \t",autModus[AP[4]])
  print("SSID ist \t\t",end='')
  if AP[5]:
    print("verborgen")
  else:
```

```
    print("sichtbar")
  print("")
```

Erwähnenswert sind folgende Zeilen.

```
  print("SSID: \t\t",(AP[0]).decode("utf-8"))
```

Die Methode **decode**() stellt aus der Bytefolge der Stationskennung einen normalen utf-8-String her. Das Gleiche geschieht mit der "hexlifizierten" Bytefolge der MAC. Kanal und Feldstärke sind Integerwerte und werden normal als Zahl gewertet. Der Wert für die Authentifizierung ist ebenfalls ein Integertyp und dient als Index in die Liste **autModus.** Der Aufruf des Listenwerts wandelt somit den Index in Klartext um. Das Feld **hidden** ist vom Typ **bool** und wird für die Ausgabe von "verborgen" oder "sichtbar" abgefragt.

```
  print("SSID ist \t\t",end='')
```

Die Ergänzung **end=''** sorgt dafür, dass der print-Befehl nicht mit einem Zeilenvorschub abgeschlossen wird. '' besteht hier aus **zwei einfachen Hochkommata**, übergibt also kein Zeichen und überschreibt somit den Defaultwert "\n" = Zeilenvorschub.

Die Bildschirmausgabe für einen gefundenen Accesspoint sieht dann wie folgt aus.

```
SSID:               <Ihre SSID>
MAC:                11-11-11-11-11-11
Kanal:              6
Feldstaerke:        -62
Autentifizierung:   WPA2-PSK
SSID ist            sichtbar
```

Der Auftrag zur Stationssuche wird über den Aufruf der Funktion erteilt.

```
zeige_ap_liste()
```

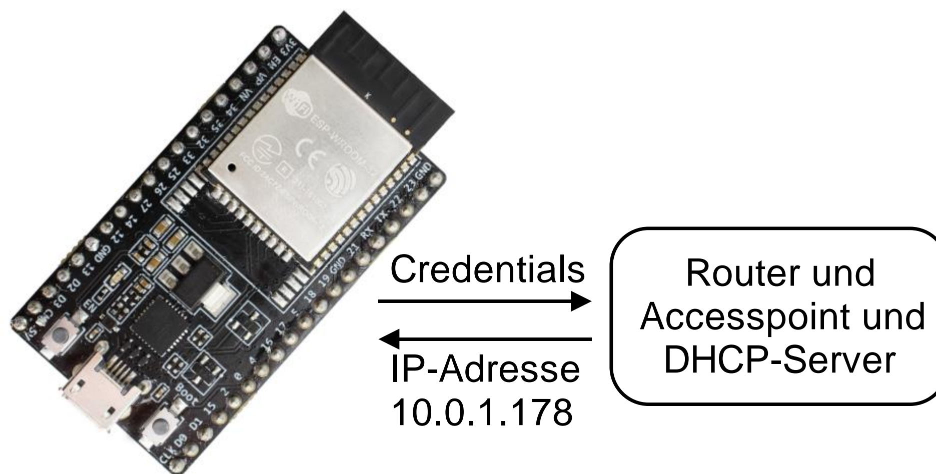
Damit beim Experimentieren mit dem Script nicht bei jedem Start wieder die Verbindung mit dem Accesspoint hergestellt werden muss, fragen wir über die Methode isconnected() des nic-Objekts den Verbindungszustand ab. Besteht noch keine Verbindung, dann soll die connect()-Methode eine solche mit dem Accesspoint herstellen. Das dauert ein paar Sekunden. Die while-Schleife wartet, bis die Verbindung steht und gibt im Sekundenabstand Punkte aus. Besteht die Verbindung bereits, dann werden sofort Status und Verbindungsdaten ausgegeben.

```
if not nic.isconnected():
  # Geben Sie hier Ihre eigenen Zugangsdaten an
  mySid = '…your SSID…'; myPass = "…your password…"
  # Zum AP im lokalen Netz verbinden und Status anzeigen
  nic.connect(mySid, myPass)
  # warten bis die Verbindung zum Accesspoint steht
  print("connection status: ", nic.isconnected())
  while nic.status() != network.STAT_GOT_IP:
    #pass
    print(".",end='')
    sleep(1)
# Wenn bereits verbunden, zeige Verbindungsstatus und Config-Daten
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
STAconf = nic.ifconfig()
```

```
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",STAconf[1],"\nSTA-
GATEWAY:\t",STAconf[2] ,sep='')
```

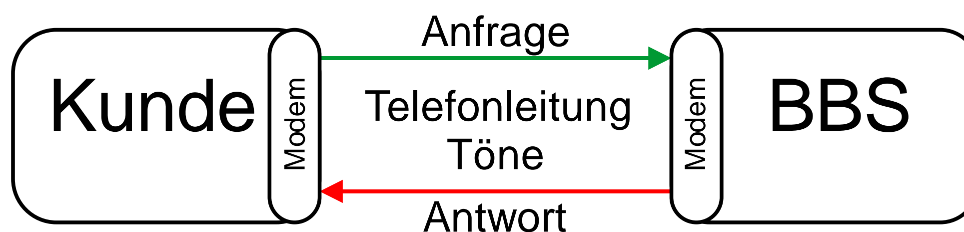
Der letzte Aufruf der print-Funktion gibt die Verbindungsdaten in drei Zeilen kommentiert aus. Die Erweiterung sep='' sorgt dafür, dass der Ausgabeteil nach einem Komma nicht durch ein Leerzeichen vom vorangehenden Teil abgesetzt wird. Das kann auch für andere Anwendungsfälle interessant sein, weil damit auch beliebige Zeichen als Trennzeichen festgelegt werden können.

Die Grafik zeigt den momentanen Verbindungszustand. ESP32 und Accesspoint sind verbunden.



Der ESP32 meldet sich mit "Hallo Welt"

Die Kommunikation zwischen dem PC und dem ESP32 fand bisher über das USB-Kabel statt. Maßgeblich für das Funktionieren waren und sind die seriellen Hardwareschnittstellen im PC, und auf dem ESP32, ferner in diesem Zusammenhang auch der USB-RS232-Adapter auf dem ESP32, der die Wandlung der übertragenen Signale von RS232 zu USB und umgekehrt übernimmt. Hinter den Hardwareschnittstellen ist auf beiden Geräten auch noch eine Softwareabteilung mit der Verarbeitung der transportierten Daten beschäftigt, die Treiber. Der Übertragungsweg ist das USB-Kabel. Das Client-Server-Prinzip auf RS232-Ebene aus der guten alten Zeit stellten damals Bulletinboardsysteme als Server dar, mit denen man sich mittels Modem und PC als Client über eine Telefonleitung verbunden hat.

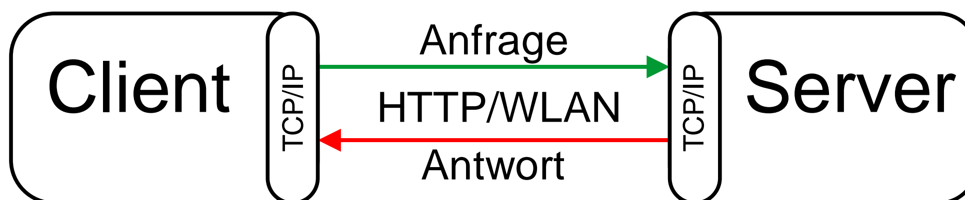


Im Prinzip geschieht die Konversation zwischen PC und dem ESP32 auch auf diese Art, nur wird keine Telefonleitung dazu benutzt, sondern das USB-Kabel zusammen mit dem CP2102.

Was hat das mit einem Server zu tun, fragen Sie sich? Ganz einfach, die Datenübertragung zwischen Server und Client funktioniert auf ähnliche Art und Weise, nur heißen die daran

beteiligte Hard- und Software anders. Der Übertragungsweg wird durch die elektromagnetischen Wellen gebildet, die zwischen Accesspoint und ESP32 gesendet und empfangen werden. Vom PC zum Accesspoint und umgekehrt wandert die Information entweder zusätzlich über ein Netzwerkkabel oder auch wieder über eine Funkverbindung. Die Treibersoftware bei der RS232-Übertragung wird im Netzwerkbetrieb durch sogenannte Sockets ersetzt. Zu so einem Socket gehören eine IP-Adresse und eine Portnummer. Die Portnummer des Servers wird durch unser Programm stets fest vergeben. Auf dem Client wird sie zufällig vom System ausgewählt.

Die Hardwareschnittstellen sind die Sende- und Empfangsteile im Accesspoint und im ESP32. Der Datenverkehr wird bei einer seriellen Verbindung durch die RS232 und USB-Protokolle geregelt, beim Netzverkehr, egal ob WLAN oder kabelgebunden, sind es die TCP/IP-Protokollstacks mit entsprechenden genormten Vorgaben, die regeln, wie der Transfer abzuwickeln ist.

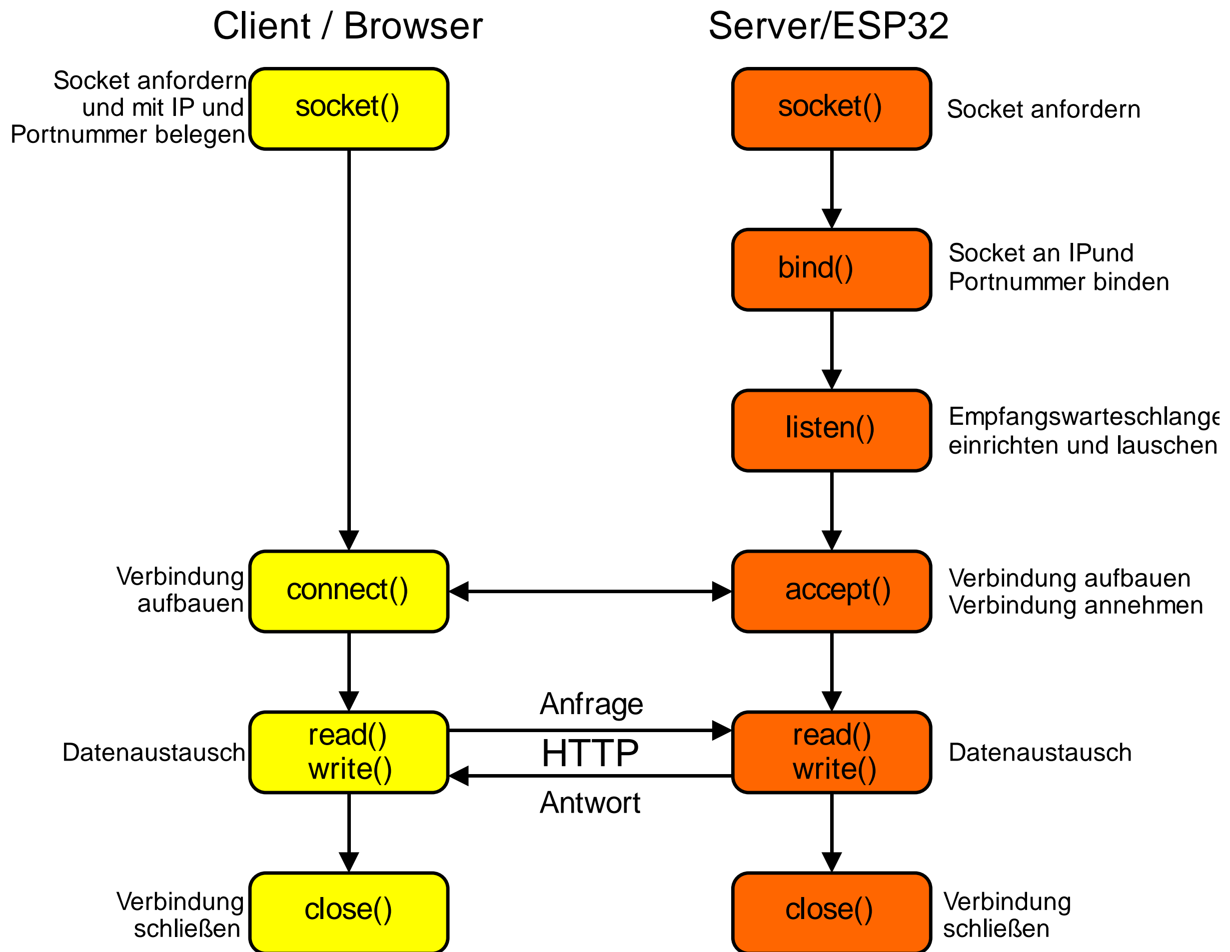


Wenn wir also per WiFi Kontakt mit dem ESP32 als Server aufnehmen wollen, müssen wir dort eine solche Schnittstelle, einen Socket einrichten, der eingehende Anfragen entgegennimmt und darauf antwortet. Auf dem PC übernimmt das Herstellen eines Sockets in transparenter Weise die Browsersoftware. Transparent bedeutet, dass der Benutzer davon nichts mitbekommt, muss er ja auch nicht.

Der Socket auf dem ESP32 muss von uns eingerichtet werden und wartet dann auf Anfragen, die von einem Client (PC, Handy…) via WLAN eintreffen. Der ESP32 untersucht die Anfrage auf ihren Inhalt, leitet dann entsprechende Aktionen ein und sendet schließlich eine Antwort an den Client zurück, z. B. in Form einer HTML-Seite. Genau das werden wir jetzt verwirklichen. Unser Server soll Aufträge zum Umschalten der LED aus dem ersten Teil entgegennehmen und ausführen. Danach erwarten wir eine Erfolgsmeldung als Webseite. Die Grafik verdeutlicht noch einmal die Zusammenhänge.

Fürs erste begnügen wir uns aber damit, dass der Server auf dem ESP32 als Antwort ein "Hallo Welt" zurücksendet. Die zweite Ausbaustufe besteht dann darin, dass wir dem Server mitteilen, wie er einen Schalter zu betätigen hat. Wir erwarten dann ferner eine Erfolgsbotschaft über den Schalterzustand.

Das "Hallo Welt"-Beispiel ist sehr einfach zu codieren und auch schnell erklärt. Schauen wir uns das MicroPython-Script dazu an. Den Teil für die Verbindungsaufnahme zum Accesspoint kennen wir ja bereits. Was konkret noch zu tun ist zeigt die folgende Grafik in den sechs orangen Feldern
.



Das Programm **server.py** übernimmt den Serverpart. Sie können den Text wieder durch copy-and-paste in eine neue Datei in µPyCraft übernehmen und via µPyCraft zum ESP32 übertragen. Wie das geht, habe ich weiter oben beschrieben.

```
# ******************** Server - Schleife **********************
"""
Hier wird zunaechst ein listening-Socket für den Serverprozess
erzeugt. Dieser wird an die IP aus der Bootsequenz, sowie eine
Portnummer gebunden, welche der Programmierer vorgibt.
Die Serverschleife wartet jetzt auf eingehende Verbindungsanfragen.
Mittels der accept-Methode nimmt der Server Anfragen an und
erzeugt in Folge einen Clientsocket, auf dem der weitere Daten-
verkehr abgewickelt wird. Sobald der Server den Job an den
Clientsocket abgegeben hat, kehrt er in die listening-Schleife
des Serversockets zurück, um weitere Anfragen anderer Clients
entgegen zu nehmen.
"""
try:
  import usocket as socket
except:
  import socket

def web_page():
  html = '<html><head><meta name="viewport" \
```

```
  content="width=device-width, initial-scale=1"></head> \
  <body><h1>Hallo Welt!</h1></body></html>'
  return html

portNum=9192
print("Fordere Server-Socket an")
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('', portNum))   # an lokale IP und Portnummer 9192 binden
server.listen(5)          # Akzeptiere bis zu 5 eingehende Anfragen
print("Empfange Anfragen auf ",STAconf[0],":",portNum, sep='')
while 1:
  c, addr = server.accept()  # anfrage entgegennehmen
  print('Got a connection from %s' % str(addr))
  request = c.recv(1024) .decode("utf-8")
  print('Content = %s' % str(request))
  response = web_page()
  c.send('HTTP/1.1 200 OK\n')
  c.send('Content-Type: text/html\n')
  c.send('Connection: close\n\n')
  c.sendall(response)
  c.close()
```

Datei: server1.py
Laden Sie das Script herunter, transferieren Sie es in den Workspace von µPyCraft. Damit es dort sichtbar wird, vergessen Sie bitte nicht, aus dem **File**-Menü **Reflush Directory** aufzurufen. Ziehen Sie das Script nun ins **device**-Directory des ESP32.

Wie arbeitet das Programm?

Es beginnt mit einem mehrzeiligen Kommentar (Blockkommentar). Erstreckt sich ein Kommentar oder eine Textkonstante (String) über mehrere Zeilen, wird der Text in dreifache Anführungszeichen gestellt.

```
try:
  import usocket as socket
except:
  import socket
```

Diese vier Zeilen dienen der Fehlerbehandlung. Mit **try** wird versucht das Modul **usocket** einzubinden. Klappt das Unterfangen, dann wird das Modul importiert und ist unter dem Namen **socket** ansprechbar. Tritt beim Importversuch ein Fehler auf, weil z. B. **usocket** nicht existiert oder nicht erreichbar ist, dann wird eine "Ausnahme (exception) geworfen", die das Programm mit **except** abfängt, um eine Fehlerbehandlung zu starten. Falls also usocket nicht importiert werden kann, wird ersatzweise versucht, **socket** einzubinden. Damit im weiteren Programm keine Unterscheidung zwischen den Namen des Moduls erforderlich ist, wird **usocket** mit dem Alias **socket** versehen. Durch die Fehlerbehandlung wird der Programmablauf nicht gestört, freilich weiß der Benutzer nicht, welches Modul nun wirklich geladen wurde, weil der Import quasi transparent erfolgt.

```
def web_page():
  html = """<html><head><meta name="viewport"
  content="width=device-width, initial-scale=1"></head>
  <body><h1>Hallo Welt!</h1></body></html>"""
  return html
```

Die Funktion web_page() erzeugt den minimalen Codeinhalt der Webseite, die der Server auf Anfragen eines Clients zurücksenden soll, als Wert der Variable **html**. Der Inhalt von **html** ist dann der Rückgabewert der Funktion. Die dreifachen Anführungszeichen, als """ oder ''' geschrieben, erlauben die Definition des Strings über mehrere Zeilen. Wir hatten das vor Kurzem schon für den Kommentar benutzt.

```
portNum=9192
print("Fordere Server-Socket an Port",portNum,"an")
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Der Variable **portNum** wird die, für den Server vorgesehene Portnummer zugewiesen. Es wird eine Fortschrittsmeldung ausgegeben und dann lassen wir ein Server-Socket-Objekt erzeugen, indem wir den Constructor der socket-Klasse aufrufen. Der erste Parameter **socket.AF_INET** steht für die Adressfamilie IPV4 und **socket.SOCK_STREAM** kennzeichnet eine TCP-Verbindung, **socket.SOCK_DGRAM** steht für UDP-Verbindungen.

```
server.bind(('', portNum))
```

Jetzt binden wir den Serversocket an die IP-Adresse, die der ESP32 vom DHCP-Server bekommen hat, zusammen mit der Portnummer, die wir ihm verpasst haben. In den Anführungszeichen kann auch eine IP übergeben werden. Die doppelten runden Klammern sind kein Schreibfehler! Die bind()-Methode erwartet als Parameter ein sogenanntes Tupel. Das ist eine Sammlung von Feldern, die durch Kommas getrennt und in runden Klammern zusammengefasst sind.

```
server.listen(5)
```

Die **listen**()-Methode bringt den Serversocket dazu, auf eingehende Anfragen zu lauschen. Die 5 gibt an, wie viele Anfragen gleichzeitig angenommen werden können. Dafür ist die Methode **accept**() zuständig, zu der wir gleich kommen werden. Vorher lassen wir uns mitteilen, unter welcher IP und welchem Port der Server erreichbar ist. STAconf[0] ist das Feld aus dem Programm wifi_connect.py. Weil die Liste **STAconf** dort global deklariert wurde, sind die Felder immer noch verfügbar, wenn ein Programm wie **server.py** im globalen Namensraum aufgerufen wird oder wenn wir vom Pythonprompt ausarbeiten.

```
print("Empfange Anfragen auf ",STAconf[0],":",portNum, sep='')
```

Die Ausgabe im Terminalfenster von µPyCraft sollte jetzt so ähnlich aussehen, wenn wir zuerst wifi_config.py und dann server.py starten - Rechtsklick auf das Programm im **device**-Directory und dann **Run**.

Ausgabe:

```
Verbindungsstatus:  STAT_GOT_IP
STA-IP:             8.8.8.50
STA-NETMASK:        255.255.255.0
STA-GATEWAY:        10.0.1.20


Fordere Server-Socket an
Empfange Anfragen auf 8.8.8.50:9192
```

```
while 1:
  c, addr = server.accept()  # anfrage entgegennehmen
```

Die Anfrage (request) von einem Client wird durch die **accept**()-Methode entgegengenommen. Als Ergebnis gibt accept() ein Connectionobjekt und die Socketdaten, IP und Port, des Clients zurück. Das Connectionobjekt c ist ein weiterer Socket, auf dem die nun folgende Kommunikation zwischen Client und Server abgewickelt wird. Sobald die Datenverbindung steht, wird der Serversocket geschlossen und der ESP32 geht wieder auf Horchposten für neue Verbindungen. Aller weiterer Datenverkehr wird über den Clientsocket c abgewickelt.

```
  print('Got a connection from %s' % str(addr))
  request = c.recv(1024).decode("utf-8")
```

Wir erfahren mit der Printanweisung, von welcher IP und mit welchem Port ein Client angeklopft hat. Bis zu 1024 Bytes der Nachricht, die im **bytes**-Format vom Connectionobjekt geliefert wird, lesen wir aus dem Datenpuffer und lassen uns das in ASCII-Text übersetzen. Mit dem Datentyp bytes hatten wir bereits bei der Abfrage der Accesspoints in der näheren Umgebung zu tun.

Vom Browser des PCs senden wir jetzt eine Anfrage an die Adresse 8.8.8.50:9192, die uns das Serverobjekt vorhin mitgeteilt hat und werden mit Hallo Welt begrüßt.

Im µPyCraft-Terminal wird angezeigt, von wem der Request kam, und dass es eine GET-Anfrage war, die sich auf das Server-Rootverzeichnis "/" bezog. Letztere Information wird später noch wichtig.

Got a connection from ('8.8.8.34', 57755)
Content = GET / HTTP/1.1
…

Danach kommt ein Schwall von Daten, die der Browser von sich aus ohne unser Zutun gesandt hat. Die Nachricht vom Browser hat uns der folgende Printbefehl beschert.

```
  print('Content = %s' % request)
```

Nun wird es Zeit, die Antwort zu generieren und zu versenden, die wir bereits im Bowserfenster gesehen haben. Danach wird die Connection-Instanz c geschlossen.

```
  response = web_page()
  c.send('HTTP/1.1 200 OK\n')
```

```
c.send('Content-Type: text/html\n')
c.send('Connection: close\n\n')
c.sendall(response)
c.close()
```

Der Server teilt dem Client (Browser) zunächst das verwendete Protokoll mit und sagt, dass die Anfrage OK war. Das beinhaltet auch der Zahlencode 200. Die Serverantwort wird Text und HTML-Code enthalten und schließlich folgt der Seitentext, den die Funktion **web_page()** geliefert hat.

Die kurzen Kopfzeilen für das http-Protokoll werden mit der **send**()-Methode verschickt. Für längere Datenströme benutzt man besser **sendall**() um sicher zu gehen, dass auch alle Zeichen gesendet werden.

So, auch das ist geschafft, Sie haben die erste Antwort von Ihrem ESP32-Server. Jetzt fehlen für diesen Teil nur noch zwei Kleinigkeiten. Der Serverprozess sollte ja eine LED schalten. Und vielleicht wäre es nicht verkehrt, wenn das wifi_connect-Script von sich aus das server-Script starten könnte, wenn wir das so wollen.

Der Server schaltet eine LED

Erinnern Sie sich noch an eine der Hausaufgaben aus dem ersten Teil?

Die Lösung zu eben dieser Hausaufgabe brauchen Sie nämlich gleich. Am besten ändern Sie die Schaltung jetzt schon ab, bevor wir uns die Software anschauen.

Stoppen Sie jetzt als erstes den laufenden Server mit dem Stoppbutton der µPyCraft-IDE. Wenn der Pythonprompt im Terminalfenster erschienen ist, können Sie loslegen.

Die folgenden Zeilen müssen an den Anfang des Scripts wifi_connect.py eingefügt werden.

```
import esp          # lästige Systemmeldungen aus
esp.osdebug(None)

import gc           # Platz für Variablen schaffen
gc.collect()
```

Für unser trickreiches Vorgehen brauchen wir eine Taste, mit der wir den Startvorgang abbrechen können, falls das nötig ist. Da hilft die Klasse **Pin**, die anfangs bereits importiert wurde. Um spezielle Methoden zur Steuerung von ESP32/ESP8266 zur Verfügung zu haben, wird das Modul esp importiert. Wir nutzen es sofort, um lästige Diagnosemeldungen abzuschalten. Außerdem holen wir uns das Modul **gc** an Bord. **gc** steht für garbage collection, was wörtlich übersetzt Müllsammlung bedeutet, hier aber als Speicherbereinigung zu verstehen ist. Der Aufruf von **gc.collect**() entfernt Leichen von Objekten aus dem RAM-Speicher und schafft Platz für neue Aufgaben.

```
job="call"          # Serverstart vorbereiten
timeDelay=5         # Verzögerung für Unterbrechung in Sekunden
```

Diese beiden Variablen habe ich für die Steuerung des Abbruchs beziehungsweise für den Start des Programms **server.py** definiert.

An das Ende des Scripts **wifi_connect.py** hängen Sie die folgenden Zeilen an. Sie geben uns die Möglichkeit, nach erfolgreicher Kontaktaufnahme, den automatischen Aufruf des Serverscripts zu zu lassen oder zu unterbinden. Beim Experimentieren mit **server.py** muss dann nicht jedes Mal die WiFi-Verbindung wieder neu hergestellt werden, wenn das Serverscript nach Abbrüchen neu gestartet werden muss. Wir starten server.py also von Hand aus dem device-Directory bis alles 1a funktioniert. Dann basteln wir in einer der nächsten Folgen eine Lösung, bei der der ESP32 autonom ohne PC läuft – versprochen.

```
# GPIO2 soll ein Ausgang werden
led = Pin(2, Pin.OUT)

# LED aus
led.value(0)

# GPIO5 brauchen wir als Eingang fuer den Abbruch-Taster
# no-Taste fuer Startunterbrechung druecken (no = normally open)
# Der Kontakt schliesst also beim Druecken
taste=Pin(5,Pin.IN)

# aktuellen Zeitstempel ermitteln
start = time()
# Abbruchzeitpunkt der while-Schleife berechnen
end = start + timeDelay
# laufende Zeit auf Startzeitpunkt setzen
currentTime=start

# wenn in der while-Schleife nicht abgebrochen wird, startet
# der exec-Aufruf die Ausführung der Hauptschleife (server)
# Abbruch der Taste an GPIO5 (Pulldown + HIGH-aktiv)

led.value(1)
while currentTime < end:
    currentTime = time()    # Zeit aktualisieren
    if taste.value() == 1:  # falls Taste gedrückt wurde
      job="break"          # Zeichen auf Abbruch

# Server starten oder Bootsequenz abbrechen
if job == "call":
  led.value(0)
  exec(open('server.py').read(),globals())
else:
  print("Die Bootsequenz wurde abgebrochen!")
  led.value(0)
```

Was passiert in den eingefügten Zeilen genau? Zum Vergleich können Sie hier die ergänzte Datei wifi_connect.py herunterladen.
Datei: wifi_connect2.py
Wir brauchen die LED an Pin GPIO2, schalten den Pin auf Ausgang und die LED aus. Den Pin GPIO5 konfigurieren wir als Eingang für eine Taste. Die Beschaltung haben Sie sicher bereits geändert.

In **start** merken wir uns den momentanen Stand der Systemzeit in Sekunden. In **end** kommt die um den Wert in **timeDelay** erhöhte Zeit. Die laufende Zeit in **currentTime** wird auf Anfang gesetzt.

Als Hinweis auf die Möglichkeit, jetzt den Start von **server.py** abzubrechen, schalten wir die LED ein. In der while-Schleife wird jetzt ständig die Tastenstellung abgefragt und die Variable für die laufende Zeit aktualisiert. Falls die Taste gedrückt wurde, wird der Wert von **job** auf "break", also Abbruch, gestellt. Dabei ist es unerheblich, ob die Taste einmal oder mehrmals gedrückt wurde.

Die Lösung mit der Schleife ist nötig, damit ein Tastendruck zu (fast) jedem beliebigen Zeitpunkt registriert und gespeichert wird. Würde man eine Zählschleife und sleep() verwenden, wäre es bedeutend schwieriger, den Tastendruck zu registrieren. Würde man den ESP32 gar für 5 Sekunden in den Schlaf schicken, hätte man gar keine Chance, denn während der Schlafenszeit passiert rein gar nichts.

Der Rest ist einfach. Wenn nach Beendigung der Schleife **job** noch den Inhalt "call" hat, wurde die Taste nicht gedrückt und der Server kann gestartet werden. Zuerst schalten wir die LED aus. Dann wird die Datei **server.py** geöffnet und der Inhalt des Dateiobjekts über die Methode **read**() gelesen.

```
open('second.py').read()
```

Die **exec**()-Methode fasst den gelesenen Text als Programmtext auf und führt ihn aus und zwar im globalen Namensraum. Das bedeutet, dass alle bisher definierten Variablen und Objekte weiterleben und wir auf sie zurückgreifen können.

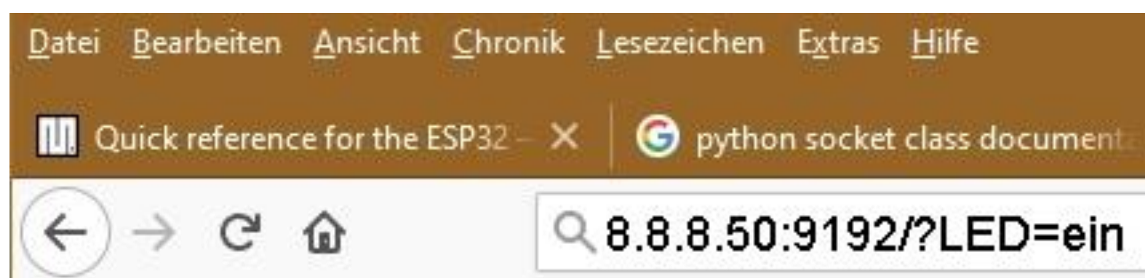
Wurde der Wert von Job geändert, endet nach dem Ausschalten der LED die Programmausführung und der Pythonprompt wird im Terminalfenster von µPyCraft angezeigt.

Zum Test wird **wifi_connect.py** gespeichert, zum ESP32 hochgeladen und im **device**-Directory mit Rechtsklick und Run aus dem Kontextmenü gestartet, einmal mit und einmal ohne Tastendruck.

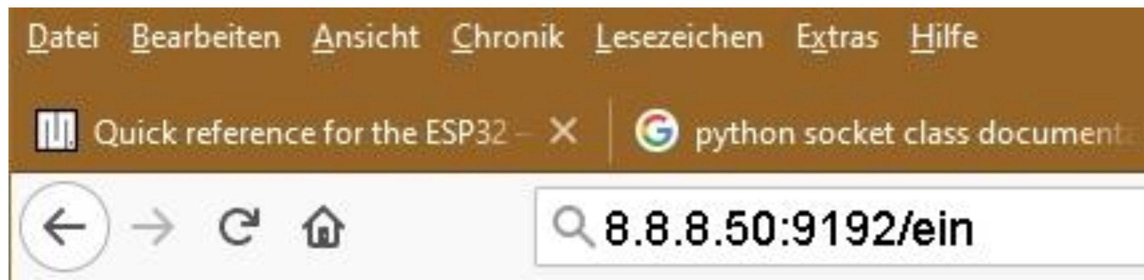
Funktioniert es? Prima!

Für die Schaltfunktion des Servers sind nur ein paar Kleinigkeiten nötig.

Es gibt verschiedene Wege die Signale "ein" und "aus" zu senden. Ein Weg wäre es, die Information als Übergabeparameter in der URL-Zeile des Browsers zu platzieren. Das sähe dann etwa so aus. **?LED=ein** bezeichnet hier kein Verzeichnis auf dem Server, sondern stellt einen Anfrageparameter dar. Diese Art der Datenübermittlung werden wir im 4. Teil verwenden, um den ESP32 via Browser zu steuern.



Noch einfacher ist die zweite Methode. Sie bedient sich der Möglichkeit, Unterverzeichnisse auf dem Server in die Anfrage einzubinden.



Natürlich gibt es auf unserem Server kein Verzeichnis "/ein". Muss es auch nicht geben. Wir parsen die Anfrage des Clients auf dem Server einfach nur nach den Zeichenfolgen "/ein" und "/aus". Ob daraufhin eine Datei aus dem "Verzeichnis" zurückgesandt oder eine andere Aktion ausgelöst wird, ist letztlich egal.

Wenn wir die Adresse "8.8.8.50:9192/ein" abschicken, kommt am Server folgendes an.

```
Got a connection from ('8.8.8.50', 53198)
Content = GET /ein HTTP/1.1
```

Wir erkennen sofort die interessante Stelle im empfangenen Text. Nur müssen wir Python dazu bringen, auch danach zu fahnden. Der Text der Anfrage steht in der Variable **request** bereit.

```
request = c.recv(1024).decode("utf-8")
```

Das haben wir schon. Die folgenden Zeilen sind neu.

```
slashPos= request.find("/") # Erste Position eines "/" im Text finden

# die naechsten 3 auf den "/" folgenden Zeichen holen
aktion  = request[slashPos+1:slashPos+4]

print('Aktion = %s' % aktion)

response = web_page(aktion)
```

Wir veranlassen Python zunächst, nach einem Slash "/" zu suchen. Das geschieht mit der Stringmethode **find**(). Wir merken uns die Position in **slashPos**. Es kommt 4 heraus, denn die Zeichenpositionen in Strings werden ja ab 0 gezählt.

Um eine Position weiter steht das "e" von "ein". Insgesamt sind es 3 Zeichen und weil Python die Obergrenze eines Bereichs ausschließt, rechnen wir mit 4, um das ganze "ein" zu bekommen. Wir kopieren somit aus dem String in **request** eine Scheibe (aka Slice) vom 5. bis zum 7. Zeichen. Den Text merken wir uns in **aktion**, melden den Erfolg im Terminal und übergeben ihn der Funktion, welche den Text für die Rückgabe zusammenbaut. Jetzt sollte klar sein, weshalb das eine Funktion macht und der Text nicht einfach im Hauptteil des Skripts zusammengebastelt wird. Die Übersicht wäre beim Teufel, denn der Text könnte noch viel umfangreicher und vielfältiger sein und die Programmstruktur verschleiern.

Zu diesem Thema ist auch anzumerken, dass ein Programm zwar durch Kommentare an Informationsgehalt gewinnt. Andererseits verliert man durch zu viele Kommentare genauso den Überblick über die Programmstrukturen. Sie können das anhand der Scripts testen, indem Sie die reichlichen Kommentare löschen. Sie dürfen dabei nur die Einrückungen der Programmzeilen nicht verändern.

Wann kommt denn nun das Schalten? Jetzt! In der Funktion **web_page**() ergänzen wir den Parameter **schalten** und ändern den Text wie folgt ab.

```
def web_page(schalten):
  mach = schalten.upper()
  if mach == "EIN" or mach == "AUS":
    act = "Die LED ist "+ mach
  else:
    act = "Ungueltiger Befehl: " + schalten
  html = """<html><head><meta name=\"viewport\"
  content=\"width=device-width, initial-scale=1\"></head>
  <body><h1>LED-Schalter</h1> """
  html = html + act+'.</body></html>'
  if mach == "EIN":
    led.value(1)
  elif mach == "AUS":
    led.value(0)
  return html
```

Damit mögliche Fehler oder Missbrauch ausgeschlossen werden können, müssen einige Dinge geprüft werden, bevor geschaltet wird.

```
  mach = schalten.upper()
```

Weil es verschiedene Schreibweisen für unsere beiden Begriffe gibt, wandeln wir den übergebenen Text in Großbuchstaben um.

```
  if mach == "EIN" or mach == "AUS":
    act = "Die LED ist "+ mach
  else:
    act = "Ungueltiger Befehl: " + schalten
```

Für die Rückmeldung belegen wir die Variable **act** vor. Dazu benutzen wir den Inhalt von **mach** oder setzen eine Fehlermeldung zusammen, in die der Übergabeparameter **schalten** eingebaut wird.

```
  html = """<html><head><meta name=\"viewport\"
  content=\"width=device-width, initial-scale=1\"></head>
  <body><h1>LED-Schalter</h1> """
  html = html + act+'.</body></html>'
```

html wird mit der Einleitung der HTML-Seite belegt, dann werden der Text für die Rückmeldung und der Schluss der HTML-Seite drangesetzt.

```
  if mach == "EIN":
    led.value(1)
```

```
elif mach == "AUS":
  led.value(0)
return html
```

Der Schaltprozess kommt ganz zum Schluss, ist reichlich unscheinbar und dokumentiert sich wahrscheinlich inzwischen selbst. Speichern Sie das Script nach allen Änderungen ab und senden Sie es zum ESP32. Um mit einem sauberen System zu starten, drücken Sie die Resettaste am ESP32-Modul. Starten Sie wifi_connect.py und lassen Sie es durchlaufen. Wenn der Server Bereitschaft meldet, sollten Sie mit den beiden URLs die LED ein- und ausschalten können.

8.8.8.50:9192/ein
8.8.8.50:9192/aus

Wie immer können Sie hier den gesamten Text des server-Scripts herunterladen.
Datei: server2.py

So, jetzt haben Sie sich eine Pause verdient. aber Sie wissen jetzt,

- wie man einen ESP32 mit MicroPython mit einem WLAN-Router verbindet
- wie ein Serverprozess funktioniert und auf dem ESP32 aufgesetzt wird
- wie man den Server zum Schalten von allen möglichen Dingen verwenden kann.
- Wie man Zeitschleifen baut, in denen außer Warten auch noch andere Dinge passieren können
- wie man auch ohne PC-Tastatur Soll-(ab)-bruchstellen in ein Programm einbaut

Statt einer LED können Sie genauso gut ein Relais oder einen Optokoppler über die GPIO-Pins ansteuern. Ferner können Sie von einem Script aus ein weiteres starten oder diesen Start durch Tastendruck abbrechen. Sie haben einige Feinheiten von Python kennengelernt, haben mit Strings und bytes-Typen gearbeitet und, und, und…

Im nächsten Teil stelle ich Ihnen ein OLED-Display und einen Buzzer vor, zeige Ihnen, wie man Timer in MicroPython einsetzt und die Analogkanäle abfragt. Das alles verbinden wir zum Schluss zu einer autonom laufenden Anwendung.

Lust auf Hausaufgaben? – Bitteschön:

- Was passiert eigentlich, wenn Sie in der URL keinen Slash eingeben? Welcher Wert steht dann in slashPos?
- Falls Sie schon einmal HTML-Seiten von Hand gestrickt haben. Können Sie eine einfache Seite mit Links herstellen, mittels deren die LED geschaltet werden kann?
- Können Sie in den Antworttext vom Server die IP und die Portnummer des Clients mit einbauen?
- Fügen Sie dem Aufbau eine zweite LED hinzu, die aufblinkt, wenn eine Anfrage am Server eingeht.

Das ist Ihnen schon jetzt alles viel zu kompliziert?

Wenn Sie der Meinung sind, MicroPython bringt's immer noch nicht, dann würde ich das zwar bedauern. Sie können Ihren ESP32/ESP8266 aber jederzeit wieder mit der Arduino-

IDE programmieren, indem Sie die MicroPython-Firmware auf dem Board einfach mit einem kompilierten Arduino-Sketch überschreiben. Mehr ist nicht zu unternehmen.

Lösung zu den Hausaufgaben aus Teil 1

- Können Sie mit MicroPython auf Ihrem Modul ermitteln, welche Zahlenwerte hinter den Konstanten Pin.IN, Pin.OUT und Pin. OPEN_DRAIN stecken?

Geben Sie am Pythonprompt in µPyCraft die folgende Anweisung ein. Achtung Python ist case sensitive.
print(Pin.IN, Pin.OUT, Pin.OPEN_DRAIN)
Antwort: 1 3 7

- Warum bleibt die LED eigentlich nach dem ersten Durchlauf zum Schluss an?

```
from machine import Pin
from time import sleep

led = Pin(1, Pin.OUT)
i = 0
while i <=10:
  led.value(not led.value())
  sleep(0.5)
  i += 1
```

Es liegt daran, dass die Zählung von i bei 0 beginnt. Angenommen die LED ist anfangs aus. Beim ersten Durchlauf geht sie an, während i noch 0 ist. Dann geht i auf 1.
Beim nächsten Durchlauf: LED aus, i wird 2.
Bei ungeraden Werten von i am Ende der Schleife wurde die LED zuvor eingeschaltet, als i noch geradzahlig war.
Der letzte Durchlauf findet statt, wenn i = 10 ist. Die LED geht an, i wird 11, die Schleife wird beendet und die LED bleibt an.

- Welche Möglichkeiten gibt es, das Programm so zu verändern, dass die LED zum Schluss aus ist?

Führen Sie genau eine der folgenden Änderungen durch.

  - i = 1 statt i = 0
  - while i<10:
  - while i<=9

- Können Sie das Programm dazu bringen, den Wert des Durchlaufzählers i auszugeben?

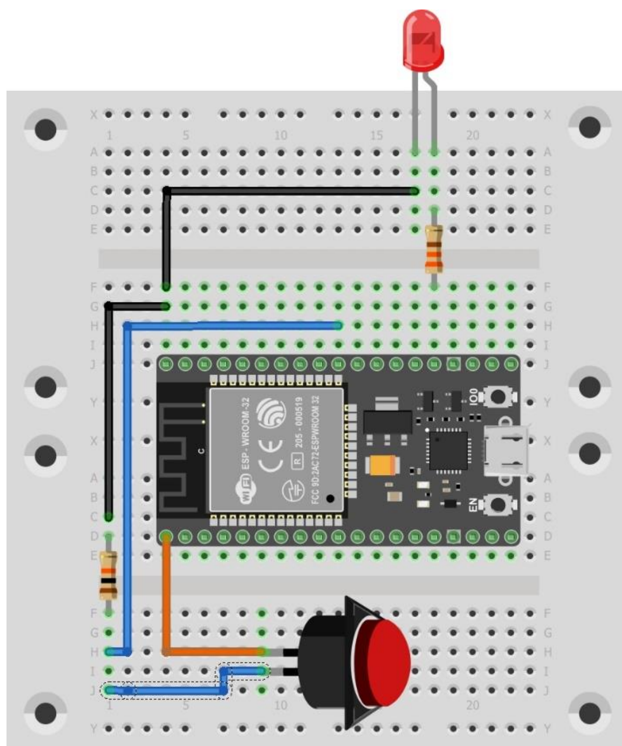
Fügen Sie die fett gedruckte Zeile ein.

```
while i <=10:
  led.value(not led.value())
  sleep(0.5)
```

```
i += 1
print(i)
```

- Ändern Sie die letzte Schaltung und das Programm so ab, dass das Negieren der Tasteninformation nicht nötig ist.

Der Kontakt schließt beim Betätigen des Tasters. Dadurch wird der Pegel an GPIO5 auf GND-Potential gezogen. Die eingelesene 0 wird von Python als False = falsch interpretiert, während ein Wert ungleich null als True = wahr angesehen wird. Soll die Taste im gedrückten Zustand beim Einlesen eine 1 liefern, dann muss an GPIO5 ein 3,3V-Pegel liegen. Wir erreichen das, wenn der Widerstand vom Tastenanschluss nicht an 3,3V sondern an GND gelegt wird und der an GND liegende Anschluss des Tasters an 3,3V angeschlossen wird. Diese Schaltung benötigen Sie auch für den oben beschriebenen Abbruch der Startsequenz. Die Fritzing-Grafik zeigt die geänderte Schaltung. Im Programm muss man jetzt die fett gedruckte Zeile ändern.




```
from machine import Pin
from time import sleep

led = Pin(2, Pin.OUT)
button=Pin(5,Pin.IN)
while True:
  taste = button.value()
  if taste:
    print (taste)
    i = 0
    while i <=10:
      led.value(not led.value())
      sleep(0.5)
```

```
i += 1
```

Glossar:

Accesspoint-Interface	Netzwerkschnittstelle eines ESP32/ESP8622, über welche direkt die Verbindung zu einem ESP-Modul bereitstellt. Über diese Schnittstelle kann man das ESP-Modul direkt, z. B. von einem Handy aus erreichen. Diese Insellösung benötigt keinen Zugang zu einem WLAN. Siehe dazu auch Station-Interface
Alias	Ersatzname für ein Modul oder Objekt oder eine Variable.
Attribut (Property, Eigenschaft)	Speicherplatz, dessen Inhalt das Erscheinungsbild oder das Verhalten eines Objekts verändern kann
case sensitive	Der Begriff drückt aus, dass Groß-Kleinschreibung beachtet wird. Start, START und start sind also z. B. Namen für drei unterschiedliche Variablen.
Constructor(-Methode)	Methode einer Klasse, welche ein Objekt gemäß den Vorgaben erzeugt (Speicherplatz reserviert) und die Attribute mit Defaultwerten oder/und durch die in den Aufrufparametern übergebenen Werte vorbelegt
Credentials	geheime Zugangsdaten hier: SSID und Passwort des Accesspoints
Defaultwert	Standard Vorgabewert
DHCP	**D**ynamic **H**ost **C**onfiguration **P**rotocol Netzwerkdienst, der einem sich anmeldenden Netzwerkgerät eine IP-Adresse zuweist. Diese kann je nach Einstellung des Dienstes zufällig aus einem Pool von Adressen stammen oder einer bestimmten MAC-Adresse zugeordnet sein.
Hexcode	Darstellung von Zahlen im Hexadezimalsystem mit 16 Ziffern von 0…15 codiert durch 0…9,A,B,C,D,E,F; die Stellenwerte sind Potenzen von 16. Die Notation erfolgt in verschiedener Syntax (=Schreibweise). Beispiel: Der Wert 46 im Dezimalsystem entspricht 0x2E = 2Eh = $2E = 2*16 + 14
HTML-Seite	**H**yper-**T**ext **M**arkup **L**anguage der minimale Aufbau einer HTML-Seite ist folgender <html> <head> <title> Seitentitel optional </title> </head> <body>

	Hier steht der Seiteninhalt </body> </html>
Index, Indizierung	Die Nummerierung der Felder einer Liste zur eindeutigen Identifizierung; Der Index ist die Nummer eines Feldes über die der Inhalt angesprochen wird. L[7] liefert den Inhalt des 8. Feldes der Liste L zurück, weil Listen ab dem Index 0 laufen.
Instanz	Ein von einer Klasse abgeleitetes Objekt
Kaltstart	Start des Systems nach dem Einschalten
Klasse (class)	Sammlung von thematisch zusammengehörigen Konstanten, Attributen und Methoden, die den Aufbau später zu erstellender Objekte beschreibt
Liste	Eine Datenstruktur in CPython oder LUA, die einem Array in der Arduino-IDE ähnelt. Listenelemente können mittels Index wie in einem Array angesprochen werden. Die Indizierung beginnt bei 0. Wenn n die Anzahl von Elementen in einer Liste ist, ist n-1 der letzte Index.
MAC-Adresse	**M**edia-**A**ccess-**C**ontrol-Adresse Physikalische Adresse eines Netzwerkgeräts; sie besteht aus 6 Bytewerten (0..255), kennzeichnet Hersteller und Seriennummer und dient dem gezielten Ansprechen des Geräts Beispiel: 32-1f-5d-ab-c7-01
Methode (Funktion)	"Vorgangsbeschreibung" für die Änderung von Attributwerten eines Objekts oder zur Ausführung von Aktionen in Verbindung mit einem Objekt
Modul	Sammlung thematisch zusammengehöriger Attribute und Methoden, die als separate Datei abgelegt werden. Bei Bedarf kann man entweder das gesamte Modul oder nur einzelne Klassen in ein Programm importieren. Durch die Verwendung von Modulen wird die Programmdatei kürzer. Ferner können Module in mehreren Projekten eingesetzt werden. Sie sind vergleichbar mit den Bibliotheken (Libraries) der Arduino-IDE.
Namenskonvention von Python	Variablennamen und Funktionsnamen dürfen in Python • nicht mit einer Ziffer beginnen • kein Leerzeichen enthalten • keinen Bindestrich "-" enthalten
Objekt	Eine aus einem "Bauplan" einer Klasse erstellte Speicherstruktur, die in einem Programm z. B. bestimmte Handlungen ausführt oder der Darstellung von Inhalten dient. Praktisch alles,

	was in Python existiert ist ein Objekt, Zahlen, Strings, Variablen, Konstanten, Funktionen…
Parameter	Wert oder Verweis (Zeiger/Pointer) auf einen Wert, der in Form einer Liste an eine Methode übergeben wird.
Parser, parsen	Programmteil welches Eingaben zerlegt und deren Inhalt analysiert.
Port	a) Schnittstelle eines Rechnersystems oder eines Microcontrollers b) in MicroPython die Bezeichnung für das gesamte System, auf dem MicroPython läuft. Beispiele: Linux, ESP32, PY Board…
Sockets	"Portale", die es erlauben, zwischen Netzwerkgeräten Informationen auszutauschen; eine Seite arbeitet als Server und wartet auf Verbindungsanfragen, die andere Seite als Client, initiiert solche Verbindungen; steht die Verbindung, können Daten ausgetauscht werden
Station-Interface	Netzwerkschnittstelle eines ESP32/ESP8622, welche die Verbindung zu einem WLAN-Accesspoint bereitstellt. Über diese Schnittstelle kann man das ESP-Modul vom lokalen Netzwerk aus über den Accesspoint erreichen und mit ihm Daten austauschen. Das Station-Interface ist nicht direkt, z. B. von einem Handy aus erreichbar. Siehe dazu auch Accesspoint-Interface
Tupel	Datenstruktur in CPython und MicroPython, die die unveränderlichen Werte enthält, die bei der Initialisierung angegeben werden. Die Struktur ähnelt einem Hash in LUA oder Perl