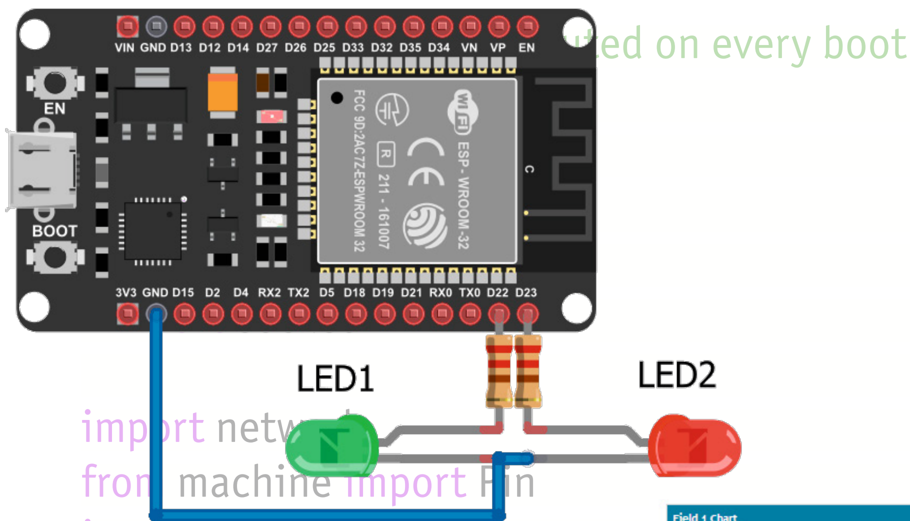
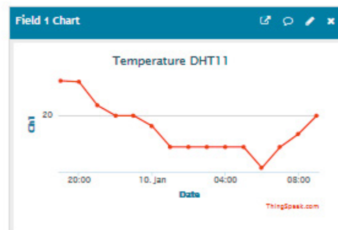
WLAN Fernsteuerung

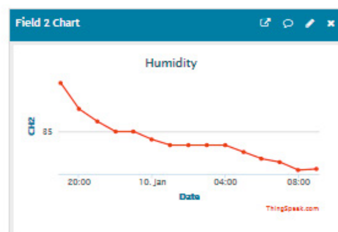
ESP32 – WLAN und Webserver in MicroPython

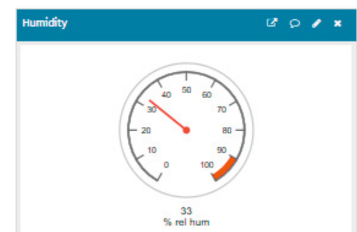
Eine der herausragenden Eigenschaften des ESP32-Controllers ist seine integrierte drahtlose Kommunikationsfähigkeit. Diese stellt einen entscheidenden Vorteil gegenüber beispielsweise den klassischen Arduino-Systemen dar. Dort war bei vielen Boards ein eigenes WLAN-"Shield" erforderlich, um sich in ein drahtloses Netzwerk einzuloggen. Beim ESP32 ist diese Funktionalität dagegen bereits mit integriert.

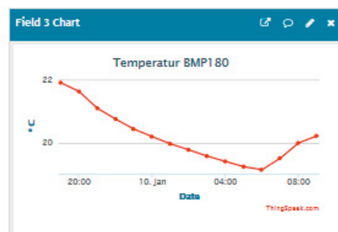


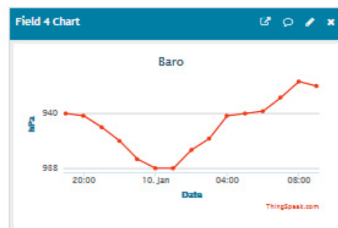

```
...ted on every boot
import netw...
from machine import Pin
import esp
esp.osdebug(None)
import gc
gc.collect()

ssid = 'xxxxxxxxxxxxx'
password = '123456789012345(

station = network.WLAN(netwc
station.active(True)
station.connect(ssid, passworc

while station.isconnected() ==
  pass

print('Connection successful')
print(station.ifconfig())
```

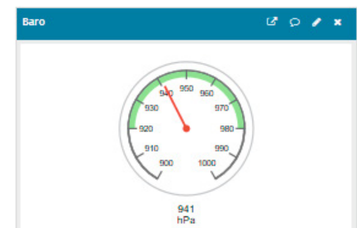












Drahtloses Schalten von Verbrauchern

In diesem Beitrag wird gezeigt, wie man insbesondere das WLAN-Interface nutzen kann, um Informationen mit dem heimischen Netzwerk auszutauschen oder auch die Steuerung von Anlagen oder Geräten zu ermöglichen. Unsere erste Anwendung ist ein Webserver, der es erlaubt, elektrische Verbraucher drahtlos über das WLAN zu schalten. Für die Programmentwicklung soll wieder MicroPython zum Einsatz kommen. Hinweise zur Installation und Anwendung der zugehörigen µPyCraft-IDE finden sich in den letzten beiden Ausgaben des ELVjournals [1].

Für dieses Projekt sind zwei Dateien erforderlich:

1. boot.py
2. main.py

Die Datei boot.py enthält den Code, der beim Booten nur einmal ausgeführt werden muss. Dieser umfasst das Importieren von Bibliotheken, Informationen zur Anmeldung im Netzwerk oder das Erstellen einer Verbindung mit dem lokalen WLAN:

```
# boot.py: This file is executed on every boot

try:                                              ❷
  import usocket as socket
except:
  import socket

import network
from machine import Pin                           ❸
import esp                                        ❹
esp.osdebug(None)
import gc                                         ❺
gc.collect()

ssid = 'xxxxxxxxxxxxx'                            ❶
password = '12345678901234567890'

station = network.WLAN(network.STA_IF)            ❻
station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:             ❼
  pass

print('Connection successful')
print(station.ifconfig())
```

Die Angaben für den Netzwerkzugang:

```
ssid = 'xxxxxxxxxxxxx'                            ❶
password = '12345678901234567890'
```

müssen natürlich durch die entsprechenden Daten des heimischen WLANs ersetzt werden. Diese finden sich meist auf der Rückseite des Routers oder aber in dessen Bedienungsanleitung.

Für die Programmierung kann wieder die bereits in den letzten Beiträgen verwendete µPyCraft-IDE verwendet werden. Dort werden die Dateien folgendermaßen erstellt:

1. Klicken auf die Schaltfläche „Neue Datei", um eine neue Datei zu erstellen
2. Über „Datei speichern" wird die Datei lokal auf dem ESP32 abgespeichert
3. Umbenennen der Dateien in „boot.py" bzw. „main.py"

Nun kann man den Code über Copy-and-paste in die Dateien einfügen. Alternativ können die Dateien auch im Feld „Ordner- und Dateien" der IDE direkt auf den Controller kopiert werden. Der Code selbst ist im Downloadpaket zum Beitrag unter [2] zu finden.

Sockets als Schnittstellen zum Internet

Der Webserver wird über sogenannten „Sockets" der Python-API erstellt. Sockets (engl. für Steckverbindung oder Steckdose) sind universelle Unterprogrammeinheiten, die als Kommunikationsendpunkte dienen. Sie erlauben es, Daten mit anderen Programmen oder Rechnern auszutauschen. Die anderen Programme können sich dabei auf demselben Rechner oder einem anderen, via Netzwerk erreichbaren Computer befinden. Die Kommunikation über Sockets erfolgt in der Regel bidirektional. Das heißt, Daten können sowohl empfangen als auch gesendet werden. Sockets bilden damit eine plattformunabhängige, standardisierte Schnittstelle zwischen dem Netzwerkprotokoll und der Anwendungssoftware.

Die Socket-Bibliothek für den ESP32 wird wie folgt importiert (s. boot.py):

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

Nach dem Import der Sockets muss die Netzwerkbibliothek (import network) importiert werden. Damit wird der ESP32 in die Lage versetzt, mit einem WiFi-Netzwerk zu kommunizieren. Um die GPIO-Pins verwenden zu können, ist zudem die Pin-Klasse aus dem Maschinenmodul erforderlich:

```
from machine import Pin                           ❸
```

Mit den folgenden Zeilen werden Debugging-Meldungen deaktiviert:

```
import esp                                        ❹
esp.osdebug(None)
```

Falls die Ausgabe von Debugging-Informationen gewünscht wird, können die Zeilen entfallen.

Dann wird ein sogenannter „Müllsammler" ❺ (engl. garbage collector – gc) aktiviert. Ein Garbage Collector stellt eine mögliche Form der automatischen Speicherverwaltung dar. Damit wird Speicherplatz freigegeben, der eventuell von Objekten belegt ist, die vom Programm nicht mehr verwendet werden.

Mit den Informationen für die Anmeldung im WLAN (Netzwerk-SSID und Kennwort des verwendeten Routers) kann der ESP32 eine Verbindung zum lokalen Router herstellen. Anschließend wird der ESP32 als WiFi-Station eingerichtet und aktiviert:

```
station = network.WLAN(network.STA_IF)            ❻
station.active(True)
```

Danach stellt der ESP32 mit der zuvor definierten SSID und dem Kennwort eine Verbindung zum Router her.

Über eine „while"-Schleife wird sichergestellt, ❼ dass das Programm nicht fortgesetzt wird, solange das ESP-Modul nicht mit dem lokalen WLAN-Netzwerk verbunden ist. Nach einer erfolgreichen Verbindung werden schließlich noch die Netzwerkparameter, u. a. die IP-Adresse des ESP32, ausgegeben.

Erstellen einer Webpage auf dem ESP32

In der main.py-Datei stehen die Anweisungen, mit dem der Webserver Daten bereitstellt und Aufgaben basierend auf den vom Client empfangenen Anforderungen ausführt. Hier erfolgt entsprechend auch der Aufbau der Server-Webpage:

```
# main.py: WebServer_WLAN_switch

led1 = Pin(23, Pin.OUT)          ❶
led2 = Pin(22, Pin.OUT)

def web_page():                  ❷
 if led1.value() == 1:           ❸
  gpio_state1="ON"
 else:
  gpio_state1="OFF"

 if led2.value() == 1:
  gpio_state2="ON"
 else:
  gpio_state2="OFF"

 html = """<html><head> <title>LED Server</title>
 <meta name=»viewport»   content=»width=device-width, initial-scale=1»>
  <link rel="icon" href="data:,">
  <style>html{font-family: Helvetica; display:inline-block; margin: 0px   auto; text-align: center;}

  h1{color: black; padding: 2vh;}p{font-size: 1.5rem;}
 .button1{display: inline-block; background-color: green; border:  none;          ❹
  border-radius: 14px; color: white; padding: 16px 40px;
text-decoration:   none; font-size: 30px; margin: 2px; cursor: pointer;}

 .button2{display: inline-block; background-color: red; border: none;
  border-radius: 14px; color: white; padding: 16px 30px;
text-decoration:   none; font-size: 30px; margin: 2px; cursor: pointer;}

 </style></head><body> <h1>WLAN switch</h1>

  <p>GPIO23 state: <strong>""" + gpio_state1 + """</strong>                       ❺
  </p><p><a href=»/led1=on»><button class=»button»>ON</button></a></p>
    <a href=»/led1=off»><button class=»button2»>OFF</button></a></p>

 <p>GPIO22 state: <strong>""" + gpio_state2 + """</strong>
  </p><p><a href=»/led2=on»><button class=»button»>ON</button></a></p>
    <a href=»/led2=off»><button class=»button2»>OFF</button></a></p>
 </body></html>"""
 return html

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)          ❻
s.bind(('', 80))
s.listen(5)

while True:
 conn, addr = s.accept()
 print('Got a connection from %s' % str(addr))                 ❼
 request = conn.recv(1024)                                     ❽
 request = str(request)

 led1_on = request.find('/led1=on')
 led1_off = request.find('/led1=off')
 led2_on = request.find('/led2=on')
 led2_off = request.find('/led2=off')
 if led1_on == 6:
  print('LED ON')
```

```
    led1.value(1)
  if led1_off== 6:
    print('LED OFF')
    led1.value(0)
  if led2_on == 6:
    print('LED ON')
    led2.value(1)
  if led2_off== 6:
    print('LED OFF')
    led2.value(0)

  response = web_page()        ❾
  conn.send(response)          ❿
  conn.close()
```

Hier werden zunächst Objekte für die anzusteuernden GPIOs (led1 an Pin 23 und led2 an Pin 22) definiert. ❶❷

Die darauf folgende Funktion web_page() gibt eine Variable namens „html“ zurück, die den kompletten HTML-Text zum Erstellen der Webseite enthält. Die Webseite hat die Aufgabe, die aktuellen GPIO-Zustände anzuzeigen.

Bevor der HTML-Text erzeugt wird, müssen die LED-Zustände überprüft werden. Sie werden in der Variablen gpio_state erfasst. ❸ Anschließend wird das Design der Webpage festgelegt. Natürlich würde eine detaillierte Einführung in den HTML-Code den Rahmen dieses Artikels sprengen. Trotzdem sollen die wesentlichen Punkte hier kurz erläutert werden. So werden im ersten Block die grundlegenden Parameter wie

Titel, Größe, Font

der Seite definiert. Dann folgt das Design für die beiden virtuellen „Buttons“ in den Farben Rot und Grün:

```
.button1{display: inline-block; background-color: green; border:  none;        ❹
   border-radius: 14px; color: white; padding: 16px 40px;
 text-decoration:   none; font-size: 30px; margin: 2px; cursor: pointer;}

  .button2{display: inline-block; background-color: red; border: none;
   border-radius: 14px; color: white; padding: 16px 30px;
 text-decoration:   none; font-size: 30px; margin: 2px; cursor: pointer;}
```

Nun kann die Webseite mit einer Überschrift gestartet werden. Danach wird die Variable gpio_state via Plus-Operator (+) in den HTML-Text eingefügt:

```
<p>GPIO23 state: <strong>""" + gpio_state1 + """</strong>        ❺
   </p><p><a href=»/led1=on»><button class=»button»>ON</button></a></p>
     <a href=»/led1=off»><button class=»button2»>OFF</button></a></p>

  <p>GPIO22 state: <strong>""" + gpio_state2 + """</strong>
   </p><p><a href=»/led2=on»><button class=»button»>ON</button></a></p>
     <a href=»/led2=off»><button class=»button2»>OFF</button></a></p>
  </body></html>"""
  return html
```

Nachdem der HTML-Code für die Webseite erstellt wurde, ist noch ein Listening-Socket erforderlich, um auf eingehende Anforderungen zu warten und den HTML-Text als Antwort zu senden. Dazu wird ein neues Socket-Objekt „s“ mit dem passenden Socket-Typ (STREAM-TCP) erstellt. Anschließend kann dieses Socket-Objekt über die bind()-Methode an eine Adresse (Netzwerkschnittstelle und Portnummer) angebunden werden. ❻

Die leere Zeichenfolge "" liefert die IP-Adresse des lokalen Hosts und damit die aktuelle IP-Adresse des ESP32. Die Portnummer 80 dient als Standard-Port.

Nun kann der Server die Verbindung zum Netz aufnehmen. Das Argument der zugehörigen listen()-Methode gibt die Anzahl der Verbindungen in der Warteschlange, maximal fünf, an.

In der „while“-Schleife wird auf Anfragen gewartet. Wenn ein Client eine Verbindung herstellt, ruft der Server die Methode accept() auf, um die Verbindung anzunehmen. Wenn ein Client eine Verbindung herstellt, speichert er ein neues Socket-Objekt, um Daten für die Variable conn zu akzeptieren und zu senden. Die Client-Adresse wird gespeichert, um eine Verbindung zum Server für die Variable addr herzustellen. Mit

```
print('Got a connection from %s' % str(addr))        ❼
```

wird die Adresse des Clients für Kontrollzwecke auf die Konsole ausgegeben.

Der Datenaustausch zwischen Client und Server erfolgt mit den Methoden send() und recv().

8 Die recv()-Methode empfängt die Daten vom Client-Socket. Das Argument der Methode gibt an, wie viele Daten maximal gleichzeitig empfangen werden können.

9 Die Variable response enthält den von der Funktion web_page() zurückgegebenen HTML-Text. Darin wird nach dem Schlüssel „ledx=on/off" gesucht und die LED entsprechend ein- bzw. ausgeschaltet.

10 Abschließend wird die Antwort mit den Methoden send() und sendall() an den Socket-Client gesendet und der Socket mit conn.close () geschlossen.

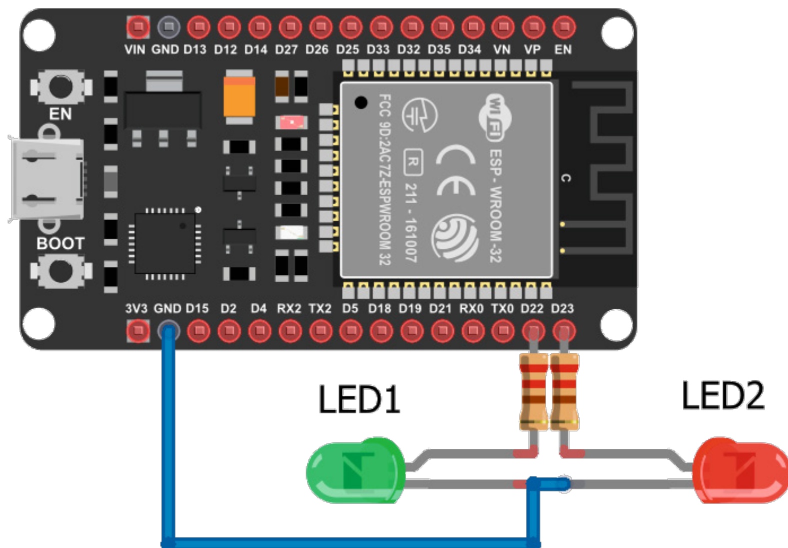


Bild 1: Der Server besteht lediglich aus dem ESP-Board und zwei LEDs.

Hardware zum LED-Webserver

Nach der Fertigstellung der Software kann der Hardwareaufbau in Angriff genommen werden. Dazu müssen lediglich zwei LEDs mit dem ESP-Board an GPIO-Port 22 und 23 verbunden werden. Natürlich dürfen die Vorwiderstände (2x 220 Ω) nicht vergessen werden (Bild 1).

Nach dem Hochladen der Dateien main.py und boot.py auf den ESP32 in den Geräteordner („device") sind die Tasten ESP EN/RST zu drücken, damit beide Dateien ausgeführt werden. Nach einigen Sekunden sollte eine Verbindung zum lokalen WLAN-Router hergestellt sein. Die zugehörige IP-Adresse wird in der Shell angezeigt (Bild 2).

Nun kann auf einem im gleichen LAN vorhandenen Rechner ein beliebiger Browser geöffnet werden. Nach Eingabe der oben genannten ESP-IP-Adresse sollte eine Webserverseite wie in Bild 3 angezeigt werden.

Wenn eine EIN-„Taste" angeklickt wird, leuchtet die zugehörige LED auf und der GPIO-Status wird auf der Seite aktualisiert. Nach dem Anklicken einer AUS-„Taste" erlischt die zugehörige LED.

Schalten größerer Lasten und Smartphone-Steuerung

Neben LEDs lassen sich auch z. B. Relais oder Leistungstransistoren schalten. Damit wird prinzipiell auch die Ansteuerung von 230-V-Geräten möglich. Nicht nur Lampen oder Beleuchtungseinrichtungen können nun zentral gesteuert werden, sondern auch Motoren, etwa für Jalousien oder Garagentore usw. Allerdings ist hier der folgende Hinweis zu beachten:

Wichtiger Hinweis:

Arbeiten an Netzspannung (230 V) dürfen nur von dafür ausgebildetem Fachpersonal durchgeführt werden.
Personen, die nicht über eine entsprechende Fachausbildung verfügen, dürfen daher auch keine Schaltsysteme für netzspannungsführende Anlagen installieren.
Die bei Halogen- oder LED-Systemen verwendeten Kleinspannungen von bis zu 12 V können dagegen gefahrlos mit einem Relais geschaltet werden. Allerdings ist auch hier zu beachten, dass die verwendeten Bauelemente und Kabel für die vergleichsweise hohen Ströme ausgelegt sein müssen.

Bild 4 zeigt ein Anwendungsbeispiel für eine WLAN-gesteuerte 12-V/20-W-Halogenlampe. Das Schalten der Lampe erfolgt hier auf der ungefährlichen 12-V-Niederpannungsseite über ein Relais-Modul.

Im Bedarfsfall kann der Code problemlos auf mehrere GPIOs erweitert werden. Der HTML-Text ist dann natürlich ebenfalls an die entsprechenden Gegebenheiten anzupassen.

Die Steuerung des EPS32-Systems muss nicht unbedingt nur über einen PC oder Laptop erfolgen. Da die

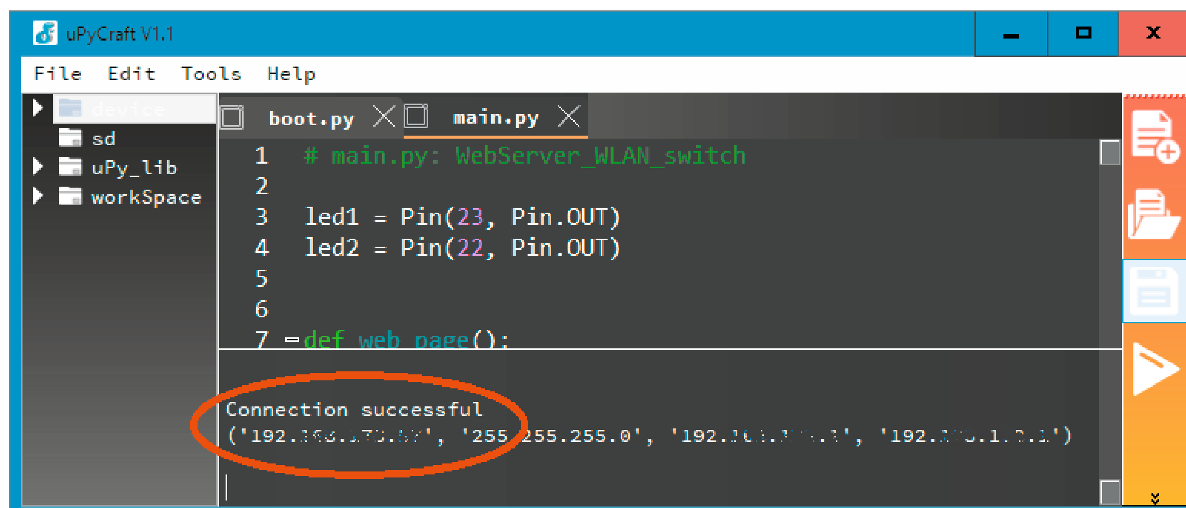


Bild 2: Erfolgreicher Verbindungsaufbau und aktuelle IP-Adresse des ESP32-Boards

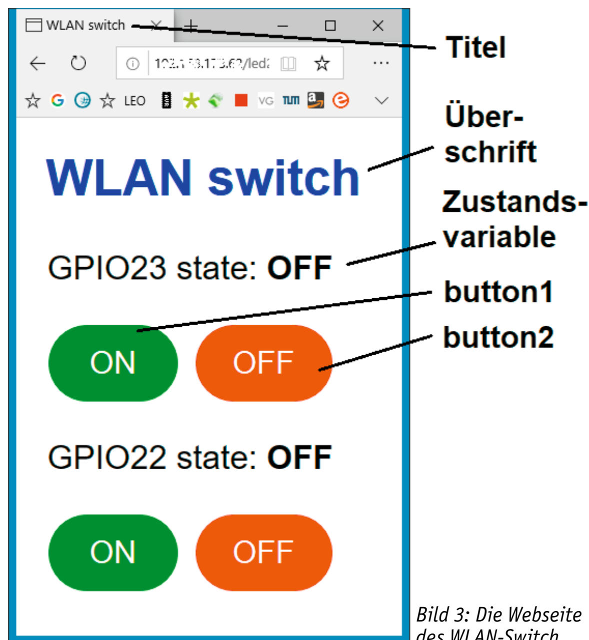


Bild 3: Die Webseite des WLAN-Switch

Bild 4: Schalten einer 12-V-Halogenlampe via ESP32 und Relaismodul

meisten mobilen Geräte wie Smartphones oder Tablets über ein WLAN-Interface verfügen, können auch diese Geräte für Steuerungsaufgaben verwendet werden. Bild 5 zeigt die ESP-Webseite in einem Smartphone.

Überwachung von Umweltparametern

Die Datenübertragung zwischen PC und ESP32 via WLAN funktioniert nicht nur in eine Richtung. Auch der ESP kann Informationen zum PC senden. Dies kann beispielsweise durch die Übertragung von Messwerten demonstriert werden. Neben der Steuerung von LEDs können auch Messwerte für Temperatur und Feuchtigkeit über einen Webserver auf dem ESP zu Verfügung gestellt werden.

Dafür wird lediglich ein DHT11-Sensor und ein 10-k-Ohm-Widerstand als Pull-up benötigt. Der Anschluss und die Auswertung dieses und anderer Sensoren wurde ja bereits im letzten Beitrag [1] dargelegt. Nun sollen die erfassten Werte allerdings nicht nur lokal in einem Display dargestellt, sondern über WLAN drahtlos übertragen werden.

Auch für dieses Projekt wird wieder eine „boot.py"-Datei und eine „main.py"-Datei benötigt. Die boot.py ist identisch mit der Version für den LED-Server. In der Zeile der main.py wird statt der LEDs nun der Sensor-Pin festgelegt:

```
sensor = dht.DHT11(Pin(14))
```
❶

Temperatur und Luftfeuchtigkeit werden über

```
def read_sensor():
```
❷

eingelesen. Die beiden Variablen „temp" und „hum" ❸ speichern die vom Sensor gelesene Temperatur und Luftfeuchtigkeit.

Mit der folgenden try/except-Konstruktion wird ❹ die Ausführung des Programms auch dann fortgesetzt, wenn eine Ausnahme auftritt. So kann verhindert werden, dass der Webserver abstürzt, wenn keine Daten vom Sensor gelesen werden können.

Bild 5: Das Smartphone als Steuerzentrale für den ESP32

Gültige Temperatur- und Feuchtigkeitsmesswerte sollten vom Typ „int", also integer sein. Daher kann mit der Funktion isinstance() geprüft werden, ob gültige Messwerte vorliegen. Die Funktion gibt „True" zurück, wenn die Variable dem eingefügten Datentyp entspricht, und „False", wenn dies nicht der Fall ist.

Wenn die Messwerte gültig sind, wird eine Meldung vorbereitet, welche die Temperatur- und Feuchtigkeitsmesswerte enthält:

```
msg = (b'{0:3.1f},{1:3.1f}'.format(temp, hum))
```
❺

Schließlich werden die Temperatur- und Luftfeuchtigkeitswerte mit return (msg) zurückgegeben. Falls keine gültigen Sensormesswerte vorliegen, wird eine Fehlermeldung erzeugt (Invalid sensor readings). Wenn der Sensor nicht ausgelesen werden kann, weil z. B. die Verbindung zur Messeinheit unterbrochen ist, wird ebenfalls eine Fehlermeldung zurückgegeben (Failed to read sensor).

Die Funktion web_page() gibt wieder eine HTML-Seite zurück. Genau wie beim LED-Server wird hier zunächst das allgemeine Design der Seite definiert. Danach werden zwei Routinen zur Anzeige der Temperatur und der Luftfeuchtigkeit erzeugt:

```
<p>
  <span class="dht-labels">Temperature:</span>
  <span>»»»+str(temp)+"""</span>
  <span class="units">&deg;C</span>
 </p>
 <p>
  <span class="dht-labels">Humidity:</span>
  <span>»»»+str(hum)+"""</span>
  <span class="units">%</span>
 </p>
```

Anschließend werden die bereits diskutierten Programmschritte zur Übergabe der Webseite ausgeführt. ❻

In der „while"-Schleife wird die Funktion read_sensor() aufgerufen, um die Sensorwerte auszugeben und die globalen ❼ Variablen temp und hum zu aktualisieren. Die vollständige main.py sieht damit so aus:

```
# main.py: ENVIRO Server DHT11

import dht
sensor = dht.DHT11(Pin(14))                ❶

def read_sensor():                         ❷
 global temp, hum                          ❸
 temp = hum = 0
 try:                                      ❹
  sensor.measure()
  temp = sensor.temperature()
  hum = sensor.humidity()
  if (isinstance(temp, int) and isinstance(hum, int)):
   msg = (b'{0:3.1f},{1:3.1f}'.format(temp, hum))    ❺
   print(msg)
   return(msg)
  else:
   return('Invalid sensor readings.')
 except OSError as e:
  return('Failed to read sensor.')

def web_page():
 html = """<!DOCTYPE HTML><html>
<head>
 <meta name=»viewport» content=»width=device-width, initial-scale=1»>
  <style>
  html { font-family: Arial; display: inline-block; margin: 0px auto; text-align: center;    }
  h2 { font-size: 2.0rem; } p { font-size: 2.0rem; }
  .units { font-size: 2.0rem; }
  .dht-labels{ font-size: 2.0rem; vertical-align:bottom; padding-bottom: 0px; }
 </style>
</head>
<body>
 <h2>ESP32 ENVIRO Server</h2>
 <p>
  <span class="dht-labels">Temperature:</span>
  <span>»»»+str(temp)+"""</span>
  <span class="units">&deg;C</span>
 </p>
 <p>
  <span class="dht-labels">Humidity:</span>
  <span>»»»+str(hum)+"""</span>
  <span class="units">%</span>
 </p>
</body>
```

```
</html>"""
  return html

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)   ❻
s.bind(('', 80))
s.listen(5)

while True:
  conn, addr = s.accept()
  print('Got a connection from %s' % str(addr))
  request = conn.recv(1024)
  print('Content = %s' % str(request))
  sensor_readings = read_sensor()                        ❼
  print(sensor_readings)
  response = web_page()
  conn.sendall(response)
  conn.close()
```

Nach dem Laden der Dateien auf den ESP32 kann wieder ein Webbrowser gestartet und die aktuelle IP-des ESP32 eingegeben werden. Nun erscheinen die Messdaten als Webpage (Bild 6).

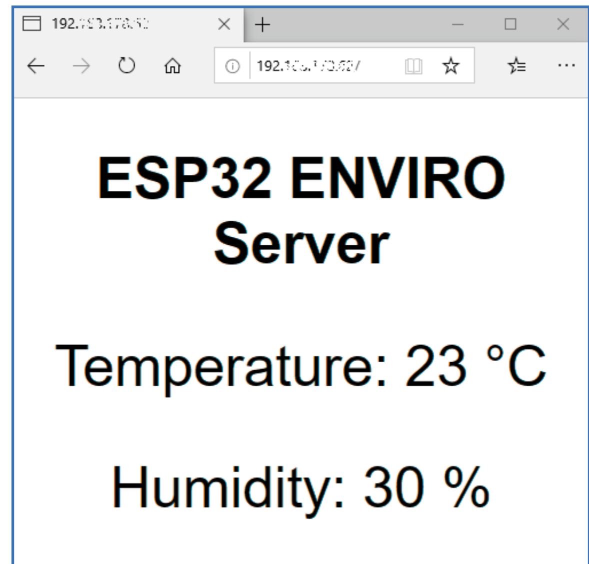


Bild 6: Der ENVIRO-Server in einem Browser

Global verfügbare virtuelle Klimastation

Bislang waren die Daten des ESP32-Servers nur im lokalen Netzwerk verfügbar. Ein Zugriff von außen über das globale Internet war nicht möglich. Dies ist aber natürlich in vielen Fällen wünschenswert, wenn man beispielsweise die heimischen Daten unterwegs, vom Arbeitsplatz aus oder im Urlaub überprüfen will. Prinzipiell wäre es zwar möglich, alle Werte global zur Verfügung zu stellen, allerdings muss dazu ein externer Zugriff auf das heimische WLAN freigeschaltet werden. Dies ist jedoch mit verschiedenen Sicherheitsrisiken verbunden. Aus diesem Grunde soll hier ein alternativer Weg vorgestellt werden.

Eine Möglichkeit, die Daten der eigenen Wohnung oder aus einer Büroumgebung etc. weltweit verfügbar zu machen, besteht in der Verwendung einer Online-Plattform. Im Folgenden werden die Messdaten an den IoT-Dienst „ThingSpeak.com“ gesendet. Die Werte werden dort protokolliert und sind dann weltweit abrufbar. Neben dem Zugriff über das Internet steht auch eine App für Mobilgeräte zur Verfügung (s. u.). Mit dem passwortgeschützten Zugriff auf die ThingSpeak-Plattform ist zudem ein ausreichender Datenschutz gewährleistet.

Das folgende, ebenfalls im Downloadpaket zu diesem Artikel enthaltene Python-Programm („ThingSpeak_server“) sorgt für die Übertragung der Werte zum Thinspeak-Server:

```
# main.py: ThingSpeak_Climate_Station
# ESP32
# Baro: BMP180, Thermo: 2x DS18x20, Hygro: DHT11,
# display: SSD1306

import network
import urequests

from machine import Pin, I2C
import ssd1306
import dht
from time import sleep
from bmp180 import BMP180

data_time_interval = 300 # 300 secs = 5 mins

# i2c pin assignmet
i2c = I2C(-1, scl=Pin(22), sda=Pin(21))

import esp
esp.osdebug(None)

import gc
gc.collect()

# baro initialization BMP180
bmp = BMP180(i2c)
bmp.oversample = 2
bmp.sealevel = 101325

# hygro initialization DHT11
sensor = dht.DHT11(Pin(14))
DHT_cal = 1

# OLED dimensions and initialization SSD1306
oled_width = 128
oled_height = 64
oled = ssd1306.SSD1306_I2C(oled_width, oled_height, i2c)
d1 = 60 # column distance 1
d2 = 90 # column distance 1
h1 = 9  # line hight

ssid = 'xxxxxxxxxxxxx'
password = '12345678901234567890'

api_key = 'ABCDEFGHIJK12345'                ❶

station = network.WLAN(network.STA_IF)
station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())
```

```
while True:

  oled.fill(0)
  oled.text('Climate Station', 0, 0)

  # DHT11: Temp & Humiture
  sensor.measure()
  temp_DHT11 = sensor.temperature() + DHT_cal
  hum_DHT11 = sensor.humidity()
  print('Temp1: %3.1f C' %temp_DHT11)
  print('Humi : %3.1f %%› %hum_DHT11)
  temperature_string_DHT11 = str(int(10*temp_DHT11)/10)
  humidity_string_DHT11 = str(int(10*hum_DHT11)/10)
  oled.text('Temp1:', 0 ,h1)
  oled.text(temperature_string_DHT11, d1 ,h1)

  oled.text(' C', d2 ,h1)
  oled.text('Humi :›, 0 , 2*h1)
  oled.text(humidity_string_DHT11, d1 ,2*h1)
  oled.text(' %', d2 ,2*h1)

  # BMP180: Temp & pressure
  temp_BMP180 = bmp.temperature
  pres_BMP180 = int(bmp.pressure/100)
  print('Temp2: %3.1f C' %temp_BMP180)
  print('Baro : %3.0f hPa› %pres_BMP180)
  temperature_string_BMP180 = str(int(10*temp_BMP180)/10)
  pressure_string_BMP180 = str(pres_BMP180)
  oled.text('Temp2:', 0 ,3*h1)
  oled.text(temperature_string_BMP180, d1 ,3*h1)
  oled.text(' C', d2 ,3*h1)
  oled.text('Baro :›, 0 ,4*h1)
  oled.text(pressure_string_BMP180, d1 ,4*h1)
  oled.text(' hPa', d2 ,4*h1)

  print()
  oled.show()
  sleep(1)

  sensor_readings = {'field1':temperature_string_        ❷
DHT11,  'field2':humidity_string_DHT11, 'field3':temperature_string_
BMP180,  'field4':pressure_string_BMP180}

  print(sensor_readings)

  request_headers = {'Content-Type': 'application/json'}

  request = urequests.post(                               ❸
  'http://api.thingspeak.com/update?api_key=' + api_key,
  json=sensor_readings,headers=request_headers)

  print(request.text)

  request.close()

  sleep(data_time_interval)
```

Die wesentlichen Teile des Programms sind bereits bekannt. Das Auslesen der Sensoren und die lokale Darstellung in einem OLED-Display wurden ausführlich im letzten Beitrag zu dieser Serie erläutert. Auch die zugehörige Hardware kann unverändert übernommen werden.

Für die SSID und das Passwort müssen wieder die realen Daten aus dem heimischen Netzwerk eingegeben werden.

Neu ist die Angabe des API-Schlüssels:

```
api_key = 'ABCDEFGHIJK12345'                    ❶
```

Dieser wird auf dem ThingSpeak-Account zur Verfügung gestellt (s. u.). Er erlaubt die eindeutige Zuordnung der Daten zu einem bestimmten Gerät oder Controller. Die Erfassung der Daten erfolgt in den Zeilen

```
 sensor_readings = {'field1':temperature_       ❷
string_DHT11,  'field2':humidity_string_
DHT11, 'field3':temperature_string_
BMP180,  'field4':pressure_string_BMP180}
```

Die einzelnen Felder (fields) stehen dabei für die Grafik-Anzeigefelder auf der ThingSpeak-Webseite. Über

```
request = urequests.post(
 'http://api.thingspeak.com/update?api_key='
 + api_key,                                     ❸
 json=sensor_readings,headers=request_headers)
```

werden die Daten zur Webseite übertragen. Die Angabe des API-Keys sorgt dafür, dass die Werte dem jeweils korrekten Account zugeordnet werden. In dieser Version sind alle erforderlichen Informationen für den Netzzugang etc. bereits in der Datei main.py enthalten. Das „boot.py"-File kann daher leer bleiben.

Datenerfassung mit ThingSpeak und ThingView

In ThingSpeak können Daten in den unterschiedlichsten Formen dargestellt werden. Das internetbasierte IoT-System ist unter

thingspeak.com/

erreichbar. Die Webseite bietet eine umfassende Unterstützung für die verschiedensten Mikrocontrollersysteme an. Die Anmeldung auf der ThingSpeak-Plattform erfolgt über eine Login-Seite. Nach erfolgreicher Login-Anmeldung kann der eindeutige Zuordnungscode, der sogenannte API-Key, unter

Channels → MyChannels

abgefragt werden. Die Auswahlpunkte „Channel Settings" und „API Keys" liefern die gesuchten Werte. Nach dem Einfügen des Keys in das oben stehende Python-Programm können bereits Sensordaten auf die Webpage geladen und anschließend weltweit abgefragt werden.

Die eigene ThingSpeak-Webseite kann individuell eingerichtet werden. Die Datenkanäle sind unter

MyChannels → New Channel

einzeln konfigurierbar. An dieser Stelle lassen sich dann mehrere Kanäle definieren. Im oben stehenden Programm wurden ja bereits vier sogenannte Felder (Fields) festgelegt:

1. Temperaturwert des DHT11-Sensors
2. Luftfeuchte
3. Temperaturwert des BMP180-Sensors
4. Aktueller Luftdruck

Nach der Eingabe der Felder ist die Einrichtung der Datenbasis abgeschlossen und alle gesammelten Daten lassen sich in verschiedenen Darstellungen grafisch anzeigen. Bild 7 zeigt ein Beispiel zur Darstellung der Sensordaten.

Die Grafiken können individuell gestaltet werden. Verschiedene Farben und Farbkombinationen sorgen

für eine breite Palette an Möglichkeiten. Dabei ist für jedes Messergebnis ein zeitlicher Verlauf und eine grafische Darstellung des aktuellen Wertes wählbar.

Die Achsen, die Anzahl der angezeigten Messpunkte und die Zeiträume lassen sich ebenso anpassen wie die Linientypen usw. Darüber hinaus können auch virtuelle Instrumente angezeigt werden. Bei diesen sind zudem bestimmte Sollwert- oder Gefahrenbereiche farblich markierbar. So lassen sich sehr übersichtliche und informative Datendisplays erzeugen.

Mit ThingView steht zudem eine spezielle App zur Auswertung der Daten zur Verfügung. Damit können weltweit alle Messwerte schnell und einfach mit Mobilgeräten wie Tablets oder Smartphones überprüft werden. Die App kann kostenlos über die bekannten Appstores (z. B. Play-Store) geladen werden.

Egal ob man im Urlaub die heimischen Werte ablesen will oder zu Hause das aktuelle Klima an der Arbeitsstelle prüfen möchte, mit der App stehen sämtliche Möglichkeiten offen. Nach der Eingabe der Channel-ID und des zugehörigen API-Keys sind die Daten auf jedem Mobilgerät verfügbar.

Die App übernimmt alle Windows-Einstellungen wie Farbe, Zeitskala, Diagrammtyp und Anzahl der Ergebnisse. Bild 8 zeigt die App in Aktion.

Bild 7: Darstellung von Sensorwerten in ThingSpeak

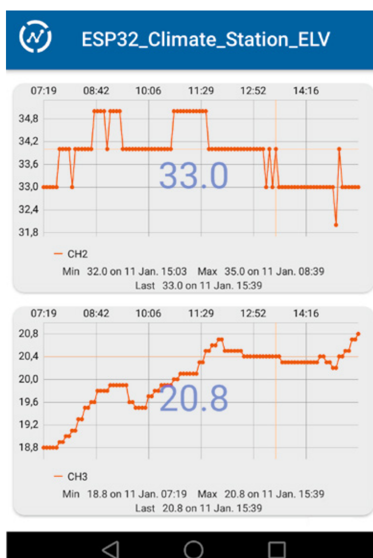


Bild 8: Mit der ThingView-App können alle Daten weltweit und via Smartphon abgefragt werden.

Fazit

In diesem Beitrag wurde gezeigt, wie man Sensorwerte drahtlos über WLAN vom ESP32 zu einem PC oder Smartphone übertragen kann. Zudem wurden Möglichkeiten vorgestellt, wie man Messwerte über eine IoT-Plattform global zur Verfügung stellen kann. Aber nicht nur die Datenübertragung vom Controller zum PC oder Mobilgerät ist möglich. Mit MicroPython kann auf dem ESP auch ein Webserver installiert werden, der das drahtlose Schalten via WLAN ermöglicht. Damit steht auch umfangreichen Projekten zum Thema IoT, Physical Computing oder Home-Automation nichts mehr im Wege. ELV

Material	Bestell-Nr.
Entwicklungsplatine NodeMCU mit ESP32	145164
Temperatur-/Feuchtigkeitssensor DHT11	250445

Weitere Infos:

[1] ELVjournal 1/2020: Messen und Steuern mit MicroPython, Bestell-Nr. 251149
ELVjournal 6/2019: Einstieg in MicroPython, Bestell-Nr. 251084
[2] Download-Skripte: de.elv.com, Bestell-Nr. 251135 (Downloads)