edv-buchversand.de

MicroPython

Das Handbuch für Maker und IoT-Development

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

Kapitel 4

Daten speichern, teilen und anzeigen

Sie haben nun die grundlegenden Elemente von MicroPython kennengelernt und wissen, wie Sie Logik in Kontrollstrukturen aufbauen und in Funktionen kapseln. Das ist wichtig, reicht aber nicht aus, denn Sie wollen die Daten ja auch speichern, über ein Netzwerk teilen und anzeigen. Wie das geht, schauen wir uns nun an.

4.1 Dateien und Dateisystem

Das Modul `os` in MicroPython stellt grundlegende Funktionen für den Umgang mit dem Dateisystem und betriebssystemnahe Informationen bereit. In Abschnitt 3.5.2 hatte ich Ihnen unter os das Modul bereits vorgestellt. Hier möchte ich das aus Anwendungssicht vertiefen.

4.1.1 Skript-Struktur: main.py, boot.py

Im Zusammenhang mit dem WDT haben Sie bereits die Startsequenz eines MicroPython-Programms kennengelernt.

Das Programm *boot.py* wird stets zuerst ausgeführt und dient dem Low-Level-Setup. Neben der boardspezifischen Initialisierung kann beispielsweise das Netzwerk vorbereitet oder der Garbage Collector konfiguriert werden. Den Quelltext in *boot.py* sollten Sie kurzhalten; Endlosschleifen und blockierende Tasks sind nicht zulässig.

Das nach *boot.py* gestartete Programm *main.py* ist das eigentliche Anwendungsprogramm. Im Gegensatz zu *boot.py* darf *main.py* Endlosschleifen enthalten. Das Verhalten von MicroPython ist abhängig vom Programm *main.py*, wie Ihnen Tabelle 4.1 zeigt.

Endlosschleife in main.py	Verhalten
nein	Programm endet. REPL erscheint (>>>).
ja	Programm läuft dauerhaft. REPL ist blockiert (außer per Interrupt).

Tabelle 4.1: Verhalten von MicroPython gesteuert durch main.py

4.1.2 Dateisystem-Funktionen

Der guten Übersicht halber liste ich in Tabelle 4.2 die Dateisystem-Funktionen auf.

Funktion	Anweisung	Beispiel
Listet Verzeichnisse und Dateien.	`os.listdir([path])`	`os.listdir('/lib')`
Erstellt neues Verzeichnis.	`os.mkdir(path)`	`os.mkdir('data')`
Löscht (leeres) Verzeichnis.	`os.rmdir(path)`	`os.rmdir('data')`
Löscht eine Datei.	`os.remove(file)`	`os.remove('data.csv')`
Benennt Datei oder Verzeichnis um.	`os.rename(old, new)`	`os.rename('old.txt', 'new.txt')`
Wechselt in neues Verzeichnis.	`os.chdir(path)`	`os.chdir('/lib')`
Gibt das aktuelle Verzeichnis zurück.	`os.getcwd()`	`cwd = os.getcwd()`
Gibt Statusinformationen zu einer Datei oder einem Verzeichnis zurück. (Typ, Größe, Zeitstempel …)	`os.stat(path)`	`info = os.stat('test.txt')`
Gibt Informationen zum Dateisystem zurück (Speicherplatz).	`os.statvfs(path)`	`fs = os.statvfs('/')`

Tabelle 4.2: Funktionen Modul os

Mit den in Tabelle 4.2 gelisteten Funktionen lassen sich die erforderlichen Aktivitäten im Dateisystem durchführen und Informationen zum Dateisystem abfragen. Abbildung 4.1 zeigt Ihnen die in Listenform ausgegebenen Informationen zu einem ESP32-C3.

```
>>> os.stat('/')
(16384, 0, 0, 0, 0, 0, 0, 0, 0, 0)
>>> os.statvfs("/")
(4096, 4096, 512, 498, 498, 0, 0, 0, 0, 255)
```

Abbildung 4.1: Ausgaben os.stat('/') und os.statvfs('/')

Die Bedeutung der Indizes in beiden Informationen zeigen Tabelle 4.3 und Tabelle 4.4.

Index	Bedeutung
`st[0]`	Dateityp und Rechte (Bitmaske)
`st[1]`	Inode (meist 0 oder nicht genutzt)

Tabelle 4.3: Statusinformationen

Index	Bedeutung
`st[2]`	Gerät (meist 0)
`st[3]`	Anzahl Links
`st[4]`	User-ID (meist 0)
`st[5]`	Group-ID (meist 0)
`st[6]`	Dateigröße in Bytes
`st[7]`	Zugriffszeit (Epoch)
`st[8]`	Änderungszeit
`st[9]`	Erstellungszeit

Tabelle 4.3: Statusinformationen (Forts.)

In der Praxis sind `st[0]` und `st[6]` relevant.

Index	Bedeutung
`fs[0]`	`f_bsize` – Blockgröße in Bytes
`fs[1]`	`f_frsize` – Fragmentgröße
`fs[2]`	`f_blocks` – Gesamtanzahl Blöcke
`fs[3]`	`f_bfree` – freie Blöcke
`fs[4]`	`f_bavail` – für Programme verfügbare freie Blöcke
`fs[5]`	`f_files` – maximale Inodes (meist nicht genutzt)
`fs[6]`	`f_ffree` – freie Inodes
`fs[7]`	`f_favail` – verfügbare Inodes
`fs[8]`	`f_flag` – Mount-Flags
`fs[9]`	`f_namemax` – maximale Dateinamenslänge

Tabelle 4.4: Speicherinformation

Hier sind in der Praxis auch nur `fs[0]`, `fs[2]`, `fs[3]` und `fs[9]` relevant.

Etwas aussagefähiger kann die Ausgabe in dem kleinen Tool *fs_info.py* gestaltet werden (Listing 4.1). Das Programm selbst weist keine Besonderheiten auf. Die Rückgabewerte

der Funktion `os.statvfs(path)` werden miteinander verknüpft und ausgegeben. Abbildung 4.2 zeigt die deutlich lesbarere Ausgabe der Informationen zum Dateisystem.

```
import os
def fs_info(path="/"):
    fs = os.statvfs(path)
    block = fs[0]
    total = fs[2] * block
    free  = fs[3] * block
    used  = total - free
    print("File System Info:", path)
    print("-------------------------")
    print("Total   :", total, "Bytes")
    print("Used    :", used,  "Bytes")
    print("Free    :", free,  "Bytes")
    print("Blocks  :", fs[2])
    print("Max Name:", fs[9], "Characters")
fs_info()
```

Listing 4.1: Programmbeispiel fs_info.py

```
File System Info: /
-------------------------
Total   : 2097152 Bytes
Used    : 57344 Bytes
Free    : 2039808 Bytes
Blocks  : 512
Max Name: 255 Characters
```

Abbildung 4.2: Ausgabe Programmbeispiel fs_info.py

Abbildung 4.2 zeigt, dass 57.344 der insgesamt zur Verfügung stehenden 2.097.152 Bytes aktuell verwendet werden, wodurch noch 2.039.808 Bytes frei sind. Diese Ausgaben sind sicher wichtig, um den noch zur Verfügung stehenden Speicherplatz zu kennen, aber interessant ist auch, wer wie viel des Speicherplatzes belegt. In der Thonny IDE sind die Verzeichnisse und Dateien im **Files**-Window sichtbar (siehe Abbildung 4.3).

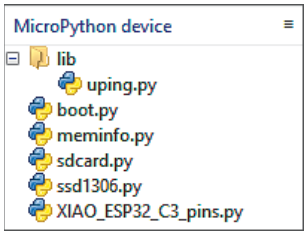

Abbildung 4.3: Dateien im Files-Window in Thonny

Aus MicroPython heraus liefert `os.listdir()` zwar die Namen der Verzeichnisse und Dateien, aber nicht deren Größe. Hier kann wieder ein kleines Tool wie *ls.py* Abhilfe schaffen (Listing 4.2).

```
import os
def ls(path="/"):
    for name in os.listdir(path):
        if path == "/":
            full = "/" + name
        else:
            full = path + "/" + name
        st = os.stat(full)
        mode = st[0]
        if mode & 0x4000:   # Verzeichnis
            print("🗀", name)
        else:               # Datei
            print("🗎", name, "-", st[6], "Bytes")
ls()
```

Listing 4.2: Programmbeispiel ls.py

Bevor der Aufruf `os.stat(full)` erfolgt, wird der Pfad zur einzelnen Datei systematisch erstellt. Handelt es sich um eine Datei, dann wird über `st[6]` ihre Größe abgefragt und zusätzlich zum Dateinamen ausgegeben. Abbildung 4.4 zeigt die Ausgaben von `os.listdir()` und `ls()` im Vergleich. Letztere zeigt zur in Abbildung 4.3 gezeigten Struktur auch noch die Dateigrößen an.

```
>>> os.listdir()
['XIAO_ESP32_C3_pins.py', 'boot.py', 'lib', 'meminfo.py', 'sdcard.py', 'ssd1306.py']
>>> ls()
  XIAO_ESP32_C3_pins.py - 381 Bytes
  boot.py - 139 Bytes
  lib
  meminfo.py - 697 Bytes
  sdcard.py - 7939 Bytes
  ssd1306.py - 4461 Bytes
>>>
```

Abbildung 4.4: Ausgaben os.listdir() und ls() im Vergleich

Die Arbeit mit den Dateisystem-Funktionen können Sie sehr gut im Programmbeispiel *rot_log.py*, einem rotierenden Datenlogger, nachvollziehen. Das Programmbeispiel finden Sie im Repository, hier möchte ich es aus Platzgründen nicht listen.

Dieser »FlashLogger« zeichnet sich durch rotierende Logdateien (*log0*, *log1* usw.) aus. Die Rotation erfolgt automatisch, wenn die maximale Dateigröße von *log0* erreicht ist. Vor je-

dem Schreiben erfolgt deshalb eine Speicherprüfung. Jeder Eintrag wird mit einem Zeitstempel versehen. Über das Kommando `ls_logs()` bekommen Sie eine Übersicht über die existierenden Logdateien.

Die Bedienung des FlashLoggers in der REPL zeigt Abbildung 4.5. Die resultierende Logdatei sehen Sie in Abbildung 4.6.

```
>>> log = FlashLogger(base_name="log", max_files=5, max_size=2048, warn_free_bytes=1024)
>>> log.write("Sensor started")
>>> log.write("Temperature: 23.5°C")
>>> log.write("Temperature: 24.1°C")
>>> log.ls_logs()
  log0 - 110 Bytes
>>> log.write("Temperature: 24.1°C")
>>> log.write("Temperature: 23.5°C")
>>> log.ls_logs()
  log0 - 174 Bytes
>>>
```

Abbildung 4.5: Ein- und Ausgaben Programmbeispiel rot_log.py in der REPL

```
=== Log started ===
[18:03:13] Sensor started
[18:03:41] Temperature: 23.5°C
[18:03:44] Temperature: 24.1°C
[18:04:09] Temperature: 24.1°C
[18:04:12] Temperature: 23.5°C
```

Abbildung 4.6: Dateiinhalt log0

4.1.3 Erweiterung des Speichers durch SD-Karten

Obwohl moderne Mikrocontroller mit immer mehr Speicher ausgestattet werden, kann bedingt durch die jeweilige Anwendung eine externe Speichererweiterung erforderlich werden.

Typische Anwendungsaspekte können

- Offlinebetrieb ohne Internetverbindung,
- Datenlogging von Sensoren, Audiodaten, IoT-Events u. a. m.,
- Konfiguration und Backups,
- große Dateien (Firmware, ML-Modelle, Bilder, Audio)

sein.

Selbst im Fall einer bestehenden Onlineverbindung kann ein Offline-Logging aus Backup- bzw. Sicherheitsgründen sinnvoll sein.

Es gibt eine Reihe von Mikrocontroller-Boards, die bereits einen SD-Card-Slot mit einem SDIO- oder SPI-Interface aufweisen. Die Unterschiede dieser beiden Interfaces zeigt Tabelle 4.5.

	SDIO / SDMMC	SPI Mode
Busbreite	1 Bit oder 4 Bit	1 Bit
Geschwindigkeit	sehr hoch	deutlich langsamer
Hardware	spezieller SD-Controller	normales SPI
Software	komplexer	sehr einfach
MCU-Support	nur manche MCUs	praktisch alle
Pins	viele	wenige

Tabelle 4.5: SDIO vs. SPI

Abbildung 4.7 zeigt mit dem LilyGo TTGO T8 v1.8 und dem Arduino MKR Zero zwei Boards, die einen SD-Card-Slot aufweisen und auch von MicroPython unterstützt werden.

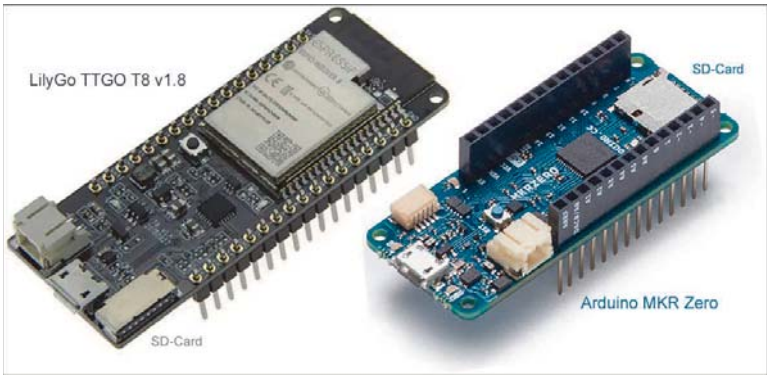

Abbildung 4.7: Mikrocontroller-Boards mit SD-Card (Auswahl)

Da das LilyGo TTGO T8 v1.8 mit einem ESP32-WROVER ausgestattet ist, ist es für zahlreiche Anwendungen das interessantere Board von den zweien.

Der ESP32 hat zwei SD Memory Card Controller (SDMMC-Hosts), die die Datenübertragung zur SD-Card optimieren. Host 0 führt an Slot 1, Host 1 an Slot 2. Damit MicroPython SDMMC nutzen kann (`machine.SDCard(slot=…)`), müssen die Pins des SD-Card-Slots hardware-seitig korrekt verdrahtet sein (Tabelle 4.6):

Signal	Slot 1 (Host 0) GPIO (typ.)	Slot 2 (Host 1) GPIO (typ.)
CLK	14	18
CMD	15	5

Tabelle 4.6: ESP32 SDMMC – Pins

Signal	Slot 1 (Host 0) GPIO (typ.)	Slot 2 (Host 1) GPIO (typ.)
D0	2	19
D1	4	optional
D2	12	optional
D3	13	optional
CS	abhängig / frei wählbar	abhängig / frei wählbar

Tabelle 4.6: ESP32 SDMMC – Pins (Forts.)

Die Pins können je nach ESP32-Variante abweichen, müssen aber mit dem SDMMC-Host verbunden sein, sonst schlägt `slot=…` fehl.

ESP32-WROVER-Kits führen die Pins nach außen, haben aber keinen SD-Card-Slot auf dem Board. Beim LilyGo TTGO T8 steht ein SD-Card-Slot auf dem Board zur Verfügung, doch leider folgt die Verdrahtung nicht den Vorgaben in Tabelle 4.6, und es bleibt nur das SPI-Interface für den Anschluss der SD-Card übrig.

SDIO- vs. SPI-Interface

Bieten Mikrocontroller SD-Card-Support durch SDMMC-Hosts, dann müssen die betreffenden Pins auch mit dem SD-Card-Slot verbunden sein, und der MicroPython-Port muss diese Kommunikation auch unterstützen. Sind diese Voraussetzungen nicht gegeben, dann bleibt das SPI-Interface die gängige Variante zum Datenaustausch zwischen Mikrocontroller und SD-Card.

Zum Test der SD-Card verwende ich das Programmbeispiel *ESP32_SD_test.py*, das ich hier aber nur ausschnittsweise erläutern möchte. Das verwendete Modul `sdcard` kann von *https://github.com/micropython/micropython-lib/blob/master/micropython/drivers/storage/sdcard/sdcard.py* heruntergeladen und auf dem Board installiert werden. Zu importieren sind außerdem die folgenden Module:

```
from machine import SPI, Pin
import os, sdcard, time
```

Die Initialisierung des SPI-Interface mit den Angaben aus dem Datenblatt erfolgt im nächsten Schritt:

```
cs = Pin(13, Pin.OUT, value=1)
spi = SPI(1, baudrate=10_000_000, polarity=0, phase=0,
          sck=Pin(14), mosi=Pin(15), miso=Pin(2))
```

Mit diesen Vorgaben kann nun das SD-Card-Objekt erstellt und als */SD* gemountet werden. Die Anweisungen `os.listdir("/sd")` und `ls("/sd")` zeigen nun den Inhalt des neuen Verzeichnisses an (siehe Abbildung 4.8):

```
try:
    sd = sdcard.SDCard(spi, cs)
    os.mount(sd, "/sd")
    print("SD mounted")
    print(os.listdir("/sd"))
    ls("/sd")
except OSError as e:
    print("SD init failed:", e)
```

Das Schreiben von Daten auf die SD-Card erfolgt nach dem Öffnen der Datei durch `open(filename, "w")` zum Schreiben durch die Anweisung `f.write()`.

```
# Write test file to sd card
print('\nWrite data to sd card...')
filename = "/sd/test.txt"
with open(filename, "w") as f:
    f.write("Hallo MicroPython!\n")
    f.write("Data written to SD card.\n")
```

Das (Zurück-)Lesen der Datei erfolgt nach dem Öffnen der Datei `open(filename, "r")` zum Lesen durch die Anweisung `f.read()`.

```
# Read file back
print('\nRead file content back from sd card...')
with open(filename, "r") as f:
    print("File content:")
    print(f.read())
```

Zum Schluss wird die auf der SD-Card erzeugte Datei wieder von der SD-Card entfernt.

```
# Remove file from sd card
print('\nRemove file from sd card...')
os.remove(filename)
```

Abbildung 4.8 zeigt die gesamten Ausgaben des Programmbeispiels. Zwischen den einzelnen Schreib- und Lesevorgängen wird jeweils der Inhalt des Verzeichnisses zur Kontrolle ausgegeben.

Im Repository können Sie zwei weitere Tools zum Test von SD-Cards finden. Das Programm *sd_benchmark.py* testet die Geschwindigkeit des Datenaustauschs für größere Dateien. Dem Einsatz in Sensorknoten näher kommt das Programm *logger_stress_test.py*.

```
ESP32 SD-Card Test via SPI
SD mounted
['System Volume Information', 'logger_test.bin']

Type  Size     Name
--------------------------
d        0 System Volume Information
-   192000 logger_test.bin

Write data to sd card...

Read file content back from sd card...
File content:
Hallo MicroPython!
Data written to SD card.

File Size: 44 Bytes

Type  Size     Name
--------------------------
d        0 System Volume Information
-   192000 logger_test.bin
-       44 test.txt
Free memory (Bytes): 15922823168

Remove file from sd card...

Type  Size     Name
--------------------------
d        0 System Volume Information
-   192000 logger_test.bin
Free memory (Bytes): 15922855936
```

Abbildung 4.8: Ausgaben Programmbeispiel ESP32-SD_test.py

4.2 Kommunikation

Gerade für IoT-Devices ist die Kommunikation ein essenzieller Aspekt, den ich in diesem Abschnitt eingehend behandeln möchte. Tabelle 4.7 zeigt die für die hier betrachteten Mikrocontroller vorhandenen Kommunikationstechnologien.

Technologie	Board-Support	Typische Reichweite	Verwendung
WiFi	ESP32, Pico W	30–100 m	Internet, MQTT, REST
LAN	ESP32, STM32	drahtgebunden	Internet, MQTT, REST
ESP-NOW	ESP32	30–100 m	Low-Latency Mesh
BLE	ESP32, nRF52	10–50 m	Sensor → App, Notifications
LoRa	ESP32 + SX127x	1–10 km	Sensornetzwerke, IoT
nRF24	ESP32, STM32	50–200 m	DIY Mesh, Remote

Tabelle 4.7: Kommunikationstechnologien

4.2.1 WiFi

Für die Konfiguration der WiFi-Verbindung weist das Modul `network` die Klasse `network.WLAN` auf. Es existieren zwei WiFi-Interfaces: eines für eine *Station* (STA) und eines für einen *Access Point* (AP). Station und Access Point sind zwei komplett unterschiedliche Rollen im Netzwerk.

WiFi vs. WLAN

WLAN (*Wireless Local Area Network*) ist ein allgemeiner technischer Begriff, während WiFi ein Markenname der WiFi Alliance ist – einer gemeinnützigen Organisation, die die gängigen und künftigen WLAN-Standards verwaltet. Im praktischen Sprachgebrauch sind beide Begriffe identisch.

Station bezeichnet einen Client oder Teilnehmer, der sich mit einem vorhandenen WLAN verbindet, um sich mit dem Internet zu verbinden, Daten zu senden (MQTT, API, Cloud) oder, allgemein gesprochen, Teil des Netzwerkes zu sein.

Ein Access Point baut ein eigenes WLAN auf, mit dem sich Clients verbinden können. Ein solcher AP kann zum Beispiel ein Konfigurations-Portal bereitstellen, ein Setup ohne Internet ermöglichen oder Webserver hosten.

Ich zeige Ihnen im Folgenden die Konfiguration von Station und AP, um anschließend beide Rollen auf einem Mikrocontroller zu vereinen.

Station (STA)

Um eine Verbindung zu einem WLAN-Router im Netzwerk aufzubauen, sind die einzelnen Schritte der Funktion `connect_wifi()` auszuführen (Listing 4.3).

```
import network, time
from credentials import WIFI_SSID, WIFI_PASSWORD
# Configuration Data in credentials.py
# Connect WiFi
def connect_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        print("Connect with WiFi...")
        wlan.connect(WIFI_SSID, WIFI_PASSWORD)
        while not wlan.isconnected():
            print('.', end="")
            time.sleep(1)
    print("\nWiFi connected:", wlan.ifconfig()[0])

connect_wifi()
```

Listing 4.3: Programmbeispiel connect_wifi.py

Die Anweisung `wlan = network.WLAN(network.STA_IF)` erstellt das Netzwerk-Interface im STA-Mode zum Router. Nach Aktivierung des Interfaces kann durch `wlan.connect (WIFI_SSID, WIFI_PASSWORD)` die Verbindung aufgebaut werden.

Die erforderlichen Einwahldaten SSID und Passwort habe ich in der Datei *credentials.py* versteckt. Im Abschnitt 3.5.3 habe ich diese Auslagerung bereits beschrieben. Zu Beginn des Programms wurden diese beiden Daten importiert.

Nehmen Sie das in Listing 4.4 gezeigte Template als Ausgangspunkt für die diversen Zugangsdaten, und benennen Sie die aktualisierte Version in *credentials.py* um, bevor diese auf dem MicroPython-Device gespeichert wird. Durch die Anweisung `from credentials import WIFI_SSID, WIFI_PASSWORD` stehen Ihnen dann SSID und Passwort im Programm zur Verfügung.

```
# credentials_template.py
WIFI_SSID = 'Your SSID'
WIFI_PASSWORD = 'Your Password'
PUSHOVER_USER_KEY  = 'Your Pushover User Key'
PUSHOVER_API_TOKEN = 'Your pushover API Token'
PUSHOVER_URL = 'Pushover URL'
```

Listing 4.4: Quelltext credential_template.py

Solange noch keine Verbindung zum WLAN aufgebaut ist, signalisiert im Programm *connect_wifi.py* eine Reihe von Punkten das Warten auf die Verbindungsaufnahme. Ist diese erfolgt, dann wird die zugewiesene IP-Adresse ausgegeben (siehe Abbildung 4.9).

```
Connect with WiFi...
....
WiFi connected: 192.168.1.230
```

Abbildung 4.9: WiFi-Verbindungsaufnahme

Access Point (AP)

Um einen Access Point im Netzwerk bereitzustellen, sind die einzelnen Schritte der Funktion `start_ap()` auszuführen (Listing 4.5).

```
import network
def start_ap():
    ap = network.WLAN(network.AP_IF)
    ap.active(True)
    ap.config(
        essid="ESP32_AP",
        password="12345678",    # mind. 8 Zeichen
        authmode=network.AUTH_WPA_WPA2_PSK
    )
```

```
    print("AP gestartet")
    print("IP:", ap.ifconfig()[0])
start_ap()
```

Listing 4.5: Programmbeispiel connect_ap.py

Die Anweisung `ap = network.WLAN(network.AP_IF)` erstellt das Netzwerk-Interface im AP-Mode. Nach Aktivierung des Interfaces können durch die Funktion `ap.config()` und die Werte `ESSID = "ESP32_AP"` das Passwort und der Authentifizierungs-Mode eingestellt werden. Der so erzeugte AP meldet sich dann im Netzwerk als `ESP32_AP` und erwartet das Passwort `12345678`. Sie sind vollkommen frei in der Wahl der Zugangsdaten.

Ist die Konfiguration erfolgt, wird die zugewiesene IP-Adresse ausgegeben (siehe Abbildung 4.10), und der ESP32_AP ist im Netzwerk sichtbar (siehe Abbildung 4.11).

```
AP gestartet
IP: 192.168.4.1
```

Abbildung 4.10: AP-Verbindungsaufnahme

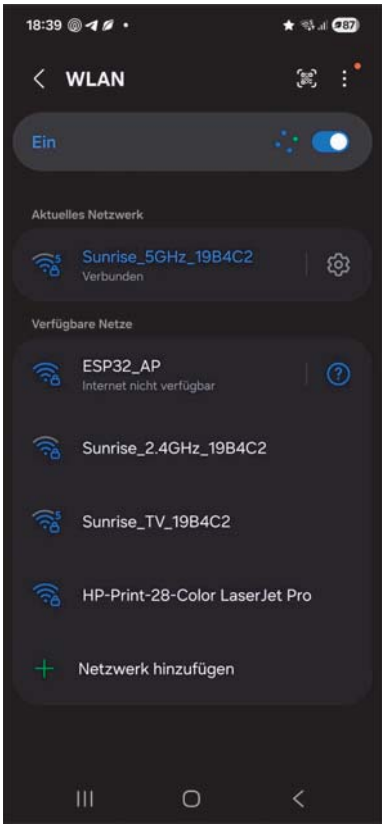

Abbildung 4.11: ESP32_AP im Netzwerk

STA und AP kombiniert

Durch die gleichzeitige Implementierung von STA und AP erhalten Sie beide Welten. Vielleicht ist Ihnen das schon mal bei der Konfiguration eines neuen IoT-Geräts begegnet.

Das noch nicht ins WLAN eingebundene Gerät startet und baut ein eigenes WLAN (hier **ESP32_AP**) auf. Mit dem Mobilphone kann eine Verbindung zum ESP32_AP aufgebaut werden. Die Zugangsdaten für das zukünftig zu verwendende Router-WLAN werden eingegeben und abgespeichert, und künftig verbindet sich das Gerät direkt als STA.

Das Programmbeispiel *STA_AP.py* zeigt den Aufbau der beiden Verbindungen, wobei hier die Zugangsdaten für den STA-Mode bekannt und wieder in *credentials.py* abgespeichert sind (Listing 4.6).

```
import network
from credentials import WIFI_SSID, WIFI_PASSWORD
ap = network.WLAN(network.AP_IF)
sta = network.WLAN(network.STA_IF)
ap.active(True)
ap.config(essid="ESP32_AP", password="12345678")
sta.active(True)
sta.connect(WIFI_SSID, WIFI_PASSWORD)
print("AP  IP:", ap.ifconfig()[0])
print("STA IP:", sta.ifconfig()[0])
```

Listing 4.6: Programmbeispiel STA_AP.py

Abbildung 4.12 zeigt die IP-Adressen für STA im Heimnetzwerk und den AP, der über die Verbindung zum ESP32_AP erreicht werden kann.

```
AP  IP: 192.168.4.1
STA IP: 192.168.1.229
```

Abbildung 4.12: IP-Adressen für AP und STA

Mit der Einwahl selbst ist es im geschilderten Fall aber noch nicht getan. Die folgenden Schritte sind erforderlich:

- WiFi Reset: `wifi_reset()`
- Laden vorhandener Zugangsdaten: `cfg = load_config()`
- Start STA: `sta = start_sta(cfg)`
- Start AP: `ap = start_ap()`
- Start Webserver zur Dateneingabe: `start_server()`

Im Programmbeispiel *web_config_pro.py* sind diese Aktivitäten vereint. Das Programm finden Sie im Repository unter *https://github.com/ckuehnel/MicroPython2026/blob/main/General/web_config_pro.py*.

Neu im Programm sind die HTML-Seite zur Eingabe der Zugangsdaten und der Webserver. Die HTML-Seite wird in der Funktion `html_page()` (Listing 4.7) aufgebaut, zum Webserver komme ich in Abschnitt 8.5.

```
def html_page():
    nets = scan_networks()
    options = ""
    for n in nets:
      options += f"<option value='{n[0]}'>{n[0]} ({n[1]}dBm)</option>"
    return f"""HTTP/1.0 200 OK
<html>
<h2>TTGO Pro Config Portal</h2>
<form action="/save">
SSID:<br>
<select name="ssid">{options}</select><br>
Password:<br>
<input name="password" type="password"><br><br>
<input type="submit">
</form>
</html>
"""
```

Listing 4.7: Quelltext HTML-Seite

Den eigentlichen HTML-Code habe ich im Quelltext hervorgehoben. Sie können diesen Teil in einem HTML-Editor betrachten und den Aufbau der späteren Website begutachten.

Damit im SSID-Feld die vorhandenen Netzwerke gelistet werden können, wird das Netzwerk zuerst gescannt, und SSID und Feldstärke in dBm werden in `options` gespeichert. Abbildung 4.13 zeigt die APs in meinem Netzwerk.

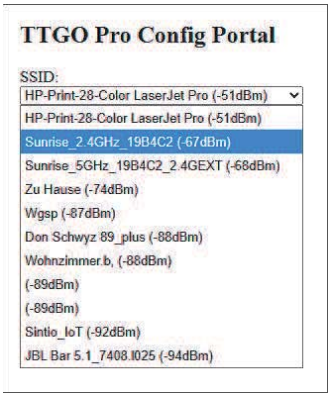

Abbildung 4.13: Aktuelle APs im Netzwerk

Rufe ich die STA-IP 192.168.1.229 vom Mobilphone aus auf, dann erscheint die wenig befriedigende Website aus Abbildung 4.14.



Abbildung 4.14: Website web_config_pro.py

Eine Anpassung an die Displaygröße des Mobilphones muss unbedingt vorgenommen werden.

Durch den Einbau von Viewport und größeren Schriften lässt sich das Erscheinungsbild deutlich verbessern. Den veränderten HTML-Code zeigt die aktualisierte Funktion `html_page()` (Listing 4.8). Das angepasste Programm *web_config_pro+.py* finden Sie ebenfalls im Repository.

```
def html_page():
    nets = scan_networks()
    options = ""
    for n in nets:
        options += f"<option value='{n[0]}'>{n[0]} ({n[1]}dBm)</option>"
    return f"""HTTP/1.0 200 OK
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>
body {{
    font-size: 20px;
    font-family: Arial;
    padding: 15px;
}}
```

```
input, select, button {{
    font-size: 20px;
    width: 100%;
    margin-top: 10px;
    padding: 10px;
}}
</style>
</head>
<body>
<h2>TTGO Pro Config Portal</h2>
<form action="/save">
SSID:<br>
<select name="ssid">{options}</select><br>
Password:<br>
<input name="password" type="password"><br><br>
<input type="submit">
</form>
</body>
</html>
"""
```

Listing 4.8: Quelltext erweiterte HTML-Seite

Beim Aufruf der STA-IP 192.168.1.229 vom Mobilphone aus wird nun die angepasste und deutlich besser lesbare Website erscheinen (siehe Abbildung 4.15).

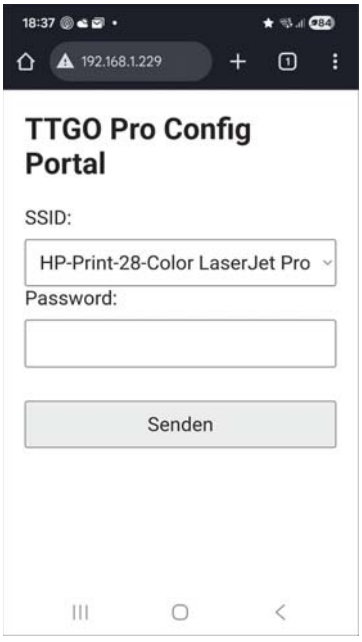

Abbildung 4.15: Website web_config_pro+.py

4.2.2 Ethernet

In den vorhergehenden Abschnitten haben Sie mit WiFi eine drahtlose Netzwerkverbindung kennengelernt. Bei kabelgebundenen LANs (*Local Area Networks*) ist Ethernet die meistbenutzte Technologie und kann als Pendant zu WiFi betrachtet werden.

Ethernet ist WiFi dann überlegen, wenn Sie ein Gateway oder einen Server betreiben, dauerhaft kommunizieren, 24/7-Onlineverfügbarkeit sicherstellen müssen und eine feste Installation haben.

Interessant ist außerdem die Möglichkeit, die Spannungsversorgung gleichzeitig über das Ethernet-Kabel vorzunehmen. Die PoE-Technik (*Power Over Ethernet*) ist gerade für stationäre IoT-Knoten eine interessante Variante.

Den grundsätzlichen Aufbau einer Ethernet-Schnittstelle zeigt Abbildung 4.16. Es gibt Mikrocontroller, die den MAC bereits enthalten und extern nur noch den PHY und die Magnetics benötigen. Mikrocontroller, die keinen MAC aufweisen, können über SPI einen externen TCP/IP-Controller, wie beispielsweise einen WIZnet W5500 oder Infineon CYW43439, ansteuern, der MAC und PHY beinhaltet.

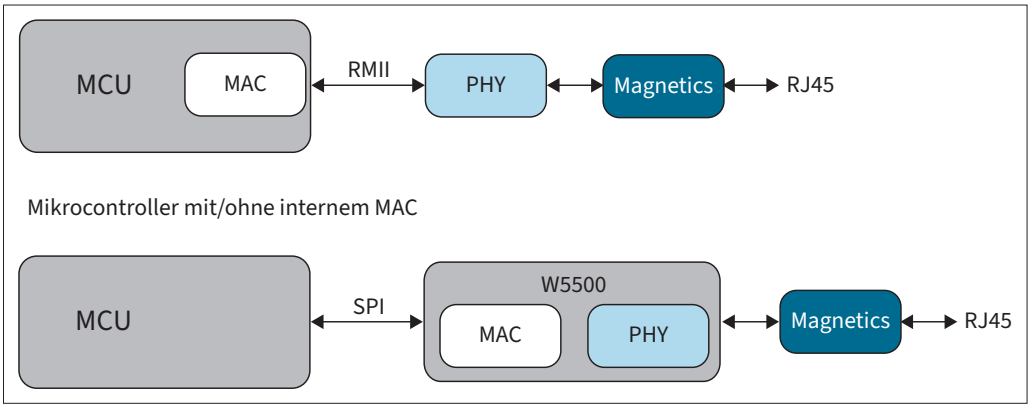

Abbildung 4.16: Aufbau einer Ethernet-Schnittstelle

Der Media Access Controller (MAC) ist die digitale Logik, die das Ethernet-Protokollhandling übernimmt, während der Physical Layer (PHY) die analoge, elektrische Schnittstelle zwischen MAC und den Magnetics darstellt. Beispiele für PHY-Chips sind LAN8720, DP83848, KSZ8081, RTL8211 (Gigabit). Die Magnetics sind bei Ethernet die Übertrager und Drosseln zwischen PHY und RJ45-Buchse. Sie sorgen dafür, dass die Signale galvanisch getrennt, sauber gefiltert und normgerecht aufs Kabel gehen. Oft sind alle diese Teile in der RJ45-Buchse integriert (»MagJack«). Abbildung 4.17 zeigt einen solchen MagJack.



Abbildung 4.17: MagJack

Ob der betreffende Port von MicroPython Ethernet unterstützt, finden Sie durch die folgenden Kommandos schnell heraus. Das verwendende Mikrocontroller-Board muss das dann natürlich hardware-seitig auch tun.

```
>>> import network
>>> if "LAN" in dir(network):
        print("LAN supported")
  LAN supported
```

Ich habe gute Erfahrungen mit dem Olimex ESP32-EVB-EA (*https://www.olimex.com/Products/IoT/ESP32/ESP32-EVB/*) gemacht, bei dem neben dem LAN- auch ein WLAN-Interface zur Verfügung steht, wodurch es für Bridge-Anwendungen sehr geeignet ist. Außerdem stehen microSD Card, CAN-Interface und zwei Relais (10 A/250 V AC Wechselspannung) mit externen Anschlüssen und Status-LED zur Verfügung.



Abbildung 4.18: Olimex ESP32-EVB-EA

Den Bootprozess für das Ethernet-Interface zeigt das Programmbeispiel *ETH_boot.py* (Listing 4.9). Während des Bootprozesses signalisieren das Relais und seine Status-LED den jeweiligen Status. Wichtig ist die LAN-Initialisierung. Sehen Sie sicherheitshalber im Schaltbild des Boards nach den Verbindungen von Mikrocontroller und PHY nach. Abbildung 4.19 zeigt Ihnen die Ausgaben des Programms während des Bootprozesses.

```
# ETH_boot.py - ESP32-EVB-EA LAN Check
import machine, network, time, _thread, socket
# --- Relay setup ---
rel1 = machine.Pin(32, machine.Pin.OUT)
rel1.off()
# --- PHY Power / Reset ---
pwr = machine.Pin(12, machine.Pin.OUT)
pwr.off()
time.sleep_ms(200)
pwr.on()
time.sleep_ms(200)
# --- LAN Initialization ---
lan = network.LAN(
    mdc=machine.Pin(23),
    mdio=machine.Pin(18),
    power=pwr,
    phy_type=network.PHY_LAN8720,
    phy_addr=0
)
lan.active(True)
# --- Wait for Ethernet link ---
print("Wait for Ethernet-Link...")
timeout = 20
link_ok = False
for _ in range(timeout):
    rel1.on()
    time.sleep(0.5)
    if lan.isconnected():
        rel1.off()
        link_ok = True
        break
    print(".", end="")
    time.sleep(0.5)
if link_ok:
    print("\nEthernet connected!")
    ip = lan.ifconfig()[0]
    print("IP-Configuration:", ip)
    rel1.off()  # steady off = LAN OK
```

```
else:
    print("\nNo Link! Check cable / switch / power-pin.")
    for _ in range(3):
        rel1.on()
        time.sleep(0.3)
        rel1.off()
        time.sleep(0.3)
print("Boot complete. LAN status:", "OK" if link_ok else "FAIL")
```

Listing 4.9: Programmbeispiel ETH_boot.py

```
Wait for Ethernet-Link...

Ethernet connected!
IP-Configuration: 192.168.1.220
Boot complete. LAN status: OK
```

Abbildung 4.19: Ausgaben Programmbeispiel ETH_boot.py

Im Repository ist noch das Programmbeispiel *ETH_EVB_Relay_Control.py* enthalten, das die Steuerung der Relais auf dem ESP32-EVB-EA über das Netzwerk zulässt.

4.2.3 Ethernet-WiFi-Kombination

Die Verbindung über Ethernet und WiFi ist im Programm *ESP32_autonet.py* kombiniert. AutoNet erkennt automatisch

- Ethernet (LAN),
- WiFi STA,
- AP Config Mode (Fallback)

und läuft auf allen ESP32.

Das Programm finden Sie im Repository. Hier möchte ich nur die Anwendung aufzeigen.

Kopieren Sie das Programm *ESP32_autonet.py* in das MicroPython-Device. Ersetzen Sie die Angaben für SSID und Passwort der Einfachheit halber im folgenden Quelltext, und kopieren Sie das Programm als *main.py* ebenfalls in das MicroPython-Device. Das Programm *boot.py* kann ohne Inhalt bleiben.

```
import autonet
net = autonet.connect(
    ssid="Your SSID",
    password="Your Password"
)
```

Listing 4.10: Verbinde autonet in main.py

4.2.4 ESP-NOW

ESP-NOW ist ein vom chinesischen Hersteller Espressif entwickeltes, drahtloses Kommunikationsprotokoll, das speziell für ESP8266- und ESP32-Mikrocontroller optimiert wurde. Es ermöglicht eine direkte Peer-to-Peer-Kommunikation zwischen Geräten – ganz ohne WLAN-Access-Point, Router oder Internetverbindung. Dadurch ist ESP-NOW besonders schnell, energieeffizient und zuverlässig.

Im Gegensatz zu klassischem WLAN werden bei ESP-NOW kleine Datenpakete direkt zwischen den Knoten ausgetauscht. Die Latenz ist sehr gering, der Overhead minimal, und mehrere Geräte können gleichzeitig miteinander kommunizieren (*one to one*, *one to many* oder *many to many*). Optional unterstützt ESP-NOW auch eine AES-Verschlüsselung, was es für sichere IoT-Anwendungen attraktiv macht.

Typische Einsatzbereiche sind Sensornetzwerke, Fernbedienungen, Smart-Home-Komponenten, mesh-ähnliche Strukturen oder batteriebetriebene Geräte, bei denen Stromverbrauch und Reaktionszeit entscheidend sind.

Auf dem ESP32 stellt MicroPython für ESP-NOW auch AES-Verschlüsselung zur Verfügung, die Teil des ESP-NOW-Protokolls ist. AES-128 wird automatisch vom WiFi-Stack genutzt.

In MicroPython wird AES-128 aktiviert, indem Peers mit einem PMK/LMK (Primary/Local Master Key) hinzugefügt werden. Die eigentliche Ver- und Entschlüsselung passiert transparent im Hintergrund. Sie können also verschlüsselt senden und empfangen, ohne selbst AES implementieren zu müssen. ESP-NOW ist ein schlanker MAC-Frame mit Espressif-Payload. Details MicroPython für ESP-NOW finden Sie unter *https://docs.micropython.org/en/latest/library/espnow.html*.

Damit eine zielgerichtete Kommunikation aufgebaut werden kann, muss der Sender die Adresse des jeweiligen Empfängers kennen. Verwendet wird hier die MAC-Adresse (Media Access Control Address) des betreffenden Controllers – eine eindeutige Hardware-Kennung, die jedes Gerät in einem Netzwerk identifiziert.

Um die MAC-Adressen der beteiligten ESP8266 oder ESP32 zu ermitteln, bietet sich das Programm *ESP32_MAC_Addr.py* an (Listing 4.11).

```
import network
sta = network.WLAN(network.STA_IF)
sta.active(True)
peer = sta.config('mac')
print('\nMAC Address:')
print(peer)
print(':'.join(f'{b:02X}' for b in peer))
```

Listing 4.11: Programmbeispiel ESP32_MAC_Addr.py

Der spätere Netzwerkknoten wird in den STA-Mode versetzt und die MAC-Adresse abgefragt. Es erfolgt eine Ausgabe als Bytearray und in der üblichen, durch : separierten

Form. Es empfiehlt sich, den Netzwerkknoten mit einem Label, der diese Adresse ausweist, zu versehen.

Ein einfaches Beispiel soll die Initialisierung und den Datenaustausch verdeutlichen. Zuerst ist die MAC-Adresse der Empfängerseite abzufragen, da sie vom Sender benötigt wird (siehe Abbildung 4.20).

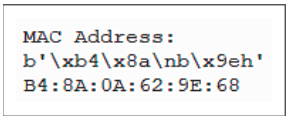

Abbildung 4.20: MAC-Adresse des Empfängers

Danach können Sie das Programm *ESP32_ESPNow_Receive.py* auf dem Empfänger installieren. Hierfür habe ich die Arduino MicroPython IDE verwendet. Das in Listing 4.12 gezeigte Programm bietet wenig Neues. Ich habe die wichtigen Stellen fett markiert. Die Initialisierung von ESP-NOW beschränkt sich auf zwei Zeilen Quelltext, und die empfangenen Daten enthalten die MAC-Adresse des Senders sowie die Message an sich (*Payload*).

```
import network, espnow, time
# WLAN im STA-Modus starten
sta = network.WLAN(network.STA_IF)
sta.active(True)
# ESP-NOW initialisieren
e = espnow.ESPNow()
e.active(True)
print("ESP-NOW Receiver started...")
while True:
    host, msg = e.recv()
    if msg:
        print("From MAC:",
              ':'.join(f'{b:02X}' for b in host),
              "\tMessage:",
              msg.decode())
    time.sleep(0.1)
```

Listing 4.12: Programmbeispiel ESP32_ESPNow_Receive.py

Anschließend installieren Sie auf der Senderseite das Programm *ESP32_ESPNow_Transmit.py* (Listing 4.13). Hierfür habe ich die Thonny IDE verwendet. Der Programmaufbau selbst ist vergleichbar mit dem Empfänger, nur dass hier die Adresse des Empfängers mitgeteilt werden muss. Kennt der Sender bereits die Empfängeradresse, dann wird der resultierende `OSError` abgefangen, anderenfalls wird die neue Adresse in die Liste der

Empfänger eingetragen. Schließlich wird hier im Takt von fünf Sekunden eine Message mit einer Countervariablen versendet, deren Empfang protokolliert wird.

```
import network, espnow, time
# MAC-Adresse des Empfängers (ANPASSEN!)
peer = b'\xb4\x8a\nb\x9eh'
# WLAN im STA-Modus starten
sta = network.WLAN(network.STA_IF)
sta.active(True)
# ESP-NOW initialisieren
e = espnow.ESPNow()
e.active(True)
# Peer hinzufügen
try:
  e.add_peer(peer)
except OSError:
  print('Peer already exists.')
print("ESP-NOW Transmit started.")
counter = 0
while True:
    msg = f"Hallo {counter}"
    e.send(peer, msg)
    print("Gesendet:", msg)
    counter += 1
    time.sleep(5)
```

Listing 4.13: Programmbeispiel ESP32_ESPNow_Transmit.py

Abbildung 4.21 zeigt die Datenübertragung der beiden ESP-NOW-Knoten in den beiden IDEs. Im Shell Window der Thonny IDE sehen Sie die Ausgaben der Senderseite, während in der Arduino MicroPython IDE die Protokollierung der empfangenen Daten erfolgt. Hier erkennen Sie an der MAC-Adresse, wer Daten gesendet hat und sehen den Inhalt der Message.

Wollen Sie die ESP-NOW-Verschlüsselung (AES-128) nutzen, dann ist Folgendes zu berücksichtigen:

Es gibt zwei jeweils 16 Byte umfassende Schlüssel:

- PMK (Primary Master Key) ist ein globaler Netzwerkschlüssel.
- LMK (Local Master Key) ist ein peer-spezifischer Schlüssel.

Beide Schlüssel müssen exakt 16 Bytes lang sein. Die Verschlüsselung funktioniert nur bei Unicast – Broadcast kann nicht verschlüsselt werden.

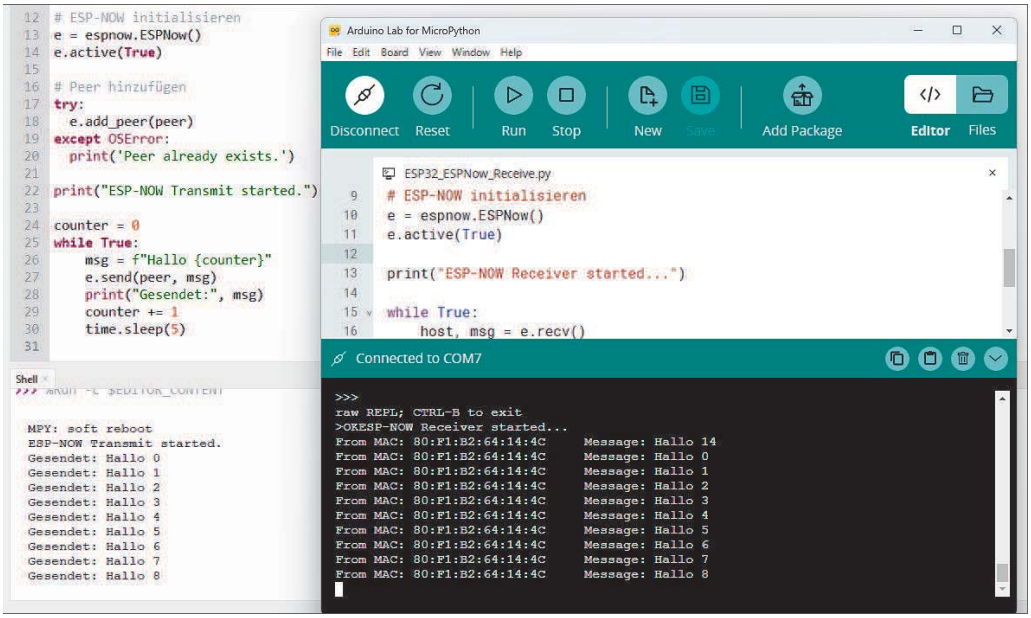

Abbildung 4.21: ESP-NOW-Datenübertragung

Die Schlüssel sollten nicht einfach vorgegeben, sondern möglichst generiert werden. Hierzu können Sie z. B. Pseudo-Zufallszahlen oder spezielle Algorithmen heranziehen. Mit dem Programmbeispiel *ESP32_ESPNow_PMK_LMK.py* steht ein kleines Tool für die Generierung der Schlüssel bereit (Listing 4.14).

```
import os, ubinascii
from hmac_sha256 import hmac_sha256
# Peer MAC-Adresse
Peer_MAC = b'\xb4\x8a\nb\x9eh'
print('\nGeneration of PMK & LMK...')
PMK = os.urandom(16)
LMK = os.urandom(16)
print('\nRandom generated:')
print("PMK:", ubinascii.hexlify(PMK))
print("LMK:", ubinascii.hexlify(LMK))
print('\nKey derivation using HMAC-SHA256:')
LMK = hmac_sha256(PMK, Peer_MAC)[:16]
print("LMK:", ubinascii.hexlify(LMK))
```

Listing 4.14: Programmbeispiel ESP32_ESPNow_PMK_LMK.py

PMK und LMK werden zuerst durch eine Pseudo-Zufallszahl gebildet. Diese beiden Schlüssel lassen sich im Flash speichern. Durch eine Schlüsselableitung (*Key Derivation Function* – KDF) kann der LMK im Peer automatisch erstellt werden. MicroPython hat kein