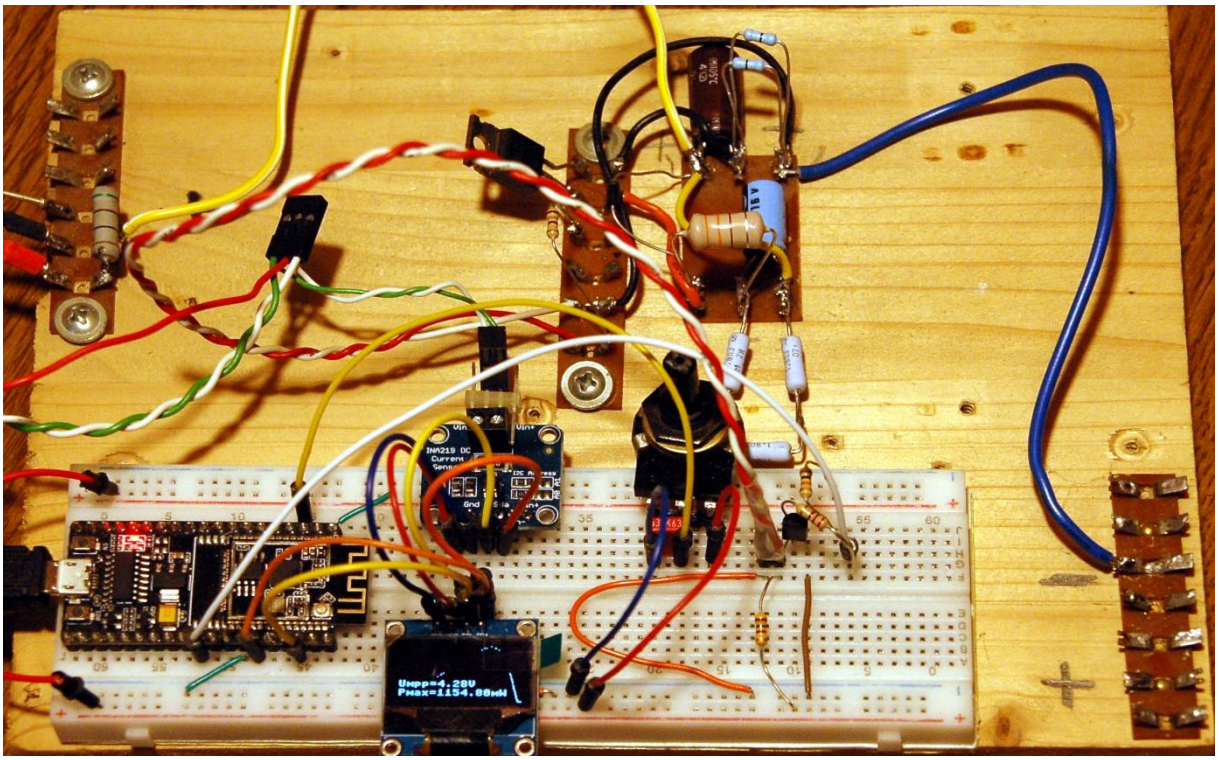

MPP-Finder - Aufbau

Diesen Beitrag gibt es auch als PDF-Datei.

Vor einiger Zeit hatte ich einen ESP32 dazu benutzt, um über eine LED und einen Fotowiderstand (LDR = Light Dependent Resistor) die Spannung eines Schaltreglermoduls einzustellen. Machbar und für den Fall höherer Stromstärken, so ab 3A, auch sinnvoll, weil, man sich nicht um den Regler an sich kümmern muss. Ein Nachteil ist der träge LDR.

Aber ein einfacher Schaltwandler im Bereich kleiner bis mittlerer Leistung kann auch durch einen ESP32 selbst aufgebaut werden. Das Schöne daran ist, dass so ein Konverter durch verschiedene Sensoren oder auch über Funk angesteuert werden kann. Je nach Konvertertyp kann man damit Gleichspannungen herunter oder hochsetzen. So eine Schaltung und das zuständige MicroPython-Programm stelle ich Ihnen heute im ersten Teil der Serie zu diesem Thema vor. Willkommen also zu einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

ESP32 und Schaltnetzteile

Schaltnetzteile müssen einen Gleichstrom zerhacken, damit die Spannung am Eingang in eine niedrigere oder höhere am Ausgang umgesetzt werden kann. Diese Aufgabe eines Schalters übernimmt ein MOSFET-Transistor, den wir von einem PWM-Ausgang unseres Controllers aus ansteuern werden. Drei weitere Bauteile,

eine Diode, eine Spule und ein Elektrolytkondensator (Elko) erledigen dann den Rest. Damit die Spule nicht zu voluminös ausfallen muss, ist es nötig, den Schaltvorgang mit höheren Frequenzen durchzuführen.

Leider ist die PWM-Funktion der ESP8266-Familie mit einer maximalen Frequenz von 1000Hz = 1kHz ungeeignet für die folgenden Projekte, denn hier benötige ich 50kHz aufwärts. Das kann nur ein ESP32, der bis zu 40 MHz liefert. Allzu hoch wollen wir aber mit der Frequenz nicht gehen, weil die Pulslänge auch noch gut einstellbar sein soll. Das Verhältnis von Pulslänge zu Periodendauer des PWM-Signals nennt man Tastgrad, engl. Duty-Cycle. Die Auflösung des Tastgrades nimmt aber mit steigender Frequenz ab.

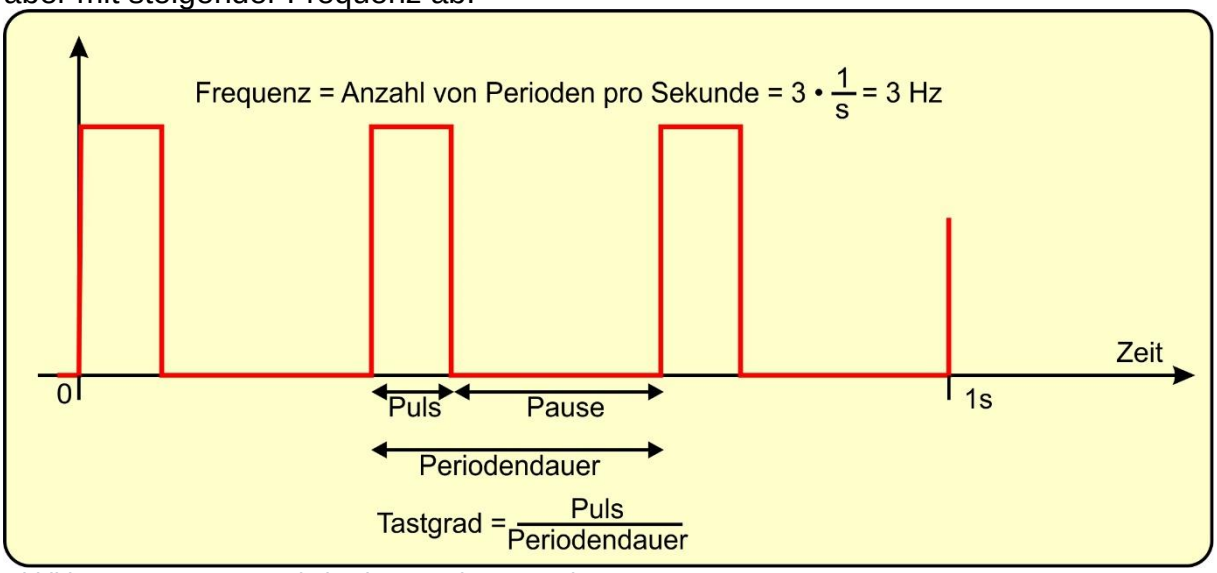

Abbildung 1: Frequenz, Periodendauer und Duty-Cycle

Für unsere Zwecke kommen zwei Schaltungen in Frage, der Hochsetzsteller (Boost-Converter) und der Tiefsetzsteller (Buck Converter). Um die Funktionsweise dieser Baugruppen kümmern wir uns als Erstes.

Der Hochsetzsteller

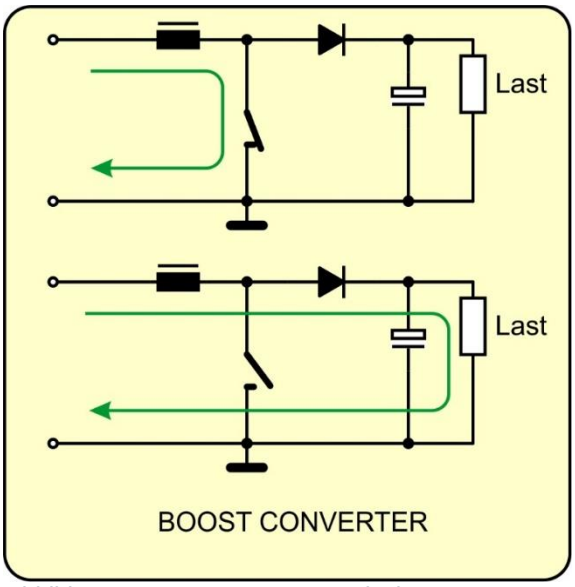

Abbildung 2: Boost Converter - Prinzip

Während der Schalter geschlossen ist, fließt ein Strom durch die Spule wodurch ein Magnetfeld aufgebaut wird. Wird der Schalter geöffnet, bricht das Magnetfeld zusammen. In der Spule fließt nun kurzzeitig ein Induktionsstrom in derselben Richtung wie zuvor der Ladestrom. Die Induktionsspannung an der Spule überlagert die am Eingang liegende Versorgungsspannung. Die Spannungswerte werden addiert wie beim Hintereinanderschalten von Batteriezellen. Sobald diese Spannung höher wird als die Summe der Spannung am Kondensator und der Schwellenspannung der Diode, fließt ein Strom durch die Diode, der den Kondensator auflädt und die Last speist. Die Spannung am Ausgang ist stets höher als die Versorgungsspannung.

Der Tiefsetzsteller

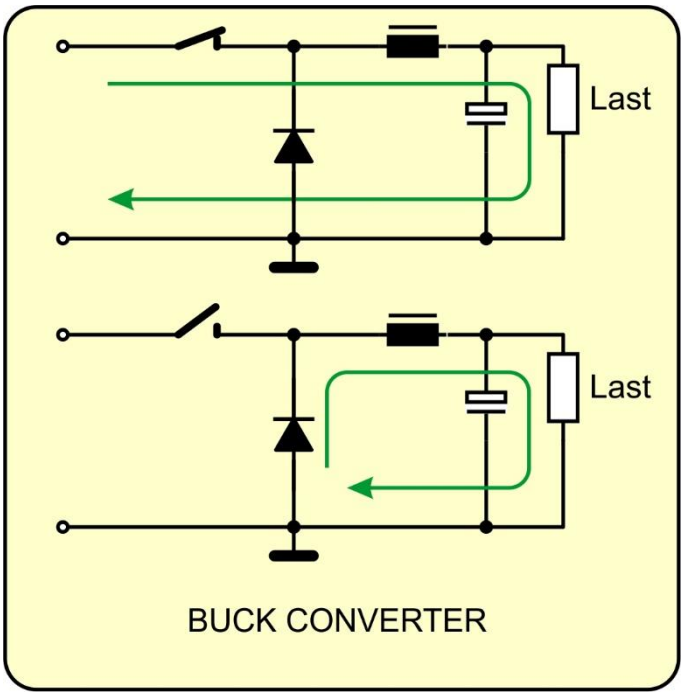

Abbildung 3: Buck Converter - Prinzip

Bei geschlossenem Schalter, fließt ein Strom durch die Spule und die Last wodurch ein Magnetfeld aufgebaut wird. Wird der Schalter geöffnet, bricht das Magnetfeld zusammen. Nach der Regel von Lenz fließt der Induktionsstrom wieder in der Richtung des Ladestroms wie beim Boost Converter. Je länger der Ladestrom durch die Spule fließt, desto mehr Energie steckt in dem Magnetfeld. Beim Öffnen des Schalters, gibt die Spule diese Energie wieder ab. Die Induktionsspannung kann aber nicht höher als die Versorgungsspannung werden.

ESP32 und ein Feldeffekt-Transistor als Schalter

Natürlich kann niemand einen mechanischen Schalter 50000-mal in der Sekunde schließen und öffnen. Deshalb ersetze ich den Schalter durch einen Leistungstransistor vom Typ IRLZ24. Das ist ein N-Kanal-MOSFET, der bis zu 55V vertragen kann. Damit lassen sich Ströme bis 18A schalten, was für unsere Zwecke wohl gut ausreicht. Wichtiger als die maximale Stromstärke sind in unserem Fall aber der recht niedrige Einschalt-Widerstand von 0,06 Ohm und die Tatsache, dass Transistor ein Logic-Level-Gate besitzt. Das bedeutet, dass der Transistor durch einen TTL-Pegel von 5V bereits voll durchgesteuert werden kann.

Leider liefert der ESP32 aber nur 3,3V, sodass wir für den Buck Converter, den wir heute aufbauen, einen weiteren Kleinleistungstransistor (BC337) als Treiberstufe benötigen. Abbildung 4 zeigt die Schaltung.

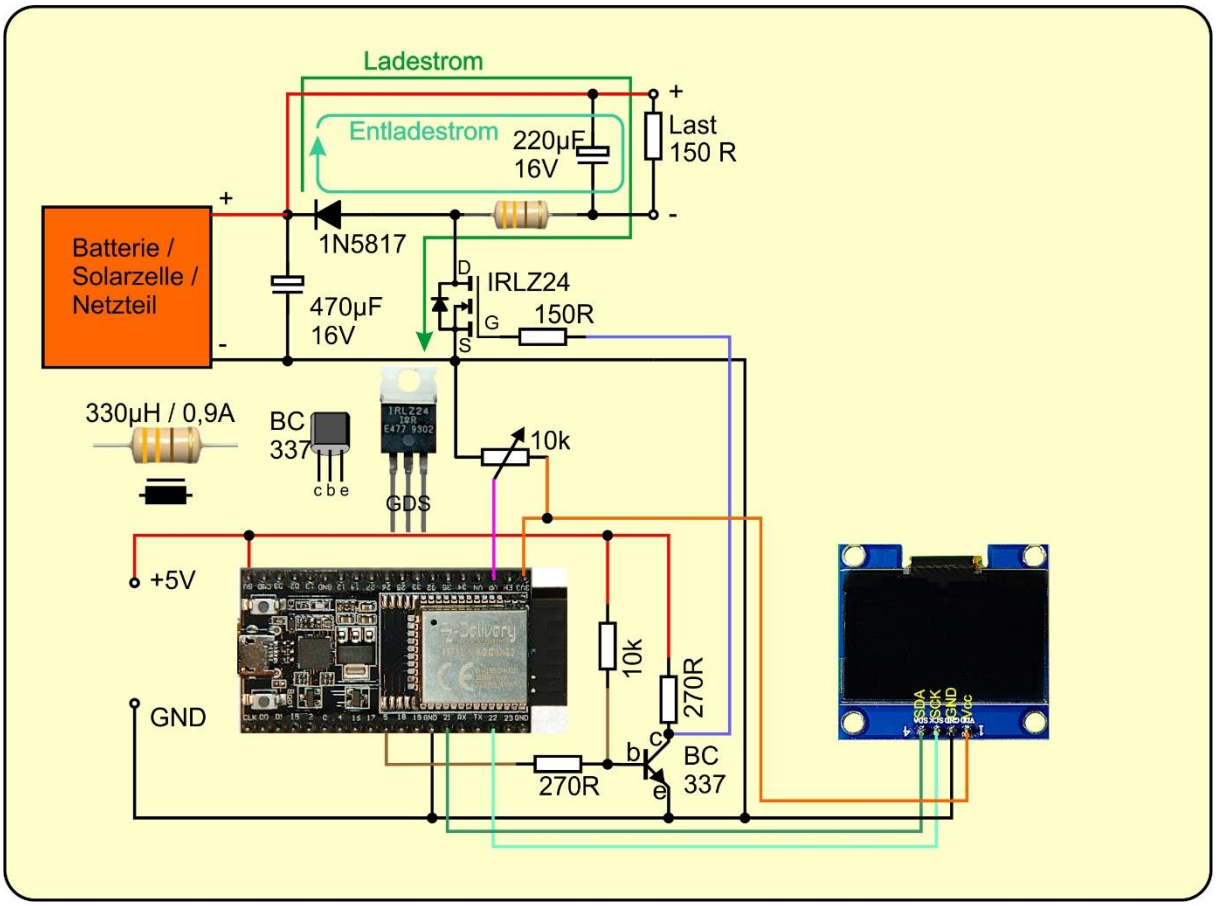

Abbildung 4: Buck Converter - Schaltung

Der ESP32 ist im Moment noch unterfordert. Er bedient nur das Display und liefert das PWM Signal für die Wandlerstufe. Aber unser Vorhaben ist ausbaufähig.

Wenn Sie die Schaltung in Abbildung 4 mit der in Abbildung 3 vergleichen, fällt Ihnen sicher auf, dass der Schalter, also der MOSFET IRLZ24 nicht in der Plus-Leitung sondern in der Minus-Leitung liegt. Die Plusleitung ist sozusagen das Masse-Potenzial. Das ist deshalb so gemacht, damit der Source-Anschluss des Transistors am Minus-Pol der Versorgungsspannung liegt, die auch mit dem GND-Level der Controller-Versorgung verbunden ist. Würde man den MOSFET in die Plusleitung legen, dann wären weitere Schaltungsmaßnahmen nötig um die Pegel anzugleichen. Die Funktionsweise der Schaltung ist identisch mit der oben beschriebenen, wie Sie aus den Schleifen für den Lade- und Entladestrom ersehen können.

Die Verwendung des BC337 bedingt einen weiteren Punkt der beachtet werden muss. Am Kollektor c des BC337 erscheint das Signal vom ESP32 in invertierter Form, weil eine logische 1 (3,3V) an der Basis b den Transistor durchsteuert. An c liegt dann nahezu GND-Potenzial, was einer logischen 0 entspricht. Der Pullup-Widerstand an der Basis sorgt übrigens für einen sauberen Start der Schaltung mit gesperrtem MOSFET. Solange der Anschluss GPIO5 beim Booten des ESP32 nämlich noch ein hochohmiger Eingang ist, schaltet der BC337 durch und hält die Gate-Leitung des MOSFET auf GND-Potenzial.

Über den Eingang VP = GPIO36 wird eine analoge Spannung von 0V bis 3,3V eingelesen. Der Wert des ADC-Wandlers (0..1023) wird dann hergenommen, um die Pulsweite des PWM-Signals einzustellen. Einzelheiten dazu erfahren Sie bei der Besprechung des Programms.

Die Wandlerschaltung ist auf Lötleisten aufgebaut. Ebenso gut könnte man auch eine Lochrasterplatine verwenden.

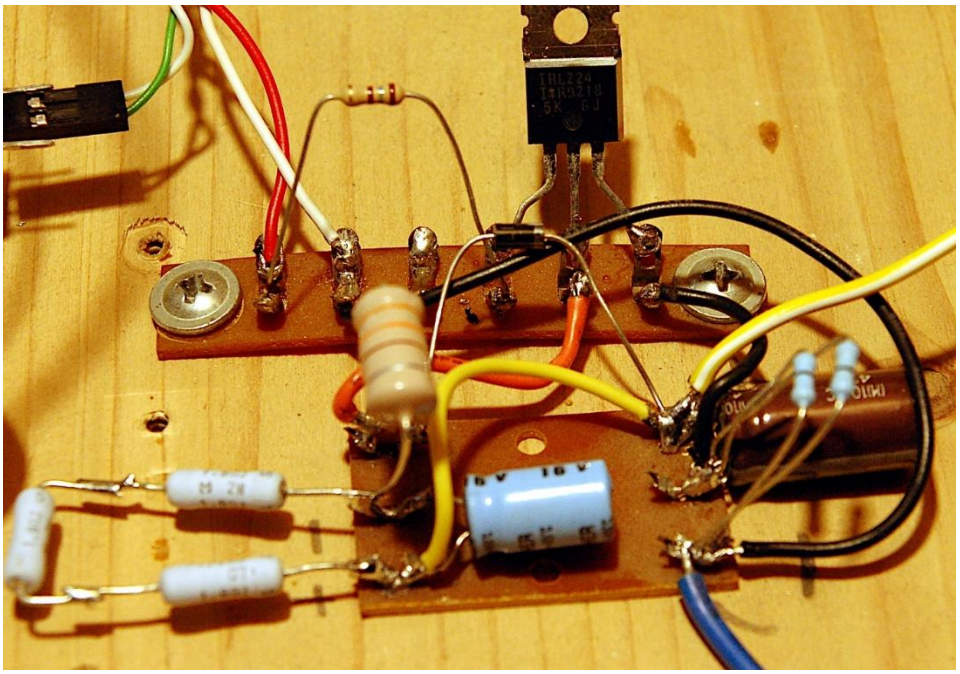

Abbildung 5: Buck Converter - Aufbau mit Kontaktleisten

Das Display wird über den I2C-Bus angesteuert. Wenn es, wie im Schaltplan angegeben, mit 3,3V versorgt wird, bedarf es keines Pegelwandlers auf den I2C-Bus-Leitungen SDA und SCL zwischen Display und ESP32.

An Folgendes sollten Sie stets denken:

- Zwischen dem Pluspol des Ausgangs und GND liegt stets die volle Versorgungsspannung der Batterie, des Solarpanels oder der Netzversorgung.
- Die Schaltung kann nicht funktionieren, wenn Sie GND mit dem Minuspol am Ausgang verbinden. Das würde auch geschehen, wenn der ESP32 aus einem USB-Anschluss versorgt würde und der Minuspol des Ausgangs weiteren Schaltungsteilen zugeführt wird, deren GND-Potenzial mit dem des PC, also über den Schutzleiter des 230V-Netzes, verbunden ist. Speziell geschieht das, wenn man mit einem am Netz betriebenen DSO die Masseleitung (Schirmung) des Messkabels mit dem Minuspol des Ausgangs verbindet. Die Erfassung von Signalen im Wandlerbereich ist daher nur mit einem Hand-Held-Scope möglich, das keine Verbindung zum Schutzleiter hat.
- Die 5V-Versorgung des ESP32 könnte auch mittels eines Linearwandlers (LM7805) von der Versorgungsspannung abgeleitet werden.

In der Tabelle habe ich die benötigten Teile zusammengefasst.

Hardware

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	Potentiometer linear 10kΩ
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	Schottky-Diode 1N5817
1	Elektrolytkondensator 470µF 16V
1	Elektrolytkondensator 220µF 16V
1	Speicherdrossel 330µH 1A
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
wandler.py Betriebssoftware des Wandlers

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.

Das Programm

Fürs Erste muss das Programm nicht viel können. Ich lasse das Poti an GPIO36 in der Hauptschleife abtasten, bereite den Wert auf und gebe das Ergebnis als Wert für den Duty-Cycle am Pin GPIO5 aus. Damit kann ich am Ausgang an der Last Spannungen von 0 bis nahe unter der Versorgungsspannung messen. Aber lassen Sie uns zunächst einmal die Vorbereitungen durchgehen.

```
# pwm-netzteil-steuerung
# wandler.py
from machine import PWM,Pin,ADC,SoftI2C
from oled import OLED
from time import sleep
from sys import exit
```

Ich komme mit den bereits im Kernel von MicroPython integrierten Modulen gut zurecht, brauche aber für die Anzeige mein oled-Modul, das seinerseits auf dem Modul ssd1306 aufsetzt. Die beiden Dateien ssd1306.py und oled.py müssen daher in den Flash-Speicher des ESP32 hochgeladen werden. Folgen sie bitte für den Download den beiden Links.

Mit dem Import der entsprechenden Module und Methoden stehen alle Vorgaben für die benötigten Objekte bereit. Durch die Formulierung **from xxxx import yyyy, zzzz** binde ich die Blaupausen für die Objekte in den globalen Namensraum ein und erspare mir damit die Referenzierung durch die Punktnotation. Allerdings würde ich bereits vorhandene, gleichlautende Instanzen damit überschreiben. Das ist hier aber nicht der Fall.

Ich erzeuge nun die nötigen Instanzen für meine Anwendung.

```
pwmPin=5
gate=PWM(Pin(pwmPin),freq=50000,duty=0)
```

Das PWM-Signal soll an GPIO5 ausgegeben werden. Die Frequenz setze ich auf 50kHz und den Duty-Cycle auf 0

```
taste=Pin(0,Pin.IN,Pin.PULL_UP)
```

Zur Programmsteuerung verwende ich die Flash-Taste des ESP32. Sie liegt an GPIO0 und wird als Eingang mit Pullup initialisiert.

```
adcPin=36
Uout=ADC(Pin(adcPin), atten=ADC.ATTN_11DB)
Uout.width(ADC.WIDTH_10BIT)
```

GPIO36 =VP nutze ich als analogen Eingang. Den Abschwächer (Attenuator) stelle ich auf ADC.ATTN_11DB und erreiche damit den maximalen Spannungsbereich von 0V .. 2,45V. Der Wandler liefert aber im Bereich 0V bis 0,15V stets den Wert 0. Die

Auflösung des Bereichs stelle ich auf 10 Bit ein. Der ADC liefert also einen maximalen Wert von 1023, welcher der Spannung 2,45 V entspricht oder besser entsprechen sollte. Ein LSB hätte dann den Wert von 2,45 V / 1023 = 2,48 mV, theoretisch. Denn praktisch ist die Wandlerkurve weder linear noch gibt es einen Offset von 0.

Aber Achtung:
Legen Sie an die Eingänge nie Spannungen von mehr als 3,6V an. Andernfalls können Sie ihren Controller auf dem Friedhof der Kuschel-Chips begraben.

```
i2c=SoftI2C(Pin(22), Pin(21))
d=OLED(i2c)
d.clearAll()
```

Zum Verbindungsaufbau zum Display muss der I2C-Bus aktiviert werden. Der Anschluss SCL liegt an GPIO22, SDA an GPIO21. Die I2C-Instanz i2c übergebe ich an den Konstruktor des Displayobjekts und lasse dann die Anzeige löschen.

Damit die Spannungswerte weniger streuen, lasse ich den ADC mehr als einmal den Wert einlesen. Dazu habe ich mir eine Funktion gebastelt, in deren Parameter **count** die Anzahl der Messungen als Argument übergeben wird.

Die Variable **wert** wird mit 0 initialisiert und dann lasse ich in der for-Schleife **count** Messungen aufaddieren. Der Index **n** läuft nach dem MicroPython-Standard von 0 bis **count-1** und enthält nach dem letzten Durchlauf den Inhalt von **count**, wodurch die Schleife abgebrochen wird. Als Funktionsergebnis wird das arithmetische Mittel aller Messungen zurückgegeben.

```
def spannung(count):
    wert=0
    for n in range(count):
        wert=wert+Uout.read()
    return wert//count
```

Mit steigender Spannung liefert der ADC ganzzahlige Werte zwischen 0 und 1023. Wegen der Invertierung des PWM-Signals durch den BC337 entsprechen aber niedrigere Duty-Cycle-Werte einer höheren Spannung an der Last. Das berücksichtige ich bei der Aufbereitung des PWM-Werts, indem ich den ADC-Wert von 1024 subtrahiere. Wir betreten die Hauptschleife.

```
while 1:
    wert=1024-spannung(25)
    d.writeAt("{}     ".format(wert),3,2)
    wert=min(max(wert,2),930)
    gate.duty(wert)
    sleep(0.1)
    d.writeAt("{}     ".format(wert),3,3)
    if taste.value()==0:
        exit()
```

Zur Kontrolle lasse ich mir den Inhalt von **wert** im Display anzeigen und sehe alles von 0 bis 1023, wenn ich das Poti durchdrehe, während das Programm läuft.

Nun stellt sich aber heraus, dass der Duty-Cycle nicht, wie es in der MicroPython-Beschreibung zum ESP32 heißt, von 0 bis 1023, sondern nur von ca. 2 bis 930 angeben werden kann. Daher grenze ich den ADC-Wert auch auf diesen Bereich ein. Ich stelle den Duty-Cycle damit ein und lasse mir auch den korrigierten Wert im Display anzeigen.

Durch Drücken der Flashtaste kann das laufende Programm beendet werden.

Es wird spannend - der Test

Wir wollen sehen, ob alles richtig funktioniert. Als Versorgungsspannung schließe ich eine Batterie oder einen Akku an den Eingang an. Parallel zur Last klemme ich ein DVM (Digital-Volt-Meter) an und den Kollektor des BC337 hänge ich den Eingang zu einem Kanal des DOSs (Digitales Speicher Oszilloskop). Dann starte ich das Programm.

Das Voltmeter zeigt mir Spannungen zwischen 0,3mV und 4,88V an, wenn das Poti auf dem linken bzw. rechten Anschlag steht. Am Schirm des DSOs erscheint die Rechteckkurve des PWM-Signals.

Abbildung 6: PWM-Signal

Dem korrigierten ADC-Wert von 233 entspricht ein Duty-Cycle von 69,3% und eine Spannung von 3,22 V bei einer Versorgungsspannung von 4,98V.

ESP32 und Stromstärkemessung – Ausblick

Könnte der ESP32 denn auch die Stromstärke durch die Last messen? Antwort von Radio Eriwan: Im Prinzip ja aber nur unter Anwendung eines Operationsverstärkers zur Anhebung des Spannungsabfalls an einem niederohmigen Shunt und nur am kalten Ende der Versorgungsspannung vor dem Sourceanschluss des MOSFET. Das Problem dabei ist aber, dass der ESP32 Spannungen bis 0,15V gar nicht erfassen kann. Und 0,15V an einem Widerstand von 0,1Ω entspricht immerhin schon einer Stromstärke von 1,5A. Das Anheben einer Spannung in der Nähe der GND-Schiene erfordert aber schon wieder spezielle Operationsverstärker, die Rail to Rail arbeiten können.

Machen Sie Sich aber keine Sorgen in den nächsten Folgen geht es um die Ermittlung des MPP (Maximum-Power-Point) von Solarpanelen. Und dort setze ich einen wahren Tausendsassa ein, der Spannung Strom und Leistung messen kann und das genauer als es sich der ESP32 je erträumen kann. Natürlich kommt auch unser Wandler wieder zum Einsatz, zwar nicht als steuerbare Spannungsquelle, sondern als variabler Lastwiderstand. Für dieses IC werde ich in der nächsten Folge ein MicroPython-Modul entwickeln.

Übrigens, unser Abwärtswandler gibt auch eine brauchbare Wechselspannungsquelle ab. Ersetzen sie doch einfach die Potidrehungen durch eine Funktion, die periodisch Werte zwischen 2 und 930 durchläuft, zum Beispiel Sinus oder Dreieck oder Sägezahn oder …

Viel Spaß beim Forschen! Bis zum nächsten Mal.

Hier kommt noch das Programm **wandler**.py als vollständiges Listing.

```
# pwm-netzteil-steuerung
# wandler.py
from machine import PWM,Pin,ADC,SoftI2C
from oled import OLED
from time import sleep
from sys import exit

pwmPin=5
gate=PWM(Pin(pwmPin),freq=50000,duty=0)

taste=Pin(0,Pin.IN,Pin.PULL_UP)

adcPin=36
Uout=ADC(Pin(adcPin), atten=ADC.ATTN_11DB)
Uout.width(ADC.WIDTH_10BIT)

i2c=SoftI2C(Pin(22),Pin(21))
d=OLED(i2c)
d.clearAll()

def spannung(count):
    wert=0
    for n in range(count):
```

```
        wert=wert+Uout.read()
    return wert//count

i=0
while 1:
    wert=1024-spannung(25)
    d.writeAt("{}     ".format(wert),3,2)
    wert=min(max(wert,2),930)
    gate.duty(wert)
    sleep(0.1)
    d.writeAt("{}     ".format(wert),3,3)
    if taste.value()==0:
        exit()
```