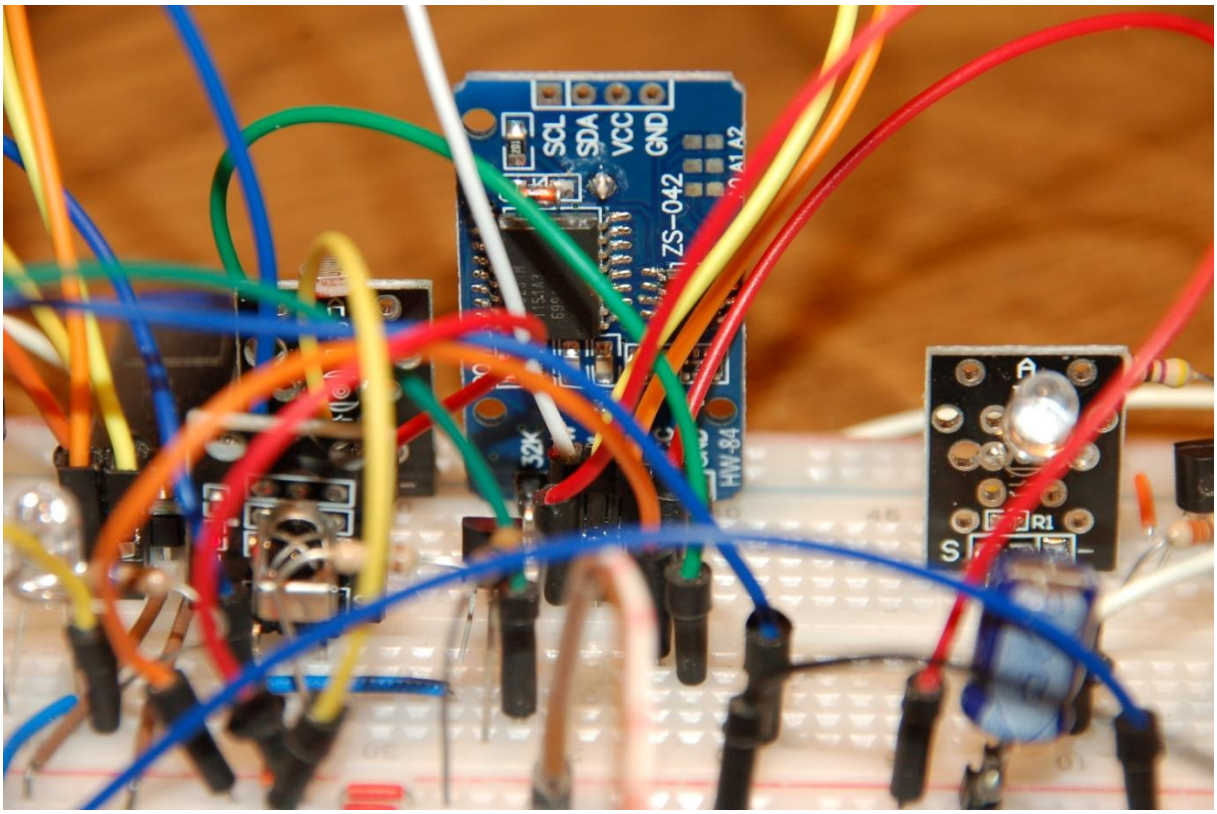

Schaltungsdetail mit DS3231

Diesen Beitrag gibt es auch als PDF-Dokument.

Das Ziel, welches wir anstreben ist ein Wecker, der, anstatt zu quengeln, ein Gerät, ein Radio oder die Stereoanlage, über die Infrarot-Fernsteuerung einschalten kann. Den ersten Teil des Weges haben wir schon geschafft, die RC (Remote Control) auszulesen und einen eigenen IR-Sender zu realisieren. Die Möglichkeit eine PS/2-Tastatur an den ESP32 anzuschließen, erlaubt dabei das Auslesen der RC5-Steuerung, ohne dass ein PC angeschlossen sein muss.

Der ESP32 besitzt nun zwar selber eine RTC (Real Time Clock), die aber zwei Unzulänglichkeiten aufweist. Erstens ist die Ganggenauigkeit nicht überzeugend. Laut MicroPython-Dokumentation zum ESP32 sollte zweitens der Controller mit einer Alarm-Methode ausgerüstet sein. Faktisch bietet MicroPython auf dem ESP32 aber keine solche Alarmfunktion an, die aber für einen Wecker gerade entscheidend ist.

```
>>> from machine import RTC
>>> RTC
<class 'RTC'>
>>> rtc=RTC()
>>> rtc.alarm(0,1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'RTC' object has no attribute 'alarm'
```

Infolgedessen müssen wir diesen Mangel beheben. Das passiert in dieser Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 und das RTC-Modul mit dem DS3231

Die bisherigen Bauanleitungen sind, ebenso wie die Programmteile und Module, in den oben verlinken PDF-Dokumenten enthalten (1, 2, 3).

Was ich über die Ganggenauigkeit der ESP-internen RTCs herausgefunden habe, habe ich in einem früheren Beitrag schon weitergegeben. In jenem Post geht es neben den RTC-Eigenschaften eines ESP8266 D1 mini auch um den Einsatz einer externen RTC in Form eines DS1302-Moduls. Dieser Chip übertrifft die ESP-eigenen Systeme an Genauigkeit, bietet aber leider auch keinen Alarmmodus. Ein Einsatz dieses Moduls hätte also zur Folge, dass der Controller diese Funktion selbst übernehmen müsste. Weil mir das zu aufwendig ist, habe ich mich für ein DS3231-Modul entschieden. Das übertrifft in Sachen Genauigkeit auch noch den DS1302 und besitzt sogar zwei Alarm-Timer, was mir für dieses Projekt recht gut in dem Kram passt. Außerdem bietet der DS3231 den Vorteil, dass er über den I2C-Bus angesprochen werden kann. Für den DS1302 müsste man drei weitere Busleitungen bereitstellen. Ebenfalls entscheidend für die Wahl des externen Moduls ist die Batterie-Pufferung. Selbst bei Stromausfall tickt die Uhr des DS3231 zuverlässig weiter, sogar temperaturstabilisiert. In diesem Beitrag werden wir uns diesen Baustein und dessen Programmierung mal ganz genau anschauen.

Das entstandene MicroPython-Modul ds3231.py besprechen wir später, jetzt werfen wir erst einmal einen Blick auf die Hardware, die sich bisher angesammelt hat und was neu hinzukommt.

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
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

Sie haben Recht, neu ist nur der DS3231. Betrachten wir als Erstes das speicherbasierte Innenleben des DS3231. Im Datenblatt des Herstellers maxim integrated (vorm. Dallas) finden wir dazu eine Tabelle, welche die vorhandenen Register auflistet. Eine Reihe von Registern halten die Zeitdaten vor (0x00 bis 0x06), andere die Alarmdaten (0x07 bis 0x0D) und zwei weitere dienen der Verwaltung (0x0E, 0x0F). Die Register 0x11 und 0x12 enthalten High- und Low-Byte der internen Temperaturmessung.

ADDRESS	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0 LSB	FUNCTION	RANGE
00h	0	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Hours	1–12 + $\overline{AM}$/PM 00–23
03h	0	0	0	0	0	Day			Day	1–7
04h	0	0	10 Date		Date				Date	01–31
05h	Century	0	0	10 Month	Month				Month/ Century	01–12 + Century
06h	10 Year				Year				Year	00–99
07h	A1M1	10 Seconds			Seconds				Alarm 1 Seconds	00–59
08h	A1M2	10 Minutes			Minutes				Alarm 1 Minutes	00–59
09h	A1M3	12/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + $\overline{AM}$/PM 00–23
0Ah	A1M4	DY/$\overline{DT}$	10 Date		Day				Alarm 1 Day	1–7
					Date				Alarm 1 Date	1–31
0Bh	A2M2	10 Minutes			Minutes				Alarm 2 Minutes	00–59
0Ch	A2M3	12/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + $\overline{AM}$/PM 00–23
0Dh	A2M4	DY/$\overline{DT}$	10 Date		Day				Alarm 2 Day	1–7
					Date				Alarm 2 Date	1–31
0Eh	$\overline{EOSC}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE	Control	—
0Fh	OSF	0	0	0	EN32kHz	BSY	A2F	A1F	Control/Status	—
10h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	Aging Offset	—
11h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	MSB of Temp	—
12h	DATA	DATA	0	0	0	0	0	0	LSB of Temp	—

Tabelle 1: ds3231-Register-Map

Die Zeit- und Alarmdaten sind BCD-codiert (Binary Coded Decimal). Zudem enthält bei den Alarmregistern das Bit 7 die Information in welchen Intervallen ein Alarm ausgelöst werden soll, jede Sekunde, Minute, Stunde oder an welchem Tag.

Die Schaltung

Das DS3231-Modul besitzt neben dem I2C-Bus einen separaten Ausgang, SQW. An diesem Pin können wahlweise zwei Signale genutzt werden. Entweder dient der Anschluss zur Ausgabe eines Sekundentakts oder als IRQ-Leitung. Wir werden letztere Möglichkeit nutzen, um den ESP32 zu benachrichtigen, wenn ein Alarmereignis vorliegt. Bis dahin kann der Controller seinen eignen Kram erledigen.

Diese Leitung liegt normalerweise auf HIGH-Pegel (3,3V) und geht nach LOW (0V), wenn Alarmzeit und Uhrzeit übereinstimmen und weitere Bedingungen erfüllt sind. Wir kommen bei der Programmierung darauf zurück.

Die ISR muss dann neben dem Auslösen der entsprechenden Aktion den Pegel auf der Leitung wieder auf HIGH setzen. Dafür gibt es eine geeignete Methode (DS3231.ClearAlarm()) in der Klasse DS3231.

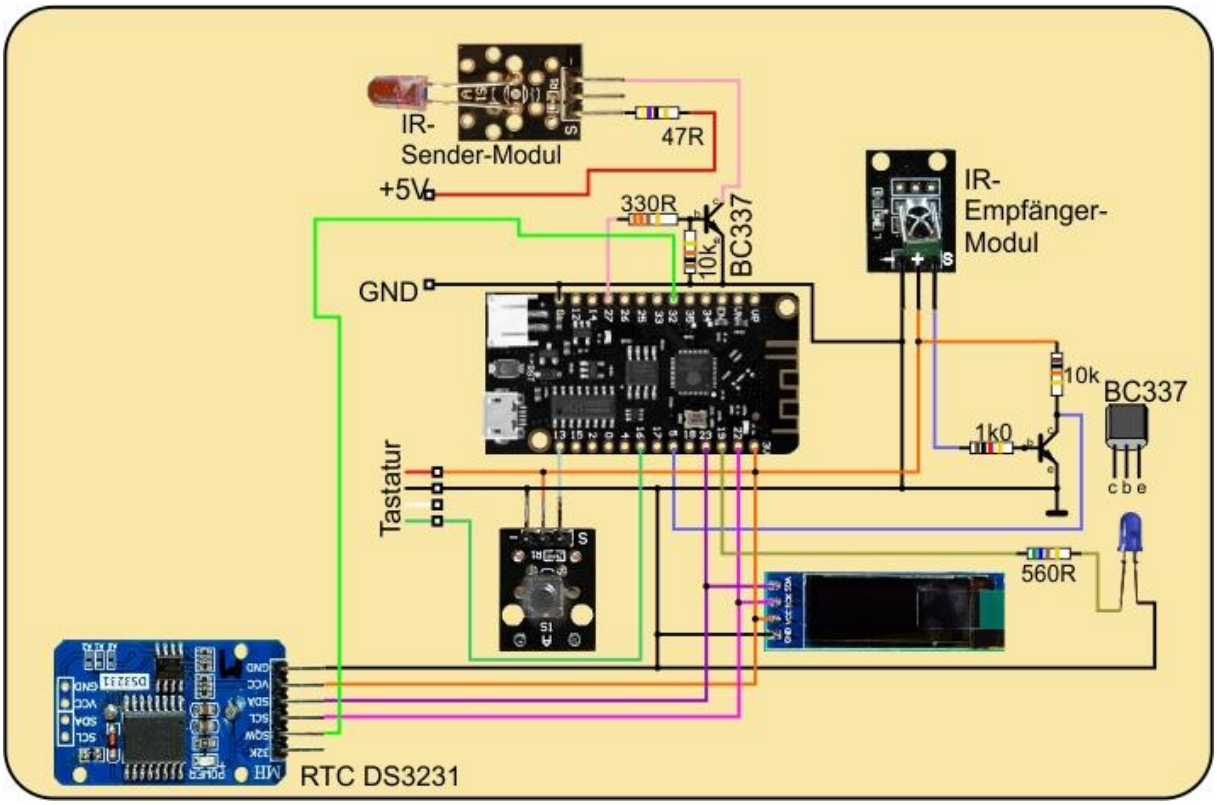

Abbildung 1: Eine genauere Uhr mit dem DS3231 - batteriegepuffert

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
SALEAE – Logic-Analyzer-Software (64 Bit) für Windows 8, 10, 11

Verwendete Firmware für einen ESP32:

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

Verwendete Firmware für einen ESP8266:

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

Die MicroPython-Programme zum Projekt:

ds3231.py Treiber-Modul
oled.py: OLED-API
ssd1306.py: OLED-Hardware-Treiber
sync_it.py: Demo-Programm zum Testen
sekundenalarm.py: Demoprogramm für eine 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 05.02.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Die Klasse DS3231

Im Zusammenhang mit dem DS3231-RTC-Modul ist es erwähnenswert, dass auf dem BOB (Break Out Board) neben dem RTC-Chip auch noch ein EEPROM mit 4KB Speicherplatz wohnt. Das ist auch der Grund, weshalb sich das Board auf dem I2C-Bus mit zwei Geräteadressen meldet.

>>> from machine import Pin, SoftI2C
>>> i2c=SoftI2C(scl=Pin(22),sda=Pin(23),freq=100000)
>>> i2c.scan()
[60, 87, 104]

60 = 0x3C: OLED-Display
87 = 0x57: EEPROM
104 = 0x68: DS3231

Das EEPROM lässt sich bei Stromausfällen nutzen, um zum Beispiel Konfigurationsdaten zwischenzulagern. Das könnte vielleicht eine Liste von Schaltzeiten sein.

Das Modul **ds3231.py** berücksichtigt dieses Feature in Form der Klasse **AT24C32**, auf die ich hier aber nicht weiter eingehe, weil sie in dieser Folge nicht zum Einsatz kommt.

Wenden wir uns also der Klasse **DS3231** zu. Mit der Deklaration der Klasse werden die Register-Adressen des DS3231 entsprechenden Klassen-Attributen zugeordnet. Die Bezeichner ergeben ein für Menschen lesbareres Programm.

```
class DS3231():
    hwadr    = const(0x68)   # 7-Bit-Adresse
    RegSec   = const(0x00)
    RegMin   = const(0x01)
    RegHor   = const(0x02)
    RegDow   = const(0x03)
    RegDay   = const(0x04)
    RegMon   = const(0x05)
    RegYea   = const(0x06)
    RegA1sec = const(0x07)
    RegA1min = const(0x08)
    RegA1hor = const(0x09)
    RegA1day = const(0x0A)
```

```
    RegA2min = const(0x0B)
    RegA2hor = const(0x0C)
    RegA2day = const(0x0D)
    RegCtl   = const(0x0E)
    RegSta   = const(0x0F)
    RegAgoff = const(0x10)
    Regtemp  = const(0x11)
```

Weitere Konstanten benennen die Intervalle, in denen ein Alarm erfolgt.

```
    jedeSekunde     = const(1) # A1 Jede Sekunde
    jedeMinute      = const(2) # A2 Jede Min. zur 0. Sekunde
    SekundenAlarm   = const(3) # A1 die Sekunden passen
    MinutenAlarm    = const(4) # Minuten und Sekunden p.
    StundenAlarm    = const(5) # hor, min und sec passen
    DatumsAlarm     = const(6) # Monatstag u. Zeit passen
    WochentagsAlarm = const(7) # Wochentag u. Zeit passen
    Maske           = [0xFE,0xFD]
```

Die Liste **dow** enthält die Klartext-Namen der Wochentage. Weil der DS3231 von Montag = 1 bis Sonntag = 7 zählt, ist das erste Listenelement mit None belegt. Damit kann der Wochentagswert des DS3231 direkt als Index in die Liste der Tagesnamen hergenommen werden.

```
    dow=[None,"Montag","Dienstag","Mittwoch","Donnerstag",\
         "Freitag","Samstag","Sonntag"]
```

Der Konstruktor **DS3231()** in Form der Methode **__init__()** nimmt ein I2C-Objekt, welches im aufrufenden Programm deklariert werden muss. Mit 0x4C = 0b01001100 schalten wir den Ausgang **SQW** als Interrupt-Ausgang, erlauben aber im Moment noch keinen Alarm. Die Bedeutung der einzelnen Bits erkläre ich später. Der Wert wird in das Kontroll-Register **RegCtl** geschrieben.

```
    def __init__(self, i2c):
        self.i2c = i2c
        self.writeReg(RegCtl, 0x4C)
        print("DS3231 bereit")
        #BBSQW + RS1 + INTCN  ->>> IRQ auf SQW-Pin
```

Es folgen eine Reihe von ähnlich konstruierten Methoden, die entweder das entsprechende Zeitelement abrufen oder setzen. Wir schauen uns dazu exemplarisch die Methode **Second**() an. Wird beim Aufruf kein Argument für den Parameter **second** übergeben, erhält dieser den Defaultwert None. Daraufhin wird das Register **RegSec** = 0x00 ausgelesen. Die Methode **bcd2dec()** formt den BCD-Code in einen Dezimalwert um, den wir zurückgeben.

```
    def Second(self, second = None):
        if second == None:
            return self.bcd2dec(self.readReg(RegSec))
        else:
            self.writeReg(RegSec, \
                          self.dec2bcd(second)&0x7F)
```

Ein übergebener Dezimalwert wird durch **dec2bcd()** ins BCD-Format übersetzt und mit 0x7F undiert. Das stellt sicher, dass Bit7 gelöscht ist, bevor wir das Ergebnis in das Register des DS3231 schreiben.

Die Methoden für Minute, Stunde, Wochentag, Monatsdatum, Monat und Jahr arbeiten analog, wobei es noch einige Spezialfälle zu beachten gilt.

Das Stunden-Register 0x02 legt mit Bit6 fest, ob im 12- oder 24-Stunden-Rhythmus gezählt wird. Weil wir im 24-Stunden-Modus arbeiten, wird Bit6 auf 0 gesetzt. Bit5 ist dann das "20"-er-Bit der Uhrzeit.

Bit7 des Monats-Registers wird getoggelt, wenn die Jahreszählung von 99 nach 0 wechselt.

Die Methode **Date**() liefert oder setzt, analog zu den Methoden **Year**(), **Month**() und **Day**(), ein Tagedatum, indem diese Methoden aufgerufen werden. Die übergebene vierstellige Jahreszahl wird modulo 100 auf Zehner- und Einerziffer gekürzt. Der Monat wird beim Setzen auf 1 … 12 eingegrenzt, der Tag auf 1 … 31. Eine sicherere Plausibilitätskontrolle könnte natürlich auch über ein range-Konstrukt erfolgen, bläht das Ganze aber unnötig auf.

```
if dat[1] in range(1,13):
    self.Month(dat[1])
```

Time() arbeitet analog in den Bereichen Stunde, Minute und Sekunde.

Mit der Methode **DateTime()** wird eine Liste der Datums- und Zeitwerte abgefragt, wenn kein Argument übergeben wird. Die Rückgabe erfolgt in einer Liste von der Form [Jahr, Monat, Tag, Wochentag, Stunde, Minute, Sekunde]. Andernfalls erwartet die Methode ein 7-er-Tupel oder eine Liste derselben Struktur. Die einzelnen Felder des Tupels übertragen die entsprechenden Methoden an den DS3231.

```
    def DateTime(self, dat = None):
        if dat == None:
            return self.Date() + [self.Weekday()] + \
                   self.Time()
        else:
            self.Year(dat[0])
            self.Month(dat[1])
            self.Day(dat[2])
            self.Weekday(dat[3])
            self.Hour(dat[4])
            self.Minute(dat[5])
            self.Second(dat[6])
```

Um eine **Zeitdauer** der Form [Stunde, Minute, Sekunde] zu einem **Zeitpunkt** zu addieren, benutzen wir die Methode **AddDelay2Time**(). Das Zeitintervall wird als Liste oder Tupel an den Parameter **delay** übergeben. Die Addition erfolgt in 60-er und 24-er Ringen. Die Zeitpartikel bleiben also stets im gültigen Bereich 0...59 und 0…23. Ein Übertrag auf Tag, Monat und Jahr erfolgt hier nicht, wäre aber grundsätzlich möglich – mit stark erhöhtem Auswand. Denken Sie an die verschiedenen Monatslängen, Schaltsekunden und Schaltjahre.

```
    def AddDelay2Time(self,delay): # delay=[hor,min,sec]
        zeit=self.Time()
        s=delay[2]+zeit[2]
        zeit[2]=s % 60
        m= s // 60 + delay[1] + zeit[1]
        zeit[1]=m % 60
        h= m // 60 + delay[0] + zeit[0]
        zeit[0] = h % 24
        d= h // 24
        return zeit
```

Den Sekundentakt am Pin **SQW** können wir mit **Sekundentakt**() ein- oder ausschalten, indem wir 1 oder 0 übergeben. Dadurch wird das Bit2 **INTCN** im Control-Register gelöscht oder gesetzt. Das Löschen machen wir durch Undieren mit der Maske 0xFB = 0b11111011. Durch Oderieren mit 0x04 setzen wir das Bit und schalten damit den Sekunden-Takt ab und die Interrupt-Funktion ein. Voraussetzung für beide Funktionen ist das gesetzte Bit6 (**BBSQW**) in **RegCtl**. Das hat bereits der Konstruktor eines DS3231-Objekts erledigt. Durch Anwendung der Masken wird der Zustand von BBSQW nicht verändert.

```
    def Sekundentakt(self, onoff): # 1s-Takt an -INT/SQW
        ctrl = self.readReg(RegCtl)
        if onoff == 1:
            ctrl &= 0xFB
        else:
            ctrl |= 0x04
        self.writeReg(RegCtl, ctrl)
```

Die aktuelle Alarmzeit können wir mit **GetAlarmTime**() erfahren. Die Nummer des Alarm-Timers wird im Parameter **alarm** übergeben. Je nachdem wird der entsprechende Registerbereich ausgelesen und als Liste zurückgegeben. Eine ungültige Timer-Nummer führt zur Rückgabe von [-1,-1,-1].

```
    def GetAlarmTime(self, alarm):
        if alarm == 1:
            sec=self.bcd2dec(self.readReg(RegA1sec)&0x7F)
            mit=self.bcd2dec(self.readReg(RegA1min)&0x7F)
            hor=self.bcd2dec(self.readReg(RegA1hor)&0x3F)
        elif alarm == 2:
            sec=0
            mit=self.bcd2dec(self.readReg(RegA2min)&0x7F)
            hor=self.bcd2dec(self.readReg(RegA2hor)&0x3F)
        else:
            sec,mit,hor=0,0,0
        return [hor,mit,sec]
```

Der Name der Methode **SetNewAlarmTime**() ist Programm. Die Routine nimmt für Alarm1 ein Tupel oder eine Liste der Form (Stunde, Minute, Sekunde) und die Timernummer 1. Das Sekunden-Register wird gelesen, um das aktuelle Bit7 zu konservieren, die restlichen Bits werden gelöscht. Die übergebene Sekundenanzahl wird in BCD-Code übersetzt und Bit7 gelöscht. Durch Oderieren mit **z** entsteht der neue Registerwert, der zum DS3231 geschickt wird. Beim Minuten-Register läuft es analog. Beim Einlesen des Stunden-Alarm-Registers werden die beiden obersten Bits maskiert und gemerkt. Dafür werden diese Bits vorsichtshalber beim BCD-Wert vor dem Oderieren ausgeblendet. Bei Alarm2 gibt es keine Sekundeneinstellung. Sonst läuft es wie bei Alarm1.

```
    def SetNewAlarmTime(self,zeit,alarm):
        if alarm==1:
            z=(self.readReg(RegA1sec)&0x80)
            self.writeReg(RegA1sec,\
                          z|(self.dec2bcd(zeit[2])&0x7F))
            z=(self.readReg(RegA1min)&0x80)
            self.writeReg(RegA1min,\
                          z|(self.dec2bcd(zeit[1])&0x7F))
            z=(self.readReg(RegA1hor)&0xC0)
            self.writeReg(RegA1hor,\
                          z|(self.dec2bcd(zeit[0])&0x3F))
        elif alarm==2:
            z=(self.readReg(RegA2min)&0x80)
            self.writeReg(RegA2min,\
                          z|(self.dec2bcd(zeit[1])&0x7F))
            z=(self.readReg(RegA2hor)&0xC0)
            self.writeReg(RegA2hor,\
                          z|(self.dec2bcd(zeit[0])&0x3F))
```

Die Interrupt-Enable-Bits **A1IE** (Bit0) und **A2IE** (Bit1) im Control-Register **RegCtl** = 0x0E erlauben, wenn sie auf 1 gesetzt sind, das Auftreten von Alarmen. Dazu muss auch Bit2 (**INTCN**) in **RegCtl** gesetzt sein. Fällt der Vergleich zwischen den Zeit- und Alarm-Registern positiv aus, wird der **SQW**-Ausgang des DS3231 auf LOW gelegt und das entsprechende Status-Bit in **RegSta** = 0x0F gesetzt.

Die Methode **AlarmAus**() stellt das Interrupt-Enable-Bit des Alarms auf 0, dessen Nummer an die Routine übergeben wird und verhindert so grundsätzlich das Auftreten des Interrupts. Nur wenn die Nummer 1 oder 2 übergeben wird, wird das entsprechende AIE-Bit zurückgesetzt, sonst macht die Routine nichts. Wir lesen den Registerinhalt von **RegCtl** wie üblich ein und undieren mit dem jeweiligen Masken-Byte aus der Liste **Maske**.

```
    Maske           = [0xFE,0xFD]
```

```
    def AlarmAus(self,alarm):
        try:
            maske=Maske[alarm-1]
        except:
            return
        ctrl = self.readReg(RegCtl)
        self.writeReg(RegCtl, ctrl & maske)
```

Alarm1 wird zugelassen, wenn wir mit **Alarm1**() Werte für Tag, Stunde, Minute und Sekunde übergeben. Der Parameter **mode** erhält einen der eingangs definierten Konstantenwerte. Der Wert entscheidet darüber, welche Maskenbits (Bit7) in den Alarmzeit-Registern gesetzt oder gelöscht werden. Diese Maskenbits bestimmen das Zeitintervall für das Auftreten von Interrupts. Dazu gleich mehr, zuvor ein Blick auf die Routine **Alarm1**() selbst.

```
    jedeSekunde     = const(1) # A1 Jede Sekunde
    jedeMinute      = const(2) # A2 Jede Min. zur 0. Sekunde
    SekundenAlarm   = const(3) # A1 die Sekunden passen
    MinutenAlarm    = const(4) #  Minuten und Sekunden p.
    StundenAlarm    = const(5) #  hor, min und sec passen
    DatumsAlarm     = const(6) #  Monatstag u. Zeit passen
    WochentagsAlarm = const(7) #  Wochentag u. Zeit passen
```

```
    def Alarm1(self, day, hour, minute, second, mode):
        ctrl = self.readReg(RegCtl)
        ctrl |= 0x45 # BBSQW + INTCN + A1IE setzen
        self.writeReg(RegCtl, ctrl)
```

Um den Alarm1 zu aktivieren, setzen wir erst einmal die richtigen Steuerbits. **BBSQW** muss zusammen mit **INTCN** auf 1 stehen, damit durch Setzen von **A1IE** ein Interrupt am **SQW**-Ausgang zugelassen wird.

RegCtl = 0x0E

$\overline{EOSC}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE
	1				1		1
Bit 7	6	5	4	3	2	1	0

Abbildung 2: Konfiguration für Alarm1

Wir lesen **RegCtl** = 0x0E ein, oderieren da drauf unser Konfig-Byte 0b01000101 = 0x45 und senden das Ergebnis an den DS3231. Dann setzen wir alle vier Maskenbits auf 1. Dass das in einer Zeile geschehen kann, dafür sorgt ein wenig sytaktischer

Zucker (sytactic suggar). Die vier Bytes rechts des Zuweisungs-Operators "=" werden im Hintergrund transparent gepackt und auf die vier Variablen wieder entpackt. Was dahintersteckt, könnte man so deuten:

>>> tup=0x80,0x80,0x80,0x80
>>> tup
(128, 128, 128, 128)
>>> m1,m2,m3,m4=tup
>>> print(m1,m2,m3,m4)
128 128 128 128

Nun fragen wir nacheinander den mode-Parameter ab und passen die Masken-Bytes entsprechend an. Den Zusammenhang ersehen wir aus Tabelle 2, die ich dem Datenblatt entnommen habe.

```
        M1,M2,M3,M4 = 0x80,0x80,0x80,0x80 # Alarm jede Sekunde
        DT = 0 # Datum muss passen, nicht Wochentag
        if mode == DS3231.jedeSekunde:
            pass
        elif mode == DS3231.SekundenAlarm: # Sek. passen
(minütlich!)
            M1 = 0
        elif mode == DS3231.MinutenAlarm:# Min. + sek. passen
(stündlich)
            M1,M2 = 0,0
        elif mode == DS3231.StundenAlarm:# Hor+Min+Sek p.
(täglich)
            M1,M2,M3 = 0,0,0
        else: # Monatstag + Zeit passen
            M1,M2,M3,M4 = 0,0,0,0
            if mode == DS3231.WochentagsAlarm:
                DT = 0x40
```

Bei der Tagesangabe müssen wir unterscheiden zwischen einem Monatsdatum (DY/-DT = 0) und einem Wochentag (DY/-DT = 1). Dann werden die übergebenen Tages- und Zeitwerte an den DS3231 geschickt.

```
        self.writeReg(RegA1sec,\
                      (self.dec2bcd(second)&0x7F)|M1)
        self.writeReg(RegA1min,\
                      (self.dec2bcd(minute)&0x7F)|M2)
        self.writeReg(RegA1hor,\
                      (self.dec2bcd(hour)&0x3F)|M3)
        self.writeReg(RegA1day,\
                      (self.dec2bcd(day)&0x3F)|M4|DT)
```

DY/$\overline{DT}$	ALARM 1 REGISTER MASK BITS (BIT 7)				ALARM RATE
	A1M4	A1M3	A1M2	A1M1	
X	1	1	1	1	Alarm once per second
X	1	1	1	0	Alarm when seconds match
X	1	1	0	0	Alarm when minutes and seconds match
X	1	0	0	0	Alarm when hours, minutes, and seconds match
0	0	0	0	0	Alarm when date, hours, minutes, and seconds match
1	0	0	0	0	Alarm when day, hours, minutes, and seconds match

DY/$\overline{DT}$	ALARM 2 REGISTER MASK BITS (BIT 7)			ALARM RATE
	A2M4	A2M3	A2M2	
X	1	1	1	Alarm once per minute (00 seconds of every minute)
X	1	1	0	Alarm when minutes match
X	1	0	0	Alarm when hours and minutes match
0	0	0	0	Alarm when date, hours, and minutes match
1	0	0	0	Alarm when day, hours, and minutes match

Tabelle 2: Steuerung der Alarm-Intervalle

Wenn die Sekunden von Zeit- und Alarmeinheit übereinstimmen (1,1,1,0), das passiert jede Minute einmal, wird der Minutenalarm ausgelöst. Jeden Tag einmal passen Stunden, Minuten und Sekunden zusammen (1,0,0,0) und einmal im Jahr stimmt auch noch das Datum überein, wenn DY/-DT = 0 ist. Einmal die Woche wird Alarm ausgelöst, wenn DY/-DT = 1 ist und der Wochentag passt.

In genau derselben Weise arbeitet Alarm2(), nur dass es hier keine Sekunden zu überprüfen gibt.

Bitte beachten Sie, dass ein Alarm nur durch den Aufruf der entsprechenden Routine Alarm1 oder Alarm2 **aktiviert** werden kann. Dadurch werden die Zeit und der Modus gesetzt sowie der Alarm zugelassen. Erst danach kann mit **SetNewAlarmTime**() eine neue Alarmzeit unter Beibehaltung des zuvor eingestellten Modus gesetzt werden.

Um einen ausgelösten Alarm zu entschärfen, drücken wir normalerweise die Taste am Wecker. Hier geschieht das, indem wir die Methode **ClearAlarm**() rufen. Mit der übergebenen Alarmnummer wird eine Maske erzeugt, 0xFE für Alarm1 und 0xFD für Alarm2. Wir lesen das Status -Register **RegSta** = 0x0F und löschen darin das Status-Bit durch Undieren mit der Maske.

Bitte beachten Sie, dass ein Alarm nur dann erneut ausgelöst werden kann, wenn nach der Aktivierung das Status-Bit zurückgesetzt wird. Dadurch geht die Leitung SQW von 0V wieder auf 3,3V.

RegSta = 0x0F

OSF				EN32kHz	BYS	A2F	A1F
x	0	0	0	x	x	x	**1**
Bit 7	6	5	4	3	2	1	0

Maske = 0xFE

1	1	1	1	1	1	1	**0**

RegSta & Maske

x	0	0	0	x	x	x	**0**

Abbildung 3: Beenden von Alarm1

Viele elektronische Wecker kennen wiederholtes Alarmieren, nach einem bestimmten Zeitintervall. Diese Funktion lässt sich mit **AddDelay2Alarm**() flexibel realisieren. Die Routine arbeitet analog zu **AddDelay2Time**(), nur dass statt der Zeit-Register die Alarmzeit-Register gelesen und hier auch gleich neu beschrieben werden. Der neue Zeitpunkt wird auch zurückgegeben.

Die Berechnung erfolgt im Ring 24h,60m,60s, das heißt, die Zeitberechnung erfolgt mit Übertrag auf Minuten und Stunden jedoch ohne Berücksichtigung des Übertrags auf den neuen Tag und ist daher in Alarm1 nur für die Modi 3 bis 5 und im Alarm2 nur für die Modi 4 und 5 brauchbar.

```
    def AddDelay2Alarm(self,delay,alarm): #
delay=[hor,min,sec]
        zeit=self.GetAlarmTime(alarm)
        s=delay[2]+zeit[2]
        zeit[2]=s % 60
        m= s // 60 + delay[1] + zeit[1]
        zeit[1]=m % 60
        h= m // 60 + delay[0] + zeit[0]
        zeit[0] = h % 24
        # d= h // 24
        if alarm == 1:
            self.SetNewAlarmTime(zeit, alarm)
        elif alarm == 2:
            self.SetNewAlarmTime(zeit[0:2], alarm)
        return zeit
```

Das Abfragen der Status-Bits kann mit **TellAlarmStatus**() erfolgen. Zurückgegeben wird der Bytewert der beiden Bits, 0, 1, 2, oder 3.

```
    def TellAlarmStatus(self):
        status=self.readReg(RegSta) & 0x03
        return status
```

Ähnlich arbeitet **TellAlarmEnabled**(). Die Routine meldet uns, welcher Alarm generell scharfgeschaltet ist.

```
    def TellAlarmEnabled(self):
        status=self.readReg(RegCtl) & 0x03
        return status
```

Wurde ein Alarm mit **AlarmAus**() deaktiviert, dann kann er, unter Beibehaltung von Zeit und Modus, mittels **EnableAlarm**() wieder zugelassen werden. Die übergebene Alarmnummer wird auf Gültigkeit hin überprüft. Dann wird im eingelesenen Status-Byte das entsprechende Bit gesetzt und das Ganze wieder zurückgeschrieben.

```
    def EnableAlarm(self,number):
        if number in range (1,3):
            ctrl = self.readReg(RegCtl)
            ctrl |= number
            self.writeReg(RegCtl, ctrl)
```

Das DS3231-Modul enthält auch eine Temperatur-Mess-Einheit, die alle 64 Sekunden einen Wert an die Zeiteinheit liefert. Dadurch wird der Oszillator getrimmt und damit dessen Temperaturdrift kompensiert. Die Messung kann auch von uns genutzt werden. Wir müssen dazu nur die beiden Temperaturregister auslesen und daraus die Temperatur im Chip zusammensetzen. Das erledigt die Methode **Temperature**().

```
    def Temperature(self):
        t1 = self.readReg(Regtemp)
        t2 = self.readReg(Regtemp + 1)
        if t1 & 0x80:
            return  (-256 + t1) - (t2 >> 6)/4)
        else:
            return t1 + (t2 >> 6)/4
```

Das MSB (**Regtemp** = 0x11) enthält den ganzzahligen Anteil des Temperaturwerts und das LSB (0x12) in den oberen beiden Bits den Nachkommaanteil. Ist im MSB (Most Significant Byte) das MSb (Most Significant Bit = Bit7) gesetzt, dann stellt der Wert in der Zweierkomplementdarstellung einen negativen Messwert dar. Den wandeln wir in das Dezimalformat durch Komplementbildung von ganzzahligem und Bruchanteil um.

Vier Service-Routinen schließen die Klasse **DS3231** ab. Hier begegnet uns **dec2bcd**(). Die Methode transformiert einen Dezimalwert von 0 bis 99 incl. ins BCD-Format. Vom Argument in **dat** bestimmen wir den Teilungsrest modulo 10, die Einer. Die Zehnerziffer liefert die Ganzzahl-Division von dat durch 10. Durch die Multiplikation mit 16 verschieben wir die Zehnerziffer in die oberen vier Bits = High Nibble. Die Einerziffer bleibt im Low Nibble. Addieren der beiden Nibble-Werte ergibt den Wert des BCD-Bytes.

>>> dat=25
>>> dat % 10
5
>>> dat // 10
2
>>> hex((dat // 10)*16)
'0x20'
>>> hex((dat // 10)*16 + dat % 10)
'0x25'

```
    def dec2bcd(self, dat):
        dat=dat%100
        return (dat//10) * 16 + (dat%10)
```

Den umgekehrten Weg geht **bcd2dec**(). Die Ganzzahldivision des Werts in **dat** durch 16 ergibt die Zehnerziffer. Ebenso gut kann man **dat** um vier Positionen nach rechts schieben. Die Einerziffer kriegen wir als Teilungsrest modulo 16 oder durch Undieren mit 0x0F.

```
    def bcd2dec(self, dat):
        return (dat//16) * 10 + (dat%16)
```

oder

```
    def bcd2dec(self, dat):
        return (dat >> 4) * 10 + (dat & 0x0F)
```

Den Schreibauftrag in ein Register erledigt die Methode **writeReg**(). Wir übergeben die Register-Adresse und das Daten-Byte. Daraus bauen wir über eine namenlose Liste ein namenloses Byte-Array zusammen, das zusammen mit der Geräteadresse des DS3231 der Routine **i2c.writeto**() übergeben wird. Dieses Vorgehen ist dadurch begründet, dass **writeto**() ein Argument erwartet, das dem Buffer-Protokoll folgt. Das Ganzzahlen-Format erfüllt das nicht, infolgedessen brauchen wir das Array.

Eine ähnliche Beschränkung gibt es bei **i2c.readfrom**(). Zuerst muss die Registeradresse gesendet werden, dann wird der Registerinhalt als ein Bytes-Objekt empfangen, das auch das Buffer-Protokoll unterstützt. Damit ein Zahlenwert zurückgegeben wird, adressieren wir in dem Bytes-Objekt das erste Element mit dem Index 0 und erhalten den Byte-Wert.

```
    def readReg(self, Reg):
        self.i2c.writeto(hwadr, bytearray([Reg]))
        return self.i2c.readfrom(hwadr, 1)[0]
```

Ein Testprogramm

Bauen wir uns ein kleines Testprogramm, das die Fähigkeiten und den Gebrauch der Klasse DS3231 demonstriert. Zunächst gilt es, die RTC zu synchronisieren. Das macht ein Aufruf von sync_it.py aus dem Editorfenster heraus. Wir starten wie üblich mit einigen Importen.

```
# sync_it.py
from ds3231 import DS3231
from machine import Pin,SoftI2C
from oled import OLED
from sys import exit
```

Ein I2C-Objekt wird instanziiert und an den Konstruktor des OLED-Objekts **d** übergeben.

```
i2c=SoftI2C(scl=Pin(22),sda=Pin(23),freq=100000)

d=OLED(i2c,heightw=32)

ds=DS3231(i2c)
```

Jetzt wartet die Eingabe auf eine Datums- und Zeitangabe in unserem bekannten Format. Wir geben einen Zeitpunkt in der unmittelbaren Zukunft ein und kurz bevor eine Vergleichsuhr diese Zeit anzeigt, drücken wir die Enter-Taste.

```
tagzeit=input("JJ,MM,TT,dow,hh,mm,0 \n-> synchronisieren mit
ENTER: ")
```

Die nächste Zeile ist erneut syntaktischer Zucker aus der MicroPython-Trick-Kiste, eine sogenannte List-Comprehension. Sie erzeugt durch Aufteilen des Eingabe-Strings eine unbenannte Liste von Teilstrings. Diese wird von der Comprehension iterativ durchlaufen, wobei aus jedem String der Quellliste eine Ganzzahl der Ziel-Liste **dt** erzeugt wird. Das ist elegant, effizient und entspricht dem folgenden Vorgehen.

```
teilListe=tagzeit.split(",")
dt=[]
for x in teilListe:
    dt.append(int(x))
```

```
dt=[int(x) for x in tagzeit.split(",")]
```

Die erzeugte Liste übergeben wir dem DS3231. Damit ist die RTC synchronisiert, wie die nachfolgenden händischen Befehlseingaben und REPL-Antworten zeigen.

```
ds.DateTime(dt)
```

>>> %Run -c $EDITOR_CONTENT
this is the constructor of OLED class
Size:128x32
DS3231 bereit
JJ,MM,TT,dow,hh,mm,0
-> synchronisieren mit ENTER: **23,8,20,7,18,18,55,0**

>>> ds.DateTime()
[2023, 8, 20, 7, 18, 19, 10]

>>> ds.dow[ds.DateTime()[3]]
'Sonntag'

Das nächste Beispiel, **sekundenalarm.py**, zeigt den Einsatz des Sekunden-Alarms des Alarm-Timers 1.

```
# sekundenalarm.py
from ds3231 import DS3231
from machine import Pin,SoftI2C, Timer
from oled import OLED
from sys import exit
from esp32 import RMT

i2c=SoftI2C(scl=Pin(22),sda=Pin(23),freq=100000)
alarmTrigger=False
taste=Pin(13,Pin.IN,Pin.PULL_UP)

d=OLED(i2c,heightw=32)

rtc=DS3231(i2c)
```

Die Funktion **alarmCallBack**() wird aufgerufen, wenn die Sekunden von Zeit- und Alarm-Timer matchen. Die Funktion braucht einen obligatorischen Parameter **pin**, der in unserem Fall vom System den Anschluss 32 zugewiesen bekommt. Die Routine kann daran erkennen an welchem Pin ein Pegelwechsel stattgefunden hat. Da eine ISR (Interrupt Service Routine) keinen Wert an ein aufrufendes Programm zurückgeben kann (an wen denn auch?) muss das Flag **alarmTrigger global** deklariert werden, sonst bekommt die Hauptschleife keine Rückmeldung. Sonst passiert hier nichts weiter, als dass dieses Flag auf **True** gesetzt wird. Eine ISR sollte so kurz wie möglich gehalten werden, kürzer geht es nicht mehr.

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

Dann legen wir den GPIO32 als IRQ-Eingang an, der mit SQW des DS3231 verbunden wird. GPIO32 wird als interruptfähiger Eingang erklärt, der auf fallende Flanken hört und beim Eintreffen einer solchen dann die Routine **alarmCallback**() aufruft.

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

Wir erklären, dass im Minutenabstand, und zwar zur Sekunde **0** ein IRQ erfolgen soll, knipsen Alarm2 aus und löschen beide Status-Flags, damit die SQW-Leitung auf 3,3V = Logisch 1 geht.

```
rtc.Alarm1(0,0,0,0,DS3231.SekundenAlarm) # jede Minute einmal
rtc.AlarmAus(2)
rtc.ClearAlarm(1)
rtc.ClearAlarm(2)
```

Bis zur ersten vollen Minute kann einige Zeit vergehen, das hängt von der Startzeit des Programms ab. Damit wir den ESP32 nicht im Nirwana wähnen, lassen wir uns am Display eine Meldung ausgeben.

```
d.clearAll(False)
d.writeAt("RTC RUNNING",0,0)
```

In der Hauptschleife wird das Flag **alarmTrigger** abgefragt. Wenn es auf **True** steht muss gehandelt werden. Wir lesen das Status-Register des DS3231 und setzen das Flag auf **False**. Dann schauen wir nach, ob der Alarm1 den Trigger gesetzt hat.

Ist das der Fall, so löschen wir das Flag im DS3231 und lesen einen Zeit-Record ein. Die Felder verteilen wir auf die drei Strings **dow**, **date** und **zeit**. Aus den Integers und den Formatanweisungen können sehr einfach passgenaue Strings für die Ausgabe erzeugt werden.

Der Format-String "{:02d}.{:02d}.{:02d}" gibt Zahlen stets zweistellig im Dezimalformat aus, getrennt durch einen ".". Ähnlich arbeitet "{:02d}:{:02d}".

Dann löschen wir verdeckt das Display und geben ebenso verdeckt Wochentag und Datum aus. Verdeckt bedeutet, die Daten werden nicht sofort zum OLED geschickt, sondern erst einmal nur in dessen Pufferspeicher auf dem ESP32. Erst nachdem auch der Zeit-String dort gelandet ist, wird der gesamte Puffer über I2C zum Display transferiert. Das unterdrückt störendes Flackern und spart Zeit, weil sonst mit jedem **writeAt**() der gesamte Puffer übertragen würde.

```
while 1:

    if alarmTrigger:
        status=rtc.TellAlarmStatus()
        alarmTrigger=False
        if status & 0x01:
            rtc.ClearAlarm(1)
            dt=rtc.DateTime()
            dow=rtc.dow[dt[3]]
            date="{:02d}.{:02d}.{:02d}".\
                  format(dt[2],dt[1],dt[0])
            zeit= "{:02d}:{:02d}".format(dt[4],dt[5])
            d.clearAll(False)
            d.writeAt(dow,0,0,False)
            d.writeAt(date,0,1,False)
            d.writeAt(zeit,0,2)
```

Nun wird mit jeder vollen Minute die Anzeige im Display aktualisiert.

Ausblick

Natürlich ist die OLED-Anzeige für eine richtige Uhr "leicht" ungeeignet. Das wollen wir in der nächsten Folge ändern. Da setzen wir ein dimmbares LED-7-Segment-Display mit 4 Digits ein. Natürlich gibt es auch dafür wieder ein Treibermodul.

Tschüss und bleiben sie dran!