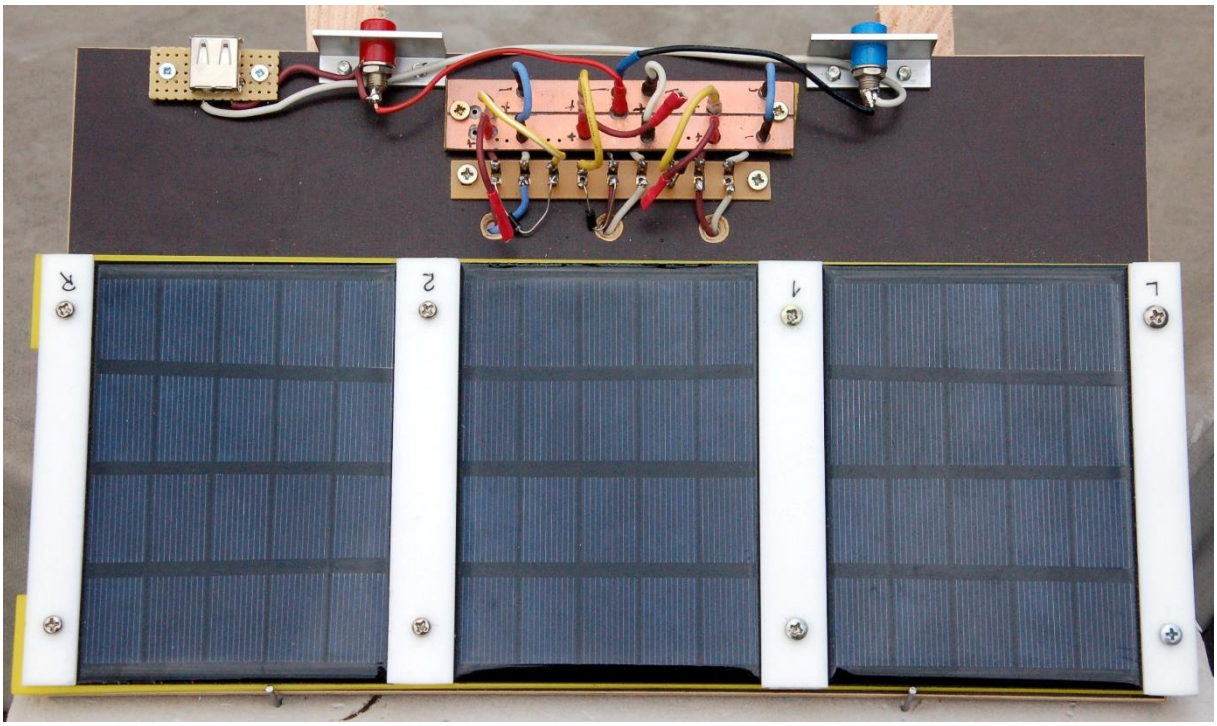


Montierte Solarpanele

Diese Blogfolge gibt es auch als PDF-Dokument.

Solarzellen liefern, wenn wir mal von den Beschaffungs- und Montagekosten absehen, kostenlos und umweltfreundlich elektrische Energie. Das ist nichts Neues. Natürlich sollte der Betrieb möglichst effektiv sein. Deshalb begeben wir uns heute auf die Suche nach dem MPP, dem Maximum-Power-Point, also dem Betriebszustand, in dem das Panel die größtmögliche Energie abgibt. Temperatur, Bestrahlungsstärke und Belastung des Moduls haben Einfluss auf die Position des MPP. Ein ESP32 und ein Buck-Konverter mit einem MOSFET-Leistungstransistor helfen uns bei der Suche. Wie das im Einzelnen läuft, das verrate ich Ihnen in der heutigen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 auf der Suche nach dem MPP

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. Es werden eine Menge an Messwerten anfallen, so viele, dass es nerven würde, alles von Hand zu erfassen und

auszuwerten. Deshalb müssen die Werte auf den PC gebeamt werden. Ein Verfahren bedient sich des USB-Anschlusses, ein anderes benutzt die Funkeigenschaft des ESP32. Und so sieht das Ergebnis aus, wenn man die Messreihe mit Hilfe eines Kalkulationsprogramms grafisch darstellt. Ich habe dafür Libre-Office verwendet.

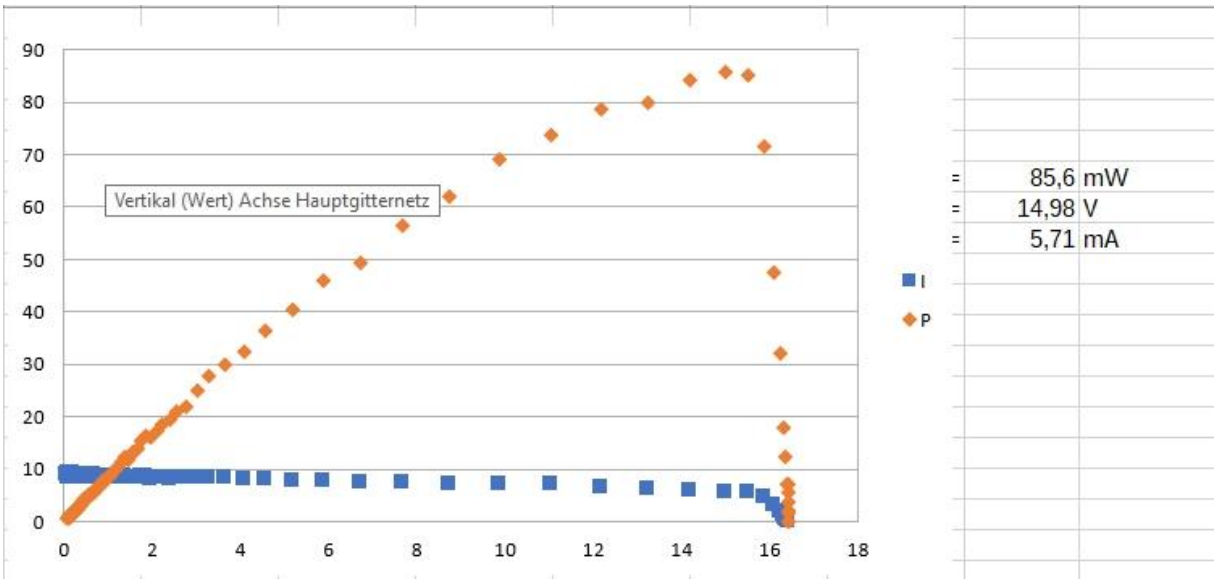


Abbildung 1: Auswertung einer Messreihe mit Libre Office

Hardware

Die Hardwareliste aus der ersten Folge wurde durch die Solarpanele und einen Widerstand von 4,7Ω /2W ergänzt.

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.py Programm zum Auffinden des MPP

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 05.02.2022) 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.

Schaltung von Solarzellen

Solarzellen sind vom Prinzip her Flächendioden. Licht kann durch die dünne Kathode bis zur Grenzschicht zwischen n-leitendem und p-leitendem Silizium vordringen und dort Elektron-Lochpaare hervorrufen. Durch die dadurch entstehende Raumladungszone werden die Elektronen in Richtung der n-leitenden Schicht beschleunigt, die Löcher zur Anode auf der Rückseite.

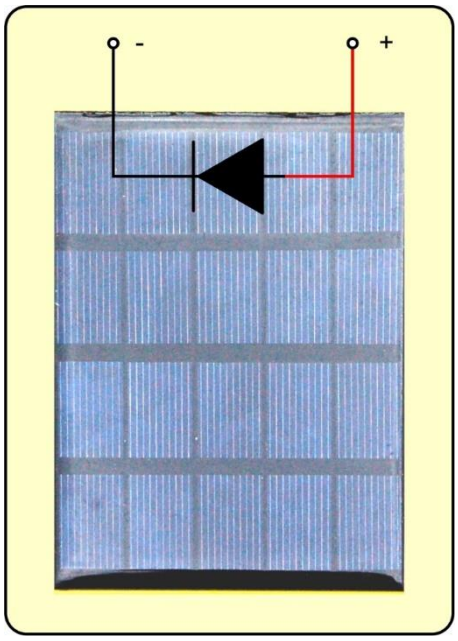


Abbildung 2: Solarzelle - Ersatzschaltbild

Aber das passiert nur, wenn die Zelle beleuchtet wird. Fällt kein Licht darauf, dann verhält sich die Solarzelle wie eine normale Diode. Was heißt das?

Nun, wenn die Spannung an der Zelle niedriger ist, wie eine extern angelegte Spannung, dann beginnt die Zelle wie eine normale Diode zu leiten. Legt man das Panel direkt an einen zu ladenden Akku, dann wird dieser tagsüber tatsächlich aufgeladen, wenn die Spannung des Panels höher ist als die Akkuspannung. In der Nacht wird der Akku aber entladen, weil dann vom Minuspol des Akkus die Elektronen über die Flächendiode, die jetzt in Durchlassrichtung geschaltet ist, zu seinem Pluspol wandern. Das muss man unterbinden. Es gelingt, indem man, wie in Abbildung 3, eine Diode in den Stromkreis einfügt, die den Strom der Elektronen zum Akku tagsüber zulässt und den Rückstrom nachts unterbindet. Dabei ist es wichtig, eine Diode zu wählen, die eine möglichst niedrige Vorwärtsspannung aufweist, damit die Verluste klein gehalten werden. Schottky-Dioden, wie die 1N5817 erfüllen diese Forderung. Die Vorwärtsspannung beträgt hier 0,45V, während sie bei normalen Silizium-Dioden bei 0,7V und mehr liegt.

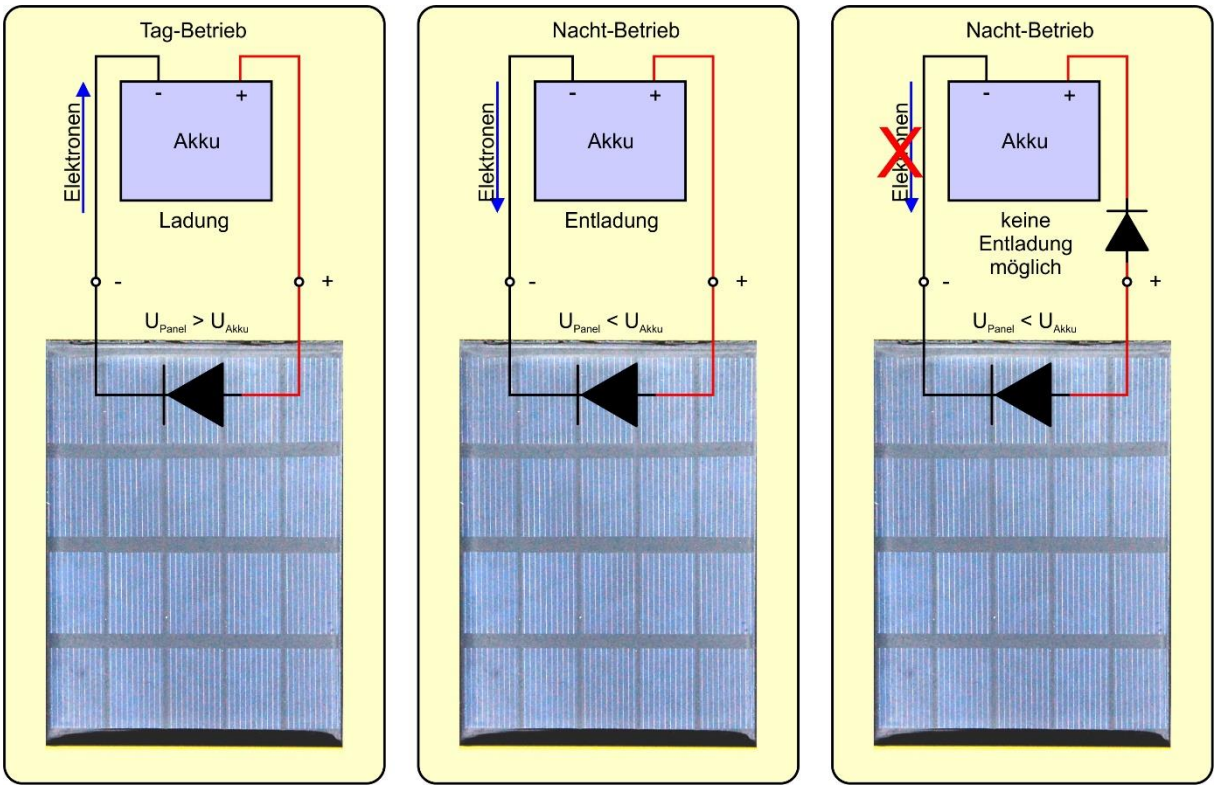


Abbildung 3: Laden eines Akkus aus einem Solarpanel

So eine Entladeschutzdiode ist auch nötig, wenn mehrere Panele parallelgeschaltet werden sollen. Man verhindert damit, dass bei Beschattung eines der Panele der Strom, den die anderen liefern, durch das beschattete Panel abgeleitet wird.

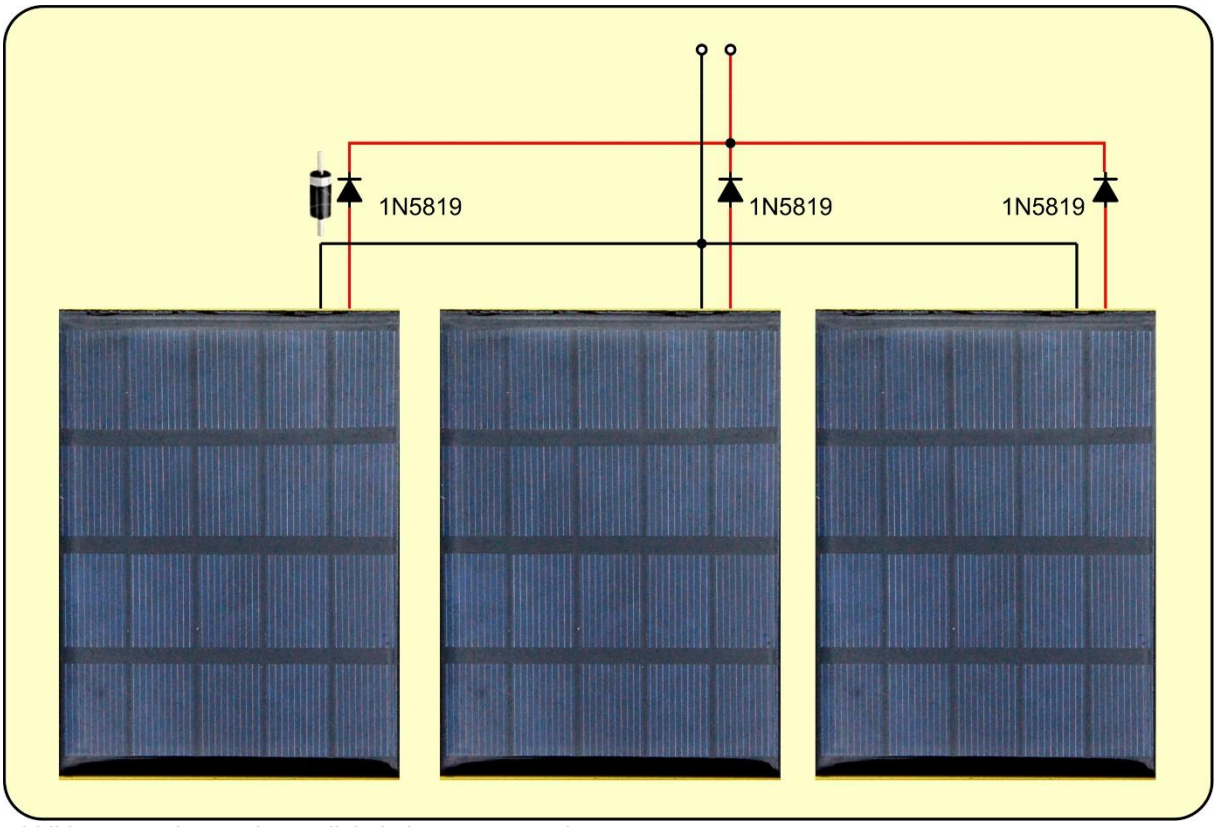


Abbildung 4: Solarpanele parallel - höhere Stromstärke

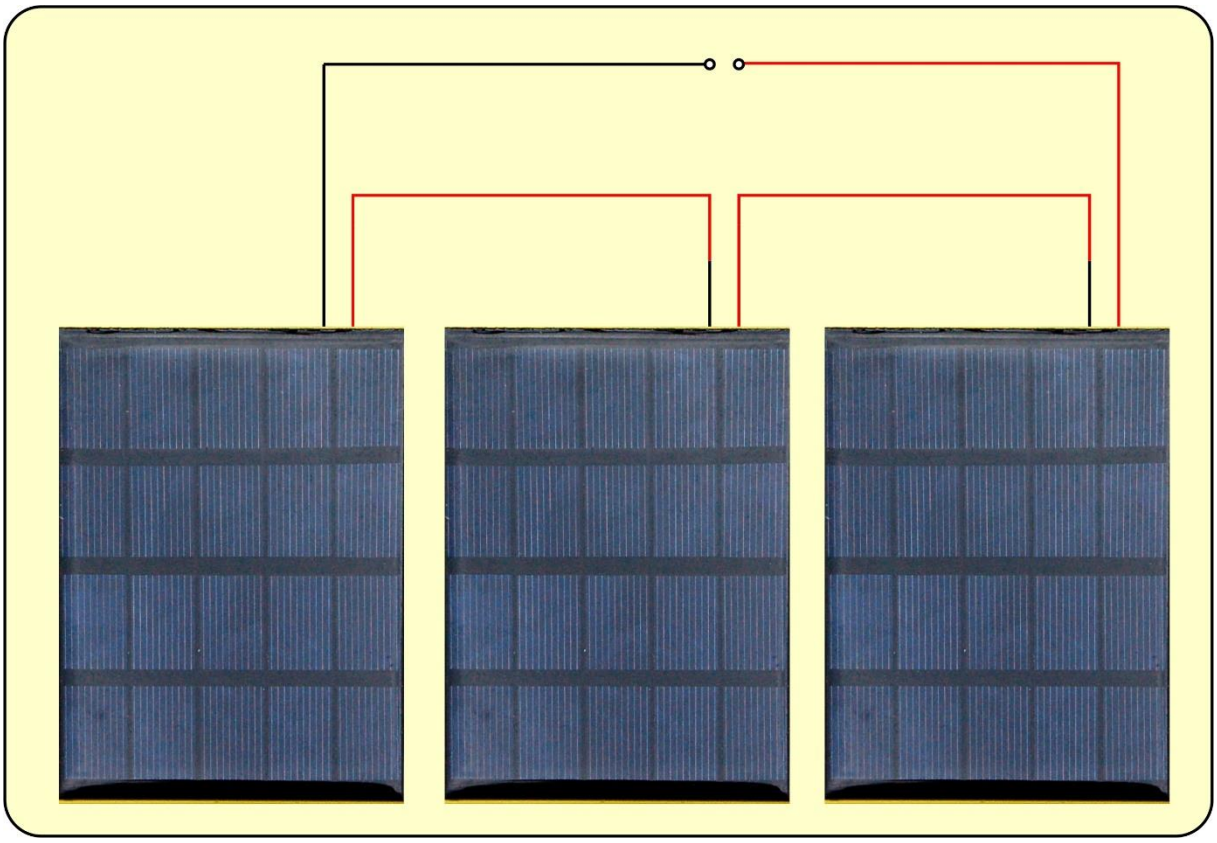


Abbildung 5: Solarpanele in Reihe - höhere Spannung

Der MPP-Finder

Erste Versuche zu dem Thema MPP hatte ich händisch durchgeführt. Mittels Volt- und Amperemeter und diversen Widerständen nahm ich die Werte für die U-P-Kennlinie eines Panels auf. Das brachte das erwartete Ergebnis, war mir aber viel zu aufwendig, um verschiedene Parameter wie Bestrahlung, Parallel- und Serienschaltung zu untersuchen.

Ich brauchte also einen Messknecht, den ich im ESP32 fand. Aufgabe einer entsprechenden Schaltung waren:

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

Klick! – Natürlich, als Stellglied könnte ich ja einen Buck-Konverter einsetzen, der durch ein PWM-Signal aus dem ESP32 angesteuert wird. Ein Hochlastwiderstand mit einem Widerstandswert von ein paar Ohm als "Verbraucher" wird durch den Step-Down-Wandler mit einer dosierbaren Menge an Energie zum Schwitzen gebracht. Der Duty-Cycle des PWM-Signals ist quasi das Ventil für die am Widerstand ankommende Energie vom Solarpanel. Der Wandler arbeitet also hier primär nicht als Spannungs- oder Stromregler, sondern als programmgesteuerte, veränderbare Last.

Probleme gab es aber bei der Messwerterfassung. Der ESP32 ist zwar mit seinen vielen darauf befindlichen Baugruppen eine nette eierlegende Wollmilchsau, aber erstens ist die Spannungskennlinie der ADCs alles andere als linear und zweitens setzt die Spannungserfassung erst bei ca. 150mV ein. Für eine Strommessung im Bereich bis 500mA mittels Messwiderstands von 0,1 Ohm (U_{Shunt} = 50mV) wären also ohne Verstärkung keine Messwerte zu erfassen gewesen.

Bei der Recherche nach einer besseren Lösung stieß ich auf ein BOB (Break-Out-Board) mit dem IC INA219. Für meine Zwecke geradezu ideal, weil es Spannungen bis 32V und Stromstärken bis 3,2A erfassen und darüber hinaus auch noch die Leistung in Watt über den I2C-Bus an den Controller liefern kann – Bingo!

Die erfassten Daten werden in einer Datei auf dem ESP32 gespeichert. Auf den PC werden die Dateien mit Thonny zum PC übertragen und dort in einer Tabellenkalkulation ausgewertet.

Die Schaltung

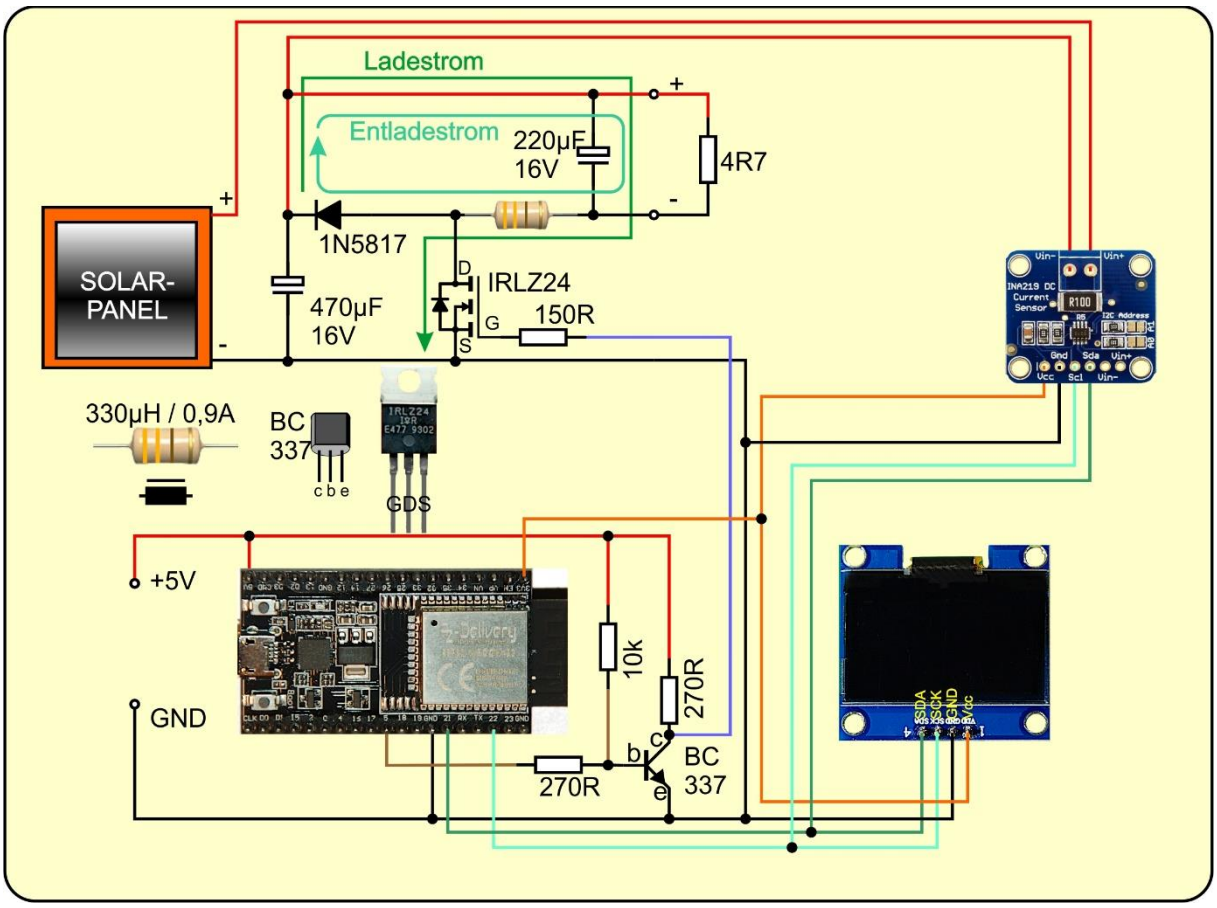


Abbildung 6: MPP-Finder - Schaltung

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.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**.

```
# powerline.py
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
```

Dann beginnen die Vorbereitungen. Ich erzeuge eine I2C-Bus-Instanz, mit der ich auch gleich das Display-Objekt instanziiere. 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 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
step=1024/steps
```

Der Konstruktor der Klasse INA219 fordert eine ganze Reihe von Argumenten, die bis auf **i2c** alle optional 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 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 liefert der Pin GPIO5 mit einer Frequenz von 50kHz. Mit einem Duty-Cycle von 1023 schaltet der BC337 durch und zieht das Gate des IRLZ24 auf GND-Potenzial, wodurch der MOSFET sperrt (siehe Abbildung 6) 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)
```

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 gefüttert 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, 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 schließen zu müssen. Sie wird automatisch geschlossen, wenn der **with**-Block verlassen wird. Die for-Schleife iteriert über die Dateieinträge. Die Zeilen werden 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 die 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:
               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 Muduls) 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.

```
        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))
```

Die except-Blöcke informieren über eventuell aufgetretene Fehler.

```
        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 wird eine **OSError-Exception** geworfen.

Um die Schleife zu beenden wird **free** 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
```

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 und 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**.py, hinein und speichere die neue Datei direkt auf dem ESP32 unter dem Namen **main**.py ab. Mit jedem Neustart oder Reset startet der ESP32 jetzt automatisch mit einem neuen Durchlauf.

Dateitransfer zum PC

Die Auswertung der Messungen erfolgt am PC. Ich verwende dazu Libre Office Calc. Welche Schritte dazu nötig sind, zeige ich jetzt.

Markieren Sie die gewünschten Dateien in Thonny im Bereich MicroPython device.

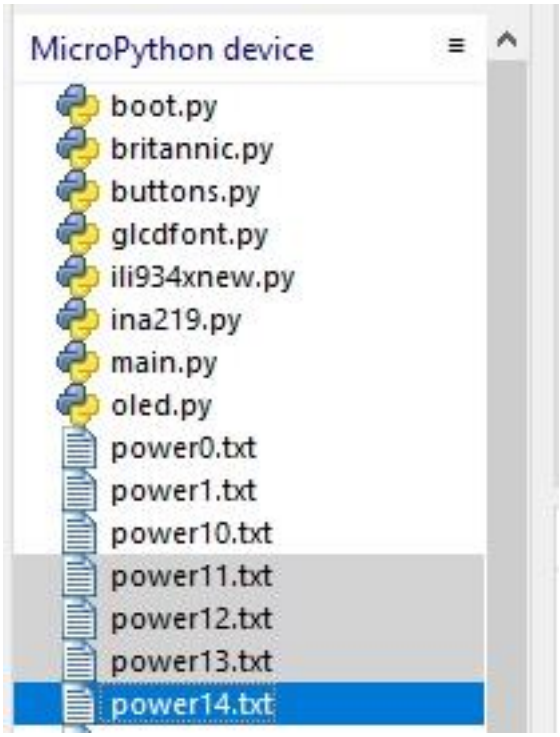


Abbildung 7: Dateien zum PC schicken

Mit Rechtsklick auf die Markierung öffnen Sie das Kontextmenü. Laden Sie Dateien auf den PC herunter.

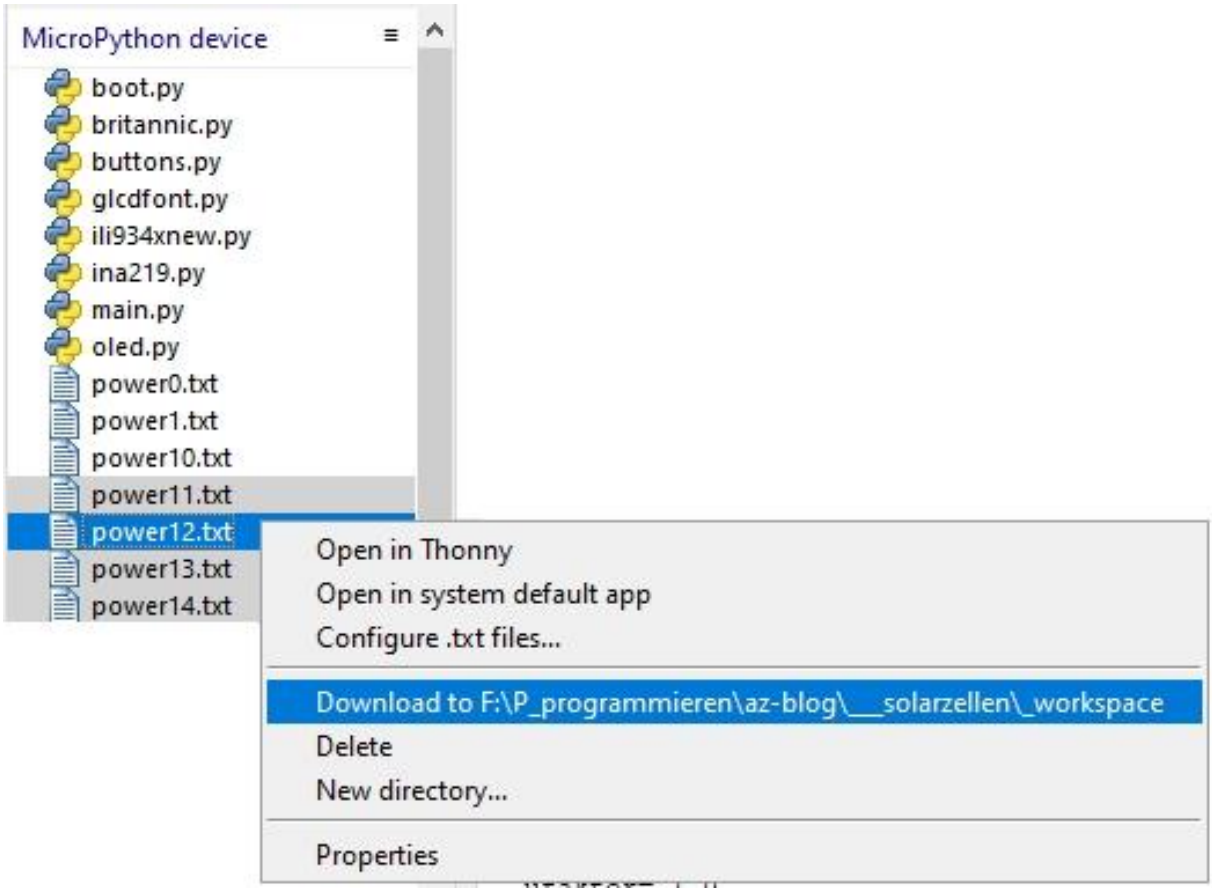


Abbildung 8: Download starten

Die Dateien befinden sich jetzt im Arbeitsverzeichnis von Thonny. Öffnen Sie eine Datei in einem Editor und markieren Sie den ganzen Text. Kopieren Sie ihn in die Zwischenablage.

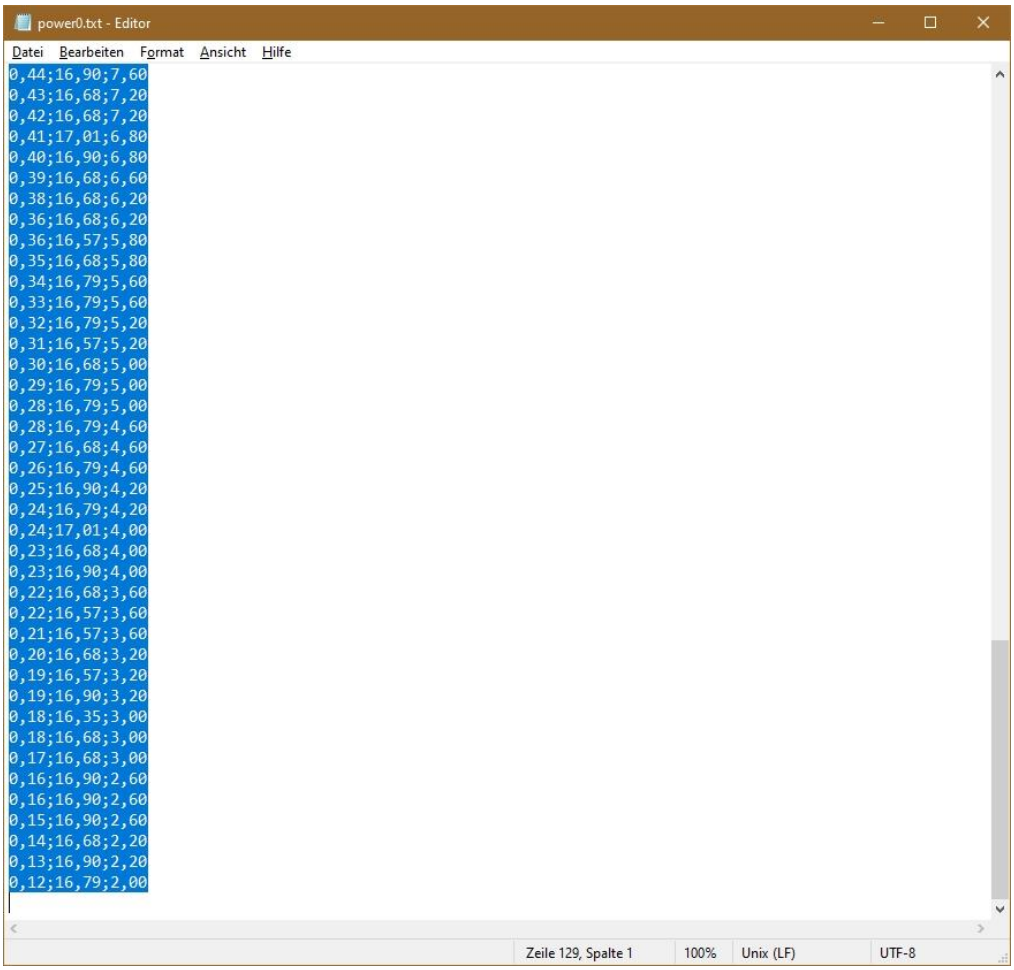


Abbildung 9: Datei im Editor öffnen

Starten Sie jetzt Libre Office Calc. Rechtsklick auf das Feld A1 öffnet das Kontextmenü – **Inhalte einfügen – Inhalte einfügen**.

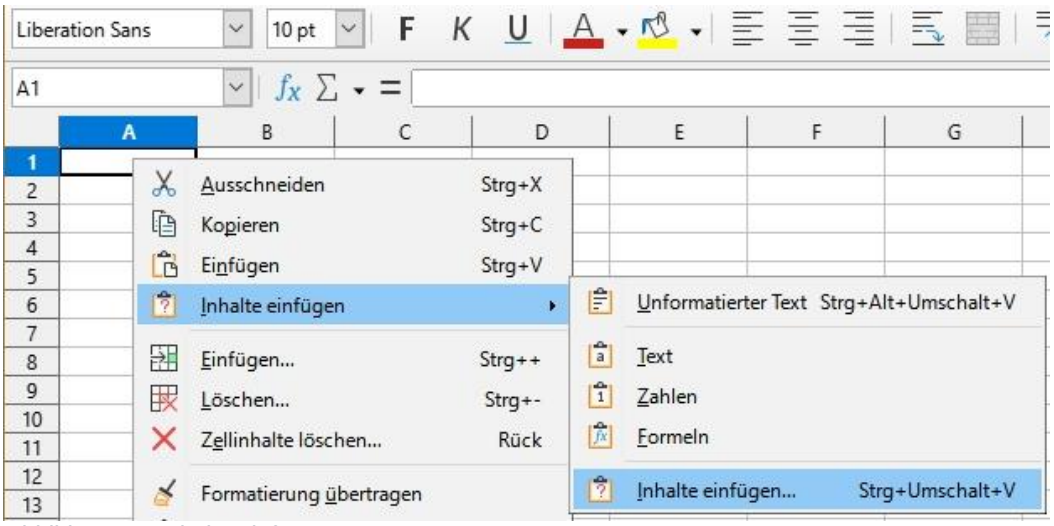


Abbildung 10: Inhalte einfügen

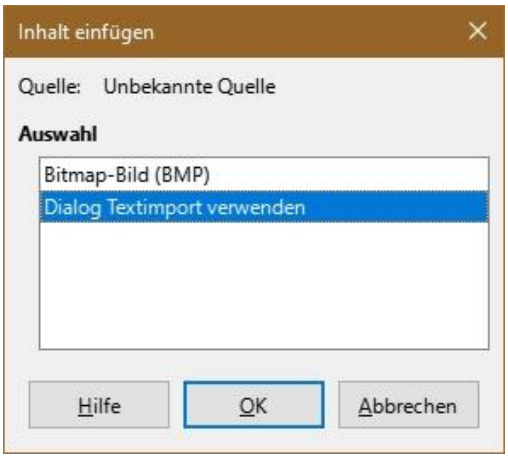


Abbildung 11: Textimport verwenden

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

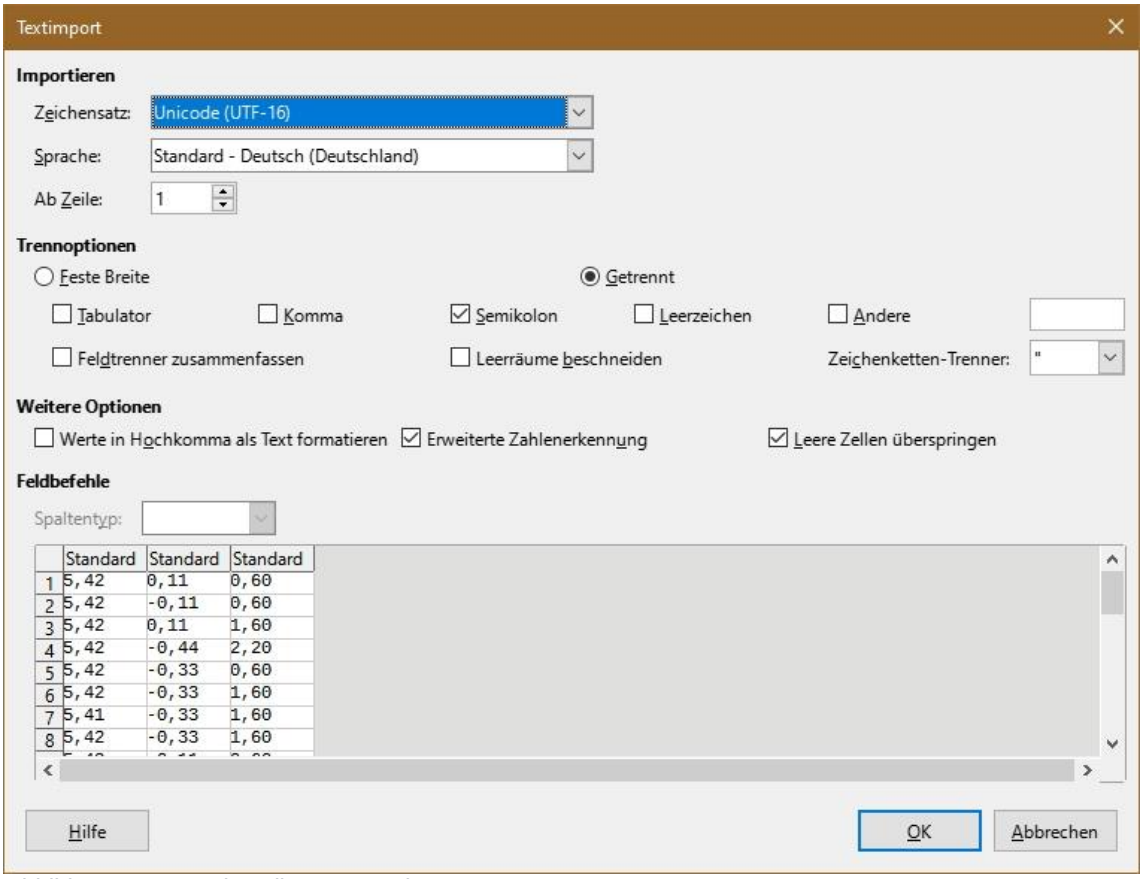
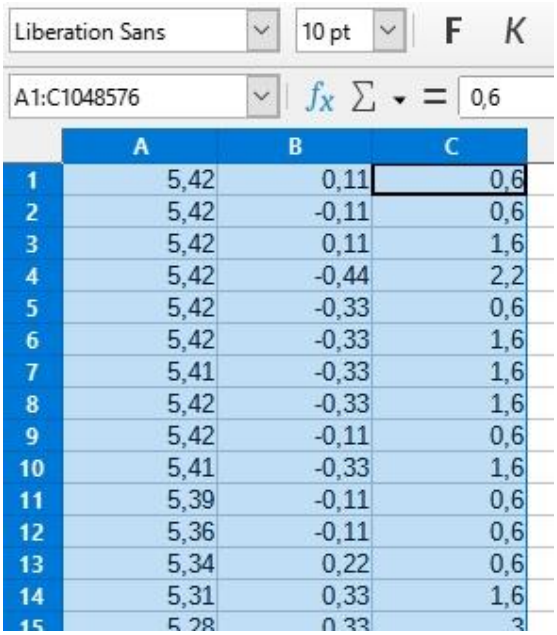


Abbildung 12: 04 - Einstellungen Textimport

Markieren Sie die Spalten A, B und C.



	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 13: Spalten markieren

Klick auf Diagramm einfügen.

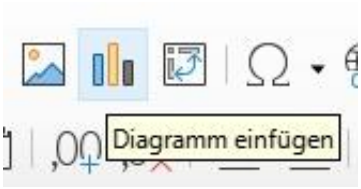


Abbildung 14: Diagramm einfügen

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

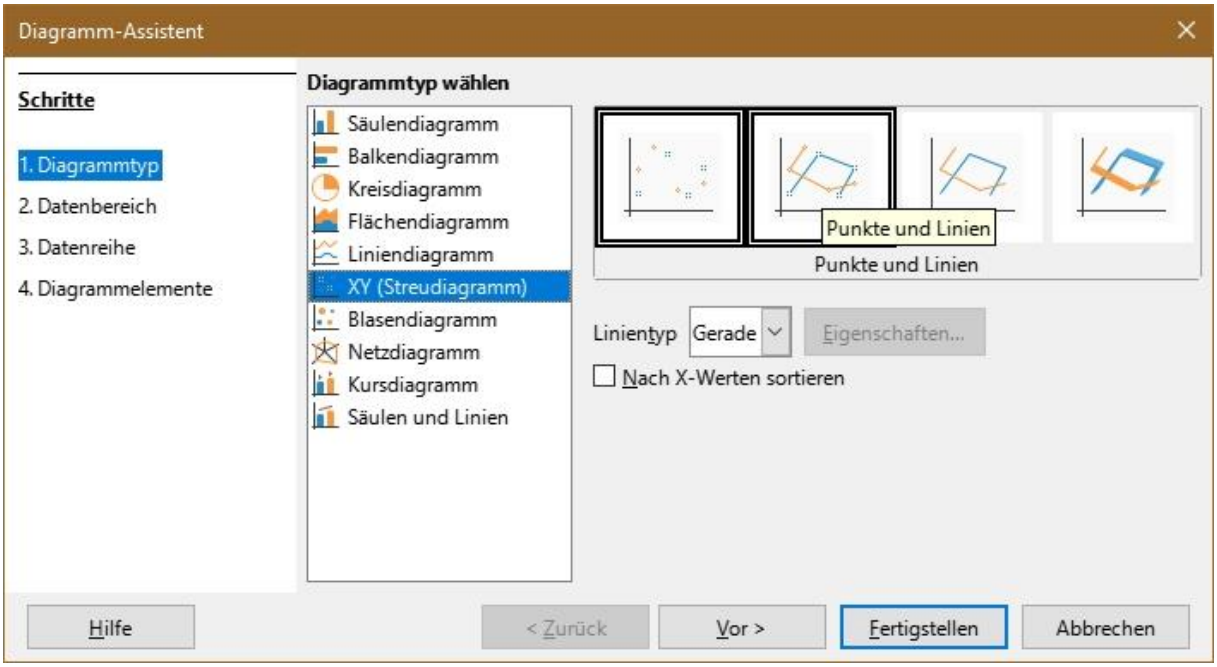
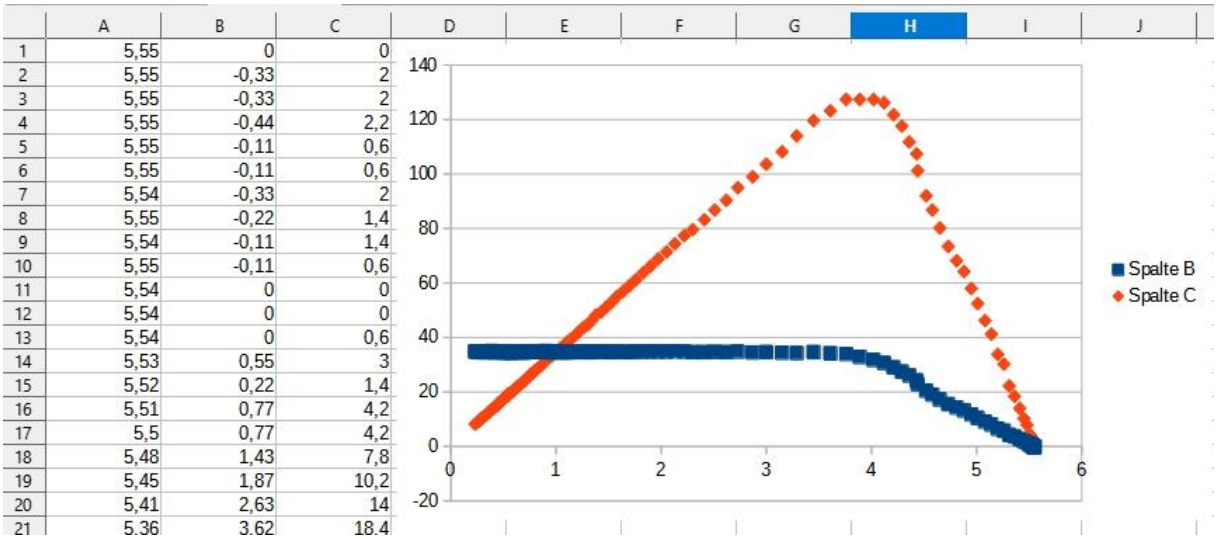


Abbildung 15: Diagrammassistent

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 16: Fertiges Diagramm

Die blaue Kurve ist die U-I-Kennlinie. Dort, wo sie nach unten abknickt, liegt das Maximum der roten Kurve, der MPP.

Ja, die Übertragung vom ESP32 auf den PC via USB-Kabel ist eine Option, aber es wäre doch schöner, wenn die Daten simultan auf dem PC landen würden, gleich nach jeder Messung. Genau diese Lösung ist Inhalt in der nächsten Folge. Für den Transfer werde ich die Funkeigenschaften des ESP32 nutzen. Das ist einfacher und geht schneller. Schließlich laufe ich ja auch nicht dem Omnibus nach, wenn ich schon drinsitze.

Also, bis dann!