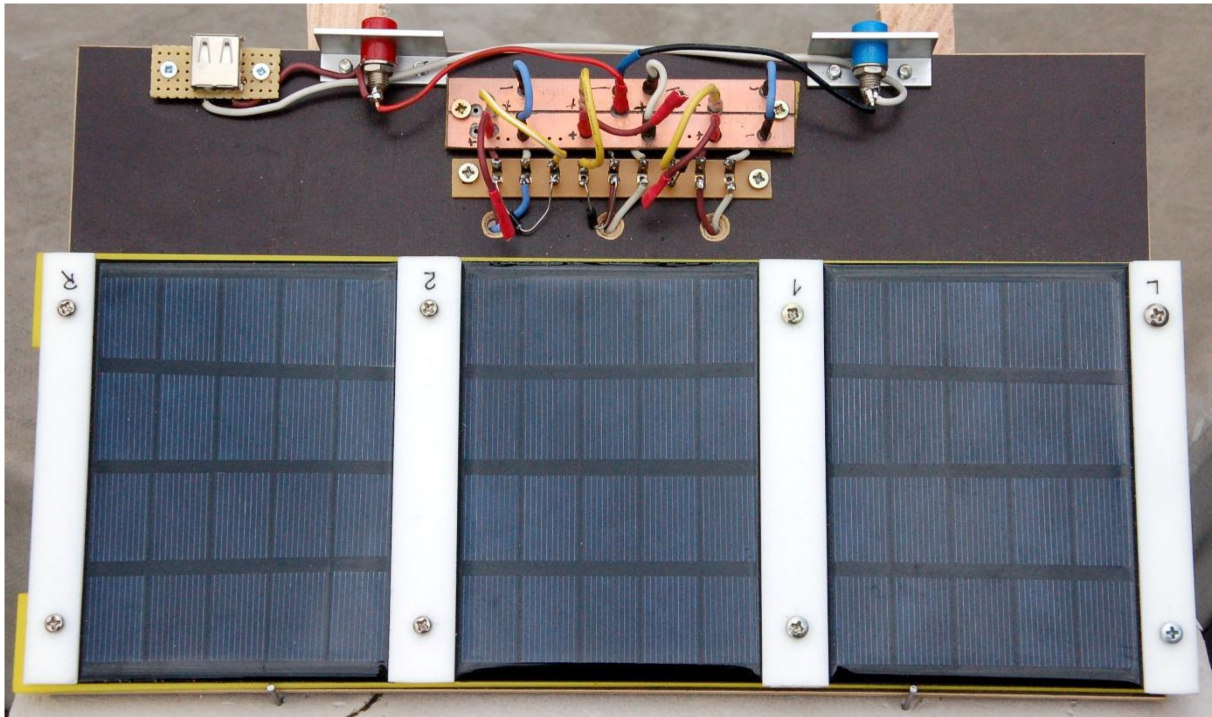


Montierte Solarpanele

Diese Blogfolge gibt es auch als PDF-Dokument.

Frage: Wie bekommt man eine große Menge an Daten von einem ESP32 zur Auswertung auf einen PC? Unser MPP-Finder aus der dritten Blogfolge liefert pro Messdurchgang 128 Messtupel zu je 4 Werten, also eine ganz schöne Menge Holz. Um die Antwort werden wir uns heute kümmern und zwar mit Hilfe von MicroPython und Python auf dem PC. Die Brücke wird die Funktauglichkeit des ESP32 sein. Natürlich muss der PC (Desktop, Notebook, etc.) ebenfalls über ein Funkinterface oder einen LAN-Anschluss verfügen. Außerdem bedarf es auch noch eines WLAN-Routers. Sie Sind gespannt, wie das gehen soll, dann willkommen bei einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Der MPP-Daten-Export vom ESP32 zum PC

Es geht darum, herauszufinden bei welcher Panelspannung der MPP erreicht ist. Die Panelspannung wiederum hängt bei gegebener Bestrahlungsstärke und Modultemperatur im Wesentlichen von der Belastung des Moduls ab. Die gilt es zu verändern, oder besser vom ESP32 verändern zu lassen. Der Buck-Konverter aus der ersten Folge der Solarreihe wird zusammen mit einem Lastwiderstand als variable Belastung und ein INA219 als Messknecht dienen, dessen Funktion ich in der zweiten Folge beschrieben habe. Die Erfassung der Messwerte ist Gegenstand in der dritten Folge. Und so sieht das Ergebnis aus, wenn man die Messreihe mit

Hilfe eines Kalkulationsprogramms grafisch darstellt. Ich habe dafür Libre-Office verwendet.

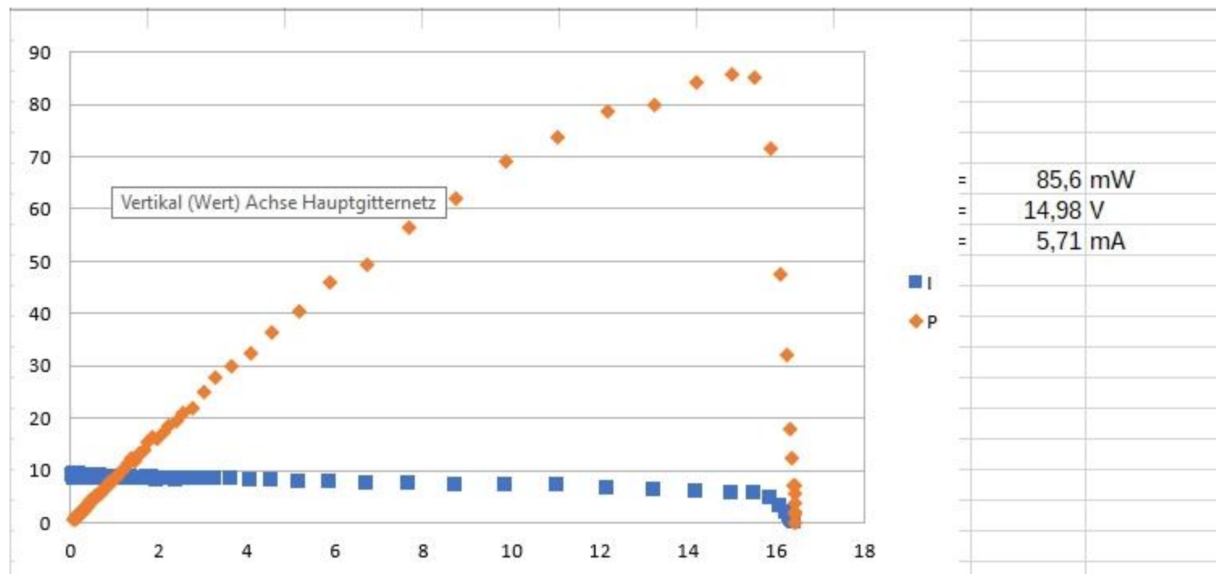


Abbildung 1: Auswertung einer Messreihe mit Libre Office

Hardware

Die Hardwareliste aus der ersten Folge wurde durch die Solarpanele und einen Widerstand von 4,7Ω /2W ergänzt und ist identisch mit der Aufstellung aus der dritten Folge. Am Aufbau ändert sich also auch nichts.

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1 - 3	Solarpanel 5V 1,5W Wasserdichtes Polysilizium Mini Solar Modul
1	INA219
1	N-Kanal MOSFET IRLZ24 (Logic Level Gate, R on = 60mΩ)
1	Transistor BC337
2	Widerstand 270 Ω
2	Widerstand 150 Ω
1	Widerstand 10 kΩ
1 - 3	Schottky-Diode 1N5817
1	Elektrolytkondensator 470µF 16V
1	Elektrolytkondensator 220µF 16V
1	Speicherdrossel 330µH 1A
1	Widerstand 4,7 Ω / 2W
diverse	Jumperkabel
1	MB-102 Breadboard Steckbrett mit 830 Kontakten 3er
4	Lötleiste mit je 6 Kontakten oder Lochrasterplatine
1	Basisbrett 16cm x 24cm

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

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

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber zum OLED-Display
oled.py API für das OLED-Display
ina219.py Treiber-Modul für den INA219
powerline_s.py Programm zum Auffinden des MPP und zur Funkübertragung
powerline_a.py Dateiübertragung ohne Router

CPython für den PC

Windows x86 MSI installer Python 2.7 für XP
Download Python 3.11.0 Neueste Version (64Bit) für den PC unter W10 / W11
Windows installer (64-bit) Version 3.9.13 für den PC
udp_receiver.py Der UDP-Empfänger auf dem PC, der unter Python 3.9 läuft.
udp_receiver_27.py Der UDP-Empfänger auf dem PC, der unter Python 2.7 läuft

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 15.12.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro

fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Signale auf dem I2C-Bus

Wie eine Übertragung auf dem I2C-Bus abläuft und wie die Signalfolge ausschaut, das können Sie in meinem Beitrag mammutmatrix_2_ger.pdf nachlesen. Ich verwende dort ein interessantes kleines Tool, mit dem Sie die I2C-Bus-Signale auf Ihren PC holen und analysieren können.

Über die Funktionsweise von Solarzellen und deren Verschaltung finden Sie einige Informationen im Teil 3. Wichtig sind die Schutzdioden, deren Anwendung dort die Abbildungen 3 und 4 zeigen.

Der MPP-Finder

Die Aufgaben des ESP32 sind:

- Ein Stellglied für die Änderung der Belastung
- Eine Messwerterfassung
- Die Übermittlung der Daten an den PC

Die erfassten Daten werden in einer Datei auf dem ESP32 gespeichert. Auf den PC werden die Dateien per Funk mit dem UDP-Protokoll übertragen und dort in einer Datei gespeichert, die schließlich mit einer Tabellenkalkulation ausgewertet werden kann. Wegen der Funkübertragung muss das Programm powerline.py an bestimmten Stellen modifiziert und ausgebaut werden. Zum Überblick, hier noch einmal die Schaltung.

Die Schaltung

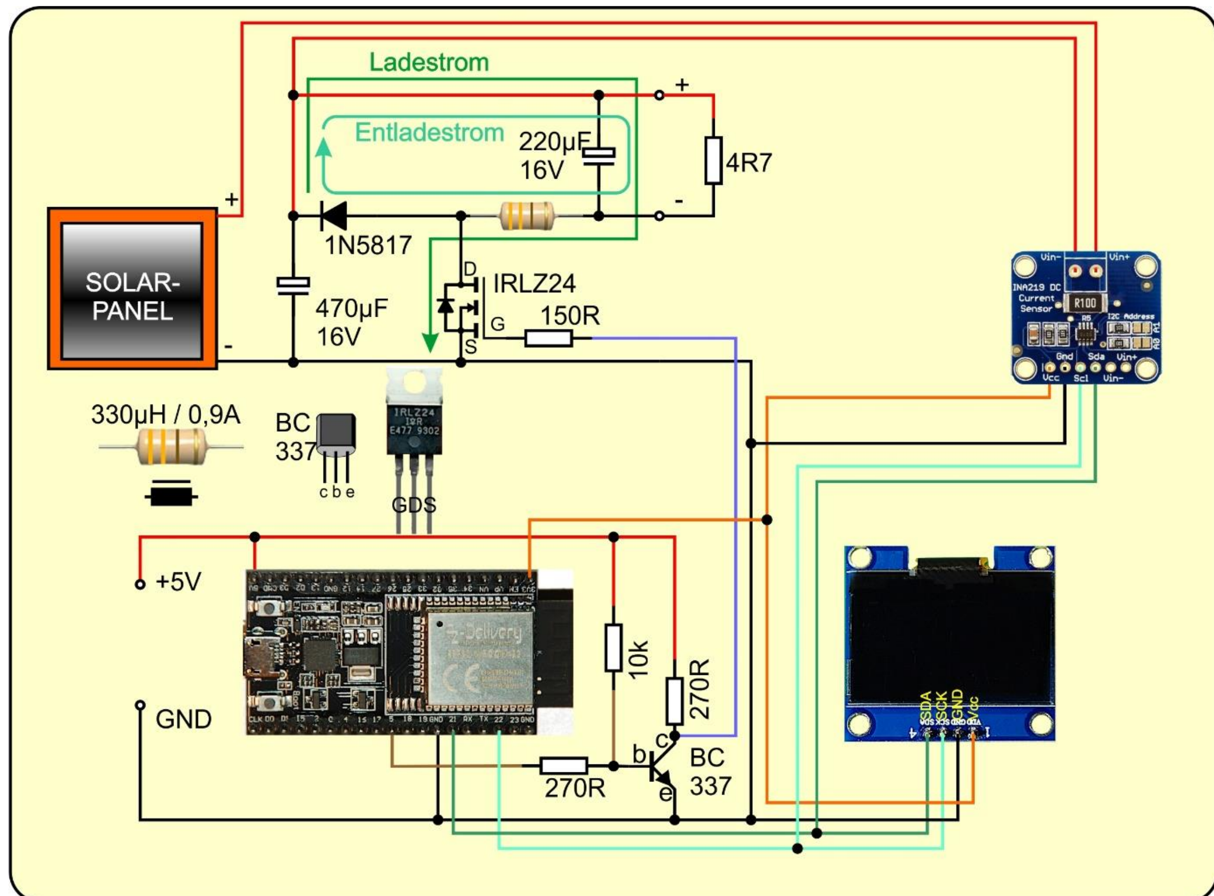


Abbildung 2: MPP-Finder - Schaltung

Die Einzelheiten und Funktionsweise der Schaltung des Wandlers habe ich in der ersten Folge genau dargelegt. Die verwendeten Solarpanele sollen laut Beschreibung bei Spannungen bis 6,3V eine Leistung bis zu 1,5W liefern. Das entspricht einer Stromstärke bis 0,3A. Bei Parallelschaltung von drei Panelen wäre also ca. 1A zu erwarten. Damit dieser Strom bei voll geöffnetem Ventil auch fließen kann, muss der Lastwiderstand einen Wert unter 5 Ohm haben.

Die OLED-Anzeige informiert über anstehende Aktionen und liefert abschließend die U-P-Kennlinie für eine erste Bewertung der Messung.

Das Programm powerline_s.py

Eingangs wird die Werkzeugkiste befüllt, ich importiere diverse Module. Wir brauchen den **I2C**-Bus zur Konversation mit dem INA219 und dem OLED-Display, **PWM** für die

Ansteuerung des Wandlers und natürlich das Modul **ina219**. Zur Steuerung des ESP32 benötige ich das Modul **buttons**, außerdem für die Dateiverwaltung **listdir** und **remove** vom Modul **os**. Die Funktion **exit**() erlaubt mir das Abbrechen des Programms mit der Flash-Taste des ESP32. Dazu gekommen sind die Module **network** und **socket**. Ich gehe im Folgenden davon aus, dass der PC sich in einem LAN mit der Netzwerkadresse 10.0.2.0 befindet und dass der WLAN-Router die Adresse 10.0.2.200 hat. Ferner wird vorausgesetzt, dass im LAN ein DHCP-Server aktiv ist, der dem ESP32 bei der Anmeldung eine IP-Adresse zuweisen kann. In der Regel läuft ein DHCP-Server auf dem WLAN-Router.

```
from machine import SoftI2C,Pin,PWM
from time import sleep
from oled import OLED
from ina219 import INA219
import buttons
from os import listdir, remove
from sys import exit
import network
import socket
```

Der ESP32 soll als UDP-Client arbeiten. Dafür benötigt er den Zugang zum WLAN-Router, also SSID und Passwort. Außerdem weisen wir ihm eine IP-Adresse nebst Portnummer zu und geben Gateway und DNS-Server im LAN bekannt. In **remote** steht die IP des Empfängers.

```
mySSIDSta="here goes yor SSID"
myPassSta="Here goes your password"
myIP="10.0.2.240"
myGW="10.0.2.200"
myDNS=myGW
myPort=9009
remote=("10.0.2.241",9091)
```

Als Portnummern sollten nur Werte außerhalb der "well known ports" von 0 bis 1023 gewählt werden, also Werte von 1024 bis 65535.

Dann beginnen die Vorbereitungen. Ich erzeuge eine I2C-Bus-Instanz, mit der ich auch gleich das Display-Objekt instanziiere. Die Anzeige löschen und vollen Kontrast einstellen. Wir arbeiten ja bei heller Umgebung.

```
i2c=SoftI2C(Pin(22),Pin(21))

d=OLED(i2c)
d.clearAll()
d.setContrast(255)
```

Der Messwiderstand auf dem INA219-BOB (Break Out Board) hat 100mΩ, wir erwarten Stromstärken bis zu 800mA. Mit 128 Einzelmessungen entspricht jede davon

einem Pixel im OLED-Display. Die Schrittweite für die PWM-Werte von 0 bis 1024 ergibt sich durch Division 1024 / Anzahl Schritte.

```
shunt = 0.1  # Ohm
imax=0.8 # Ampere
steps=128 # Anzahl Schritte
step=1024/steps
```

Der Konstruktor der Klasse INA219 fordert eine ganze Reihe von Argumenten, die bis auf **i2c** alle optionale Schlüsselwort-Parameter sind. **ModeBoth** fordert die Erfassung von Bus- und Shunt-Spannung an, **Samp4** steht für eine Auflösung von 12 Bit mit 4-fachem Oversampling, **PGA2** stellt den Abschwächer auf 1/2 und **Ubus16V** lässt Busspannungen bis 16V zu.

```
ina=INA219(i2c,mode=INA219.ModeBoth,
           busres=INA219.Samp4,
           shuntres=INA219.Samp4,
           shuntpga=INA219.PGA2,
           ubus= INA219.Ubus16V,
           Imax=imax,
           Rshunt=shunt)
```

t ist ein Buttons-Objekt an GPIO0. Die Flash-Taste des ESP32 schließt gegen GND, deshalb wird **invert** auf True gesetzt. Das macht bei gedrückter Taste aus der 0 an GPIO0 ein **True** für die Rückgabe der Methode **waitForTouch**(). Die Taste erhält den Namen **Start,** und der interne Pullup-Widerstand wird eingeschaltet. Das PWM-Signal für das Stellglied liefert der Pin GPIO5 mit einer Frequenz von 50kHz. Bei einem Duty-Cycle von 1023 liegt an GPIO5 ein Pegel von 5V, somit schaltet der BC337 durch und zieht das Gate des IRLZ24 auf GND-Potenzial, wodurch der MOSFET sperrt (siehe Abbildung 2) und die Last vom Panel trennt. Am Panel liegt jetzt die Leerlaufspannung.

```
t=buttons.Buttons(0,invert=True,pull=True,name="Start)

pwmPin=5
gate=PWM(Pin(pwmPin),freq=50000,duty=1023)
```

Der Verbindungsstatus zum Router wird als Nummer zurückgegeben. Damit man nicht extra nachschlagen muss, stellt **connectstatus** ein Dictionary bereit, das die Nummer in Klartext übersetzt. Dictionaries (kurz: Dict) sind in MicroPython das, was man in Perl oder LUA als Hash kennt. Solche Strukturen weisen einem Schlüssel ein beliebiges Objekt zu. Das kann, wie hier, ein String sein, aber auch ein Zahlenwert, eine Liste oder ein beliebiges anderes Objekt. Werte können auch mehrfach vorkommen. Die Schlüssel müssen allerdings eindeutig sein. So liefert etwa connectStatus[1] den String "STAT_CONNECTING".

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
```

```
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
    }
```

Damit die Verbindung zum Router klappt, ist es nötig, dem Router die MAC-Adresse des ESP32 im Station-Modus bekanntzugeben. Dazu muss man diese aber erst einmal selber kennen. Die Funktion **hexMac**() liest eine Bytes-Folge vom ESP32 ein und formatiert sie so, dass ein Human Body sie auch problemlos entziffern kann. Bevor Sie also eine Verbindung mit dem Router aufbauen, muss die MAC dort in eine Liste eingetragen werden. Konsultieren Sie dazu bitte das Handbuch des Routers.

```
def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode
  entgegen und bildet daraus einen String fuer die Rueckgabe
  """
  macString =""
  for i in range(0,len(byteMac)):     # Fuer alle Bytewerte
    macString += hex(byteMac[i])[2:]  # ab Position 2 bis Ende
    if i <len(byteMac)-1 :# Trennzeichen bis auf letztes Byte
      macString +="-"
  return macString
```

Jetzt definiere ich einige Passagen, die mehrfach gebraucht werden als Funktionen. **ausgabe**() erhält in dem Positionsparameter **ersteZeile** bis zu 16 Zeichen, die in der ersten Zeile des OLED-Displays ausgegeben werden. Mit ***weitere** können (bis zu fünf) weitere Positionsparameter mit Strings zugefüttert werden. **weitere** stellt eine Liste mit den übergebenen Argumenten dar, die in der for-Schleife abgerufen werden und in den Folgezeilen im Display landen.

```
def ausgabe(ersteZeile,*weitere):
    d.clearAll()
    d.writeAt(ersteZeile,0,0)
    for z in range(len(weitere)):
        d.writeAt(weitere[z],0,z+1)
```

Die komplexe Funktion **ergebnis**() liest die Datei, deren Name in **name** übergeben wird und filtert aus den Daten den MPP heraus. Die Datei wird in der Messschleife mit Messwerten gefüttert.

```
def ergebnis(name):
    try:
        n=0
        pmax=0
        umpp=0
        impp=0
        pulsmpp=0
```

try fängt Fehler ab, die beim Lesen der Datei passieren können, etwa, wenn die Datei nicht existiert. Der Zähler für die Messwerte und die Positionen für den MPP werden zurückgesetzt, und die Listen für die Messwerte geleert.

Die **with**-Struktur erlaubt den Zugriff auf die zu lesende Datei **name** über das Dateihandle **f**, ohne die Datei explizit am Ende schließen zu müssen. Sie wird automatisch geschlossen, wenn der **with**-Block verlassen wird. Die for-Schleife iteriert über die Dateieinträge.

Die Zeilen kommen als ASCII-Zeichen aus der Datei. Die Methode **sendto**(), welche die Zeichenfolge übertragen soll, erwartet allerdings ein Objekt, das auf dem Bufferprotokoll aufbaut. Ich lasse also den String in eine Bytesfolge codieren. Das macht die Methode **encode**(). Den Sendvorgang verpacke ich in ein try – except – Konstrukt, damit im fall eines Fehlers, das Programm nicht abbricht. Fehler werden durch pass einfach ignoriert.

Dann wird die Zeile von Wagenrücklauf (\r) und Zeilenvorschub (\n) befreit und im Terminal ausgegeben. Ich lasse die Kommas durch Dezimalpunkte ersetzen und teile die Zeile an den Strichpunkten in Einzelwerte auf, die ich in numerische Werte umwandeln lasse, damit ich den arithmetischen Vergleich darauf anwenden kann. Ist der aktuelle Wert der Leistung P[n] größer als das bisher gefundene Maximum, dann merke ich mir diesen und mit ihm auch die Werte für die Busspannung und die Stromstärke sowie den PWM-Wert. Abschließend wird die Anzahl der gelesenen Werte erhöht.

```
        with open(name,"r") as f:
           for z in f:
               zs=z.encode()
               try:
                   s.sendto(zs,remote)
               except:
                   pass
               z=z.strip("\n\r")
               print(z)
               z=z.replace(",",".")
               u,i,p,puls=z.split(";")
               U.append(float(u))
               I.append(float(i))
               P.append(float(p))
               Puls.append(int(puls))
               if P[n] > pmax:
                   pmax=P[n]
                   umpp=U[n]
                   impp=I[n]
```

```
                    pulsmpp=Puls[n]
                n+=1
```

Ein weiteres **try** fängt einen Division by zero – Fehler ab, der auftreten kann, wenn **pmax** oder das Maximum der Spannungswerte aus irgendeinem Grund 0 sein sollte.

Die maximale Leistung wird in die 64 Pixel Displayhöhe eingepasst, die maximal aufgetretene Spannung (= Leerlaufspannung des Moduls) in die 128 Pixel Displayweite. Nachdem das Display gelöscht ist, platziert dort die for-Schleife die Punkte im U-P-Diagramm. Der MPP und die zugehörige Spannung werden im Display und im Terminal ausgegeben, aber nicht an den PC gesandt. Der soll nur die reine Messwertdatei erhalten. Zusätzliche Inhalte würden die Auswertung stören. Vor dem Schließen des Sockets senden wir noch den Text "ende", damit das Programm am PC weiß, wann es die Datei mit den empfangenen Datensätzen schließen kann.

```
        try:
            pfaktor=63/pmax
            ufaktor=127/max(U)
            print("ufaktor=",ufaktor)
            d.clearAll()
            for num in range(n):
                pos=63-int(P[num]*pfaktor)
                x=int(U[num]*ufaktor)
                #print(num,x,"    ",U[num],"    ",pos)
                d.setPixel(x,pos,1)
            d.writeAt("Umpp={:.2f}V".format(umpp),0,4)
            d.writeAt("Pmax={:.2f}W".format(pmax),0,5)
            print("MPP: Umpp={:.2f}V Impp={:.2f}A Pmax={:.2f}W
Duty={:.2f}% ({})".\
                  format(umpp,impp,pmax,\
                  pulsmpp/1023*100,pulsmpp))
            s.sendto("ende",remote)
            s.close()
```

Die except-Blöcke informieren über eventuell aufgetretene Fehler, ohne das Programm abzubrechen.

```
        except ZeroDivisionError:
            print ("Teilen durch 0 - Fehler")
            d.clearAll()
            d.writeAt("Teilen durch 0!",0,3)
            d.writeAt("letzte Messung",1,4)
            d.writeAt("fehlerhaft",3,5)
    except OSError:
        print("Fehler beim Lesen aus Datei")
        ausgabe("DATEI-FEHLER")
```

Die Funktion **test**() erzeugt die Daten für eine Parabel mit der man die Funktion **ergebnis**() testen kann. Mit n übergebe ich die Seriennummer für den Dateinamen, den ich danach der Funktion **ergebnis**() übergebe. Damit das von der

Kommandozeile aus funktioniert, muss das Programm **powerline**.py wenigstens einmal gelaufen sein.

```
def test(n):
    f=open("power"+str(n)+".txt","w")
    for x in range(steps):
        y=-(0.01587*(x-64)**2)+63.0
        s="{:.2f};7.24;{:.2f};512\n".format(x,y)
        f.write(s)
    f.close()
```

Mit der Funktion **findName**() ermittle ich den nächsten noch nicht verwendeten Dateinamen. Die Namen bestehen aus dem String "power" und einer angehängten Seriennummer sowie der Erweiterung ".txt". Die lokale Variable **free** wird auf False gesetzt, sie zeigt an, ob ein Name verfügbar ist.

Solange **free** nicht **True** ist, wird in der while-Schleife mit dem Zähler **n**, beginnend bei 0, ein Name zusammengesetzt und geprüft, ob sich die Datei zum Lesen öffnen lässt. Das ist der Fall, wenn die Datei mit diesem Namen existiert. Andernfalls ist der Name verfügbar, und es wird eine **OSError-Exception** geworfen.

Except fängt den Fehler ab. **free** wird auf **True** gesetzt, der Name im Terminal ausgegeben und n sowie der Name zurückgegeben.

```
def findName():
    n=0
    free=False
    while not free:
        try:
            name="power"+str(n)+".txt"
            with open(name,"r") as f:
                pass
            n+=1
        except OSError:
            free=True
            print(name)
            return n,name
```

Zu den Vorbereitungen einer Messung, zu denen ich jetzt komme, gehört die Verbindungsaufnahme mit dem Router.

Ich habe festgestellt, dass hin und wieder das Accesspoint-Interface de ESP32 den Regelbetrieb stört, wenn der Controller im Station-Modus laufen soll. Deshalb erzeuge ich ein AP-Interface, um es auch sogleich wieder wegzuputzen. Dann erzeuge ich das Station-Interface-Objekt und aktiviere es.

```
# ****************** Station einrichten *******************
# Netzwerk-Interface-Instanz erzeugen
# und ESP32-Stationmodus aktivieren;
# moeglich sind entweder network.STA_IF oder network.AP_IF
```

```
# beide gleichzeitig, wie in LUA oder AT-based
# oder Adruino-IDE ist in MicroPython nicht moeglich

nic = network.WLAN(network.AP_IF)
nic.active(False)  # AP-Interface-Objekt sicher ausschalten

nic = network.WLAN(network.STA_IF)  # Constructoraufruf
erzeugt Station-Objekt nic
nic.active(True)  # Objekt nic einschalten
```

Mit **nic.config('mac')** rufe ich die MAC-Adresse des St-Interfaces ab, wandle diese in eine Klartextzeichenfolge um und lasse diese im Terminalfenster ausgeben.

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

Im Normalfall besteht noch keine Verbindung zum Router, weshalb der Körper der if-Struktur durchlaufen wird. Ich versuche die Verbindung aufzubauen, indem ich der Methode connect die SSID und das Passwort des Routers übergebe. Dann frage ich den Verbindungsstatus erneut ab und lande in der while-Schleife, welche auf die Erteilung einer IP-Adresse vom Router oder einem speziellen DHCP-Server wartet. So lange das nicht geschehen ist, wird im Terminal im Sekundenabstand ein Punkt ausgegeben.

```
if not nic.isconnected():
  # Zum AP im lokalen Netz verbinden und Status anzeigen
  nic.connect(mySSIDSta, myPassSta)
  # warten bis die Verbindung zum Accesspoint steht
  print("connection status: ", nic.isconnected())
  while nic.status() != network.STAT_GOT_IP:
    print(".",end='')
    sleep(1)
```

Nach drei bis 5 Sekunden sollte die Verbindung aufgebaut sein. Klappt das nicht, dann liegt es vielleicht daran, dass die MAC-Adresse nicht korrekt oder noch gar nicht beim Router eingetragen wurde.

Jetzt lasse ich mir den Status im Klartext ausgeben und weise danach meine oben angegebenen IP-Daten zu. Im Clientbetrieb ist das zwar nicht zwingend erforderlich, wenn die Informationen vom DHCP-Dienst kommen, aber es ermöglicht uns eine eindeutige Filterung zugelassener Stationen beim UDP-Server auf dem PC.

ifconfig() liest die Konfigurationsdaten zur Kontrolle aus. Terminalfenster und OLED-Display zeigen die Werte an.

```
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
STAconf = nic.ifconfig((myIP,"255.255.255.0",myGW,myDNS))
STAconf = nic.ifconfig()
```

```
print("STA-IP:\t\t",STAconf[0],"\nSTA-
NETMASK:\t",STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
d.clearAll()
d.writeAt(STAconf[0],0,2)
d.writeAt(STAconf[1],0,3)
d.writeAt(STAconf[2],0,4)
sleep(3)
```

Mit ein paar weiteren Zeilen wir der UDP-Client eingerichtet. Das UDP-Protokoll ist im Vergleich zu TCP schneller und einfacher zu handhaben, es lässt einen bidirektionalen Datenaustausch zu, wie eine RS232-Leitung, sichert aber den Datenverkehr nicht ab. Es gibt also kein Handschake wie bei TCP. Es wird zum Beispiel nicht geprüft, ob die Daten richtig oder überhaupt angekommen sind. Entscheidend ist in der ersten Zeile das Argument socket.SOCK_DGRAM, welches dem Konstruktor der socket-Instanz s übergeben wird, es erzeugt einen UDP-Socket.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
print("sending on port {}...".format(myPort))
s.settimeout(0.1)
d.writeAt("sending on {}".format(myPort),1,5)
sleep(3)
```

Die zweite Zeile sorgt dafür, dass nach einem Neustart dieselbe IP-Adresse erneut vergeben werden kann. Dann binde ich die Portnummer an die oben zugeteilte IP-Adresse. Mit einem Timeout von 0,1 Sekunden verhindere ich, dass das Programm bei der Übertragung aufgrund eines nicht vorhersehbaren Fehlers hängen bleibt. Tritt ein Fehler auf, dann fange ich ihn mit except ab. Näheres dazu später.

Die Taste t dient zur Steuerung des Hauptprogramms. Zu Beginn kann damit eine neue Messung ausgelöst werden. Ich lasse die Information im Display ausgeben, dann bleiben 3 Sekunden, um die Taste zu drücken. Der Puffer von einer Sekunde stellt sicher, dass die Taste losgelassen wurde. Den Tastenzustand merke ich mir in **messen**. True steht für "Taste gedrückt", ist der Timeout abgelaufen, kommt **None** zurück.

```
ausgabe("FLASH-TASTE","FUER MESSUNG","DRUECKEN")
messung=t.waitForTouch(t,delay=3)
t.waitForRelease(t,delay=1)
```

Nur wenn gemessen werden soll, wird nachgefragt, ob die bestehenden Messwertdateien alle gelöscht werden sollen. Ist das der Fall (erneuter Tastendruck), dann lasse ich mir eine Liste der vorhandenen Dateien erstellen. Sind die ersten 5 Zeichen im Dateinamen "power", wird der Dateiname ausgegeben und die Datei gelöscht. Dieser Filter bewahrt die Datei main.py, die das Programm enthält sowie die Moduldateien davor, auch gelöscht zu werden. Danach fahnde ich nach einem verfügbaren Dateinamen und bekomme Nummer und Dateiname zurück. Damit kann eine von zwei Aktionen durchgeführt werden.

```
if messung:
    ausgabe("FLASH-TASTE"," ","ZUM LOESCHEN","DER DATEIEN")
    clearFiles=t.waitForTouch(t,delay=3)
    if clearFiles:
        for datei in listdir():
            if datei[:5]=="power":
                print(datei)
                remove(datei)

counter,name = findName()
```

Ist **messung None**, dann wurde die Taste t zu Beginn nicht gedrückt. Das führt zur Ausgabe der Messkurve der letzten Messung im OLED-Display sowie der Messwerte im Terminal. In **counter** steht die Nummer des nächsten freien Dateinamens. Counter – 1 spricht also die letzte verfügbare Messung an.

```
else: # Ergebnis darstellen
    name="power"+str(counter-1)+".txt"
    ergebnis(name)
```

Mit **messung** gleich **True** startet eine neue Messreihe. Try blockt Schreibfehler in die Datei ab. Ich öffne eine Datei zum Schreiben, setze den PWM-Wert auf 1023 (MOSFET sperrt) und den Wertezähler auf 0.

```
if messung:
    ausgabe("MESSUNG","GESTARTET")
    fehler=False
    try:
        with open(name,"w") as f:
            pulse=1023
            n=0
```

Solange **pulse** größer als 0 ist, werden Einzelmessungen gemacht. Die Schrittweite **step** ist eine Fließkommazahl, sie wird nach jeder Messung von **pulse** subtrahiert. **pulse** ist also auch vom Typ **float**. Für die Verwendung von pulse als PWM-Wert muss daraus eine Ganzzahl gemacht werden. Nummer der Messung und die Pulsweite werden im OLED-Display angezeigt.

```
            while pulse>=0:
                intpulse=int(pulse)
                gate.duty(intpulse)
                d.writeAt("{}. {}
".format(n,intpulse),0,3,True)
```

getCurrent() und **getPower**() können eine Overflow-Exception werfen, die ich mit try abfange. In diesem Fall wird **fehler** auf **True** gesetzt und der Leistung der Wert -1 zugewiesen.

```
                try:
                    u=ina.getBusVoltage()
                    i=ina.getCurrent()
                    p=ina.getPower()
                except OverflowError:
                    u,i,p=0,0,-1
                    fehler=True
```

Die Messwerte werden zu einem String mit Zeilenvorschub zusammengesetzt, wobei der Strichpunkt als Trennzeichen dient. Beim Einlesen eines Messwertfiles als CSV-Datei in ein Kalkulationsprogramm erkennt dieses so die Trennung der Werte und kann sie in verschiedene Spalten aufteilen. Damit das Kalkulationstool die Strings als Zahlenwerte deutet, müssen die Dezimalpunkte in Kommas umgewandelt werden, das tut die Methode **replace**(). Der String wird in die Datei geschrieben und **pulse** um die Schrittweite verringert. Mit Beendigung des with-Blocks wird die Datei automatisch geschlossen.

```
                s="{:.2f};{:.2f};{:.2f};{}\n".\
                    format(u,i,p,intpulse)
                s=s.replace(".",",")
                f.write(s)
                pulse-=step
                n+=1
```

Nach Abschluss der Messreihe setze ich den Duty-Cycle auf 1023 und trenne damit das Panel von der Last ab. Es folgen die Ausgabe vom Ergebnis und die Behandlung eventuell aufgetretener Fehler.

```
        gate.duty(1023)
        ergebnis(name)
        if fehler:
            d.writeAt("OverflowERROR",0,0)
    except OSError:
        print("Fehler beim Schreiben in Datei")
        ausgabe("DATEI-FEHLER")
```

Programmausführung im Freien

Damit das Programm in der Pampa auch ohne PC ausgeführt werden kann, muss es unter dem Namen **main**.py auf den ESP32 geladen werden. Ich erzeuge dazu immer eine **neue Datei** und kopiere den Programmtext, hier also den von **powerline_s**.py, hinein und speichere die neue Datei direkt auf dem ESP32 unter dem Namen **main**.py ab. Mit jedem Kaltstart oder Reset startet der ESP32 jetzt automatisch mit einem neuen Durchlauf.

Dateitransfer zum PC

Gut, den Sender der Daten hätten wir damit fertiggestellt. Kommen wir zum Empfänger. Mit Thonny wird auf dem PC auch eine CPython-Version installiert. Das ist keine abgemagerte Version wie MicroPython, CPython ist der viel mächtigere Vater von MicroPython. Aber, was in MicroPython geht, geht in CPython erst recht. Na ja, bis auch diejenigen Features, die auf der Infrastruktur der Controllerarchitektur basieren. In CPython gibt es zum Beispiel keine GPIOs, keine Hardwaretimer … Natürlich verfügt CPython auch über das Modul socket. Dadurch sind wir in der Lage, mit dem Vokabular von MicroPython einen UDP-Server auf dem PC zu entwerfen, der die Daten vom ESP32 empfängt und in eine Datei verpackt. Sobald die Daten aus der Datei der aktuellen Messung auf dem ESP32 ausgelesen werden, werden sie simultan auf den PC gebeamt. Dabei spielt es keine Rolle, ob der PC unter Windows oder Linux läuft.

Ich habe auf meinem Windows 10 zusätzlich das Python-Release 3.8.5 installiert. Es bringt die IDE IDLE mit. Dort gebe ich das folgende Programm ein, das ich später über die Windows-Kommandozeile oder einen entsprechenden Linkt vom Desktop aus starten kann. Natürlich kann man das Programm auch mit Thonny editieren und dann im Workspace oder einem anderen Verzeichnis abspeichern. In dem Verzeichnis, in dem die Datei zur Ausführung aufgerufen wird, werden auch die von ihr erzeugten cvs-Dateien abgelegt.

Eine Netzwerkverbindung brauche ich nicht aufzubauen, weil der PC im Normalfall bereits über eine solche verfügt. Ich importiere also nur das Modul **socket**.

```
import socket
```

Dann lege ich die Verbindungsdaten des PCs fest, die mit dem Konsolenbefehl

ipconfig -all (Windows Eingabeaufforderung)   oder

ifconfig (Linux Terminal)

abrufen kann.

```
myIP="10.0.1.208"
myGW="10.0.1.200"
myDNS="10.0.1.1"
myPort=9091
```

Das Dict connectStatus und die Funktion hexMAC() kennen Sie bereits. Sie erfüllen auf dem PC denselben Zweck wie auf dem ESP32.

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
```

```
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
    }

def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode
entgegen und
  bildet daraus einen String fuer die Rueckgabe
  """
  macString =""
  for i in range(0,len(byteMac)):      # Fuer 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
```

Auch das Instanziieren des UDP-Sockets ist nichts Neues.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
print("waiting on port {}...".format(myPort))
s.settimeout(0.1)
```

Das Programm soll jetzt nach einem Dateinamen fragen. Ich empfehle bei der Ausführung als Namensergänzung **.csv** (Character separated values) zu verwenden. Dieses Format wird von Libre Office als Tabellenformat erkannt, wenn Sie die Datei dort öffnen wollen.

```
name=input("Dateiname: ")
print("waiting on port",myPort)
```

Auch die with-Struktur kennen Sie bereits. Wir öffnen die Datei zum Schreiben.

Die while-Schleife versucht einen Datensatz aus dem Empfangspuffer zu lesen. Liegt im Moment kein Datensatz vor, dann wird eine OSError-Exception geworfen, hervorgerufen durch den oben vereinbarten Timeout. Die Ausnahme fangen wir mit except ab. Wenn nichts zur Verarbeitung vorliegt, dann warten wir eben indem wir nichts tun – **pass**.

```
with open(name,"w") as f:
    while 1:
        try:
```

```
            # Nachricht empfangen
            rec,adr=s.recvfrom(150)
            rec=rec.decode()
            print(rec,end="")
            if rec=="ende":
                print("\nProgramm beendet")
                print("Die Datei",name,"wird geschlossen.")
                s.close()
                break
            else:
                f.write(rec)
        except OSError:
            pass
```

Liegt aber eine Nachricht vor, dann wird sie in die Variable **rec** verfrachtet und die Absenderadresse nach **adr** verschoben und vergessen, es ist ja keine Antwort an den ESP32 erforderlich. Beide Variablen sind zum Entpacken des Tuples, das die Methode **recvfrom**() liefert, nötig.

Die Bytes-Folge wird in einen String dekodiert und im Terminalfenster ausgegeben. Weil der String bereits ein Linefeed (\n = Zeilenvorschub) enthält, ersetze ich das automatische Linefeed des print-Befehls durch einen Leerstring.

Besteht die Nachricht aus dem Text "**ende**", dann wurden alle Datensätze übertragen und der Job wird mit break abgebrochen, nachdem der Socket s geschlossen wurde. Damit wird auch die with-Struktur verlassen und das Dateihandle **f** somit automatisch geschlossen. Alle Zeilen, die nicht auf "ende" lauten, werden in die Datei geschrieben.

Speichern Sie den Programmtext jetzt unter dem Namen **udp_receiver**.py ab. Wenn Sie außerdem einen Link darauf auf dem Desktop anlegen, können Sie die Datei mit einem Doppelklick starten.

Abbildung 3: udp-receiver.py als Link auf dem Desktop

Ist die Hardware betriebsbereit und das Panel angeschlossen? Dann starten Sie als Erstes **udp_receiver**.py auf dem PC und vergeben einen Dateinamen mit der Ergänzung **.csv**. Booten Sie dann den ESP32 oder drücken Sie dort die RST-Taste. Der Controller sollte sich am Router anmelden, sodass nach ein paar Sekunden die Verbindungsdaten am OLED-Display angezeigt werden. Mit der Flashtaste initiieren sie nun die Messung. Ob Sie bisherige Messwertdateien löschen lassen oder nicht, liegt bei Ihnen. Nachdem die Messreihe durchgelaufen ist, wird die Messkurve am Display erzeugt und die Datensätze werden zum PC gefunkt, wo sie im

Arbeitsfenster des Empfängerprogramms erscheinen. Dieses wird beendet, nachdem die Messwert-Datei geschlossen wurde. Sie steht jetzt bereit zur Auswertung.

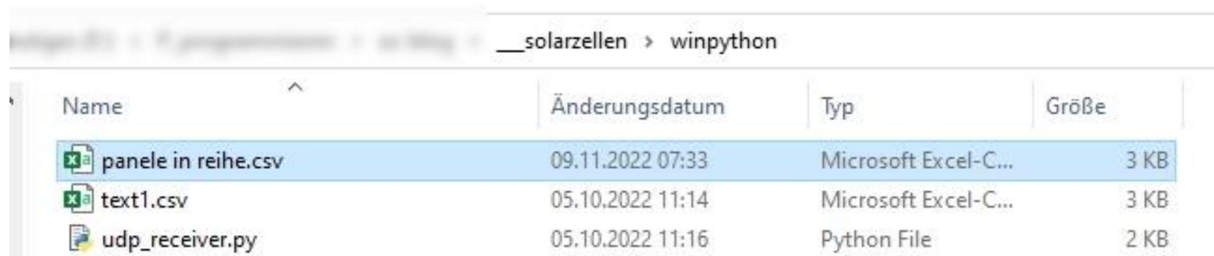


Abbildung 4: Die Datei ist angekommen

Starten Sie jetzt Libre Office (oder eine analoge Anwendung) und öffnen Sie das Datei-Menü – Öffnen – Öffnen.

Abbildung 5: Datei öffnen

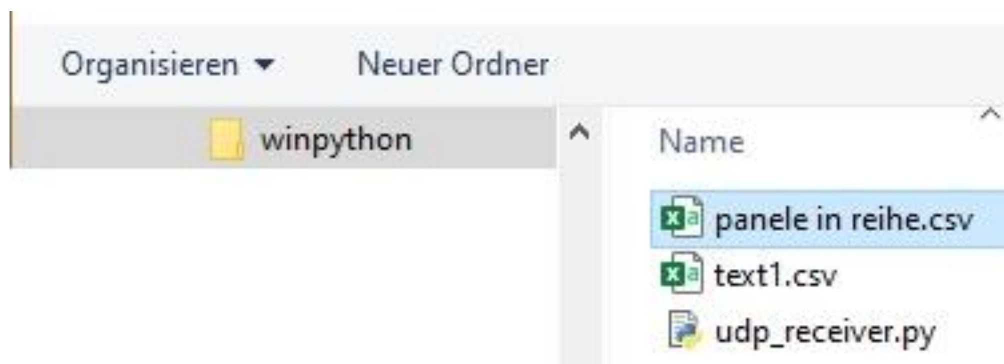


Abbildung 6: Laden der Messwert-Datei

Libre Office erkennt anhand der Namenserweiterung .csv, dass es sich um eine Textdatei mit Trennzeichen zwischen den Feldern handelt und zeigt den Dialog zum Einstellen der Trennoptionen.

Bei **Trennoptionen** setzen Sie den Haken bei **Semikolon** - **OK**

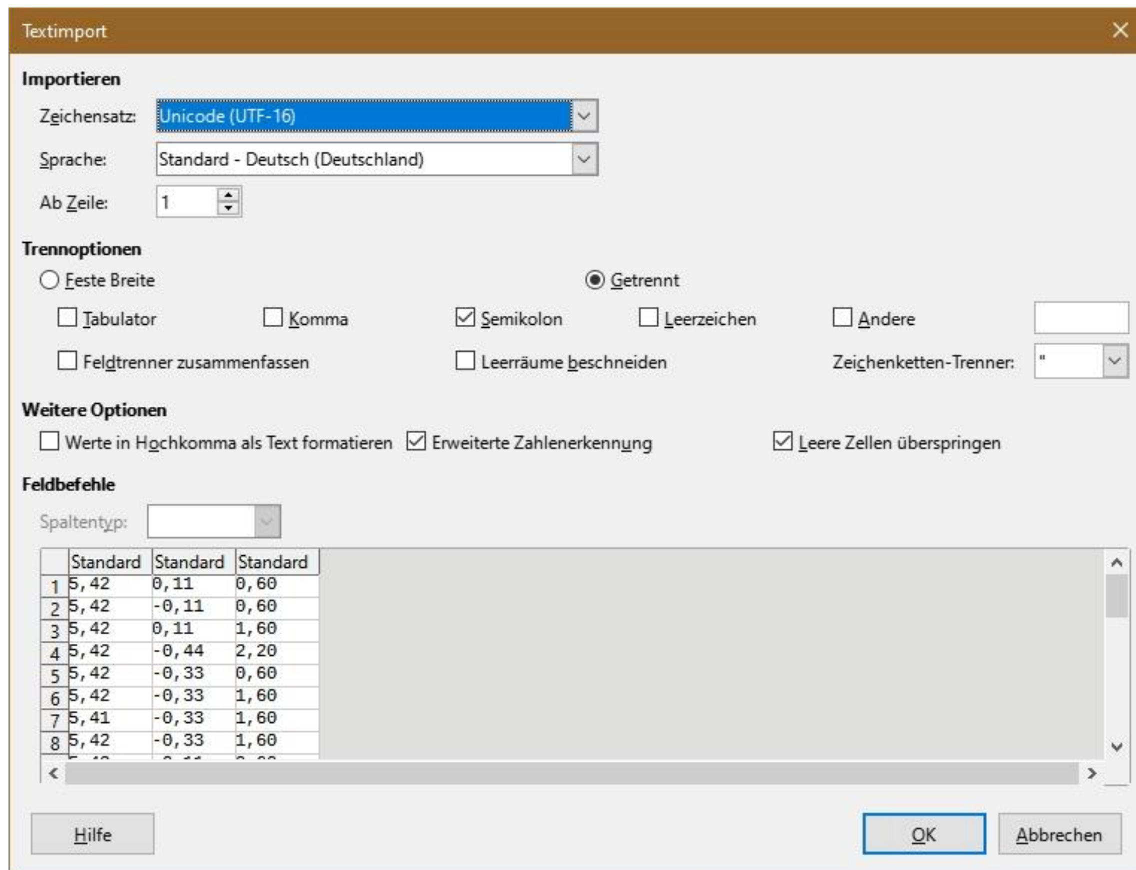


Abbildung 7: Einstellungen Textimport

Markieren Sie die Spalten A, B und C.

Liberation Sans | 10 pt | F | K

A1:C1048576 | f_x ∑ = | 0,6

	A	B	C
1	5,42	0,11	0,6
2	5,42	-0,11	0,6
3	5,42	0,11	1,6
4	5,42	-0,44	2,2
5	5,42	-0,33	0,6
6	5,42	-0,33	1,6
7	5,41	-0,33	1,6
8	5,42	-0,33	1,6
9	5,42	-0,11	0,6
10	5,41	-0,33	1,6
11	5,39	-0,11	0,6
12	5,36	-0,11	0,6
13	5,34	0,22	0,6
14	5,31	0,33	1,6
15	5,28	0,33	3

Abbildung 8: Spalten markieren

Klick auf Diagramm einfügen.

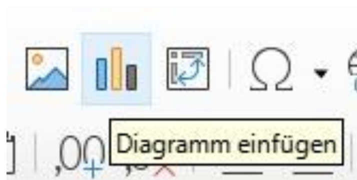


Abbildung 9: Diagramm einfügen

Wählen Sie **XY-Streudiagramm** - **Punkte** oder **Punkte und Linien**

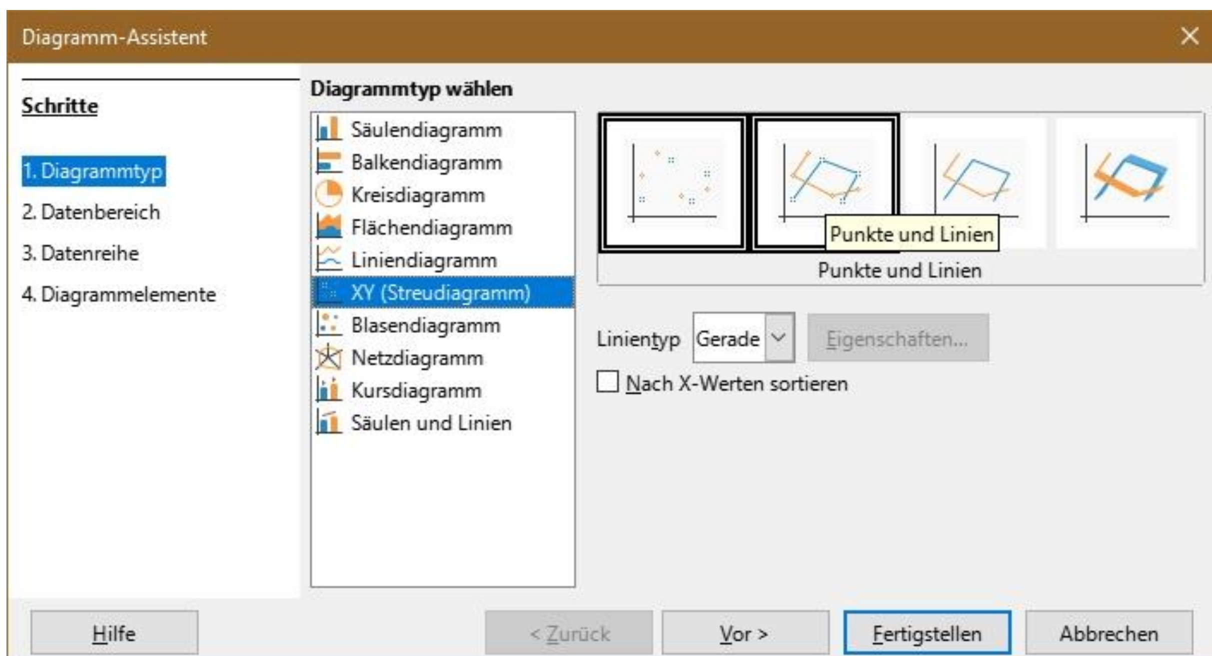
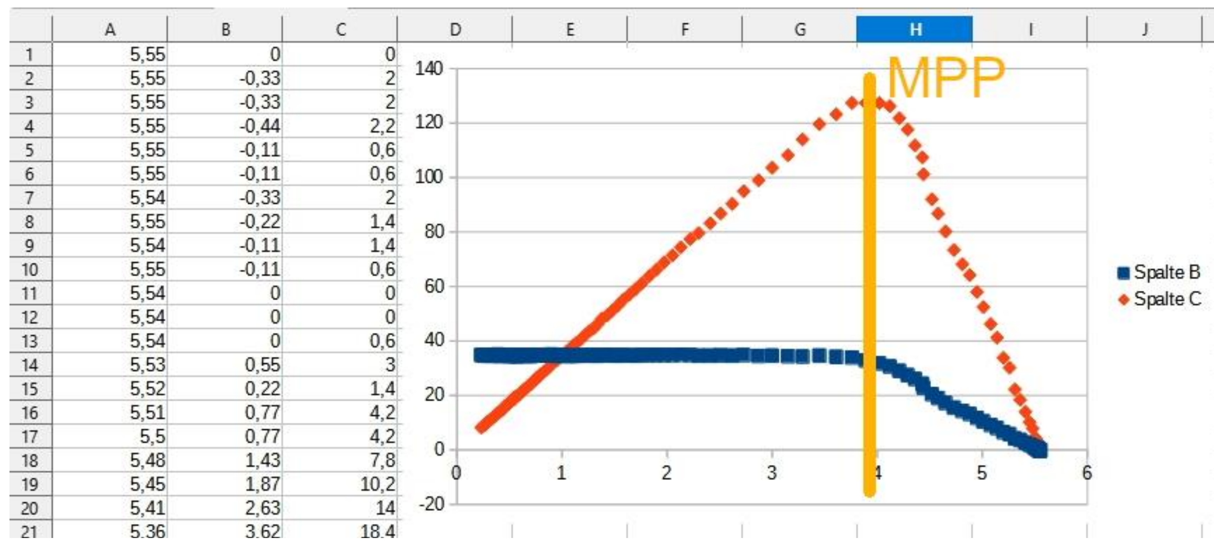


Abbildung 10: Diagrammassistent

Fertigstellen.



	A	B	C
1	5,55	0	0
2	5,55	-0,33	2
3	5,55	-0,33	2
4	5,55	-0,44	2,2
5	5,55	-0,11	0,6
6	5,55	-0,11	0,6
7	5,54	-0,33	2
8	5,55	-0,22	1,4
9	5,54	-0,11	1,4
10	5,55	-0,11	0,6
11	5,54	0	0
12	5,54	0	0
13	5,54	0	0,6
14	5,53	0,55	3
15	5,52	0,22	1,4
16	5,51	0,77	4,2
17	5,5	0,77	4,2
18	5,48	1,43	7,8
19	5,45	1,87	10,2
20	5,41	2,63	14
21	5,36	3,62	18,4

Abbildung 11: Fertiges Diagramm

Die blaue Kurve ist die U-I-Kennlinie. Dort, wo sie nach unten abknickt, liegt das Maximum der roten Kurve, der MPP. Verschiedene Messreihen haben gezeigt, dass der MPP bei etwa 70% der Leerlaufspannung am Panel liegt. Im Produktionsbetrieb wird es die Aufgabe des Stellglieds sein, den Wandler auf diesen Punkt zu justieren. Die äußeren Einflüsse müssen dabei natürlich berücksichtigt werden, deshalb wird es nötig sein, die Lage des MPP in gewissen zeitlichen Abständen zu überprüfen. Ein Gerät welches das durchführt nennt man MPP-Tracker. Vermutlich wird sich ein zukünftiger Beitrag mit diesem Thema beschäftigen.

Ein Tipp zum Schluss. In der Pampa hat man selten ein LAN zu Verfügung. Für eine Insellösung genügt auch ein Notebook mit WLAN-Karte.

Das Vorgehen erfordert allerding einige Änderungen. Das beginnt damit, dass der ESP32 jetzt als Accesspoint konfiguriert wird, was an zwei Stellen im Programm passiert. Wir vergeben eine SSID und die IP eines beliebigen Teilnetzwerks an den ESP32. Die IP des Notebooks (remote) muss aus demselben Netzwerk stammen.

```
mySSID="finder"
myPass="gibtsnicht"
myIP="10.0.2.100"
myGW=myIP
myDNS=myIP
myPort=9091
remote=("10.0.2.240",9091)
```

Auf dem Notebook stellen Sie nun eine neue WLAN-Verbindung mit dem ESP32 her. Das geht sehr schnell, weil die Verbindung offen ist, also kein Passwort-Handshake erfolgt. Eine gesicherte Verbindung ist nach derzeitigem Stand von MicroPython nicht möglich. Dennoch muss beim Erstellen des AP-Interfaces ein solches angegeben werden. Damit die Verbindung klappt, muss der ESP32 schon einmal gestartet worden sein.

Die zweite Anpassung ist die Einrichtung eines AP-Interfaces anstelle des STA-Interfaces. Alles andere bleibt unverändert.

```
# **************** Accesspoint einrichten *****************
nic = network.WLAN(network.AP_IF)
nic.active(True)  # Objekt nic einschalten
#
MAC = nic.config('mac')             # # binaere MAC-Adresse
abrufen und
myMac=hexMac(MAC)      # in eine Hexziffernfolge umwandeln
print("STATION MAC: \t"+myMac+"\n") # ausgeben
#
# AP-Verbindung bereitstellen
MAC = nic.config('mac')  # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)        # in eine Hexziffernfolge umgewandelt
print("AP MAC: \t"+myMac+"\n") # ausgeben
#
nic.ifconfig(("10.0.2.100","255.255.255.0",
              "10.0.2.100","10.0.2.100"))
print(nic.ifconfig())
nic.config(authmode=0)
print("Authentication mode:",nic.config("authmode"))
nic.config(essid=mySSID, password=myPass)
while not nic.active():
  print(".",end="")
  sleep(1)
print("NIC active:",nic.active())

APconf = nic.ifconfig()
print("STA-IP:\t\t",APconf[0],"\nSTA-
NETMASK:\t",APconf[1],"\nSTA-GATEWAY:\t",APconf[2] ,sep='')
d.clearAll()
d.writeAt(APconf[0],0,2)
d.writeAt(APconf[1],0,3)
d.writeAt(APconf[2],0,4)
sleep(3)
```

Das veränderte Programm powerline_a.py können Sie hier herunterladen.

Auf dem Notebook muss natürlich auch Python installiert sein. Unter XP kommt dafür höchstens die Version 2.7 in Frage. Damit das Programm **udp_receiver**.py dort problemlos läuft, bedarf es auch hier an zwei Stellen einer Änderung. Am besten, sie laden sich das gepatchte Programm **udp_receiver_27.py** herunter, das dann aus der IDE Idle heraus aufgerufen wird. Unter W10 muss nichts geändert werden, weil dort die neueste Version von Python (3.9) dieselben Features bereitstellt wie MicroPython.

So, jetzt können Sie weitere Messreihen durchführen, bei verschiedener Einstrahlung, für parallel geschaltete Panele oder für Serienschaltungen derselben.

Viel Freude beim Forschen!