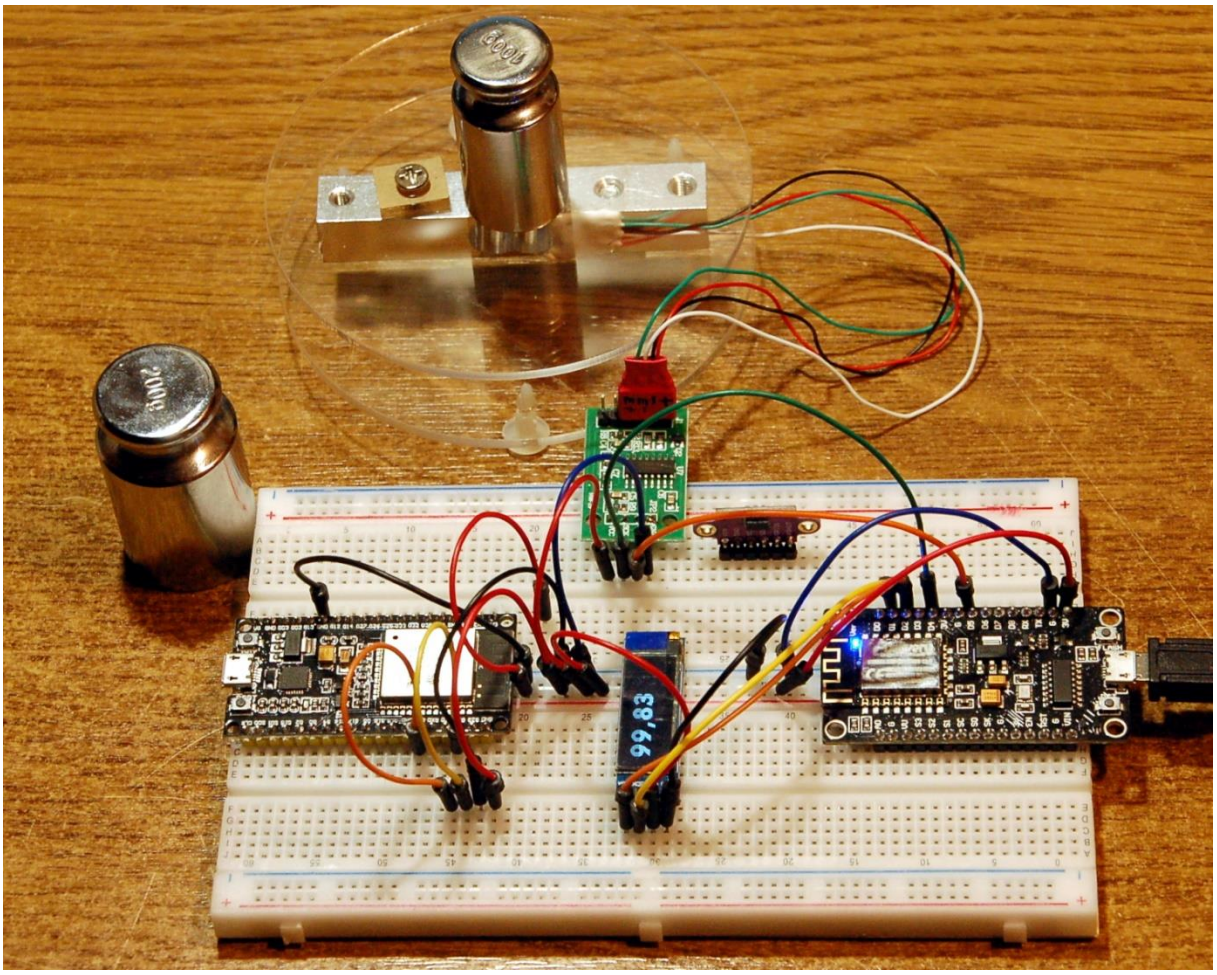

Kalibrieren am ESP8266

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

Eine Waage ohne bewegliche Teile – geht nicht – sagen Sie? Geht doch sagt das Ergebnis meines aktuellen Projekts und zwar mit einer sagenhaften Auflösung und Genauigkeit. Aus, mit dem Auge nicht wahrnehmbaren Verbiegungen eines Aluminiumquaders, einem 24-Bit-ADC (Analog-Digital-Wandler), einem ESP8266 oder ESP32 (abkürzend im folgenden Text ESP) und einem OLED-Display wird eine Digitalwaage, die in meinem Fall Massen bis 1kg erfassen kann, mit einer Auflösung von 0,01g! Wie das geht und welche Tricks dahinterstecken, das verrät Ihnen dieser Beitrag aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Digitalwaage mit dem HX711

Ich habe es ursprünglich auch nicht für möglich gehalten und war vom Resultat absolut überrascht. Der eingesetzte Aluquader ist eine sogenannte Wägezelle. Zwei Bohrungen in der Mitte dünnen das Metall aus, und an der Wandung sind

Dehnungsmessstreifen aufgeklebt, in Abbildung1 oben und unten. Die Materialstärke beträgt dort ca. 1mm.

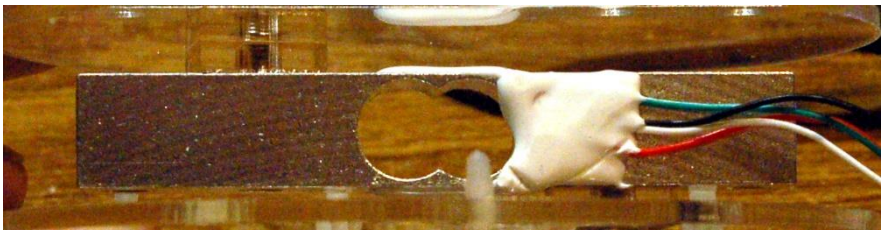

Abbildung 1: Wägezelle von der Seite

Die Wägezelle ist rechts mit der Bodenplatte verschraubt, links ist die Trägerplatte der Waage angebracht.

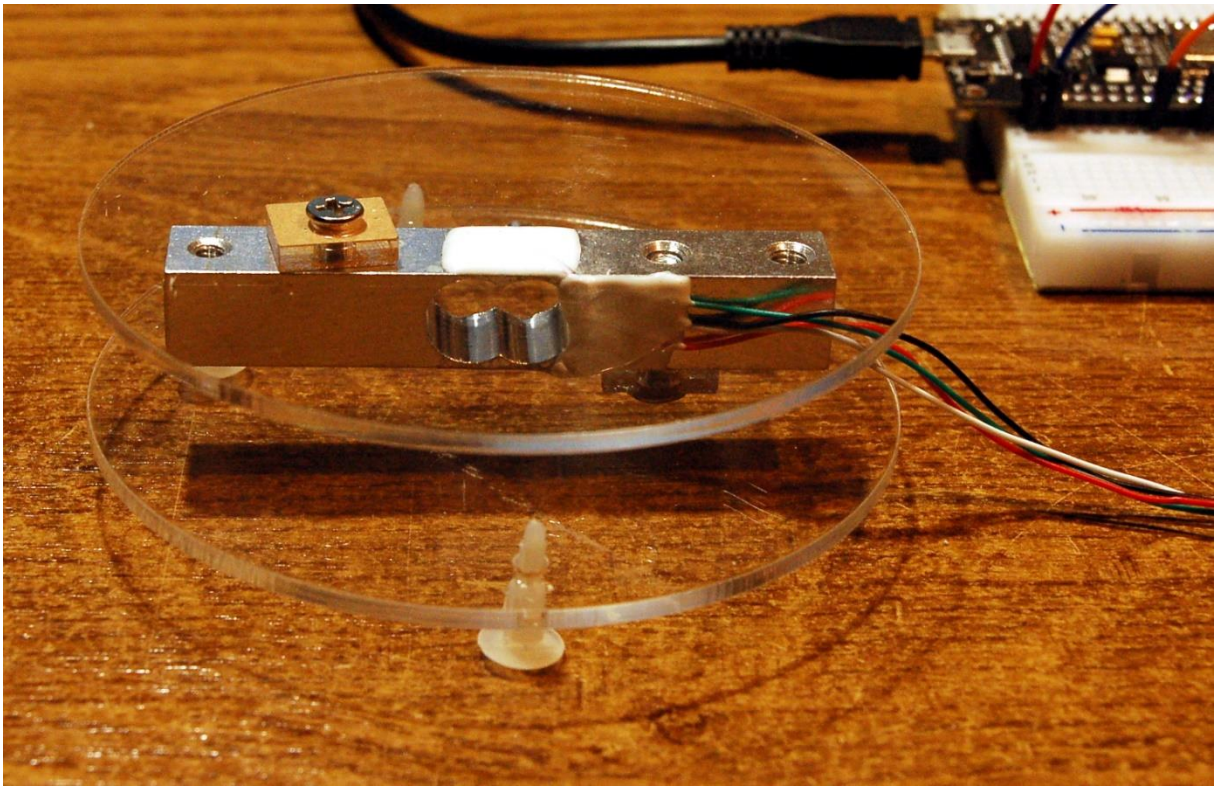

Abbildung 2: Waage für 1kg

Ein Blatt Papier der Größe 10cm x 15 cm verbiegt die Wägezelle nun immerhin so viel, oder eher wenig, dass die Waage das Gewicht von 0,9g anzeigt.

Wie funktioniert das? Dehnungsmesstreifen sind hauchdünne Leiterbahnen, die auf einen Kunststoffträger aufgebracht sind. Diese Pads werden auf ein Trägermaterial aufgeklebt.

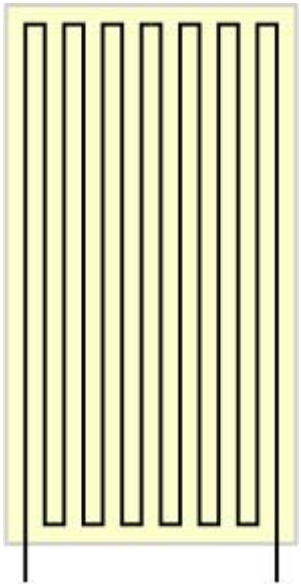

Abbildung 3: Dehnungsmessstreifen schematisch

Durch Verbiegen des Trägers werden die Leiter geringfügig gedehnt und damit dünner und länger. Das bewirkt eine Änderung des elektrischen Widerstands, der bekanntlich von den beiden Parametern abhängt, ρ ist der spezifische Widerstand, eine Materialkonstante.

$$R = \rho \bullet \frac{l}{A}$$

Abbildung 4: Widerstandsformel

Das aufgelegte Papier dürfte höchstens eine Verbiegung des 12,5mm hohen Quaders auslösen, die in der Größenordnung eines Atoms liegt. Die daraus resultierende Längenänderung des Messstreifens ebenfalls. Und das reicht aus, um die Spannung an der Messbrücke mit den vier Messstreifen, zwei oberhalb, zwei unterhalb des Quaders, so weit zu verändern, dass der HX711 daraus eine messbare und vor allem reproduzierbare Spannungsänderung ableiten kann.

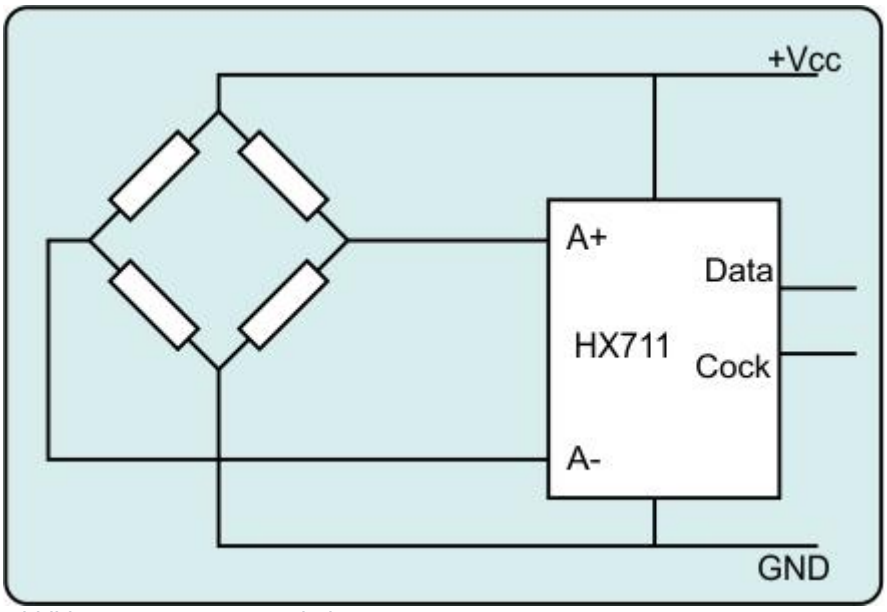

Abbildung 5: HX711 - Beschaltung

Hardware

Als Controller eignen sich sowohl beliebige ESP8266- wie auch ESP32-Modelle mit mindestens vier freien GPIOs. Die Anzeige des Messwerts erfolgt über ein OLED-Display. Als Wägezelle kann natürlich auch ein anderes Modell dienen. Es gibt sie ab 100g aufwärts bis zu 100kg Wägevolumen und mehr.

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder D1 Mini V3 NodeMCU mit ESP8266-12F oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI oder ESP32 Dev Kit C unverlötet oder ESP32 Dev Kit C V4 unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder NodeMCU-ESP-32S-Kit oder ESP32 Lolin LOLIN32 WiFi Bluetooth Dev Kit
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Wägezelle 1kg
1	HX711 AD-Wandler für Wägezellen
1	MB-102 Breadboard Steckbrett mit 830 Kontakten
diverse	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F evtl. auch 65Stk. Jumper Wire Kabel Steckbrücken für Breadboard
optional	Logic Analyzer

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

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

Verwendete Firmware für den ESP8266:

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

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
geometer_30.py großer Zeichensatz für die Ziffernanzeige
hx711.py API für den AX711
scale.py Das Betriebsprogramm
zeichensatz.rar Arbeitsumgebung zum Erzeugen eigener Zeichensätze

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 18.06.2022) 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.

Die Schaltung

Hier die Schaltbilder für das Projekt, es funktioniert wahlfrei für ESP32 und ESP8266. Die GPIOs für den Anschluss des HX711 sind so gewählt, dass sie den Start des ESP8266 nicht behindern und für beide Controllertypen dieselben Bezeichnungen haben. Lediglich die Anschlüsse für den I2C-Bus sind unterschiedlich, werden aber vom Programm automatisch zugeordnet.

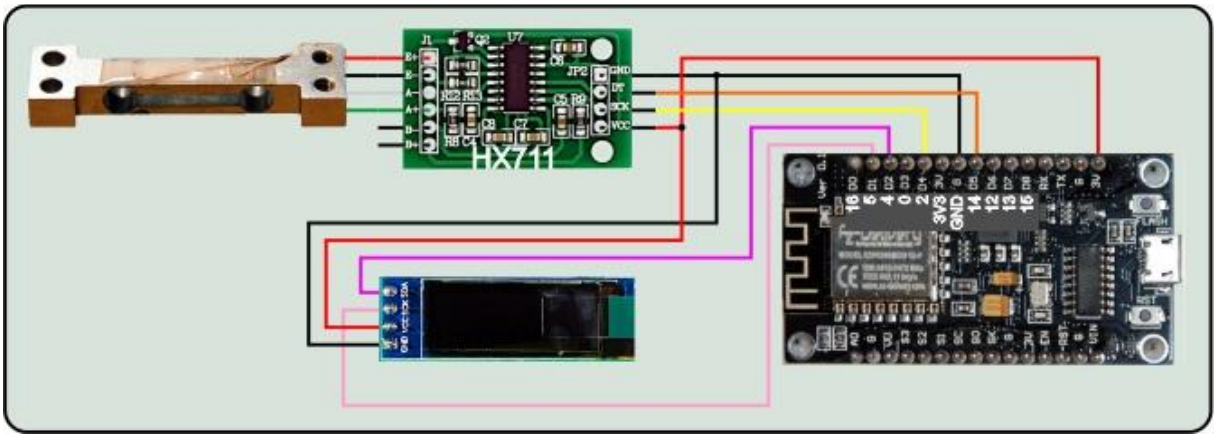

Abbildung 6: hx711-Waage am ESP8266

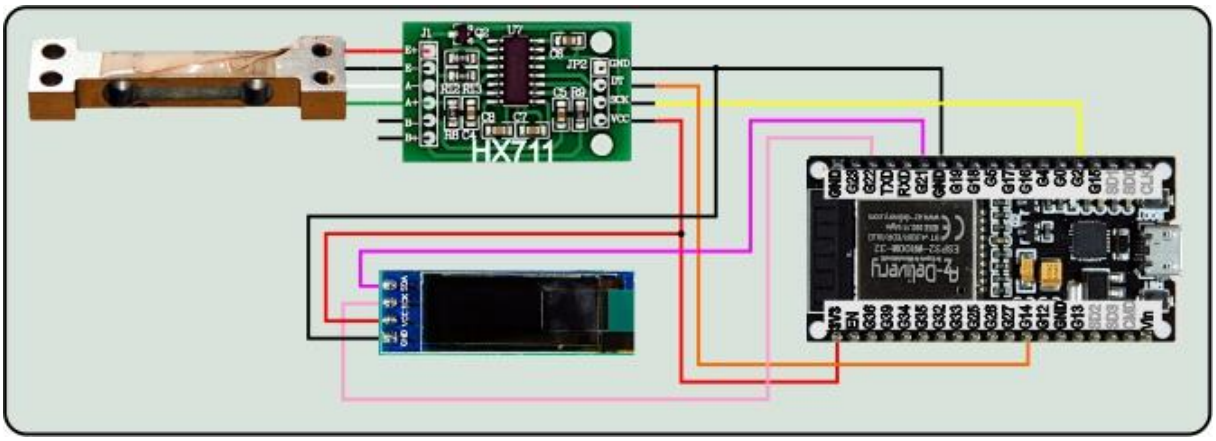

Abbildung 7: hx711-Waage am ESP32

Das MicroPython-Modul für den HX711

Wie die meisten Sensorbaugruppen, braucht auch das HX711-Modul eine Betriebssoftware. Leider ist der HX711 nicht I2C-fähig. Die Datenübermittlung erfolgt aber auch seriell über die Leitungen **dout** und **dpclk**. Es werden stets 24 Bit plus 1 bis 3 Bit für die Auswahl des Kanals A oder B und die Einstellung der Verstärkung übertragen. Das MSBit (Most Significant Bit = hochwertigstes Bit) sendet der HX711 als erstes Bit.

Für die Programmierung des Moduls hx711.py habe ich das Datenblatt des Herstellers benutzt.

```
from time import sleep_us, ticks_ms

class DeviceNotReady(Exception):
    def __init__(self):
        print("Fehler\nHX711 antwortet nicht")
```

Die Exception-Klasse behandelt den Fall, dass der HX711 nicht ansprechbar ist. Es folgt die Deklaration der Klasse HX711, die von **DeviceNotReady** erbt. Mit dem Werfen der Exception wird eine Instanz davon erzeugt, und der Konstruktor sorgt für die Ausgabe der Fehlermeldung.

```
class HX711(DeviceNotReady):
    KselA128 = const(1)
    KselB32 = const(2)
    KselA64 = const(3)
    Dbits =const(24)
    Frame = const(1<<Dbits)
    ReadyDelay = const(3000) # ms
    WaitSleep =const(60) # us
    ChannelAndGain={
        1:("A",128),
        2:("B",32),
        3:("A",64),
        }
    KalibrierFaktor=2205.5
```

Wir beginnen mit einigen Konstanten. Der Kalibrierfaktor wird durch einige Wägungen mit verschiedenen bekannten Massestücken und einem Kalkulationstool, zum Beispiel Libre Office, bestimmt. Dazu komme ich später.

```
    def __init__(self, dOut, pdSck, ch=KselA128):
        self.data=dOut
        self.data.init(mode=self.data.IN)
        self.clk=pdSck
        self.clk.init(mode=self.clk.OUT, value=0)
        self.channel=ch
        self.tare=0
        self.cal=HX711.KalibrierFaktor
        self.waitReady()
        k,g=HX711.ChannelAndGain[ch]
        print("HX711 bereit auf Kanal {} mit Gain {}".\
              format(k,g))
```

Der Konstruktor nimmt die beiden Pin-Objekte **dOut** und **pdSck** sowie optional den Kanal in **ch**. dOut wird als Eingang geschaltet, denn er soll ja die Daten vom Ausgang des HX711 empfangen. Über die Leitung dpSck gibt der ESP den Takt vor. Wir deklarieren die Attribute **channel**, **tare** und **cal**, dann warten wir auf das Ready-Signal des HX711. Kommt es nicht, dann wirft **waitReady**() eine **DeviceNotReady**-Exception. Hat es geklappt, dann holen wir die Kanal- und Gain-Einstellung und geben eine Meldung im Terminal aus. Das Dictionary **ChannelAndGain** wandelt die Kanalnummer in Klartext um.

```
    def TimeOut(self,t):
        start=ticks_ms()
        def compare():
            return int(ticks_ms()-start) >= t
        return compare
```

Die Closure **TimeOut**() realisiert einen Softwaretimer, der den Programmablauf nicht blockiert, wie es **sleep**() & Co. tun. Die zurückgegebene Funktion **compare**() wird einem Bezeichner zugewiesen, über den abgefragt wird, ob die übergebene Zeit in Millisekunden schon abgelaufen ist (True) oder nicht (False).

```
    def isDeviceReady(self):
        return self.data.value() == 0
```

Die Methode **isDeviceReady**() gibt True zurück, wenn die Datenleitung auf GND-Potenzial liegt. Das ist laut Datenblatt der Zustand, wenn der HX711 bereit ist, Daten zu senden. Mit der ersten positiven Flanke auf der Taktleitung stellt der HX711 das MSBit auf der dOut-Leitung zur Verfügung.

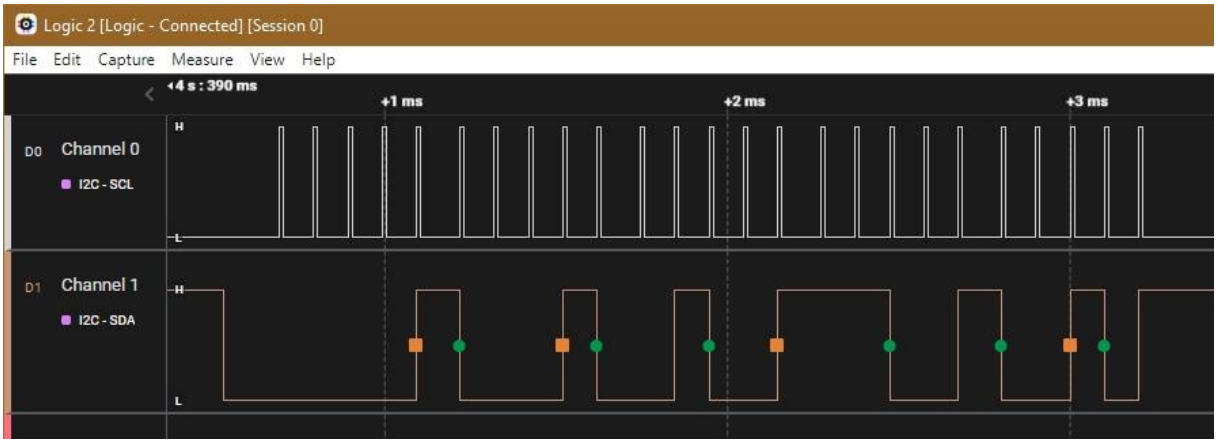

Abbildung 8: logic_2-Scan

Mit jedem weiteren Puls werden die Bits der Reihe nach hinausgeschoben. In der Zwischenzeit muss der ESP den Zustand der Leitung einlesen und verarbeiten. Durch den Takt gibt der ESP gibt das Tempo vor. Die Pulsfolge liegt bei 125µs, das entspricht einer Taktung mit ca. 8kHz.

Nach dem Einlesen der 24 Daten-Bits, werden noch ein bis drei weitere Pulse an **pdSck** ausgegeben, die Steuerbits. Sie haben folgende Bedeutung:

pdSck-Pulse	Input-Kanal	Gain
25	A	128
26	B	32
27	A	64

Abbildung 9: Bedeutung der Steuerpulse

```
    def waitReady(self):
        delayOver = self.TimeOut(ReadyDelay)
        while not self.isDeviceReady():
            if delayOver():
                raise DeviceNotReady()
```

Der Name ist Programm, **waitReady**() wartet auf ein True von **isDeviceReady**(), stellt aber zuvor noch den Timer auf den Wert in **ReadyDelay**, das sind drei Sekunden. Geht dOut in dieser Zeit nicht auf LOW, wird eine **DeviceNotReady**-Exception geworfen. Dadurch wird das aufrufende Programm abgebrochen, falls es die Exception nicht abfängt.

```
    def convertResult(self,val):
        if val & MinVal:
            val -= Frame
        return val
```

Der HX711 sendet die Daten als Zweierkomplement-Werte. convertResult() erkennt eigentlich negative Werte am gesetzten MSBit, dem Bit 23. Ist es gesetzt, wird vom eingelesenen Wert die Stufenzahl 2^{24} subtrahiert um wirklich eine negative Zahl zu bekommen.

0xC17AC3 = 12679875 hat ein gesetztes MSBit
12679875 – 0x1000000 = -4097341

```
    def clock(self):
        self.clk.value(1)
        self.clk.value(0)
```

clock() erzeugt einfach nur einen positiven Puls von 12,4µs auf der pdSck-Leitung.

```
    def kanal(self, ch=None):
        if ch is None:
            ch,gain=HX711.ChannelAndGain[self.channel]
            return ch,gain
        else:
            assert ch in [1,2,3],\
            "Falsche Kanalnummer: {}\n \
             Korrekt ist 1,2 3".format(ch)
            self.channel=ch
            if not self.isDeviceReady():
                self.waitReady()
            for n in range(Dbits + ch):
                self.clock()
```

Übergibt man an **kanal**() kein Argument, dann liefert die Funktion die aktuellen Kanal und Gain-Werte zurück. Das Dictionary **HX711.ChannelAndGain** übernimmt die Übersetzung in Klartext.

Ist eine Kanalnummer übergeben worden, prüfen wir auf den korrekten Bereich, schauen nach, ob der HX711 bereit ist, und schieben dann die entsprechende Anzahl Pulse auf die pdSck-Leitung, 24 plus die Steuerbits.

```
    def getRaw(self, conv=True):
        if not self.isDeviceReady():
            self.waitReady()
        raw = 0
        for b in range(Dbits-1):
            self.clock()
            raw=(raw | self.data.value())<< 1
        self.clock()
        raw=raw | self.data.value()
        for b in range(self.channel):
            self.clock()
        if conv:
            return self.convertResult(raw)
        else:
            return raw
```

Die Rohwerte, wie sie der HX711 liefert, werden von **getRaw**() empfangen. Wird **True** oder gar kein Argument übergeben, dann ist der Rückgabewert rosa, also eine Ganzzahl mit Vorzeichen. Mit False wird der Wert **bloody** zurückgegeben, wie er eben der Bitfolge entspricht, roh.

Wir warten auf die Sendebereitschaft des HX711, und setzen den Empfangspuffer auf 0.

In der for-Schleife senden wir 23 Pulse auf pdSck. Mit jedem Durchgang oderieren wir an die Stelle des LSBits (Least Significant Bit = niederwertigstes Bit) den Zustand auf der Datenleitung und schieben dann die Bits um eine Position nach links. Damit wandert das erste empfangene Bit an die Position 23, das MSBit. Nach einem weiteren Taktpuls schiebt der HX711 das LSBit auf die Datenleitung, das wir nur noch auf das LSBit von **raw** oderieren müssen. Außer Konkurrenz müssen dann noch die Steuerbits für die nächste Messung getaktet werden.

```
    def mean(self, n):
        s=0
        for i in range(n):
            s += self.getRaw()
        return int(s/n)
```

Um das Rauschen der Werte zu glätten verwenden wir nicht nur einen einzigen Messwert, sondern den Mittelwert aus mehreren Messungen. Das besorgt die Methode **mean**(), der wir die Anzahl der Einzelmessungen übergeben.

```
    def tara(self, n):
        self.tare = self.mean(n)
```

Die unbelastete Waage liefert natürlich auch schon einen ADC-Wert, die Tara. Wir rufen die Methode also stets beim Booten des Wägeprogramms auf, um die Anzeige auf 0 setzen zu können. Der Tara-Wert wird im attribut **self.tare** gespeichert, damit ihn die Methode **masse**() zur Verfügung hat, n ist wieder die Anzahl von Einzelmessungen.

```
    def masse(self,n):
        g=(self.mean(n)-self.tare) / self.cal
        return g
```

Die Methode **masse**() zieht vom Messwert die Tara ab und berechnet dann den wahren Messwert in Gramm durch Division mit dem Kalibrierfaktor. Wie der bestimmt wird zeige ich später.

```
    def calFaktor(self, f=None):
        if f is not None:
            self.cal = f
        else:
            return self.cal
```

Während der Kalibrierung ist es von Vorteil, wenn man den Kalibrierfaktor händisch angleichen kann, ohne das Modul hx711.py jedes Mal neu auf den ESP hochladen zu müssen. Ohne Argumentübergabe wird der aktuelle Wert zurückgegeben.

```
    def wakeUp(self):
        self.clk.value(0)
        self.kanal(self.channel)

    def toSleep(self):
        self.clk.value(0)
        self.clk.value(1)
        sleep_us(WaitSleep)
```

wakeUp() und **toSleep**() kitzeln den HX711 aus dem Schlafmodus oder versetzen ihn in denselben. Die Signalfolge auf den Leitungen ist durch das Datenblatt des HX711 vorgegeben.

Ein großer Zeichensatz für das OLED-Display

Große Ziffern erleichtern die Ablesung der Messung erheblich. Statt der üblichen acht Pixel, verwenden wir 30 als Zeichenhöhe. Die Datei **geometer_30.py** enthält die entsprechenden Informationen dazu.

Und so stellen Sie sich einen eigenen Zeichensatz aus dem TTF-Vorrat von Windows her.

Laden Sie das Archiv **zeichensatz.rar** herunter und entpacken Sie den Inhalt in ein beliebiges Verzeichnis. Um Tipparbeit zu sparen, empfehle ich ein Verzeichnis mit einem kurzen Namen im Root-Pfad der Festplatte oder eines Sticks. Bei mir ist es **F:\fonts**.

Öffnen Sie aus dem Explorer heraus ein Powershell-Fenster in diesem Verzeichnis, indem Sie mit gedrückter Shift-Taste einen Rechtsklick auf das Verzeichnis machen. Dann Linksklick auf **PowerShell-Fenster hier öffnen**.

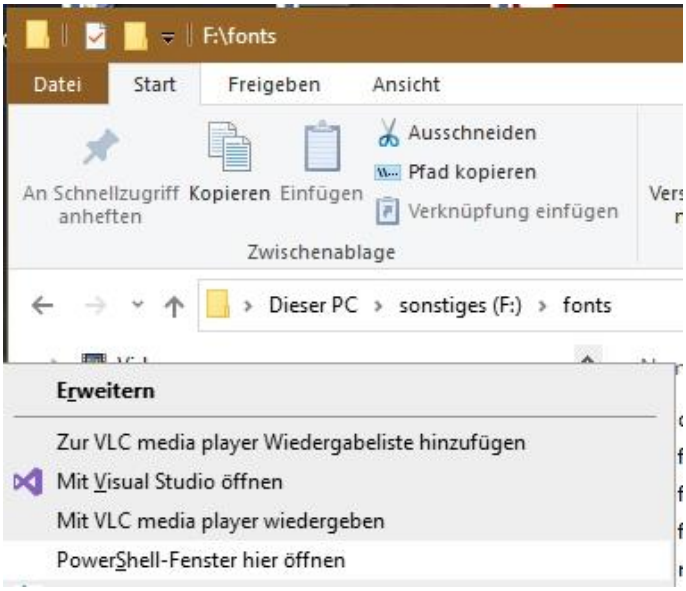

Abbildung 10: Powershell-Fenster öffnen

Geben Sie am Prompt folgende Zeile ein und drücken Sie **Enter**.

.\makecharset.bat britannic 30 """0123456789,-+KG""" "F:\fonts\quellen\"

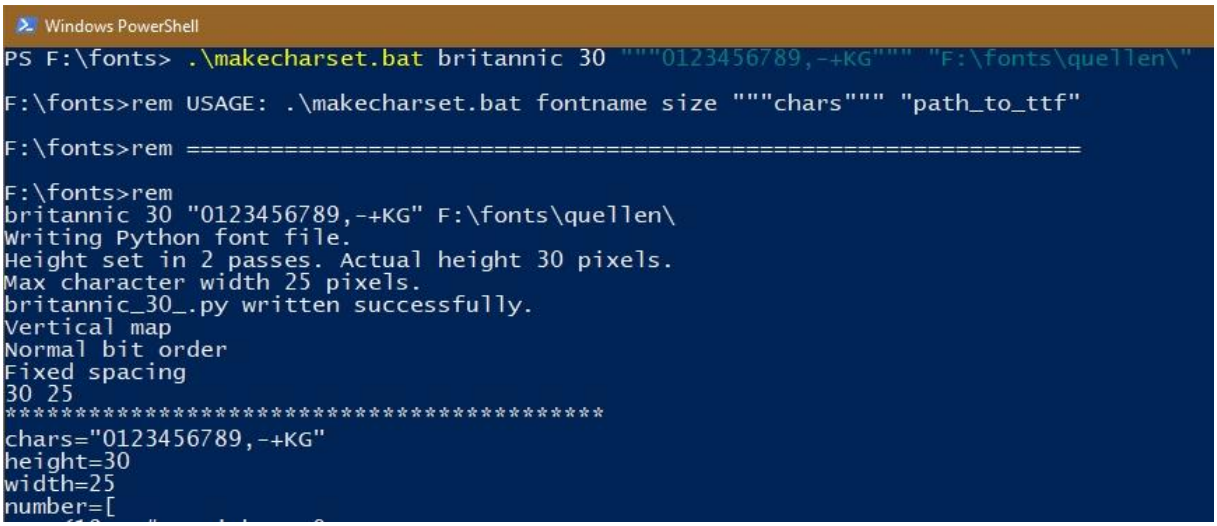

Abbildung 11:Der Zeichensatz ist fertig

Im Verzeichnis befindet sich jetzt eine Datei **britannic_30.py** mit den Pixeldaten des neuen Zeichensatzauszugs. Umgesetzt wurden nur die Zeichen in **"""0123456789,-+KG"""**, das spart Speicherplatz. Weitere Zeichensätze können Sie aus dem **fonts**-Verzeichnis von Windows in das Verzeichnis **quellen** kopieren und wie oben

angegeben umwandeln. Beachten Sie bitte, dass der Dateiname ohne die Ergänzung .TTF angegeben wird.

Die Betriebssoftware der Waage

Das Programm **scale.py** greift auf vier externe Module zu, die vor Beginn in den Flash des ESP hochgeladen werden müssen, **hx711.py**, **oled.py**, **ssd1306.py** und **geometer_30.py**.

```
from machine import Pin,freq,SoftI2C
from time import sleep
from oled import OLED
import geometer_30 as cs
import sys
from hx711 import HX711
```

Über die Variable **sys.platform** kann ein ESP seinen Typ selbst feststellen. Danach werden die GPIO-Pins für den I2C-Bus deklariert.

```
if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4),freq=400000)
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=400000)
else:
    raise UnkownPortError()
```

Wir instanziieren ein Display-Objekt, löschen die Anzeige und deklarieren die Pin-Objekte für die Verbindung zum HX711, außerdem **taste**, die Instanz für die Tara-Taste..

```
d=OLED(i2c,128,32)
d.clearAll()

dout=Pin(14) # D5
dpclk=Pin(2) # D4
taste=Pin(0,Pin.IN,Pin.PULL_UP) # D3
```

Die Funktion **putNumber**() positioniert das Pixelmuster des Zeichens mit der Nummer **n** (bezogen auf den oben erzeugten Auszug, nicht ASCII). Die Position der linken oberen Ecke des Musters steht in **xpos**, **ypos**.

```
def putNumber(n,xpos,ypos,show=True):
    breite=cs.number[n][0]
    for row in range(1,cs.height):
        for col in range(breite-1,-1,-1):
            c=cs.number[n][row] & 1<<col
            d.setPixel(xpos+breite+3-col,ypos+row,c,False)
    if show:
        d.show()
    return xpos+breite+2
```

Ein Blick in die Datei **geometer_30.py** erklärt das Vorgehen. Im String **chars** sind die Zeichen aufgelistet. Der Index in den String ist die Zeichennummer. Das "-"-Zeichen ist somit Nummer 11, die "0" ist Nummer 0. In der Liste **number** identifiziert diese Nummer gleichzeitig das Tupel mit der Breitenangabe und der Pixelmatrix. Diese wird nun zeilenweise Spalte für Spalte abgetastet und das Pixel gesetzt, wenn eine 1 gefunden wurde. Das geschieht verdeckt. Nur wenn **show** den Wert **True** hat, wird der gesamte Puffer zum Display geschickt. Der Rückgabewert ist die nächste freie Zeichenposition.

scale.py | hx711.py | main.py | geometer_30.py

```
chars="0123456789,-"
height=31
width=29
number=[
    (17,  #  Zeichen: 0
     0b00000000000000000,
     0b00000000000000000,
     0b00000000000000000,
     0b00000011111000000,
     0b00001111111110000,
     0b00011111111111000,
     0b00011111111111000,
     0b00111110001111100,
     0b00111110001111100,
     0b01111100000111110,
     0b01111100000111110,
     0b01111100000111110,
     0b01111100000111110,
     0b01111100000111110,
     0b01111100000111110,
     0b01111100000111110,
     0b01111100000111110,
     0b00111110001111100,
     0b00111110001111100,
     0b00011111111111000,
     0b00011111111111000,
     0b00001111111110000,
     0b00000011111000000,
     0b00000000000000000,
     0b00000000000000000,
     0b00000000000000000,
     0b00000000000000000,
     0b00000000000000000,
     0b00000000000000000,
     0b00000000000000000,
    ),
```

Die Zeichenposition wird auf 0 gestellt, als Startinformation gebe ich eine Reihe von "-"-Zeichen aus. Dann versucht der ESP den HX711 zu kontakten. Gelingt das, wecken wir ihn auf, stellen Kanal A mit Gain 128 ein und stellen den Tarawert fest, der im Attribut **self.tare** landet.

```
pos=0
try:
    for n in range(8):
        pos=putNumber(11,pos,0)
    hx = HX711(dout,dpclk)
    hx.wakeUp()
    hx.kanal(1)
    hx.tara(25)
    print("Waage gestartet")
except:
    print("HX711 initialisiert nicht")
    for n in range(8):
        pos=putNumber(0,pos,0)
    sys.exit()

freq(160000000)
```

Klappt die Verbindung nicht, bekommen wir eine Meldung im Terminal und im Display eine Reihe von Nullen.

160MHz ist für den ESP8266 Topspeed, der ESP32 pack auch noch 240MHz.

Dann geht's in die Hauptschleife. Wir sehen nach, ob die Tara-Taste gedrückt ist, bestätigen das mit einer Reihe von Kommas im Display und holen einen neuen Tara-Wert. Diese Feature ist nützlich, weil die Wägezelle einer Temperaturdrift unterliegt und damit der Nullpunkt jederzeit nachjustiert werden kann.

```
while 1:
    if taste.value() == 0:
        d.clearAll(False)
        pos=0
        for n in range(14):
            pos=putNumber(10,pos,0)
        hx.tara(25)
```

Der Format-String für die Ausgabe wird mit dem Messwert gefüttert. Verdeckt das Display löschen, Zeichenposition auf 0, Dezimalpunkt durch ein Komma ersetzen und die Zeichen bis zum vorletzten verdeckt in den Puffer schreiben.

```
    m="{:0.2f}".format(hx.masse(10))
    d.clearAll(False)
    pos=0
    m=m.replace(".",",")
    for i in range(len(m)-1):
        z=m[i]
        n=cs.chars.index(z)
        pos=putNumber(n,pos,0, False)
```

Mit der Ausgabe des letzten Zeichens wird der Pufferinhalt zum Display geschickt. Nach einer halben Sekunde Pause startet die nächste Runde.

```
    z=m[len(m)-1]
    n=cs.chars.index(z)
    pos=putNumber(n,pos,0)
    state=(taste.value() == 0)
    sleep(0.5)
```

Das komplette Programm können Sie hier herunterladen.

Die Waage kalibrieren

Abbildung 12:Massenstücke fürs Kalibrieren der Waage

Sie brauchen dazu ein paar Massenstücke und gegebenenfalls eine Küchen- oder Briefwaage, falls die Massenstücke nicht geeicht sind. In diesem Fall können Sie sich einen Wägesatz aus fetten Muttern oder Schrauben oder ähnlichem herstellen. Natürlich müssen die Massen vor Verwendung mit einer anderen Waage bestimmt werden.

Starten Sie jetzt einmal das Programm **scale.py** in Thonny und brechen Sie in der Hauptschleife mit **Strg+C** ab. Das HX711-Objekt ist unter **hx** instanziiert und von der Terminal-Konsole aus ansprechbar. Wir starten mit folgenden Befehlen.

>>> hx.tara(50)
>>> hx.tare
562708
>>> hx.mean(25)-hx.tare
-17
>>> hx.mean(25)-hx.tare
8

Vom Mittelwert aus 25 Einzelmessungen subtrahieren wir den Tara-Wert. Die Ergebnisse sollten im Bereich zwischen -50…+50 liegen, wenn die Waage nicht belastet ist.

Jetzt legen wir die Massenstücke nacheinander und in Gruppen auf die Waage, so dass sie zunehmend mehr belastet wird. Jedes Mal starten wir erneut den letzten Befehl und notieren die Masse in Gramm und den im Terminal angezeigten Wert.

Die Tabelle geben wir in Libre Office Calc oder einem anderen Programm ein und lassen uns die Messkurve zusammen mit dem Bestimmtheitsmaß (Korrelationskoeffizient) und der Formel anzeigen.

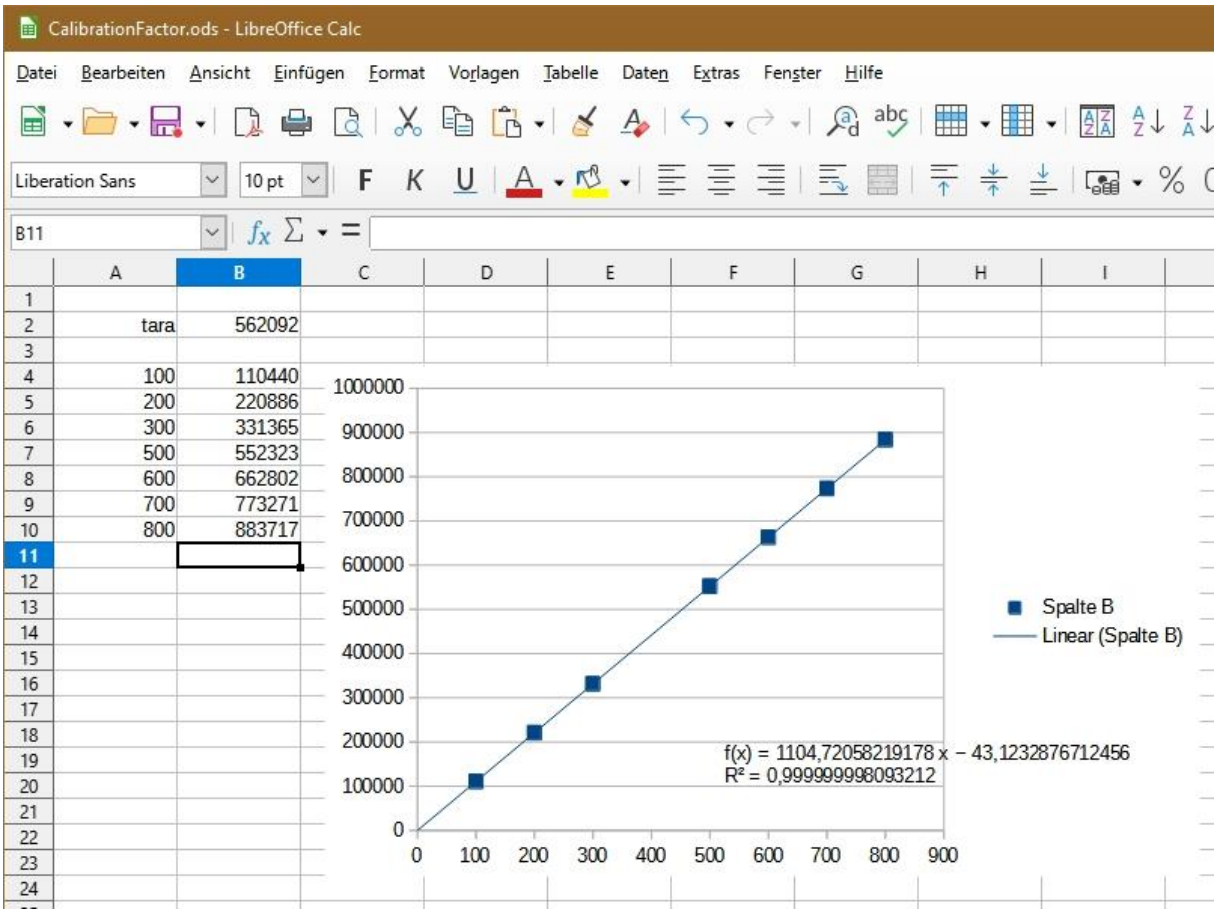

Abbildung 13: Kalibrierkurve

Der Korrelationskoeffizient R^2 ist faktisch 1, das spricht für die Präzision unserer Messung. Den Achsenabschnitt von -43.blabla können wir bei einer Größenordnung von 100000 vernachlässigen. Der Steigungsfaktor 1104 der Geraden ist unser gesuchter Kalibrierfaktor. Teilen Sie diesen Wert nun dem Objekt **hx** mit.

>>> hx.calFaktor(1104)
>>> hx.calFaktor()
1104

Wenn Sie jetzt die Methode **masse**() aufrufen, sollt ein Wert von 0,0blabla angezeigt werden.

>>> hx.masse(25)
0.0244565

Tragen Sie den Kalibrierfaktor in der Datei **hx711.py** ein, laden Sie diese zum ESP hoch und starten Sie das Programm **scale.py** neu.

```
    KalibrierFaktor=1104
```

Wenn jetzt beim Auflegen der Massenstücke deren Masse angezeigt wird, haben Sie gewonnen und können sich belobigend auf die Schulter klopfen.

Liegt der angezeigte Wert leicht neben dem erwarteten Gewicht, dann können Sie noch ein wenig am Kalibrierfaktor-Wert feilen. Wenn Sie ihn vergrößern, wird der Massenwert sinken und umgekehrt.