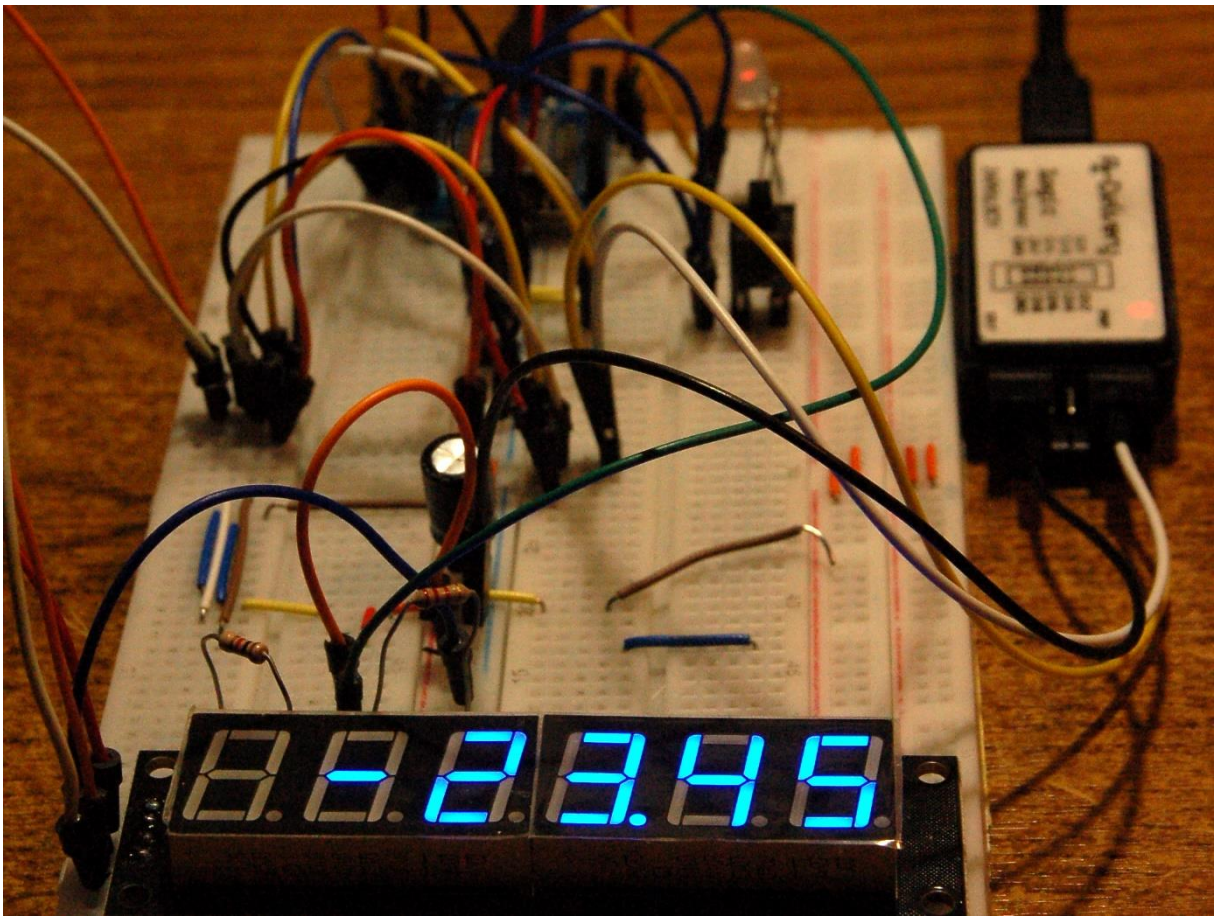

Anzeige in der Entwicklung

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

LCD- und OLED-Displays sind schön. Es lassen sich einige Informationen auf den 2, 3 oder 6 Zeilen darstellen. OLEDs sind sogar grafikfähig. Aber manchmal wünsche ich mir eine Anzeige mit wirklich großen Ziffern. Bei meinen kleinen Selbstbau-Waagen mit 100g und 1000g habe ich OLED-Displays eingesetzt. Nun kam ein 20kg-Exemplar dazu, und da fand ich die 0,96"-Anzeige, aber auch eine übliche 1602-LCD denn doch zu popelig.

Beim Suchen im Netz stieß ich dann auf eine 6-fach-LED-Anzeige mit 14mm hohen Ziffern. Das war es genau, was ich brauchte, zumal die Ansteuerung nur über zwei Leitungen erfolgt. Mit Hilfe des Datenblatts, stellte sich schnell heraus, dass das Übertragungsprotokoll ziemlich genau dem I2C-Protokoll entspricht, aber eben nicht ganz. Die einzige Abweichung: es wird keine Hardwareadresse zu Beginn des Transfers gesendet. Aber sonst gibt es eine Start-Condition, eine Stop-Condition und ein Acknowledge-Bit, wie beim I2C-Bus.

Natürlich kann durch diese Umstände das, im Kernel von MicroPython eingebaute, I2C-Modul leider nicht verwendet werden. Also habe ich ein Ersatzmodul auf der Basis des Datenblatts gestrickt, das optimal die Bedingungen für das Display der Waage erfüllt. Eine Überraschung hatte das Display dennoch auf Lager. Doch dazu später mehr. Wie man das Display dazu bringt Klartext-Zahlen rechtsbündig darzustellen, das lesen Sie in dieser Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Digitalwaage mit LED-Display

Kümmern wir uns zuerst einmal um die Hardware des Displays. Außer diesem selbst wird neben den bisherigen Baugruppen für die Waage nichts weiter benötigt. Der Treiberbaustein für die Sieben-Segment-Anzeigen sitzt auf der Unterseite des Moduls.

Abbildung 1: TM1637 von oben

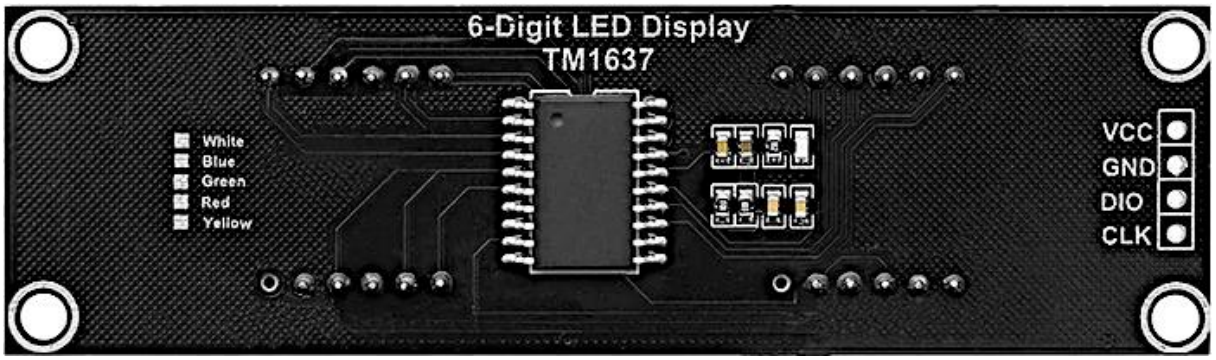

Abbildung 2: TM1637 von unten

Das bisherige Drumherum sind der ADC für die Waage, ein HX711-Modul und ein Controller vom Typ ESP8266 oder ESP32 sowie eine Taste, um die Tara zu berücksichtigen. Wie der Controller mit dem Treiberbaustein zusammenarbeitet, das erfahren Sie im ersten Post zum Thema Waage. Wir werden hier das Modul für die neue Anzeige durchleuchten und das Betriebsprogramm der Waage auf das neue Display anpassen.

Hardware

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	TM1637 6 Digit blaue LED-Anzeige 7 Segment Display Modul mit 0,56 Zoll
1	Wägezelle 20 kg
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

Der Logic Analyzer ist ein sehr nützliches Instrument, wenn es bei der seriellen Datenübertragung hakt. Er ersetzt in vielen Fällen ein teures DSO (Digitales Speicher Oszilloskop) und bietet darüber hinaus noch den Vorteil längerer Aufzeichnungen, in die man dann gezielt hineinzoomen kann. Zu dem hier verlinkten Gerät gibt es eine kostenlose Betriebs-Software, das Teil wird über den PC angesteuert. Mir hat es schon in vielen verzweifelten Fällen geholfen, auch in diesem Fall. Das Protokoll des TM1637 ist zwar im Datenblatt ausreichend dargestellt, doch übersieht man schon gerne mal ein Detail. Vergleicht man dann das Impulsdiagramm im Datenblatt mit dem selbst erstellten, kommt man sehr schnell auf die Lösung des Problems.

Hier sind die Schaltungen für ESP32 und ESP8266.

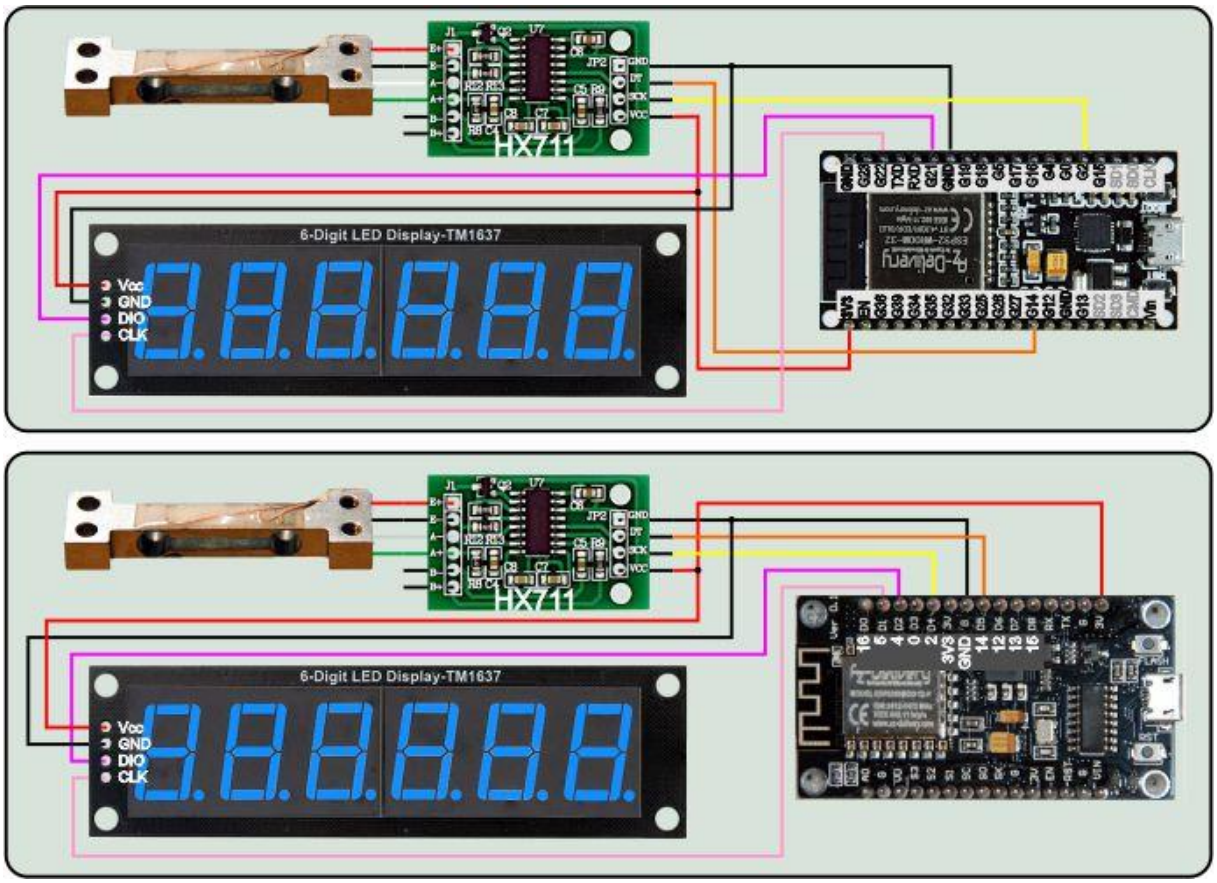

Abbildung 3: Waage - Schaltung für ESP32 und ESP8266

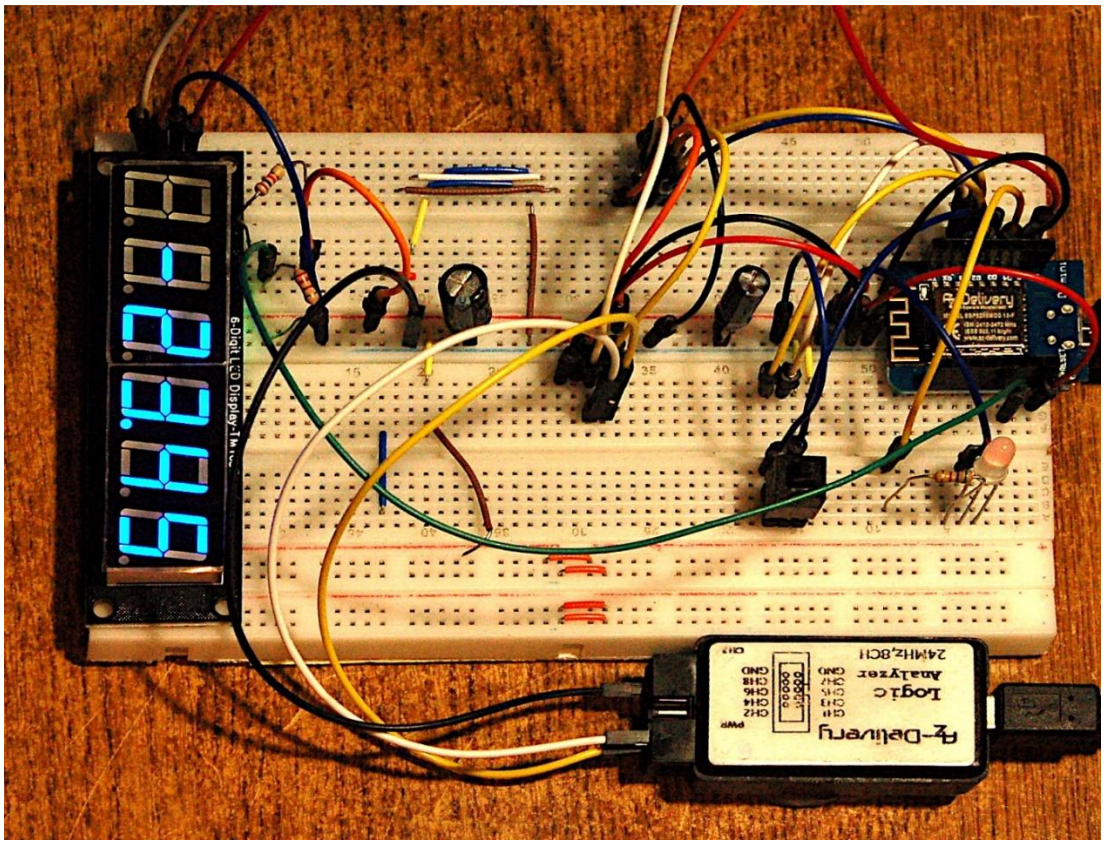

Abbildung 4: Anzeige Aufbau mit TM1637 im Test mit dem ESP8266 D1 mini

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:

hx711neu.py API für den AX711
scale1637.py Das Betriebsprogramm
tm1637.py API für die 7-Segment-Anzeige

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 Klasse TM1637

Der TM1637 verwendet keine Hardwareadresse wie es normalerweise auf dem I2C-Bus üblich ist, das habe ich oben schon erwähnt. Es gibt auch keine Register, sondern nur Kommandos, Commands, nämlich drei: Data command, Display and control command und Address command. Die Signalfolge in der Abbildung stellt den Schreibzugriff mit automatischem Hochzählen der Adresse nach jedem gesendeten Daten-Byte dar.

Die Sequenz beginnt mit einer Start-Condition, DIO geht auf LOW, während CLK auf HIGH ist.

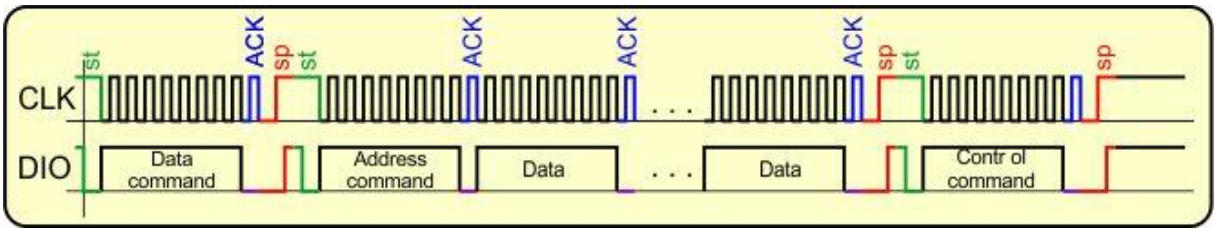

Abbildung 5: Signalverlauf beim Schreiben ins SRAM des TM1637

Mit der fallenden Taktflanke stellt der Controller das erste Datenbit auf die DIO-Leitung und setzt daraufhin CLK auf HIGH, der TM1637 übernimmt das Bit. Das erste Byte ist der Data command, 0x40. Sind 8 Bits, beginnend beim LSB (Least Significant Bit = niederwertigstes Bit), übertragen, zieht der TM1637 mit fallender Taktflanke DIO auf LOW, wenn die Übertragung OK war. Die neunte steigende Taktflanke triggert das Acknowledge-Bit. Es folgt eine Stop-Condition (CLK ist HIGH, DIO zieht nach einer Verzögerung nach) und sofort danach eine erneute Start-Condition.

Danach sendet der Controller mit dem Address command 0xC0 die erste Speicheradresse, ab welcher die Daten fortfolgend abgelegt werden. Nach jedem Daten-Byte kommt ein Acknowledge und nach dem letzten Byte eine Stop-Condition.

Das dritte Kommando, mit eigener Start-Condition, Acknowledge und Stop-Condition, steuert das Display. Die unteren drei Bits setzen die Helligkeit, Bit 3 schaltet die Anzeige an oder aus.

Schauen wir uns an, wie das alles programmtechnisch umgesetzt werden kann. Wir starten mit einem geringen Importaufkommen.

```
from machine import Pin
from time import sleep_us, sleep_ms
```

Es folgen ein paar Exception-Klassen für die Fehlerbehandlung. Die Container-Klasse **TM1637_Error** erbt von **Exception**, der Mutter aller Ausnahmeklassen. Die Subklassen erben von **TM1637_Error**.

```
class TM1637_Error(Exception):
    pass

class BrightnessError(TM1637_Error):
    def __init__(self):
        super().__init__("Falscher Kontrastwert",
                         "0 <= Wert <= 7")

class PositionError(TM1637_Error):
    def __init__(self):
        super().__init__("Falscher Positionswert",
                         "0 <= Wert <= 5")

class StringLengthError(TM1637_Error):
    def __init__(self):
        super().__init__("String zu lang",
                         "0 <= Wert <= 5")
```

Die Klasse TM1637 wird deklariert. Die Konstanten setzen die Basiswerte für die Commands. **MSB** dient zum Aktivieren des Dezimalpunkts eines Digits, indem es zum Segmentcode oderiert wird.

```
class TM1637():
    DataCmd = const(0x40)  # data cmd - write, autoincr.,
normal
    AdrCmd = const(0xC0) # address command f. Register 0
    DispCntrl = const(0x80) # disp ctrl cmd - an/aus Kontrast
    DispOn = const(0x08) # display an
    MSB = const(0x80)  # Dezimalpunkt
    a=[2,1,0,5,4,3]

Segm=bytearray(b'\x3F\x06\x5B\x4F\x66\x6D\x7D\x07\x7F\x6F')
```

Zu den Variablen, der Liste **a** und dem Bytearray **Segm** muss ich etwas ausholen.

Die Abfolge der Digits im Display war zu meinem Erstaunen nicht von links nach rechts, oder meinetwegen auch umgekehrt, sondern so wie in Abbildung 3. Das verkompliziert die Sache ein wenig.

Abbildung 6: Displayanordnung

Wenn ich einen Anzeigestring aus einem Messwert bilde, können die Ziffern nicht in ihrer natürlichen Reihenfolge an das Display gesendet werden, weil das ein kleines Durcheinander erzeugt. Aus 123456 würde 321654, mal was anderes! Die Liste a=[2,1,0,5,4,3] stellt die Zuordnung zwischen String und realer Anzeigeposition her. Was im String an der Position 0 steht, muss in den Speicher für das zweite Digit geschrieben werden, damit die Ziffer ganz links in der Anzeige auftaucht. Die 1 muss in Digit 2 landen. Der Index in die Liste ist also die Position im Ziffernstring, das Listenelement, die Digitnummer, wo die Ziffer, oder besser, deren Segmentmuster, landen soll. Ich komme später noch einmal darauf zurück.

Das Bytearray **Segm** enthält die Segmentmuster der Ziffern 0 bis 9 nach dem Schema in Abbildung 5.

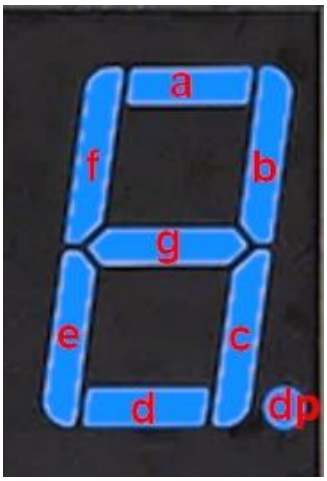

Abbildung 7: Segmentanordnung

Jedes Segment entspricht einer Bitposition nach folgendem Muster.

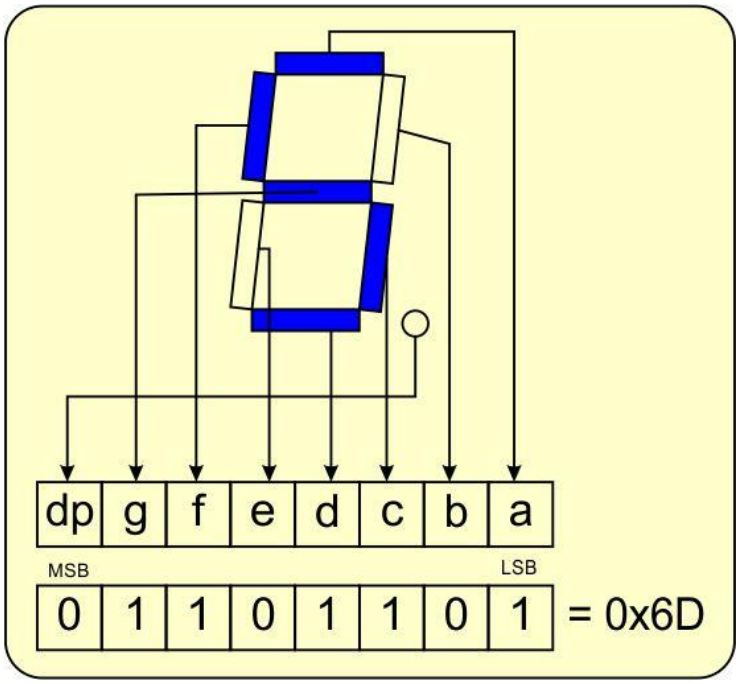

Abbildung 8: Zifferncodierung

Wenn wir 0x6D in den Anzeigespeicher 3 schreiben, erscheint eine 5 in der Position rechts außen im Display, und wenn wir mit der Adresse 0xC0 beginnen, dann muss 0x6D als vierter Wert übertragen werden, um in 0xC3 zu landen.

Weiter geht es mit dem Konstruktor der Klasse TM1637, der Methode **__init__**().

```
    def __init__(self, clk=Pin(5), dio=Pin(4), brightness=3):
        self.clk = clk
        self.dio = dio
        if not 0 <= brightness <= 7:
            raise BrightnessError
        self.brightness = brightness
        self.clk.init(Pin.OUT, value=1)
```

```
        self.dio.init(Pin.OUT, value=1)
        self.delay=5
        sleep_us(10) # 10us warten
        self.clearDisplay()
        print("TM1637 ready")
```

Es können drei optionale Schlüsselwortparameter übergeben werden, die Pin-Objekte für CLK und DIO sowie für den Kontrast oder auch die Helligkeit, wie Sie wollen. Wird kein Argument übergeben, dann gelten die Defaultwerte. Alle Parameter werden Attributen zugewiesen, der Kontrastwert wird über dies auf Einhaltung des Wertebereichs überprüft. Liegt **brightness** nicht im zulässigen Bereich, dann wird eine **BrightnessError**-Exception geworfen.

Die Pins werden auf Ausgang gesetzt. Als Verzögerung für den Takt lege ich 5µs vor, das entspricht einer Frequenz von 100kHz. Wir warten kurz, löschen das Display, dann meldet der Konstruktor die Einsatzbereitschaft des Objekts im Terminal.

Mit **latency**() können wir das ganzzahlige Argument in **val** als den Wert der Verzögerung im Attribut **delay** ablegen, nachdem der Wertebereich (1…20 für 500kHz…50kHz) gegebenenfalls eingegrenzt wurde. Ohne Argument aufgerufen, liefert die Methode den aktuellen Wert von **delay** zurück.

```
    def latency(self, val=None):
        if val is None:
            return self.delay
        else:
            if type (val) != int:
                raise LatencyTypeError
            val = min(max(val,1),20)
            self.delay=val
            return val
```

Die Methode **startCond**() folgt den oben genannten Vorgaben für die Signalsequenz. Der Ruhezustand auf beiden Leitungen ist HIGH. **DIO** geht zuerst auf LOW, dann folgt **CLK**.

```
    def startCond(self):
        self.dio(0)
        sleep_us(self.delay)
        self.clk(0)
        sleep_us(self.delay)
```

Für das Erzeugen einer Stop-Condition muss DIO zuerst sicher auf LOW sein und die Taktleitung auf HIGH. verzögert geht dann DIO auf HIGH.

```
    def stopCond(self):
        self.dio(0)
        sleep_us(self.delay)
        self.clk(1)
        sleep_us(self.delay)
        self.dio(1)
```

Zwischen Start- und Stop-Condition eingebettet ist der Transfer des Data-Command-Bytes.

```
    def writeDataCmd(self):
        self.startCond()
        self.writeByte(DataCmd)
        self.stopCond()
```

Das Nämliche gilt für **writeDispCntrl**(). Allerdings werden auf das nackte Kommandobyte 0x80 weitere Bits durch Oderieren aufgepfropft. Mit **DispOn** = 0x08 setzen wir Bit 3. Die drei Kontrastbits 2:0 stehen in **brightness**.

```
    def writeDispCntrl(self):
        self.startCond()
        self.writeByte(DispCntrl | DispOn | self.brightness)
        self.stopCond()
```

writeByte() ist die universelle Methode zum Versenden eines Bytes unter Berücksichtigung des Acknowledge-Bits, das aber nicht gescannt wird. Wir müssten sonst DIO auf Eingang schalten, den Zustand einlesen und anschließend wieder auf Ausgang schalten. Bislang ist kein Fehler aufgetreten, also habe ich die Prüfung weggelassen.

```
    def writeByte(self, b):
        for i in range(8):
            self.dio((b >> i) & 1)
            sleep_us(self.delay)
            self.clk(1)
            sleep_us(self.delay)
            self.clk(0)
            sleep_us(self.delay)
        sleep_us(self.delay) # ACK-Takt folgt
        self.clk(1)
        sleep_us(self.delay)
        self.clk(0) # naechstes Byte vorbereiten
        sleep_us(self.delay)
```

Die for-Schleife schiebt das übergebene Byte mit dem LSB beginnend auf die DIO-Leitung. CLK ist von der Start-Condition her noch auf LOW. Das Byte wird um i=0 bis 7 Positionen nach rechts geschoben und jetzt das LSB maskiert. Das Ergebnis ist 0 oder 1. Damit wird der Ausgang gesteuert.

Nachdem der Zustand stabilisiert ist, erzeugen wir an CLK eine steigende Flanke, der TM1637 sampelt den Zustand auf DIO. Nachdem der Takt wieder auf LOW ist, folgt die Bereitstellung des nächsten Bits, der Vorgang wiederholt sich, bis alle Bits draußen sind. CLK bleibt nach dem letzten Bit für **delay** Sekunden auf LOW, dann folgt als letztes der Acknowledge-Takt, der wieder mit CLK=LOW endet. Es kann nun ein weiteres Byte oder eine Stop-Condition folgen.

Zum Testen der Anzeige aber auch zur Ausgabe ganz spezifischer Muster, zum Beispiel für ASCII-Zeichen, dient die Methode **segment**(). In **seg** wird das Muster übergeben (default 0xFF) und in **pos** die Nummer des Digits (default 0x00). Die Ausgabeposition wird überprüft.

```
    def segment(self,seg=0xFF,pos=0):
        if not 0 <= pos <= 5:
            raise PositionError
        self.writeDataCmd()
        self.startCond()
        self.writeByte(AdrCmd | TM1637.a[pos])
        self.writeByte(seg)
        self.stopCond()
        self.writeDispCntrl()
```

writeDataCmd() hat eigene Start- und Stop-Conditionen. Bevor die Adresse gesendet wird, muss aber eine Start-Condition eingebaut werden. Nach der Basis-Speicheradresse mit oderierter Digitnummer folgen das Segmentbeschreibungs-Byte, die Stop-Condition und das display-Control-Byte. pos spricht die reale Position des Digits in der Anzeige an, das indizierte Listenelement die physikalische Speicheradresse, aus 0 wird so die 2, aus 5 die 3 etc.

```
>>> from tm1637 import TM1637
>>> tm=TM1637()
>>> aber=bytearray(b'\x77\x7C\x79\x50')
>>> for i in range(len(aber)):
        tm.segment(aber[i],i)
```

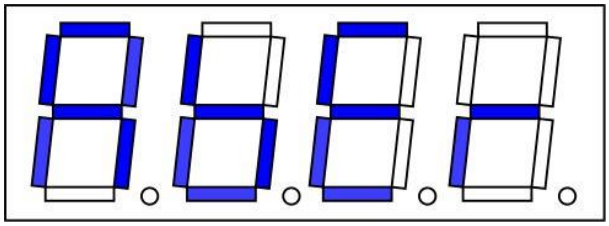

Abbildung 9: Schriftzug AbEr

kontrast() funktioniert ähnlich wie **latency**(). Ohne Argument wird der aktuelle Wert zurückgegeben. Mit einem Wert zwischen 0 und 7 inclusive der Grenzen wird die Helligkeit neu eingestellt. Im Zusammenhang mit einem Fotowiderstand könnte man zum Beispiel so die Helligkeit der Anzeige dem Umgebungslicht anpassen.

```
    def kontrast(self, val=None):
        if val is None:
            return self._brightness
        if not 0 <= val <= 7:
            raise BrightnessError
        self.brightness = val
        self.writeDataCmd()
        self.writeDispCntrl()
```

Um das Display zu löschen sende ich sechs Null-Bytes.

```
    def clearDisplay(self):
        segments=(bytearray(b'\x00\x00\x00\x00\x00\x00'),-1)
        self.writeSegments(segments)
```

Das Tupel **segments** enthält ein Bytearray mit den Segmentcodes und eine Ganzzahl. Diese gibt die Nummer des Digits an, bei dem der Dezimalpunkt angesteuert werden muss, falls es sich bei der Zahl um den Typ **float** handelt. Der Wert -1 deutet auf eine Ganzzahl hin. Wir kommen weiter unten noch genauer darauf zu sprechen. Das Tupel übergeben wir an **writeSegments**().

Einen Funktionstest aller Filamente erledigt **lampTest**() nach demselben Muster wie **clearDisplay**().

```
    def lampTest(self):
        segments=(bytearray(b'\xFF\xFF\xFF\xFF\xFF\xFF'),-1)
        self.writeSegments(segments)
```

Bis zu sechs Segmentmuster ab einer vorgegebenen Position senden, das kann **writeSegments**(). Die Muster stehen im Tupel **segmente**, dahinter kommt die Position. Für diesen Wert führen wir eine Plausibilitätskontrolle durch.
Nun dröseln wir das Tupel in Muster und Dezimalpunkt-Position auf. Der String darf nur so lang sein, wie ab **pos** noch Digits dafür da sind, wir testen das.

Passt alles, schicken wir den Data-Command, gefolgt von einer Start-Condition und der Start-Adresse. Die for-Schleife bringt die Ziffern an die korrekte Position.

```
    def writeSegments(self, segmente, pos=0):
        if not 0 <= pos <= 5:
            raise PositionError
        s,p=segmente
#         print(s,p)
        if len(s) + pos > 6:
            raise StringLengthError
        self.writeDataCmd()
        self.startCond()
        self.writeByte(AdrCmd | pos)
        for i in range(pos,6):
            c=s[TM1637.a[i]]
            if p==TM1637.a[i]:
                c|=MSB
            self.writeByte(c)
        self.stopCond()
        self.writeDispCntrl()
```

Die Segmentmuster für Zahlen, die wir mit **number2Segments**() erzeugen beginnen alle ab der realen Digit-Position ganz links außen. Das ist die pysikalische Position 2 im Speicher. Beginnen müssen wir die Sendesequenz aber mit der relativen Speicheradresse 0, absolut 0xC0, sonst müssten wir jedem Datenbyte die Adresse vorausschicken. Wir wollen aber das Autoincrement nutzen und die sechs Daten-Bytes in einem Abwasch senden. Auch hier hilft wieder die Liste a= [2,1,0,5,4,3]. Sie sagt uns nämlich, welches Zeichen des Strings an welche Speicherstelle gesendet werden muss.

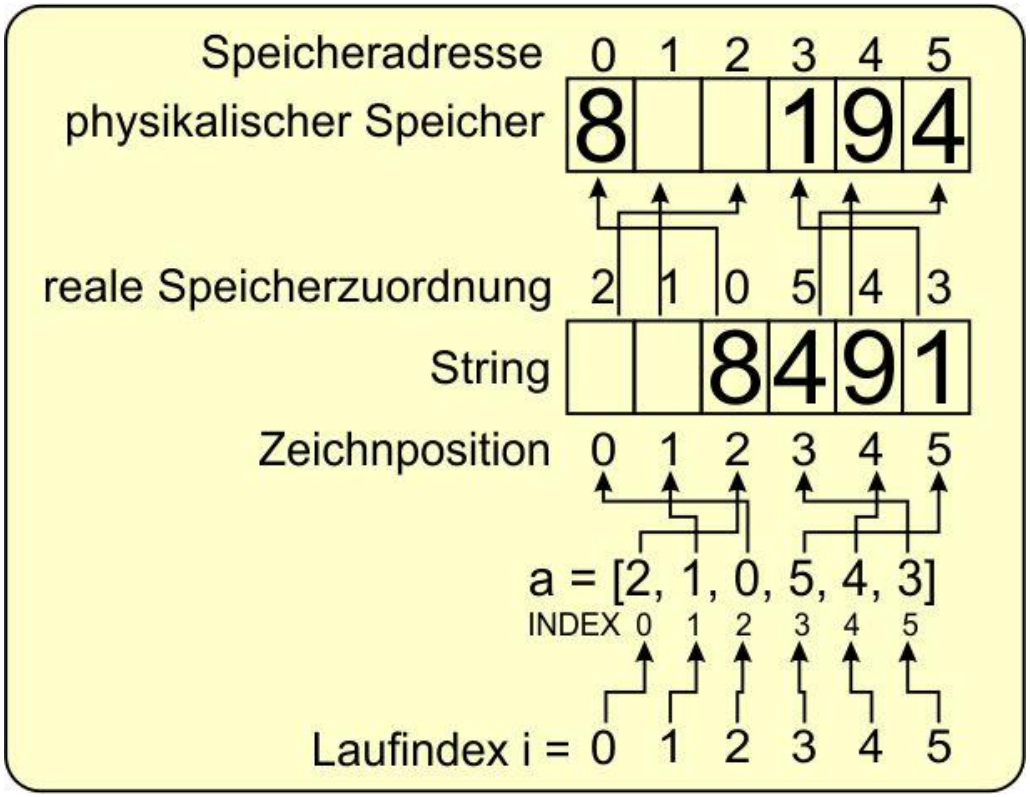

Abbildung 10: String auf Speicher zuweisen

Das i in der for-Schleife durchläuft die physikalischen Speicherpositionen. Es dient als Zeiger in die Liste a. Das Element an dem jeweiligen Listenplatz ist ein Zeiger auf die Position des Zeichens im String beziehungsweise Bytearray. Der Code für dieses Zeichen wird in die Speicherstelle geschrieben die gerade mit i adressiert wird.

Wenn p den Wert von a[i] hat, wird zu dem Segment-Code noch das MSB oderiert, was dazu führt, dass der Dezimalpunkt aktiviert wird. Dann wird das Byte zum TM1637 geschickt.

Nach den, in der Regel sechs Bytes kommt eine Stop-Condition und danach der Display-Control-Command.

Fehlt noch die Codierung von Ganzzahlen und Fließkommazahlen in Segmentcodes. **Number2Segments**() nimmt die Zahl, die mit Komma und Vorzeichen natürlich nicht länger als 6 Zeichen sein darf, und ein optionales Argument **k**. Mit diesem geben wir die Anzahl von Nachkommastellen an, falls die Zahl vom Typ float ist. Auf den Typ prüfen wir als Erstes.

```
    def number2Segments(self, n, k=1):
        if type(n)==int:
            s="{:>6}".format(n)
        elif type(n)==float:
            s="{:>7."+str(k)+"f}"
            s=s.format(n)
        else:
            raise TypeError
```

Ist der Typ **int**, also Ganzzahl, dann wandeln wird den Wert über den Formatstring in einen rechtsbündig formatierten (">") String, mit eventuell führenden Leerzeichen, von der minimalen Länge 6 um. Ist die Zahl von Typ **float**, müssen wir berücksichtigen, dass bei der Nachbehandlung des Strings der Dezimalpunkt als separates Zeichen wegfällt. Deswegen geben wir dem String eine minimale Breite von 7 Zeichen. In den Formatstring arbeiten wir die Anzahl Nachkommastellen ein.

```
        pos=s.find(".")
        if pos != -1:
            s=s.replace('.','')
            pos-=1
```

Dann suchen wir nach der Position eines potenziellen Dezimalpunkts. Existiert keiner, dann ist es eine Ganzzahl, pos erhält den Wert -1. Andernfalls enthält pos den Index auf den Punkt. In diesem Fall ersetzen wir den Punkt im String durch ein leeres Zeichen. Die Position verringern wir um 1, denn der Punkt muss beim Digit davor berücksichtigt werden.

```
        if len(s)>6:
            raise StringLengthError
        segments = bytearray(len(s))
```

Ist jetzt der aufbereitete String länger als 6 Zeichen, werfen wir eine **StringLengthError**-Exception. Ist alles im grünen Bereich, erzeugen wir ein Bytearray von der Länge des Strings. Der dürfte nach der momentanen Lage stets die Länge 6 haben. Jetzt geht es ans eigentliche Codieren. Die for-Schleife klappert jedes Zeichen ab.

```
        for i in range(len(s)):
            if s[i] == " ":
                segments[i]=0x00
```

Ist das Zeichen an der Position i ein Leerzeichen, darf kein Filament leuchten – Segmentcode 0x00.

```
            elif s[i] == "-":
                segments[i]=0x40
```

Ist es ein Minuszeichen, dann brauchen wir nur den Mittelstrich – Code 0x40

```
            else:
                segments[i] = TM1637.Segm[ord(s[i]) - 48]
        return (segments,pos)
```

In allen anderen Fällen holen wir den Code aus dem Bytearray **Segm**. Als Index dient uns der um 48 verringerte ASCII-Code der Ziffer. 48 ist der ASCII-Code der "0".

Zurückgegeben wird das Bytearray zusammen mit der Punktposition als Tupel.

Das Betriebsprogramm für die Waage

Durch den Einsatz des LED-Displays ist das Betriebsprogramm deutlich schlanker geworden. Das liegt an der simpleren Art der Displayansteuerung. Im Zusammenhang mit der Waage habe ich dem Display auch ein wenig Klartext beigebracht, es kann "Error" und "tara", nicht gerade künstlerisch wertvoll, aber für den Zweck ausreichend. Mit den effektiv 40 Programmzeilen ist das Programm sehr übersichtlich. OK, die Hauptarbeit wird in den Modulen hx711 und tm1637 erledigt, aber selbst die sind mit 139 beziehungsweise 161 Zeilen noch recht schnuckelig. Auf jeden Fall passt alles ganz locker auch in einen ESP8266.

```
from machine import Pin
from time import sleep
from tm1637 import TM1637
from hx711neu import HX711
```

Wir brauchen Pins für die GPIO-Steuerung, sleep für Pausen und natürlich die Klassen TM1637 und HX711.

Ein TM1637-Objekt wird instanziiert und die Pin-Objekte für die Bedienung des HX711 werden erzeugt.

Die Pin-Objekte beim Konstruktoraufruf des Display-Objekts muss ich nicht angeben, weil ich einen ESP8266 verwende und daher die Default-Pins, GPIO5 und GPIO4, benutzt werden. Als Taste dient wieder die Flash-Taste oder eine externe Taste an GPIO0.

```
tm=TM1637()

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

Eine einzige Funktion gibt es. **putNumber**() erledigt die Messwertausgabe. Der Wert vom HX711 wird in ein **Segments**-Tupel codiert und zum Display geschickt.

```
def putNumber(n):
    s=tm.number2Segments(n)
    tm.writeSegments(s)
```

Es folgt der Versuch, die Waage zu initialisieren. Dem Konstruktor übergeben wir die Pin-Objekte für die Daten- und Taktleitung. Der Chip wird aufgeweckt. Wir arbeiten mit Kanal 1, die Wägezelle liegt am Eingang A des HX711, und wir arbeiten mit voller Verstärkung. Bei jedem Start des Programms wird automatisch die Tara bestimmt und zwar mit 25 Einzelmessungen. Ein Lampentest informiert über die Funktion aller Filamente und darüber, dass bislang alles fehlerfrei gelaufen ist.

```
try:
    hx = HX711(dout,dpclk)
    hx.wakeUp()
    hx.kanal(1)
    hx.tara(25)
    tm.lampTest()
    sleep(1)
    print("Waage gestartet")
```

Sollte ein Fehler aufgetreten sein, meldet der Except-Block "Error" am Display und das Programm wird beendet.

```
except:
    print("HX711 initialisiert nicht")
    s=(b"\x79\x50\x50\x5C\x50\x00",-1)
    tm.writeSegments(s)
    sys.exit()
```

In der Hauptschleife gibt es zwei Jobs. Wenn die Taste gedrückt ist, wird ein neuer Tara-Wert ermittelt und gespeichert. Das erlaubt uns, das Verpackungsgewicht abzuziehen oder das Zuwiegen von Zutaten. In der Anzeige erscheint "tArA". Nach dem Messvorgang arbeitet das Programm erst weiter, wenn die Taste losgelassen wurde.

```
while 1:
    if taste.value() == 0:
        s=(b"\x00\x78\x77\x50\x77\x00",-1)
        tm.writeSegments(s)
        hx.tara(25)
        while taste.value()==0:
            pass
```

Meine 20kg-Wägezelle liefert Werte, die auf der Zehntel-Gramm-Stelle wackeln. Anders ausgedrückt, 0,1 Gramm ist die unsichere Stelle. Ich habe sie deshalb ausgeblendet und gebe mich damit zufrieden, dass die Waage auf 1 Gramm genau misst. Das sind 0,005% vom maximalen Wert von 20000 Gramm. Diese Auflösung ist voll super!

Natürlich muss die Waage geeicht werden, bevor man sie wirklich verwenden kann. Im Vergleich zum Vorgängermodul hx711.py habe ich ein paar neue Features eingebaut, die dafür hilfreich sind und uns die Rechenarbeit, sowie Änderungen am Programm abnehmen.

Die Eichsequenz beginnt mit dem Starten von **scale1637**.py im Editorfenster von Thonny. Brechen Sie das Programm mit **Strg+C** ab, wenn der Lampentest begonnen hat, also alle Segmente leuchten.

Jetzt können Sie alle Methoden der Klasse HX711 händisch vom Terminal aus aufrufen.

Legen Sie jetzt nichts auf die Waage, und setzen Sie jetzt folgende Kommandos ab:

>>> hx.tara(25)
108978
>>> hx.tare
108978

Legen Sie jetzt ein Wägestück auf die Waage, dessen Masse sie möglichst genau wissen. Ich habe hier zwei Eichgewichte von je 500g genommen. Die Masse in Gramm übergeben Sie an **calculateFactor**().

>>> hx.calculateFactor(1000)
102.966

Das war's auch schon. Die Methode hat eine Wägung mit 25 Einzelmessungen gemacht, den Tara-Wert davon abgezogen und das Ergebnis durch die übergebene Masse dividiert. Das Endergebnis hat sie in der Datei **config.txt** im Flash des Controllers abgelegt.

```
    def calculateFactor(self,masse):
        self.cal = (self.mean(25)-self.tare)/masse
        with open("config.txt","w") as f:
            f.write(str(self.cal)+"\n")
        return self.cal
```

Beim Instanziieren des HX711-Objekts, versucht der Konstruktor, diese Datei zu öffnen und den Inhalt auszulesen. Sollte das nicht gelingen, wird ein Wert genommen, der in der Variablen **HX711.KalibrierFaktor** abgelegt ist. Sie können dafür den Wert hernehmen, den Sie eben bestimmt haben. Die Snippets finden Sie alle in der Datei hx711neu.py

```
    KalibrierFaktor=102.966
```

```
    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
        try:
            self.readFactor()
        except:
            self.cal=HX711.KalibrierFaktor
        self.waitReady()
        k,g=HX711.ChannelAndGain[ch]
        print("HX711 bereit auf Kanal {} mit Gain {}".\
              format(k,g))
        print("Kalibrierfaktor is {}".\
              format(self.cal))
```

```
    def readFactor(self):
        with open("config.txt","r") as f:
            self.cal=float(f.readline())
        return self.cal
```

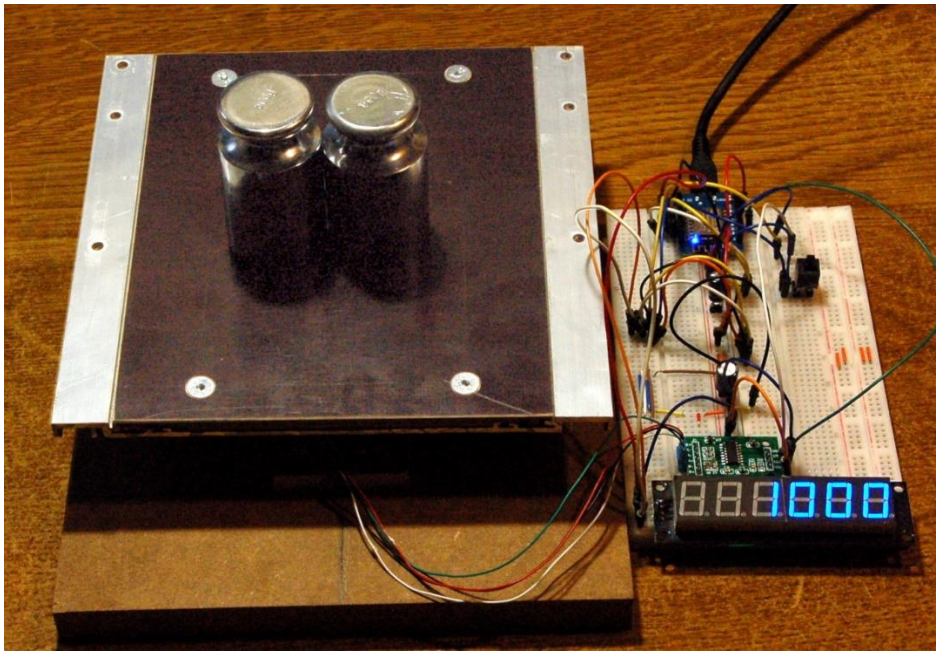

Abbildung 11: Waage mit Schaltung bei der Eichung

Viel Spaß und Erfolg mit der neuen DIY-Waage!