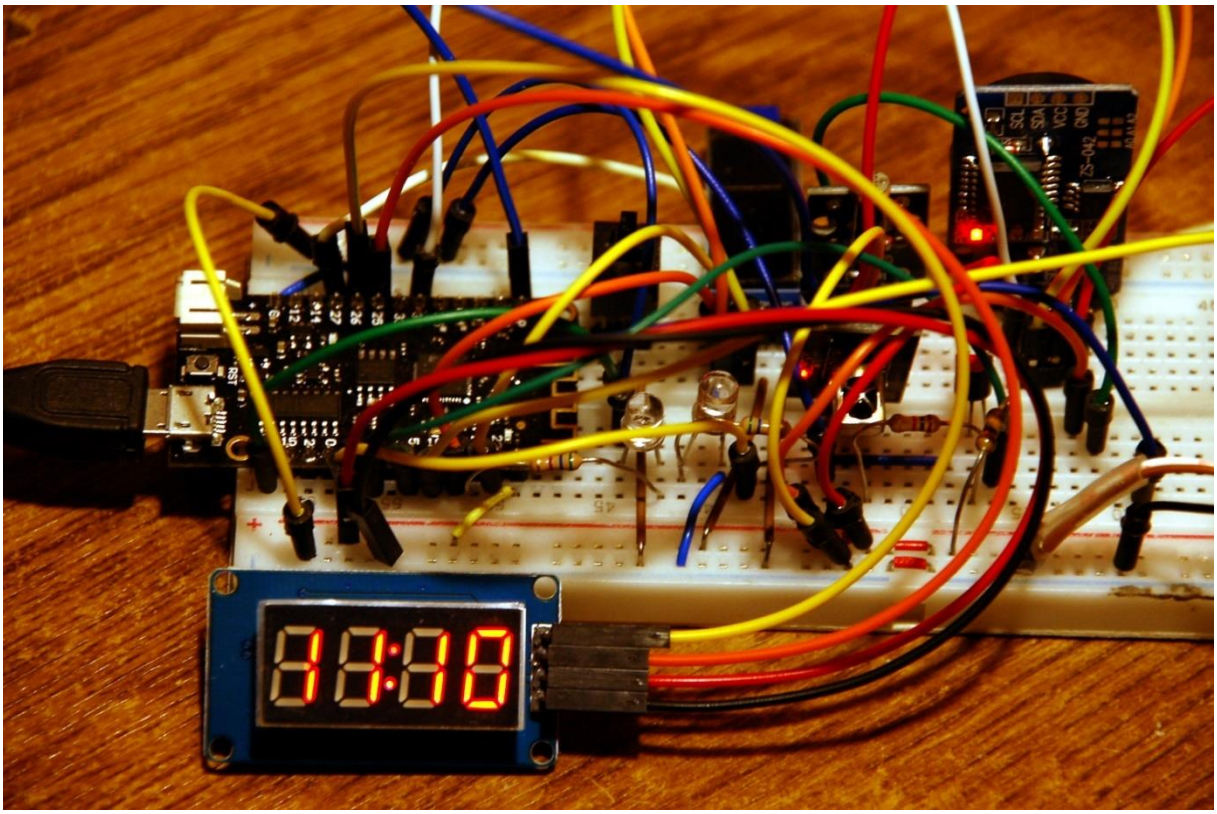

Vierfach 7-Segment-LED-Display

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

Für eine Uhr braucht man eine standesgemäße Anzeige, die auch aus einiger Entfernung lesbar ist. Das OLED war – und ist – für die Entwicklung und Pflege einer Schaltung nebst Programmierung des Controllers sehr nützlich, aber für unser Ziel, den Wecker, eher suboptimal. Daher geht es heute um die Integration eines 4-fachen Sieben-Segment-LED-Displays. Was ich gefunden habe, ist ein Display mit 4 Digits und einer Ziffernhöhe von ca. 10mm, ein guter Kompromiss zwischen Lesbarkeit und Raumparasit. Zudem ist es möglich, die Helligkeit in acht Stufen einzustellen. Das ist gut für eine Nachtabsenkung der Helligkeit.

Die Ansteuerung erfolgt über nur zwei Leitungen. 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. Weil es keine Geräteadresse gibt, darf sich natürlich auch nur ein Anzeigebaustein am Bus befinden.

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 Uhr erfüllt. Eine Überraschung hatte das Display dennoch auf Lager. Doch dazu später mehr. Wie man das Display dazu bringt Stunden und Minuten korrekt darzustellen, das lesen Sie in dieser Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Eine Uhr mit LED-Anzeige

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

Abbildung 1: TM1637 von oben

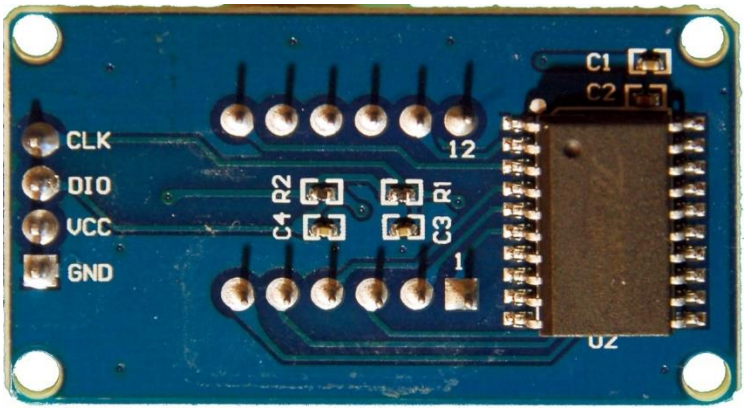

Abbildung 2: TM1637 von unten

Der bisherige Aufbau enthält den ESP32 Lolin, IR-Empfänger und IR-Sender, das DS3231-Modul eine Taste und eine LED sowie ein OLED-Display.

In den vorangegangenen Teilen haben wir schon einiges geschafft. Wir können die RC (Remote Control) auslesen und haben einen eigenen IR-Sender realisiert. Die Möglichkeit eine PS/2-Tastatur an den ESP32 anzuschließen, erlaubt dabei das Auslesen der RC5-Steuerung, ohne dass ein PC angeschlossen sein muss. und zuletzt haben wir dem ESP32 ein RTC-Modul von hoher Ganggenauigkeit zur Seite gestellt.

Heute gesellt sich also das LED-Modul dazu und damit der ESP32 die Helligkeit der Anzeige automatisch der Umgebung anpassen kann, spendieren wir auch noch einen LDR (Light Dependend Resistor = Fotowiderstand).

Hardware

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	KY-022 Set IR Empfänger
1	KY-005 IR Infrarot Sender Transceiver Modul
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
1	KY-004 Taster Modul
diverse	Jumper Wire Kabel 3 x 40 STK
1	Real Time Clock RTC DS3231 I2C Echtzeituhr
1	TM1637 4 Digit 7-Segment LED-Display Modul
1	KY-018 Foto LDR Widerstand Photo Resistor Sensor
2	NPN-Transistor BC337 oder ähnlich
1	Widerstand 1,0 kΩ
1	Widerstand 10 kΩ
1	Widerstand 330 Ω
1	Widerstand 47Ω
1	Widerstand 560Ω
1	LED (Farbe nach Belieben)
1	Adapter PS/2 nach USB oder PS/2-Buchse
1	Logic Analyzer
1	PS/2 - Tastatur

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 ist die Schaltung für den aktuellen Stand der Dinge. Abbildung 3 zeigt die Schaltung des Displays, in Abbildung 4 ist auch das DS3231-Modul noch enthalten.

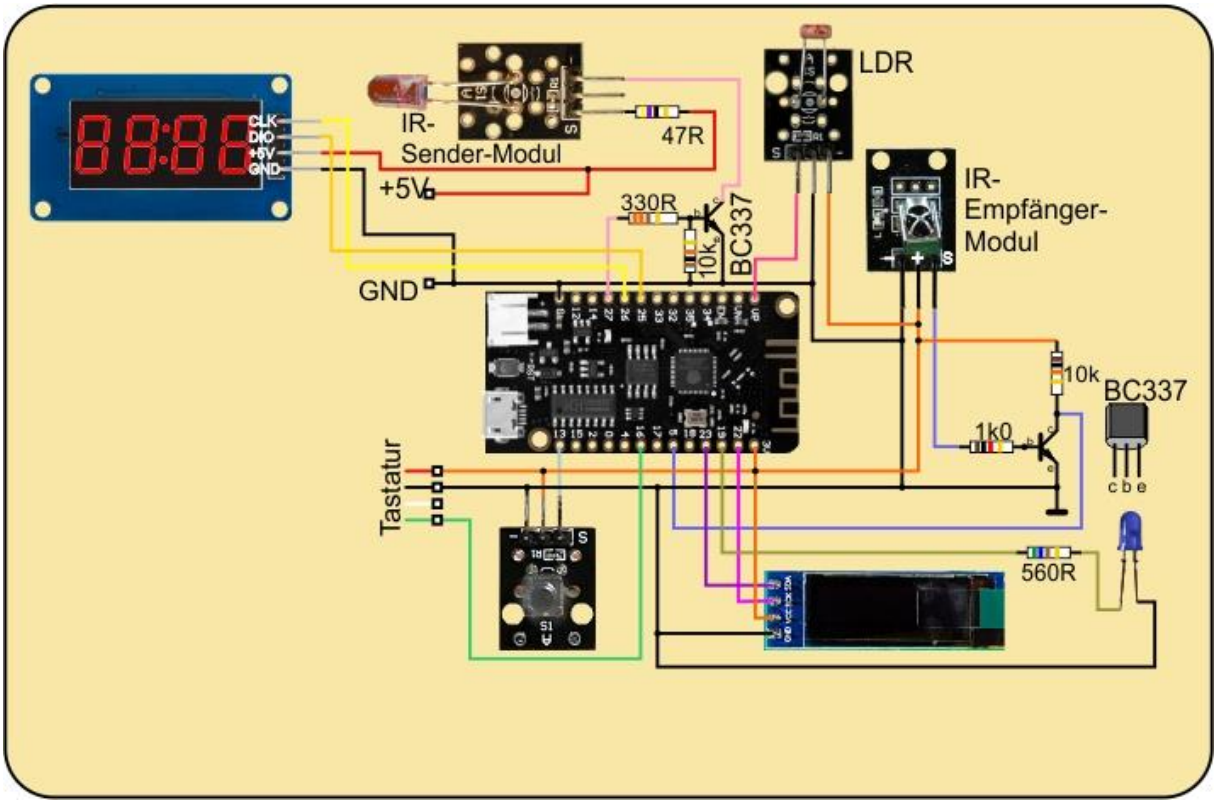

Abbildung 3: Das 7-Segmentdisplay für die Uhr

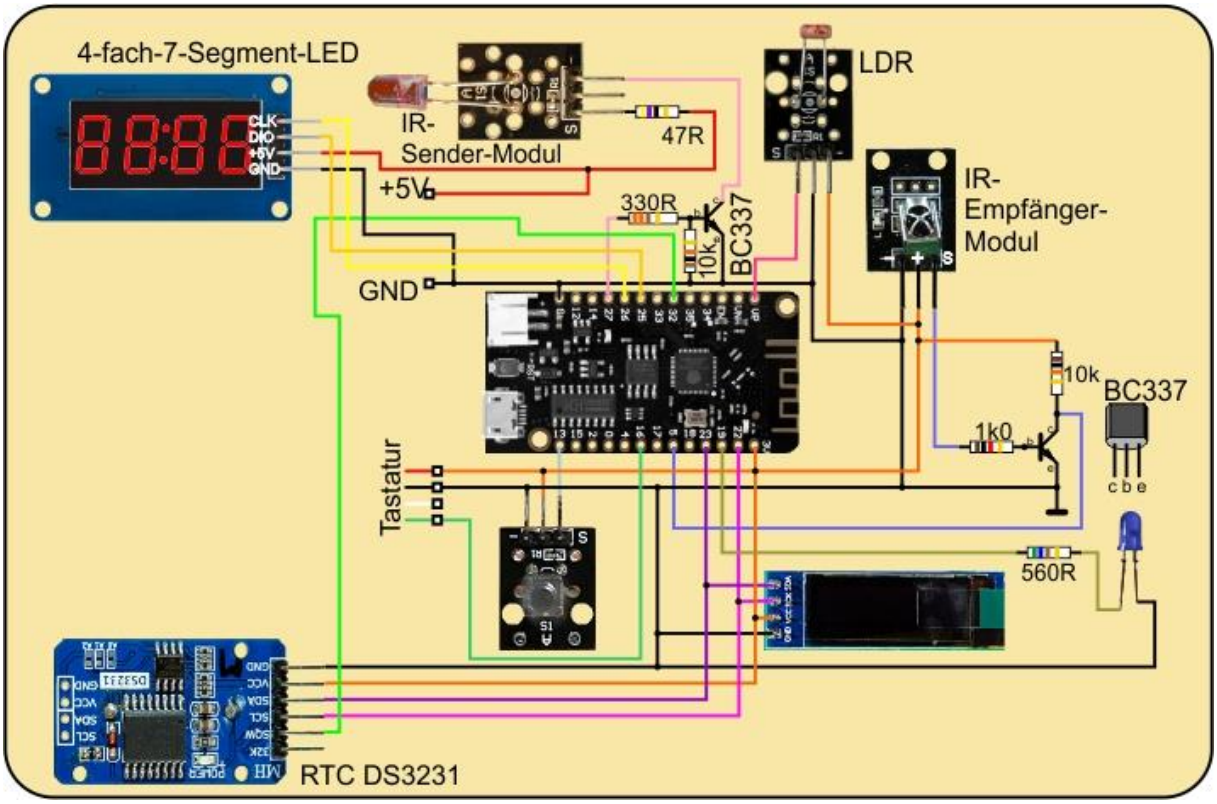

Abbildung 4: DS3231 und Wecker-Display

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft
Betriebs-Software Logic 2 von SALEAE

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:

tm1637_4.py: API für die 4- und 6-stellige 7-Segment-Anzeige mit dem TM1637
ds3231.py: Treiber-Modul für das RTC-Modul
oled.py: OLED-API
ssd1306.py: OLED-Hardware-Treiber
7segment.py: Demoprogramm einer Uhr

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

Es gibt vier- und sechsstellige LED-Displays, die mit dem Chip TM1637 angesteuert werden. Dabei habe ich Unterschiede im Aufbau, beziehungsweise in der Zuordnung der einzelnen Digits festgestellt. Die hier beschriebene Klasse weicht daher von einer früher verfassten Version an den Stellen ab, an denen die Länge des Displays und/oder die Digitanordnung eine Rolle spielen. Es deckt aber beide Anzeigetypen ab.

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 (Autoincrement) nach jedem gesendeten Daten-Byte dar.

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

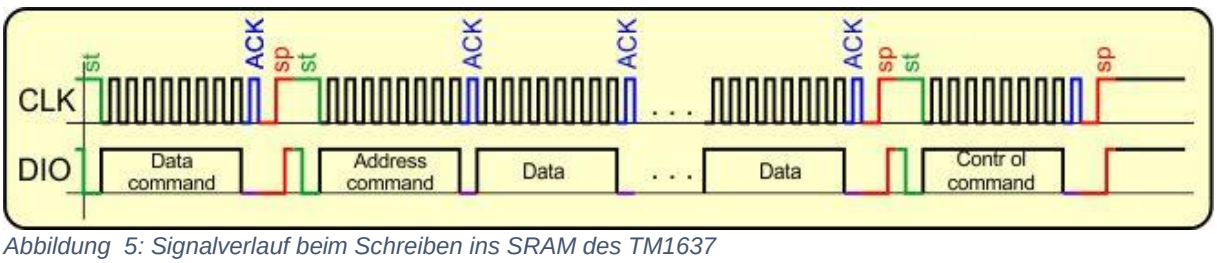

Abbildung 5: Signalverlauf beim Schreiben ins SRAM des TM1637

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")

class DisplayLengthError(TM1637_Error):
    def __init__(self):
        super().__init__("Unbekannter DisplaytypTyp",
                         "4 oder 8 Digits")
```

Die Klasse **TM1637** wird deklariert. Die Konstanten setzen die Basiswerte für die Commands. **MSB** dient zum Aktivieren des Dezimalpunkts (beziehungsweise Doppelpunkts) eines Digits, indem es zum Segmentcode oderiert wird. Worauf diese Operation beruht, werden wir in Kürze sehen.

```
class TM1637(TM1637_Error):
    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/Doppelpunkt
    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, was nur das 6-stellige Display betrifft.

Der TM1637 kann bis zu sechs Digits ansteuern. Die Abfolge der Digits in einem 6-stelligen 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, auch für unser vierstelliges Display, bei dem Digitabfolge und Speicherzuordnung 1:1 erfolgen. Die Anzeigepositionen werden durch den Adressbefehl stets in der Reihenfolge der relativen Speicheradressen von 0 bis 5 angesprochen, wenn man Autoincrement verwenden möchte. Beim vierstelligen Display entspricht das auch der Digitanordnung, beim 6-stelligen dagegen nicht.

Abbildung 6: Displayanordnung

Wenn ich nun 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 Stringposition und Anzeigeposition her. Was im String an der Position 0 steht, muss in den Speicher für das Digit 2 geschrieben werden, damit die Ziffer ganz links in der Anzeige auftaucht. Die 1 muss also in Digit 2 landen. Der Index in die Liste ist damit die Position im Ziffernstring, das adressierte Listenelement, ist 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 7.

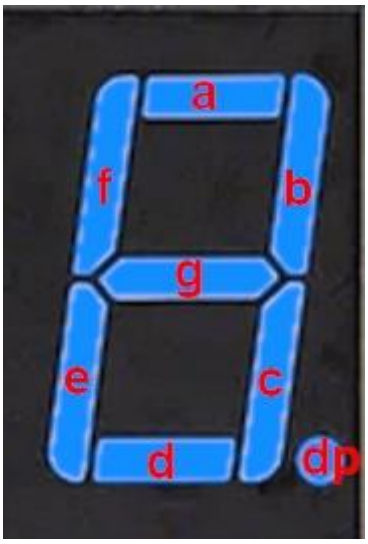

Abbildung 7: Segmentanordnung

Jedes Segment entspricht einer Bitposition nach folgendem Muster.

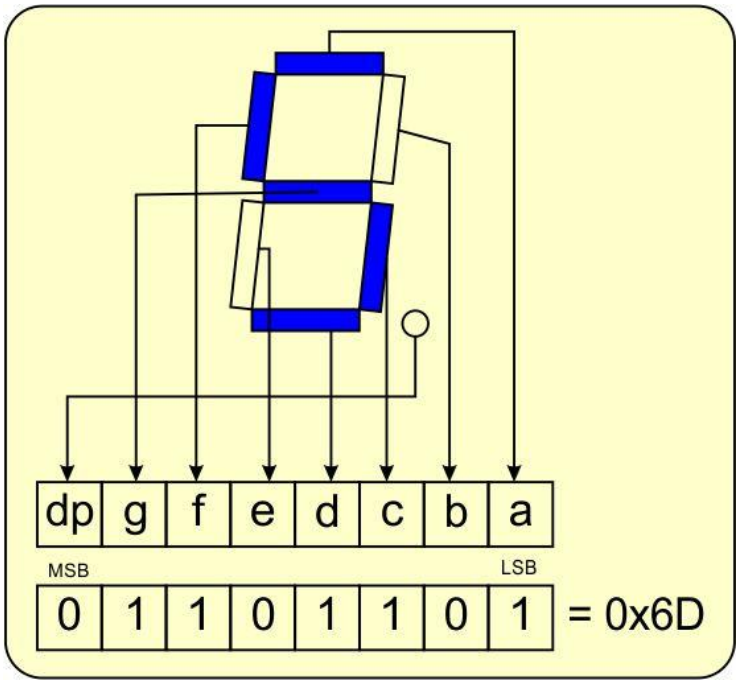

Abbildung 8: Zifferncodierung

Wenn wir 0x6D in den Anzeigespeicher 3 schreiben, erscheint eine 5 in der Position rechts außen im 6-er-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, nofdig, clk, dio, brightness=3):
        self.clk = clk
        self.dio = dio
        if not 0 <= brightness <= 7:
            raise BrightnessError
        if nofdig not in [4,6]:
```

```
            raise DisplayLengthError
        self.nod=nofdig
        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 müssen drei Positionsparameter übergeben werden, die Anzahl der Digits in **nofdig**, die Pin-Objekte für **clk** und **dio** sowie der optionale Schlüsselwortparameter **brightness** für den Kontrast oder auch die Helligkeit, wie Sie wollen. Wird hierfür kein Argument übergeben, dann gilt der Defaultwert 3. Alle Parameter werden Instanz-Attributen zugewiesen, der Kontrastwert wird überdies auf Einhaltung des Wertebereichs überprüft ebenso **nofdig**. Liegt **brightness** nicht im zulässigen Bereich, dann wird eine **BrightnessError**-Exception geworfen. **delay** legt die Verzögerung für die Pegelwechsel bei Start- und Stop-Condition sowie zwischen den Taktflanken fest und bestimmt damit die Frequenz auf der Taktleitung.

Die Pins werden auf Ausgang gesetzt. Als Verzögerung für den Takt lege ich 5µs vor, das entspricht einer Nenn-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…25kHz) 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. Damit schalten wir das Display ein. 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 DIO 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** Millisekunden 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. Im Gegensatz zum 6-stelligen Display mit der oben beschriebenen eigenwilligen Digitanordnung, sind die Digits beim 4-stelligen Display in der Reihenfolge der Speicherzellen von 0 bis 3 von links nach rechts platziert. Die Übersetzung mittels Liste a ist hier also nicht nötig. Die Abfrage der Digitanzahl entscheidet über entsprechend korrekte Platzierung des Segmentcodes im SRAM des TM1637. Eine PositionError-Exception wird geworfen, wenn die Position nicht im zulässigen Bereich ist.

```
    def segment(self,seg=0xFF,pos=0):
        if not 0 <= pos <= self.nod-1:
            raise PositionError
        self.writeDataCmd()
        self.startCond()
        if self.nod == 4:
            self.writeByte(AdrCmd | pos)
        elif self.noc == 6:
            self.writeByte(AdrCmd | TM1637.a[pos])
        else:
            raise DisplayLengthError
        self.writeByte(seg)
        self.stopCond()
        self.writeDispCntrl()
```

writeDataCmd() hat eigene Start- und Stop-Conditions. Bevor die Adresse gesendet wird, muss aber eine Start-Condition eingebaut werden. Nach der Basis-Speicheradresse 0xC0 mit oderierter Digitnummer, als relativer Adressanteil, folgen das Segmentbeschreibungs-Byte, die Stop-Condition und das Display-Control-Byte. Hier ein Beispiel für ein 4-er-Display.

```
>>> from tm1637 import TM1637
>>> tm= TM1637(4,Pin(26),Pin(25))
>>> aber=bytearray(b'\x77\x7C\x79\x50')
>>> for i in range(len(aber)):
        tm.segment(aber[i],i)
```

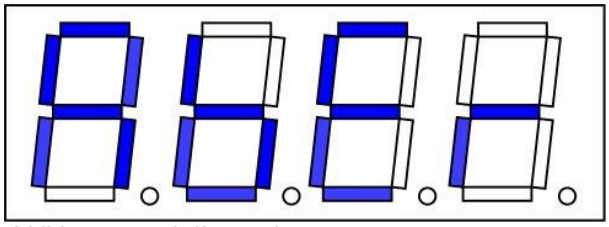

Abbildung 9: Schriftzug AbEr

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 automatisch anpassen. Genau das werden wir später tun.

```
    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 vier oder sechs Null-Bytes.

```
    def clearDisplay(self):
        segments=(bytearray(b'\x00'*self.nod),-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 gegebenenfalls der Dezimalpunkt angesteuert werden muss, falls es sich bei der Zahl um den Typ **float** handelt. Der Wert -1 kennzeichnet eine Ganzzahl. 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'*self.nod),-1)
        self.writeSegments(segments)
```

Bis zu sechs Segmentmuster ab einer vorgegebenen Position senden, das kann **writeSegments**(). Die Muster stehen im Tupel, dahinter kommt die Position **pos** ab welcher geschrieben wird. Für diesen Wert führen wir eine Plausibilitätskontrolle durch. Auch die Länge des Musterstrings wird überprüft.

Wir dröseln das Tupel **segmente** in Muster und Dezimalpunkt-Position auf. Der String darf nur so lang sein, wie ab **pos** noch Digits dafür da sind, wir testen das.

```
    def writeSegments(self, segmente, pos=0):
        s,p=segmente # p = Position des Dezimalpunkts
#         print(s,p)
        if not 0 <= pos <= self.nod-1:
            raise PositionError
        if len(s) + pos > self.nod:
            raise StringLengthError
        self.writeDataCmd()
        self.startCond()
        self.writeByte(AdrCmd | pos)
        for i in range(pos,self.nod):
            if self.nod == 6:
                c=s[TM1637.a[i]]
                if p==TM1637.a[i]:
                    c|=MSB
                self.writeByte(c)
            else:
                c=s[i]
                if p==i:
                    c|=MSB
                self.writeByte(c)
        self.stopCond()
        self.writeDispCntrl()
```

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, je nach Typ des Displays.

Das i in der for-Schleife durchläuft die physikalischen Speicherpositionen im SRAM des TM1637. Beim 6-er-Display dient es 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] beziehungsweise 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/vier Bytes kommt eine Stop-Condition und danach der Display-Control-Command.

Hier noch die Methode **number2segments**(), die Ganzzahlen oder Fließkommazahlen rechtsbündig in Segmentcode-Strings umwandelt.

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 eines 6-er-Displays oder Position 0 beim 4-er-Display. Beginnen müssen wir die Sendesequenz stets mit der relativen Speicheradresse 0, absolut 0xC0, sonst müssten wir jedem Datenbyte die Adresse vorausschicken. Wir wollen aber das Autoincrement nutzen und die Daten-Bytes in einem Abwasch senden. Den Formatstring für Ganzzahlen "{:>6}" oder "{:>4}" beziehungsweise Kommazahlen "{:>7.}" oder "{:>5.}" generieren wir in Abhängigkeit von der Displaylänge.

```
    def number2Segments(self, n, k=1):
        Int = "{:>"+str(self.nod)+"}"
        Flt = "{:>"+str(self.nod + 1)+"."
        if type(n)==int:
            s=Int.format(n)
        elif type(n)==float:
            s=Flt+str(k)+"f}"
            s=s.format(n)
        else:
            raise TypeError
        pos=s.find(".")
        if pos != -1:
            s=s.replace('.','')
            pos-=1
        if len(s)>self.nod:
            raise StringLengthError
        segments = bytearray(len(s))
        for i in range(len(s)):
            if s[i] == " ":
                segments[i]=0x00
            elif s[i] == "-":
                segments[i]=0x40
            else:
                segments[i] = TM1637.Segm[ord(s[i]) - 48]
        return (segments,pos)
```

Findet sich im Ziffernstring ein Dezimalpunkt, was nur beim 6-er Display sinnvoll ist, dann merken wir uns dessen Position und löschen ihn aus dem String. Weil der Punkt im Display der vorangehenden Ziffer zugeordnet werden muss, subtrahieren wir aber 1.

Dann erzeugen wir ein Bytearray von der Länge des Strings, das wir mit den Segmentcodes füllen. Sonderfälle wie Leerzeichen oder Minuszeichen werden berücksichtigt. Normale Ziffern liefern mit **ord**() den ASCII-Code des Zeichens. Wir erhalten den Index in die Liste **Segm**, wenn wir 48 davon subtrahieren. Den Segmentcode bauen wir ins Bytearray **segments** ein. Ist alles erledigt, geben wir das Array und die Punktposition als Tupel zurück.

Das Programm zur Uhr

Nachdem das Modul **tm1637.py** zur Steuerung der LED-Anzeige bereit ist, bauen wir daraus gleich noch eine Uhr, beziehungsweise wir modifizieren das Skript sekundenalarm.py vom letzten Mal. Die Importliste ergänzen wir um die Klasse TM1637_4.

```
# 7segment.py
#
from ds3231 import DS3231
from machine import Pin,SoftI2C, Timer,ADC
from oled import OLED
from tm1637_4 import TM1637
from sys import exit
from time import sleep
```

Wir erzeugen ein I2C-Objekt, und setzen das Alarm-Flag **alarmTrigger** zurück. Zur Ablaufsteuerung instanziieren wir ein Tasten-Objekt, **taste**.

```
i2c=SoftI2C(scl=Pin(22),sda=Pin(23),freq=100000)
alarmTrigger=False
taste=Pin(13,Pin.IN,Pin.PULL_UP)
dt=[0 for _ in range(7)]
```

Die I2C-Instanz übergeben wir an die Konstruktoren der Klassen OLED und DS3231.

```
d=OLED(i2c,heightw=32)
rtc=DS3231(i2c)
print(rtc.TellAlarmStatus())
```

Die 7-Segmentanzeige liegt an den GPIO-Pins 26 (CLK) und 25 (DIO). Mit der Digitanzahl und den entsprechenden Pin-Objekten füttern wir den Konstruktor der Klasse TM1637 und löschen gleich mal das Display.

```
tm=TM1637(4,Pin(26),Pin(25))
tm.clearDisplay()
```

Als Sensor für die Helligkeitssteuerung dient ein LDR-Modul. Bitte beachten Sie, dass die Anschlüsse für GND und +Vcc vertauscht sind (Abbildungen 3 und 4).

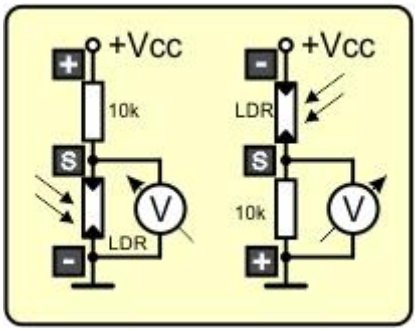

Abbildung 10: Schaltung des LDR

Mit der Beschaltung, wie sie auf dem Break Out Board vorgegeben ist (links in Abbildung 12), ergibt sich am Abgriff S eine niedrige Spannung, wenn der LDR stark beleuchtet wird, weil sein Widerstand dann sinkt. Weil sich die Spannungen an einer Serienschaltung von Widerständen wie die Widerstandswerte verhalten, fällt am LDR gegen GND also immer weniger Spannung ab, je stärker die Beleuchtung wird.

Das ist für unseren Fall suboptimal, weil wir gerade das umgekehrte Verhalten wünschen. Die Spannung an S soll steigen, wenn es heller wird, damit wir auch einen höheren Abtastwert über den ADC erhalten. Also kommt der "+"-Anschluss an GND und der "-"-Anschluss an +Vcc = 3,3V. Bei einem LDR kann man die Umpolung problemlos vornehmen, weil er keine Polung aufweist wie eine Fotodiode oder ein Fototransistor. Bei diesen beiden wäre das so nicht möglich.

Zum Abtasten der Spannung an S verbinden wir den Anschluss mit GPIO36 (VP) am ESP32. Für diesen Anschluss generieren wir ein ADC-Objekt und passen es an die Spannung von 3,3V an, indem wir den Abschwächer mit 11db wählen. Es genügt auch die kleinste Auflösung mit 9 Bit (3,3V -> 512 counts).

```
ldr=ADC(Pin(36))
ldr.atten(ADC.ATTN_11DB) # 3
ldr.width(ADC.WIDTH_9BIT)
```

Das Kernstück der Anzeigeeinheit ist die Funktion **timeOutput**(). Damit dem Hauptprogramm der Timestamp, der vom DS3231 abgefragt wird, auch zur Verfügung steht, deklarieren wir die Variable **dt** global. Das bedingt, dass sie bereits im Vorfeld im Hauptprogramm bekannt gemacht werden muss (siehe oben).

Wir holen einen Timestamp ab, weisen die Liste der Variablen **dt** zu und lassen sie in REPL ausgeben. Den Stunden- und Minutenwert picken wir heraus und schaufeln die Werte nach **hour** und **minute**.

```
def timeOutput():
    global dt
    dt=rtc.DateTime()
    print(dt)
    hour=dt[4]
    minute=dt[5]
```

Für die Anzeige müssen die Dezimalwerte in Ziffern ausgedröselt werden. Das machen wir wieder durch die Ganzzahlteilung mit Rest. Die Zehner- und Einer-Ziffern nehmen wir als Index in die Liste der zugehörigen Segmentcodes aus **Segm**. Bei den Stunden-Einern oderieren wir das MSB dazu und schalten damit den Doppelpunkt ein, der mit diesem Digit verknüpft ist. Die Segmentcodes fassen wir in der Liste **zeit** zusammen.

```
    zeit=[tm.Segm[hour//10],tm.Segm[hour%10]|0x80,
          tm.Segm[minute//10],tm.Segm[minute%10],]
```

Die for-Schleife liest die Liste aus und verfrachtet die Codes mit Hilfe der Methode **segment**() an die richtige Stelle im 4-er-Display.

```
    for i in range(4):
        tm.segment(zeit[i],i)
```

Die Funktion **alarmCallback**() und die damit zusammenhängende IRQ-Verwaltung kennen wir bereits aus der vorigen Folge.

```
def alarmCallback(pin):
    global alarmTrigger
    alarmTrigger=True

rtcIRQ=Pin(32, Pin.IN)
rtcIRQ.irq(handler=alarmCallback, trigger=Pin.IRQ_FALLING)
```

Wir lassen uns die aktuelle Zeit anzeigen und aktivieren den Alarm1 zu jeder vollen Minute. Alarm2 schalten wir vorerst aus und setzen die Alarm-Flags beider Alarme zurück.

```
timeOutput()
rtc.Alarm1(0,0,0,0,DS3231.SekundenAlarm) # zur vollen Minute
rtc.AlarmAus(2)
rtc.ClearAlarm(1)
rtc.ClearAlarm(2)
```

In der Hauptschleife lesen wir den Beleuchtungsstand ein. Die Werte mit dem maximalen ADC-Wert von 511 müssen wir auf den Bereich 0…7 reduzieren. Damit stellen wir die Helligkeit des Displays nach, das im Dunklen nur schwach leuchten soll, bei großer Umgebungshelligkeit aber dafür stärker.

```
while 1:
    dim=ldr.read()//64
    tm.kontrast(dim)
```

Bei gesetztem Alarm-Trigger muss die Uhrzeitanzeige aktualisiert und gegebenenfalls ein Weckalarm behandelt werden. Der Trigger wird in jedem Fall zurückgesetzt, ebenfalls die Alarmflags.

```
    if alarmTrigger:
        status=rtc.TellAlarmStatus()
        alarmTrigger=False
        if status & 0x01:
            rtc.ClearAlarm(1)
            timeOutput()
        if status & 0x02:
            rtc.ClearAlarm(2)
```

Für einen sauberen Ausstieg aus dem Programm mit geflissentlichem Aufräumen wird die Taste abgefragt. Ist sie gedrückt, deaktivieren wir den **rtcIRQ**. Damit reagiert der ESP32 nicht mehr automatisch auf Pegelwechsel an GPIO32. **exit**() beendet das Programm.

```
    if taste.value() == 0:
        rtcIRQ.irq(handler=None)
        exit()
```

Was tut man nun, wenn zum Stellen der Uhr keine andere auf weiter Flur zu sehen ist und wenn auch kein Zugriff auf einen Zeitserver im Web zur Verfügung steht?

Was es geschlagen hat, kann unserem ESP32 dann ein Langwellensender sagen, der bei Mainflingen bei Frankfurt steht. Ein kleiner DCF77-Empfänger wird uns die Zeitzeichen liefern, mit denen wir unsere Uhr genau mit der gesetzlichen Zeit (MEZ) synchronisieren können.

Wie das geht verrate ich in der nächsten Folge.

Bis dann!