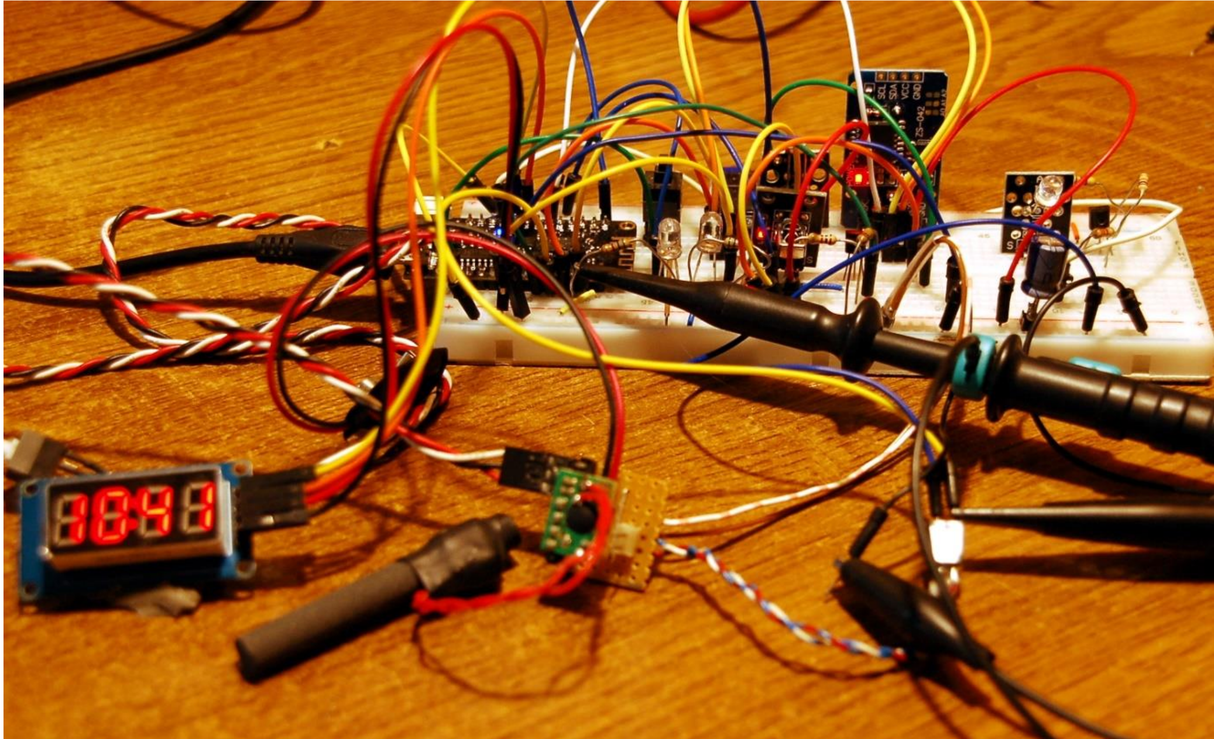

DCF77-Empfangsmodul am ESP32

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

Nachdem unsere ESP32-DS3231-Hybriduhr schon exzellent läuft, wollen wir ihr heute den Zugriff auf die amtliche Zeit in Deutschland gewähren. Ermöglicht wird das durch den Langwellensender DCF77, den die PTB (Physikalisch-Technische Bundesanstalt Braunschweig) mit Standort Mainflingen, nahe Frankfurt, betreibt.

Wir werden uns heute mit der Codierung der Zeitzeichen dieses Senders beschäftigen und daraus ein MicroPython-Modul ableiten. Damit können wir dann unsere RTC- zu einer "Atomuhr" aufmotzen. Mit unserer hochgenauen RTC (Real-Time-Clock) im DS3231 reicht es völlig, wenn wir einmal am Tag mit dem DCF77 einen Abgleich durchführen. Es würde wohl auch einmal die Woche reichen, denn nach dem letzten Test, habe ich lediglich ein Nachgehen der RTC um ca. 3 Sekunden in einem Zeitraum von ca. 10 Tagen festgestellt.

Wie wir an den DCF77 mit ESP32 und MicroPython andocken können, das schauen wir uns in dieser Folge an, aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Die Zeitzeichen des DCF77

Das Rufzeichen DCF77 setzt sich zusammen aus D für Deutschland, C für Langwellensender und F für die Nähe zu Frankfurt. 77,5kHz ist die Trägerfrequenz des Senders, der eine Leistung von 30kW über die Antenne abstrahlt.

Im Sekundentakt wird die Trägeramplitude für 100 oder 200 Millisekunden auf 15% abgesenkt. Eine 100ms-Absenkung entspricht einer logischen 0, 200ms codieren eine logische 1. Ab der 16. Sekunde enthalten die Bits Zeitinformationen. Bit 20 setzt mit einer 1 den Beginn des Zeit- und Datumcodes, für uns geht es mit Bit 21 somit richtig zur Sache.

Mit jedem Sekundentakt wird ein Bit eines BCD-Codes (Binary Coded Decimal) für die Minuten, Stunden und das Datum inclusive Wochentag übertragen. Dazwischen gibt es Prüfbits. So ein Paritätsbit ist 1, wenn die Anzahl von 1-Bits im Datenfeld ungerade ist, und 0, wenn eine gerade Anzahl von 1-en enthalten ist. Wir sprechen von gerader Parität oder even Parity. Abbildung 1 gibt einen Überblick über die Beschaffenheit eines Code-Frames.

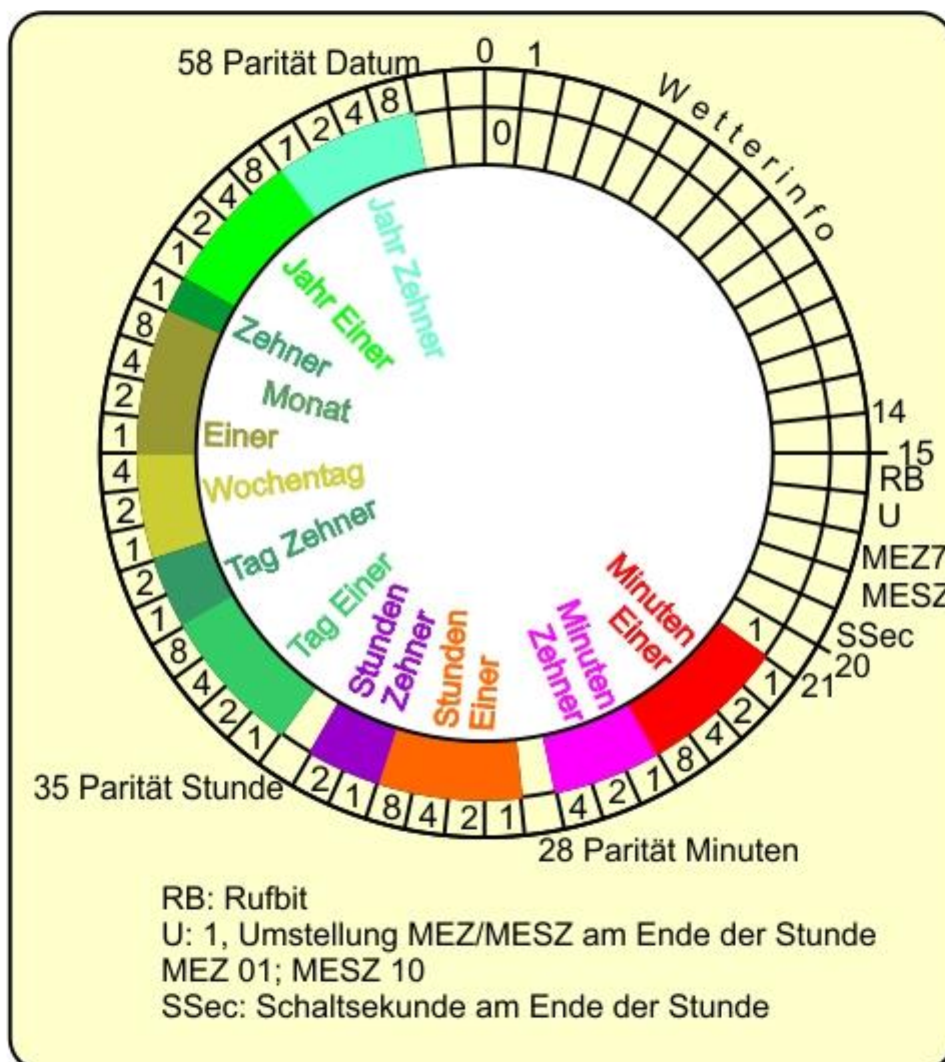

Abbildung 1: DCF77-Codierung

Eine Amplitudenabsenkung des Trägers beim Sender entspricht einem Impuls von Vcc von gleicher Dauer auf unserem Empfängermodul. Am Ausgang des DCF77-Moduls liegt also quasi das negierte Sendersignal an.

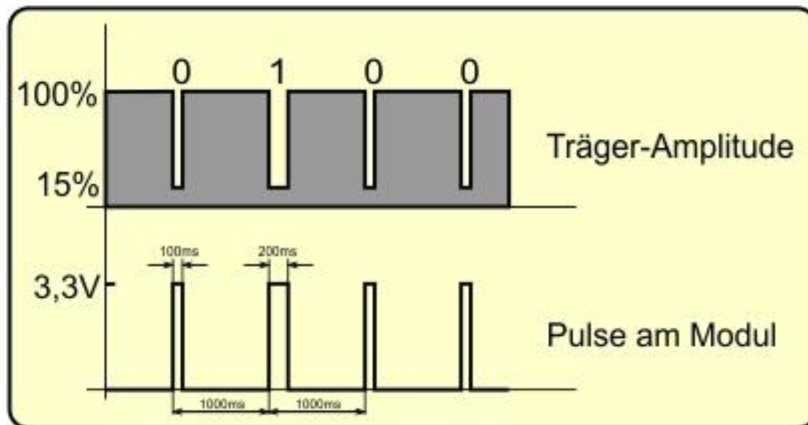

Abbildung 2: Bit-Codierung des DCF77-Signals

Um uns mit dem Sender zu synchronisieren, müssen wir den Anfang eines Zeitframes finden, das Bit 0, Dazu hilft uns die 59. Sekunde, für die keine Absenkung erfolgt und am Modulausgang deshalb auch kein Impuls erzeugt wird.

Nach der fallenden Flanke des Parity-Bits vom Datum kommt daher länger als eine Sekunde keine steigende Flanke. Diesen Moment warten wir ab. Die nächste steigende Flanke läutet dann den Beginn einer neuen Minute ein, und wir beginnen mit der Abtastung der Sekundenimpulse. Die Bitwerte legen wir in einem Bytearray ab. Sind alle Bits eingetrudelt, dann können wir aus dem Array bitweise den BCD-Code entnehmen und zu einem Zeitstempel zusammensetzen. Damit synchronisieren wir unsere RTC.

Hardware

Die Liste mit der bisherigen Hardware aus den vorangegangenen Beiträgen (RC auslesen, RC-IR-Code senden, PS/2-Tastatur am ESP32, eine gute RTC, LED-Display) habe ich nur um das DCF77-Empfangsmodul erweitert, das wir zur Synchronisation unserer Uhr brauchen.

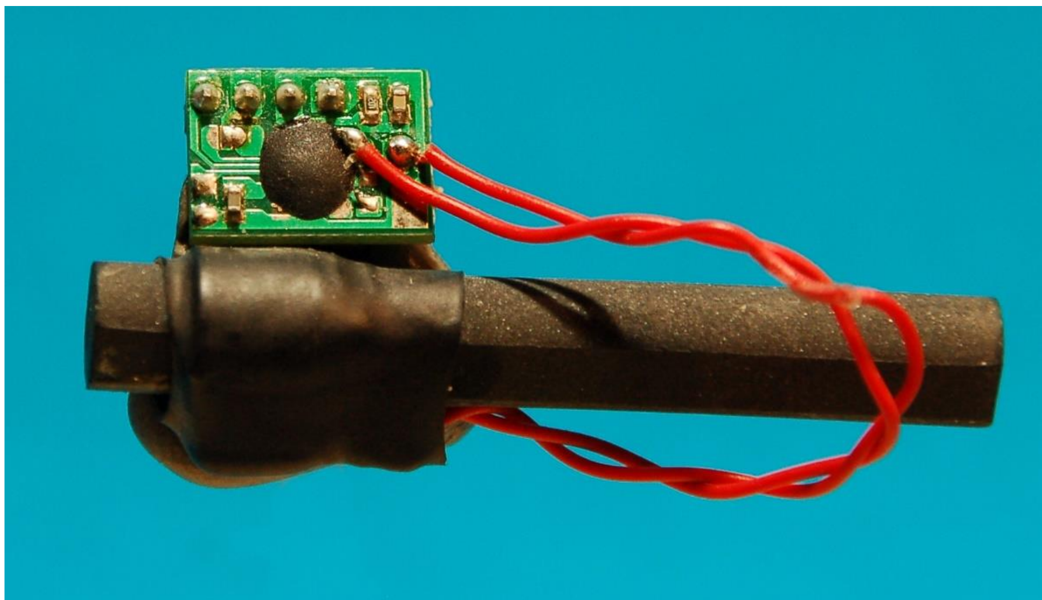
Abbildung 3: DCF77-Modul

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
1	DCF77-Empfänger-Modul
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

Das Modul kann mit Spannungen von 1,2 bis 3,3V betrieben werden und belastet mit weniger als 90µA das 3,3V-Bordnetz des ESP32 nur marginal.

Die Empfangsfrequenz von 77,5 kHz liegt in dem Bereich, in dem Schaltnetzteile arbeiten. Das hat zur Folge, dass zum Beispiel Energiesparlampen durch Interferenzen den Empfang stören können. Leider arbeitet auch unser LED-Display mit einer Multiplexrate von ca. 45kHz. Wenn der Empfang des DCF77-Moduls dadurch gestört wird, sollten wir während der Synchronisation die Anzeige ausschalten. Die Antenne, Ferritstab und Spule, darf auch nicht in die Nähe der Verbindung zwischen Modulausgang und ESP32 kommen. Offenbar führt das durch Rückkopplungen zu wilden Folgen kurzer Impulse.

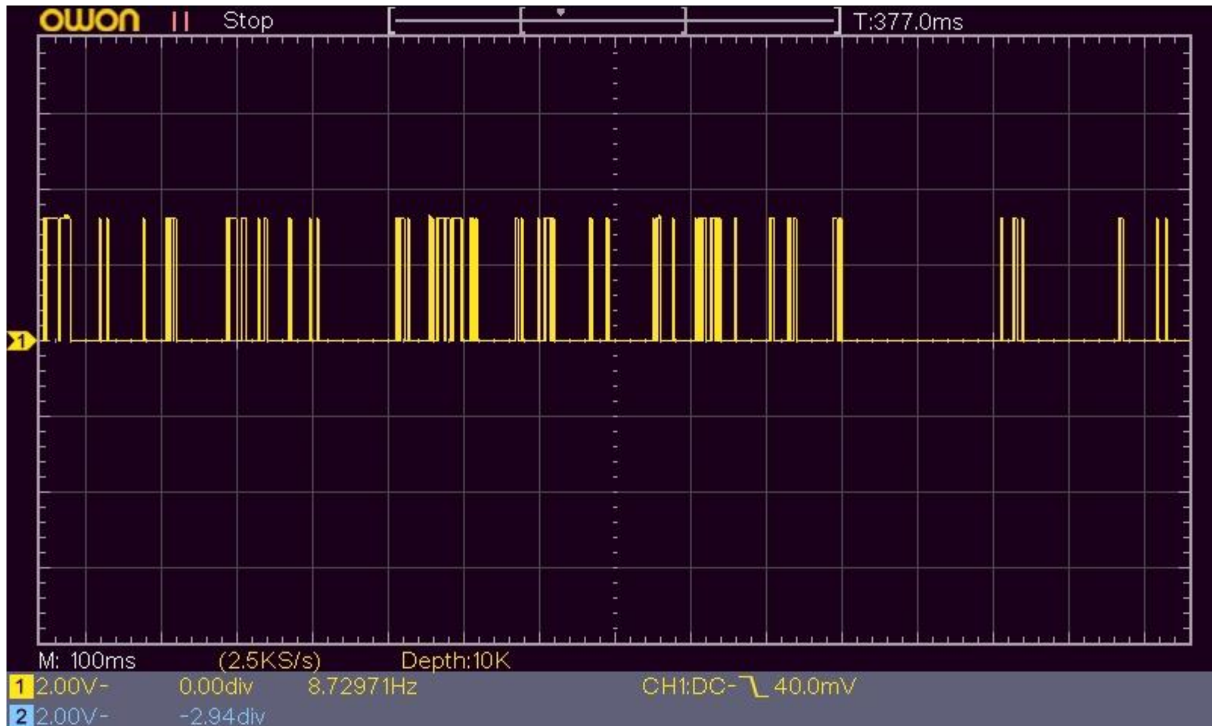

Abbildung 4: Störsignale durch die Kreuzung von Antennenstab und Signalausgeng

Aussehen sollte es so.

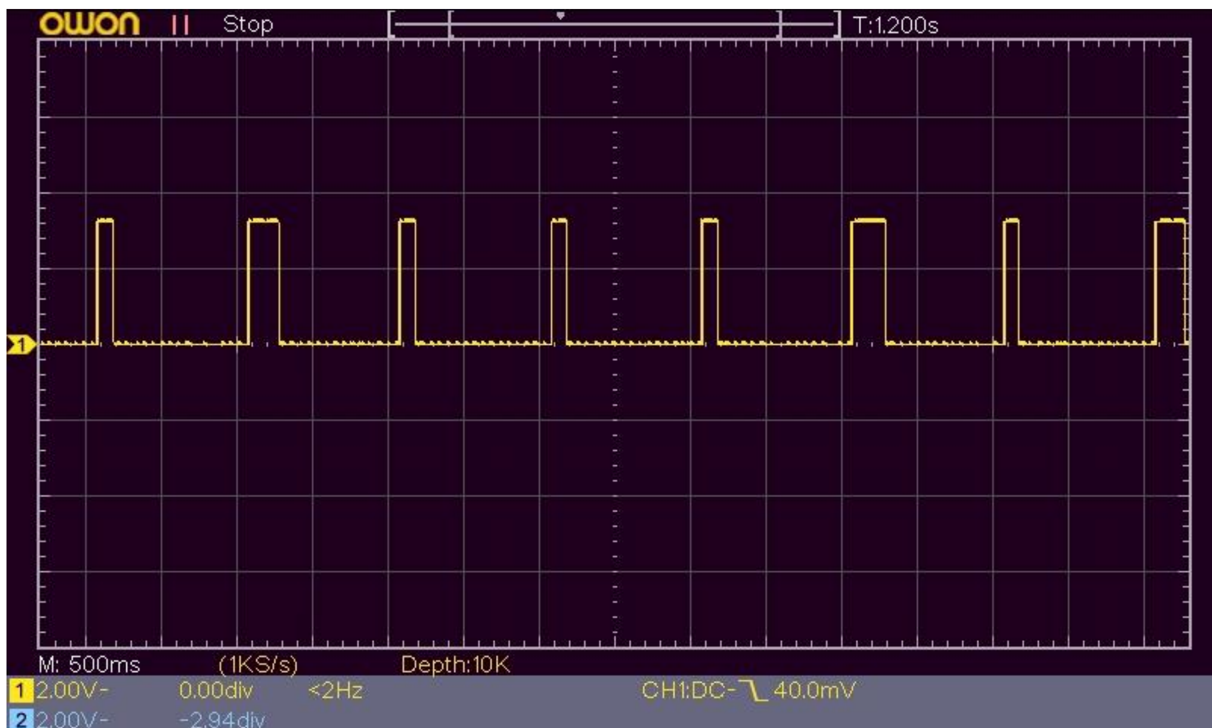

Abbildung 5: Reguläres Impulsbild

In Abbildung 6 sehen Sie die gesamte Schaltung der Funkuhr.

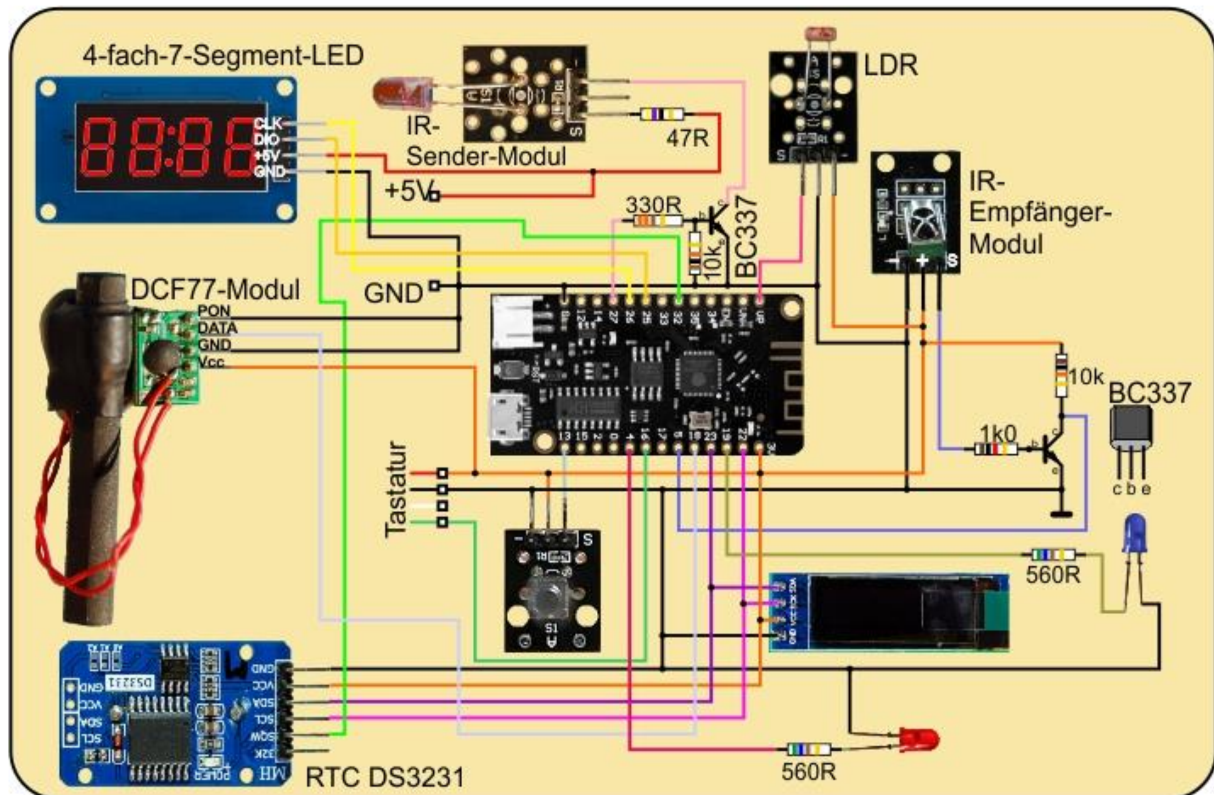

Abbildung 6: Alles zusammen = Funkuhr mit IR-RC-Ambitionen

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
dcf77.py: Treiber für das DCF77-Modul
ir_rx-small.zip: Paket zum IR-Empfangs-Modul

irsend.py: IR-Sende-Modul
timeout.py: Softwaretimer

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.

Wie arbeitet das MicroPython-DCF77-Modul?

Wir beginnen wie üblich mit ein paar Importen, **Pin** von **machine**, einige Methoden aus **time** zur Zeitrechnung und zum Ausruhen und **array** aus **array** für das Bytearray zum Merken der Bitwerte.

```
from machine import Pin, Timer
from time import ticks_us, ticks_diff, sleep, sleep_ms
from array import array
```

Die Klasse DCF77 hat einen recht umfangreichen Konstruktor. **__init__()** nimmt die GPIO-Nummern für den Signaleingang **dcf**, den Anschluss **sec** für die rote Sekundentakt-LED und den Ausgang **wait** für die blaue LED.

```
    def __init__(self,dcf=18,sec=4,wait=19):
        self.seconds =array("B",0 for _ in range(60))
        self.start = 0
        self.ende = 0
        self.delay = 0
        self.counter = 0
        self.triggered = False
        self.flash=False
        self.sec59=False
        self.secLed=Pin(sec,Pin.OUT, value=0)
        self.waitLed=Pin(wait,Pin.OUT,value=0)
        self.dcf=Pin(dcf, Pin.IN)
        self.dcf.irq(handler = None, trigger=Pin.IRQ_RISING)
        print("DCF77 initialisiert")
```

Wir deklarieren einige Instanzattribute und erzeugen die GPIO-Objekte. Dann melden wir schon mal den Interrupt für den Signaleingang vom DCF77-Modul an, weisen jedoch noch keine ISR (Interrupt Service Routine) zu. Schließlich meldet der Konstruktor die Bereitschaft des DCF77-Objekts.

Für Blinksignale deklarieren wir die Methode **blink**(). Sie nimmt die Pulsdauer, eine nachfolgende Pausendauer und das GPIO-Objekt der LED.

```
    def blink(self,puls,pause,led):
        led.on()
        sleep_ms(puls)
        led.off()
        sleep_ms(pause)
```

Die LED wird eingeschaltet, wir warten **puls** Millisekunden, schalten die LED aus und warten **pause** Millisekunden.

Die Länge der Sekundenpulse messen wir mit unserer ISR stopwatch(). Mit dem Parameter **pin** erhalten wir das GPIO-Objekt, das den IRQ ausgelöst hat. Wir fragen

den Pegel ab. Ist er auf 1, dann ging eine steigende Flanke voraus, ein neuer Sekundenauftakt.

```
    def stopwatch(self,pin):
        if pin.value()==1:
            self.dcf.irq(handler = None)
            self.start=ticks_us()
            self.secLed.on()
            sleep_ms(10)
            self.dcf.irq(handler = self.stopwatch,\
                         trigger=Pin.IRQ_FALLING)
        else:
            self.dcf.irq(handler = None)
            self.ende=ticks_us()
            self.delay=ticks_diff(self.ende,self.start)
            sleep_ms(10)
            self.dcf.irq(handler = self.stopwatch,\
            trigger=Pin.IRQ_RISING)
            self.triggered=True
            self.secLed.off()
```

Wir deaktivieren den IRQ, indem wir den Handler auf **None** setzen. Dann merken wir uns den Stand des Microsekundenzählers und machen die rote LED an. Nach einer kurzen Wartezeit schalten wir den IRQ wieder scharf, allerdings wird er jetzt durch eine fallende Flanke getriggert.

Die fallende Flanke erlaubt es uns, über den µs-Zähler die Laufzeit des Impulses zu berechnen. Auch hier deaktivieren zunächst den IRQ. **ticks_diff**() berechnet mit den beiden Flankenzeiten die Pulsbreite und berücksichtigt dabei auch einen eventuellen Zählerüberlauf. Kurze Wartezeit, dann schalten wir wieder auf steigende Flanke um. Wir setzen **triggered** auf True, damit an anderer Stelle die vollendete Zeitmessung erkannt wird. Diese Berechnung hier vorzunehmen ist nicht ratsam, weil eine ISR so kurz wie möglich gehalten werden soll. Die LED wird ausgeschaltet und **stopwatch**() hat ihren Job erledigt.

```
    def wait(self,pin):
        self.start=ticks_us()
        self.triggered=True
        self.flash=True
```

Auch die Methode **wait**() ist eine ISR, die benutzt wird, um auf den Start einer neuen Minute zu warten. Sie speichert die Startzeit einer Flanke, setzt die beiden Attribute **triggered** und **flash** auf True.

Wir müssen auf den Beginn einer neuen Minute warten, bevor wir mit dem Aufzeichnen eines Zeitrahmens beginnen können. Das macht die Methode **waitForStart**(). Wir verfolgen hier eine andere Strategie als bei der Impulslängenmessung, deshalb verwenden wir auch eine andere ISR, nämlich **wait**(). Der normale Sekunden IRQ wird deaktiviert, dafür setzen wir **wait**() als Handler ein und zwar starten wir mit einer fallenden Flanke. Das Flag **flash** für das Blinken der blauen LED setzen wir auf False und gehen in die while-Schleife.

```
    def waitForStart(self):
        self.dcf.irq(handler = None)
        self.dcf.irq(handler = self.wait,
trigger=Pin.IRQ_FALLING)
        print("Warte auf Minutenstart")
        self.flash=False
        while 1:
            if self.triggered and ticks_diff(ticks_us(),\
                  self.start) > 1200000:
                self.dcf.irq(handler = None)
                self.counter = 0
                self.dcf.irq(handler = self.stopwatch, \
                             trigger=Pin.IRQ_RISING)
                break
            if self.triggered and ticks_diff(ticks_us(),\
                  self.start) < 300000 and self.flash:
                self.dcf.irq(handler = None)
                self.flash = False
                sleep_ms(50)
                if self.dcf.value()==0:
                   self.blink(20,1,self.waitLed)
                self.dcf.irq(handler = self.wait,\
                             trigger=Pin.IRQ_FALLING)
```

Wir sind fündig geworden, wenn **wait**() durch eine fallende Flanke getriggert wurde und bis jetzt mehr als 1,2 Sekunden lang keine weitere Flanke aufgetreten ist. Der Minutenzähler wird auf 0 gesetzt und der IRQ-Handler wieder auf **stopwatch**() mit steigender Flanke umgestellt. Mit **break** verlassen wir die while-Schleife und damit auch die Routine. Jetzt ist das System bereit, die kommende, steigende Flanke der nullten Minute zu registrieren.

Sind weniger als 0,3 s vergangen und **flash** ist durch **wait**() auf True gesetzt worden, dann stellen wir **flash** auf False zurück, lassen die LED kurz aufblinken und warten weiter auf die nächste fallende Flanke in ca. 0,65 Sekunden.

So war das ursprünglich gedacht. Leider sagte die LED, durch mehrfaches Flackern, dass da irgendwas nicht stimmt. Mit dem DSO (Digitales Speicher-Oszilloskop) kam ich schnell dahinter, was die Ursache war. Die Sekundenimpulse waren nicht sauber. Sporadisch tauchten bei der steigenden und/oder auch fallenden Flanke kurze Störimpulse auf, wie bei einem Tastenprellen. In Abbildung 7 wird bereits mit der ersten fallenden Flanke des ersten Nadelimpulses ein Blinken ausgelöst.

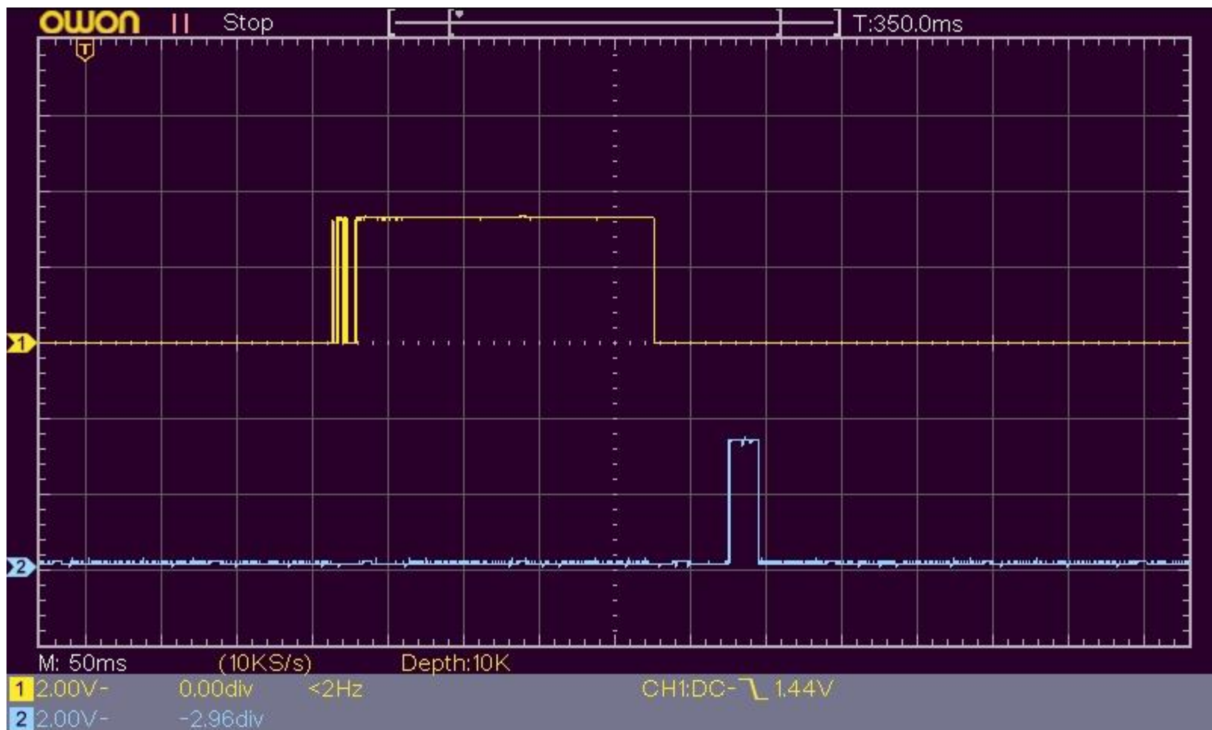

Abbildung 7: Störimpulse am Anfang des Sekunden-Impulses

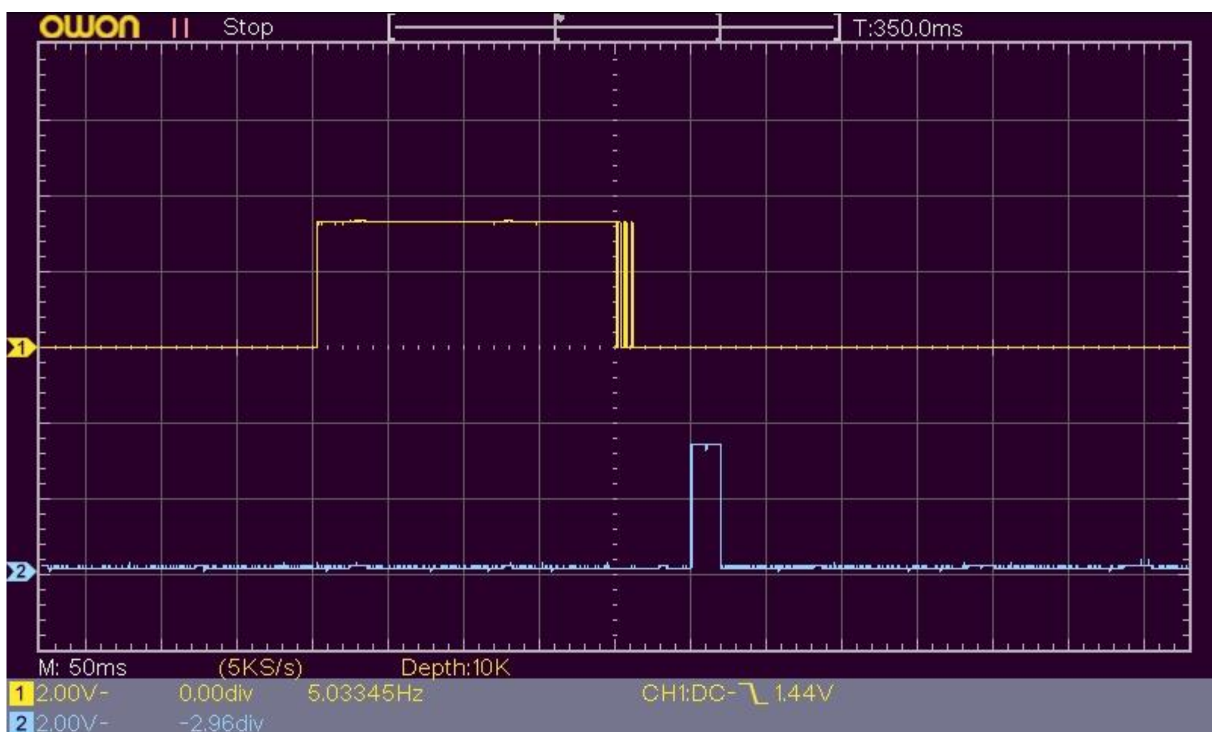

Abbildung 8: Störimpulse nach der fallenden Flanke des Sekundenimpulses

Um Nadelimpulse auszuschalten, habe ich also erst mal den IRQ deaktiviert. Damit wird ein eventueller zweiter Triggerevent weggebügelt. Dann setzen wir **flash** auf False und eine Pause von 50ms sorgt dafür, das Prellen sicher zu überspringen. Ist danach der Pegel auf HIGH, wird das Blinken übergangen. Bei Nadelimpulsen am Ende des Sekundenimpulses wird nur geblinkt, wenn nach den 50 ms der Pegel immer noch LOW ist. Dann wird der IRQ-Handler restauriert, und wir warten auf die nächste fallende Flanke.

Zur Absicherung der Übertragung werden senderseitig die Datenblöcke auf gerade Parität gebracht. Das Paritätsbit wird gesetzt, wenn im Datenblock Minute, Stunde oder Datum eine ungerade Anzahl von 1-en vorkommt. Der Block erreicht damit eine gerade Anzahl von Einsen, even Parity. Das Prüfbit ist 0, wenn die Anzahl von 1-en im Datenteil gerade ist.

```
    def checkParity(self):
        minuten=0
        for i in range(21,29):
            minuten += self.seconds[i]
        minuten %= 2
        stunden=0
        for i in range(29,36):
            stunden += self.seconds[i]
        stunden %= 2
        datum = 0
        for i in range(36,59):
            datum += self.seconds[i]
            datum %= 2
        return (minuten,stunden,datum)
```

Die Methode **checkParity**() zählt die 1-en im Block inclusive Paritätsbit. Eine gerade Anzahl ergibt beim Zweier-Teilungsrest eine 0, eine ungerade Anzahl eine 1, was einen Fehler erkennen lässt. Das Ergebnis wird in Form eines Tupels zurückgegeben und sollte (0,0,0) sein.

Bei der Berechnung von Datum und Uhrzeit müssen die BCD-Bits aus dem Array **seconds** herausgefischt, zu den Dezimalziffern zusammengefügt und daraus die Dezimalzahl berechnet werden. Das macht die Funktion **bcd2dec**(), die lokal in **calcDateTime**() deklariert ist.

Übergeben werden die Startpositionen (c und cc) der Einer- und Zehner-Bits sowie die Anzahl an BCD-Bits (n und m). Die Zifferwerte x und xx werden zu Beginn auf 0 gesetzt. In der for-Schleife wird nacheinander ein Bitwert mit dem binären Stellenwert 2 hoch i multipliziert und zum bisherigen Ergebnis addiert. Die Zehnerziffer mal 10 plus die Einerziffer ergibt die Dezimalzahl.

```
    def calcDateTime(self):
        def bcd2dec(c,n,cc,m):
            x,xx=0,0
            for i in range(n):
                x += self.seconds[c+i]*(2**i)
            for i in range(m):
                xx += self.seconds[cc+i]*(2**i)
            x=x+xx*10
            return x

        m=bcd2dec(21,4,25,3)
        h=bcd2dec(29,4,33,2)
        d=bcd2dec(36,4,40,2)
        dow=0
```

```
        for i in range(3):
            dow += self.seconds[42+i]*(2**i)
        M=bcd2dec(45,4,49,1)
        y=bcd2dec(50,4,54,4)
        return y,M,d,dow,h,m
```

In der Folge muss ich jetzt nur noch die entsprechenden Parameter übergeben, die aus der Abbildung 1 entnommen werden können. Einzige Ausnahme ist der Wochentag, der als einstelliger Wert in einer eigenen Schleife nach derselben Methode berechnet wird. Schließlich erfolgt die Rückgabe des Tupels (Jahr, Monat, Tag, Wochentag, Stunde, Minute).

Die Methode **synchronize**() führt alle bisherigen Programmpartikel zusammen und gibt einen Timestamp der Normzeit der PTB zurück.

```
    def synchronize(self):
        self.waitForStart()
        self.dcf.irq(handler = self.stopwatch, \
                     trigger=Pin.IRQ_RISING)

        while 1:
            if self.triggered and self.delay > 20000:
                self.triggered=False
                code=(((self.delay//1000) + 20 ) // 100) - 1
                print(self.counter,code)
                self.seconds[self.counter]=code
                self.counter =(self.counter +1) % 59
                if self.counter == 0:
                    sleep(0.95)
                    self.sec59=True
        #         if counter == 0: break
            if self.counter == 0 and self.dcf.value() == 0 \
                                    and self.sec59:
                self.sec59=False
                parity = self.checkParity()
                if parity == (0,0,0):
                    y,M,d,dow,h,m=self.calcDateTime()
                    dt=(y+2000,M,d,dow,h,m,0,0)
                    print(dt)
                    # rtc-Zählung startet ,mit 0 am Montag,
Sonntag=6
                    # rtc berechnet den Wochentag nach dem
Datum
                    # dcf startet mit 1 am Montag, Sonntag=7
                    self.dcf.irq(handler = None)
                    return dt
                else:
                    print("Parity:",parity)
                    self.dcf.irq(handler = None)
                    sleep(2)
                    self.waitForStart()
```

Zunächst warten wir auf den Start einer Minute und aktivieren dann sofort den DCF-IRQ mit steigender Flanke. Die löst dann den Aufruf der ISR **stopwatch**() aus.

Wir gehen in die Endlosschleife. **triggered** und **sec59** sind erst einmal durch den Konstruktoraufruf mit False vorbelegt, **counter** mit 0. Irgendwann setzt eine steigende Flanke die Stopuhr in Gang, das heißt, wir merken uns die Startzeit in **start**. Die folgende fallende Flanke setzt **triggered** auf True und übergibt die Dauer des Pulses als Zeitdifferenz an **delay**.

Wenn ein Puls getriggert wurde und dieser länger als 20ms war, wird er als Sekundenimpuls gewertet. Kürzere Pulse werden als Störimpuls ausgesondert.
Wir setzen **triggered** auf False.

Die Sekundenimpulse weichen in der Regel von der nominalen Länge, 100 ms beziehungsweise 200 ms, nach oben und unten leicht ab. Die Ausreißer nach unten werden bei der Berechnung des Bitwerts durch die Addition von 20 ms berücksichtigt. Aus den Microsekunden in **delay** machen wir zuerst Millisekunden, addieren dann 20 und führen danach eine Ganzzahldivision mit 100 durch. Es ergibt sich ein Wert von 1 oder 2. Subtrahieren wir 1, so erhalten wir den Bitwert 0 oder 1. Die Impulsdauer muss für eine sichere Erkennung somit mindestens 80 ms beziehungsweise 180 ms sein.

Der Sekundenzähler **counter** und der Bitwert in **code** werden in REPL ausgegeben. Der Bitwert landet im Array **seconds** mit **counter** als Index. Der Zähler wird modulo 59 erhöht, das heißt beim Erreichen der 59. Sekunde wird er auf 0 gesetzt.

Ist dieser Fall eingetreten, verschlafen wir den Rest der Sekunde und setzen **sec59** wieder auf False.

Wenn der Zähler den Wert 0 hat, der Pegel des dcf-Pins auf LOW liegt und die 59. Sekunde erreicht ist, haben wir alle Bits zur Berechnung von Uhrzeit und Datum. **sec59** geht auf False und wir führen den Paritycheck durch. Wenn dieser positiv ausfällt, lassen wir die Zeit- und Datumswerte berechnen. Die Jahreszahl bringen wir auf vier Stellen und fügen alles zu einem Tupel zusammen. Der Timestamp wird ausgegeben. Nachdem der IRQ gezwickt ist erfolgt die Rückgabe.

Natürlich kann ein Übertragungsfehler auftreten, der anhand der Party-Bits erkannt wird. In welchem Bereich das passiert ist sieht man an dem ausgegebenen Tupel. Auch hier schalten wir den DCF-IRQ erst mal ab, warten 2 Sekunden ab und stoßen dann eine neue Suche nach dem Minutenbeginn an.

Wenn man die auskommentierten Zeilen reanimiert (markieren und Alt + 4), dann kann man mit dem Start der Datei **dcf77.py** im Editorfenster ein DCF77-Objekt erzeugen und damit gleich die RTC synchronisieren.

```
# if __name__ == "__main__":
#     from machine import Pin,SoftI2C
#     from ds3231 import DS3231
#     from machine import Pin,SoftI2C
#     i2c=SoftI2C(scl=Pin(22),sda=Pin(23),freq=100000)
#     rtc=DS3231(i2c)
#     dc=DCF77()
#     dt=dc.synchronize()
#     rtc.DateTime(dt)
```

Damit sind wir am Puls der amtlichen Zeit in Deutschland und haben außerdem die verschiedenen Zutaten zusammengetragen die einen Wecker ausmachen. Genau diesen bauen wir in der nächsten Folge zusammen. Gemeinsam mit der Handy-App, die in dem darauffolgenden Post gebastelt wird, haben wir dann eine funksynchronisierte Uhr mit Wecker, der via IR-Sendediode ein anderes Gerät mittels RC5-Code steuern kann. Die Codes einer RC-Steuerung kann unsere eierlegende Wollmilchsau auslesen, speichern und bei Bedarf abrufen. Über das Handy lassen sich so an den Accesspoint des ESP32 Befehle übermitteln, welche dieser als IR-RC5-Code weitergibt.

Bleiben Sie dran!