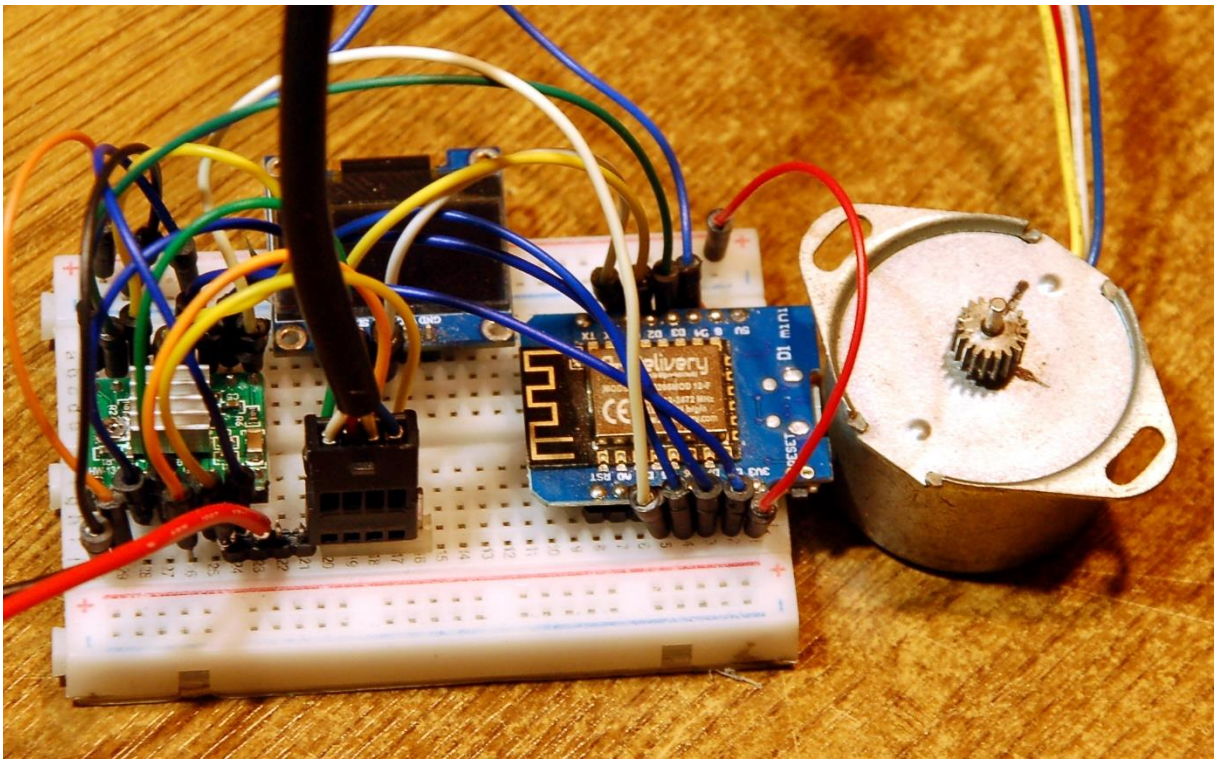
Ein A4988 am ESP8266

Diesen Beitrag gibt es auch als PDF-Dokument zum Download.

In der letzten Folge hatten wir uns das Arduino-Motor-Shield und die Grundlagen der Schrittmotoransteuerung angeschaut. Für das Verständnis der Steuervorgänge ist das Shield ideal, weil man daran den Fortgang, jeden Schritt, gut verfolgen kann. Nur wenn man Größeres vorhat, dann wird erstens der Schaltungsaufwand rasch ebenfalls größer und mehr als Halbschritt ist ohne wiederum schnell wachsendem Programmieraufwand nicht drin. Man muss auch nicht unbedingt "das Rad neu erfinden", denn es gibt professionelle Lösungen von Allegro und Texas Instruments (TI). Der A4988 von Allegro und auch der DRV8825 von TI sind beide nicht einmal halb so groß wie ein ESP8266 D1 mini und bieten ein ganzes Bündel an guten Eigenschaften, die das Motor-Shield nicht hat.

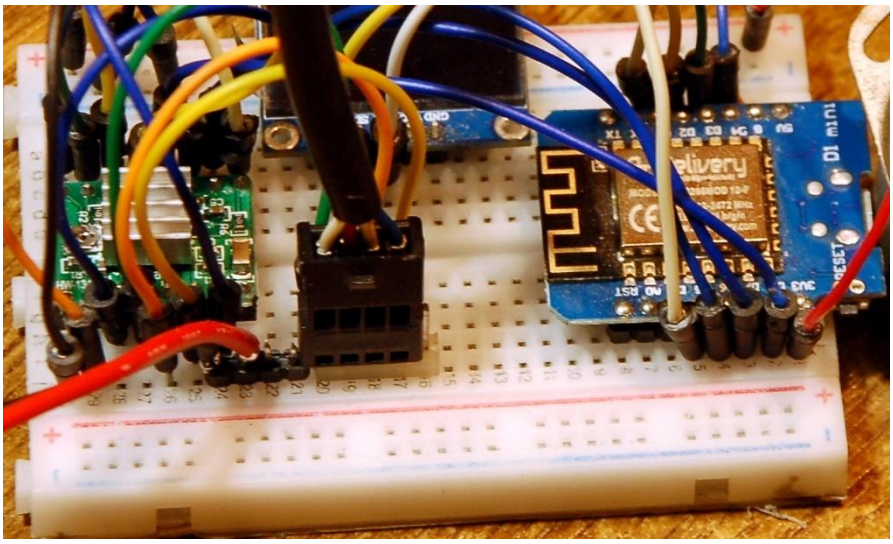
Abbildung 1:Ein A4988 am ESP8266 - Größenvergleich

Wir werden uns heute die beiden Treibermodule genau anschauen und ein MicroPython-Programm-Modul zu deren Ansteuerung bauen. Dabei kam es mir darauf an, alle wesentlichen Parameter, welche die kleinen Knirpse bieten, mit einzubeziehen. Außer den GPIOs 4, 5 und 16 sind somit alle Pins des ESP8266 D1 mini, den ich in diesem Anwendungsbeispiel als Controller verwende, belegt. Genauso gut kann natürlich auch der Amica aus dem letzten Post hergenommen werden. Das MicroPython-Modul arbeitet sowohl im Interrupt-Betrieb als auch manuell. Lassen Sie sich überraschen mit dieser neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Schicke Treiber für Schrittmotoren

Ich beginne mit einem kurzen Nachtrag zur vorangegangenen Folge. Dort hatte ich mit zwei Schrittmotoren aus einem alten Drucker die ersten Experimente am Motor-Shield unternommen. Einer der beiden war offensichtlich defekt und so entschloss ich mich, das Innenleben zu erforschen. Sie können in Abbildung 1 eindeutig die beiden Spulen eines bipolaren Motors identifizieren. Es handelt sich offensichtlich um einen Permanentmagnet-Schrittmotor. Der Rotor ist ein Ferritmagnet. Interessant ist die Erzeugung und Verteilung der Pole des Statormagneten. Nord- und Südpole wechseln sich durch geschickte Anordnung gegenseitig ab. Auf diese Weise können mit dem Motor 24 Schritte pro Umdrehung mit nur zwei Wicklungen erreicht werden. Die Schrittweite ist also 15°.

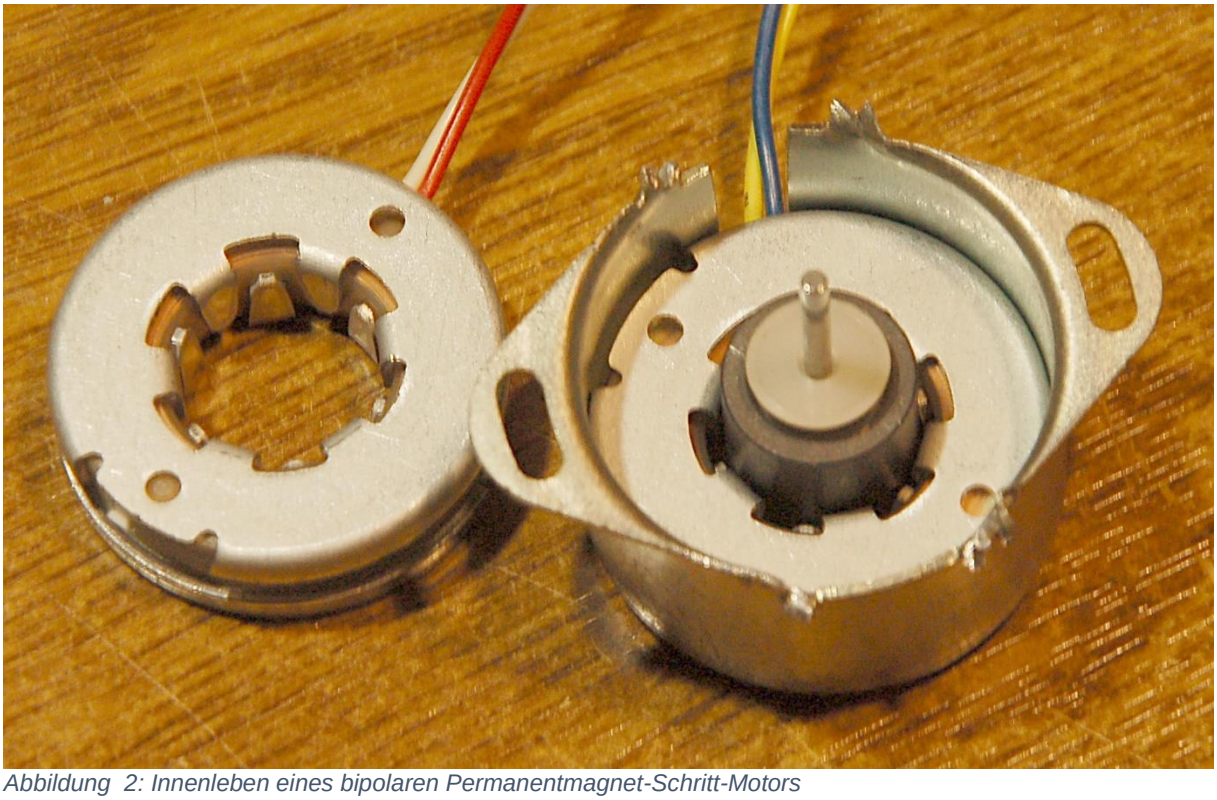

Abbildung 2: Innenleben eines bipolaren Permanentmagnet-Schritt-Motors

Nach dem kurzen Rückblick jetzt zum heutigen Programm. Untersuchen wir zunächst die beiden Module genauer.

Die Treibermodule

Beide Boards werden fertig verlötet mit zwei 8-Poligen Stiftreihen und einem kleinen Kühlkörperchen geliefert. Ich habe irgendwo gelesen, dass der Kühlkörper erst ab ca. 1,2A nötig ist. Bei den hier verwendeten Motoren liegt die Stromstärke gut unter einem halben Ampere, Man kann ihn also getrost weggelassen.

Eine Warnung aber vorweg: Die Chips sind gegen Überhitzung mit einem Schutz versehen, der den Strom zu den Motorwicklungen abschaltet, wenn eine Temperatur von 150°C (Datenblatt Allegro) erreicht wird. Das langt schon ein ganzes Ende vorher, um sich die Finger zu verbrennen.

Der A4988 von Allegro

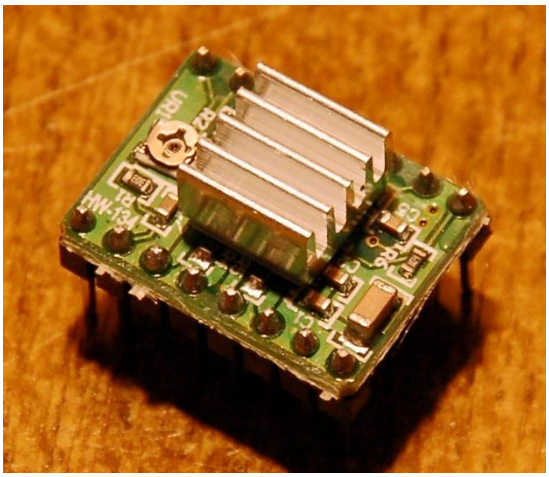

Abbildung 3:A4988-Treibermodul

Die Abmessungen fallen sehr klein aus, grade mal 15,3 x 20,4 mm. Der Abstand der Stiftleisten ist 5 Rastereinheiten. Damit passen die Boards problemlos auf ein Breadboard. Die Anschlussbelegung des A4988 ist wie folgt.

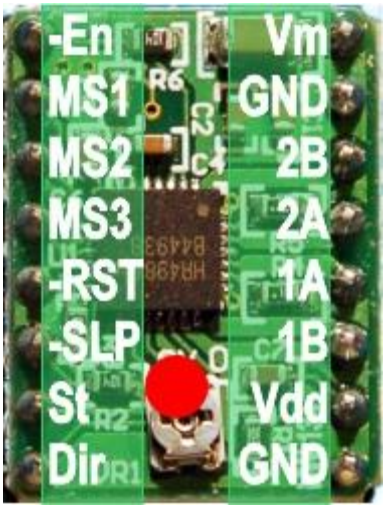

Abbildung 4: A4988 - Pinout

Der Chip benötigt zwei Betriebsspannungen, **Vm** für den Motor und **Vdd** für den Logikteil. **Vm** sollte im Bereich von 8,2 bis 35 V liegen und **Vdd** zwischen 3,3V und 5,5V. Die Stromstärke durch die Motorwicklungen darf bis zu 2 Ampere betragen.

Die uns bereits bekannten H-Brücken fallen sofort ins Auge, wenn man das Blockschaltbild des Treiberchips betrachtet. Die Anschlüsse 2B, 2A und 1B, 1A führen zu den Spulen eines bidirektionalen Schrittmotors. Unipolare Motoren kann das Board nicht ansteuern.

Auf dem Board befinden sich zwei Serienwiderstände **RS1** und **RS2** von je 0,1Ω in der Masse-Leitung zu den H-Brücken. Sie dienen der Stromstärkemessung für die Strombegrenzung. Die sorgt dafür, dass an **Vm** höhere Spannungen angelegt werden können, als die Berechnung $U = I_{max} \bullet R_{SP}$ ergibt. Wir kommen später darauf zurück.

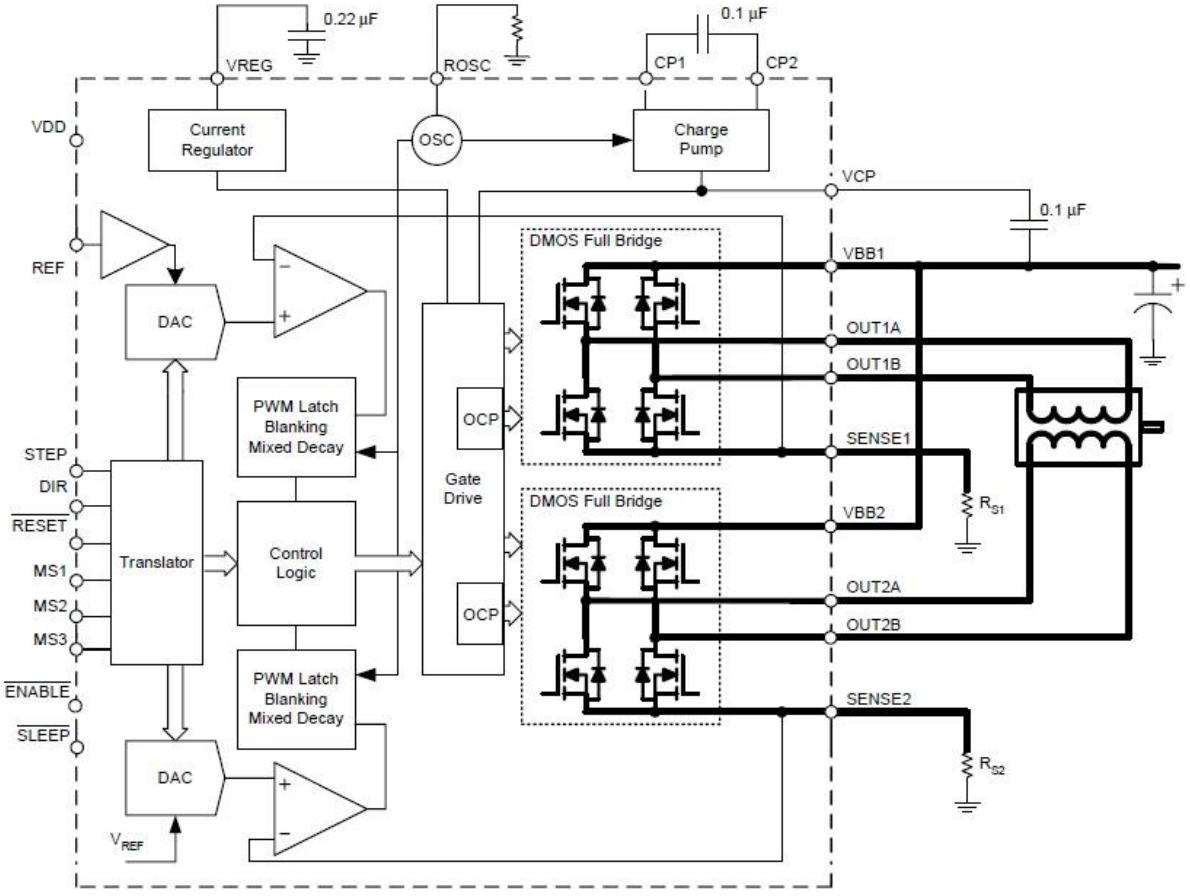

Abbildung 5: Blockschaltbild des A4988 (Datenblatt von Allegro)

Die Steuereingänge liegen alle auf der linken Seite des Boards. Damit die H-Brücken überhaupt aktiv werden können, muss der Eingang **-ENABLE** auf 0V gelegt werden. Diese Aufgabe übernimmt der ESP8266 an GPIO14 (D5). Liegt an diesem Eingang eine logische 1, dann werden die H-Brücken abgeschaltet, der Translator und die restliche Logik bleiben aktiv. Der Eingang wird mit einem Pulldown-Widerstand von 100kΩ auf logisch 0 gehalten, wenn er nicht extern beschaltet wird.

Der Eingang **-SLEEP** versetzt den Chip in den Schlafmodus. Es werden alle Schaltkreise abgeschaltet, inclusive der H-Brücken, wenn der Eingang auf GND-Potenzial gelegt wird. Intern ist er mit einem Pullup-Widerstand von 100kΩ auf Vcc gelegt.

Wir verbinden **-SLEEP** mit dem Eingang **-RESET**. Damit wirkt der Pullup an **-SLEEP** auch als Pullup für **-RESET**. Beide Eingänge werden wir nicht diskret ansteuern. Der Pullup hält den Chip insofern also in Arbeitsbereitschaft.

Der Pegel an **DIR** gibt die Drehrichtung vor. Ich habe den Motor so angeschlossen, dass eine 1 eine Drehung im Uhrzeigersinn und eine 0 im Gegenuhrzeigersinn ergibt. Die Richtung steuern wir über GPIO2 (D4).

Mit einer steigenden Flanke an **STEP** führt der Motor einen Schritt in der an **DIR** eingestellten Richtung aus. Die Pulse kommen GPIO0 (D3). Weil weder DIR noch STEP interne Pullups oder Pulldowns haben, können wir sie problemlos mit GPIO0 und GPIO2 verbinden, ohne das Startverhalten des ESP8266 zu stören oder zu verändern. Dieser Punkt ist für GPIO15 (D8) entscheidend, an dem die Modusleitung M3 angeschlossen ist und der beim Booten auf logisch 0 liegen muss.

Über die Pins M1, M2 und M3, die mit Pulldown-Widerständen versehen sind, kann man das **Microstepping** einstellen. Was ist das? Im vorangegangenen Blogpost haben wir schon das Halbschrittverfahren besprochen, bei dem der Rotor durch Überlagern, Ein- und Ausschalten der Motorspulen jeweils nur um einen halben Schrittwinkel weiterbewegt wird.

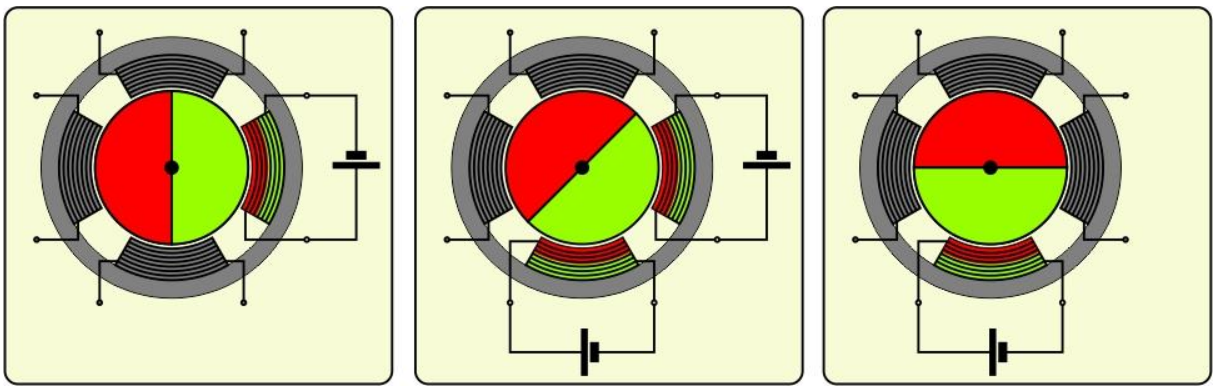

Abbildung 6: Halbschrittbetrieb (beim Unipolarmotor)

Stellt man den Strom durch die Spulen in Stufen auf Werte zwischen an und aus ein, dann kann man Schrittwinkel erreichen, die nur einem Teil des Vollwinkels entsprechen. Der gesamte Zyklus für den Vollschritt wird damit in mehrere Abschnitte, Phasen, aufgeteilt, für die sich jeweils die vollen 100% Kraftwirkung auf den Anker ergeben sollen, aber auch nicht mehr. Mit dem Motorshield hatten wir eigentlich im Zusammenspiel von zwei aktivierten Spulen ohne reduzierten Strom 141% erreicht, denn wenn jede Spule in Abbildung 6 mit 100% arbeitet, erhalten wir mit der Formel

$$I_{ges} = \sqrt{I_{S1}^{\ 2} + I_{S2}^{\ 2}}$$

Abbildung 7: Berechnung des Gesamtstroms

100%² + 100%² = 20000 und daraus die Wurzel, macht 141%. Der A4988 reduziert durch Pulsweitenmodulation (PWM) die Stromstärke im Halbschrittmodus auf 70%.

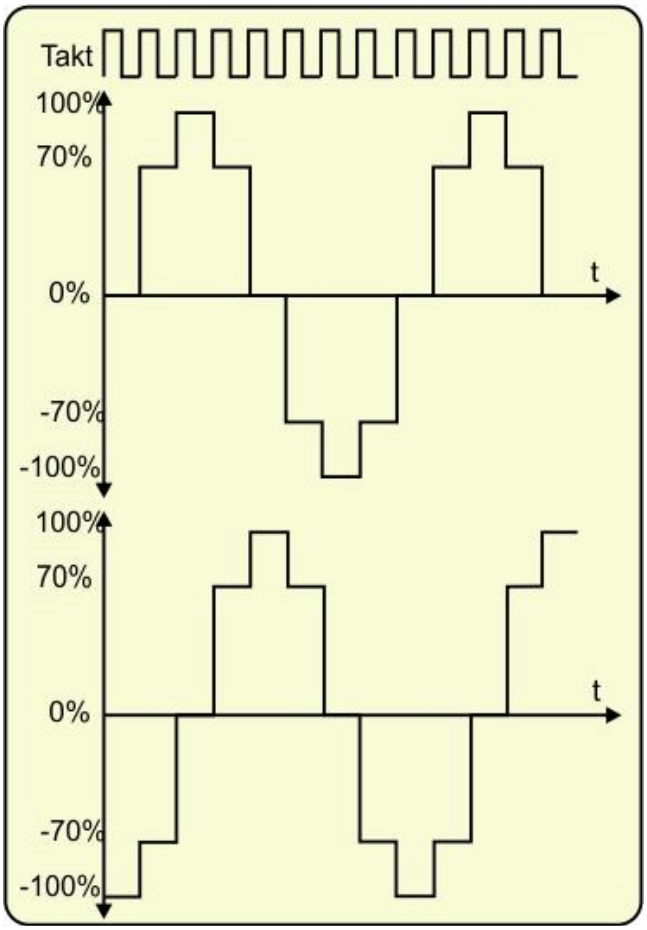

Abbildung 8: Halbschrittmodus über Spulenstromsteuerung

Aber wie kommt man auf die 70%? Ganz einfach, mit ein bisschen Mathematik.

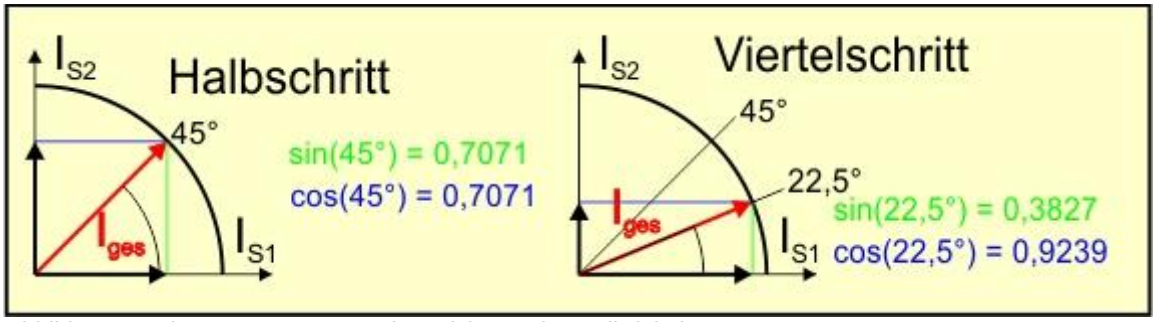

Abbildung 9: Die Prozentwerte ergeben sich aus den Teilwinkeln

Beim Halbschritt soll der Rotor genau zwischen den beiden Spulen stehen bleiben, also auf 45°. Die Sinus- und Cosinus-Funktion liefern beide den Wert 0,7071. Das sind die Längen der grünen und blauen Linie, wenn der rote Pfeil die Länge 1 haben soll. Auf die Prozentwerte kommen wir, wenn wir jeden Wert mit 100% multiplizieren. Genauso geht es beim Viertelschritt-Modus, wir erhalten die Schritt-Winkel 0°, 22,5° 45°, 67,5° und 90°. Sinus und Cosinus liefern die entsprechenden Werte für die Stromreduzierung.

Bei weiterer Verfeinerung nähert sich der Verlauf der Stromkurven den Graphen der Sinus- und Cosinus-Funktion an. Natürlich funktioniert das auch mit den Spulenpaaren bei Bipolar-Motoren.

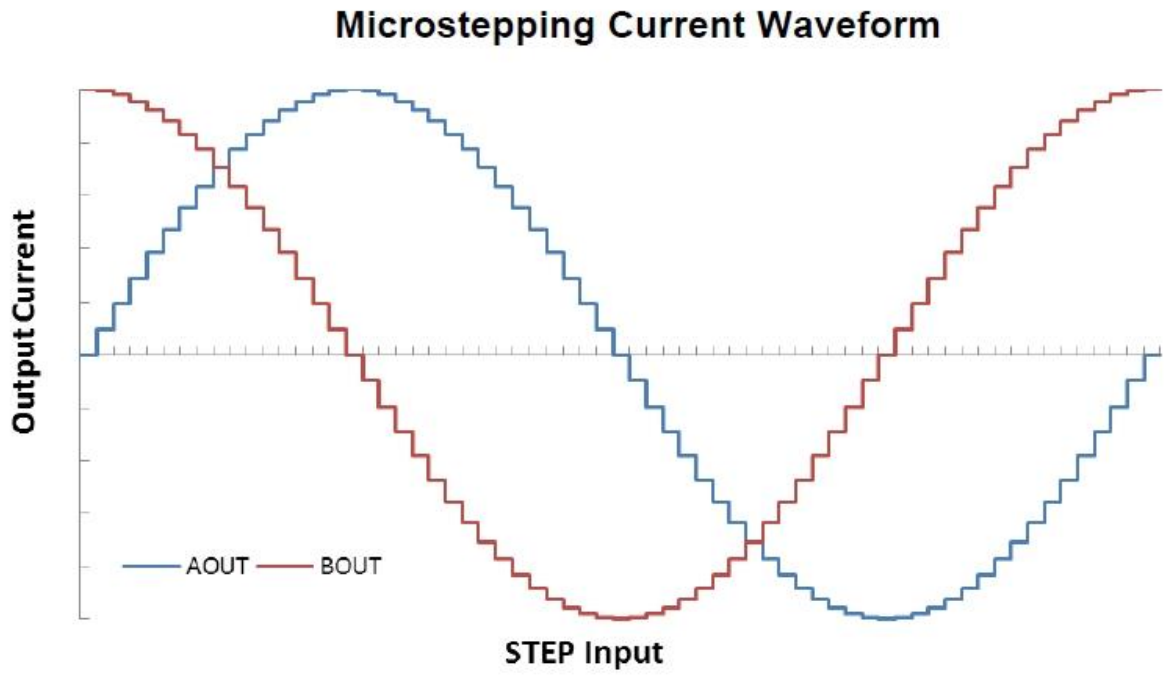

Abbildung 10: Microstepping mit dem DRV8825 (Datenblatt von TI)

Je nach Beschaltung der Modus-Eingänge M1, M2 und M3 am A4988, kann man Microstepping bis zu 16-tel-Schritten einstellen. Die Vorteile bei feinerer Schrittteilung sind in erster Linie natürlich die bessere Winkelauflösung und der ruhigere Lauf, weil Vibrationen beim Abbremsen vermieden werden. Beim Stoppen des Rotors nach einem größeren Schritt ergibt sich eine gedämpfte Schwingung, wenn er um die Stoppposition hin und her schwingt, bis er endlich zum Stillstand kommt.

Modus	M1	M2	M3
1/1	0	0	0
1/2	1	0	0
1/4	0	1	0
1/8	1	1	0
1/16	1	1	1

Abbildung 11: Modussteuerung beim A4988

Mit dem kleinen Trimmer kann man die Stromstärkebegrenzung durch die Motorwicklungen einstellen. Die maximale Stromstärke I_{max} berechnet sich aus der, am Abgriff (roter Pfeil) gemessenen Spannung V_{ref} nach der Formel

$I_{max} = V_{ref} / (8 \cdot R_s)$

R_S sind die Serien-Widerstände R4 und R5 von 0,1Ω in der Masseleitung der H-Brücken (rechts neben dem Chip), die als Stromstärkesensoren arbeiten.

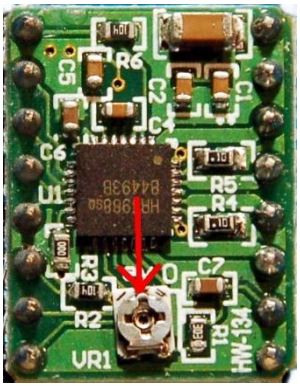

Abbildung 12: A4988 - Messpunkt

Die Stromstärke lässt sich natürlich auch direkt messen. Stöpseln sie in diesem Fall unbedingt zuerst die Motorspannung ab. Trennen Sie dann eine Strangzuführung auf und schließen Sie ein Amperemeter an. Nach dem Einschalten der Motorspannung, aktivieren Sie den Chip und lesen den Stromstärkewert ab. Mit dem Trimmer stellen sie die gewünschte Stromstärke ein. Vor dem Entfernen des Messgeräts auch wieder die Motorspannung abschalten!

Der DRV8825

Der DRV8825 kann Spannungen bis 45V verarbeiten. Für Strang-Stromstärken bis 1,5 A ist kein Kühlkörper nötig, mit Kühlkörper sind laut Datenblatt Stromstärken bis maximal 2,5A erlaubt. Durch die Strombegrenzung können Spannungen an die Wicklungen gelegt werden, die höher sind, als sie für den Motor angegeben oder berechnet sind. Eine höhere Motorspannung hat ein höheres Drehmoment beim Einschalten der Wicklung zur Folge. Die Strombegrenzung schützt die Wicklungen dennoch vor Überlastung.

Das Treibermodul mit dem DRV8825 von TI entspricht in Größe und Pinout dem A4988 bis auf eine Kleinigkeit. Die Steuereingänge und Spulenausgänge sind an der gleichen Position und haben dieselbe Funktion aber eine leicht abweichende interne Beschaltung. Der Pin -SLEEP hat beim A4988 einen Pullup, hier ist er mit 1MΩ auf GND gezogen. Der Hauptunterschied liegt aber in der Spannungsversorgung. Der DRV8825 benötigt nur eine Motorspannung, aus der er die niedrigere Spannung für die Logik selbst ableitet.

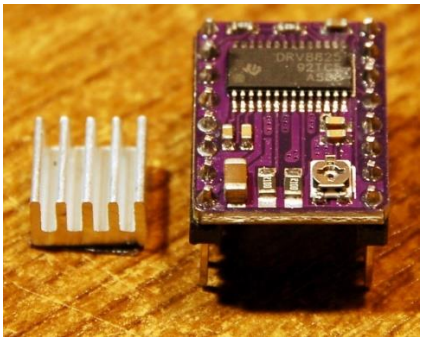
Abbildung 13: DRV8825-Modul

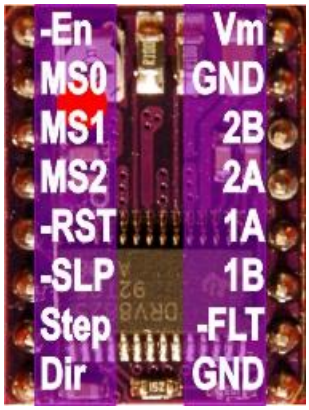

Abbildung 14: DRV8825 - Pinout

An dem Pin, an dem beim A4988 die Logikspannung zugeführt wird, liegt der Ausgang **-FAULT**. Der wird auf GND-Potenzial gezogen, wenn der DRV8825 eine Fehlfunktion feststellt wie Überstrom, Überhitzung, Kurzschluss oder zu niedrige Betriebsspannung. Er benötigt einen externen Pullup.

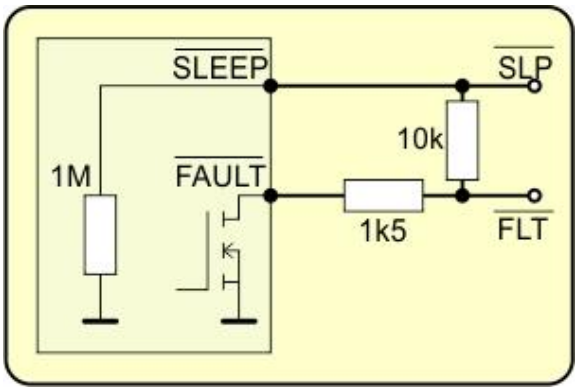

Abbildung 15: Beschaltung von FAULT und SLEEP

An **-FLT** liegt bei dem Board der **+Vdd**-Pin des A4988. Daher bekommt der Pin die 3,3V vom ESP8266 ab, wenn man den A4988 mit einem DRV8825 ersetzt. Der 10kΩ-Widerstand dient als Pullup am Pin **-SLP** und zieht auch den Eingang **-SLEEP** am DRV8825-Chip hoch. Solle dennoch **-FAULT** auf logisch 0 gehen, dann sichert der 1,5kΩ-Widerstand die Schaltung gegen Kurzschluss ab.

Die Strangstomstärke lässt sich auch beim DRV8825 mit dem Trimmer einstellen, allerdings mit einem anderen Wert für den Faktor A. Durch Messung des Spulenstroms (203mA), der Referenzspannung (209mV) und R_s = 0,1Ω komme ich nicht, wie im Datenblatt angegeben auf A = 5 sondern auf A = 10. Die Referenzspannung wird wieder am Abgriff des Trimmers gemessen (roter Pfeil).

$I_{max} = V_{ref} / (A \cdot R_s)$

Deshalb empfehle ich, den Spulenstrom direkt zu messen. Vorsicht! Immer die Motorspannung abschalten, bevor eine Motorwicklung abgetrennt wird.

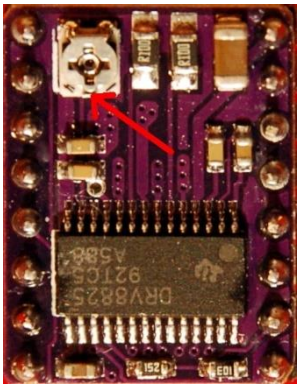

Abbildung 16: DRV8825 - Messpunkt

Dem Microstepping liegt eine etwas geänderte Tabelle zugrunde. Auch die Pins werden anders durchnummeriert. Der DRV8825 kann bis 32-stel Schrittweite programmiert werden.

Modus	M0	M1	M2
1/1	0	0	0
1/2	1	0	0
1/4	0	1	0
1/8	1	1	0
1/16	0	0	1
1/32	1	1	1

Abbildung 17: Modussteuerung beim DRV8825

Für beide Module geht man nach folgendem Impulsschema vor, um einen Schritt oder Microschritt einzuleiten.

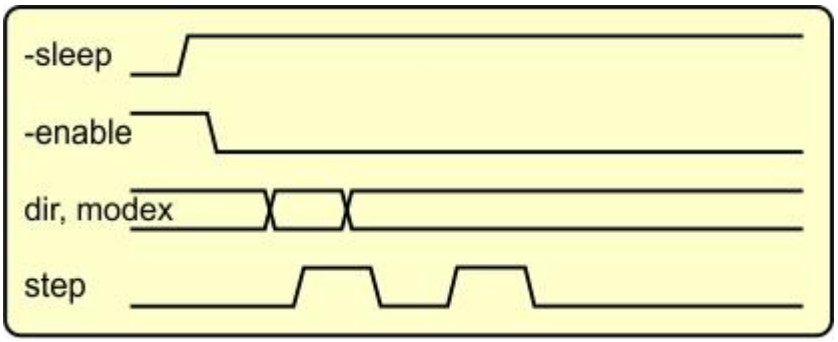

Abbildung 18: Impulsfolge am DRV8825 und A4988

-SLEEP liegt bei uns immer auf logisch 1, dafür sorgt der Pullup von 10kΩ. Nachdem **-ENABLE** auf logisch 0 gesetzt wurde, stellen wir die Richtung und den Microschrittmodus ein. Danach wird mit jeder positiven Flanke am Pin **STEP** ein Schritt ausgelöst.

Hardware

Nach der Theorie wird es jetzt wird handwerklich. Zum Aufbau der Schaltung sind folgende Teile nötig.

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit oder NodeMCU Lua Amica Modul V2 oder ESP8266 ESP-01S WLAN WiFi Modul oder **D1 Mini V3 NodeMCU mit ESP8266-12F**
1	A4988 Schrittmotor-Treiber-Modul mit Kühlkörper
	oder DRV8825 Schrittmotor-Treiber-Modul mit Kühlkörper
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
1	Schrittmotor uni- oder bipolar z.B. Pollin Best.-Nr. 310689 oder 310690
diverse	Jumperkabel
Optional	Logic Analyzer

Der Logic Analyzer ist sehr nützlich, wenn es um die Überwachung der Logikleitungen geht.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
packetsender zum Testen des ESP8266 als UDP-Client und -Server
SALEAE – Logic-Analyzer-Software (64 Bit) für Windows 8, 10, 11

Verwendete Firmware für einen ESP32:

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

Verwendete Firmware für einen ESP8266:

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

Die MicroPython-Programme zum Projekt:

stepper.py MicroPython-Modul

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.

Das MicroPython-Modul

In dem Modul konnte ich die Klassen für den **DRV8825** und den **A4988** vereinen, weil die beiden Chips fast gleich anzusteuern sind. Wie wir schon wissen gibt es lediglich verschiedene Modus-Tabellen. Das wirkt sich letztlich auch auf die Methode **setMode**() aus, welche die Modus-Pins setzt. Das Problem habe ich aber elegant gelöst, indem ich einfach die Klasse **DRV8825** von der Klasse **A4988** erben lasse und sowohl die Tabelle **ModeTable** und die Methode **setMode**() einfach nur überschreibe.

```
# stepper.py
# Treiber fuer bidirektionale Motoren mit A4988 / DRV8825
# am ESP8266
#LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8 RX TX
#ESP8266 Pins 16  5  4  0  2 14 12 13 15  3  1
#beim Start   hi sc sd hi hi          lo hi hi
#Pinbelegung           ST DI EN M1 M2 M3
#
# dir,step,enable=2,0,14 (D4,D3,D5)
# Sleep + RST liegen an Vcc
#
# M1,M2,M3 = 12,13,15 (D6,D7,D8)
from machine import Pin, Timer
from time import sleep_ms,sleep_us
```

Vom Modul **machine** brauchen wir **Pin** und **Timer** und von **time sleep_ms** und **sleep_us** für kleine Schlafpausen.

Ich erkläre einige Klassen für die Fehlerbehandlung, die ich von der Basisklasse Exception ableite.

```
class A4988_Error(Exception):
    pass

class MicroStepping_Error(A4988_Error):
    def __init__(self):
        super().__init__("Wrong microstepping!",
                         "Use 1, 2, 4, 8 or 16")

class Direction_Error(A4988_Error):
    def __init__(self):
        super().__init__("Wrong direction value!",
                         "Use -1, 0 or 1")
```

Mit der Klassendeklaration von A4988 definiere ich auch gleich das Klassenattribut **ModeTable**. Die Schlüssel im Dictionary sind die Schrittteiler und in Dreier-Tupeln folgen die logischen Werte für die Modus-Pins.

```
class A4988(A4988_Error):
    ModeTable={
        1:(0,0,0),
```

```
        2:(1,0,0),
        4:(0,1,0),
        8:(1,1,0),
        16:(1,1,1),
        }
```

Der Konstruktor ist die umfangreichste Methode der Klasse. Das liegt an den zahlreichen Attributen, der zuerstellenden Objekte. Als Parameter übergebe ich die Pinnummern der GPIOs sowie die Anzahl Schritte für eine Umdrehung und erzeuge dann die Pin-Objekte. Die Pins für die Modussteuerung setze ich in eine Liste, damit ich sie später in einer for-Schleife über den Index referenzieren kann.

```
    def __init__(self,
                 Dir=2,
                 Step=0,
                 Enable=14,
                 M0=12,
                 M1=13,
                 M2=15,
                 StepsPerRev=24,):
        self.dir=Pin(Dir,Pin.OUT, value=0)
        self.step=Pin(Step,Pin.OUT, value=0)
        self.en=Pin(Enable,Pin.OUT, value=1)
        self.modePin=[
                       Pin(M0,Pin.OUT,value=0),
                       Pin(M1,Pin.OUT,value=0),
                       Pin(M2,Pin.OUT,value=0),
                      ]
        self.direction=0 # Stillstand
        self.freq=10000 # Timerfrequenz
        self.delay=1000 # Modul fuer Frequenzteilung
        self.velocity=self.freq//self.delay # Geschwindigkeit
        self.count=0 # Wiederholungszaehler in der ISR
        self.aPos=0 # aktuelle Position
        self.tPos=0 # Zielposition
        self.timer=Timer(0) # Timer fuer IRQ
        self.timerActive=False # Flag f. Timer an / aus
        self.stepsPerRev=StepsPerRev # Schritte pro Runde
        self.stepping=1 # Microschrittmodus
        self.degPerStep=360/self.stepsPerRev/self.stepping
        self.setMode(self.stepping) # Schrittmodus anmelden
        self.timeJob=self.ISR # Timer ISR festlegen
        self.freilauf=False # Freilauf ausgeschaltet
        self.setFreerun(self.freilauf)# Modus setzen
```

Zum Aktivieren der H-Brücken rufen wir die Methode **enable**() auf. Der Parameter **e** wird auf den Defaultwert None gesetzt. Wird beim Aufruf kein Argument übergeben, dann wird der Zustand zurückgegeben. Das macht die Methode **enabled**(), die als Property maskiert ist. Dadurch ist sie wie ein Attribut referenzierbar.

```
    @property
    def enabled(self):
        return (True if self.en.value()==0 else False)
```

```
    def enable(self, e=None): #  e: True | False
        if e is None:
            return self.enabled
        if type(e) != "bool":
            e=bool(e)
        if e:
            self.en.value(0)
            sleep_ms(2)
        else:
            self.en.value(1)
            self.timerStop()
        print(self.enabled)
```

An **e** kann 0 oder 1 aber auch True und False übergeben werden. Für eine einheitliche Behandlung wandeln wir numerische Werte in boolsche um. Ist **e** jetzt wahr, dann legen wir den Ausgang **en** auf logisch 0 und warten kurz bis der A4988 bereit ist. Andernfalls legen wir **en** auf 1 und stoppen vorsichtshalber auch den IRQ-Timer. Abschließend lassen wir uns den Zustand ausgeben.

Wenn wir nur den Timer anhalten wollen und die Ausgangsstufen sollen aktiv bleiben, damit der Motor die Position hält, können wir **timerStop**() aufrufen. Der wird Timer gestoppt und der Zustand in **timerAktive** gemerkt.

```
    def timerStop(self):
        self.timer.deinit()
        self.timerActive=False
```

Beim Starten des Timers mit **start**() initialisieren wir den Timer für periodische Wiederholung des Interrupts, mit der Frequenz 10000Hz und setzen die Callback-Routine, also die ISR, auf die Referenz in **timeJob**. Weil wir dafür ein Attribut verwenden, ist es möglich, eine selbstgeschriebene Serviceroutine zu verfassen und dem Attribut **timeJob** zu zuweisen. Das geschieht dann über die Methode **setCallback**(), der wir die Referenz auf unsere eigene ISR im Parameter **funk** übergeben.

```
    def start(self):
        self.timer.init(mode=Timer.PERIODIC,
                        freq=10000,
                        callback=self.timeJob)
        self.timerActive=True
        self.enable(1)
```

```
    def setCallback(self, funk):
        self.timeJob=funk
```

Mit **mode**() rufen wir den aktuellen Microstepping-Modus ab, den wir mit **setMode**() einstellen können. Auch ein Aufruf von **setMode**() ohne Argument liefert den Modus zurück.

```
    @property
    def mode(self):
        return self.stepping
```

```
    def setMode(self, res=None):
        if res is None:
            return self.modus
        else:
            if res not in [1,2,4,8,16]:
                raise MicroStepping_Error
            else:
                self.stepping = res
                for i in range(3):
                    self.modePin[i].\
                         value(A4988.ModeTable[res][i])
                self.degPerStep=\
                         360/self.stepsPerRev/self.stepping
                print("{} Grad / Schritt".\
                      format(self.degPerStep))
```

Ist der Wert von **res** kein Element der Liste [1,2,4,8,16], dann werfen wir eine **MicroStepping_Error**-Exception. Sonst übergeben wir den Wert an **stepping** und weisen in der for-Schleife den Pins der Liste **modePin** den entsprechenden Wert des Tupel-Elements aus dem Dictionary **ModeTable** zu. Außerdem muss der Schrittwinkel neu berechnet werden. Wir teilen den Vollwinkel durch die Anzahl Schritte pro Umdrehung und dann noch durch die Anzahl an Microschritten. Das Ergebnis lassen wir uns in REPL anzeigen.

resetPos() setzt die Positionszeiger auf 0.

```
    def resetPos(self):
        self.aPos,self.tPos = 0,0
```

Die Methode **richtung**() liefert den Wert für die aktuelle Drehrichtung zurück, die mit **setDir**() gesetzt werden kann.

```
    @property
    def richtung(self):
            return self.direction
```

```
    def setDir(self, r=None): # r=-1|0|1
        if r is None:
            return self.richtung
        else:
            if r not in [-1,0,1]:
                raise Direction_Error
            else:
                self.direction = r
                self.dir.value(r if r== 1 else 0)
```

Ist der Wert von r nicht in der Liste [-1,0,1] enthalten, werfen wir eine **Direction_Error**-Exception. Sonst merken wir uns den Wert im Attribut **direction** und setzen den Steuerausgang **dir** entsprechend auf 1 oder 0.

Die Ausführung eines Schritts ist der Job von **schritt**(), wenn **direction** nicht gleich 0 ist. Wir geben einen kurzen Puls an **step** aus und sorgen dafür, dass die Buchführung stimmt. Zur aktuellen Position wird 1 oder -1 addiert. Es wird davon ausgegangen, dass sich der Schrittmodus während des Programmlaufs nicht ändert, sonst müsste der Modus mit einbezogen werden. Das könnte so erfolgen, dass in der kleinsten Microschrittweite gezählt wird, 16 für einen Vollschritt, 8 für den Halbschritt … und 1 für den 16-tel-Schritt.

```
    def schritt(self):
        if self.direction != 0:
            self.step.value(1)
            self.step.value(0)
            self.aPos += self.direction
```

Die Interrupt-Service-Routine **ISR**() des Timers erledigt die automatische Ausführung von Schritten im Freilaufmodus oder beim Anfahren der Zielposition in **tPos**, wenn **direction** nicht 0 ist.

```
    def ISR(self,t):
        if self.direction != 0:
            self.count = (self.count + 1) % self.delay
            if self.count==0:
                if self.freilauf:
                    self.schritt()
                elif(self.direction == 1 and \
                    (self.aPos < self.tPos)) or \
                    (self.direction == -1 and \
                    (self.aPos > self.tPos)):
                     self.schritt()
```

Wenn der Timer durch **start**() aktiviert wurde, wird **ISR**() im Abstand von 100µs (f=10000Hz) ISR() aufgerufen. In t wird die Nummer des Timers übergeben. Der Parameter ist obligatorisch, muss also angegeben werden. Würden mehrere Timer auf dieselbe ISR zugreifen, ließe sich über t feststellen, welcher Timer gefeuert hat.

Wenn **direction** auf 1 oder -1 steht, erhöhen wir den Zähler **count**. Er zählt also die 100µs-Intervalle. Ist der Teilungsrest des Zählers durch den Wert in **delay** gleich 0, dann wurde eine durch **delay** festgelegte Zeitdauer erreicht und es wird nachfolgend geprüft, ob ein Schritt durchzuführen ist.

Wenn **freilauf** True ist, wird jetzt stets ein Schritt ausgeführt. Sonst prüfen wir ob im Fall einer Drehung im Uhrzeigersinn die aktuelle Position kleiner ist als die Zielposition oder ob im Gegenuhrzeigersinn die aktuelle Position größer als die Zielposition ist. Trifft eines von beiden zu, wird ein Schritt ausgeführt.

Eine bestimmte Anzahl von Schritten wird durch den Aufruf von **schritte**() ausgeführt. In **n** wird die Anzahl übergeben. Dabei ist die Drehrichtung durch das Vorzeichen festgelegt. Es passiert nichts, wenn n = 0 ist. Sonst wird die Drehrichtung gesetzt, und die for-Schleife führt die Schritte aus, gefolgt von einer Pause mit **delay** • 100µs, wie in **ISR**(). Die Routine wirkt blockierend, das heißt, dass während der Ausführung kein anderer Befehl ausgeführt werden kann. Die Ausführung von Schritten durch die ISR ist nicht blockierend, weil zwischen den Interrupt-Aufrufen andere Aktivitäten möglich sind.

```
    def schritte(self,n): # n != 0
        if n == 0:
            return
        self.setDir(1 if n>0 else -1)
        for i in range(abs(n)):
            self.schritt()
            sleep_us(self.delay*100)
```

Die aktuelle Position wird durch **position**() abgerufen.

```
    @property
    def position(self):
        return self.aPos
```

Die Methode **ziel**() setzt die in **pos** übergebene absolute Zielposition, wenn sie nicht der aktuellen Position entspricht. Die Drehrichtung wird aus der aktuellen Position in **aPos** und dem Wert in **pos** ermittelt. Nachdem sichergestellt ist, dass der Timer läuft, setzen wir die Zielposition durch Zuweisung an **tPos**, und die Timer-ISR sorgt für das Ausführen der Schritte. Rufen wir **ziel**() ohne Argument auf, wird die aktuelle Zielposition zurückgegeben.

```
    def ziel(self,pos=None): # absolute Position
        if pos is None:
            return self.tPos
        if pos == self.aPos:
            return
        self.setDir(1 if pos > self.aPos else -1)
        if not self.timerActive:
            self.start()
        self.tPos = pos
```

Die Methode **relativ**() geht die in **dist** übergebene Anzahl von Schritten nichtblockierend. **dist** darf positiv oder negativ sein, 0 wird ausgesondert. Die Drehrichtung wird entsprechend dem Vorzeichen gesetzt und die Zielposition berechnet, die dann an **ziel**() übergeben wird.

```
    def relativ(self,dist): # +/- Schritte
        if dist == 0:
            return
        else:
            self.setDir(1 if dist > 0 else -1)
            pos = self.aPos + dist
            self.ziel(pos)
```

Soll der Motor um einen bestimmten Winkel (positiv oder negativ) drehen, dann ist **winkel**() die richtige Wahl. Wir ermitteln die Anzahl von Schritten, indem wir den Wert in **w** durch den Schrittwinkel teilen und den Quotienten ganzzahlig machen. Wenn **schritte** nicht 0 ist, übergeben wir den Wert an **relativ**().

```
    def winkel(self,w):
        schritte=int(w/self.degPerStep)
        if abs(schritte)>=1:
            self.relativ(schritte)
```

Ob der Freilaufmodus aktiv ist, fragen wir mit **freerun**() ab. Mit **setFreerun**() schalten wir den Modus ein (**state** = 1 oder True) oder aus (**state** = 0 oder False). Wenn **state** True ist und der Timer nicht aktiv, wird er gestartet. Mit der Wertzuweisung an **freilauf** startet oder stoppt der Motor. Der Timermodus bleibt unverändert, wenn **state** False ist.

```
    @property
    def freerun(self):
        return self.freilauf
```

```
    def setFreerun(self, state=None): # state=0|1
        if state is None:
            return self.freerun
        if type(state) != "bool":
            state=bool(state)
        if state and (not self.timerActive):
            self.start()
        self.freilauf=state
```

Mit **speed** fragen wir die Geschwindigkeit ab, die mit **setSpeed**() gesetzt wird. Wird **setSpeed**() ohne Argument aufgerufen, bekommen wir den auch aktuellen Wert zurück.

```
    @property
    def speed(self):
        return self.velocity
```

```
    def setSpeed(self,s=None):
        if s is None:
            return self.speed
        else:
            self.delay = self.freq // s
            self.velocity = s
            print(self.delay/10,"ms/step", \
                  1000/(self.delay/10), "st/sec")
```

An **delay** weisen wir die Verzögerung in 100µs- Einheiten zu. Das ist der ganzzahlige Anteil des Quotienten aus der Timerfrequenz **freq** und der Schrittanzahl pro Sekunde. Nach dieser Zeit ist ein Schritt zu machen. Den Wert merken wir uns in **velocity**. Im Parameter **s** übergeben wir die Anzahl Schritte pro Sekunde.

Statt für den DRV8825 eine eigene ganze Klasse zu deklarieren, lasse ich die Klasse DRV8825 von der Klasse A4988 einfach erben. Mit diesem Schritt sind alle Attribute und Methoden aus A4988 auch im Namensraum der Klasse DRV8825 verfügbar.

```
class DRV8825(A4988):
    ModeTable={
        1:(0,0,0),
        2:(1,0,0),
        4:(0,1,0),
        8:(1,1,0),
        16:(0,0,1),
        32:(1,1,1),
        }
```

Natürlich braucht die neue Klasse wegen der abweichenden Schrittmodus-Werte eine eigene **ModeTable**. Die Werte entnehmen wir dem Datenblatt. Durch die Definition der Liste unter demselben Namen wird die Definition in A4988 überschrieben. Dasselbe machen wir gleich mit der Methode **setMode**().

Zuvor müssen wir der Klasse aber einen eigenen Konstruktor verpassen. Die zuübergebenden Parameter sind die gleichen wie für A4988(). Die Werte, die wir bekommen, reichen wir mit dem Aufruf von A4988.__init__() über die Funktion **super**() weiter.

```
    def __init__(self,
                 Dir=2,
                 Step=0,
                 Enable=14,
                 StepsPerRev=24,
                 M0=12,
                 M1=13,
                 M2=15):
        super().__init__(
```

```
                  Dir,
                  Step,
                  Enable,
                  StepsPerRev,
                  M0,
                  M1,
                  M2)
```

In der Methode **setMode**() ändern wir nur die **Vergleichsliste** und Referenz auf **ModeTable**.

```
    def setMode(self, res=None):
        if res is None:
            return self.stepping
        else:
            if res not in [1,2,4,8,16,32]:
                raise MicroStepping_Error
            else:
                self.stepping = res
                for i in range(3):

self.modePin[i].value(DRV8825.ModeTable[res][i])
```

Schließlich erzeugen wir noch ein Motor-Objekt **m**, wenn **stepper**.py im Editorfenster gestartet wird, und können dann mit den ersten manuellen Tests beginnen. Dieser Teil wird nicht ausgeführt, wenn wir **stepper**.py als Modul mit **import** einbinden.

```
if __name__ == "__main__":
    m=DRV8825()
#     m=A4988()
```

Die Schaltung und die ersten Tests

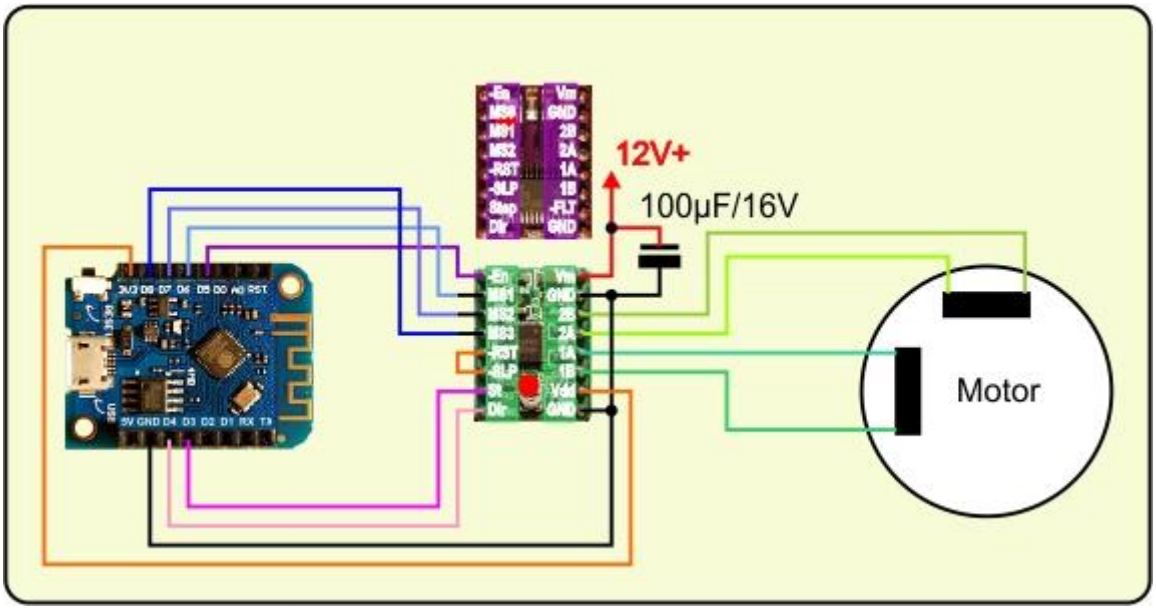

Abbildung 19: Schaltung der Treibermodule A4988 und DRV8825

Der Aufbau ist überschaubar. Im Datenblatt wird ein Elektrolytkondensator über der Motorspannung empfohlen um die Boards vor Spannungsspitzen zu schützen.

Wir starten das Programm **stepper.py** im Editorfenster. Unser Motor-Objekt heißt m**.**

H-Brücken aktivieren:

>>> m.enable(1)
True

Der Rotor darf sich jetzt nicht mehr frei von Hand durchdrehen lassen.

Rechtslauf und Vollschritt:
>>> m.setDir(1)
>>> m.setMode(1)

Einen Schritt auslösen:
>>> m.schritt()

Linkslauf:
>>> m.setDir(-1)

Einen Schritt auslösen:
>>> m.schritt()

Eine Umdrehung ausführen (Motor mit 24 steps / rev):
>>> m.schritte(24)

Positionen auf 0 setzen:

>>> m.resetPos()

48 Schritte vorwärts:
m.relativ(48)

Aktuelle Position abfragen:
>>> m.position
48

Den Timer gestartet?
>>> m.isActive
True

Absolute Zielposition 96 anfahren und überprüfen:
>>> m.ziel(96)
>>> m.position
96

Einen Winkel von 90° zurück:
>>> m.winkel(-90)
>>> m.position
90

Freilauf:
>>> m.setFreerun(1)

Richtung ändern:
>>> m.setDir(1)

Timer anhalten:
>>> m.timerStop()

H-Brücken ausschalten:
m.enable(0)
False

Dieses Modul werde ich im nächsten Post für mein optisches Radar einsetzen. In einem Umkreis von bis zu zwei Metern kann man mit einem VL53L0X Gegenstände orten. Die Darstellung erfolgt auf einem OLED-Display.

Bis dann, bleiben sie dran!